On Parallel Integer Sorting

Sanguthevar Rajasekaran[1]
Dept. of Computer and Information Science
University of Pennsylvania
Sandeep Sen[1]
Dept. of Computer Science, Duke University

Abstract. We present an optimal algorithm for sorting n integers in the range $[1, n^c]$ (for any constant c) for the EREW PRAM model where the word length is n^ϵ, for any $\epsilon > 0$. Using this algorithm, the best known upper bound for integer sorting on the ($O(\log n)$ word length) EREW PRAM model is improved. In addition, a novel parallel range reduction algorithm which results in a near optimal randomized integer sorting algorithm is presented. For the case when the keys are uniformly distributed integers in an arbitrary range, we give an algorithm whose expected running time is optimal.

1 Introduction

1.1 Sequential Sorting Algorithms

The importance of sorting in Computer Science applications cannot be overemphasized. The problem of sorting a sequence of elements (also called *keys*) is to rearrange this sequence in either ascending order or descending order. When the keys to be sorted are *general*, i.e., when the keys have no known structure, a lower bound result [1] states that any sequential algorithm (on the comparison tree and many other sequential models of interest) will require $\Omega(n \log n)$ time to sort a sequence of n keys. Many *optimal* algorithms like QUICK_SORT and HEAP_SORT whose run times match this lower bound can be found in the literature [1, 13].

In computer science applications, very often, the keys to be sorted are from a finite set. In particular, the keys are integers of at most a polynomial (in the input size) magnitude. For keys with this special property, sorting becomes much simpler. The BUCKET_SORT algorithm [1] sorts n *integer*

[1]Supported by NSF-DCR-85-03251 and ONR contract N00014-87-K-0310

keys in $O(n)$ sequential steps. Notice that the run time of BUCKET_SORT matches the trivial $\Omega(n)$ lower bound for this problem. In this paper we are concerned with randomized parallel algorithms for sorting *integer keys*.

1.2 Known Parallel Sorting Algorithms

The performance of a parallel algorithm can be specified by bounds on its principal resources namely, processors and time. If we let P denote the processor bound, and T denote the time bound of a parallel algorithm for a given problem, the product PT is, clearly, bounded from below by the minimum sequential time, T_s, required to solve this problem. We say a parallel algorithm is *optimal* if $PT = O(T_s)$. Optimal parallel sorting for both *general* and *integer keys* remained an open problem for a long time.

Many optimal algorithms (both deterministic and randomized) for sorting *general* keys in $O(\log n)$ time can be found in the literature (see [20],[19],[2], and [7]). As in the sequential case, many parallel applications of interest need only sort *integer keys*. Until recently, no optimal parallel algorithm existed for sorting n *integer keys* with a run time of $O(\log n)$ or less. Rajasekaran and Reif[18] have given a randomized optimal algorithm for sorting n integers in the range $[1, n(\log n)^{O(1)}]$. It remains an open problem to find an optimal algorithm for sorting keys in the range $[1, n^c]$, for any constant c (using small word length). Hagerup [10] has published an algorithm that sorts n integers in the range $[1, n^c]$ in time $O(\log n)$ using $\frac{n \log\log n}{\log n}$ processors. The algorithm uses a stronger model, namely the Priority CRCW PRAM model, and $O(n^{1+\epsilon})$ space, for any $\epsilon > 0$.

1.3 Definitions

Given a sequence of keys $k_1, k_2, \ldots, k_n$ drawn from a set S having a linear order $<$, the problem of *sorting* this sequence is to find a permutation σ such that $k_{\sigma(1)} < k_{\sigma(2)} < \ldots < k_{\sigma(n)}$.

By *general keys* we mean a sequence of n elements drawn from a linearly ordered set S whose elements have no known structure. The only operation that can be used to gain information about the sequence is the comparison of two elements. If each of the n elements in a sequence is an integer in the range $[1, n^c]$ (for any constant c) we call these keys *integer keys*.

GENERAL_SORT is the problem of sorting a sequence of general keys, and *INTEGER_SORT* is the problem of sorting a sequence of integer keys.

Throughout this paper we let $[m]$ denote $\{0, 1, 2, \ldots, m-1\}$. Let $B(n, p)$

stand for a binomial variable with parameters n and p where n is the number of trials and p is the probability of success.

A sorting algorithm is said to be *stable* if equal elements remain in the same relative order in the sorted sequence as they were in originally. In more precise terms, a sorting algorithm is stable if on input $k_1, k_2, \ldots, k_n$, the algorithm outputs a sorting permutation σ of $(1, 2, \ldots, n)$ such that for all $i, j \in [n]$, if $k_i = k_j$ and $i < j$ then $\sigma(i) < \sigma(j)$. A sorting algorithm that is not guaranteed to output a stable sorted sequence is called *non-stable*.

Just like the big-O function serves to represent the complexity bounds of deterministic algorithms, we employ $\widetilde{O}$ to represent complexity bounds of randomized algorithms. We say a randomized algorithm has resource (like time, space, etc.) bound $\widetilde{O}(g(n))$ if there is a constant c such that the amount of resource used by the algorithm (on any input of size n) is no more than $c\alpha g(n)$ with probability $\geq 1 - 1/n^{\alpha}$, for any $\alpha \geq 1$.

1.4 Our Model of Computation

We assume the CRCW PRAM model proposed by Shiloach and Vishkin [21] (except for the algorithm in section 3, where we assume the EREW PRAM). In a PRAM model, a number (say P) of processors work synchronously communicating with each other with the help of a common block of memory. Each processor is a RAM. A single step of a processor is an arithmetic operation, a comparison, or a memory access. The CRCW PRAM is a version of the PRAM that allows both concurrent writes and concurrent reads of shared memory and the EREW PRAM does not allow any concurrent access to the same location. Write conflicts are resolved arbitrarily. Unless otherwise mentioned, PRAMs are assumed to have a word length of $O(\log n)$. For the randomized algorithms, the processors have access to $O(\log n)$ bit random numbers.

1.5 Contents of this Paper

In this paper we characterize the dependence of the run time of an integer sorting algorithm on the machine word length. Specifically, we show that if the word length is sufficiently large, the problem of integer sorting reduces to the problem of *prefix sum computation*. As a corollary we get an integer sorting algorithm that runs in time $O(\log n)$ using $n/\log n$ processors with a word length of n^{ϵ}, for any constant $\epsilon > 0$.

Several non-optimal deterministic algorithms have been given for INTEGER_SORT. For example, Kruskal, Rudolph, and Snir [14] gave an algorithm that runs in time $O(\frac{n}{p}\frac{\log n}{\log(n/p)})$ using p processors. Following this, Hagerup gave an algorithm that runs in $O(\log n)$ time using $\frac{n \log\log n}{\log n}$ processors. This algorithm is very close to being optimal. However, both the above algorithms use $O(n^{1+\epsilon})$ (for any $\epsilon > 0$) space. Since nowadays the cost of processing elements is nearly the same as the cost of memory elements, the space requirements of these algorithms should be viewed with some concern. We address in this paper the question of INTEGER_SORT when the memory usage is restricted to $O(n)$. All the following results assume only $O(n)$ space.

We give a deterministic INTEGER_SORT algorithm that runs in time $(\log n \frac{n \log\log\log n}{p \log\log n})$ using $p \leq \frac{n}{\log\log n}$ processors. We also present a novel parallel range reduction algorithm that results in a randomized algorithm for INTEGER_SORT that uses $\frac{n}{(\log n)^{2\epsilon}}$ processors and runs in time $O((\log n)^{1+\epsilon})$, for any $0 < \epsilon \leq 1/2$.

For the case of uniformly distributed integer keys in an arbitrary range, we give a CRCW PRAM algorithm that is optimal on the average. This algorithm runs in time $\widetilde{O}(\log n)$ using $\frac{n}{\log n}$ processors for all but a small fraction ($\leq 2^{-\Omega(n/(\log n \log\log n))}$) of the inputs. Recently, in independent works, Chlebus[6] and Hagerup [11] have presented optimal algorithms for sorting n random integers in the range $[1, n^{O(1)}]$. Our algorithm runs optimally for a larger fraction of possible inputs, and for each such input the time bound holds with high probability.

All our sorting algorithms are stable.

2 Preliminary Results

Let Σ be a domain and let $\circ$ be an associative operation that takes $O(1)$ sequential time over this domain. The *prefix computation problem* is defined as follows.

- **input** $(X(1), X(2), \ldots, X(n)) \in \Sigma^n$
- **output** $(X(1),\ X(1) \circ X(2),\ \ldots,\ X(1) \circ X(2) \circ \ldots \circ X(n))$.

The special case of prefix computation when Σ is the set of all natural numbers and $\circ$ is integer addition is called *prefix sum computation*. Ladner

and Fischer [15] show that prefix computation can be done by a circuit of depth $O(\log n)$ and size n. The processor bound of this algorithm can be improved as follows. (This is folklore; see for example [18]).

Lemma 2.1 *Prefix computation can be done in time $O(\log n)$ using $n/\log n$ EREW PRAM processors.*

Recently, Cole and Vishkin [9] have proved the following

Lemma 2.2 *Prefix sum computation of n integers ($O(\log n)$ bits each) can be performed in $O(\log n/\log\log n)$ time using $n\log\log n/\log n$ CRCW PRAM processors.*

When the keys to be sorted are in the range $[1, n(\log n)^{O(1)}]$, there is an optimal randomized algorithm for INTEGER_SORT [18]:

Lemma 2.3 *Given n integers in the range $[n(\log n)^{O(1)}]$, they can be sorted in $\widetilde{O}(\log n)$ time using $n/\log n$ CRCW PRAM processors.*

Given a sequence of keys $k_1, k_2, \ldots, k_n$, the problem of *selection* is to find the lth largest element in the sequence for a given l. Cole [8] has shown

Lemma 2.4 *The problem of selection can be solved in $O(\log n)$ time using $n/\log n$ CRCW PRAM processors.*

3 An Optimal INTEGER_SORT Algorithm

In this section we prove that if the word length of a computer is sufficiently large, then there exists an optimal algorithm for INTEGER_SORT. INTEGER_SORT in this case reduces to prefix sum computation. We give an algorithm for stable sorting n keys in the range $[n^\epsilon]$, for any $\epsilon > 0$, and then use the following folklore lemma (see [18]) to extend the range to $[n^c]$ for any constant c. In the remaining paper we shall use the abbreviations LSB and MSB to denote Least Significant Bit and Most Significant Bit respectively.

Lemma 3.1 *If n keys in the range $[R]$ can be stable sorted in $O(\log n)$ time using $n/\log n$ processors, then n keys in the range $[R^{O(1)}]$ can be stable sorted in $O(\log n)$ time using the same number of processors.*

To sort n integers in the range $[n^\epsilon]$, we use a word length of $n^\epsilon \log n$. Each word is thought of as a sequence of n^ϵ blocks $B_{n^\epsilon-1}, B_{n^\epsilon-2}, \ldots, B_0$, where each block B_i is exactly $\log n$ bits long. $P = n/\log n$ processors are used. Each processor $\pi \in [P]$ is assigned the key indices $J(\pi) = \{j | (\pi - 1)\log n < j \leq min(n, \pi \log n)\}$. Details of the algorithm follow.

step1

Each processor $\pi(\pi \in [P])$ generates $\log n$ words $W_{\pi,1}, W_{\pi,2}, \ldots, W_{\pi,\log n}$, one for each key index given to it. If the key has a value k, then it writes a 1 in the LSB of the kth block of the corresponding word.

step2

$P = n/\log n$ processors collectively compute the prefix sum of the n numbers $(W_{1,1}, W_{1,2}, \ldots, W_{P,\log n})$. Let $S_{1,1}, S_{1,2}, \ldots, S_{P,\log n}$ be the prefix sum computed. This prefix sum orders keys with equal values according to their input indices.

step3

Let $S_{P,\log n}$ be $B_{n^\epsilon-1}, B_{n^\epsilon-2}, \ldots, B_0$. Then, notice that B_i is simply the number of keys in the input that have a value i for $0 \leq i \leq n^\epsilon - 1$.
n^ϵ processors in parallel compute the prefix sum of $B_0, B_1, \ldots, B_{n^\epsilon-1}$.

step4

Each processor uses the results of step 2 and step 3 to compute unique ranks for the $\log n$ keys assigned to the processor, and outputs these keys in the right order.

Clearly, all these four steps can be performed in $O(\log n)$ time using $n/\log n$ processors. The correctness of the algorithm is self evident. Lemma 3.1 together with this algorithm yields the following

Theorem 3.1 *There exists an optimal stable algorithm for INTEGER_SORT that runs in time $O(\log n)$ on the EREW PRAM, provided that the word length is n^ϵ (for any $\epsilon > 0$).*

The following corollary is immediate.

Corollary 3.1 *We can sort n numbers in the range $[m]$ in $O(\log n)$ time using $n/\log n$ EREW PRAM processors if the word length is $m^\epsilon \log n$ (for any $\epsilon > 0$).*

4 An Improved INTEGER_SORT Algorithm

The algorithm to be presented stable sorts n integers $k_1, k_2, \ldots, k_n$ in the range $[n]$. In accordance with lemma 3.1, this will also be an algorithm to sort keys in the range $[n^{O(1)}]$ with the same resource bounds. The algorithm has a time bound of $O(\log n \frac{n \log\log\log n}{p \log\log n})$ using p processors, for $p \leq \frac{n}{\log\log n}$. This algorithm uses $O(n)$ space on the $O(\log n)$ word length CRCW PRAM.

Let $T(n, m)$ denote the time needed to sort n integers in the range $[m]$. The problem of sorting n keys in the range $[m]$ can be reduced to two subproblems of sorting n integers in the range $[\sqrt{m}]$, using the idea of radix sorting, as follows.

Let $k'_i = \lfloor k_i/\sqrt{m} \rfloor$ and $k''_i = k_i - k'_i * \sqrt{m}$ for every $i \in [n]$. First, stable sort $k''_1, k''_2, \ldots, k''_n$, obtaining a permutation σ. Then stable sort $k'_{\sigma(1)}, k'_{\sigma(2)}, \ldots, k'_{\sigma(n)}$, obtaining a permutation ρ. Output the given input sequence applying the permutation $\rho \circ \sigma$ on it.

Clearly, the keys k'_i's and k''_i's are in the range $[\sqrt{m}]$. Thus we have the following recurrence relation for $T(n, m)$.

$$T(n, m) \ \leq \ 2T(n, \sqrt{m}) \tag{1}$$

Another recurrence relation for $T(n, m)$ is given by the following algorithm.

step1

Given the keys $k_1, k_2, \ldots, k_n$ in the range $[m]$, partition the keys into $\sqrt{n}$ groups of size $\sqrt{n}$ each. Sort each group in parallel. Let N_{ij} stand for the number of keys with value i in group j. $(i = 1, 2, \ldots, m;\ j = 1, 2, \ldots, \sqrt{n})$.

step2

Compute the prefix sum of

$(N_{11}, N_{12}, \ldots, N_{1\sqrt{n}},$
$N_{21}, N_{22}, \ldots, N_{2\sqrt{n}},$
$\ldots$
$N_{m1}, N_{m2}, \ldots, N_{m\sqrt{n}})$

This prefix sum gives the sorted order of all n elements.

We may thus write the recurrence equation for the running time of this phase as:

$$T(n, m) \ \leq \ T(\sqrt{n}, m) \ + \ R(m\sqrt{n}) \tag{2}$$

where $R(l)$ is the time needed to perform the prefix sum of l integers. Thus the complete algorithm has the following structure.
It has two kinds of recursive calls - one on the word-size and the other on the number of elements. To sort n elements in the range $[m]$, we perform two stable sorting in the range $[\sqrt{m}]$. Each of these sorts is done by partitioning the n elements into groups of $\sqrt{n}$ and combining them using the procedure described in the previous paragraph. No mention is made in equations 1 and 2 about the number of processors used. The processor bound will be computed corresponding to a run time of $O(\log n \log\log\log n)$. ¿From equations 1 and 2 it follows that

$$T(n, n) \ \leq \ 2T(n, \sqrt{n}) \ \ \leq \ 2T(\sqrt{n}, \sqrt{n}) \ + \ 2R(n)$$

If we use [9]'s $O(\log n/(\log\log n))$ time algorithm for prefix sums, then the above equation can be rewritten as

$$T(n, n) \leq 2^i \ T(n^{1/2^i}, n^{1/2^i}) \ + \ \sum_{j=1}^{i} \frac{\log n}{\log\log n - j + 1} \text{ for any } 1 \leq i \leq \log\log n$$

whose solution is $T(n, n) = O(\log n \log\log\log n)$. The number of processors needed to perform the required prefix computations at the ith level $(1 \leq i \leq \log\log n)$ of recursion is $\frac{n}{\log n}(\log\log n - i)2^i$. Also at the ith level of recursion, there are $n^{1-1/2^i}$ sorting subproblems. By unfolding the recurrence for time, we see that it has 2^i additive terms for sorting $n^{1/2^i}$ numbers in the range $[n^{1/2^i}]$. If we let the recurrence run until the subproblems are of size $O(1)$, $\theta(n)$ processors will be needed at the very last step. To get a processor bound of $O(n/(\log\log n))$, the recurrence is stopped when the subproblems are of size $\theta((\log n)^{\log\log\log n})$ which we shall denote by S. S numbers in the range $[\log n]$ can be stable sorted in time $O(\log S)$ using

$S/(\log S)$ processors, in accordance with corollary 3.1. Thus, using this fact and the idea of radix sorting, S numbers in the range $[S]$ can be stable sorted in $O(\log S \log\log\log n)$ time, using $S/(\log S)$ processors.

Therefore, all the existing subproblems in the base case can be solved in time $O(\frac{\log n}{\log S} \log S \log\log\log n) = O(\log n \log\log\log n)$. The number of processors needed is $\frac{n}{\log S} = O(\frac{n}{\log\log n \log\log\log n})$.

Also, corresponding to this base case, the time needed to perform all the recursive steps can easily be seen to be $O(\log n \log\log\log n)$. The number of processors needed to perform the recursive steps is $O(\frac{n}{\log\log n})$. This establishes the stated resource bounds. Thus we get the following

Theorem 4.1 *There is an algorithm for INTEGER_SORT that takes time* $(\log n \frac{n \log\log\log n}{p \log\log n})$ *using* $p(\leq n/(\log\log n))$ *CRCW PRAM processors.*

Note that the same algorithm when implemented on the EREW PRAM will yield a time bound of $O(\log n \log\log n)$ and a PT bound of $O(\frac{n \log n}{\log\log n})$. The boundary condition in this case is when the problem sizes are reduced to $\theta((\log n)^{\log\log n})$. The reason for the slightly superior PT bound is that at the base level of recurrence, the number of processors required to sort the subproblems and merge them are balanced.

5 Parallel Range Reduction

A radix sort algorithm for sorting integers can be thought of as an algorithm that successively reduces the value-range of the keys sorted. In [12], an algorithm is given that reduces in $O(n)$ sequential time the problem of sorting n integers in the range $[2^{2k(n)}]$ to the problem of sorting n integers in the range $[2^{k(n)}]$. But no parallel analogue of this algorithm exists. In this section we present a parallel range reduction algorithm that results in a PT bound of $\widetilde{O}(n\sqrt{\log n})$ for INTEGER_SORT on the CRCW PRAM.

Rajasekaran and Reif's [18] optimal algorithm for INTEGER_SORT uses radix sort starting from the LSB's of the keys. This algorithm consists of two phases. To sort n keys in the range $[n \log n]$, in the first phase keys are sorted with respect to their $(\log n - \log\log n)$ LSBs. In the second phase the resultant sequence is stable sorted with respect to the $2\log\log n$ MSBs of the keys. One of the problems with LSB first radix sorting is that except for the first phase all the other phases should be stable sort algorithms. It seems to be difficult to get a randomized stable sort algorithm. The need for stability can be eliminated if we employ MSB first sorting. But the

problem with MSB first radix sorting is the unequal size of the subproblems that may make the value-range disproportionately larger than the number of keys (say $O(\log n)$ keys in the range $[1, n]$).

Given n integers in the range $[R]$, assume we sort them with respect to their k MSBs. Let $S(i)$ be the set of keys with value i for their k MSBs ($0 \leq i \leq 2^k - 1$). Sorting the given n keys is now reduced to 2^k disjoint subproblems, namely the problems of sorting $S(0), S(1), \ldots, S(2^k - 1)$. The required output is indeed

$$\text{sort}(S(0)) \circ \text{sort}(S(1)) \circ \ldots \circ \text{sort}(S(2^k - 1)),$$

where $\text{sort}(S(i))$ is the sequence of keys in $S(i)$ in sorted order. Thus the problem of sorting integers in the range $[R]$ can be reduced to 2^k disjoint subproblems, after sorting the input keys with respect to their k MSBs. Here each subproblem is that of sorting integers in the range $[\lceil R/2^k \rceil]$.

¿From hereon let 'size' of a subproblem refer to the number of keys in it, and 'range' refer to the interval of values the keys in the subproblem fall in. Let 'range-length' denote the length of this interval. For instance when n keys are sorted with respect to their k MSBs, 2^k subproblems arise. The range-length of each subproblem is $\lceil R/2^k \rceil$. But the sizes of the subproblems can be arbitrary (anywhere from 0 to n). This is precisely the problem with MSB first sorting.

The starting point of this section is the optimal (but unstable) sorting algorithm of Rajasekaran and Reif. Realize that if n keys in the range $[n]$ can be stable sorted in time T (on any PRAM) using P processors, we can also stable sort n keys in the range $[n^c]$, for any constant c, in $O(T)$ time using the same number of processors. Thus we may assume that the input keys are in the range $[n]$ and are already sorted according to their values but may be unordered according to their input position (among keys having same values). Our objective is to produce a stable sorted output. We first present an algorithm that uses $n/\log n$ processors and runs in time $(\log n)^{1.5}$. Later we modify this algorithm to use $\frac{n}{(\log n)^{2\epsilon}}$ processors. The run time of the modified algorithm will be $(\log n)^{1+\epsilon}$.

Our first algorithm runs in $O(\sqrt{\log n})$ phases. Each phase takes $\widetilde{O}(\log n)$ time using $n/\log n$ processors. To begin with the n keys in the range $[n]$ are sorted using the non-stable sorting algorithm of [18]. Call this as the preprocessing step. At the end of this step, we have an ordered sequence of at the most n nonempty subproblems (one corresponding to each value in the range $[n]$). The range-length of each subproblem is n (because keys in any

subproblem have to be sorted with respect to their input positions). In each phase thereafter, every subproblem is replaced with an ordered sequence of new subproblems, such that the range-length of each new subproblem is no more than $\frac{1}{l}$ times the range-length of the original subproblem, l being the number of keys in the original subproblem. If during any phase, the size of some subproblem becomes $O(1)$, or its range-length becomes $O(1)$, keys in this subproblem are sorted easily. As will be shown, after $O(\sqrt{\log n})$ phases every subproblem will have either $O(1)$ size or $O(1)$ range-length.

We shall give a verbal description of the idea and follow it up with a more formal description. Consider a subproblem S at the beginning of phase i $(1 \leq i \leq O(\sqrt{\log n}))$. Let X and Y be the keys in S of maximum and minimum values respectively. (Here key values refer to the positions of the corresponding keys in the input, and hence these are integers in the range $[1, n]$.) Also let l be the number of keys in S. In phase i, S will be replaced with an ordered sequence of l subproblems $S(0), S(1), \ldots, S(l-1)$, where $S(i), 0 \leq i < l$ is the collection of keys in S whose values fall in the interval $\left[Y + i\left\lceil\frac{X-Y}{l}\right\rceil , Y + (i+1)\left\lceil\frac{X-Y}{l}\right\rceil - 1\right]$. Realize that some of the $S(i)$'s could be empty. But in any phase, the total number of subproblems will not exceed n.

We process each of the subproblems existing at the beginning of phase i in the manner described above. This completes phase i. We proceed to phase $i+1$. The algorithm terminates when each of the remaining subproblems either has constant size or constant range-length. Notice that if a subproblem S at the beginning of phase i has range length R, all the new subproblems arising out of S at the end of phase i will have range-length no more than $\frac{R}{l}$ (l being the number of keys in S).

In any phase i, in order to process all the subproblems in parallel, we associate with each key a number in the range $[n]$ and make use of the integer sorting algorithm of [18]. The number associated with each key will preserve the ordering of subproblems, i.e., keys in the same subproblem will get the same number and the subproblems themselves will be ordered. Details follow.

step1

while $\exists$ a subproblem such that (its size is $> O(1)$ and its range is $> O(1)$) *do*

for each subproblem S in parallel do

Find the maximum (call it X) and the minimum (call it Y) of the keys in S.
Let $|S| = l$.
Associate with each key k in S a value of $rank(Y) + \left\lceil \frac{(k-Y)}{(X-Y)/l} \right\rceil$;

Sort all the n keys with respect to their associated values.

step2

Sort all the subproblems in parallel and output them in the right order.

In the above algorithm, *rank*(Y) stands for the true rank of Y from among the n input keys.

Theorem 5.1 *The above algorithm runs in time $\widetilde{O}((\log n)^{1.5})$ using $n/\log n$ CRCW PRAM processors.*

Proof First we will show that the while loop of step1 is executed $O(\sqrt{\log n})$ times. Then we will prove that one iteration of the while loop takes time $\widetilde{O}(\log n)$ using $n/\log n$ processors.

Consider a subproblem S at iteration i of the while loop. If the range-length of S is R at the beginning of this iteration (or phase), then at the end of this phase the range-length will be $\leq R/|S|$. If S is any subproblem after $O(\sqrt{\log n})$ phases, then the claim is that either $|S| = O(1)$ or the range-length of S is $O(1)$.

Notice that at every iteration, the maximum subproblem size is at most half the maximum subproblem size in the previous iteration. Therefore, if at some iteration, a subproblem S has $2^{\sqrt{\log n}}$ keys in it, then after at most $\sqrt{\log n}$ further phases all the subproblems arising out of S will become $O(1)$ in size. Also, if there is a subproblem S of size $> 2^{\sqrt{\log n}}$ after $O(\sqrt{\log n})$ phases, it means that the keys will be in the range $\frac{n}{l_1 l_2 \ldots l_{O(\sqrt{\log n})}} = O(1)$ since each l_m is $> 2^{\sqrt{\log n}}$. That is, if the range-length has to be greater than a constant after $\theta(\sqrt{\log n})$ phases, then there can not be more than $2^{\sqrt{\log n}}$ keys in any subproblem. Thus if S is any subproblem

after $\theta(\sqrt{\log n})$ stages, then either $|S| = O(1)$ or the range-length of S is $O(1)$.

It remains to show that each iteration of the while loop takes $\widetilde{O}(\log n)$ time using $n/\log n$ processors. The final sorting step can be performed within the stated bounds [18]. Let $S_1, S_2, \ldots, S_h$ be all the subproblems of size and range $> O(1)$ in a given iteration. The operations performed on the S_i's are finding MAX and MIN of each S_i. Group the subproblems with $\log n$ subproblems in each group. If M is the number of keys in a group G, allocate $M/\log n$ processors to G. Now all the operations on the S_i's can be performed within the stated resource bounds (see lemmas 2.2, 2.3, and 2.4). All the steps in each iteration are deterministic except for integer-sorting in the range $[1, n]$. The probability of success is not dependent on the subproblem sizes since the integer-sorting algorithm is applied to the n keys (and not separately on the subproblems).□

5.1 Modified Algorithm

The only modification needed is the following. We choose a random sample of $2^{(\log n)^{1-2\epsilon}}$ keys, and use these keys as the splitter keys to partition the sorting problem into $2^{(\log n)^{1-2\epsilon}}$ parts. The random splitters partition the problem into roughly equal-sized subproblems (with high probability). This rough partitioning suffices to prove the resource bounds. The following fact is crucial to the algorithm.

Let $X = k_1, k_2, \ldots, k_n$ be a given sequence of n keys. Sample $m(<< n)$ keys at random from X, with replacement. Let $l_1, l_2, \ldots, l_m$ be the sampled keys in sorted order. These *splitter keys* partition X into $(m+1)$ parts $X_i, 0 \leq i \leq m$, where $X_0 = \{k : \ k \in X \text{and } k \leq l_1\}$, $X_i = \{k : \ k \in X \text{and } l_i < k \leq l_{i+1}\}$ for $1 \leq i < m$, and $X_m = \{k : \ k \in X \text{and } k > l_m\}$. It has been shown ([20], lemma 5) that there exists a constant c such that for each $0 \leq i \leq m$, $|X_i| \leq c\alpha\frac{n}{m}\log n$ with high probability (i.e., with a probability of $\geq 1 - n^{-\alpha}$).

The algorithm follows.

step1

while ∃ a subproblem such that (its size is $> O(1)$ and its range is $> O(1)$) *do*
begin
for each subproblem S of size $> 2^{(\log n)^{1-2\epsilon}}$ *do*

In parallel choose a random sample of keys of size $2^{(\log n)^{1-2\epsilon}} = q$, say, and partition the problem S into $(q+1)$ subproblems $S_0, S_1, \ldots, S_q$ as described above.
Associate values with all the n keys in parallel (each key gets a value equal to the sum of the rank of its lower boundary and number of the subproblem it belongs to).

for each subproblem S of size and range $> O(1)$ *do*

Find the maximum (call it X) and the minimum (call it Y) of the keys in S.
Let $|S| = l$.
Divide the range $X - Y$ into l equal parts.
i.e., Associate with each key k in S a value of $rank(Y) + \left\lceil \frac{(k-Y)}{(X-Y)/l} \right\rceil$

Sort all the n keys with respect to their associated values.

end;

step2

Sort all the subproblems and output them in the right order.

Theorem 5.2 *The above algorithm runs in time* $\widetilde{O}((\log n)^{1+\epsilon})$ *using* $\frac{n}{(\log n)^{2\epsilon}}$ *CRCW PRAM processors, for any* $0 < \epsilon \leq 1/2$.

Proof First we'll show that the while loop of step 1 is executed $O((\log n)^{\epsilon})$ times. Then we'll prove that one iteration of the while loop takes time $\widetilde{O}(\log n)$ using $\frac{n}{(\log n)^{2\epsilon}}$ processors.

Consider a subproblem S at iteration t of the while loop. If the range-length of S is $[R]$ at the beginning of this iteration, then at the end of this iteration it will be $\leq R/|S|$. If S is any subproblem after $O((\log n)^{\epsilon})$ phases, then the claim is either $|S| = O(1)$ or the range-length of S is $O(1)$.

Notice that at every iteration, the maximum subproblem size is $O(\frac{\log n}{2^{(\log n)^{1-2\epsilon}}})$ times the maximum subproblem size in the previous iteration with high probability. Therefore, if at some iteration, a subproblem S has $2^{(\log n)^{1-\epsilon}}$ keys in it, then after at most $O((\log n)^{\epsilon})$ further phases all the subproblems arising out of S will become $O(1)$ in size. Also, if there is a subproblem S of size $> 2^{(\log n)^{1-\epsilon}}$ after $O((\log n)^{\epsilon})$ phases, it means that the range-length is $\frac{n}{l_1 l_2 ... l_{O((\log n)^{\epsilon})}} = O(1)$ since each l_m is $> 2^{(\log n)^{1-\epsilon}}$. That is, if the range-length were to be $> O(1)$ after $O((\log n)^{\epsilon})$ phases, then there cannot be more than $2^{(\log n)^{1-\epsilon}}$ keys in any subproblem. Thus the claim follows.

It remains to show that each iteration of the while loop takes $O(\log n)$ time using $n/(\log n)^{2\epsilon}$ processors. For each problem S of size $> 2^{(\log n)^{1-2\epsilon}}$ that exists at the beginning of the first *for* loop in any iteration, we have to choose random splitters and partition the problem. This can be done in time $O((\log n)^{1-2\epsilon})$ using n processors (see [18]) as follows: 1) sort the splitter keys using Cole's general sorting algorithm, 2) assign a single processor to each key. Each processor in parallel does a binary search on the appropriate splitter sequence, 3) after having found the subproblem that each key belongs to, all the n keys are sorted according to their subproblem numbers using the optimal integer sort algorithm. Equivalently (by slowing down) this partitioning can also be done in $O(\log n)$ time using $\frac{n}{(\log n)^{2\epsilon}}$ processors.

The second *for* loop can be performed within the stated resource bounds as proved in the previous section $\Box$.

6 The Case of Uniformly Distributed Keys

In this section we assume that the keys to be sorted are uniformly distributed integers in an arbitrary range. This is not an unreasonable assumption in many applications (see [13](pp. 170-178), [16], and [17](pp. 84-85)). In [16], a sequential algorithm is given for this sorting problem; it is stable and runs in time $O(n)$ on the average (and hence is optimal). The key idea behind [16]'s algorithm is to sort the keys with respect to some number of their MSBs. This very nearly sorts the file. In particular, this sorting partitions the given input into an ordered sequence of subproblems (just like in the previous section) one for each possible value of the MSBs. With high probability each one of these subproblems will be of 'small' size and hence any of the General Sorting algorithms can then be used to sort each subproblem independently.

Given the keys $k_1, k_2, \ldots, k_n$, let $S(i)$ stand for the set of keys with value i $(0 \leq i \leq (n \log n - 1))$ for their $\lceil (\log n + \log\log n) \rceil$ MSBs. Then, if we sort these keys with respect to their $\lceil (\log n + \log\log n) \rceil$ MSBs, the required output will be

$$\text{sort}(S(0)) \circ \text{sort}(S(1)) \circ \ldots \circ \text{sort}(S(n \log n - 1)),$$

as was discussed in section 5. The total time needed is $O(n) + \sum_{i=0}^{n-1} |S(i)| \log(|S(i)|)$. The expected value of the second term can be shown to be $O(n)$ (see e.g., [17](pp. 84-85)). Hence this is an algorithm that is optimal on the average. Next we show how to modify this algorithm to get a run time of $\widetilde{O}(\log n)$ using $\frac{n}{\log n}$ processors for all but a small fraction $(\leq 2^{-\Omega(n/(\log n \log\log n))})$ of the inputs. That is, our algorithm works for a large fraction of inputs, and for each such input the time bound holds with high probability $(\geq 1 - n^{-\alpha}$, for any $\alpha > 1)$.

step1.

Sort the keys with respect to their $\lceil (\log n + \log\log n) \rceil$ MSBs.

step2.

Each processor $\pi \in [n/\log n]$ gets $\log n$ successive keys from the sorted sequence of step 1.

In $O(\log n)$ time $n/\log n$ processors collectively determine all the subproblems (call them $S'_1, S'_2, \ldots, S'_l$) of size $> C$ (where C is a constant).

step3.

Do a prefix computation and place the subproblems $S'_1, S'_2, \ldots, S'_l$ in successive cells of the memory. Let N be the total number of keys in these l subproblems.

step4.

Sort the N keys of step 3 using any of the parallel GENERAL_SORT algorithms (such as [7]).

step5.

Sort every subproblem of size $\leq C$ independently using a total of $\frac{n}{\log n}$ processors.

step6.

The relative rank of each key in its subproblem is obtained from two prefix sum computations (one for keys in subproblems of size $> C$ and one for keys in subproblems of size $\leq C$) and the two sorted sequences of step 4 and step 5 are then merged using $n/\log n$ processors in $\log n$ time.

Analysis. We claim that N is at most $\widetilde{O}(n/\log n)$. Given this, all the above six steps can be performed within the given resource bounds. Steps 1,2,3,4, and 6 take $\widetilde{O}(\log n)$ time using $n/\log n$ processors. In step 5, a subproblem of size m needs $m \log m$ operations to sort it. Since each m is $\leq C$, a constant, the total number of operations needed to sort all the subproblems individually is $\leq n$. We group these subproblems (using prefix sums) with $\log n$ subproblems in each group and sort these groups. Then step 5 can be performed in time $O(\log n)$ using $\frac{n}{\log n}$ processors.

It remains to show that N is $\widetilde{O}(n/\log n)$. Since we are sorting the given n keys with respect to their $\lceil \log n + \log\log n \rceil$ MSBs, and the key values are uniformly distributed, it is as though we are throwing n keys into $n \log n$ subproblems at random. The probability that any subproblem (i.e., any $S(i)$) contains more than C keys is $O(1/\log n)^C$, using the Chernoff bounds (appendix A, equation 3). In particular, this probability is $\leq \frac{1}{\log^2 n \log\log n}$. Thus the number ($l$) of subproblems with $> C$ keys is upper bounded by the binomial variable $B(n \log n, \frac{1}{\log^2 n \log\log n})$. Using Chernoff bounds (appendix A, equation 4), l is $\leq (1+\epsilon)n/(\log n \log\log n)$ with probability $\geq 1 - 2^{-\epsilon^2 n/(3 \log n \log\log n)} = 1 - P = 1 - 2^{-\Omega(n/(\log n \log\log n)}$.

Consider **any** $n/(\log n \log\log n)$ subproblems from out of the total $n \log n$ subproblems. We will show that the total number of keys falling into these $n/(\log n \log\log n)$ subproblems will not exceed $n/\log n$ with the same probability $1 - P$. Fix some $n/(\log n \log\log n)$ subproblems. Call this collection of subproblems Q. The Probability that any particular key (from out of the n keys) falls into Q is $= 1/(\log^2 n \log\log n)$. Therefore, the expected number of keys falling into Q is $n/(\log^2 n \log\log n)$. Using equation 3 (appendix A), the probability that more than $n/\log n$ keys fall into Q is $2^{-\Omega(n \log\log n/\log n)}$.

The number of choices for Q equals

$$\binom{n \log n}{n/(\log n \log\log n)} = 2^{O(n/\log n)}$$

Putting together, the probability that any $n/(\log n \log\log n)$ subproblems have more than $n/\log n$ keys is $\leq P$.□

Note that the above sorting algorithm is stable since in step 4, the GENERAL_SORT algorithm used is stable, and in step 5, subproblems with $\leq C$ keys are individually stable sorted. Also, stability is not lost in the merging performed in step 6. Another way of proving the stability of the output is to append each key with its input index (which is a $\lceil \log n \rceil$ bit integer) and sort the resultant keys.

Theorem 6.1 *There exists an optimal randomized algorithm on the CRCW PRAM for sorting n uniformly distributed integers in an arbitrary range. This algorithm uses $n/\log n$ processors and $\widetilde{O}(\log n)$ time. The probability of success is $1 - 2^{-\Omega(n/(\log n \log\log n))}$.*

7 Open Problems

It is an open problem to find an optimal deterministic or randomized algorithm for sorting integers in the range $[n^c]$ for any constant c on the $O(\log n)$ word length PRAM models. This would lead to improvements and/or simplification of a number of parallel algorithms for some fundamental problems in graphs and geometry.

Postscript

We recently learned that Bhatt, Diks, Hagerup, Prasad, Radzik and Saxena [4] have been able to adapt Hagerup's algorithm to the arbitrary CRCW model. However, the space bound is still $O(n^{1+\epsilon})$.

Acknowledgement

We are very grateful to the anonymous referees whose meticulous comments helped in polishing up the paper significantly. In particular, the referees had pointed out errors in the bounds of Theorem 4.1 in an earlier version of the manuscript.

References

[1] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] Ajtai, M., Komlós, J., and Szemerédi, E., "An $O(n \log n)$ Sorting Network," Combinatorica 3, 1983, pp. 1-19.

[3] Angluin, D., and Valiant, L.G., "Fast Probabilistic Algorithms for Hamiltonian Paths and Matchings," Journal of Computer and Systems Science, 18, 1979, pp. 155-193.

[4] Bhatt P., Diks K., Hagerup T., Prasad V., Radzik T. and Saxena S, "Improved Deterministic Parallel Integer Sorting," Unpublished Manuscript, 1989.

[5] Chernoff, H., "A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations," Annals of Math. Statistics 23, 1952, pp. 493-507.

[6] Chlebus, H., "Parallel Iterated Bucket Sort," Information Processing Letters 31, 1989, pp. 181-183.

[7] Cole, R., "Parallel Merge Sort," SIAM Journal on Computing, vol. 17, no.4, 1988, pp. 770-785.

[8] Cole, R., "An optimally efficient selection algorithm", Information Processing Letters, 26, January 88, pp. 295-299.

[9] Cole, R., and Vishkin, U., "Faster Optimal Parallel Prefix Sums and List Ranking," Information and Computation 81, 1989, pp. 334-352.

[10] Hagerup, T., "Towards Optimal Parallel Bucket Sorting," Information and Computation 75, 1987, pp. 39-51.

[11] Hagerup, T., "Hybridsort Revisited and Parallelized," Information Processing Letters 32, 1989, pp. 35-39.

[12] Kirkpatrick, D., and Reisch, S., "Upper Bounds for Sorting Integers on Random Access Machines," Theoretical Computer Science, vol. 28, 1984, pp. 263-276.

[13] Knuth, D.E., *The Art of Computer Programming. Vol.3: Sorting and Searching*, Addison-Wesley Publishing Co., 1973.

[14] Kruskal, C., Rudolph, L., and Snir, M., "Efficient Parallel Algorithms for Graph Problems," Algorithmica 5, 1990, pp. 43-64.

[15] Ladner, R.E., and Fischer, M.J., "Parallel Prefix Computation," Journal of the ACM, 27(4), 1980, pp. 831-838.

[16] MacLaren, M.D., "Internal Sorting by Radix Plus Sifting," JACM, vol.13, no.3, 1966, pp. 404-411.

[17] Mehlhorn, K., *Data Structures and Algorithms 1: Sorting and Searching,* Springer-Verlag Publications, 1984.

[18] Rajasekaran, S., and Reif, J.H., "Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms," SIAM Journal on Computing, vol. 18, no.3, 1989, pp. 594-607.

[19] Reif, J.H., and Valiant, L., "A Logarithmic Time Sort for Linear Size Networks," JACM, vol.34, no.1, 1987, pp. 60-76.

[20] Reischuk, R., "Probabilistic Parallel Algorithms for Sorting and Selection," SIAM Journal on Computing, vol. 14, no. 2, 1985, pp. 396-409.

[21] Shiloach, Y., and Vishkin, U., "Finding the Maximum, Merging, and Sorting in a Parallel Computation Model," Journal of Algorithms 2, 1981, pp. 81-102.

Appendix A: Probabilistic Bounds

A *binomial variable* X with parameters (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p. Chernoff [5] and Angluin and Valiant [3] have shown that the tail ends of a binomial distribution can be approximated as follows.

Lemma A.1 *If X is binomial with parameters (n, p), and $m > np$ is an integer, then*

$$Probability(X \geq m) \ \leq \ \left(\frac{np}{m}\right)^m \ e^{m-np}. \tag{3}$$

Also,

$$Probability(X \geq \lceil (1+\epsilon)np \rceil) \ \leq \ exp(-\epsilon^2 np/3) \tag{4}$$

for all $0 < \epsilon < 1$.