Improved Parallel Integer Sorting without Concurrent Writing

Susanne Albers and Torben Hagerup

MPI–I–94–137 August 1994

Improved Parallel Integer Sorting without Concurrent Writing

Susanne Albers and Torben Hagerup

MPI–I–94–137 August 1994

Improved Parallel Integer Sorting without Concurrent Writing[*]

Susanne Albers[†‡] Torben Hagerup[†]

Abstract

We show that n integers in the range $1\,..\,n$ can be stably sorted on an EREW PRAM using $O(t)$ time and $O(n(\sqrt{\log n \log\log n} + (\log n)^2/t))$ operations, for arbitrary given $t \geq \log n \log\log n$, and on a CREW PRAM using $O(t)$ time and $O(n(\sqrt{\log n} + \log n/2^{t/\log n}))$ operations, for arbitrary given $t \geq \log n$. In addition, we are able to sort n arbitrary integers on a randomized CREW PRAM within the same resource bounds with high probability. In each case our algorithm is a factor of almost $\Theta(\sqrt{\log n})$ closer to optimality than all previous algorithms for the stated problem in the stated model, and our third result matches the operation count of the best known sequential algorithm. We also show that n integers in the range $1\,..\,m$ can be sorted in $O((\log n)^2)$ time with $O(n)$ operations on an EREW PRAM using a nonstandard word length of $O(\log n \log\log n \log m)$ bits, thereby greatly improving the upper bound on the word length necessary to sort integers with a linear time-processor product, even sequentially. Our algorithms were inspired by, and in one case directly use, the fusion trees of Fredman and Willard.

1 Introduction

A parallel algorithm is judged primarily by its *speed* and its *efficiency*. Concerning speed, a widely accepted criterion is that it is desirable for a parallel algorithm to have a running time that is polylogarithmic in the size of the input. The efficiency of a parallel algorithm is evaluated by comparing its *time-processor* product, i.e., the total number of *operations* executed, with the running time of an optimal sequential algorithm for the problem under consideration, the parallel algorithm having *optimal speedup* or simply being *optimal* if the two agree to within a constant factor. A point of view often put forward and forcefully expressed in (Kruskal *et al.* 1990b) is that the efficiency of a parallel algorithm, at least given present-day technological constraints, is far more important than its raw speed, the reason being that in all probability, the algorithm must be slowed down to meet a smaller number of available processors anyway.

[*] *A preliminary version of this paper was presented at the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'92) in Orlando, Florida in January 1992.*

[†] *Max-Planck-Institut für Informatik, D–66123 Saarbrücken, Germany. Supported in part by the Deutsche Forschungsgemeinschaft, SFB 124, TP B2, VLSI Entwurfsmethoden und Parallelität, and in part by the ESPRIT Basic Research Actions Program of the EU under contracts No. 3075 and 7141 (projects ALCOM and ALCOM II).*

[‡] *Part of this work was done while the author was a student at the Graduiertenkolleg Informatik, Universität des Saarlandes, D–66041 Saarbrücken, Germany, and supported by a graduate fellowship of the Deutsche Forschungsgemeinschaft.*

The problem of integer sorting on a PRAM has been studied intensively. While recent research has concentrated on algorithms for the CRCW PRAM (Rajasekaran and Reif, 1989; Rajasekaran and Sen, 1992; Bhatt *et al.*, 1991; Matias and Vishkin, 1991a, 1991b; Raman, 1990, 1991a, 1991b; Hagerup, 1991; Gil *et al.*, 1991; Hagerup and Raman, 1992, 1993; Bast and Hagerup, 1993; Goodrich *et al.*, 1993, 1994), in this paper we are interested in getting the most out of the weaker EREW and CREW PRAM models. Consider the problem of sorting n integers in the range $1\,..\,m$. In view of sequential radix sorting, which works in linear time if $m = n^{O(1)}$, a parallel algorithm for this most interesting range of m is optimal only if its time-processor product is $O(n)$. Kruskal *et al.* (1990a) showed that for $m \geq n$ and $1 \leq p \leq n/2$, the problem can be solved using p processors in time $O\left(\frac{n}{p} \cdot \frac{\log m}{\log(n/p)}\right)$, i.e., with a time-processor product of $O\left(\frac{n \log m}{\log(n/p)}\right)$. The space requirements of the algorithm are $\Theta(pn^{\epsilon} + n)$, for arbitrary fixed $\epsilon > 0$, which makes the algorithm impractical for values of p close to n. Algorithms that are not afflicted by this problem were described by Cole and Vishkin (1986, remark following Theorem 2.3) and by Wagner and Han (1986). For $n/m \leq p \leq n/\log n$, they use $O(n)$ space and also sort in $O\left(\frac{n}{p} \cdot \frac{\log m}{\log(n/p)}\right)$ time with p processors. All three algorithms are optimal if $m = (n/p)^{O(1)}$. However, let us now focus on the probably most interesting case of $m \geq n$ combined with running times of $(\log n)^{O(1)}$. Since the running time is at least $\Theta(n/p)$, it is easy to see that the time-processor product of the above algorithms in this case is bounded only by $O(n \log m/\log\log n)$, i.e., the integer sorting algorithms are more efficient than algorithms for general (comparison-based) sorting, which can be done in $O(\log n)$ time using $O(n \log n)$ operations (Ajtai *et al.*, 1983; Cole, 1988), by a factor of at most $\Theta(\log\log n)$, to be compared with the potential maximum gain of $\Theta(\log n)$. This is true even of a more recent EREW PRAM algorithm by Rajasekaran and Sen (1992) that stably sorts n integers in the range $1\,..\,n$ in $O(\log n \log\log n)$ time using $O(n \log n/\log\log n)$ operations and $O(n)$ space.

We describe an EREW PRAM algorithm for stably sorting n integers in the range $1\,..\,n$ that exhibits a tradeoff between speed and efficiency. The minimum running time is $\Theta(\log n \log\log n)$, and for this running time the algorithm essentially coincides with that of Rajasekaran and Sen (1992). Allowing more time, however, we can sort with fewer operations, down to a minimum of $\Theta(n\sqrt{\log n \log\log n})$, reached for a running time of $\Theta((\log n)^{3/2}/\sqrt{\log\log n})$. In general, for any given $t \geq \log n \log\log n$, the algorithm can sort in $O(t)$ time using $O(n(\sqrt{\log n \log\log n} + (\log n)^2/t))$ operations. Run at the slowest point of its tradeoff curve, the algorithm is more efficient that the algorithms discussed above by a factor of $\Theta(\sqrt{\log n}/(\log\log n)^{3/2})$.

On the CREW PRAM, we obtain a much steeper tradeoff: For all $t \geq \log n$, our algorithm sorts n integers in the range $1\,..\,n$ in $O(t)$ time using $O(n(\sqrt{\log n} + \log n/2^{t/\log n}))$ operations; the minimum number of operations of $\Theta(n\sqrt{\log n})$ is reached for a running time of $\Theta(\log n \log\log n)$. We also consider the problem of sorting integers of arbitrary size on the CREW PRAM and describe a reduction of this problem to that of sorting n integers in the range $1\,..\,n$. The reduction is randomized and uses $O(\log n)$ time and $O(n\sqrt{\log n})$ operations with high probability. It is based on the fusion trees of Fredman and Willard (1990) and was discovered independently by Raman (1991a).

The algorithms above all abide by the standard convention that the *word length* available to sort n integers in the range $1..m$ is $\Theta(\log(n+m))$ bits, i.e., that unit-time operations on integers of size $(n+m)^{O(1)}$ are provided, but that all integers manipulated must be of size $(n+m)^{O(1)}$. A number of papers have explored the implications of allowing a larger word length. Paul and Simon (1980) and Kirkpatrick and Reisch (1984) demonstrated that with no restrictions on the word length at all, arbitrary integers can be sorted in linear sequential time. Hagerup and Shen (1990) showed that in fact a word length of about $O(n \log n \log m)$ bits suffices to sort n integers in the range $1..m$ in $O(n)$ sequential time or in $O(\log n)$ time on an EREW PRAM with $O(n/\log n)$ processors. The practical value of such results is doubtful because of the unrealistic assumptions: Hardly any computer has a word length comparable to typical input sizes. We show that n integers in the range $1..m$ can be sorted with a linear time-processor product in $O((\log n)^2)$ time on an EREW PRAM with a word length of $O(\log n \log\log n \log m)$ bits. At the price of a moderate increase in the running time, this greatly improves the known upper bound on the word length necessary to sort integers with a linear time-processor product, even sequentially. Another, perhaps more illuminating, perspective is obtained by noting that providing a PRAM with a large word length amounts to adding more parallelism of a more restrictive kind. A single register of many bits is akin to a whole SIMD machine, but without the important ability to let individual processors not participate in the current step. As suggested by this view, using a word length that allows k integers to be stored in each word potentially could decrease the running time by a factor of up to $\Theta(k)$. What our algorithm actually achieves is a reduction in the running time by a factor of $\Theta(k/\log k)$ (curiously, it does so by calling a sequential version of an originally parallel algorithm as a subroutine). This can be interpreted as saying that if one happens to sort integers of which several will fit in one word, then advantage can be taken of this fact. Our result also offers evidence that algorithms using a nonstandard word length should not hastily be discarded as unfeasible and beyond practical relevance.

2 Preliminaries

A *PRAM* is a synchronous parallel machine consisting of processors numbered $1, 2, \ldots$ and a global memory accessible to all processors. An *EREW PRAM* disallows concurrent access to a memory cell by more than one processor, a *CREW PRAM* allows concurrent reading, but not concurrent writing, and a *CRCW PRAM* allows both concurrent reading and concurrent writing. We assume an instruction set that includes addition, subtraction, comparison, unrestricted shift as well as the bitwise Boolean operations AND and OR. The shift operation takes two integer operands x and i and produces $x \uparrow i = \lfloor x \cdot 2^i \rfloor$.

Our algorithms frequently need to compute quantities such as $\log m\sqrt{\lambda/\log\lambda}$, where $m, \lambda \geq 2$ are given integers. Since we have not assumed machine instructions for multiplication and division, let alone extraction of logarithms and square roots, it is not obvious how to carry out the computation. In general, however, it suffices for our purposes to compute the quantities of interest approximately, namely up to a constant factor, which can be done using negligible resources: Shift operations provide exponentiation, logarithms can be extracted by exponentiating all candidate values in parallel and picking the right one, and approximate multiplication, division and extraction of square roots reduce to extraction of logarithms, followed by addition, subtraction or halving (i.e., right shift by one bit), followed by exponentiation. On some occasions, we need more accurate calculations, but the numbers involved are small, and we can implement multiplication by repeated addition, etc. The details will be left to the reader.

Two basic operations that we shall need repeatedly are prefix summation and segmented broadcasting. The *prefix summation* problem of size n takes as input an associative operation $\circ$ over a semigroup S as well as n elements $x_1, \ldots, x_n$ of S, and the task is to compute $x_1, x_1 \circ x_2, \ldots, x_1 \circ x_2 \circ \cdots \circ x_n$. The *segmented broadcasting* problem of size n is, given an array $A[1..n]$ and a bit sequence $b_1, \ldots, b_n$ with $b_1 = 1$, to store the value of $A[\max\{j : 1 \le j \le i \text{ and } b_j = 1\}]$ in $A[i]$, for $i = 1, \ldots, n$. It is well-known that whenever the associative operation can be applied in constant time by a single processor, prefix summation problems of size n can be solved on an EREW PRAM using $O(\log n)$ time and $O(n)$ operations (see, e.g., (JáJá, 1992)). Segmented broadcasting problems of size n can be solved within the same resource bounds by casting them as prefix summation problems; the details can be found, e.g., in (Hagerup and Rüb, 1989).

It can be shown that if an algorithm consists of r parts such that the ith part can be executed in time t_i using q_i operations, for $i = 1, \ldots, r$, then the whole algorithm can be executed in time $O(t_1 + \cdots + t_r)$ using $O(q_1 + \cdots + q_r)$ operations, i.e., time and number of operations are simultaneously additive. We make extensive and implicit use of this observation.

We distinguish between *sorting* and *ranking*. To *sort* a sequence of records, each with a key drawn from a totally ordered domain, is to rearrange the records such that in the resulting sequence the keys occur in nondecreasing order. The sorting is *stable* if nonidentical records with the same key occur in the same order in the output sequence as in the input sequence, i.e., if there are no unnecessary interchanges. To (stably) *rank* a sequence $R_1, \ldots, R_n$ of records as above is to compute a permutation $i_1, \ldots, i_n$ of $1, \ldots, n$ such that $R_{i_1}, \ldots, R_{i_n}$ is a possible result of (stably) sorting $R_1, \ldots, R_n$.

For records that consist of just a key with no associated information, the stability requirement adds nothing to the sorting problem — it is vacuously satisfied. In such cases we follow the tradition of using the term "stable sorting" in the sense of stable ranking. Note that any algorithm for (stable) ranking on a PRAM implies an algorithm for (stable) sorting with the same resource bounds, up to a constant factor. On the other hand, an algorithm for stable ranking can be derived from any sorting algorithm that can be modified as follows: Let the sequence of records to be sorted be $R_1, \ldots, R_n$, and let x_i be the key of R_i, for $i = 1, \ldots, n$. The modified algorithm uses as the key of R_i the pair (x_i, i), for $i = 1, \ldots, n$, and orders these pairs lexicographically (i.e., $(x_i, i) < (x_j, j) \Leftrightarrow x_i < x_j$ or $(x_i = x_j$ and $i < j)$). We shall refer to this process as *stabilizing* the original algorithm.

Given an element x of a totally ordered domain U and a finite sequence L of elements of U, the *rank* of x in L is the number of elements in L no larger than x.

Our basic sorting algorithm is nonconservative, following the terminology of Kirkpatrick and Reisch (1984), i.e., the word length allowed for sorting n integers in the range $1..m$ is not limited to $O(\log(n+m))$ bits. More precisely, we will use a word length of $\Theta(kl + \log n)$ bits, where k is a power of 2 with $2 \le k \le n$ and l is an integer with $l \ge \lceil \log(m+k) \rceil + 2$. This word length enables us to store more than one number in a word, which is essential for our technique.

Throughout the paper we number the bit positions of words and integers from right to left, with the least significant bit occupying position 0. We assume a word length of at least $2kl$ bits and partition the rightmost $2kl$ bits into $2k$ *fields* of l bits each. The fields are numbered from right to left, starting at 0, and the number of a field is also called its *address*. The most significant bit of each field, called the *test bit*, is usually set to 0. The remaining bits of the field can represent a nonnegative integer coded in binary and called an *entry*. A word normally stores k entries $x_0, \ldots, x_{k-1}$ in its rightmost k fields and thus has the following structure:

$$\underbrace{\boxed{0\,0\cdots 0}}_{\text{field } 2k-1}\ \cdots\ \underbrace{\boxed{0\,0\cdots 0}}_{\text{field } k}\underbrace{\boxed{0\ \ x_{k-1}}}_{\text{field } k-1}\ \cdots\ \underbrace{\boxed{0\ \ x_0}}_{\text{field } 0}$$

The fields $k, \ldots, 2k-1$ serve as temporary storage. Two fields in distinct words *correspond* if they have the same address, and two entries correspond if they are stored in corresponding fields. For $h \geq 1$, a word that contains the test bit b_i and the entry x_i in its field number i, for $i = 0, \ldots, h-1$, and whose remaining bits are all 0 will be denoted $[b_{h-1} : x_{h-1}, \ldots, b_0 : x_0]$. If $b_0 = \cdots = b_{h-1} = 0$, we simplify this to $[x_{h-1}, \ldots, x_0]$. By a *sorted word* we mean a word of the form $[x_{k-1}, \ldots, x_0]$, where $x_0 \leq x_1 \leq \cdots \leq x_{k-1}$. A sequence $x_1, \ldots, x_n$ of n integers is said to be given in the *word representation* (with parameters k and l) if it is given in the form of the words $[x_k, \ldots, x_1], [x_{2k}, \ldots, x_{k+1}], \ldots, [x_n, \ldots, x_{(\lceil n/k \rceil - 1)k+1}]$.

The remainder of the paper is structured as follows: We first study algorithms for the EREW PRAM. Section 3 tackles the humble but nontrivial task of merging (the sequences represented by) two sorted words. Based on this, Section 4 develops a nonconservative parallel merge sorting algorithm, from which the conservative algorithm that represents our main result for the EREW PRAM is derived in Section 5. Section 6 describes our algorithms for the CREW PRAM.

3 Merging two words

In this section we describe how a single processor can merge two sorted words in $O(\log k)$ time. We use the bitonic sorting algorithm of Batcher (1968) and need only describe how bitonic sorting can be implemented to work on sorted words.

A sequence of integers is *bitonic* if it is the concatenation of a nondecreasing sequence and a nonincreasing sequence, or if it can be obtained from such a sequence via a cyclic shift. E.g., the sequence $5, 7, 6, 3, 1, 2, 4$ is bitonic, but $2, 5, 3, 1, 4, 6$ is not. Batcher's bitonic sorting algorithm takes as input a bitonic sequence $x_0, \ldots, x_{h-1}$, where h is a power of 2. The algorithm simply returns the one-element input sequence if $h = 1$, and otherwise executes the following steps:

(1) For $i = 0, \ldots, h/2 - 1$, let $m_i = \min\{x_i, x_{i+h/2}\}$ and $M_i = \max\{x_i, x_{i+h/2}\}$.

(2) Recursively sort $m_0, \ldots, m_{h/2-1}$, the sequence of minima, and $M_0, \ldots, M_{h/2-1}$, the sequence of maxima, and return the sequence consisting of the sorted sequence of minima followed by the sorted sequence of maxima.

Although we shall not here demonstrate the correctness of the algorithm, we note that all that is required is a proof that $m_0, \ldots, m_{h/2-1}$ and $M_0, \ldots, M_{h/2-1}$ are bitonic (so that the recursive application is permissible), and that $m_i \le M_j$ for all $i, j \in \{0, \ldots, h/2 - 1\}$ (so that the concatenation in Step (2) indeed produces a sorted sequence).

Our implementation works in $\log k + 1$ *stages* numbered $\log k, \log k - 1, \ldots, 0$, where a stage corresponds to a recursive level in the above description. In the beginning of Stage t, for $t = \log k, \ldots, 0$, there is a single word Z containing $2^{\log k - t}$ bitonic sequences of length 2^{t+1} each, each of which is stored in 2^{t+1} consecutive fields of Z. Furthermore, if z is an element of a sequence stored to the left of a different sequence containing an element z', then $z \ge z'$. Stage t, for $t = \log k, \ldots, 0$, carries out the above algorithm on each of the $2^{\log k - t}$ bitonic sequences, i.e., each sequence of 2^{t+1} elements is split into a sequence of 2^t minima and a sequence of 2^t maxima, and these are stored next to each other in the 2^{t+1} fields previously occupied by their parent sequence, the sequence of maxima to the left of the sequence of minima.

Because of the close connection to the recursive description, the correctness of this implementation is obvious. In order to use the algorithm for merging, as opposed to bitonic sorting, we introduce a preprocessing step executed before the first phase. Given words $X = [x_{k-1}, \ldots, x_0]$ and $Y = [y_{k-1}, \ldots, y_0]$, the preprocessing step produces the single word $Z = [y_0, \ldots, y_{k-1}, x_{k-1}, \ldots, x_0]$. The important thing to note is that if $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ are sorted, then $x_0, \ldots, x_{k-1}, y_{k-1}, \ldots, y_0$ is bitonic.

We now describe in full detail the single steps of the merging algorithm. We repeatedly use the two constants

$$K_1 = \underbrace{[1:0, 1:0, \ldots, 1:0]}_{2k \text{ fields}}$$

and

$$K_2 = [2k-1, 2k-2, \ldots, 2, 1, 0].$$

It is easy to compute K_1 and K_2 in $O(\log k)$ time. This is accomplished by the program fragment below.

```
$K_1 := 1 \uparrow (l - 1)$;
**for** $t := 0$ **to** $\log k$ **do**
  (* $K_1 = \underbrace{[1:0, 1:0, \ldots, 1:0]}_{2^t \text{ fields}}$ *)
  $K_1 := K_1$ OR $(K_1 \uparrow (2^t \cdot l))$;

$K_2 := (K_1 \uparrow (-l + 1)) - 1$;
**for** $t := 0$ **to** $\log k$ **do**
  (* $K_2 = \underbrace{[2^t, \ldots, 2^t, 2^t, 2^t - 1, 2^t - 2, \ldots, 2, 1, 0]}_{2k \text{ fields}}$ *)
  $K_2 := K_2 + ((K_2 \uparrow (2^t \cdot l))$ AND $\mathit{CopyTestBit}(K_1))$;
```

The function *CopyTestBit*, used in the last line above to remove spurious bits by truncating the shifted copy of K_2 after field number $2k - 1$, takes as argument a word with the rightmost $l - 1$ bits in each field and all bits to the left of field number $2k - 1$ set to 0 and returns the word obtained from the argument word by copying the value of each test bit to all other bits within the same field and subsequently setting all test bits to 0. It can be defined as follows:

$$\mathit{CopyTestBit}(A) \equiv A - (A \uparrow (-l+1)).$$

We next describe in detail the preprocessing step of our merging algorithm. The central problem is, given a word $Y = [y_{k-1}, \ldots, y_0]$, to produce the word

$$[y_0, \ldots, y_{k-1}, \underbrace{0, 0, \ldots, 0}_{k \text{ fields}}].$$

In other words, the task is to reverse a sequence of $2k$ numbers. First observe that if we identify addresses of fields with their binary representation as strings of $\log k + 1$ bits and denote by $\bar{a}$ the address obtained from the address a by complementing each of its $\log k + 1$ bits, then the entry in the field with address a, for $a = 0, \ldots, 2k-1$, is to be moved to address $\bar{a}$. To see this, note that the entry under consideration is to be moved to address $a' = 2k - 1 - a$. Since k is a power of 2, $a' = \bar{a}$. By this observation, the preprocessing can be implemented by stages numbered $0, \ldots, \log k$, executed in any order, where Stage t, for $t = 0, \ldots, \log k$, swaps each pair of entries whose addresses differ precisely in bit number t. In order to implement Stage t, we use a mask M to separate those entries whose addresses have a 1 in bit number t from the remaining entries, shifting the former right by 2^t fields and shifting the latter left by 2^t fields. The complete preprocessing can be programmed as follows:

```
for $t := 0$ to $\log k$ do
  begin (* Complement bit $t$ *)
    $M := \mathit{CopyTestBit}((K_2 \uparrow (l-1-t))$ AND $K_1)$;
    (* Mask away fields with 0 in position $t$ of their address *)
    $Y := ((Y$ AND $M) \uparrow (-2^t \cdot l))$ OR
          $((Y - (Y$ AND $M)) \uparrow (2^t \cdot l))$;
  end;
$Z := X$ OR $Y$;
```

Let us now turn to the bitonic sorting itself, one stage of which is illustrated in Fig. 1. In Stage t of the bitonic sorting, for $t = \log k, \ldots, 0$, we first extract the entries in those fields whose addresses have a 1 in bit position t and compute a word A containing these entries, moved right by 2^t fields, and a word B containing the remaining entries in their original positions. Numbers that are to be compared are now in corresponding positions in A and B. We carry out all the comparisons simultaneously in a way pioneered by Paul and Simon (1980): All the test bits in A are set to 1, the test bits in B are left at 0, and B is subtracted from A. Now the test bit in a particular position "survives" if and only if the entry in A in that position is at least at large as the corresponding entry in B. Using the resulting sequence of test bits, it is easy to create a mask M' that separates the minima $((B$ AND $M')$ OR $(A - (A$ AND $M')))$ from the maxima $((A$ AND $M')$ OR $(B - (B$ AND $M')))$ and to move the latter left by 2^t fields. The complete program for the bitonic sorting follows.

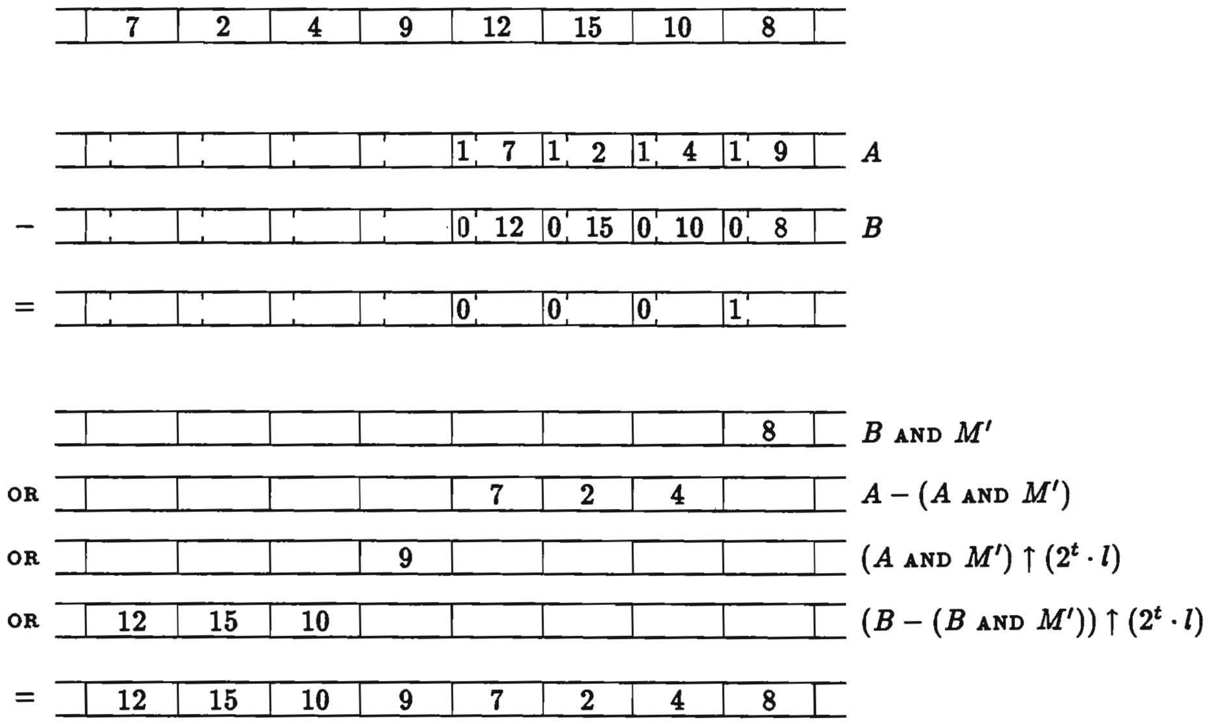

Fig. 1. One stage of bitonic sorting.

```
for $t := \log k$ downto 0 do
  begin
    $M := \mathit{CopyTestBit}((K_2 \uparrow (l - 1 - t))$ AND $K_1)$;
    (* Mask away fields with 0 in position $t$ of their address *)
    $A := (Z$ AND $M) \uparrow (-2^t \cdot l)$;
    $B := Z - (Z$ AND $M)$;
    $M' := \mathit{CopyTestBit}(((A$ OR $K_1) - B)$ AND $K_1)$;
    $Z := (B$ AND $M')$ OR
         $(A - (A$ AND $M'))$ OR
         $((A$ AND $M') \uparrow (2^t \cdot l))$ OR
         $((B - (B$ AND $M')) \uparrow (2^t \cdot l))$;
  end;
```

This completes the description of the algorithm to merge two sorted words. If the input numbers $x_0, \ldots, x_{k-1}$ and $y_0, \ldots, y_{k-1}$ are drawn from the range $1 \,..\, m$, a field must be able to store nonnegative integers up to $\max\{m, 2k-1\}$, in addition to the test bit; a field length of $\lceil \log(m+k) \rceil + 2$ bits therefore suffices. The running time of the algorithm clearly is $O(\log k)$.

4 Nonconservative sorting on the EREW PRAM

Given the ability to merge two sorted words, it is easy to develop a procedure for merging two sorted sequences given in word representation. Let $\mathcal{X}$ and $\mathcal{Y}$ be such sequences, each comprising r words. Although the sequences $\mathcal{X}$ and $\mathcal{Y}$ may contain repeated values, we will consider the elements of $\mathcal{X}$ and $\mathcal{Y}$ to be pairwise distinct, and we impose a total order on these elements by declaring an element z of $\mathcal{X}$ or $\mathcal{Y}$ to be smaller than another element z' of $\mathcal{X}$ or $\mathcal{Y}$ exactly if its value is smaller, or if it precedes z' in the same input sequence, or if z and z' have the same value, z belongs to $\mathcal{X}$, and z' belongs to $\mathcal{Y}$ (i.e., ties are broken by considering elements in $\mathcal{X}$ to be smaller). Define a *representative* as the first (smallest, rightmost) entry in a word of either $\mathcal{X}$ or $\mathcal{Y}$ and begin by extracting the representative of each word of $\mathcal{X}$ ($\mathcal{Y}$, respectively) to form a sorted sequence $x_1, \ldots, x_r$ ($y_1, \ldots, y_r$, respectively). Merge $x_1, \ldots, x_r$ and $y_1, \ldots, y_r$, according to the total order defined above, by means of a standard algorithm such as the one described by Hagerup and Rüb (1989), which uses $O(\log r + 1)$ time and $O(r)$ operations, and let $z_1, \ldots, z_{2r}$ be the resulting sequence. For $i = 1, \ldots, 2r$, associate a processor with z_i. The task of this processor is to merge the subsequences of $\mathcal{X}$ and $\mathcal{Y}$ comprising those input numbers z with $z_i \le z < z_{i+1}$ (take $z_{2r+1} = \infty$). One of these subsequences is part of the sequence stored in the word containing z_i, while the second subsequence is part of the sequence stored in the word containing z_j, where j is the maximal integer with $1 \le j < i$ such that z_i and z_j do not belong to the same input sequence; if there is no such j, the second subsequence is empty. Segmented broadcasting using $O(\log r + 1)$ time and $O(r)$ operations allows each processor associated with an element of $z_1, \ldots, z_{2r}$ to obtain copies of the at most two words containing the subsequences that it is to merge, after which the processor can use the algorithm described in the previous section to merge its words in $O(\log k)$ time. For $i = 1, \ldots, 2r$, the processor associated with z_i then locates z_i and z_{i+1} in the resulting sequence by means of binary search, which allows it to remove those input numbers z that do not satisfy $z_i \le z < z_{i+1}$. Finally the pieces of the output produced by different processors must be assembled appropriately in an output sequence. By computing the number of smaller input numbers, each processor can determine the precise location in the output of its piece, which spreads over at most three (in fact, two) output words. The processors then write their pieces to the appropriate output words. In order to avoid write conflicts, this is done in three phases, devoted to processors writing the first part of a word, the last part of a word, and neither (i.e., a middle part), respectively. To see that indeed no write conflict can occur in the third phase, observe that if two or more processors were to write to the same word in the third phase, at least one of them would contribute a representative coming from the same input sequence as the representative contributed by the processor writing the last part of the word (in the second phase); this is impossible, since no output word can contain two or more representatives from the same input sequence. We have proved

Lemma 1 *For all given integers n, m, k and l, where k is a power of 2 with $2 \le k \le n$, $m \ge 1$ and $l \ge \lceil \log(m + k) \rceil + 2$, two sorted sequences of n integers in the range $1\,..\,m$ each, given in the word representation with parameters k and l, can be merged on an EREW PRAM using $O(\log n)$ time, $O((n/k)\log k)$ operations, $O(n/k)$ space and a word length of $O(kl + \log n)$ bits.*

Theorem 1 *For all given integers n, k and m with $2 \le k \le n$ and $m \ge 2$ such that $\lfloor \log\log m \rfloor$ is known, n integers in the range $1\,..\,m$ can be sorted on an EREW PRAM using $O((\log n)^2)$ time, $O((n/k)\log k \log n + n)$ operations, $O(n)$ space and a word length of $O(k \log m + \log n)$ bits.*

Proof: Assume that n and k are powers of 2. For $m \le (\log n)^2$, the result is implied by the standard integer sorting results mentioned in the introduction. Furthermore, if $k \ge (\log n)^2 \ge 2$, we can replace k by $(\log n)^2$ without weakening the claim. Assume therefore that $k \le m$, in which case an integer l with $l \ge \lceil \log(m+k) \rceil + 2$, but $l = O(\log m)$ can be computed in constant time, using the given value of $\lfloor \log\log m \rfloor$. We use the word representation with parameters k and l and sort by repeated merging of ever longer sorted sequences, as in sequential merge sorting. During a first phase of the merging, consisting of $\log k$ merging rounds, the number of input numbers per word doubles in each round, altogether increasing from 1 to k. The remaining $O(\log n)$ rounds operate with k input numbers per word throughout. Both phases of the merging are executed according to Lemma 1. For $i = 1, \ldots, \log k$, the ith merging round inputs $n/2^{i-1}$ words, each containing a sorted sequence of 2^{i-1} input numbers, and merges these words in pairs, storing the result in $n/2^i$ words of 2^i input numbers each. By Lemma 1, the cost of the first phase of the merging, in terms of operations, is $O(\sum_{i=1}^{\log k}(n/2^{i-1})(i-1)) = O(n)$. The cost of the second phase is $O((n/k)\log k \log n)$, and the total time needed is $O((\log n)^2)$. □

Corollary 1 *For all integers $n, m \ge 4$, if $\lfloor \log\log m \rfloor$ is known, then n integers in the range $1\,..\,m$ can be sorted with a linear time-processor product in $O((\log n)^2)$ time on an EREW PRAM with a word length of $O(\log n \log\log n \log m)$ bits.*

5 Conservative sorting on the EREW PRAM

The sorting algorithm in the previous section is inherently nonconservative, in the sense that in most interesting cases, the algorithm can be applied only with a word length exceeding $\Theta(\log(n+m))$ bits. In this section we use a cascade of several different simple reductions to derive a conservative sorting algorithm. In general terms, a reduction allows us to replace an original sorting problem by a collection of smaller sorting problems, all of which are eventually solved using the algorithm of Theorem 1.

The first reduction is the well-known *radix sorting*, which we briefly describe. Suppose that we are to stably sort records whose keys are tuples of w components with respect to the lexicographical order on these tuples. Having at our disposal an algorithm that can sort with respect to single components, we can carry out the sorting in w successive phases. In the first phase, the records are sorted with respect to their least significant components, in the second phase they are sorted with respect to the second-least significant components, etc. Provided that the sorting in each phase is stable (so that it does not upset the order established by previous phases), after the wth phase the records will be stably sorted, as desired. We shall apply radix sorting to records with keys that are not actually tuples of components, but nonnegative integers that can be viewed as tuples by considering some fixed number of bits in their binary representation as one component.

For the problem of sorting n integers in the range $1\,..\,m$, define n as the *size* and m as the *height* of the sorting problem. For arbitrary positive integers n, m and w, where m is a power of 2, radix sorting can be seen to allow us to reduce an original sorting problem of size n and height m^w to w sorting problems of size n and height m each that must be solved one after the other (because the input of one problem depends on the output of the previous problem). The requirement that m should be a power of 2 ensures that when input numbers are viewed as w-tuples of integers in the range $1\,..\,m$, any given component of a tuple can be accessed in constant time (recall that we do not assume the availability of unit-time multiplication).

Our second reduction, which we call *group sorting*, allows us to reduce a problem of sorting integers in a sublinear range to several problems of sorting integers in a linear range. More precisely, if n and m are powers of 2 with $2 \le m \le n$, we can, spending $O(\log n)$ time and $O(n)$ operations, reduce a sorting problem of size n and height m to n/m sorting problems of size and height m each that can be executed in parallel. The method is as follows: Divide the given input numbers into $r = n/m$ groups of m numbers each and sort each group. Then, for $i = 1, \ldots, m$ and $j = 1, \ldots, r$, determine the number $n_{i,j}$ of occurrences of the value i in the jth group. Because each group is sorted, this can be done in constant time using $O(rm) = O(n)$ operations. Next compute the sequence $N_{1,1}, \ldots, N_{1,r}, N_{2,1}, \ldots, N_{2,r}, \ldots, N_{m,1}, \ldots, N_{m,r}$ of prefix sums of the sequence $n_{1,1}, \ldots, n_{1,r}, n_{2,1}, \ldots, n_{2,r}, \ldots, n_{m,1}, \ldots, n_{m,r}$ (the associative operation being usual addition), which takes $O(\log n)$ time and uses $O(n)$ operations. For $i = 1, \ldots, m$ and $j = 1, \ldots, r$, the last occurrence of i in (the sorted) jth group, if any, can now compute its position in the output sequence simply as $N_{i,j}$, after which all remaining output positions can be assigned using segmented broadcasting, an occurrence of i in the jth group d positions before the last occurrence of i in the jth group being placed at output position $N_{i,j} - d$.

We now describe an algorithm to sort n integers in the range $1\,..\,n$ with a word length of $O(\lambda)$ bits, where $\lambda \ge \log n$. Our approach is to sort numbers in a restricted range $1\,..\,2^s$, where s is a positive integer, and to apply the principle of radix sorting to sort numbers in the larger range $1\,..\,n$ in $O(\log n/s)$ phases, s bits at a time. Within each radix sort phase, n numbers in the range $1\,..\,2^s$ are sorted using the method of group sorting, where the algorithm of Theorem 1 is applied as the basic algorithm to sort each group. Hence each radix sort phase takes $O(s^2 + \log n)$ time and uses $O((n/k)s \log k + n)$ operations. Employed in a straightforward manner, the algorithm of Theorem 1 is able to sort only s-bit keys. Extending each key by another s bits and stabilizing the algorithm as described in Section 2, however, we can assume that the algorithm stably sorts the full input numbers (in the range $1\,..\,n$) by their s-bit keys within each group, a necessary prerequisite for the use of radix sorting.

Lemma 2 *For all given integers $n \ge 4$, $s \ge 1$ and $\lambda \ge \log n$, n integers in the range $1\,..\,n$ can be stably sorted on an EREW PRAM using*

$$O\left(\log n \left(s + \frac{\log n}{s}\right)\right) \quad \textit{time,}$$

$$O\left(n \log n \left(\frac{s \log \lambda}{\lambda} + \frac{1}{s}\right) + n\right) \quad \textit{operations,}$$

$O(n)$ space and a word length of $O(\lambda)$ bits.

Proof: If $s > \lfloor \log n \rfloor$, we can replace s by $\lfloor \log n \rfloor$, and if $\lambda > s^3$, we can replace λ by s^3, in each case without weakening the claim. Assume therefore that $1 \le s \le \log n$, in which case we can use the algorithm sketched above, and that $\lambda \le s^3$. Choose k as an integer with $2 \le k \le n$ such that $k = \Theta(\lambda/s)$, which causes the necessary word length to be $O(ks + \log n) = O(\lambda)$ bits, as required. The time needed by the algorithm is $O((\log n/s)(s^2 + \log n))$, as claimed, and the number of operations is

$$\begin{aligned} & O\left(\frac{\log n}{s}\left(\frac{ns\log k}{k} + n\right)\right) \\ = \; & O\left(n\log n\left(\frac{\log k}{k} + \frac{1}{s}\right)\right) \\ = \; & O\left(n\log n\left(\frac{s\log \lambda}{\lambda} + \frac{1}{s}\right)\right). \quad \square \end{aligned}$$

We obtain our best conservative sorting algorithm by combining Lemma 2 with a reduction due to Rajasekaran and Sen (1992), who showed that when n is a perfect square, a sorting problem of size and height n reduces to two batches of sorting problems, each batch consisting of $\sqrt{n}$ problems of size and height $\sqrt{n}$. The sorting problems comprising a batch can be executed in parallel, whereas the first batch must be completed before the processing of the second batch can start, and the reduction itself uses $O(\log n)$ time and $O(n)$ operations. The reduction is simple: Using the principle of radix sorting, the original sorting problem of size and height n is reduced to two sorting problems of size n and height $\sqrt{n}$, each of which in turn is reduced, using the principle of group sorting, to a collection of $\sqrt{n}$ sorting problems of size and height $\sqrt{n}$.

In general, we cannot assume that n is a perfect square, and the reduction above needs unit-time division, which is not part of our instruction repertoire. Without loss of generality, however, we can assume that n is a power of 2, and we can modify the argument to show that in this case a sorting problem of size and height n reduces, using $O(\log n)$ time and $O(n)$ operations, to two batches of sorting problems, each batch comprising sorting problems of total size n and individual size and height $2^{\lceil \log n/2 \rceil}$. Iterating this reduction $i \ge 1$ times yields a procedure that reduces a sorting problem of size and height n to 2^i batches of sorting problems, each batch comprising sorting problems of total size n and individual size and height $2^{\lceil \log n/2^i \rceil}$. The reduction itself uses $O(i \log n)$ time and $O(2^i n)$ operations; the latter quantity is always dominated by the number of operations needed to solve the subproblems resulting from the reduction, so that we need not account for it in the following. Our plan is to solve the small subproblems generated by the reduction using the algorithm of Lemma 2, with i chosen to make the total running time come out at a pre-specified value. This gives rise to a tradeoff between running time and work: Increasing i has the effect of lowering the running time, but raising the total number of operations.

In the case $m > n$ we appeal to yet another reduction. Suppose that we are given a sorting problem of size n and height $m > n$, where n is a power of 2, and view each input number as written in the positional system with basis n as a sequence of $w = \Theta(\log m/\log n)$ digits. As stated earlier, radix sorting can be thought of as reducing the original problem of size n and height m to w sorting problems of size and height n each that must be solved one after the other. Vaidyanathan *et al.* (1991) gave a different reduction that also results in a collection of $O(w)$ sorting problems of size and height n each. The difference is that the sorting problems resulting from the reduction of Vaidyanathan *et al.* can be partitioned into $O(\log(2+w))$ batches such that all subproblems in the same batch can be solved in parallel. Moreover, the number of subproblems in the ith batch, for $i = 1, 2, \ldots$, is at most $2^{1-i/3}w$. The reduction works as follows: Suppose that we sort the input numbers just by their two most significant digits (to basis n). This is one sorting problem of size n and height n^2, or (using radix sorting) two sorting problems of size and height n each. After the sorting we can replace each pair of first two digits by its rank in the sorted sequence of pairs without changing the relative order of the input numbers. The rank information can be obtained via segmented broadcasting within resource bounds dominated by those needed for the sorting, and since the ranks are integers in the range $1 \,..\, n$, we have succeeded in replacing two digit positions by just one digit position. We can do the same in parallel for the 3rd and 4th digit position, for the 5th and 6th position, and so on; if the total number of digit positions is odd, we sort the last digit position together with the two positions preceding it. This involves a total of w sorting problems of size and height n divided into three batches (one for each radix sorting phase), and it reduces the number of digit positions to at most $w/2$. We now simply proceed in the same way until the input numbers have been sorted by a single remaining digit.

Theorem 2 *Let $n, m \geq 4$ be integers and take $h = \min\{n, m\}$. Then, for all given integers $t \geq \log n + \log h \log\log h \log(2 + \log m/\log n)$ and $\lambda \geq \log(n+m)$, n integers in the range $1 \,..\, m$ can be stably sorted on an EREW PRAM using $O(t)$ time,*

$$O\left(n\left(\log m\sqrt{\frac{\log\lambda}{\lambda}} + \frac{\log h \log m}{t} + 1\right)\right)$$

operations, $O(n)$ space and a word length of $O(\lambda)$ bits.

Proof: Assume that $\lambda \leq (\log m)^3$, since otherwise we can replace λ by $2^{3\lfloor\log\log m\rfloor}$ without weakening the claim. Assume further that n and m are powers of 2 and consider first the case $m \leq n$. Using the principle of group sorting and spending $O(\log n)$ time and $O(n)$ operations, we can reduce the given problem to a batch of sorting problems of total size n and individual size and height m. We then compute positive integers τ and i with $\tau = \Theta(\min\{t, \log m\sqrt{\lambda/\log\lambda}\})$ and $2^i = \Theta((\log m)^3/\tau^2 + 1)$, carry out the i-level reduction described above and solve the resulting subproblems using the algorithm of Lemma 2 with $s = \Theta(\tau/\log m)$. What remains is to analyze the time and the number of operations needed. First note that

$$\lceil\log m/2^i\rceil = O(\log m/2^i) = O(\tau^2/(\log m)^2)$$

and that

$$2^i = O((\log m)^2/\tau + 1).$$

Since $i = O(\log\log m)$, the time needed for the i-level reduction is $O(i \log m) = O(t)$, and the time needed for the processing of 2^i batches of subproblems of size $2^{\lceil\log m/2^i\rceil}$ each is

$$O\left(2^i\lceil\log m/2^i\rceil\left(s + \frac{\lceil\log m/2^i\rceil}{s}\right)\right)$$

$$= O\left(\log m\left(\frac{\tau}{\log m} + \frac{\tau^2/(\log m)^2}{\tau/\log m}\right)\right) = O(\tau) = O(t).$$

The number of operations needed for the processing of 2^i batches of subproblems, each batch consisting of subproblems of total size n, is

$$\begin{aligned}
&O\left(2^i\left(n\lceil\log m/2^i\rceil\left(\frac{s\log\lambda}{\lambda} + \frac{1}{s}\right) + n\right)\right)\\
= \; &O\left(n\log m\left(\frac{\tau\log\lambda}{\lambda\log m} + \frac{\log m}{\tau}\right) + 2^i n\right)\\
= \; &O\left(n\left(\frac{\tau\log\lambda}{\lambda} + \frac{(\log m)^2}{\tau} + 1\right)\right)\\
= \; &O\left(n\left(\log m\sqrt{\frac{\log\lambda}{\lambda}} + \frac{(\log m)^2}{\tau} + 1\right)\right)\\
= \; &O\left(n\left(\log m\sqrt{\frac{\log\lambda}{\lambda}} + \frac{\log h\log m}{t} + 1\right)\right).
\end{aligned}$$

This proves Theorem 2 in the case $m \le n$. Consider now the case $m > n$ and suppose first that $\min\{\lambda/\lfloor\log\lambda\rfloor, t\} \le (\log n)^2$. We then use the reduction of Vaidyanathan *et al.* to reduce the original problem of size n and height m to $O(\log(2+w))$ batches of subproblems of size and height n each, where $w = \Theta(\log m/\log n)$, such that the ith batch comprises at most $2^{1-i/3}w$ subproblems. Each of the resulting subproblems is solved using the algorithm of Theorem 2 for the case $m = n$. The minimum time needed is $O(\log n\log\log n\log(2+w)) = O(\log h\log\log h\log(2+w))$, as claimed. In order to achieve a total time of $\Theta(t)$, we spend $\Theta(t)$ time on each of the six first batches, and from then on we spend only half as much time on each group of six successive batches as on the previous group. This clearly does indeed yield an overall time bound of $O(t)$. Since the number of subproblems in each group of six successive batches is at most $2^{-6/3} = 1/4$ of the number of subproblems in the previous group, while the available time is half that available to the previous group, it can be seen that the total number of operations executed is within a constant factor of the number of operations needed by the first batch, i.e., the total number of operations is

$$\begin{aligned}
&O\left(\frac{\log m}{\log n}\left(n\left(\log n\sqrt{\frac{\log\lambda}{\lambda}} + \frac{(\log n)^2}{t} + 1\right)\right)\right)\\
= \; &O\left(n\left(\log m\sqrt{\frac{\log\lambda}{\lambda}} + \frac{\log n\log m}{t} + \frac{\log m}{\log n}\right)\right)\\
= \; &O\left(n\left(\log m\sqrt{\frac{\log\lambda}{\lambda}} + \frac{\log h\log m}{t}\right)\right),
\end{aligned}$$

where the last transformation uses the upper bounds on λ and t assumed above.

On the other hand, if $\min\{\lambda/\lfloor\log\lambda\rfloor, t\} > (\log n)^2$, we use the algorithm of Theorem 1 with $k = \Theta(\lambda/\log m)$. The time needed is $O((\log n)^2) = O(t)$, the word length is $O(\lambda)$, and the number of operations is

$$O\left(\frac{n}{k}\log k\log n + n\right)$$

$$
\begin{aligned}
&= O\Big(\frac{n\log m}{\lambda}\log\lambda\log n + n\Big)\\
&= O\Big(n\Big(\log m\sqrt{\frac{\log\lambda}{\lambda}}\cdot\sqrt{\frac{\log\lambda}{\lambda}}\log n + 1\Big)\Big)\\
&= O\Big(n\Big(\log m\sqrt{\frac{\log\lambda}{\lambda}} + 1\Big)\Big). \quad \square
\end{aligned}
$$

Theorem 2 exhibits a general tradeoff between time, operations and word length. Specializing to the case of conservative sorting of integers in a linear range, we obtain

Corollary 2 *For all given integers $n \ge 4$ and $t \ge \log n \log\log n$, n integers in the range $1\,..\,n$ can be stably sorted on an EREW PRAM using $O(t)$ time, $O(n(\sqrt{\log n \log\log n} + (\log n)^2/t))$ operations, $O(n)$ space and a standard word length of $O(\log n)$ bits.*

Another interesting consequence of Theorem 2 is given in Corollary 3 below. A result corresponding to Corollary 3 was previously known only for $m = (\log n)^{O(1)}$, even for the stronger CRCW PRAM.

Corollary 3 *For all integers $n \ge 4$ and $m \ge 1$, if $m = 2^{O(\sqrt{\log n/\log\log n})}$, then n integers in the range $1\,..\,m$ can be stably sorted on an EREW PRAM using $O(\log n)$ time, $O(n)$ operations, $O(n)$ space and a standard word length of $O(\log n)$ bits.*

6 Algorithms for the CREW PRAM

If concurrent reading is allowed, we can use table lookup to merge two sorted words in time $O(1)$, rather than $O(\log k)$. This possibility was pointed out to us by Richard Anderson. A table that maps two arbitrary sorted words given in the word representation with parameters k and l to the words obtained by merging these has at most 2^{2kl} entries, each of which can be computed sequentially in $O(\log k)$ time using the algorithm of Section 3. Similarly, a table that maps each pair consisting of a sorted word and an l-bit integer to the rank of the given integer in the given word contains at most 2^{2kl} entries, each of which can be computed in $O(\log k)$ time by means of binary search. These tables can therefore be constructed using $O(\log k)$ time and $O(2^{2kl}\log k)$ operations. In the following we wish to distinguish between the resources needed to construct such tables and those consumed by merging or sorting proper, the reason being that if an original problem is reduced to a collection of subproblems with the same parameters (size, height, etc.), then all subproblems can be solved using the same tables, so that the cost of constructing the tables can be amortized over the subproblems. We choose to use the term "preprocessing" to denote the resources needed to construct and store tables that depend only on the size parameters of the input problem, but not on the actual numbers to be merged or sorted.

Lemma 3 *For all given integers n, m, k and l, where k is a power of 2 with $2 \le k \le n$, $m \ge 8$ and $l \ge \lceil\log(m+k)\rceil + 2$, and for all fixed $\epsilon > 0$, two sorted sequences of n integers in the range $1\,..\,m$ each, given in the word representation with parameters k and l, can be merged on a CREW*

PRAM using $O(\log n)$ preprocessing time, $O(2^{2kl}\log k + n)$ preprocessing operations and space, $O(\log\log\log m)$ time, $O(n/k)$ operations, $O(nm^{\epsilon}/k)$ space and a word length of $O(kl + \log n)$ bits. Moreover, if $m = O(n/(k\log n))$, the merging time and space can be reduced to $O(1)$ and $O(n)$, respectively.

Proof: We use an algorithm similar to that of Lemma 1, the main item of interest being how to merge the two sequences of $O(n/k)$ representatives. By a process of repeated squaring, executed during the preprocessing, we can either compute $\lfloor\log\log\log m\rfloor$ or determine that $m \geq 2^n$, in both cases without ever creating an integer of more than $\Theta(\log(n+m))$ bits. If $m \geq 2^n$, the representatives can be merged in $O(\log\log n) = O(\log\log\log m)$ time with the algorithm of Kruskal (1983). Otherwise the representatives can be merged in $O(\log\log\log m)$ time with the algorithm of Berkman and Vishkin (1993), which is responsible for the superlinear space requirements. If $m = O(n/(k\log n))$, finally, the representatives can be merged in constant time and linear space, as noted by Chaudhuri and Hagerup (1994); the latter result assumes the availability of certain integer values that can be computed during the preprocessing. In each case the number of operations needed is $O(n/k)$.

When the representatives have been merged, each processor associated with a representative can access the two words that it is to merge directly, without resorting to segmented broadcasting. The merging itself and the removal of input numbers outside of the interval of interest is done in constant time using table lookup, the relevant tables having been constructed during the preprocessing as described above. The remainder of the computation works in constant time using $O(n/k)$ operations even on the EREW PRAM. $\square$

The remaining development for the CREW PRAM parallels that for the EREW PRAM, for which reason we provide a somewhat terse description. We refrain from developing a series of nonconservative CREW PRAM algorithms that employ fast merging, but not table lookup.

Theorem 3 *For all given integers n, k and m with $2 \leq k \leq n$ and $m \geq 8$ such that $\lfloor\log\log m\rfloor$ is known and for all fixed $\epsilon > 0$, n integers in the range $1\,..\,m$ can be sorted on a CREW PRAM using $O(\log n)$ preprocessing time, $O(2^{O(k\log m)} + n)$ preprocessing operations and space, $O(\log n\log\log\log m)$ time, $O((n/k)\log n + n)$ operations, $O(nm^{\epsilon})$ space and a word length of $O(k\log m + \log n)$ bits. Moreover, if $m = O(n)$, the time and space bounds for the sorting can be reduced to $O(\log n)$ and $O(n)$, respectively.*

Proof: We can assume that n and k are powers of 2 and that $k \leq \log n \leq m$, so that an integer $l \geq \lceil\log(m+k)\rceil + 2$ with $l = O(\log m)$ can be computed in constant time. We use the word representation with parameters k and l and sort using repeated merging as in the algorithm of Theorem 1. Each round of merging is executed using the algorithm of Lemma 3, and the complete sorting can be carried out in $O(\log n\log\log\log m)$ time using $O((n/k)\log n)$ operations.

If $m = O(n)$, we can use radix sorting to replace the original sorting problem of size n and height m by two sorting problems of size n and height $O(n/(k\log m))$ each. $\square$

Lemma 4 *For all given integers $n \geq 2$, $s \geq 1$ and $\lambda \geq \log n$, n integers in the range $1\,..\,n$ can be stably sorted on a CREW PRAM using $O(\log n)$ preprocessing time, $2^{O(\lambda)}$ preprocessing operations and space,*

$$O\left(\frac{(\log n)^2}{s} + \log n\right) \text{ time,}$$

$$O\left(n \log n \left(\frac{s}{\lambda} + \frac{1}{s}\right) + n\right) \text{ operations,}$$

$O(n)$ space and a word length of $O(\lambda)$.

Proof: We can assume that $s \leq \log n$ and that $\lambda \leq s^2$. We use radix sorting and group sorting to reduce the original problem to $O(\log n/s)$ batches, each comprising sorting problems of total size n and individual size and height 2^s. We solve each subproblem using the algorithm of Theorem 3 with k chosen as an integer with $2 \leq k \leq n$ such that $k = \Theta(\lambda/s)$. The total time needed is $O((\log n/s)\log n)$, and the number of operations is

$$O\left(\frac{\log n}{s}\left(\frac{ns}{k} + n\right)\right) = O\left(n \log n\left(\frac{s}{\lambda} + \frac{1}{s}\right)\right). \quad \Box$$

The theorem below is our main result for the CREW PRAM. Since it describes a stand-alone algorithm, the cost of table construction is not indicated separately, but incorporated into the overall resource bounds. We fix the word length at the standard $O(\log(n+m))$ bits, since otherwise the preprocessing cost would be so high as to make the result uninteresting.

Theorem 4 *Let $n, m \geq 2$ be integers and take $h = \min\{n, m\}$ and $t' = \log h \log(2 + \log m/\log n)$. Then, for all given integers $t \geq t' + \log n$, n integers in the range $1\,..\,m$ can be stably sorted on a CREW PRAM using $O(t)$ time,*

$$O\left(n\left(\frac{\log m}{\sqrt{\log n}} + \frac{\log h \log m}{\log n \cdot 2^{t/t'}} + 1\right)\right)$$

operations, $O(n)$ space and a standard word length of $O(\log(n+m))$ bits.

Proof: Assume that n and m are powers of 2 and consider first the case $m \leq n$. Using group sorting, we begin by reducing the given problem to a batch of sorting problems of total size n and individual size and height m. We then compute positive integers a, s and i with $a \geq t/\log m$, but $a = O(t/\log m)$, $s = \Theta(\sqrt{\log n} + \log m/(a \cdot 2^a))$ and $2^i = \Theta(\log m/s + 1)$, use the i-level reduction of Rajasekaran and Sen and solve all the resulting subproblems using the algorithm of Lemma 4.

Recall that, used with a word length of λ, the algorithm of Lemma 4 needs $2^{O(\lambda)}$ preprocessing operations and space. We choose $\lambda = \Theta(\log n)$ sufficiently small to make the preprocessing cost $O(n)$. For this we must ensure that the algorithm of Lemma 4 is applied to inputs of size at most 2^λ. But since the input size is $2^{\lceil \log m/2^i \rceil} \leq 2^{\lceil \log n/2^i \rceil}$, this is simply a matter of always choosing i larger than a fixed constant.

Since $2^i = O(a \cdot 2^a)$, the time needed for the i-level reduction is $O(i \log m) = O(a \log m) = O(t)$. Furthermore, since $\lceil \log m/2^i \rceil = O(s)$, the time needed for the processing of 2^i batches of subproblems of size $2^{\lceil \log m/2^i \rceil}$ each is

$$O\left(2^i\left(\frac{\lceil \log m/2^i \rceil^2}{s} + \lceil \log m/2^i \rceil\right)\right) = O(2^i s) = O(\log m + \sqrt{\log n}) = O(t).$$

The number of operations needed for the processing of 2^i batches of subproblems, each batch consisting of subproblems of total size n, is

$$\begin{aligned}
& O\left(2^i\left(n\lceil \log m/2^i \rceil\left(\frac{s}{\log n} + \frac{1}{s}\right) + n\right)\right) \\
= {} & O\left(n \log m\left(\frac{s}{\log n} + \frac{1}{s}\right) + n\right) \\
= {} & O\left(n \log m\left(\frac{\sqrt{\log n} + \log m/(a \cdot 2^a)}{\log n} + \frac{1}{\sqrt{\log n}}\right) + n\right) \\
= {} & O\left(n \log m\left(\frac{1}{\sqrt{\log n}} + \frac{(\log m)^2}{t \log n \cdot 2^a}\right) + n\right) \\
= {} & O\left(n\left(\frac{\log m}{\sqrt{\log n}} + \frac{\log h \log m}{\log n \cdot 2^{t/t'}} + 1\right)\right).
\end{aligned}$$

This proves Theorem 4 in the case $m \le n$. In the case $m > n$ we use the reduction of Vaidyanathan *et al.* to reduce the original problem to $O(\log(2 + \log m/\log n))$ batches of subproblems of size and height n each and solve each subproblem using the algorithm of Theorem 4 for the case $m = n$, with $\Theta(t \log n/t')$ time allotted to each batch. This gives a total time of $O(t)$, and the total number of operations needed is

$$\begin{aligned}
& O\left(\frac{\log m}{\log n}\left(n\left(\sqrt{\log n} + \frac{\log n}{2^{t/t'}} + 1\right)\right)\right) \\
= {} & O\left(n \log m\left(\frac{1}{\sqrt{\log n}} + \frac{1}{2^{t/t'}}\right)\right) \\
= {} & O\left(n\left(\frac{\log m}{\sqrt{\log n}} + \frac{\log h \log m}{\log n \cdot 2^{t/t'}} + 1\right)\right). \quad \square
\end{aligned}$$

Corollary 4 *For all given integers $n \ge 2$ and $t \ge \log n$, n integers in the range $1\,..\,n$ can be sorted on a CREW PRAM using $O(t)$ time, $O(n(\sqrt{\log n} + \log n/2^{t/\log n}))$ operations, $O(n)$ space and a standard word length of $O(\log n)$ bits.*

It is instructive to compare the tradeoff of Corollary 4 with that of the algorithm of Kruskal *et al.* (1990a). Put in a form analogous to that of Corollary 4, the result of Kruskal *et al.* states that for all $t \ge 2\log n$, n integers in the range $1\,..\,n$ can be sorted in $O(t)$ time with a time-processor product of $O\left(\frac{n \log n}{\log(t/\log n)} + n\right)$. Our algorithm and that of Kruskal *et al.* therefore pairs the same minimum time of $\Theta(\log n)$ with the same operation count of $\Theta(n \log n)$, i.e., no savings relative to comparison-based sorting. Allowing more time decreases the number of operations in both cases, but the number of operations of our algorithm decreases doubly-exponentially faster than the corresponding quantity for the algorithm of Kruskal *et al.* and reaches its minimum of $\Theta(n\sqrt{\log n})$ already for $t = \Theta(\log n \log\log n)$. If we allow still more time, our algorithm does not become any cheaper, and the algorithm of Kruskal *et al.* catches up for $t = 2^{\Theta(\sqrt{\log n})}$ and is more efficient from that point on.

Placed in a similar context, our EREW PRAM sorting algorithm of Corollary 2 exhibits a tradeoff that is intermediate between the two tradeoffs discussed above. As the time increases, the number of operations decreases exponentially faster than for the algorithm of Kruskal *et al.*

Corollary 5 *For all integers $n \geq 2$ and $m \geq 1$, if $m = 2^{O(\sqrt{\log n})}$, then n integers in the range $1\,..\,m$ can be stably sorted on a CREW PRAM using $O(\log n)$ time, $O(n)$ operations, $O(n)$ space and a standard word length of $O(\log n)$ bits.*

Allowing randomization and assuming the availability of unit-time multiplication and integer division, we can extend the time and processor bounds of Corollary 4 to integers drawn from an arbitrary range. Suppose that the word length is λ bits, where $\lambda \geq \log n$, so that the numbers to be sorted come from the range $0\,..\,2^\lambda - 1$. Following Fredman and Willard (1990), we assume the availability of a fixed number of constants that depend only on λ. Using an approach very similar to ours, Raman (1991a) recently showed that n integers of arbitrary size can be sorted on a CREW PRAM in $O(\log n \log\log n)$ time using $O(n \log n / \log\log n)$ operations with high probability.

The basic idea is very simple. First we choose a random sample V from the set of input elements, sort V by standard means and determine the rank in V of each input element. We then use the algorithm of Corollary 4 to sort the input elements by their ranks and finally sort each group of elements with a common rank, again by a standard sorting algorithm. If each group is relatively small, the last step is not too expensive. The other critical step is the computation of the rank in V of each input element. An obvious way to execute this step is to store the elements in V in sorted order in a search tree T, and then to carry out n independent searches in T, each using a different input element as its key. Since we aim for an operation count of $O(n\sqrt{\log n})$, however, T cannot be a standard balanced binary tree, and we have to resort to more sophisticated data structures, namely the fusion tree of Fredman and Willard (1990) and the priority queue of van Emde Boas (1977). Building on an approach outlined in Section 6 of (Fredman and Willard, 1990), we use a fusion tree that is a complete d-ary tree, where $d \geq 2$ and $d = 2^{\Theta(\sqrt{\log n})}$, with the elements of the sample V stored in sorted order in its leaves. The distinguishing property of a fusion tree is that in spite of its high node degree, a search can proceed from a parent node to the correct child node in constant time. Since the depth of our fusion tree is $O(\sqrt{\log n})$, it allows the rank of all input elements to be determined using $O(n\sqrt{\log n})$ operations, which precisely matches what we have to pay for sorting the input elements by their ranks.

The fusion tree makes crucial use of very large integers. Specifically, the constraint is that we must be able to represent numbers of $d^{O(1)}$ bits in a constant number of words. It follows that we can use the fusion tree if $\lambda \geq 2^{\lfloor\sqrt{\log n}\rfloor}$. If $\lambda < 2^{\lfloor\sqrt{\log n}\rfloor}$, we replace the fusion tree as our search structure by a van Emde Boas (vEB) structure (van Emde Boas, 1977). The latter supports sequential search operations in $O(\log \lambda)$ time. Since we use a vEB structure only if $\lambda < 2^{\sqrt{\log n}}$, this is again sufficient for our purpose. The main outstanding difficulty is the construction of fusion trees and vEB structures, for which we use randomized algorithms. When discussing randomized algorithms below, we always intend these to "fail gracefully", i.e., if they cannot carry out the task for which they were designed, they report failure without causing concurrent writing or other error conditions.

A fusion tree node contains various tables that can be constructed sequentially in $O(d^4)$ time. The sequential algorithm can be parallelized to yield a randomized algorithm that constructs a fusion tree node with probability at least $1/2$ and uses $O(\log d)$ time, $d^{O(1)}$ operations and $d^{O(1)}$ space; we present the details in an appendix. Letting this algorithm be executed in parallel by $\lceil \log v \rceil + 2$ independent processor teams, for $v \geq 1$, we can reduce the probability that a fusion tree node is not constructed correctly to at most $1/(4v)$. Hence the entire fusion tree for a sorted set of v elements, which has fewer than $2v$ nodes, can be constructed in $O(\log v)$ time using $d^{O(1)} v \log v$ operations and $d^{O(1)} v \log v$ space with probability at least $1/2$. As concerns the construction of a vEB structure for a sorted set of v elements, Raman (1991a) gives a randomized algorithm for this task that uses $O(\log v)$ time, $O(v)$ operations and $O(v)$ space and works with probability at least $1/2$ (in fact, with a much higher probability). We now give the remaining details of the complete sorting algorithm.

Theorem 5 *For every fixed integer $\alpha \geq 1$ and for all given integers $n \geq 2$ and $t \geq \log n$, n (arbitrary) integers can be sorted on a CREW PRAM using $O(t)$ time, $O(n(\sqrt{\log n} + \log n / 2^{t/\log n}))$ operations and $O(n)$ space with probability at least $1 - 2^{-2^{\alpha\sqrt{\log n}}}$.*

Proof: Let $x_1, \ldots, x_n$ be the input elements, assume these to be pairwise distinct and let $\beta \geq 1$ be an integer constant whose value will be fixed below. Execute the following steps.

1. Draw $v = \lceil n / 2^{\beta \lfloor \sqrt{\log n} \rfloor} \rceil$ independent integers $i_1, \ldots, i_v$ from the uniform distribution over $\{1, \ldots, n\}$. Construct the set $V = \{x_{i_1}, \ldots, x_{i_v}\}$ and sort it.

2. Construct a search structure T for the set V. If $\lambda \geq 2^{\lfloor \sqrt{\log n} \rfloor}$ let T be a fusion tree; otherwise let T be a vEB structure. In order to obtain T with sufficiently high probability, carry out $2^{(\alpha+1)\lceil \sqrt{\log n} \rceil}$ independent attempts to construct T, each attempt succeeding with probability at least $1/2$. By the discussion before the statement of Theorem 5, this can be done in $O(\log n)$ time using $2^{(\alpha+1)\lceil \sqrt{\log n} \rceil} \cdot d^{O(1)} v \log v = 2^{O(\sqrt{\log n})} \cdot v$ operations and space. For β chosen sufficiently large, the bound on operations and space is $O(n)$, and the probability that T is not constructed correctly is at most $2^{-2^{(\alpha+1)\sqrt{\log n}}}$.

3. Use T to compute the rank of x_j is V, for $j = 1, \ldots, n$. This uses $O(\sqrt{\log n})$ time and $O(n\sqrt{\log n})$ operations.

4. Use the algorithm of Corollary 4 to sort the input elements $x_1, \ldots, x_n$ with respect to their ranks. This uses $O(t)$ time and $O(n(\sqrt{\log n} + \log n / 2^{t/\log n}))$ operations. For $i = 0, \ldots, v$, let X_i be the set of those input elements whose rank in V is i. Step 4 moves the elements in X_i to consecutive positions, for $i = 0, \ldots, v$.

5. Sort each of $X_0, \ldots, X_v$ using, e.g., Cole's merge sorting algorithm (Cole, 1988). If $M = \max\{|X_i| : 0 \leq i \leq v\}$, this takes $O(\log n)$ time and $O(\sum_{i=0}^{v} |X_i| \log(|X_i| + 1)) = O(n \log M)$ operations.

The resources used by Step 1 are negligible. Hence all that remains is to show that with sufficiently high probability, $\log M = O(\sqrt{\log n})$. But if $\log M > u + 1$, where $u = 2\beta\sqrt{\log n}$, the sampling in Step 1 misses at least 2^u consecutive elements in the sorted sequence of input elements, the probability of which is at most

$$n\left(1 - \frac{2^u}{n}\right)^v \le n \cdot e^{-2^u v/n}$$
$$\le n \cdot e^{-2^{2\beta\sqrt{\log n} - \beta\sqrt{\log n}}} = n \cdot e^{-2^{\beta\sqrt{\log n}}}.$$

For β chosen sufficiently large, the latter probability and the failure probability of $2^{-2^{(\alpha+1)\sqrt{\log n}}}$ in Step 2 add up to at most $2^{-2^{\alpha\sqrt{\log n}}}$. $\square$

For $t \ge \log n \log\log n$, the algorithm of Theorem 5 exhibits optimal speedup relative to the sequential randomized algorithm described by Fredman and Willard (1990).

Acknowledgment

We are grateful to Christine Rüb and Richard Anderson, whose suggestions allowed us to improve our results and to simplify the exposition.

References

Ajtai, M., Komlós, J., and Szemerédi, E. (1983), An $O(n \log n)$ sorting network, *in* Proc. 15th Annual ACM Symposium on Theory of Computing, pp. 1–9.

Bast, H., and Hagerup, T. (1993), Fast parallel space allocation, estimation and integer sorting (revised), Tech. Rep. no. MPI–I–93–123, Max-Planck-Institut für Informatik, Saarbrücken.

Batcher, K. E. (1968), Sorting networks and their applications, *in* Proc. AFIPS Spring Joint Computer Conference, 32, pp. 307–314.

Berkman, O., and Vishkin, U. (1993), On parallel integer merging, *Inform. and Comput.* **106**, pp. 266–285.

Bhatt, P. C. P., Diks, K., Hagerup, T., Prasad, V. C., Radzik, T., and Saxena, S. (1991), Improved deterministic parallel integer sorting, *Inform. and Comput.* **94**, pp. 29–47.

Chaudhuri, S., and Hagerup, T. (1994), Prefix graphs and their applications, *in* Proc. 20th International Workshop on Graph-Theoretic Concepts in Computer Science, Springer Lecture Notes in Computer Science, to appear.

Cole, R. (1988), Parallel merge sort, *SIAM J. Comput.* **17**, pp. 770–785.

Cole R., and Vishkin, U. (1986), Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control* **70**, pp. 32–53.

Fredman, M. L., and Willard, D. E. (1990), BLASTING through the information theoretic barrier with FUSION TREES, *in* Proc. 22nd Annual ACM Symposium on Theory of Computing, pp. 1–7.

Gil, J., Matias, Y., and Vishkin, U. (1991), Towards a theory of nearly constant time parallel algorithms, *in* Proc. 32nd Annual Symposium on Foundations of Computer Science, pp. 698–710.

Goodrich, M. T., Matias, Y., and Vishkin, U. (1993), Approximate parallel prefix computation and its applications, *in* Proc. 7th International Parallel Processing Symposium, pp. 318–325.

Goodrich, M. T., Matias, Y., and Vishkin, U. (1994), Optimal parallel approximation for prefix sums and integer sorting, *in* Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 241–250.

Hagerup, T. (1991), Constant-time parallel integer sorting, *in* Proc. 23rd Annual ACM Symposium on Theory of Computing, pp. 299–306.

Hagerup, T. and Raman, R. (1992), Waste makes haste: Tight bounds for loose parallel sorting, *in* Proc. 33rd Annual Symposium on Foundations of Computer Science, pp. 628–637.

Hagerup, T. and Raman, R. (1993), Fast deterministic approximate and exact parallel sorting, *in* Proc. 5th Annual ACM Symposium on Parallel Algorithms and Architectures, pp. 346–355.

Hagerup, T., and Rüb, C. (1989), Optimal merging and sorting on the EREW PRAM, *Inform. Process. Lett.* **33**, pp. 181–185.

Hagerup, T., and Shen, H. (1990), Improved nonconservative sequential and parallel integer sorting, *Inform. Process. Lett.* **36**, pp. 57–63.

JáJá, J. (1992), An Introduction to Parallel Algorithms, Addison-Wesley, Reading, Mass.

Kirkpatrick, D., and Reisch, S. (1984), Upper bounds for sorting integers on random access machines, *Theoret. Comput. Sci.* **28**, pp. 263–276.

Kruskal, C. P. (1983), Searching, merging, and sorting in parallel computation, *IEEE Trans. Comput.* **32**, pp. 942–946.

Kruskal, C. P., Rudolph, L., and Snir, M. (1990a), Efficient parallel algorithms for graph problems, *Algorithmica* **5**, pp. 43–64.

Kruskal, C. P., Rudolph, L., and Snir, M. (1990b), A complexity theory of efficient parallel algorithms, *Theoret. Comput. Sci.* **71**, pp. 95–132.

Matias, Y., and Vishkin, U. (1991a), On parallel hashing and integer sorting, *J. Algorithms* **12**, pp. 573–606.

Matias, Y., and Vishkin, U. (1991b), Converting high probability into nearly-constant time – with applications to parallel hashing, *in* Proc. 23rd Annual ACM Symposium on Theory of Computing, pp. 307–316.

Paul, W. J., and Simon, J. (1980), Decision trees and random access machines, *in* Proc. International Symposium on Logic and Algorithmic, Zürich, pp. 331–340.

Rajasekaran, S., and Reif, J. H. (1989), Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM J. Comput.* **18**, pp. 594–607.

Rajasekaran, S., and Sen, S. (1992), On parallel integer sorting, *Acta Inform.* **29**, pp. 1–15.

Raman, R. (1990), The power of collision: Randomized parallel algorithms for chaining and integer sorting, *in* Proc. 10th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer Lecture Notes in Computer Science, Vol. 472, pp. 161–175.

Raman, R. (1991a), The power of collision: Randomized parallel algorithms for chaining and integer sorting, Tech. Rep. no. 336, Computer Science Department, University of Rochester, New York, March 1990 (revised January 1991).

Raman, R. (1991b), Optimal sub-logarithmic time integer sorting on the CRCW PRAM, Tech. Rep. no. 370, Computer Science Department, University of Rochester, New York, January 1991.

Vaidyanathan, R., Hartmann, C. R. P., and Varshney, P. K. (1991), Optimal parallel lexicographic sorting using a fine-grained decomposition, Tech. Rep. no. SU–CIS–91–01, School of Computer and Information Science, Syracuse University, Syracuse, NY.

van Emde Boas, P. (1977), Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.* **6**, pp. 80-82.

Wagner, R. A., and Han, Y. (1986), Parallel algorithms for bucket sorting and the data dependent prefix problem, *in* Proc. 1986 International Conference on Parallel Processing, pp. 924–930.

Appendix: Constructing a fusion-tree node in parallel

In this appendix we show that if λ and d are positive integers with $2 \le d^6 \le \lambda$, then a fusion-tree node for $d-1$ given integers $y_1, \ldots, y_{d-1}$ drawn from the set $U = \{0, \ldots, 2^\lambda - 1\}$ can be constructed on a CREW PRAM with a word length of λ bits using $O(\log d)$ time, $d^{O(1)}$ operations and $d^{O(1)}$ space with probability at least 1/2. Our algorithm is a straightforward parallelization of the corresponding sequential algorithm of Fredman and Willard (1990).

For any integer x and any finite set S of integers, denote by $rank(x, S)$ the rank of x in S, and let $Y = \{y_1, \ldots, y_{d-1}\}$. Recall that the purpose of the fusion-tree node is to enable $rank(x, Y)$ to be determined in constant time by a single processor, for arbitrary given $x \in U$. If $y_1, \ldots, y_{d-1}$ are numbers of just $O(\lambda/d)$ bits, this can be done essentially as described in Section 3: Create a word B of $O(\lambda)$ bits containing $y_1, \ldots, y_{d-1}$ in separate fields, with test bits set to zero, and another word A containing $d-1$ copies of x, with test bits set to one, then subtract B from A and clear all bit positions, except those of the test bits, in the resulting word C. The test bits of C can be added and their sum placed in a single field by means of a suitable multiplication; this number is $rank(x, Y)$. In the following we will reduce the general rank computation to two such rank computations in sequences of small integers.

For arbitrary $x, y \in U$, define $msb(x, y)$ as -1 if $x = y$, and otherwise as the largest number of a bit position in which x and y differ. Fredman and Willard demonstrated that $msb(x, y)$ can be determined in constant time by a single processor, for arbitrary given $x, y \in U$. Without loss of generality assume that $0 < y_1 < \cdots < y_{d-1} < 2^\lambda - 1$. In order to avoid special cases, we introduce the two additional keys $y_0 = 0$ and $y_d = 2^\lambda - 1$. Let $P = \{msb(y_{i-1}, y_i) : 1 \le i \le d\}$

and write $P = \{p_1, \ldots, p_r\}$ with $p_1 < \cdots < p_r$; clearly $r \le d$. Define $f : U \to \{0, \ldots, 2^r - 1\}$ as the function that extracts the bit positions in P and packs them tightly, i.e., bit number i in $f(y)$ is bit number p_{i+1} in y, for $i = 0, \ldots, r-1$ and for all $y \in U$, while bits number $r, \ldots, \lambda - 1$ are zero in $f(y)$. It is important to note that $f(y_1) < \cdots < f(y_{d-1})$. We write $f(Y)$ for $\{f(y_1), \ldots, f(y_{d-1})\}$.

Let $\Delta = \{0, \ldots, d\} \times \{0, \ldots, r\} \times \{0, 1\}$ and define a function $\phi : U \to \Delta$ as follows: For $x \in U$, let $i = \mathit{rank}(f(x), f(Y))$ and choose $j \in \{i, i+1\}$ to minimize $\mathit{msb}(f(x), f(y_j))$, resolving ties by taking $j = i$. Note that this means that among all elements of $f(Y \cup \{y_0, y_d\})$, $f(y_j)$ is one with a longest prefix in common with $f(x)$ (the *longest-prefix property*). Furthermore take $l = \mathit{rank}(\mathit{msb}(x, y_j), P)$ and let $a = 1$ if $x \ge y_j$, $a = 0$ otherwise. Then $\phi(x) = (j, l, a)$. Fredman and Willard showed that for $x \in U$, $\mathit{rank}(x, Y)$ depends only on $\phi(x)$. (This is easy to see by imagining a digital search tree T for the bit strings $y_0, \ldots, y_d$. The root-to-leaf path in T corresponding to y_j is a continuation of the path taken by a usual search in T for x, and P is the set of heights of children of nodes in T of degree 2, so that l determines the maximal path in T of nodes of degree ≤ 1 on which the search for x terminates.) As a consequence, once $\phi(x)$ is known, $\mathit{rank}(x, Y)$ can be determined in constant time as $R[\phi(x)]$, where R is a precomputed table with $2(d+1)(r+1)$ entries.

Since both images under f and bit positions are sufficiently small integers, the two rank computations in the algorithm implicit in the definition of ϕ can be carried out in constant sequential time as described earlier in the appendix. As a consequence, we are left with two problems: How to construct the table R, and how to evaluate f in constant time.

As $\mathit{rank}(x, Y)$ can be determined in $O(\log d)$ sequential time for any given $x \in U$, it suffices for the construction of R to provide a set $X \subseteq U$ of $d^{O(1)}$ "test values" that "exercise" the whole table, i.e., such that $\phi(X) = \phi(U)$. This is easy: Take $p_0 = -1$ and $p_{r+1} = \lambda$ and define $x_{j,l,a}$, for all $(j, l, a) \in \Delta$, as the integer obtained from y_j by complementing the highest-numbered bit whose number is smaller than p_{l+1} and whose value is $1 - a$; if there is no such bit, take $x_{j,l,a} = y_j$. We will show that if $(j, l, a) \in \phi(U)$, then $\phi(x_{j,l,a}) = (j, l, a)$, which proves that $X = \{x_{j,l,a} : (j, l, a) \in \Delta\}$ is a suitable "test set". It may be helpful to visualize the following arguments as they apply to the digital search tree mentioned above.

Fix $(j, l, a) \in \Delta$ and let $S = \{x \in U : p_l \le \mathit{msb}(x, y_j) < p_{l+1} \text{ and } (x \ge y_j \Leftrightarrow a = 1)\}$. Elements of S are the only candidates for being mapped to (j, l, a) by ϕ.

Suppose first that $\mathit{msb}(x_{j,l,a}, y_j) > p_l$. Then $f(x_{j,l,a}) = f(y_j)$, so that choosing $i = j$ clearly achieves the unique minimum of -1 of $\mathit{msb}(f(x_{j,l,a}), f(y_i)\}$. By the longest-prefix property and the fact that $x_{j,l,a} \in S$, it now follows that $\phi(x_{j,l,a}) = (j, l, a)$.

If $\mathit{msb}(x_{j,l,a}, y_j) < p_l$, it is easy to see that $S = \emptyset$, so that $(j, l, a) \notin \phi(U)$.

If $\mathit{msb}(x_{j,l,a}, y_j) = p_l$, finally, it can be seen that $\mathit{msb}(x, y_j) = p_l$ for all $x \in S$. Consider two cases: If $\mathit{msb}(x_{j,l,a}, y_i) < p_l$ for some $i \in \{0, \ldots, d\}$, then $\mathit{msb}(f(x), f(y_i)) < \mathit{msb}(f(x), f(y_j))$ for all $x \in S$. Then, by the longest-prefix property, no $x \in S$ is mapped to (j, l, a), i.e., $(j, l, a) \notin \phi(U)$. If $\mathit{msb}(x_{j,l,a}, y_i) \ge p_l$ for all $i \in \{0, \ldots, d\}$, on the other hand, $\phi(x) = \phi(x_{j,l,a})$ for all $x \in S$, so that $\phi(x_{j,l,a}) = (j, l, a)$ if $(j, l, a) \in \phi(U)$. Summing up, the useful part of R can be constructed in $O(\log d)$ time with d^2 processors.

We actually do not know how to evaluate f efficiently, and Fredman and Willard employ a different function g that still extracts the bit positions in P, but packs them less tightly. More precisely, for nonnegative integers $q_1, \ldots, q_r$ of size $O(\lambda)$ to be determined below, bit number p_i in y is bit number $p_i + q_i$ in $g(y)$, for $i = 1, \ldots, r$ and for all $y \in U$, while all other bits in $g(y)$ are zero. The integers $q_1, \ldots, q_r$ will be chosen to satisfy the following conditions.

(1) $p_1 + q_1 < p_2 + q_2 < \ldots < p_r + q_r$;

(2) $(p_r + q_r) - (p_1 + q_1) \le 2r^5$;

(3) The r^2 sums $p_i + q_j$, where $1 \le i, j \le r$, are all distinct.

Condition (1) ensures that $rank(g(x), g(Y)) = rank(f(x), f(Y))$ and that minimizing $msb(g(x), g(y_j))$ is equivalent to minimizing $msb(f(x), f(y_j))$, for all $x \in U$, so that substituting g for f leaves the algorithm correct. Condition (2) ensures that images under g are still sufficiently small, following a fixed right shift by $p_1 + q_1$ bit positions, to allow constant-time computation of $rank(g(x), g(Y))$, and Condition (3) implies that g can be implemented as a multiplication by $\sum_{i=1}^{r} 2^{q_i}$ followed by the application of a bit mask that clears all bits outside of the positions $p_1 + q_1, \ldots, p_r + q_r$ of interest.

Fredman and Willard described a deterministic procedure for computing $q_1, \ldots, q_r$. We obtain $q_1, \ldots, q_r$ through a randomized but faster procedure that essentially amounts to choosing $q_1, \ldots, q_r$ at random. More precisely, choose $Z_1, \ldots, Z_r$ independently from the uniform distribution over $\{1, \ldots, 2r^4\}$ and take $q_i = \lambda - p_i + 2(i-1)r^4 + Z_i$, for $i = 1, \ldots, r$. It is easy to see that $q_1, \ldots, q_r$ are nonnegative and that Conditions (1) and (2) are satisfied. Condition (3) may be violated, but we can check this in $O(\log d)$ time with d^2 processors. For fixed i, j, k, l with $1 \le i, j, k, l \le r$ and $(i, j) \ne (k, l)$, the condition $p_i + q_j \ne p_k + q_l$ is violated with probability at most $1/(2r^4)$, so that altogether Condition (3) is violated with probability at most $r^4/(2r^4) = 1/2$.