Optimal Parallel Randomized Renaming

Martin Farach* S. Muthukrishnan†

September 11, 1995

Abstract

We consider the *Renaming Problem*, a basic processing step in string algorithms, for which we give a simultaneously work and time optimal Las Vegas type PRAM algorithm. The Renaming Problem is closely related to the *Multiset Sorting Problem.*

1 Introduction

The *Renaming Problem* is defined as follows:

Input: An array $A[1, n]$ consisting of $k \leq n$ distinct elements from an ordered universe.

Output: An array B such that $B[i] = j$ if j is the rank of $A[i]$ in A, i.e. $A[i]$ is the jth smallest element of the array from amongst the k distinct elements of A.

Throughout this paper, we will be working in the comparison model, that is, we may only access the elements of A by comparing them with each other. Furthermore, we will take k to be unknown so that k will be an input-sensitive parameter for our problem.

In the sequential setting this problem is equivalent to the *Multiset Sorting Problem*, defined as follows:

Input: An array $A[1, n]$ consisting of $k \leq n$ distinct elements from an ordered universe.

Output: An array B containing a sorted permutation of A.

These problem are equivalent in that there are linear time reductions in both directions. Sequential algorithms for these problem can be made to run in $O(n \log k)$ time via e.g. 2–3 trees. A standard counting argument gives a lower bound of $\Omega(n \log k)$ for the multiset sorting problem, which, combined with the linear time reductions, shows that $\Theta(n \log k)$ is the correct bound for both problems in the sequential setting.

*Rutgers University; farach@cs.rutgers.edu; http://www.cs.rutgers.edu/∼farach; Supported by an NSF Career Advancement Award.

†University of Warwick and DIMACS; S.Muthukrishnan@dcs.warwick.ac.uk; Supported by DIMACS (Center for Discrete Mathematics and Theoretical Computer Science), a National Science Foundation Science and Technology Center under NSF contract STC-8809648.

Consider now the parallel setting[1]. We consider *work-optimal* algorithms which are those with processor-time product of $O(n \log k)$ for this problem. Multiset Sorting and Renaming are not equivalent on PRAMs, as can be seen by considering the case where k is known to be 2. The multiset sorting problem then is equivalent to the problem of determining the parity of n bits which has a well-known lower bound of $\Omega(\log n / \log \log n)$ even with polynomially many processors [3]. On the other hand, the renaming problem can be solved in this case in $O(1)$ time and $O(n)$ work. So renaming is provably easier than multiset sorting in terms of parallel time.

We consider the general case with arbitrary unknown k. Any work-optimal algorithm for this problem must take $\Omega(\log k)$ time. Otherwise, $\omega(n)$ processors would have to be allocated[2]. If k turns out to be a constant, any such algorithm would end up doing $\omega(n \log k)$ work, thus violating work-optimality. Therefore, only $O(n)$ processors can be allocated, forcing the algorithm to take $\Omega(\log k)$ time. Finally, all lower bounds above apply equally to randomized algorithms.

In this paper, we give the first both time and work optimal randomized algorithm for the renaming problem, that is, an algorithm which takes $O(\log k)$ time and $O(n \log k)$ work w.h.p. (that is, the failure probability is at most inverse polynomial in n). Throughout we use $|A|$ to denote the size of an array A.

1.1 Motivation

A great deal of attention has been focused on the model of computation for string matching problems. In the seventies and eighties, the main focus for this type of research was automata-theoretic. More recently, the central issue has been how characters may be accessed and compared on RAMs [1]. The upshot of the character-theoretic research is that there seems to be two distinct classes of problems, i.e., those which require an ordered character universe and those which rely only on character equality.

The result in this paper is revelant to the former class of string matching problems. In particular, a typical preprocessing in the sequential setting involves renaming characters to integers so that they may be hashed, used as array indices etc. Parallel algorithms for this class of problems also rely on the same integer based techniques. However since optimal PRAM algorithms for these problems are a relatively recent development, it was generally assumed that renaming could be implemented via comparison sorting. This was possible because sorting was never the work-bottleneck. For example, when the most efficient parallel suffix tree construction algorithm took $O(n \log n)$ work and $O(\log n)$ time with integer characters, the authors justify their claim that the same bounds hold in the weaker comparison model since sorting can be performed as a preprocessing within these bounds [2].

Suffix trees can now by constructed in $O(n)$ work and $O(\log n)$, for n bit strings [5]. This algorithm, combined with the renaming algorithm provided in this paper, gives the first work optimal, i.e. $O(n \log \sigma)$, suffix tree construction for an n character string consisting of σ distinct characters drawn from an ordered universe. Note that this is exactly the standard model assumed in sequential suffix tree construction. We also obtain the first work optimal parallel dictionary

[1] Throughout this paper we assume the arbitrary Concurrent Read Concurrent Write Parallel Random Access Machine (CRCW PRAM) [8].

[2] $f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$.

matching algorithm [**?**]. In general, therefore, we provide a technique for converting comparison problems to integer problem without work degradation on the CRCW PRAM.

2 Renaming

We solve a related problem called DistinctSort in which we will output the sorted list of only the distinct items from the input array. Notice that this differs from Multiset Sorting in that each distinct element is represented *once* in the sorted list. Once this list has been computed, each element of A can find its rank in $O(\log k)$ time with a single processor via binary search, thus completing the computation for renaming.

Given quadratically many processors DistinctSort can be performed as follows.

Lemma 2.1 *Given an array A of size $\sqrt{n}$ with k distinct elements, we can sort the distinct elements of A in $O(\log k)$ time and $O(n \log k)$ work.*

Proof:

1. Each element $A[i]$ is assigned $\sqrt{n}$ processors, with which it checks in constant time if it is the leftmost occurrence of its value in the array. If not, $A[i]$ becomes inactive.
2. Each active element $A[i]$ find its nearest preceeding active neighbor in constant time via the rightmost-ones algorithm of [6] using a total of n processors.
3. A list ranking is performed by pointer doubling on the active elements of A. They are thus placed by rank in a new array B. This takes $O(\log k)$ time and $\sqrt{n}$ processors.
4. The elements of B are sorted via Parallel Mergesort [4], in $O(\log k)$ time with k processors.

■

Thus, if we choose a small sample from our input array, we can DistinctSort it within our target time bounds. Notice that a byproduct of such a procedure is a lowerbound on k, the number of distinct elements. Further, if we find that this lower bound is large (say $\Omega(n^{1/4})$), then we can sort the array directly, via Cole's Mergesort [4], and determine the output by directly computing the rank of the distinct elements via prefix sums. Otherwise, the lower bound is small and since we show that almost all elements of the input array are represented in our sample, we can recurse on the (proportionately) smaller surviving set by eliminating those represented in the sample from further consideration. We will show that we terminate after a constant number of rounds of this procedure. While our algorithm is obtained by random sampling somewhat reminiscent of randomized parallel sorting algorithms [9], the fact that k is unknown and that we seek an algorithm with time dependent on only k introduces new technical difficulties in our setting, which we overcome.

More concretely the algorithm is as follows.

DistinctSort($C[1, m]$): Procedure to find the distinct elements of array C and sort them, using n processors.

1. If $m \leq \sqrt{n}$, exit from the recursion and apply Lemma 2.1.
2. Set $\mu = \frac{\sqrt{n}}{m \log n}$. Independently select each element of C with probability μ. (Therefore, we expect to have selected $\sqrt{n}/\log n$ elements.)
3. Compact selected items from C into an array D; See Lemma 2.4.
4. Sort the unique elements of D into array E via Lemma 2.1.
5. If $|E| \geq n^{1/4}$, exit from the recursion and sort C directly using Mergesort [4] and determine the output by directly computing the rank of the distinct elements via prefix sums.
6. Each element $C[i]$ searches for its value in E via binary search with a single processor. If its value is found $C[i]$ becomes inactive. Otherwise it remains active.
7. The active elements of C along with the elements in E are compacted into F. Return the list output by DistinctSort(F).

To solve our problem, we need to invoke DistinctSort($A[1, n]$).

2.1 Analysis

We first prove some general lemmas.

Lemma 2.2 *Given an array A of size n, a sample S in which each element from A is chosen independently and randomly with probability p, with K being the set of distinct elements from S, let $k = |K|$ and let $A - K$ be the compacted array in which all elements from K are removed from array A. Then, $A - K$ is of size $O(k \log n/p)$ w.h.p.*

Proof: Let B be the sorted array of the elements in A. For our proof, we consider B. The elements in S divide B into disjoint intervals. Consider an element $A[i]$. Suppose it lies in the interval between two elements of S in B and these two elements are both equal to, say e. Then it follows that $A[i] = e$ since B is sorted. In that case $A[i]$ cannot be in $A - K$. Therefore the size of $A - K$ is at most the total number of elements in B that lie in any interval between two elements of S that are distinct. Note that there are at most k such intervals. In what follows we will prove that w.h.p no interval has $\Omega(\log n/p)$ elements; that would prove $|A - K| = O(k \log n/p)$ w.h.p. We have

$$\begin{aligned} Pr(\text{Some interval has} \geq l \text{ elements}) &= \text{No. of intervals} \quad Pr(\text{An interval has} \geq l \text{ elements}) \\ &\leq n(1-p)^l \\ &\leq ne^{-pl} \qquad \text{since } (1-x) \leq e^{-x} \text{ for } x < 1 \\ &\leq 1/n^c \text{ for any constant } c \text{ setting } l = O(\log n/p) \end{aligned}$$

It follows that w.h.p. no interval has $\Omega(\log n/p)$ elements. That proves the lemma. ∎

Lemma 2.3 DISTINCTSORT *performs only* $O(1)$ *recursive rounds.*

Proof: We will simply unravel the first few rounds of recursion and determine the sizes of various arrays. Let the subscript i denote the ith round of recursion with any parameter in the algorithm. For the top level, $i = 1$; thus $m_1 = 1$ and $\mu_1 = \frac{1}{\sqrt{n}\log n}$. Since $|E_1| < n^{1/4}$ for round 2 to begin, we have $m_2 = O(\frac{n^{1/4}\log n}{\mu_1}) = O(n^{3/4}\log^2 n)$ by applying Lemma 2.2. Now it is easy to check that $\mu_2 = \Omega(\frac{1}{n^{1/4}\log^2 n})$ and $m_3 = O(n^{1/2}\log^3 n)$ if round 3 commences. Continuing similarly, $\mu_3 = \Omega(1/\log^3 n)$ and $m_4 = O(n^{1/4}\log^4 n)$ if round 4 commences. But $m_4 = O(n^{1/4}\log^4 n) \leq n^{1/2}$ for sufficiently large n. Thus the recursion cannot proceed beyond that round (that is, round 4) in Step 1. That proves the lemma. ∎

We now analyze the time and work complexity of each round. We argue that each round takes $O(\log k)$ time and $O(n \log k)$ work w.h.p. Steps 2, 5, and 6 have trivial implementations within these bounds. We focus on the other steps. In steps 3 and 7, we need to compact arrays. It is easy to see that each such time the arrays have at most n^ϵ active elements for some fixed $\epsilon < 1$. Then, the following result can be applied to get the desired bounds.

Lemma 2.4 ([7]) *Given an array of size n with n^ϵ selected items for any fixed $\epsilon < 1$, there is a randomized algorithm to compact the selected items into an array linear in the number of selected items in $O(1)$ time and $O(n)$ work.*

It remains to consider Step 7. It is easy to see that $Pr(|C| = \Omega(m\mu \log n))$ is polynomially small in n since the expected value of $|C|$ is $m\mu$. Thus, w.h.p, $|C| = O(\sqrt{n})$ so Lemma 2.4 as well as Lemma 2.1 may be applied. That leads to

Theorem 2.5 DISTINCTSORT *takes $O(\log k)$ time and $O(n \log k)$ work on an arbitrary CRCW PRAM w.h.p.*

Proof: ¿From the preceding discussion, it follows that each round takes $O(\log k)$ time and $O(n \log k)$ work w.h.p. Combined with Lemma 2.3, that gives the bound in the theorem. The correctness follows easily since we only eliminate those elements which are not unique in the array and we always retain at least one of the copies. Also, we finally end up with precisely one copy of each item. ∎

Finally as remarked earlier, the renaming problem can be solved in $O(\log k)$ time and $O(n \log k)$ work once DISTINCTSORT is performed. That the resultant algorithm is of the Las Vegas type follows since Lemma 2.4 is of the Las Vegas type (that is, it detects an error if there are more than n^ϵ active items) and we can check if the output of DISTINCTSORT contains all the elements in C in $O(\log k)$ time and $O(n \log k)$ work. That proves our main result.

We leave it open to design a fast deterministic optimal parallel algorithm for this problem.

References

[1] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two dimensional pattern matching. *SIAM Journal on Computing*, 23(2), 1994.

[2] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Scieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[3] Paul Beame and Johan Hastad. Optimal bounds for decision problems on the CRCW PRAM. In *Proc. of the 19th Ann. ACM Symp. on Theory of Computing*, pages 83–93, 1987.

[4] Richard Cole. Parallel merge sort. In *Proc. of the 27th IEEE Annual Symp. on Foundation of Computer Science*, pages 511–516, 1986.

[5] M. Farach and S. Muthukrishnan. A fast and efficient suffix tree contruction algorithm. Technical report, DIMACS, 1995.

[6] Faith E. Fich, Prabhakar L. Ragde, and Avi Wigderson. Relations between concurrent-write models of parallel computation (preliminary version). In *Proceedings 3rd Annual ACM Symposium on Principles of Distributed Computing*, pages 179–189, 1984.

[7] M. Goodrich. Using approximation algorithms to design parallel algorithms that may ignore processor allocation. *Proc. of the 32nd IEEE Annual Symp. on Foundation of Computer Science*, pages 711–722, 1991.

[8] Joseph JáJá. *Introduction to Parallel Algorithms*. Addison-Wesley, Reading, MA, 1991.

[9] Rüdiger Reischuk. A fast probabilistic parallel sorting algorithm. In *Proc. of the 22nd IEEE Annual Symp. on Foundation of Computer Science*, pages 212–219, 1981.