# An Optimal Parallel Algorithm for Sorting Multisets[1]

Sanguthevar Rajasekaran

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

**Abstract**. In this paper we consider the problem of sorting $n$ numbers such that there are 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 algorithm is clearly optimal. 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

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

Another interesting case of sorting is when the number of distinct keys is $k < n$. A lower bound of $\Omega(n \log k)$ on the work is easy to derive. An algorithm with a sequential run time of $O(n \log k)$ is also straight forward.

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 $b[\ ]$ such that the entries in $b[\ ]$ are 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

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

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.

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

## 2 Some Preliminaries

The amount of resource (like time, space, etc.) used by any randomized algorithm is said to be $\widetilde{O}(f(n))$ if the amount used is no more than $c\alpha f(n)$ with probability $\geq (1-n^{-\alpha})$, where $c$ is some 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,

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

Also,

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

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

## 3 The Algorithm

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

**Algorithm** MultisetSort

**Step 1**. Each processor is assigned $\frac{n}{P}$ keys from the input. Every input key is independently and randomly chosen to be in the 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 in $S'$ to form a sequence $Q$ of distinct values in $S$, i.e., if $S$ has more than one key of the same value then only one key with this value is retained in $Q$. Note that $|Q|$ can possibly be less than $k$. If $|Q| > \sqrt{n}$, sort the input using any general sorting algorithm, output and quit.

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

**Step 5**. Those input keys whose values are not represented in $Q$ are collected 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 of each value. Let $U$ be the resultant sequence.

**Step 7**. Perfrom a binary search for every input key in $U$ and assign a label to this key in the range $[1, k]$. If a key $k_i$ has a value equal to the $j$th smallest value in the input then it gets a label of $j$.

**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* $\widetilde{O}(\log n)$ *using* $\frac{n \log k}{\log n}$ *CRCW PRAM processors and solves the multiset sorting problem.*

**Proof**. The correctness of the algorithm is quite evident.

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

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

Step 3 takes $\widetilde{O}(\log n)$ 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. Or equivalently, it can be done in $O(\log n)$ time the total work done being $O(n \log k)$.

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 will be 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 $\widetilde{O}(\log^3 n)$. This implies that the cardinality of $R$ is $\widetilde{O}(k \log^3 n)$.

Assume that there are more than $N = \sqrt{n} \log^3 n$ distinct values in the input. Let $q_1, q_2, \ldots, q_N$ 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 $Q$ will be $\widetilde{\Omega}(\sqrt{n} \log n)$. Therefore, if $|Q| \leq \sqrt{n}$, the value of $k$ has to be $\widetilde{O}(\sqrt{n} \log^3 n)$.

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

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

Finally, Step 8 takes $\widetilde{O}(\log n)$ 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 $\widetilde{O}(\log n)$ time using $\frac{n}{\log n}$ arbitrary CRCW PRAM processors. □

# 4 Sub-Logarithmic Time Sorting

In this section we show that multiset sorting can be done in $\widetilde{O}\left(\frac{\log n}{\log \log n}\right)$ time the total work done being $\widetilde{O}(n(\log k)^{1+\epsilon})$, 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** *We can sort $n$ keys with $k$ distinct values in $O\left(\frac{\log n}{\log\log n}\right)$ time with a total work of $\widetilde{O}(n(\log k)^{1+\epsilon})$, for any fixed $\epsilon > 0$.*

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

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

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

In Step 3, if $|Q| > \sqrt{n}$, the input keys can be sorted using the general sorting algorithm of [6]. The work done will be 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 we assign $(\log k)^{\epsilon}$ processors to each key and perform a $(\log k)^{\epsilon}$-ary search. Thus the search takes $O\left(\frac{\log k}{\log\log k}\right)$ time the total work done being $O(n(\log k)^{1+\epsilon})$.

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

# References


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

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

[3] M. Farach and S. Muthukrishnan, Optimal Parallel Randomized Renaming, *Information Processing Letters* 61(1), 1997, pp. 7-10.

[4] T. Hagerup, Fast Parallel Space Allocation, Estimation and Integer Sorting, *Proc. IEEE Symposium on Foundations of Computer Science*, 1991.

[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, pp. 307-316.

[6] S. Rajasekaran and J.H. Reif, Optimal and Sub-Logarithmic Time Randomized Parallel Sorting Algorithms, *SIAM Journal on Computing*, 18(3), 1989, pp. 594-607.

[7] R. Raman, The Power of Collision: Randomized Parallel Algorithms for Chaining and Integer Sorting, Technical Report 336, Dept. of Computer Science, University of Rochester, January 1991.