

An Optimal Parallel Algorithm for Sorting Multisets¹

Sanguthevar Rajasekaran

Dept. of CISE, Univ. of Florida, Gainesville, FL 32611.

Abstract. We consider the problem of sorting n numbers that contain only k distinct values. We present a randomized arbitrary CRCW PRAM algorithm that runs in $O(\log n)$ time using $\frac{n \log k}{\log n}$ processors. The same algorithm runs in $O\left(\frac{\log n}{\log \log n}\right)$ time with a total work of $O(n(\log k)^{1+\epsilon})$ for any fixed $\epsilon > 0$. All the stated bounds hold with high probability.

Keywords: multiset sorting, randomized algorithms, arbitrary CRCW PRAM

1 Introduction

$\Omega(n \log n)$ is a well-known lower bound on the work required for sorting n general keys. When additional information about the keys to be sorted is available, sorting can be done with less work. For instance, sorting n keys where each key is an integer in the range $[1, n^{O(1)}]$ can be accomplished in $O(n)$ time using radix sort.

A lower bound on the time for sorting n numbers with k distinct keys, $k < n$, is $\Omega(n \log k)$, and algorithms with this sequential running time exist.

Recently, Farach and Muthukrishnan [3] looked at the related problem of *renaming* the keys. Here the input is an array a of n keys. The output is an array b of integers in the range $[1, k]$. Also, if $a[i] = a[j]$, for any $1 \leq i, j \leq n$, then $b[i] = b[j]$. They presented a randomized CRCW PRAM algorithm that runs in $O(\log k)$ time and does $O(n \log k)$ work with high probability. Note that if the keys can be sorted, then the renaming problem can be solved trivially.

¹This research was supported in part by an NSF Award CCR-95-03-007 and an EPA Grant R-825-293-01-0.

In this paper we present a randomized algorithm for sorting an array of n numbers given that there are only $k < n$ distinct values, where k need not be given as a part of the input.

2 Preliminaries

The amount of resource (time, space, etc.) used by any randomized algorithm is said to be $\tilde{O}(f(n))$ if the amount used is no more than $cf(n)$ with probability $\geq (1 - n^{-\alpha})$, where c is some positive constant. Let $B(n, p)$ denote a binomial random variable with parameters n and p . If X is a random variable with a distribution of $B(n, p)$, then Chernoff bounds can be used to get tight upper bounds on the tail ends of X . In particular,

$$\text{Prob.}[X \geq (1 + \epsilon)np] \leq n^{-\epsilon^2 np/2}.$$

Also,

$$\text{Prob.}[X \leq (1 - \epsilon)np] \leq n^{-\epsilon^2 np/3},$$

for any fixed $0 < \epsilon < 1$.

3 The Algorithm

Our algorithm is based on random sampling. Pick a random sample of size $\frac{n}{\log^2 n}$ and sort it using any general sorting algorithm. This allows k to be estimated. If $k = \Omega(\sqrt{n})$, sort the whole input, since then the work done will be $O(n \log k)$. Otherwise, collect all the distinct keys and sort them. Perform a binary search for each input key so that each key is assigned a label in the range $[1, k]$, depending on its value. Finally, sort the keys with respect to the assigned labels using the algorithm of Rajasekaran and Reif [6]. More details follow. Let k_1, k_2, \dots, k_n be the input sequence. The number of processors used is $P = \frac{n \log k}{\log n}$.

Algorithm MultisetSort

Step 1. Assign each processor $\frac{n}{P}$ keys from the input. Every input key is independently and randomly chosen to be in sample S with probability $\frac{1}{\log^2 n}$.

Step 2. Collect the sample in successive cells of common memory using a prefix computation and sort S . Let S' be the sorted sample.

Step 3. Perform a prefix computation on S' to form a sequence Q of distinct values in S , i.e., exactly one key per value is retained in Q . Note that $|Q| < k$ is possible. If $|Q| > \sqrt{n}$, sort the input using any general sorting algorithm and terminate.

Step 4. For each input key, perform a binary search in Q .

Step 5. Collect the input keys whose values are not represented in Q using a prefix computation. Let R be this collection.

Step 6. Sort Q and R together. Perform a prefix computation and keep only one key per value. Let U be the resultant sequence.

Step 7. Perform a binary search for every input key in U and assign a label to this key in the range $[1, k]$. A key k_i with a value equal to the j th smallest value in the input is assigned label j .

Step 8. Sort the input keys with respect to the labels assigned in Step 7. The resultant sequence is the desired output.

Theorem 3.1 *Algorithm MultisetSort runs in time $\tilde{O}(\log n)$ using $\frac{n \log k}{\log n}$ CRCW PRAM processors and solves the multiset sorting problem.*

Proof. The correctness of the algorithm is evident.

Step 1 takes $\frac{\log n}{\log k}$ time. The number of samples in S has a distribution of $B\left(n, \frac{1}{\log^2 n}\right)$, so the cardinality of S is $\tilde{O}\left(\frac{n}{\log^2 n}\right)$.

Prefix computation in Step 2 can be performed in $O(\log n)$ time, the total work done being $O(n)$. Sorting takes $\tilde{O}(\log n)$ time using $\frac{n}{\log^2 n}$ processors using Cole's [1] parallel merge sort algorithm.

Step 3 takes $\tilde{O}(\log n)$ time using $\frac{n}{\log^3 n}$ processors.

Since $|Q| \leq k$, Step 4 can be completed in $O(\log k)$ time using n processors. Equivalently, it can be done in $O(\log n)$ time, with total work $O(n \log k)$.

Step 5 takes $O(\log n)$ time using $O\left(\frac{n}{\log n}\right)$ processors.

If a value is represented m times in the input, then the expected number of occurrences of this value in S is $\frac{m}{\log^2 n}$. If $m \geq 5\alpha \log^3 n$, then with probability $\geq (1 - n^{-16\alpha/15})$, there are at least $\log n$ copies of this value in S (for any fixed $\alpha \geq 1$). In other words, if a value is not represented in S , then with high probability the number of occurrences of this value in the input is $\tilde{O}(\log^3 n)$. This implies that the cardinality of R is $\tilde{O}(k \log^3 n)$.

Assume that there are more than $N = \sqrt{n} \log^3 n$ distinct values in the input. Let q_1, q_2, \dots, q_N be any N keys of the input with distinct values. Then, from among these keys, we expect $\sqrt{n} \log n$ of them to be in S . That is, the cardinality of Q is $\tilde{\Omega}(\sqrt{n} \log n)$. Therefore, if $|Q| \leq \sqrt{n}$, k is $\tilde{O}(\sqrt{n} \log^3 n)$.

As a consequence, Step 6 can be completed in $\tilde{O}(\log n)$ time using $\frac{n}{\log n}$ processors, since $|Q| + |R| = \tilde{O}(\sqrt{n} \log^6 n)$.

Step 7 takes $O(\log n)$ time, with total work $O(n \log k)$.

Finally, Step 8 takes $\tilde{O}(\log n)$ time using $\frac{n}{\log n}$ processors. The algorithm of [6] can sort n integers in the range $[1, n(\log n)^{O(1)}]$ in $\tilde{O}(\log n)$ time using $\frac{n}{\log n}$ arbitrary CRCW PRAM processors. \square

4 Sub-Logarithmic Time Sorting

Multiset sorting can be done in $\tilde{O}\left(\frac{\log n}{\log \log n}\right)$ time, with total work $\tilde{O}(n(\log k)^{1+\epsilon})$, for any fixed $\epsilon > 0$.

Since $\Omega(\log n / \log \log n)$ is a lower bound on the parallel time needed to sort n bits (given only a polynomial number of processors), the time bound is the best possible.

The sub-logarithmic time algorithm is the same as `MultisetSort`, with some modifications.

Theorem 4.1 *n keys with k distinct values can be sorted in $O\left(\frac{\log n}{\log \log n}\right)$ time, with total work $\tilde{O}(n(\log k)^{1+\epsilon})$, for any fixed $\epsilon > 0$.*

Proof. Use $P = n(\log k)^{1+\epsilon}$ processors, for any fixed $\epsilon > 0$.

In Step 1, use $\frac{n \log \log n}{\log n}$ processors to pick the sample S in $\frac{\log n}{\log \log n}$ time.

In Step 2, sort sample S using the general sorting algorithm given in [6]. This algorithm sorts N keys in $\tilde{O}\left(\frac{\log N}{\log \log N}\right)$ time, with total work $\tilde{O}(N(\log N)^{1+\epsilon})$ for any constant $\epsilon > 0$. Thus Step 2 can be completed in $\tilde{O}\left(\frac{\log n}{\log \log n}\right)$ time using the given processors. The same bounds hold for Step 6 as well.

In Step 3, if $|Q| > \sqrt{n}$, sort the input keys using the general sorting algorithm of [6]. The work done is optimal.

Prefix computations in Steps 2, 3, 5, and 6 can be done in $O\left(\frac{\log n}{\log \log n}\right)$ time using $\frac{n \log \log n}{\log n}$ processors using the algorithm of Cole and Vishkin [2], since the sequences operated on in these steps are binary.

In Steps 4 and 7, assign $(\log k)^\epsilon$ processors to each key and perform a $(\log k)^\epsilon$ -ary search. The search takes $O\left(\frac{\log k}{\log \log k}\right)$ time, with total work $O(n(\log k)^{1+\epsilon})$.

Step 8 requires a sub-logarithmic time integer sorting algorithm. An algorithm for sorting N integers in the range $[1, N(\log N)^{O(1)}]$ in $\tilde{O}\left(\frac{\log N}{\log \log N}\right)$ time, with total work $\tilde{O}(N \log \log N)$, was given in [6]. The total work done in this algorithm was later improved to $\tilde{O}(N)$ independently by Hagerup [4], Matias and Vishkin [5], and Raman [7]. Thus, Step 8 can also be completed within the stated resource bounds. \square

Acknowledgement

The author thanks David Gries for his critical comments.

References

- [1] R. Cole. Parallel merge sort. *SIAM J. Computing* 17, 4(1988), 770-785.
- [2] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation* 81(1989), 334-352.
- [3] M. Farach and S. Muthukrishnan. Optimal parallel randomized renaming. *IPL* 61(1)(1997), 7-10.
- [4] T. Hagerup. Constant-time parallel integer sorting. *Proc. ACM Symposium on Theory of Computing*, 1991, 299-306.
- [5] Y. Matias and U. Vishkin. Converting high probability into nearly-constant time – with applications to parallel hashing. *Proc. ACM Symposium on Theory of Computing*, 1991, 307-316.
- [6] S. Rajasekaran and J.H. Reif. Optimal and sub-logarithmic time randomized parallel sorting algorithms. *SIAM J. Computing*, 18(3)(1989), 594-607.
- [7] R. Raman. The power of collision: randomized parallel algorithms for chaining and integer sorting. TR 336, Dept. of Computer Science, University of Rochester, January 1991.