

Using Difficulty of Prediction to Decrease Computation: Fast Sort, Priority Queue and Convex Hull on Entropy Bounded Inputs *

Shenfeng Chen and John H. Reif
Department of Computer Science
Duke University
Durham, NC 27708

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,Krishnan91], [Karlin,Philips,Raghavan92], [Raghavan92] for use of Markov models for on-line algorithms, e.g., caching and prefetching). Their results used the fact that compressible sources are predictable (and vice versa), and showed 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 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

*Email addresses: reif@cs.duke.edu and chen@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.

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 an *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 also give an application of our sorting algorithm to 2-D convex hull problem proving the same parallel complexity bound for this problem as for sorting.

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.

1 Introduction

1.1 Sorting and Priority Queues

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. A *priority queue*[3, 18] is a data structure representing a single set S on which the instructions INSERT, DELETE and MIN can be executed *on-line* (i.e., in some arbitrary order and such that each instruction should be executed before reading the next one).

Sorting is of great practical importance: it has been estimated that 20% of the computing work on mainframes is *sorting*. Therefore, even a factor of 2 improvement will have a large impact in practice. Though the theoretical research

has already had large impact, there are still problems remaining: the computational model, problem definition and algorithms must be refined to

- fit current architectures and allow us to give high performance implementations on real machines.
- also, proposed 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 important since most high performance machines have parallelism - even the CRAY can be viewed as a large vector machine).

We believe that the above questions are still not completely answered in spite of very large body of literature on sorting, particularly for someone interested in actual implementation. Next, we summarize previous approaches for sorting and priority queues problems and then describe our pragmatically oriented approach.

1.2 Computational Models

The *Comparison tree* model, though an excellent model for theoretical analysis, has a drawback that it requires a $\Omega(\log n)$ lower bound per key for sorting (even if the keys are uniformly distributed). On the other hand, the unit cost RAM has the advantage that it is known to be the reasonable model (excepting hierarchical memory) for current sequential architectures, and furthermore, as we will see below, it can go below $O(\log n)$ time per key with appropriate and reasonable assumptions. Therefore, we will adopt unit cost RAM in this paper.

1.3 Deterministic vs. Randomized

Furthermore, randomized sorts, such as Quick Sort which was initially considered only of theoretic interest, are used on many if not most system sorts for large inputs. Later parallel variants, such as FLASHSORT [32] and SAMPLESORT[14], have given some of the best implementations for parallel sorting (see [15] and [4] respectively). So we feel it is reasonable to adopt the randomized RAM model in this paper.

1.4 Approaches Assuming Bounded Number of Bits L per Key

Radix sort and sorting algorithms based on trie structure assume bounded number of bits L per key. Radix sort [18] requires $O(L/\log n)$ sequential work per key and trie approach [3, 18] requires $O(\log L)$ sequential work per key. However, the assumption of bounded L 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 .

1.5 Worse Case Input vs. Random Input

The advantage of assuming worse case input is that the sorting time bound can be guaranteed. Using bucket techniques [18], sorting each key needs only $O(1)$ expected time per key if we assume uniform distribution of the input. But this assumption as well does not seem to hold very frequently in practice (see below 1.6 for a refutation). Then we may

ask: is there anything we can say about large files occurring in practice? As computer scientists, we would like to prove a theorem applicable to files found in practice (see [22] for a different approach of an optimal sorting algorithm by measuring presortedness of input).

1.6 Using Entropy Estimates to Determine Facts About Large Files Occurring in Practice

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 section 2.1). There is a large body of careful and thorough empirical tests to back this up. For example, the normal lossless compression ratio is 3-4 for English text, 1.5-20 for system files and generally no more than 20 for files containing business and financial records [1, 35]. Lossy compression ratio can be larger. What do these compression bounds say about the probability distribution of files in practice?

(i) We conclude that it is not reasonable to assume uniformly distributed sets of keys (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.

(ii) However, due to large body of empirical tests, we know that entropy in practice is nearly always bounded by a relatively small constant (i.e., $H \geq 1/20$).

1.7 Sorting and Priority Queue Algorithm with Assumption of Bounded Entropy

The above observation leads us to the question: can we use the empirically established fact (ii) that entropy is bounded in almost all practically occurring files?

As our computational model, we assume (as justified above) randomized unit cost RAM. In order to have general application (i.e., data base join applications), our algorithm must not depend on the number of bits per key.

We use randomized sampling technique to learn an approximation of probability distribution of inputs. A fast data structure combining hash table and trie is set up using the sampled distribution (see details in section 2.4).

Theorems 1,2,3 and 4 are our main results (see later sections).

1.8 Organization of This Paper

Section 2 discusses about entropy bounded input sets. Section 3 gives a description of our sequential algorithm and analyze the complexity. Our parallel sorting algorithm is stated in section 4. In section 5, we present an application to computational geometry problem: a parallel algorithm for 2-D convex hull.

2 Entropy Bounded Sets of Keys

2.1 Entropy

Here we wish to argue that an important aspect of key probability distribution is the compressibility of the input keys. The input keys are uniformly distributed iff they are non-compressible. Thus, if keys are noncompressible (i.e., $\rho = 1$),

they are uniformly distributed and it is known [18] that they can be easily sorted by bucket sort in linear time. Normally, the purpose of data compression is to remove redundancy from data so that it takes less time to transmit and less space to store. This is not what we intend to do. Instead we will use techniques borrowed from data compression so as to learn the probability distribution and speed up key searching.

To see how entropy is related to bucket sort, let us review some basic concepts in information theory. *Entropy* [1, 8] is one of the fundamental ideas. It is a measure of the amount of order or redundancy in a given message. Entropy is large when there is a lot of disorder and small when there is a lot of order. However, entropy can only be defined relative to a given model which estimates in a certain way the probabilities of different messages. For example, consider a set of possible events with probabilities p_1, p_2, \dots, p_n that sum to 1. As demonstrated in his pioneering work, Shannon showed that the only function suitable for describing entropy in this case is

$$H(p_1, p_2, \dots, p_n) = -\sum_{i=1}^n p_i \log_2 p_i. \quad (1)$$

In the analysis of data compression techniques, another equivalent definition of entropy is often used which is the inverse of the optimal data compression ratio, or the ratio of the output file generated after optimal compression to the original size of input file F submitted for data compression,

$$H(F) = \frac{1}{\rho_{opt}} = \frac{\text{number of bits of optimal output}}{\text{number of bits of input}} \quad (2)$$

It is well known that the entropy of a stationary ergodic information source is intimately related to coding and certain data compression schemes, most notably the universal compression scheme of Lempel and Ziv (see [8, 40, 41, 21]), which gives optimal compression for large input files.

2.2 Relating Compression Ratio to Entropy

There are various approaches to data compression, e.g., *Huffman coding* [16] and *textual compression* [1]. Huffman coding estimates a probability for each character and chooses codes based on this probability. Textual compression achieves compression by replacing groups of consecutive characters with indices into some dictionary. The dictionary is a list of phrases that are expected to occur frequently. The indexes are chosen so that on average they take less space than the phrase they replace, thereby achieving compression.

Ziv-Lempel coding [40, 41] is a general class of dictionary compression methods using adaptive schemes. The central idea of LZW coding is that better compression could be obtained by replacing a repeated string by a reference to an earlier occurrence. We will assume here the later version (known as LZW [25]) when dictionary is represented as a suffix tree. This dictionary is constructed incrementally from a nearly empty dictionary $\{0,1\}$, by finding in the text the longest prefix match of the remaining text that matches a word in the dictionary (which will correspond to a current leaf of the tree), and then adding to the dictionary a string consisting of this match concatenated with the next character in the text (i.e., creating a new leaf just below the current matching leaf).

LZW compression methods outperform any other dictionary coding schemes and can achieve optimal compression

ratio when the number of keys is large. It is also known and proved in [40, 41, 8] that LZW gives optimal compression, i.e., for a large file F_n of size n generated by a stationary and ergodic source (see next section), the compression ratio ρ limits to $1/\text{Entropy}(F_n)$ for $n \rightarrow \infty$.

2.3 Relating Entropy to \bar{L}

Consider the problem of optimal data compression of a given input source. Suppose the input source is a stationary and ergodic sequence built over a binary alphabet $\{0,1\}$. "Ergodicity" means that as any sequence produced in accordance with a model grows longer and longer, it becomes entirely representative of the entire model. Thus, by the definition of stationary and ergodic sequence, the entropy of any given piece of sequence is not changing over time.

Let $x_i, i = 1 \dots n$ be the input keys which are assumed to be generated by a stationary and ergodic source. Define F_n to be the input file of keys x_1, x_2, \dots, 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 when we consider compressing F_n .

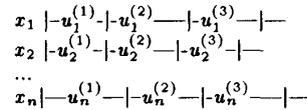


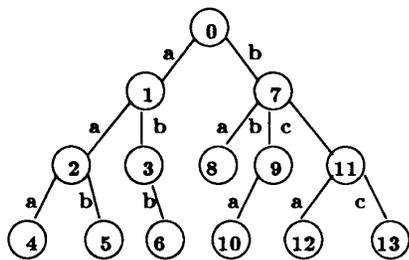
Figure 1: Stationary and ergodic sequence.

In Figure 1, x_i denotes the i th input key with arbitrary length and u_j^i denotes the j th prefix match of the i th key.

Let \bar{L} be the average length of the first prefix match, i.e., $\bar{L} = E(u_i^{(1)})$. The optimal LZW code for n keys has $\log n$ bits [36]. Thus, the expected optimal compression ratio when we consider the sequence of concatenating $u_1^{(1)}, u_2^{(1)}, u_3^{(1)} \dots$ is $\frac{\bar{L}}{\log n}$ [25]. However, as the whole input is a stationary and ergodic sequence, this compression ratio is also $\frac{\bar{L}}{\log n}$ when we consider compressing the input file F_n . Thus, we have the following lemma:

Lemma 1 For large n as $n \rightarrow \infty$, $H = \text{Entropy}(u_i^{(1)}) = \text{Entropy}(F_n)$ for each i .

2.4 Data Structures: Trie and Hash Table



Original code : abbcbb
 Encoded code : 6ccb

Figure 2: Trie in dictionary encoding.

Trie and *hash table* are two important data structures in dictionary encoding techniques. They are also the main structures used in our algorithm.

A *trie* [18] is a multiway tree with a path from the root to a unique node for each string represented in the tree. Figure 2 gives an example of a trie indexing a string. In a trie, only the prefix of each string is stored so a longest match can be found by tracing the tree until no match is found, or the path ends at a leaf.

An efficient implementation of trie is a single hash table [18] with an entry for each node. To determine the location of the child of the node at location n in the table for input character c , the hash function is supplied with both n and c . The hash function which maps the prefix string to a position of the hash table must be carefully selected so that the collision is minimized.

The trie structure can be used to perform priority queue operations. [3] has shown that the priority queue operation requires $O(\log(L_{max}))$ expected time per key where L_{max} is the length of the longest input key. Although it had apparently not been previously noted, we note that if the input keys are taken from a uniform distribution, the expected sequential time per operation becomes $O(\log \bar{L})$ where \bar{L} is the average length of the input keys. Tries will be used in our algorithm for construction of suffix trees and fast indexing.

3 Our Entropy Bounded Sequential Sorting Algorithm

3.1 Sketch of Our Algorithm

We assume the bit values of each input key are generated by the same stationary and ergodic source which is initially unknown. The distribution of the input is learned by subsampling from the input and building a dictionary using trie structure from the sampled keys. Then all input keys are indexed to buckets associated with the leaves of the dictionary. A probabilistic analysis (Lemma 2) 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 (in Theorem 1) 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 $\rho \leq (\log n)^{O(1)}$.

3.2 Detailed Description of Our Sequential Sorting Algorithm

We present a randomized algorithm for sorting n keys x_1, x_2, \dots, x_n .

First we randomly choose a set S of $m = n/\log n$ sample keys from the input and list them in random order. We will apply a modification of the usual LZW compression algorithm to construct a dictionary D_s from the first prefixes of the sample keys in S as described just below. Once D_s is constructed (and represented by a trie search data structure [3]), we will consider each key x_i . Let $x_i = u_i^{(1)}u_i^{(2)}\dots$ be the LZW prefix decomposition of x_i using the dictionary D_s . Thus $u_i^{(1)}$ is the longest prefix of x_i that is in the dictionary D_s .

Let a *leaf* of D_s be an element of D_s which is not a strict prefix of any other element of D_s . Let LD_s be the leaves of 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 $d_i = \text{length of } Index(x_i)$. Both $u_i^{(1)}$ and $Index(x_i)$ can be found in $O(\log d_i)$ sequential time by using trie search techniques.

It still remains for us to describe how to build D_s . D_s will be represented by a trie structure. 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 sample key x_{s_i} (considered at random order) in the current constructed dictionary, and then adding the concatenation of $u_{s_i}^{(1)}$ with first bit of $u_{s_i}^{(2)}$ to the dictionary. This is different from the standard LZW scheme as we stop processing this key now and go on to the next. Each step takes time $O(\log |u_{s_i}^{(1)}|) = O(\log d_{s_i})$.

For any key x_i , we can find the $Index(x_i)$ for x_i in $O(\log d_i)$ time in a trie data structure where d_i is the length of the index of x_i (see [3]). As in bucket sort, the buckets are sorted separately and then chained by the order of sample keys.

A sequential version of the algorithm is described as follows:

Input n keys x_1, x_2, \dots, x_n , generated by a stationary ergodic source.

Output sorted n keys.

Step 1

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

Step 2

2.1 Randomly permute $S = \{x_{s_1}, x_{s_2}, \dots, x_{s_m}\}$.

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), $\text{Bucket}[w_j] \leftarrow \emptyset$.

Step 3

Indexing using D_s .

for each key $x_i, i = 1, \dots, n$ do

3.1 Use trie search [3] to find the $Index(x_i)$ (where index is defined above).

3.2 Insert x_i into the corresponding $\text{Bucket}[Index(x_i)]$ associated with the index of x_i .

Step 4

- 4.1 Sort leaves of D_s using a standard sorting algorithm.
- 4.2 Sort elements within each bucket using standard sorting algorithm.
- 4.3 All buckets are linked under the order of the sorted leaves.

We have the following lemma:

Lemma 2 *The size of each resulting bucket is $O(\log n/H)$ with probability $\geq 1 - \frac{1}{n^{\Omega(1)}}$ for large n .*

Proof: Recall that LD_s is defined to be the leaves of D_s . In [36] it is shown that $\exists c, c'$ such that any $y \in LD_s$ has probability between $c/|LD_s|$ and $c'/|LD_s|$ when $|D_s| \rightarrow \infty$.

The result of [36] also implies that for each key x_i , with the same distribution, for each $y \in LD_s$, \exists constants c'' and c''' ,

$$c''/|LD_s| \leq \text{Prob}(y = \text{Index}(x_i)) \leq c'''/|LD_s|.$$

Furthermore, [36, 37] shows that $|LD_s|$ limits to $H|D_s|$ for large $|D_s|$. Note that $|D_s| = |S|$. Thus the number of keys in

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

is expected to be $O(\log n/H)$. \square

3.3 Analysis of Complexity

Theorem 1 *The sorting algorithm takes $O(n \log(\frac{\log n}{H}))$ sequential expected time where H is the entropy of the input keys.*

Proof: Step 1 takes $O(S)$ time. In step 3, Define average index length

$$\bar{L} = \frac{\sum_{i=1}^n d_i}{n}$$

and compression ratio

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

Searching the index of x_i takes time $O(\log d_i)$ where d_i is the length of $\text{Index}(x_i)$. Inserting key x_i into the corresponding bucket takes $O(1)$ time. So the time complexity for step 3 is $\sum_{i=1}^n \log(d_i) = \sum_{i=1}^n \log(d_i) = \log \prod_{i=1}^n d_i \leq \log \bar{L}^n = n \log \bar{L} = n \log(\rho \log n) = n \log(\frac{\log n}{H})$. Similarly, the time for step 2 is also bounded by $n \log(\log n/H)$. 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 $O(u_i \log u_i)$ where u_i is the size of the i th bucket. Note that

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

By Lemma 2, the size of each bucket is bounded by $O(\log n/H)$ with high likelihood, the expected time complexity of step 4 therefore is

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

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

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

Proof: Straightforward. \square

3.4 Experimental Testing of Sequential Algorithm

We implemented the sequential version of our sorting algorithm on SPARC-2 machine. We tested the algorithm on 20 files with 24,000 to 1,500,000 keys from a wide variety of sources, whose compression ratio ranged from 2-4. We also derived an empirical formula for the runtime bound of our algorithm. For comparison, we also did this for the UNIX system sorting routine - quick sort.

It is well known 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 section 3.3, the corresponding empirical formula for our sequential algorithm with fixed sample size is: $T_e = d_0 + d_1 \log \log N$.

We determined the constants using the experimental results :

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

We found that the experimental results fit well with the empirical formulas and predict well the cutoff with quicksort. By extrapolation from these empirical tests we found our sequential sorting algorithm will beat quicksort when $N > 32,000,000$.

3.5 Priority Queue Operations

Our sequential algorithm can be easily modified to perform priority queue operations: INSERT, DELETE and MIN. The time complexity of each operation does not depend on the length of the longest input key. Instead, it is only related to the entropy of input keys. We have the following theorem.

Theorem 2 *After subsampling $O(n/(\log n)^{O(1)})$ keys, priority queue operations can be performed in $O(\log(\log n/H))$ expected sequential time where H is the entropy of input keys.*

Proof: This is only a natural extension of our sequential sorting algorithm. \square

4 Parallelization of Our Algorithm

Reischuk [34] and Reif & Valiant [32] 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 the time bound is still too large for practical use though it has an execution time of $O(\log n)$ using $O(n)$ processors. Bilardi & Nicolau [5] achieve the same performance with a practical small constant. Cole [7] has given a practical deterministic method of sorting on an EREW comparison PRAM in time $O(\log n)$ using $O(n)$ processors. Reif & Rajasekaran [29] gave an optimal $O(\log n)$ time PRAM algorithm using $O(n)$ processors

for integer sorting where key length is $\log n$ and Hagerup [12] gave an $O(\log n)$ time algorithm using $O(n \log \log n / \log n)$ processors for integer sorting with $O(\log n)$ bits/key.

The known lower bound for list-ranking is $\Omega(\log n / \log \log n)$ expected time (Beame & Hastad [2]) for all algorithms using polynomial number of processors on CRCW PRAM. However, in many problems only the relative ordering information is needed from sorting which does not include a list-ranking procedure and can be accomplished much faster. For example, [23] gives a *Padded Sort* (sort n items into $n + o(n)$ locations) algorithm which requires $\Theta(\log \log n)$ time using $n / \log \log n$ processors, assuming the items are taken from a uniform distribution.

We now present a parallel version of our algorithm on CRCW PRAM model. We have the following theorem.

Theorem 4 *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 $P = O(\frac{n \log L}{\log n})$ 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 $i = 1 \dots \log \log n$

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

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 [24, 17].

Stage 1: we arrange n keys in random order on an array with length n and randomly assign $O(\frac{\log n}{\log L})$ keys to each processor. Each processor j ($1 \leq j \leq P$) then works on keys from $(j-1) \frac{\log n}{\log L}$ to $j \frac{\log n}{\log L}$ 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 $\log n / \log L$ 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 $(j-1) \frac{\log n}{c_3 \log L}$ to $j \frac{\log n}{c_3 \log L}$. To bound the number of unindexed keys per processor, we have the following lemma :

Lemma 3 *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 ϵ ($0 < \epsilon < 1/2$)*

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 [13]), we have

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

for large c_2 and given ϵ . \square

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

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

which is simply $O(\log n)$.

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 $\sum_{i=1}^{|LD_s|} B_i = n$, 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 [17]).

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 processors. \square

5 Application: 2-D Convex Hull

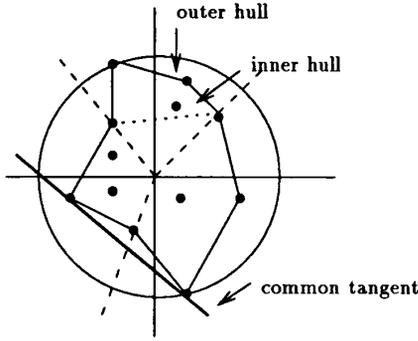


Figure 3: 2-D Convex Hull.

It is well known that the 2-D convex hull problem can be solved in linear time if the points are presorted by their x coordinates [19]. Hence we have the following corollary if we apply our sorting algorithm to sort the x -coordinates of the input points:

Corollary 2 *Given a set of points, the 2-D 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.*

Here we present a parallel 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 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.

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 [33] 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 (polling is done by choosing $O(\log^2 n)$ subsamples and testing for a good sample on a fraction $n/(\log n)^{O(1)}$ of the inputs, for details refer to [33]).

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 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 $\frac{2}{\epsilon'}$ 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 [17] 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 (Figure 4).

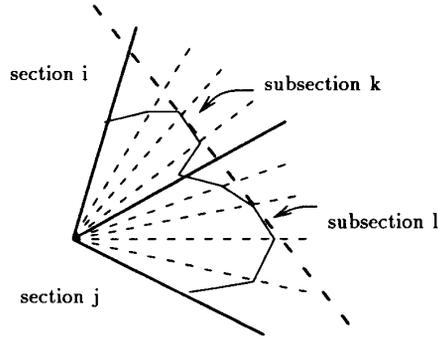


Figure 4: Finding common tangent in a refining process.

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 [6].

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

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

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:

Theorem 3 *Given a set of 2-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 sorting algorithm to sort the angular coordinates of the input points:

Corollary 3 Given a set of 2-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 $O(n(\log(\log n/H)))$ expected time using $O(n)$ processors on CRCW PRAM. \square

6 Conclusion

Studies have indicated that sorting comprises about 20% of all computing on mainframes. Perhaps the largest use of sorting in computing (particularly business computing) is the sort required for large database operations (e.g., required by joint operations). In these applications the keys are many words long. 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 that case the cutoff is much lower. In case that the compression ratio is high, which can be determined after building the dictionary, we just adopt the previous sorting algorithm, e.g., quick sort. The same techniques can be extended to other problems (e.g., computational geometry problems) to decrease computation by learning the distribution of the inputs.

References

- [1] T.C.Bell, J.G.Cleary and I.H.Witten, *Text Compression*, Prentice Hall Publisher, 1990.
- [2] P.Beam and J.Hastad, Optimal bounds for decision problems on the CRCW PRAM, *J.ACM*, 36:643-670.
- [3] P.Van Emde Boas, R.Kaas and E.Zulstra, Design and implementation of an efficient priority queue, *Math. Systems Theory*, 10:99-127.
- [4] 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.
- [5] G.Bilardi and A.Nicolau, Bitonic sorting with $O(N \log N)$ comparisons, *Proc.20th Ann. Conf. on Information Science and Systems*, Princeton, NJ(1986)309-319.
- [6] T.H.Cormen, C.E.Leiserson and R.L.Rivest, *Introduction to algorithms*, McGraw-Hill Book Company, 1990.
- [7] R.Cole, Parallel merge sort, *SIAM J.Comput.*, 17(1988)770-785.
- [8] T.M.Cover and J.A.Thomas, *Elements of information theory*, John Wiley & Sons, 1990.
- [9] C.Derman, *Finite state Markov decision processes*, Academic Press, New York, 1970.
- [10] 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.
- [11] J.Gil and Y.Matias, Fast hashing on a PRAM - Designing by expectation, *Proc. 2nd. Ann. ACM Symp. on Discrete Algorithms*.
- [12] T.Hagerup, Towards optimal parallel bucket sorting, *Inform. and Comput.*, 75(1987)39-51.
- [13] T.Hagerup and C.Rüb, A guided tour of Chernoff bounds, *Information Processing Letter* 33(1989/90)305-308, North-Holland.
- [14] 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.
- [15] W.L.Hightower, J.F.Prins and J.H.Reif, Implementations of randomized sorting on large parallel machines, *SPAA Conference*, 1992.
- [16] D.A.Huffman, A method for the construction of minimum-redundancy codes, *Proc. Institute of Electrical and Radio Engineering*, 40 (9),1098-1101, September.
- [17] J.JáJá, *An introduction to parallel algorithm*, Addison-Wesley, 1992.
- [18] D.E.Knuth, *The art of computer programming*, Volume 3, Sorting and searching, Addison-Wesley, Reading, 1973.
- [19] K.Mulmuley, Computational geometry: an introduction through randomized algorithms, Prentice Hall, 1994.
- [20] A.R.Karlin, S.J.Philips and P.Raghavan, Markov paging, *Thirty-third Annual Symposium on Foundations of Computer Science*, Pittsburgh, Pennsylvania,1992.
- [21] M.R.Nelson, LZW data compression, *Dr.Dobb's Journal*, October 1989.
- [22] H.Mannila, Measures of Presortedness and Optimal Sorting Algorithms, *IEEE Trans. Computers*, 318-325, 1985
- [23] P.D.MacKenzie and Q.F.Stout, Ultra-Fast Expected Time Parallel Algorithms, *SODA Conference*, 1991.
- [24] Y.Matias and U.Vishkin, On parallel hashing and integer sorting, *Proc. of 17th ICALP, Springer LNCS 443*, 729-743, 1990.
- [25] 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.
- [26] B.Pittel, Asymptotical growth of a class of random trees, *The Annals of Probability*, 1985, Vol 13, No.2. 414-427.

- [27] P.McIlroy, Optimistic Sorting and Information Theoretic Complexity, *SPAA 93*, 467-474, 1993.
- [28] P.Raghavan, A statistical adversary for on-line algorithms, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, Vol 7, 1992.
- [29] S.Rajasekaran and J.H.Reif, Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM J.Comput.*, 18:594-607, 1989.
- [30] S.Rajasekaran and S.Sen, On parallel integer sorting, *Acta Informatica*, 29, 1-15(1992).
- [31] J.H.Reif, An optimal parallel algorithm for integer sorting, *Proc.26th Ann. IEEE Symp. on Foundations of Computer Science* (1985)496-504.
- [32] J.H.Reif and L.G.Valiant, A logarithmic time sort for linear size networks, *Journal of the ACM*, Vol. 34, No. 1, January 1987, pp. 60-76.
- [33] J.H.Reif and S.Sandeep, Optimal randomized parallel algorithms for computational geometry, *Algorithmica* (1992)7:91-117.
- [34] R.Reischuk, A fast probabilistic sorting algorithm, *SIAM J.Comput.*, 14(1985)396-409.
- [35] J.A.Storer, *Data compression: methods and theory*, Computer Science Press, Rockvill, Maryland, 1988.
- [36] W.Szpankowski, A typical behavior of some data compression schemes, *Data Compression Conference*, 1991.
- [37] W.Szpankowski, (Un)expected behavior of typical suffix trees, *Data Compression Conference*, 1991.
- [38] Y.Shiloach and U.Vishkin, Finding the maximum, merging, and sorting in a parallel computation model, *Journal of Algorithms* 2:88-102(1981).
- [39] J.S.Vitter and P.Krishnan, Optimal prefetching via data compression, *Thirty-second Annual IEEE Symposium on Foundations of Computer Science*, 1991.
- [40] J.Ziv and A.Lempel, A universal algorithm for sequential data compression, *IEEE Trans.Information Theory*, 23, 3, 337-343(1977).
- [41] J.Ziv, Coding theorems for individual sequences, *IEEE Trans.Information Theory*, 24, 405-412(1978).