Using Learning and Difficulty of Prediction to Decrease Computation: A Fast Sort and Priority Queue on Entropy Bounded Inputs *

Shenfeng Chen and John H. Reif

Abstract

There is an upsurge in interest in the Markov model and also more general stationary ergodic stochastic distributions in theoretical computer science community recently, (e.g. see [Vitter,Krishnan,FOCS91], [Karlin,Philips,Raghavan,FOCS92] [Raghavan92]) for use of Markov models for on-line algorithms e.g., cashing and prefetching). Their results used the fact that compressible sources are predictable (and vise versa), and show that on-line algorithms can improve their performance by prediction. Actual page access sequences are in fact somewhat compressible, so their predictive methods can be of benefit.

This paper investigates the interesting idea of decreasing computation by using learning in the opposite way, namely to determine the difficulty of prediction. That is, we will approximately learn the input distribution, and then *improve the performance of the computation when the input is not too predictable, rather than the reverse.* To our knowledge, this is first case of a computational problem where we do not assume any particular fixed input distribution and yet computation is decreased when the input is less predictable, rather than the reverse.

We concentrate our investigation on a basic computational problem: sorting and a basic data structure problem: maintaining a priority queue. We present the first known case of sorting and priority queue algorithms whose complexity depends on the binary entropy $H \leq 1$ of input keys where assume that input keys are generated from an unknown but arbitrary stationary ergodic source. This is, we assume that each of the input keys can be each arbitrarily long, but have entropy H. Note that H

*Surface mail address: Department of Computer Science, Duke University Durham, NC 27706. Email addresses: reif@@duke.cs.duke.edu and chen@duke.cs.duke.edu. Supported by DARPA/ISTO Contracts N00014-88-K-0458, DARPA N00014-91-J-1985, N00014-91-C-0114, NASA subcontract 550-63 of prime contract NAS5-30428, US-Israel Binational NSF Grant 88-00282/2, and NSF Grant NSF-IRI-91-00681. Note: full proofs of all results of this extended abstract appear either in the main text or in the appendices.

can be estimated in practice since the compression ratio ρ using optimal Ziv-Lempel compression limits to $1/H$ for large inputs. Although sets of keys found in practice can not be expected to satisfy any fixed particular distribution such as uniform distribution, there is a large well documented body of empirical evidence that shows this compression ratio ρ and thus $1/H$ is a constant for realistic inputs encountered in practice [1, 31], say typically around 3 to at most 20. Our algorithm runs in $O(n \log(\frac{\log n}{H}))$ sequential expected time to sort n keys in a unit cost sequential RAM machine. This is $O(n \log \log n)$ with the very reasonable assumption that the compression ratio $\rho = \frac{1}{H}$ of the input keys is no more than $\log^{O(1)} n$.

Previous sorting algorithms are all $\Omega(n \log n)$ except those that (i) assume a bound on the length of each key or (ii) assume a fixed (e.g., uniform) distribution. Instead, we learn an approximation to a *unknown* probability distribution (which can be any stationary ergodic source, not necessarily a Markov source) of the input keys by randomized subsampling and then implicitly build a suffix tree using fast trie and hash table data structures.

We can also apply this method for priority queue. Given a subsampling of size $n/(\log n)^{O(1)}$ which we use to learn the distribution, we then have $O(\log(\frac{\log n}{H}))$ expected sequential time per priority queue operation, with no assumption on the length of a key.

Also we show our sequential sorting algorithm can be optimally speed up by parallelization without increase in total work bounds (though the parallel time bounds depend on an assumed maximum length L of each key). In particular, if $L \leq n^{O(1)}$, we get $O(\log n)$ expected time using $O(n \log(\frac{\log n}{H})/\log n)$ processors for parallel sorting of n keys on a CRCW PRAM.

We have implemented the sequential version of our sorting algorithm on SPARC-2 machine and compared to the UNIX system sorting routine - quick sort. We found that our algorithm beats quicksort for large n on extrapolated empirical data. Our algorithm is even more advantageous in applications where the keys are many words long.

Sorting is one of the most heavily studied problems in computer science. Given a set of n keys, the problem of **sorting** is to rearrange this sequence either in ascending order or descending order. There has been extensive research in sorting algorithms, both in sequential and parallel settings (see next section for detail).

Sorting is of great practical importance in scientific computation and data processing. It has been estimated that twenty percent of the total computing work on mainframes is sorting. Therefore, even an improvement of a constant factor will have a large impact in practice. Though the theoretical research has already had large impact, there are still fundamental problems remaining. For example, in the study of sequential sorting algorithms, the *comparison tree* model assumes that the only operation allowed is comparison and it is well known that the *comparison sort* (i.e. merge sort and heapsort [**?**]) based on comparison tree model has a lower bound of $\Omega(n \log n)$ for sorting n elements.

Sorting algorithms	*Assumptions on inputs*	*Running time*
Counting sort	integers in range $[1, k]$	$O(k)$
Radix sort	bounded length d of key	$O(nd)$
Bucket sort	random distribution	$O(n)$
Our sort	bounded entropy H	$O(n \log(\frac{\log n}{H}))$
Andersson's sort	bounded length B of distinguishing prefix	$\Theta(n \log(\frac{B}{n \log n} + 2))$

Table 1: Sorting algorithms not based on comparison-tree model.

However, this bound can be relaxed by allowing operations other than comparison to achieve time bound less than $O(n \log n)$. For example, radix sort and bucket sort which are not based on comparison tree model have a linear running time assuming that the input keys are drawn from uniform distribution.

As parallel algorithms now play a bigger and bigger role in algorithm implementations, proposed parallel sorting algorithms must be capable of being optimally sped up (so the product of time and processor equals the total work of optimal sequential algorithm to parallel machines). This is essential in implementations on real machines since most high performance machines have parallelism. For example, the powerful CRAY computer can be viewed as a large vector machine.

Another important aspect in designing fast sorting algorithms is to explore the input data so that the sorting algorithm can take advantage of input data satisfying certain properties (e.g. certain probability distribution). This type of algorithms includes bucket sort and radix sort, with different assumptions on the input data. In this chapter, we design a sorting algorithm based on a specific statistical property, namely the bounded entropy, which is true for most practical files. We give both sequential and parallel versions of the algorithm. In order to achieve high performance in practice, we use the digital search tree (trie) data structure used in data compression algorithm to speed up operations on binary strings. We also give several applications based on the sorting algorithm.

1 Previous Work and Our Approach

Before we present our sorting algorithm, we first summarize previous approaches for sorting problems and explain why we choose various theoretical assumptions for our algorithm design.

Computational Model. The *Comparison tree* model assumes that the only operation allowed in sorting is comparison between keys to gain order information [**?**]. That is, given two keys x_i and x_j, we perform one of the following tests $x_i < x_j$, $x_i \leq x_j$, $x_i = x_j$, $x_i \geq x_j$, or $x_i > x_j$ to determine

their order. Other ways such as inspecting the values of the keys are not allowed to tell the order. Though an excellent model for theoretical analysis, the comparison model has a drawback that it requires a $\Omega(\log n)$ lower bound per key for sorting even with the assumption that keys are uniformly distributed. Many sorting algorithms are based on comparison tree model and achieve this optimal time bounds (e.g. quicksort, mergesort [**?**]).

On the other hand, there have been sorting algorithms (bucket sort, radix sort, etc) proposed which are not based on comparison trees (see Table **??** for a list of sorting algorithms which are not based on comparison tree model). For example, many algorithms instead adopt a more general *unit cost RAM* model which assumes that the usual operations such as addition, shift, multiplication, bit comparison are regarded as one single step. The unit cost RAM is a more realistic model for the actual machine architecture and thus has the advantage to be used to design algorithms which do not have the striction of $O(\log n)$ lower time bound.

Two well known examples are counting sort, radix sort and bucket sort. Counting sort assumes the input consists of small integers to achieve a linear running time. The radix sort makes the assumption that every input key is a d-digit integer where d is a constant and also runs in linear time. The bucket sort runs in linear expected time because it assumes that the input is generated by a random process that distributes elements uniformly over the interval $[0, 1)$.

In our sorting algorithm, we assume the unit cost RAM as our computational model with the general assumption that the input keys are drawn from a bounded-entropy stationary and ergodic source.

Randomized vs. Determininistic Algorithms. Randomized sorts, such as quicksort which were initially considered only of theoretic interest, are used on many if not most system sorting routines for large inputs. In general, randomized algorithms (sequential or parallel) are simpler and easier to implement than the deterministic algorithms which achieve the same time complexity, though the time complexity for the algorithm is with high probability in the case of randomized algorithms. The parallel variants of randomized algorithms, such as FLASHSORT [**?**] and SAMPLESORT[**?**], have given some of the best implementations of parallel sorting (see [**?**] and [**?**] respectively). Therefore, considering the actual performance of the algorithm, it is reasonable to adopt the randomized method for parallel version of our algorithm.

Assumptions on Input Keys (i) *Maximum number of bits per key.* On the unit cost RAM, the input keys of sorting algorithms can be regarded as a sequence of binary numbers with different length (the keys can be easily converted to binary representation if it is not the case). Some sorting

algorithms, such as the algorithms based on data structure of digital search tree (trie), and Radix Sort, assume that the number of bits per key is bounded by L. Radix Sort [**?**] requires $O(L/\log n)$ sequential work for sorting each key. The sorting algorithms using trie approach [**?**, **?**] requires $O(\log L)$ sequential work per key. However, the assumption of bounded maximum length does not seem to hold widely in practice: in fact, many major users of sorting routines, e.g., data base joins, require moderately large key size L. In our algorithm, we does not impose an upper bound on the maximum length of input keys. In other words, the running of our algorithm is independent of the maximum length of the input keys. Instead, since we use prefix matching to find the order of the input keys, the complexity of the algorithms depends on the length of the first prefix match between two given keys.

(ii) *Random input distribution vs. worst-case analysis.* Another commonly used assumption in sorting algorithm is to analyze the performance of the sorting algorithm based on certain distribution of the input, e.g. random distribution or worst case input. The apparent advantage of the worst-case input assumption is that the sorting time bound can be guaranteed even in the worst case. As we illustrated in the introduction (see Chapter 1), in practice the probability of sorting a worse case input is very small. On the other hand, if we assume random distribution of the inputs, in the bucket sorting algorithm [**?**], sorting each key needs only $O(1)$ expected time. But the random input assumption also does not hold very frequently in practice as well. Therefore, we do not make assumption on the probability distribution of the inputs, nor do we base our algorithm analysis on the worst case inputs. Instead, we only assume that the input keys are drawn from a stationary and ergodic source with the underlying probability unknown before running the algorithm.

(iii) *Entropy bounded input sets.* As computer scientists, we would like to design algorithms which are applicable to files found in practice. Therefore it is a good question to ask what general properties of the practical inputs we can exploit to speed up the sorting algorithm. There have been various sorting algorithms which are not based on comparison-tree model and make different assumptions about the input source. For example, Manilla [**?**] gave an optimal sorting algorithm by measuring presortedness of input. As discussed in the introduction of this thesis, for large files occurring in practical applications, the lossless compression ratio is often at least 1.5 and no more than a relatively small constant, say 20 (see definition of entropy and compression ratio in Chapter 1). It is not reasonable to assume uniformly distributed sets of keys for practical files (otherwise, compression ratio would almost always be 1, which is not the case), so bucket sort is not appropriate or at least has limited applicability for sorting large files in practice. Since the performance of our sorting algorithm depends

on the compressibility of the input keys, we will assume that the source from which the input keys are generated has a bounded compressibility.

Based on the above observations, we choose assumptions carefully for our new sorting algorithm to suit the purpose of designing a practical sorting algorithm. We assume the randomized unit cost RAM as the computational model. In order for the algorithm to have general applications (e.g., database join applications). we assume no upper bound for maximum number of bits that each key can have. Since we utilize data structures and techniques used in data compression algorithms, the time complexity of our algorithm depends on the compressibility of the input source. We assume that the compressibility is bounded which is true for practical inputs so that the algorithm runs well in practice.

1.1 Computational Model

Recall from the introduction that the unit cost RAM model we use allows the basic operations such as addition, shift, and multiplication to be accomplished in one single step. We regard the input keys as strings of binary bits. Our analysis depends on the total number of keys we are processing instead of the total number of bits the input has. Therefore, we assume no bound on the maximum length of a given input key. The actual machine memory layout of the key is irrelevant to the discussion of the algorithm. Specifically, we make the following assumption to the computational model. Any input key x, represented by a binary string can be regarded as a binary number representing an integer. In other words, the input keys can also be viewed as a sequence of integers with no upper bound. This assumption is needed to insure that operation on trie data structure, such as finding the longest prefix match in the trie for a new key, can be accomplished efficiently. We will describe in detail such operations in next section.

1.2 Statistical Properties of Input Keys

We address the problem of relating the complexity of sorting to some statistical properties of the input keys such as optimal compression ratio and entropy. Suppose the input source from which the input keys $X_1, X_2, X_3, ..., X_n$ are generated is a stationary and ergodic sequence built over a binary alphabet {0,1}. Recall from the introduction that informally, "ergodicity" means that as any sequence produced in accordance with a statistical source grows longer enough, it becomes entirely representative of the entire model. On the other hand, "stationarity" means the underlying probability mass function of the input source is not changing over time

Let $\{X_i, i = 1...n\}$ be the set of input keys which are generated by a stationary source with the *mixing condition*. Recall from the introduction that a

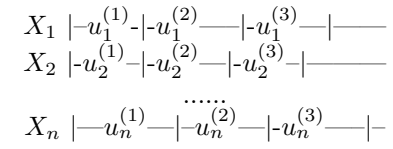

Figure 1: Prefix matches of input keys.

sequence $\{X_k\}_1^\infty$ is said to satisfy the mixing condition if there exists two constants $c_1 \leq c_2$ and integer d such that for all $1 \leq m \leq m+d \leq n$ the following holds:

$$c_1 \Pr\{\mathcal{B}\} \Pr\{\mathcal{C}\} \leq \Pr\{\mathcal{BC}\} \leq qc_2 \Pr\{\mathcal{B}\} \Pr\{\mathcal{C}\} \tag{1}$$

where $\mathcal{B} \in \mathcal{F}_{-\infty}^m$ and $\mathcal{C} \in \mathcal{F}_{m+d}^n$. This mixing condition implies ergodicity of the sequence $\{X_k\}_{-\infty}^\infty$.

We now define F_n to be the input file of keys $X_1, X_2, ..., X_n$ concatenated together. In our algorithm, we only need to consider the first prefix match of any given key to the dictionary. However, the definition of compression ratio is consistent with the compression ratio that we achieve when we consider compressing the concatenated file F_n, since F_n can also be viewed as generated by the same statistical source. In Figure **??** below, X_i denotes the ith input key with arbitrary length and u_i^j denotes the jth prefix match of the ith key achieved by matching against the LZ dictionary (see section below for details). In our sorting algorithm, only the first prefix match u_i^1 will be involved in the analysis of the algorithm.

Let $\bar{L}$ be the average length of the first prefix match, i.e., $\bar{L} = E(u_i^{(1)})$. The optimal LZ code for n keys has $\log n$ bits [**?**]. Thus, the expected optimal compression ratio when we consider the sequence of concatenating $u_1^{(1)}$,$u_2^{(1)}$,$u_3^{(1)}$... is $\frac{\bar{L}}{\log n}$ as $n \to \infty$ [**?**]. However, as the whole input is generated from a stationary and ergodic source, this compression ratio also limits to $\frac{\bar{L}}{\log n}$ when we consider compressing the input file F_n. Let H denote the entropy of the input source. We have the following lemma:

For large n as $n \to \infty$, $H = Entropy(u_i^{(1)}) = Entropy(F_n)$ for each i where F_n is the concatenation of n input keys.

1.3 Random-Sampling Techniques

In our algorithm, a well known technique of randomization is used to make the algorithm run efficiently in practice. Randomization has been successfully used in a large number of applications and has recently been used to obtain efficient algorithms in sorting. For example, sample sort [**?**] is a sorting algorithm using random sampling techniques to adapt well to inputs with different distributions.

The general approach taken by random sampling algorithms is as follows. We randomly choose a subset R of the input set S to partition the problem into smaller ones. The partitioned sub-problems can be proven to have a bounded size with high likelihood. We then solve the sub-problems recursively using known techniques and form the solution of the whole problem based on the solutions of the sub-problems. Clarkson [**?**] has proved that for a wide class of problems in algorithm design, the *expected* size of each subproblem is $O(|S|/|R|)$ and more over the *expected* total size of the subproblem is $O(|S|)$. Clarkson's results show that by using a straight-forward random sampling technique any randomly chosen subset is within the expected size with constant probability, implying that it may be not with constant probability. Consequently, his methods yields *expected* resource bounds but cannot be used to obtain high-likelihood bounds (i.e. the bounds that hold with probability greater than $1 - n^c$ for any $c > 0$). This makes it very difficult to extend his methods to the context of parallel algorithms.

Later a random sampling technique called *over-sampling* is introduced which chooses $s|R|$ number of *splitters* and then partitions the set into $|R|$ groups separating by the first key, the s-th key, the $2s$-th key and so on. The number s is called *oversampling ratio*. This technique overcomes the shortcomings of Clarkson's method. Reif and Sen [**?**] also proposed a random sampling technique called *polling* to obtain "good" samples with high probability with small overhead.

These random sampling techniques are especially useful for designing parallel algorithms where the primary goal is to distribute the work evenly across the processors. There has been extensive research on implementing various sorting algorithms on parallel architectures [**?**, **?**, **?**]. Also various parallel sorting algorithms have been implemented on real parallel machine architectures. The random sampling techniques are used in parallel sorting to distribute the input keys to processors according to selected buckets where the size of each bucket is bounded within a constant of expected size with high likelihood. The main advantage is that the *load balancing* between processors can be achieved and the parallel running time which depends on the longest running time of processor can also be bounded. Reif and Sen [**?**] used random sampling techniques to solve many parallel computation geometry problems (e.g. Voronoi diagram, trapezoidal decomposition) where the problems are divided into subproblems by random sampling and the size of the subproblem is crucial to the parallel running time of the algorithm.

Another area in which the random sampling techniques are found useful is designing efficient algorithms whose performance depends on the distribution of the inputs. In these algorithms, the inputs are normally randomly subsampled in order to learn the underlying probability distribution of the inputs. Our sorting algorithm uses random sampling mainly as a tool to learn the unknown probability distribution of the input source and use this information to aid our computation.

2 Sequential Sorting Algorithm

Our sequential sorting algorithm consists of two phases. In the first phase, we use randomized sampling technique to learn an approximation of the probability distribution of inputs. A digital search tree (trie) is constructed using the sampled keys in a way similar to the dictionary construction in LZ compression. The trie is used to reflex an approximation of the actual probability distribution of the input keys. The trie is also used to partition the inputs into separate buckets.

In the second phase, the input keys are indexed into buckets based on its prefix match in the trie. We use a standard $O(n \log n)$ time sorting algorithm (e.g. quicksort) to sort each bucket separately and link buckets together using trie structure.

2.1 Trie Data Structure

The basic idea of our sorting algorithm is to divide the input keys almost evenly into partitioned buckets. We use a method similar to the construction of a *dictionary* in LZ compression to index the input keys into buckets. It differs from the original dictionary construction in the way that the main purpose for our algorithm is approximating the unknown distribution of the input keys using the dictionary instead of actually compressing the inputs.

We now show how to build the dictionary D using a *trie* data structure (see Chapter 2 for a detailed presentation on trie). A trivial implementation of the trie is to use a simple binary tree whose nodes correspond to phrases of the dictionary. However, the binary tree method is inefficient because the comparison between an input key and the phrases in the tree is done only on a bit-by-bit basis, taking $O(L)$ time to find the maximum prefix match between the key and trie where L is the length of the maximum prefix match.

In our algorithm, we adopt the *hash table* implementation of the trie where comparisons can be done more efficiently than the simple binary tree implementation. Each phrase in the trie is now stored in a certain slot of the hash table where the slot number is determined by a hash function. For the purpose of a clear presentation, we use the terminologies dictionary (in the context of data compression), trie (the data structure) and hash table (the actual implementation) interchangeably for the same data structure in the following description of the algorithm.

We now show how to find the longest prefix match for each input key. The running time analysis is based on our assumption of the computational model. Specifically, we assume that any given key can be viewed as an integer and comparison of integers and hash function computation can all be accomplished in constant time. Let D_s be the current dictionary we have constructed. For any new input key x, let $Index(x)$ be the longest prefix of x that is already stored in the dictionary and d_x is the length of $Index(x)$. We can find $Index(x)$ in

$O(\log d)$ time by searching in the hash table where d_i is the length of $Index(x_i)$ (see [**?**]). The basic idea of the search is that $Index(x_i)$ can be identified by doubling the length of prefix of x for comparison each time (similar to binary search). Without loss of generality, we assume $d = 2^k$ (k is an intger) be the length of the maximum prefix match of x. We first search in hash table to check whether the prefix consists of only the first character of x is already stored stored in the trie. This is done by computing the hash function $h(x, 1)$ to find the index number in the hash table where this character belongs. If we find that the first character is already stored in the hash table, we double the length of the prefix under inspection to check the prefix which consists of the first two characters of x_i and so on. When we find that the prefix of length 2^{k+1} of x is not in the trie, we reverse the order of search using the same doubling (or sometimes halving) method to locate the longest prefix of x which is already in the trie. Since the query of whether a given prefix is in the hash table can be answered in constant time, the total search time depends on the length of the first prefix match. Specifically, we find the longest prefix match of input key x in $O(\log d)$ time where d is the length of the longest prefix match.

2.2 Description of the Algorithm

We are now ready to present the sequential sorting algorithm. We assume the binary strings representing input keys are generated by the same stationary and ergodic source whose probability distribution is initially unknown. Our algorithm consists of two phases. In the first phase, the unknown distribution of the inputs is learned by subsampling from the input and a dictionary is built using a trie structure from the sampled keys. In the second phase, all input keys are indexed to buckets associated with the leaves of the dictionary and each bucket is sorted independently and then concatenated together to produce the output. A probabilistic analysis (Lemma **??**) shows that the size of the largest bucket can be bounded by $O(\log n/H)$ with high likelihood (by high likelihood, we mean with probability $\geq 1 - 1/n^{\Omega(1)}$). We also show that the sequential time complexity of this algorithm is expected to be $O(n \log\log n)$, given that the input keys are not highly compressible, i.e., the compression ratio $\rho \leq (\log n)^{O(1)}$.

We present a randomized algorithm for sorting n keys $X_1, X_2, ..., X_n$. In the first phase, the distribution of the input keys is learned by building a data structure similar to a dictionary used in LZ compression.

2.2.1 Random Sampling

First we randomly choose a set S of $m = n/\log n$ sample keys from the input keys and arrange them in random order. We will apply a modification of the usual LZ compression algorithm to construct a dictionary D_s from the first prefixes of the sample keys in S as described just below. The leaves of D_s represent the

set of points which partition the range of inputs into separate buckets according to the prefixes of input keys. Once D_s is constructed (and implemented by an efficient hash table [**?**]), we consider each key X_i for $1 \leq i \leq n$ and index it to the corresponding bucket which is represented by leaves in the trie. Let $X_i = u_i^{(1)} u_i^{(2)} ...$ be the LZ prefix decomposition of X_i using the dictionary D_s. Let $u_i^{(1)}$ be the longest prefix of X_i that is already stored in the dictionary D_s.

2.2.2 Dictionary Construction

We now describe the construction of dictionary D_s. It is similar to original LZ algorithm except that we do not actually compress the input key. D_s will be incrementally constructed from an initially nearly empty set $\{0, 1\}$ by finding the first prefix match $u_{s_i}^{(1)}$ of each sampled key X_{s_i} (considered at random order) in the current constructed dictionary, and then adding the concatenation of $u_i^{(1)}$ with first bit of $u_i^{(2)}$ to the dictionary. This is different from the standard LZ scheme as we stop processing this key now and go on to the next key. Each step takes time $O(\log |u_{s_i}^{(1)}|) = O(\log d_{s_i})$. The main purpose of this dictionary is to form an approximation of the probability distribution of the input source so that we can partition the range of input keys into buckets of nearly even size.

2.2.3 Indexing of Keys

Let a *leaf* of D_s be an phrase in D_s which is not a strict prefix of any other phrases in D_s. Let LD_s be the set of leaves in D_s (these are the leaves of the suffix tree for D_s had it been explicitly constructed). Let $Index(x_i)$ be the maximal element of LD_s which is lexicographically less than x_i. Let d_i =length of $Index(x_i)$. Both $u_i^{(1)}$ and $Index(x_i)$ can be found in $O(\log d_i)$ sequential time by using trie search techniques.

2.2.4 Sorting

After all keys have been indexed to the corresponding bucket, the buckets are sorted separately using a standard sorting algorithm (e.g. quicksort) and then chained together according to the order of sample keys.

The sequential sorting algorithm is described as follows:

Sequential Sorting Algorithm

Input n keys $X_1, X_2, ..., X_n$, generated by a stationary ergodic source.
Output sorted n keys.

Step 1: Random sampling
Randomly choose a set S of $m = n/(\log n)$ sample elements from n input elements.

Step 2: Building buckets
2.1 Randomly permute $S = \{X_{s_1}, X_{s_2}, ..., X_{s_m}\}$.
2.2 Construct the dictionary D_s as described.
2.3 For each leaf w_j of D_s (recall a leaf is an element of D_s which is not a prefix of any other element of D_s), Bucket$[w_j] \leftarrow \emptyset$.
Step 3: Indexing keys
Indexing using D_s.
for each key $X_i, i = 1, ..., n$ do
3.1 Use trie search to find the $Index(X_i)$ (where index is defined above).
3.2 Insert X_i into the corresponding Bucket$[Index(X_i)]$ associated with the index of X_i.
Step 4: Sorting
4.1 Sort leaves of D_s using a standard sorting algorithm (e.g. Quicksort).
4.2 Sort elements within each bucket using standard sorting algorithm (e.g. Quicksort).
4.3 All buckets are linked under the order of the sorted leaves.

2.3 Analysis of Complexity

To bound the maximum size of the buckets, we have the following lemma: The size of each resulting bucket is $O(\log n/H)$ with probability $\geq 1 - \frac{1}{n^{\Omega(1)}}$ for large n where H is the entropy of the input source.

Proof: Recall that LD_s is defined to be the set of leaves of dictionary D_s. In [**?**] it is shown that there exist c, c' where c and c' are two positive numbers such that any leaf $y \in LD_s$ has probability between $c/|LD_s|$ and $c'/|LD_s|$ when $|D_s| \rightarrow \infty$.

This result of [**?**] implies that for each key X_i $(1 \leq i \leq n)$ with the same probability distribution, for each leaf $y \in LD_s$, there exist constants c'' and c''', such that

$$c''/|LD_s| \leq Prob(y = Index(X_i)) \leq c'''/|LD_s|.$$

Furthermore, [**?**, **?**] shows that the size of the set of leaves $|LD_s|$ limits to $H|D_s|$ for large $|D_s|$. Recall that in our algorithm $|D_s| = |S| = O(n/\log n)$. Combine the above results and the fact that the the number of indexed keys falling into a certain bucket follows binomial distribution, we conclude that the number of keys in

$$Bucket(y) = \{x_i | Index(x_i) = y\}$$

is less than $O(\log n/H)$ with probability $\geq 1 - \frac{1}{n^{\Omega(1)}}$. □

We state the time complexity theorem of our algorithms as follows.

The sorting algorithm takes $O(n \log(\frac{\log n}{H}))$ sequential expected running time where H is the entropy of the input keys.

Proof: Step 1 of the algorithm takes $O(|S|)$ time where $|S| = n/\log n$ is the number of the sampled keys. In step 3, we define average length of the first prefix match

$$\bar{L} = \frac{\Sigma_{i=1}^{n} d_i}{n} \tag{2}$$

Note that the optimal compression ratio ρ of the inputs is related to the expected length of first prefix match of the inputs such that

$$\rho = \frac{\bar{L}}{\log n}.$$

Also recall that searching the index of X_i takes $O(\log d_i)$ time where d_i is the length of $Index(X_i)$. Inserting key X_i into the corresponding bucket takes $O(1)$ time. Therefore the time complexity for step 3 is $\Sigma_{i=1}^{n} \log(d_i) = \Sigma_{i=1}^{n} \log(d_i) = \log \Pi_{i=1}^{n} d_i \leq \log \bar{L}^n = n \log \bar{L} = n \log(\rho \log n) = n \log(\frac{\log n}{H})$.

Similarly, the time for step 2 can also bounded by $n \log(\log n/H)$. Finally In step 4, sorting leaves in D_s only takes time no more than $O(n)$ as $|S| = n/\log n$ and $|LD_s| \leq |S|$. Then each bucket is sorted separately using a sorting algorithm which takes time $O(u_i \log u_i)$ for each bucket where u_i is the size of the ith bucket. Note that

$$|LD_s| \leq |S| = n/\log n.$$

By Lemma **??**, the size of each bucket is bounded by $O(\log n/H)$ with high likelihood, the expected time complexity of step 4 therefore is

$$\begin{aligned} T_4 &= \Sigma_{i=1}^{|LD_s|} t_i \\ &= \Sigma_{i=1}^{|LD_s|} u_i \log u_i \\ &\leq O(\Sigma_{i=1}^{n/\log n} u_i \log(\log n/H)) \\ &= O(n \log(\tfrac{\log n}{H})) \text{ as } \Sigma_{i=1}^{|LD_s|} u_i = n. \end{aligned}$$

Thus, the total sequential time complexity of the algorithm is $O(n \log(\frac{\log n}{H}))$.
□

We immediately have the following corollary assuming that the entropy H of the input keys is bounded:

The algorithm takes $O(n \log \log n)$ expected time if $H \geq \frac{1}{(\log n)^{O(1)}}$.

2.4 Experimental Testing

We implemented the sequential version of our sorting algorithm on SPARC-2 machine. The standard sorting algorithm we use for sorting the keys inside each bucket is the Sun OS/4 system call *qsort*. We tested the algorithm on 20 files with $24,000$ to $1,500,000$ keys from a wide variety of sources, whose compression ratios ranged from 2 to 4. We also derived an empirical formula for the runtime bound of our algorithm. For comparison, we also did this for

Figure 2: Comparison of our sorting algorithm with Quick sort.

the UNIX system sorting routine - Quicksort. The results are shown in Figure **??**.

It is well know that the time of sorting each key using Quick sort can be expressed in terms of the total number of keys: $T_q = c_0 + c_1 \log N$. As we already showed in the description of our sorting algorithm, the corresponding empirical formula for our sequential algorithm with fixed sample size is (regard the entropy as a small constant): $T_e = d_0 + d_1 \log\log N$.

We determined the constants using the experimental results:

$$c_0 = 14.2, c_1 = 1.68, d_0 = 40.00, d_1 = 3.73.$$

From the empirical formulas, it is easy to see that our sorting algorithm has a larger constant for the linear factor and that explains why our algorithm does not perform very well when the size of the input is small. The larger constant for the linear factor is due to the large overhead caused by random sampling and dictionary indexing. However, the linear factor becomes less dominant when the size of the input grows large enough, as the formula shows. As predicted, our sorting algorithm performs better and better when the input size becomes larger and larger. By extrapolation from these empirical tests we found our sequential sorting algorithm will beat quicksort when the total number of input keys $N > 3.2 \times 10^7$.

2.5 Discussion

In this section, we have presented the sequential sorting algorithm and its implementation. Like other non-trivial algorithms, our algorithms performs well when the size of the input becomes large enough to cancel out the effect of large overhead. The major cost of any sorting algorithm is the time spent for any key to decide its relative position in the whole input. Our basic idea is based on this observation and we try to eliminate unnecessary comparisons by dividing keys into ordered subgroups so that the only information needed for any key to be sorted is to find its relative position in its subgroup. Since the input key itself, especially its prefix, contains informations on which subgroup it belongs to, the preprocessing, or learning phrase of our algorithm uses this information to index keys to buckets instead of redundant comparisons between two keys.

We utilize the trie data structure to execute string operations such as finding the maximum prefix of any given key in the trie and inserting new phrases to the trie. We also use the idea of constructing a dictionary using a method similar to dictionary construction in LZ compression. The dictionary we construct has a nice property that it becomes a better and better approximation of the probability distribution of the input source as it incrementally inserts more and

more input keys. Intuitively, if certain prefix has a high probability of appearing in input keys, there will be proportionally many input keys with this prefix that are sampled. The dictionary will reflex the underlying probability distribution with high likelihood if the number of input keys is large enough. Furthermore, as a property of the trie, the probability of an input key reaching to any leaf of the trie is about the same (see Lemma **??**). This makes the task of finding a good partition of buckets especially easy.

Studies have indicated that sorting comprises about twenty percent of all computing on mainframes. Perhaps the largest use of sorting in computing (particularly in business computing) is the sort required for large database operations (e.g., required by joint operations). In these applications the keys have length of many machine words. Since our sorting algorithm hashes the key (rather than compare entire keys as in comparison sorts such as quicksort), our algorithm is even more advantageous in the case of large key lengths in which case the cutoff is much lower. In case that the compression ratio is high (i.e. the input source is highly predictable), which can be determined after building the dictionary, we just adopt the previous sorting algorithm, e.g., quick sort.

In later sections, we will demonstrate that the same techniques used in sequential sort (trie data structure, fast prefix matching etc) can be extended to other problems (e.g., computational geometry problems) to decrease computation by learning the distribution of the inputs.

3 Priority Queue Operations

A *priority queue*[**?**, **?**, **?**] is a data structure representing a single set S of elements each with an associated value called a *key*. A priority queue supports the following operations:
Insert(S, x): insert the element x into the set S. This operation can be written as $S \leftarrow S \cup x$.
Max(S): return the element of S with the largest key.
Extract-Max(S): remove and return the element of S with the largest key.

These operations can be executed *on-line* (i.e., in some arbitrary order and such that each instruction should be executed before reading the next one). The priority queues are useful in job scheduling and event-driven simulator. In some applications, the operations **Max** and and **Extract-Max** are replaced by operations **Min** and **Extract-Min** which returns the element with the smallest key.

Our sequential algorithm can be modified to perform priority queue operations. The bottleneck of the algorithm of [**?**] is the time for prefix matching which depends on the maximum length of all possible keys. Applying our algorithm to build an on-line hash table which contains all the keys presently in the priority queue, the time for prefix matching an arbitrary key can be reduced

to $O(\log\log n/H)$ by searching through the hash tables. As an example, given a priority queue S and a key x to be inserted, we use the precomputed hash table to search for the maximum prefix match for x in S. This effectively gives the position of key x in S. We then proceed to insert x into S which takes constant time once the position of x in the hash table is known. After key x is inserted into the trie, we use a pointer to link it to the key which is *left neighbor* of x had all the keys been explicitly sorted (i.e. the biggest key in all the keys that are smaller than x). The left neighbor can be identified in the process of searching. We keep a pointer pointing to the maximum key of the input keys. The Extract-Max operation can be performed in constant time by moving the pointer pointing to the maximum key to its left neighbor.

The time complexity of each operation does not depend on the length of the longest input key. Instead, since the running time depends on the length of the maximum prefix match for any key in the priority queue, the complexity is only related to the entropy of input keys. We have the following corollary.

Assume that the input keys are generated by a stationary and ergodic source satisfying mixing condition, the priority queue operations can be performed in $O(\log(\log n/H))$ expected sequential time where H is the entropy of input keys. □

4 Randomized Parallel Sorting Algorithm

The sequential sorting algorithm we presented was based on a divide and conquer strategy. That is, the problem is divided into subproblem after learning certain statistical properties of the input keys. The division of the subproblems is carefully carried out so that the subproblems are of even size. The subproblems are then solved independently and the results are combined together to form the overall solution of the problem. Therefore, this parallalism underlying this algorithm can be utilized to design an efficient parallel version of our algorithm which will be presented in this section.

There has been extensive research and also implementations in the area of parallel sorting. Reischuk [**?**] and Reif & Valiant [**?**] gave randomized sorting methods that use n processors and run in $O(\log n)$ time with high probability. The first deterministic method to achieve such performance was an EREW comparison PRAM algorithm based on the Ajtai-Komlós-Szemerédi sorting network. However, the constant factor in this time bound is still too large for practical use though it has an execution time of $O(\log n)$ using $O(n)$ processors. Later, Bilardi & Nicolau [**?**] achieved the same performance with a practical small constant. Cole [**?**] also has given a practical deterministic method of sorting on an EREW comparison PRAM in time $O(\log n)$ using $O(n)$ processors. Reif & Rajasekaran [**?**] presented an optimal $O(\log n)$ time PRAM algorithm using $O(n)$ processors for integer sorting where key length is $\log n$ and Hagerup [**?**] gave an $O(\log n)$ time algorithm using $O(n\log\log n/\log n)$ processors for integer sorting

with a bound of $O(\log n)$ bits per key.

List ranking problem is that given a singly linked list L with n objects, we wish to compute, for each object, the distance from the end of the list. The known lower bound for list-ranking is $\Omega(\log n/\log\log n)$ expected time (Beame & Hastad [**?**]) for all algorithms using polynomial number of processors on CRCW PRAM. If the results of the sorting algorithm are required to rank all the keys instead of just a relative ordering of them, this lower bound applies to the sorting problems as well. However, in many problems only the relative ordering information of the keys is needed and the sorting algorithm does not include a list-ranking procedure at the end, the sorting can be accomplished much faster. If this is the situation, the sorted keys can be loosely placed in the final list with empty places between any two of them. For example, [**?**] gives a *Padded Sort* (sorting n items into $n + o(n)$ locations) algorithm which requires only $\Theta(\log\log n)$ time using $n/\log\log n$ processors, assuming the items are taken from a uniform distribution.

We present a parallel version of our algorithm on CRCW PRAM model (see introduction for CRCW). We have the following theorem.

Let L_{max} be the number of bits of the longest sample key. If $L_{max} \leq n^{O(1)}$, we get $O(\log n)$ expected time using $O(n\log(\frac{\log n}{H})/\log n)$ processors for parallel sorting where H is the entropy of the input.

Proof: Assume that $L_{max} \leq n^{O(1)}$ and all keys are originally put in an array A_1 of length n.

The first part of our algorithm is indexing n keys to $|LD_s|$ buckets in parallel using $P = O(\frac{n\log\bar{L}}{\log n})$ processors. There are $\log\log n + 1$ stages which we will define below.

Stage 0: here we do random subsampling of size $n/\log n$ from the n input keys, build up the hash table for indexing, precompute all random mapping functions for $i = 1...\log\log n$,

$$R_i : [1...n_i] \rightarrow [1...n_i]$$

where $n_i = \frac{n}{(c_3)^i}$ for some constant c_3 which will be determined below. All those operations take $O(\log n)$ parallel time using $O(n/\log n)$ processors [**?**, **?**].

Stage 1: we arrange n keys in random order on an array with length n and randomly assign $O(\frac{\log n}{\log\bar{L}})$ keys to each processor. Each processor $j(1 \leq j \leq P)$ then works on keys from $(j-1)\frac{\log n}{\log\bar{L}}$ to $j\frac{\log n}{\log\bar{L}}$ assigned to it separately for $c_0\log n$ time where c_0 is a constant greater than 1. In particular, each processor indexes keys to buckets using the same technique described in our sequential algorithm. We can show at the end of this time there will be at most a constant factor decrease in the number of all keys left unindexed, say n/c_1 where $c_1 > 1$. Then we use the precomputed random mapping function R_1 to map all keys to another array A_1 with the same size of n which takes $\log n/\log\bar{L}$ time using P processors. We now subdivide A_1 into segments each with size $c_2\log n$. Then a standard prefix computation is performed within each segment to delete the

already indexed keys which takes $O(\log\log n)$ parallel time using n processors, so we can slow it down to $O(\log n)$ time using $n\log\log n/\log n$ processors. Define $c_3 = c_2/(1+\epsilon)$ where ϵ is a given constant say $1/3$. The size of array A_1 now becomes n/c_3. Then we assign processor j keys from $(j-1)\frac{\log n}{c_3 \log \bar{L}}$ to $j\frac{\log n}{c_3 \log \bar{L}}$. To bound the number of unindexed keys per processor, we have the following lemma :

The number of unindexed keys for each processor after random mapping is at most $\frac{c_2}{c_1}\log n(1+\epsilon)$ with probability $1 - 1/n^{\Omega(1)}$ for some small constant $\epsilon(0 < \epsilon < 1/2)$ and large c_2.

Proof: As the mapping function is random, the number U of unindexed keys falling into a given segment in A_2 is binomial distributed with probability $1/c_1$ and mean $\mu = \frac{c_2}{c_1}\log n$. Applying Chernoff Bounds (see [**?**] and also introduction), we have

$\Pr(U \geq \frac{c_2}{c_1}(1+\epsilon)\log n) \leq e^{-\frac{\epsilon^2\mu}{2}} = \frac{1}{n^{\Omega(1)}}$

for large c_2 and given ϵ. □

Stage i: now we apply all operations similar to those in stage 1 on the array A_{i-1}. We assume A_{i-1} has length $n/(c_3)^{i-1}$ and contains only remaining keys still to be indexed. Each of these stages takes $\frac{c_0}{(c_3)^{i-1}}\log n$ time using P processors. For example, now the prefix sum can be done in $O(\log\log n)$ time using $\frac{n}{(c_3)^{i-1}}$ processors. So slowing down the time to $\frac{O(\log n)}{(c_3)^{i-1}}$, we only need $n\log\log n/\log n$ processors which is less than P. All other time bounds are achieved similarly.

Repeat this stage until the number of keys left unindexed becomes $n/\log n$ which is less than P. Then we can finally assign each key a processor to finish the job. The final work takes $O(\log n)$ time using P processors as long as $L_{max} \leq n^{O(1)}$.

It is easy to see that the number of stages required is $\log\log n$. The total parallel time complexity of these stages then is

$$O(\Sigma_{i=0}^{\log\log n}(\frac{1}{(c_3)^i}c_0\log n)) \leq O(\log n)$$

.

Next, we sort each bucket separately using a standard comparison based parallel sorting algorithm. Each bucket i of size B_i takes $O(\log B_i)$ time using $O(B_i)$ processors. As each bucket size is bounded by $O(\log n/H)$ and $\Sigma_{i=1}^{|LD_s|}B_i = n$, this sorting of buckets takes $O(\log(\log n/H))$ time using $O(n)$ processors. To relax the time bound to $O(\log n)$, we require $O(n\frac{\log(\log n/H)}{\log n})$ processors which is upper bounded by P.

Finally, we use standard list-ranking algorithm to rank each key which takes only $O(\log n)$ time using $O(n/\log n)$ processors (see [**?**]).

The initial construction of the dictionary D_s takes $O(\log n)$ time using P processors by similar techniques. Thus, the total time is $O(\log n)$ using P

Figure 3: Two dimensional convex hull.

processors. □

5 Application: Convex Hull Problem

In this section, we apply our sorting algorithm to a computational geometry problem - the convex hull problem. We show that the efficient sorting algorithm can improve the performance of convex hull algorithm with certain statistical assumptions on the inputs.

The convex hull problem is defined as follows.

Convex hull problem: Given a set of points in E^d (the Euclidean d-dimensional space), find the smallest polygon containing these points.

The sequential convex hull problem takes $\Omega(n \log n)$ operations for n points in two dimensions and there are algorithms with matching bound. However, It is well known that the two dimensional convex hull problem can be solved in linear time if the points are presorted by their x coordinates. Hence we have the following corollary if we apply our sorting algorithm to sort the x-coordinates of the input points:

Given a set of points, the two dimensional convex hull problem can be solved in $O(n(\log(\log n/H)))$ sequential time where H is the entropy of the x-coordinates of the input points.

Since the sequential algorithm is straightforward, we are interested in designing an efficient parallel algorithm for the convex hull problem, using some well known techniques in parallel algorithm design. Below we present a parallel version of the algorithm for solving the two dimensional convex hull problem. The input is given as a set of n points represented by their polar coordinates (r, θ). We also assume that the set of points has a random distribution on the angular coordinate θ while the radial coordinate r has an arbitrary distribution. Also we assume that the points have been presorted by their angular coordinates. The parallel machine model we use is CRCW PRAM.

First we choose independently $O(\log n)$ sets of random samples each of size $O(n^\epsilon)$ (ϵ is a small constant, say 1/3) and apply the *polling* technique of [**?**] to determine a good sample which divides the set into $O(n^{1/3})$ sections (with respect to θ) each with $O(n^{2/3})$ points with probability $\geq 1 - n^{-\alpha}$ where α is a constant. The polling is done by chosing $O(log^2 n)$ subsamples and testing for a good sample on a fraction $n/(\log n)^{O(1)}$ of the inputs and it can be proven that by applying the polling technique a good sample of the inputs can be obtained with high probability (refer to [**?**] for further details).

Within each section, the convex hull is constructed by applying recursively the merging procedure which we will describe below. This merging procedure is also applied to construct the final convex hull. The convex hull for the base

Figure 4: Finding common tangent in a refining process.

case where each section contains less than three points can be constructed in constant time by a linear number of processors. We now describe the merging procedure (after constructing the convex hull of each section) in detail as follows:

Let ϵ' be another small constant, say 1/9. We assign $n^{\epsilon'}$ processors for each pair of sections to find their common tangent. The total number of processors used is $(n^{1/3})^2 n^{\epsilon'} = O(n)$. As the problem of finding common tangent between any pair of sections are virtually the same, we focus our attention on only one pair of sections, say section i and j. However, all the common tangents between pairs of sections are to be found in parallel. Finding the common tangent between section i and j is carried out in a constant number of steps, precisely $\frac{2/3}{\epsilon'}$ steps. In the first step, we divide both sections into $O(n^{\epsilon'})$ subsections and solve the problem of finding the common tangent between these two sections after we replace each part by the line segment linking its two endpoints (the points with minimum and maximum value of θ in the subsection). This problem can be solved in constant time using the method discussed in [**?**] which finds the common tangent in $O(1)$ parallel time using a linear number of processors.

This common tangent (between subsection k in section i and subsection l in section j) we found is not exactly the one we need but very close because the real common tangent (between section i and j) must go through subsection k and subsection l by the convex property (see Figure **??**).

Then subsection k and l are further divided into $n^{\epsilon'}$ parts. The size of the problem is now reduced to $O(n^{2/3-\epsilon'})$. In the following steps, we apply the same method ($O(1)$ time for each step) to refine the common tangent until we find the one needed. The total time is still a constant.

After we find all the common tangents between pairs of sections, we construct the convex hull by allocating edges of convex hull in a counter-clockwise direction. For each section, find the common tangent with the minimum counter-clockwise angle with respect to the zero-degree line of the polar system. This can be accomplished in $O(1)$ time by assigning each section $O(n^{2/3})$ processors to find the minimum-angle common tangent on a CRCW PRAM [**?**].

So the merging step takes only $O(1)$ time and $O(n)$ processors. The time complexity for the parallel algorithm is therefore

$$\bar{T}(n) = \bar{T}(n^{2/3}) + O(1) = O(\log \log n).$$

In each recursive step, the expected number of processors required is always $O(n)$. As the work in each recursive step is bounded by $O(n)$ with high probability, the total work for the parallel algorithm is thus $O(n \log \log n)$ with high probability $\geq 1 - 1/n^{\Omega(1)}$.

The above algorithm can be summarized as follows:

Given a set of two dimensional points represented by their polar coordinates

(r, θ) and assuming that the angular coordinates are presorted, the convex hull of n points can be constructed in $O(\log \log n)$ expected time using $O(n)$ processors.

We immediately have the following corollary by applying our parallel sorting algorithm to sort the angular coordinates of the input points:

Given a set of two dimensional points represented by their polar coordinates (r, θ) and let H be the entropy of the angular coordinates, the convex hull of n points can be constructed in $O((\log(\log n/H)))$ expected time using $O(n)$ processors on CRCW PRAM. □

6 Related Work

Following our entropy based sorting algorithm, Andersson and Nilsson [**?**] later presented a radix sort algorithm with improved time complexity. Their algorithm analyzes the complexity of the algorithm in terms of a more general measurement, *distinguishing prefixes*. Specifically, their algorithm sorts a set of n binary strings in $\Theta(n \log(\frac{B}{n \log n} + 2))$ time, where B is the number of all distinguishing prefixes, i.e., the minimum number of bits that have to be inspected to distinguish the strings.

The distinguishing prefix is a more general measurement of difficulty of sorting than the entropy of the source from which the inputs are generated. For a stationary ergodic process satisfying a certain *mixing condition* (see [**?**] for details), the following equation holds:

$$\lim_{n \to \infty} \frac{E(\bar{B})}{\log n} = \frac{1}{H}.$$

Therefore, for any algorithm where the complexity can be expressed in terms of $\bar{B}$, the complexity can be expressed in terms of entropy H as well. The reverse is not true for general inputs. Therefore, the approach analyzing the complexity of an algorithm in terms of distinguishing prefix is more general than one using the entropy of a stationary ergodic source.

However, it is noteworthy that their computational model used in the radix sort is different from ours in that their model assumes that rearranging the binary strings (the input keys) can be all accomplished by moving pointers. Also, more importantly, their algorithm depends on the assumption that the expected length of distinguishing prefix of each key is bounded by a constant machine words, i.e. $w = \Omega(\bar{B})$, while our sorting algorithm does not depend on this assumption. In fact, in our algorithm, the expected length of distinguishing prefix is $O(\log n/H)$ rather the length of a constant machine words. If we impose such assumptions, our algorithm runs in $O(n)$ expected time.

7 Summary

In this chapter we investigated the first problem, sorting problem, in our study of applying data compression techniques to improve efficiency of algorithms. After a survey of previous work on sorting problem, we introduced a new randomized sorting algorithm with certain statistical assumptions based on trie data structure and binary string processing widely used in data compression algorithms. The algorithm was implemented and showed good performance comparing with system sorting routine. In addition, we gave a parallel version of the randomized algorithm in the hope that the divide and conquer nature of our algorithm can lay a solid ground for a highly parallel implementation. Finally we extend our method to convex hull problem to reduce the overall time complexity of the algorithm to $O(\log(\log n/H))$.

Our primary motivation of the study of the sorting problem is not trying to design a fastest sorting algorithm. Instead, we concentrated on studying the problem of how to apply data structures and associated computational techniques to design efficient sorting algorithm and how to evaluate the validity of certain statistical assumptions on the input data and utilize these assumptions to aid algorithm design. Also we presented several results in parallel settings because the divide-and-conquer nature of our algorithm makes a parallel implementation extremely attractive.

In the next chapter, we will study another fundamental algorithmic problem - string matching. The motivation to study string matching is the sheer importance of this problem and its difference from the sorting algorithm. As sorting is an important operation in scientific computations, string matching is crucial in many areas such as data processing, information retrieval and database design. Furthermore, unlike the sorting problem where the inputs are a set of independent keys, the string matching problem regards the input as a consecutive data stream. Thus a different approach in handling the input data is needed to design new algorithms based on data compression ideas. Instead of applying certain data structure in design of new algorithms, we will emphasize on how to analyze the time complexity of a new algorithm based on statistical assumptions of the inputs.

References

[1] A. Amir and G. Benson. Efficient two dimensional compressed matching. *Proc. of Data Compression Conference, Snow Bird, Utah*, pages 279-288, Mar 1992.

[2] A. Amir and G.Benson and M.Farach. Witness-free dueling: The perfect crime! (or how to match in a compressed two-dimensional array). submitted for publication, 1993.

[3] A. Amir and G. Benson and M. Farach. Let sleeping files lie: pattern matching in Z-compressed files. *Symposium on Discrete Algorithms*, 1994.

[4] A. Andersson and S. Nilson. A new efficient radix sort. *35th Symposium on Foundations of Computer Science*, 714–721, 1994.

[5] S. Azhar and G. Badros and A. Glodjo and M.Y. Kao and J.H. Reif. Data Compression Techniques for Stock Market Prediction. *Data Compression Conference*, 1994.

[6] A.V. Aho and M.J. Corasick. Efficient string matching: and aid to bibliographic search. *Communications of the ACM*, Vol 18, 6:333-340, 1975.

[7] T.C. Bell, J.G. Cleary and I.H. Witten, *Text Compression,* Prentice Hall Publisher, 1990.

[8] P. Beam and J. Hastad, Optimal bounds for decision problems on the CRCW PRAM, *J.ACM,* 36:643-670.

[9] P. Billingsley, *Ergodic Theory and Information.* John Wiley & Sons, New York 1965.

[10] P. Van Emade Boas, R. Kaas and E. Zulstra, Design and implementation of an efficient priority queue, *Math. Systems. Theory,* 10:99-127.

[11] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith and M. Zagha, A comparison of sorting algorithms for the Connection Machine CM-2, *3rd Annual ACM Symposium on Parallel Algorithms and Architecture,* July 21-24, 1991.

[12] A. Bookstein and S.T. Klein and T. Raita and I.K. Ravichandra Rao and M.D. Patil, Can Random Fluctuation Be Exploited In Data Compression, Data Compression Conference, 1993.

[13] R.S. Boyer and J.S. Moore, A fast string searching algorithm, *Communications of the ACM,* 20(10):762–772, 1977.

[14] G. Bilardi and A. Nicolau, Bitonic sorting with $O(N \log N)$ comparsions, *Proc.20th Ann. Conf. on Information Science and Systems,* Princeton, NJ(1986)309-319.

[15] D. Bino and V. Pan. *Polynomial and matrix computations*, Vol 1, Birkhauser.

[16] R. Board and L. Pitt, On the Necessity of Occam Algorithms. *Proceedings of the 22nd Annual ACM Symposium on Theory of Computation*, May 1990, pp 54-63.

[17] A. Borodin and P. Tiwari. On the Decidability of Sparse Univariate Polynomial Interpolation, *Proc. 22nd Ann. ACM Symp. on Theory of Computing*, 535–545, ACM Press, New York, 1990.

[18] R.L. Baker and Y.T. Tse. Compression of high spectral resolution imagery. In *Proceedings of SPIE: Applications of Digital Image Processing*, pages 255–264, August 1988.

[19] P. Cignoni and L.D. Floriani and C. Montani and E. Puppo and R. Scopigno, Multiresolution modeling and visualization of volume data based on simplicial complexes, *Proceedings of Symposium on Volume Visualization*, 1994.

[20] R.J. Clarke, *Transform coding of images*, Academic Press, 1985.

[21] K.L. Clarkson, Applications of random sampling in computational geometry ii. *Proc of the 4th Annual ACM Symp on Computational Geometry*, pages 1–11, 1988.

[22] A. Czumaj and Z. Galil and L. Gasieniec and K. Park and W. Plandowski, Work-Time-Optimal Parallel Algorithms for String Problems, *Syposium on Theoretical Computing, 1995.*

[23] R. Cole and R. Hariharan. Tighter bounds on the exact complexity of string matching. *33th Symposium on Foundations of Computer Science*, 600–609, 1992.

[24] K. Curewitz,P. Krishnan and J.S. Vitter. Practical Prefetching via Data Compression, *Proceedings of the 1993 SIGMOD International Conference on Management of Data (SIGMOD '93)*, Washington, D.C, May 1993, 257–266.

[25] M. Crochemore and L. Gasieniec and W. Rytter. Two-dimensional pattern matching by sampling. *Information Processing Letters*, 46:159–162, 1993.

[26] T.H. Cormen, C.E. Leiserson and R.L. Rivest, *Introduction to algorithms,* McGraw-Hill Book Company, 1990.

[27] R. Cole, Parallel merge sort, *SIAM J.Comput,* 17(1988)770-785.

[28] S. Chen and J. H. Reif. Using difficulty of prediction to decrease computation: Fast sort, priority queue and computational geometry for bounded-entropy inputs. *34th Symposium on Foundations of Computer Science*, 104–112, 1993.

[29] S. Chen and J. H. Reif. Fast pattern matching for entropy-bounded inputs. *Data Compression Conference*, 282–291, Snowbird, April, 1995.

[30] S. Chen and J.H. Reif. Survey on predictive computing. Duke Univ. Technical Report No. CS–1995–14, October, 1995.

[31] S. Chen and J.H. Reif. Efficient lossless compression for trees and graphs. *Data Compression Conference*, Snowbird, 1996.

[32] S. Chen and J.H. Reif. Adaptive compression models for non-stationary sources. Manuscript, 1996.

[33] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annuals of Mathematical Statistics,* 23:493–507, 1952.

[34] T.M. Cover and J.A. Thomas, *Elements of information theory,* John Wiley & Sons, 1990.

[35] J.G. Cleary and I.H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Tran. on Communications*, 32:396–402, 1984.

[36] V. Cuperman. *Efficient waveform coding of speech using vector quantization.* PhD thesis, University of California, Santa Barbara, Feb 1983.

[37] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. *Proceedings of the 1992 Workshop on Volume Visualization*, pages 91–106, Boston, 1992.

[38] D. Dudgeon and R.Mersereau. *Multidimensional Signal Processing.* Prentice-Hall, N.J., 1984, pp.81, 82, 363–383

[39] C. Derman, *Finite state Markov decision processes,* Academic Press, New York, 1970.

[40] P. Van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Information Processing Letters,* volume 3, No.3, 1977.

[41] M. Farach and M.'Noordewier and S. Savari and L. Shepp. On the Entropy of DNA: Algorithms and Measurements based on Memory and Rapid Convergence. *SODA*, 1995.

[42] T.R. Fischer and D.J. Tinnen. Quantized control with differential pulse code modulation. In *Conference Proceedings: 21st Conference on Decision and Control*, pages 1222-1227, December 1982.

[43] T.R. Fischer and D.J. Tinnen. Quantized control using differential encoding. *Optimal Control Applications and Methods*, pages 69-83, 1984.

[44] T.R. Fischer and D.J. Tinnen. Quantized control with data compression constraints. *Optimal Control Applications and Methods*, pages 593-596, May 1982.

[45] Z. Galil. A constant-time optimal parallel string-matching algorithm. *24th Symposium on Theory of Computation*, 69-76. 1992.

[46] R.G. Gallager. Variations on a theme by Huffman. *IEEE Trans. on Information Theory*, 24:668–674, 1978.

[47] J. Gil and Y. Matias, Fast hashing on a PRAM - Designing by expectation, *Proc. 2nd. Ann. ACM Symp. on Discrete Algorithms.*

[48] Z. Galil and Kunsoo Park. An improved algorithm for approximate string matching. *SIAM J. Comput.*, Vol 19, 6:989-999, December 1990.

[49] Z. Galil and J. Seiferas, Time-space-optimal string matching. *Journal of computer and System Sciences*, 26(3):280-294, 1983.

[50] R.M. Gray. Vector Quantization. *IEEE ASSP Magazine*, 1:4-29, April 1984.

[51] R.M. Gray. *Entropy and information theory.* Sringer-Verlag, 1990.

[52] R.M. Gray. *Source coding theory.* Kluwer Academic Publishers, 1990.

[53] S. Gupta and A. Gersho. Feature predictive vector quantization of multispectral images, *Trans. on Geoscience & remote sensing*, 30(3):491-501, May 1992.

[54] T. Hagerup, Towards optimal parallel bucket sorting, *Inform. and Comput.,* 75(1987)39-51.

[55] T. Hagerup and C.RÜB, A guided tour of Chernoff bounds, *Information Processing Letter* 33(1989/90)305-308, North-Hollend.

[56] M.C. Harrison. Implementation of the substring test by hashing. *Communications of the ACM*, 14:777-779, 1971.

[57] G. Held. *Data compression,* John Wiley & Sons, 1991.

[58] K.H. Holm. Graph matching in operational semantics and typing. In *Proceedings of Colloquium on Trees in Algebra and Programming*, pages 191–205, 1990.

[59] J.S. Huang and Y.C. Chow, Parallel sorting and data partitioning by sampling, *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference,* 627-631, November 1983.

[60] W.L. Hightower, J.F. Prins and J.H. Reif, Implementations of randomized sorting on large parallel machines, *Symposium on Parallel Algorithm and Architecture, 1992.*

[61] D.A. Huffman, A method for the construction of minimum-redundancy codes, *Proc. Institute of Electrical and Radio Engineering, 40* (9),1098-1101, September.

[62] J.JáJá, *An introduction to parallel algorithm,* Addison-Wesley, 1992.

[63] R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. Technical Report TR-31-81, Aiken Computation Laboratory, Harvard University, 1981.

[64] D.E. Knuth, *The art of computer programming,* Volume 3, Sorting and searching, Addison-Wesley, Reading, 1973.

[65] E. Karatsuba and Y.N. Lakshman and J.M. Wiley, Modular Rational Sparse Multivariate Polynomial Interpolation, *Proc. ACM-SIGSAM Int. Symp. on Symb. and Alg. Comp. (ISSAC '90)*, 135–139, ACM Press, New York, 1990.

[66] D.E. Knuth and J.H. Morris and V.R. Pratt. Fast pattern matching in strings. *SIAM J. Comput*, 8:323-350, 1977.

[67] A.R. Karlin, S.J. Philips and P. Raghavan, Markov paging, *Thirty-third Annual Symposium on Foundations of Computer Science,* Pittsburgh, Pennsylvania,1992.

[68] P. Krishnan and J.S. Vitter, Optimal Prefetching in the worst case, *Proceedings of the 5th Annual SIAM/ACM Symposium on Discrete Algorithms*, Alexandria, VA, January 1994.

[69] M. Levoy. Display of surface from volume data, *IEEE Computer Graphics and Applications*, 8(3):29–37, 1988.

[70] J.S. Lim. *Two Dimensional Image and Signal Processing* Prentice-Hall, 1990.

[71] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Computer Graphics*, 28(4):451–458, 1994.

[72] T. Linder, G. Lugosi, and K. Zeger. Universality and rates of convergence in lossy source coding. *preliminary draft*, 1992.

[73] M. Li and P.M.B. Vitanyi. *An introduction to Kolmogorov complexity and its applications* Springer-Verlag, New York, Second Edition, 1997.

[74] M. Li and P.M.B. Vitanyi. On prediction by data compression, *Proc. 9th European Conference on Machine Learning*, Lecture notes in artificial intelligence, Vol.XXX, Springer-Verlag, Heidelberg, 1997.

[75] M. Li and P.M.B. Vitanyi. Kolmogorov complexity arguments in combinatorics, *J. Comb. Th.,*, Series A, 66:2(1994), 226-236.

[76] T. Malzbender. Fourier volume rendering. *ACM Transactions on Graphics*, 12(3):233–250, July 1993.

[77] M.R. Nelson, LZW data compression, *Dr.Dobb's Journal,* October 1989.

[78] H. Mannila, Measures of Presortedness and Optimal Sorting Algorithms, *IEEE Trans. Computers,* 318-325, 1985

[79] S. Muthukrishnan and K. Palem. Highly efficient dictionary matching in parallel. *extended abstract*, 7:51–75, 1992.

[80] T. Markas and J. Reif. Multispectral image compression algorithms, In *Proceedings of 3rd Annual Data Compression Conference*, pages 391–400, April 1993.

[81] T. Markas and J. Reif. Image compression methods with distortion control capabilities. In *Proceedings of 1st Annual Data Compression Conference*, pages 121–130, April 1991.

[82] P.D. MacKenzie and Q.F. Stout, Ultra-Fast Expected Time Parallel Algorithms, *SODA Conference,* 1991.

[83] Y. Matias and U. Vishkin, On parallel hashing and integer sorting, *Proc. of 17th ICALP, Springer LNCS 443,* 729-743, 1990.

[84] Y. Matias and J.S. Vitter and W.C. Ni. Dynamic Generation of Discrete Random Variates, *Proceedings of the 4th Annual SIAM/ACM Symposium on Discrete Algorithms*, January 1993.

[85] V.S. Miller and M.N. Wegman, Variations on a theme by Ziv and Lempel, *Combinatorial algorithms on words,* edited by A.Apostolico and Z.Galil, 131-140, NATO ASI Series, Vol. F12., Springer-Verlag, Berlin.

[86] M. Palmer and S. Zdonik, "Fido: A Cache that Learns to Fetch," *Proceedings of the 1991 International Conference on Very Large Databases*, September 1991.

[87] B. Pittel, Asymptotical growth of a class of random trees, *The Annals of Probability,* 1985, Vol 13, No.2. 414-427.

[88] W.B. Pennebaker and J.L. Mitchell. *JPEG still image data compression standard*. Van Nostrand Reinhold, 1992.

[89] P. Mcllroy, Optimistic Sorting and Information Theoretic Complexity, *SPAA 93,* 467-474, 1993.

[90] P. Raghavan, A statistical adversary for on-line algorithms, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, Vol 7, 1992.

[91] M.O. Rabin, Probabilistic algorithms, *in J.F. Traub, ed. Algorithms and Complexity,* pages 21–36, 1976.

[92] M. Rodeh, V.R. Pratt, and S. Even. Linear algorithm for data compression via string matching. *J. of ACM*, 28:1:16–24, 1981.

[93] S. Rajasekaran and J.H. Reif, Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM J.Comput.,* 18:594-607, 1989.

[94] S. Rajasekaran and S. Sen, On parallel integer sorting, *Acta Informatica,* 29, 1-15(1992).

[95] T. Raita and J. Teuhola. Predictive text compression by hashing. New Orleans, LA.

[96] J. Reif, An optimal parallel algorithm for integer sorting, *Proc.26th Ann. IEEE Symp. on Foundations of Computer Science* (1985)496-504.

[97] J.H. Reif and S. Sen, Optimal randomized parallel algorithms for computational geometry. *Proc. of the 16th International conference on Parallel Processing,* 1987.

[98] J.H. Reif and L.G. Valiant, A logarithmic time sort for linear size networks, *Proc. 21st Ann. ACM Symp.on Theory of Computing* (1989)264-273.

[99] R. Reischuk, A fast probabilistic sorting algorithm, *SIAM J.Comput.,* 14(1985)396-409.

[100] F. Rubin. Experiments in text file compression. *Communication of the ACM*, 19:11:617–623, 1976.

[101] P. Sabella. A rendering algorithm for 3d scalar fields. *Computer Graphics,* 22(4):51–58, 1988.

[102] S. Sen, *Random sampling techniques for efficient parallel algorithms in computational geometry*, Ph.D thesis, Duke University, 1989.

[103] R. Solovay and V.Strassen. A fast monte-carlo test for primality, *SIAM Journal of Computing,* 84–85, 1977.

[104] J.A. Storer, *Data compression: methods and theory,* Computer Science Press, Rockvill, Maryland, 1988.

[105] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules, *Communications of the ACM,* Vol 28, No.2, pp. 202-208, February 1985.

[106] W. Szpankowski, A typical behavior of some data compression schemes, *Data Compression Conference,* 1991.

[107] W. Szpankowski, (Un)expected behavior of typical suffix trees, *Data Compression Conference,* 1991.

[108] Y. Shiloach and U. Vishkin, Finding the maximum, merging, and sorting in a parallel computation model, *Journal of Algorithms* 2:88-102(1981).

[109] C. Upson and M. Keeler. V-buffers: Visible volume rendering. *Computer Graphics,* 22(4):59–64, 1988.

[110] J.S. Vitter and P. Krishnan, Optimal prefetching via data compression, In *Journal of ACM*, 43(5), September, 1996. A shortened version appeears in *Thirty-second Annual IEEE Symposium on Foundations of Computer Science*, 1991.

[111] L.A. Westover. Splatting: a parallel, feed-forward volume rendering algorithm, Ph.D thesis, University of North Carolina, Department of Computer Science, 1991.

[112] L. Westover, Footprint evaluation for volume rendering, *Computer Graphics,*, Volume 24, Number 4, August 1990.

[113] I.H. Witten and T.C. Bell and H. Emberson and S. Inglis and A. Moffat, Textual image compression: two-stage lossy/lossless encoding of textual images, *Proceedings of the IEEE*, No.6, 1994.

[114] J. Wilhelms and A.V. Gelder. Muti-dimensional trees for controlled volume rendering and compression, *IEEE Proceedings of Symposium on volume visualization*, 1994.

[115] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Trans. Information Theory,* 23, 3, 337-343(1977).

[116] J. Ziv. Coding theorems for individual sequences, *IEEE Trans. Information Theory,* 24, 405-412(1978).

[117] A.C.C. Yao. The complexity of pattern matching for a random string. *SIAM J.Comput*, 8:3:368–387, 1979.