Optimal and Sublogarithmic Time Randomized Parallel Sorting Algorithms[1]

Sanguthevar Rajasekaran[2]
John H. Reif[2]
Aiken Computation Laboratory, Harvard University, Cambridge, MA 02138.

Abstract. We assume a parallel RAM model which allows both concurrent reads and concurrent writes of a global memory.

Our main result is an optimal randomized parallel algorithm for *INTEGER_SORT* (i.e., for sorting n integers in the range $[1, n]$). Our algorithm costs only logarithmic time and is the first known that is *optimal*: the product of its time and processor bounds is upper bounded by a linear function of the input size. We also give a deterministic sub-logarithmic time algorithm for *prefix sum*. In addition we present a sub-logarithmic time algorithm for obtaining a random permutation of n elements in parallel. And finally, we present sub-logarithmic time algorithms for *GENERAL_SORT* and *INTEGER_SORT*. Our sublogarithmic *GENERAL_SORT* algorithm is also optimal.

Key words. Randomized algorithms, parallel sorting, parallel random access machines, random permutations, radix sort, prefix sum, optimal algorithms.

AMS(MOS) subject classifications. 68Q25.

[1]A preliminary version of this paper appeared as "An Optimal Parallel Algorithm for Integer Sorting" in the 18th IEEE Symposium on FOCS, Portland, Oregon, October 1985.

[2]Supported by NSF-DCR-85-03251 and ONR contract N00014-80-C-0647

RANDOMIZED PARALLEL SORTING

1 Introduction

1.1 Sequential Sorting Algorithms

Sorting is one of the most important problems not only of computer science but also of every other field of science. The importance of efficient sorting algorithms has been long realized by computer scientists. Many application programs like compilers, operating systems, etc. use sorting extensively to handle tables and lists. Both due to its practical value and theoretical interest, sorting has been an attractive area of research in computer science.

The problem of sorting a sequence of elements (also called *keys*) is to rearrange this sequence in either ascending order or descending order. When the keys to be sorted are *general*, i.e., when the keys have no known structure, a lower bound result [1] states that any sequential algorithm (on the random access machine (RAM) and many other sequential models of interest) will require at least $\Omega(n \log n)$ time to sort a sequence of n keys. Many *optimal* algorithms like QUICK_SORT, and HEAP_SORT whose run times match this lower bound can be found in the literature [1].

In computer science applications, more often, the keys to be sorted are from a finite set. In particular, the keys are integers of at most a polynomial (in the input size) magnitude. For keys with this special property, sorting becomes much simpler. If each one of the n elements in a sequence is an integer in the range $[1, n]$ we call these keys *integer keys*. The BUCKET_SORT algorithm [1] sorts n integer keys in $O(n)$ sequential steps. Notice that the run time of BUCKET_SORT matches the trivial $\Omega(n)$ lower bound for this problem.

In this paper we are concerned with randomized parallel algorithms for sorting both *general keys* and integer keys.

1.2 Known Parallel Sorting Algorithms

The performance of a parallel algorithm can be specified by bounds on its principal resources viz., processors, and time. If we let P denote the processor bound, and T denote the time bound of a parallel algorithm for a given problem, the product PT is, clearly, lower bounded by the minimum sequential time, T_s, required to solve this problem. We say a parallel algorithm is *optimal* if $PT = O(T_s)$. Discovering optimal parallel algorithms for sorting both *general* and integer keys remained an open problem for a long time.

Reischuk [25] proposed a randomized parallel algorithm that used n synchronous PRAM processors to sort n *general keys* in $O(\log n)$ time. This algorithm however is impractical owing to its large word-length requirements. Reif and Valiant [24] presented a randomized sorting algorithm that ran on a fixed connection network called *cube connected cycles* (CCC). This algorithm employed n processors to sort n *general keys* in time $O(\log n)$. Since $\Omega(n \log n)$ is a sequential lower bound for this problem, their algorithm is indeed optimal. Simultaneously, [4] discovered a deterministic parallel algorithm for sorting n *general keys* in time $O(\log n)$ using a sorting network of $O(n \log n)$ processors. Later Leighton [17] showed that this algorithm could be modified to run in $O(\log n)$ time on an n-node fixed connection network.

As in the sequential case, many parallel applications of interest need only to sort integer keys. Until now, no optimal parallel algorithm existed for sorting n integer keys with a run time of $O(\log n)$ or less.

1.3 Some Definitions and Notations

Given a sequence of keys $k_1, k_2, \ldots, k_n$ drawn from a set S having a linear order $<$, the problem of *sorting* this sequence is to find a permutation σ such that $k_{\sigma(1)} < k_{\sigma(2)} < \ldots < k_{\sigma(n)}$.

By *general keys* we mean a sequence of n elements drawn from a linearly ordered set S whose elements have no known structure. The only operation that can be used to gain information about the sequence is the comparison of two elements.

GENERAL_SORT is the problem of sorting a sequence of general keys, and *INTEGER_SORT* is the problem of sorting a sequence of integer keys.

Throughout this paper we let $[m]$ stand for $\{1, 2, \ldots, m\}$.

A sorting algorithm is said to be *stable* if equal elements remain in the same relative order in the sorted sequence as they were in originally. In more precise terms, a sorting algorithm is stable if on input $k_1, k_2, \ldots, k_n$, the algorithm outputs a sorting permutation σ of $(1, 2, \ldots, n)$ such that for all $i, j \in [n]$, if $k_i = k_j$ and $i < j$ then $\sigma(i) < \sigma(j)$. A sorting algorithm that is not guaranteed to output a stable sorted sequence is called *non-stable*.

Just like big-O function serves to represent the complexity bounds of deterministic algorithms, we employ $\widetilde{O}$ to represent complexity bounds of randomized algorithms. We say a randomized algorithm has resource (like time, space, etc.) bound $\widetilde{O}(g(n))$ if there is a constant c such that the amount of resource used by the algorithm (on any input of size n) is no more than $c\alpha g(n)$ with probability $\geq 1 - 1/n^{\alpha}$ for any $\alpha > 1$.

1.4 Our Model of Computation

We assume the CRCW PRAM model proposed by Shiloach, and Vishkin [26]. In a PRAM model, a number (say P) of processors work synchronously communicating with each other with the help of a common block of memory. Each processor is a RAM. A single step of a processor is an arithmetic operation, a comparison, or a memory access. CRCW PRAM is a version of PRAM that allows both concurrent writes and concurrent reads of shared memory. Write conflicts are resolved by priority.

All the algorithms given in this paper, except the prefix sum algorithm, are randomized. Every processor, in addition to the operations allowed by the deterministic version of the model, is also capable of making independent (n-sided) coin flips. Our stated resource bounds will hold for the worst case input with overwhelming probability.

1.5 Contents of this Paper

Our main contributions in this paper are:

1) an optimal parallel algorithm for INTEGER_SORT. This algorithm uses $n/\log n$ processors and sorts n integer keys in time $\widetilde{O}(\log n)$, and

2) sub-logarithmic time algorithms for GENERAL_SORT and INTEGER_SORT. GENERAL_SORT algorithm employs $n(\log n)^\epsilon$ (for any $\epsilon > 0$) processors and INTEGER_SORT algorithm employs $\frac{n(\log\log n)^2}{\log n}$ processors. Both these algorithms run in time $\widetilde{O}\left(\frac{\log n}{\log\log n}\right)$.

The problem of optimal parallel sorting of n integers in the range $[n^{O(1)}]$ still remains an open problem. Our sub-logarithmic time algorithm for GENERAL_SORT is optimal as implied by a recent result of Alon and Azar [2].

In our sub-logarithmic time sorting algorithms we reduce the problem of sorting to the problem of *prefix sum computation*. We show in this paper that *prefix sum* can be computed in time $O(\log n/\log\log(P\log n/n))$ using $P \geq n/\log n$ processors. We also present a sub-logarithmic time algorithm for computing a random permutation of n given elements with a run time of $\widetilde{O}(\log n/\log\log n)$ using $n(\log\log n)^2/\log n$ processors.

Some of the results of this paper appeared in preliminary form in [22], but are substantially simplified in this manuscript. In section 2 we present some relevant preliminary results. Section 3 contains our optimal INTEGER_SORT algorithm. In section 4 we describe our sub-logarithmic time algorithms.

2 Preliminary Results

2.1 Prefix Circuits

Let Σ be a domain and let $\circ$ be an associative operation that takes $O(1)$ sequential time over this domain. The *prefix computation problem* is defined as follows.

- **input** $(X(1), X(2), \ldots, X(n)) \in \Sigma^n$
- **output** $(X(1),\ X(1) \circ X(2),\ \ldots,\ X(1) \circ X(2) \circ \ldots \circ X(n))$.

The special case of prefix computation when Σ is the set of all natural numbers and $\circ$ is integer addition is called *prefix sum computation.* Ladner and Fischer [18] show that prefix computation can be done by a circuit of depth $O(\log n)$ and size n. The processor bound of this algorithm can be improved as follows.

Lemma 2.1 *Prefix computation can be done in time $O(\log n)$ using $n/\log n$ PRAM processors.*

Proof. Given $X(1), X(2), \ldots, X(n)$, each one of the $n/\log n$ processors gets $\log n$ successive keys. Every processor sequentially computes the prefix sum of the $\log n$ keys given to it in $\log n$ time. Let $S(i)$ be the sum of all the $\log n$ keys given to processor i (for $i = 1, \ldots, n/\log n$). Then, $n/\log n$ processors collectively compute the prefix sum of $S(1), S(2), \ldots, S(n/\log n)$, using Ladner and Fischer[18]'s algorithm. Using this prefix sum, each processor sequentially computes $\log n$ prefixes of the original input sequence. $\Box$

The above idea of processor improvement was originally used by Brent in his algorithm for expression evaluation, and hence we attribute lemma 2.1 to him. Recently Cole and Vishkin [9] have proved the following

Lemma 2.2 *Prefix sum computation of n integers ($O(\log n)$ bits each) can be performed in $O(\log n/\log\log n)$ time using $n\log\log n/\log n$ CRCW PRAM processors.*

2.2 An Assignment Problem

Given a set $Q = \{1, 2, \ldots, n\}$ of n indices. Each index belongs to exactly one of m groups $G_1, G_2, \ldots, G_m$. Let g_i stand for the number of indices belonging to group $G_i, i = 1, \ldots, m$. Given a sequence $N(1), N(2), \ldots, N(m)$ where $\sum_{i=1}^{m} N(i) = O(n)$ and $N(i)$ is an upper bound for $g_i, i = 1, 2, \ldots, m$. The problem is to find in parallel a permutation of $(1, 2, \ldots, n)$

in which all the indices belonging to G_1 appear first, all the indices belonging to G_2 appear next, and so on. (Assume that given an index i, the group $G_{i'}$ that i belongs to can be found in $O(1)$ time).

As an example, if $n = 5$, $m = 2$, $G_1 = \{2, 5\}$, $G_2 = \{1, 3, 4\}$, then $(5, 2, 1, 3, 4)$ and $(2, 5, 3, 1, 4)$ are (two of the) valid answers.

Lemma 2.3 *The above assignment problem can be solved in $\widetilde{O}(\log n)$ parallel time using $n/\log n$ PRAM processors.*

Proof. We present an algorithm. We use a shared memory of size $2\sum_{i=1}^{m} N(i)$ $(= L$, say). This memory is divided into m blocks $B_1, B_2, \ldots, B_m$ the size of B_i being $2N(i)$. A unique assignment for the indices belonging to G_i will be found in the block B_i, for $i = 1, 2, \ldots, m$.

Each one of the P $(= n/\log n)$ processors is given $\log n$ successive indices. Precisely, processor π is given the indices $(\pi - 1)\log n + 1$, $(\pi - 1)\log n + 2$, $\ldots$, $\pi \log n$, for $\pi = 1, 2, \ldots, P$. There are three phases of the algorithm. In the first phase boundaries of the m blocks are computed. In the second phase every processor sequentially finds unique assignments for the $\log n$ indices given to it in their **respective** blocks. In the third phase, a prefix sum computation is done to eliminate the unused cells and the position of each index in the output is read. Details follow.

step1

P processors collectively do a prefix sum of $(N(1), N(2), \ldots, N(m))$ and hence compute the boundaries of blocks in the common memory.

step2

Each processor π is given a total time of $d \log n$ (d being a constant to be fixed) to find assignments for all its indices sequentially.

π starts with its first index (call it) l. If $G_{l'}$ is the group that l belongs to, π chooses a random cell in $B_{l'}$ and tries to write its id in it. If the chosen cell did not contain the id of any other processor and π succeeds in writing, then that cell is assigned to l. The probability of success in one trial is $\geq 1/2$. If π has failed in this trial then it tries as many times as it takes to find an assignment for l and then it takes up the next index.

After $d \log n$ steps, even if there is a single processor that has not found assignments for all its keys, the algorithm is aborted and started anew.

step3

> Each processor π writes a 1 in the cells that have been assigned to its indices. Unassigned cells in the common memory will have 0's. P processors perform a prefix sum computation on the contents of the memory cells $(1, 2, \ldots, L)$. Finally, every processor reads out from the prefix sum the position of each one of its indices in the output.

Analysis. Steps 1 and 3 can be completed in $O(\log n)$ time in accordance with lemma 2.1.

In step2, the probability that a particular processor π successfully finds an assignment for one of its keys in a single trial is $\geq 1/2$. Let Y be the random variable equal to the number of successes of π in $d \log n$ trials.We require Y to be $\geq \log n$ for every processor. Clearly Y is lower bounded by a binomial variable (see appendix A for definitions) with parameters $(d \log n, 1/2)$. It follows from the Chernoff bounds (see appendix A, equation 3) that the probability that there will be at least a single processor which has not found assignments for all of its indices after $d \log n$ trials can be made $\leq n^{-\alpha}$ for any $\alpha \geq 1$, if we choose a proper constant d.
Therefore the whole algorithm runs in time $\widetilde{O}(\log n)$. This completes the proof of lemma 2.3. $\square$

It should be mentioned here that when the number of groups, m, is 1 the above algorithm outputs a random permutation of $(1, 2, \ldots, n)$. An algorithm for this special case was given by Miller and Reif [19].

2.3 Some Known Results

We state here the existence of optimal sequential algorithms for INTEGER_SORT and optimal parallel algorithms for GENERAL_SORT.

Lemma 2.4 *Stable INTEGER_SORT of n keys can be done in time $O(n)$ by a deterministic sequential RAM [1].*

Lemma 2.5 *GENERAL_SORT of n keys can be performed in time $O(\log n)$ using n PRAM processors ([4] and [8]).*

3 An Optimal INTEGER_SORT Algorithm

In this section we present an optimal algorithm for INTEGER_SORT. This algorithm employs $n/\log n$ processors and runs in time $\widetilde{O}(\log n)$.

3.1 Summary of the Algorithm

The main idea behind our algorithm is radix sorting [15]. As an example of radix sorting, consider the problem of sorting a sequence of two-bit decimal integers. One way of doing this is to sort the sequence with respect the least significant bits (LSB) of the keys and then to sort the resultant sequence with respect to the most significant bits (MSB) of the keys. This will work provided, in the second sort keys with equal MSBs will remain in the same relative order as they were in originally. In otherwords, the second sort should be stable.

Given a sequence of keys $k_1, k_2, \ldots, k_n \in [n]$, where each key is a $\log n$-bit integer. We first (non-stable) sort this sequence with respect to the $(\log n - 3\log\log n)$ LSBs of the keys. (Call this sort *Coarse_Sort*). In the resultant sequence we apply a stable sort with respect to the $3\log\log n$ MSBs of the keys. (Call this sort *Fine_Sort*).

Even though the sequential time complexity of stable sort is no different from that of non-stable sort, it seems that parallel stable sort is inherently more complex than parallel non-stable sort. This is the reason why we have divided the bits of the keys unevenly.

In Coarse_Sort we need to (non-stable) sort a sequence of n keys, each key being in the range $[1, n/\log^3 n]$ and, in Fine_Sort we have to (stable) sort n keys in the range $[1, \log^3 n]$. In terms of notations our algorithm can be summarized as follows.

Let $D = n/\log^3 n$ and $k'_i = \lfloor k_i/D \rfloor$ and $k''_i = k_i - k'_i * D$ for all $i \in [n]$.

Coarse_Sort. Sort $k''_1, k''_2, \ldots, k''_n \in [D]$. Let σ be the resultant permutation.

Fine_Sort. Stable-sort $k'_{\sigma(1)}, k'_{\sigma(2)}, \ldots, k'_{\sigma(n)} \in [\log^3 n]$. Let ρ be the resultant permutation.

Output. The permutation $\rho.\sigma$, the composition of ρ and σ.

In sections 3.2 and 3.3 we describe Fine_Sort and Coarse_Sort respectively.

3.2 Fine_Sort

We give a deterministic algorithm for Fine_Sort. First we will show how to stable-sort n keys in the range $[\log n]$ using $n/\log n$ processors in time $O(\log n)$ and then apply the idea of radix sorting to prove that we can stable-sort n keys in the range $[(\log n)^{O(1)}]$ within the same resource bounds.

Lemma 3.1 *n keys $k_1, k_2, \ldots, k_n \in [\log n]$ can be stable-sorted in $O(\log n)$ time using $P = n/\log n$ processors.*

Proof. In Fine_Sort algorithm, each processor π is given $\log n$ successive keys. Each one of the P processors starts by sequentially stable-sorting the keys given to it. Then, collectively, the P processors group all the keys with equal values. (There are $\log n$ groups in all). Finally, they output a rearrangement of the given sequence in which all the 1's (i.e., keys with a value 1) appear first, all the 2's appear next, and so on. Throughout the algorithm the relative order of equal keys is preserved. More details follow.

To each processor $\pi \in [P]$ we assign the key indices $J(\pi) = \{j | (\pi - 1)\log n < j \leq min(n, \pi \log n)\}$. There are three steps in the algorithm.

step1

Each processor π sequentially stable-sorts the keys $\{k_j | j \in J(\pi)\}$ in time $O(\log n)$ (see lemma 2.4), and hence constructs $\log n$ lists $J_{\pi,k} = \{j \in J(\pi) | k_j = k\}$ for $k \in [\log n]$. Elements in $J_{\pi,k}$ are ordered in the same relative order as in the input.

step2

The P processors collectively perform the prefix sum of

$$
\begin{array}{l}
(|J_{1,1}|, |J_{2,1}|, \ldots, |J_{P,1}|, \\
|J_{1,2}|, |J_{2,2}|, \ldots, |J_{P,2}|, \\
\ldots \\
|J_{1,q}|, |J_{2,q}|, \ldots, |J_{P,q}|)
\end{array}
$$

where $q = \log n$. Call this sum

$$
\begin{array}{l}
(S_{1,1}, S_{2,1}, \ldots, S_{P,1}, \\
S_{1,2}, S_{2,2}, \ldots, S_{P,2}, \\
\ldots \\
S_{1,q}, S_{2,q}, \ldots, S_{P,q}).
\end{array}
$$

step3

> Each processor π sequentially computes the position of each one of its keys in the output using the prefix sum. The position of keys in the list $J_{\pi,l}$ will be $S_{\pi-1,l}+1, S_{\pi-1,l}+2, \ldots, S_{\pi,l}$.

Analysis. It is easy to see that steps 1 and 3 can be performed within the stated resource bounds. Step 2 also can be completed within the stated resource bounds as stated in lemma 2.1. $\square$

Lemma 3.2 *If n keys in the range $[R]$ (for any $R = n^{O(1)}$) can be stable-sorted in $O(\log n)$ time using $P = n/logn$ processors, then n keys $k_1, k_2, \ldots, k_n \in [R^2]$ can be stable-sorted in time $O(\log n)$ using the same number of processors.*

Proof. Let $k_i' = \lfloor k_i/R \rfloor$ and $k_i'' = k_i - k_i' * R$ for every $i \in [n]$. First, stable-sort $k_1'', k_2'', \ldots, k_n''$ obtaining a permutation σ. Then stable-sort $k'_{\sigma(1)}, k'_{\sigma(2)}, \ldots, k'_{\sigma(n)}$ obtaining a permutation ρ. Output $\rho.\sigma$. Clearly both these sorts can be completed in time $O(\log n)$ using P processors. $\square$

Lemmas 3.1 and 3.2 immediately imply the following

Lemma 3.3 *n integer keys in the range $[(\log n)^{O(1)}]$ can be stable-sorted in time $O(\log n)$ using $n/\log n$ processors.*

3.3 Coarse_Sort

In this sub-section we fix a key domain $[D]$ where $D = n/\log^3 n$. We assume, w.l.o.g., $\log^3 n$ divides n. Let the input keys be $k_1, k_2, \ldots, k_n \in [D]$. Define the *index sequence* for each key $k \in [D]$ to be $I(k) = \{i | k_i = k\}$. The randomized algorithm for Coarse_Sort to be presented in this sub-section employs $P = n/\log n$ processors and runs in time $\widetilde{O}(\log n)$. The sorted sequence is non-stable.

The main idea is to calculate the cardinalities of the index sequences $I(k), k \in [D]$ approximately, and then to use the assignment algorithm of section 2.2 to rearrange the given sequence in sorted order.

Lemma 3.4 *Given as input $k_1, k_2, \ldots, k_n \in [D]$ we can compute $N(1), N(2), \ldots, N(D)$ in $\widetilde{O}(\log n)$ time using $P = n/\log n$ processors such that $\sum_{k\in[D]} N(i) = O(n)$ and furthermore, with very high likelihood $N(k) \geq |I(k)|$ for each $k \in [D]$.*

Proof. The following sampling algorithm serves as a proof.

step1

Each processor $\pi \in [D \log n]$ in parallel chooses a random index $s_\pi \in [n]$. Let S be the sequence $\{s_1, s_2, \ldots, s_{D \log n}\}$.

step2

The P processors collectively sort the keys with the chosen indices. That is, they sort $k_{s_1}, k_{s_2}, \ldots, k_{s_{D \log n}}$ and compute index sequences $I_S(k) = \{i \in S | k_i = k\}$ (for each $k \in [D]$).

step3

D of the P processors in parallel set $N(k) = d(\log^2 n)\ max(|I_S(k)|, \log n)$ for $k \in [D]$, d being a constant to be fixed in the analysis. Output $N(1), N(2), \ldots, N(D)$.

Analysis. Trivially, steps 1 and 3 can be performed in $O(1)$ time. Step 2 can be performed using any of the optimal GENERAL_SORT algorithms in $O(\log n)$ time (see lemma 2.5). (Notice that we have to sort only $n/\log^2 n$ keys in step2). It remains to be shown that $N(i)$'s computed by the sampling algorithm satisfy the conditions in lemma 3.4.

If $|I(k)| \leq d \log^3 n$, then always $N(k) \geq d \log^3 n \geq |I(k)|$. So suppose $|I(k)| > d \log^3 n$. Then it is easy to see that $|I_S(k)|$ is a binomial variable with parameters $(\frac{n}{\log^2 n}, \frac{|I(k)|}{n})$. The Chernoff bounds (see appendix A, equation 2) imply that for all $\alpha \geq 1$, there exists a c such that

$$\text{Prob.}\Big(|I_S(k)| \leq c\alpha |I(k)|/\log^2 n\Big) \leq \tfrac{1}{n^\alpha}.$$

Therefore, if we choose $d = (c\alpha)^{-1}$ then $N(k) \geq |I(k)|$ (for every $k \in [D]$) with probability $\geq 1 - n^{-\alpha}$. The Chernoff bounds (equation 3) also imply that for all $\alpha \geq 1$ there exists a h such that $N(k) \leq (h\alpha)|I(k)|$ (for every $k \in [D]$) with probability $\geq 1 - n^{-\alpha}$.

The bound on $\sum_{k\in[D]} N(k)$ clearly holds since

$$\begin{aligned}\sum_{k\in[D]} N(k) &\leq \sum_{k\in[D]} d \log^2 n [|I_S(k)| + \log n] \ = d \log^3 n D + d \log^2 n \sum_{k\in[D]} |I_S(k)| \\ &= dn + d \log^2 n D \log n \ = \ 2dn\end{aligned}$$

This concludes the proof of lemma 3.4. □

Having obtained the approximate cardinalities of the index sets, we apply the assignment algorithm of section 2.2. The set Q is the set of key indices viz., $\{1, 2, \ldots, n\}$. An index i belongs to group $G_{i'}$ if the value of the key with index i is i'. Under this definition, group G_j is the same as index sequence $I(j)$, $j = 1, 2, \ldots, D$. Since we can find approximate cardinalities of these groups (lemma 3.4), we can use the assignment algorithm of section 2.2 to rearrange the given sequence in sorted order. Thus we have the following

Lemma 3.5 *n keys $k_1, k_2, \ldots, k_n \in [D]$ can be sorted in time $\widetilde{O}(\log n)$ time using $n/\log n$ processors.*

Lemmas 3.3 and 3.5 together with the algorithm summary in section 3.1 prove the following

Theorem 3.1 *INTEGER_SORT of n keys can be performed in randomized $\widetilde{O}(\log n)$ time using $n/\log n$ CRCW PRAM processors.*

4 Sub-Logarithmic Time Algorithms

In the previous section we presented an optimal algorithm for INTEGER_SORT. In this section we will be presenting non-optimal sublogarithmic time algorithms for 1) prefix sum computation, 2) finding a random permutation of n elements, 3) GENERAL_SORT, and 4) INTEGER_SORT. Algorithms 3 and 4 are direct consequences of algorithms 1 and 2. Our prefix algorithm employs $P \geq n/\log n$ processors and runs in time $O(\log n/\log\log(P\log n/n))$. Algorithms 2,3, and 4 run in time $\widetilde{O}(\log n/\log\log n)$. GENERAL_SORT uses $n(\log n)^\epsilon$ processors and algorithms 2 and 4 use $n(\log\log n)^2/\log n$ processors.

4.1 A Sub-Logarithmic Prefix Algorithm

Given a sequence of integers $X(1), X(2), \ldots, X(n)$. We need to find the prefix sum of this sequence. This problem can be solved in sub-logarithmic time if we use more than $n/\log n$ processors as is stated by the following

Lemma 4.1 *Prefix sum computation can be performed in time $O(\log n/\log\log(P\log n/n))$ using $P \geq n/\log n$ CRCW PRAM processors.*

Proof. The algorithm can be summarized as follows. 1) Divide the given sequence into blocks of d (to be determined later) successive keys; 2) sequentially compute prefix sums in each block; 3)apply prefix to the final prefixes in each block; and 4) compute prefices in each block by using the result from 3 for the previous block.

More details follow. Let $n_1 = n/d$.

step1

In $O(d)$ time using $n_1 \leq P$ processors compute $X'(i,m), i \in [n_1], m \in [d]$, where $X'(i,m) = \sum_{j=(i-1)d+1}^{(i-1)d+m} X(j)$.

step2

Compute the prefix sum of the total sum of each part, i.e., compute $Y'(1), Y'(2), \ldots, Y'(n_1)$ where $Y'(i) = \sum_{j=1}^{i} X(j,d)$, for $i = 1, \ldots, n_1$.

step3

In time $O(d)$, using n_1 processors compute

$(X'(1,1), X'(1,2), \ldots, X'(1,d)$,
$Y'(1) \circ X'(2,1), Y'(1) \circ X'(2,2), \ldots, Y'(1) \circ X'(2,d)$,
$\ldots$
$Y'(n_1 - 1) \circ X'(n_1,1), Y'(n_1 - 1) \circ X'(n_1,2), \ldots, Y'(n_1 - 1) \circ X'(n_1,d)$

which is the required output.

Analysis. Clearly, steps 1 and 3 can be performed with n_1 processors in time $O(d)$. It remains to show that step 2 can be performed within the same time using P processors.

Let $C_{n,2}$ be a circuit of size n and in-degree 2 that computes the prefix sum of n elements in depth $O(\log n)$. Obtain an equivalent circuit $C_{n_2,b}$ of size $n_2 = n/b$ ($n_2 \geq n_1$) and in-degree b in the obvious way (by collapsing sub-circuits of height $\log b$ into single nodes starting from the bottom of the circuit [11]). We will simulate $C_{n_2,b}$.

Each input key is a $\log n$ bit integer. Each one of the keys is divided into d parts each comprising $\log n/d$ successive bits. The simulation proceeds in d stages. In the first stage, we input the $\log n/d$ least significant bits of the keys to the circuit $C_{n_2,b}$. In the second stage, we input the next most $\log n/d$ significant bits of the input keys to the circuit. Similarly we pipeline all the parts of the keys one part per stage. The computation in the circuit proceeds in a pipeline fashion.

At any stage, every node v of $C_{n_2,b}$ has to compute the sum of b integers that arrive at this node from its children and the carry it stored from the previous stage. v also has to store the carry from this stage to be used in the next. Each one of these b integers and the carry can be of at the most $2\log n/d \; = \; s$ bits. Therefore, the computation at v can be made to run in time $O(1)$ if we replace v by a constant depth circuit of size $b2^{s(b+1)}$. The depth of $C_{n_2,b}$ is $\log_b n_2$. Thus, the run time of the circuit (and hence the simulation time) will be $\log_b n_2 \; + \; O(d)$. The size of the circuit is $n_2 b 2^{s(b+1)}$.

We require $b2^{s(b+1)} \leq P/n_2, \; s = 2\log n/d, \; n_1 = n/d, \; n_1 \leq n_2$ and $\log_b n_2 = O(d)$. It is easy to see that choosing $s = \log\log(P\log n/n)$ will satisfy all the above constraints. This concludes the proof of lemma 4.1. $\square$

4.2 A Sub-Logarithmic Permutation Algorithm

The problem is to compute a random permutation of $(1, 2, \ldots, n)$ in sub-logarithmic parallel time. The algorithm presented in this sub-section is very similar to the assignment algorithm of section 2.2. It employs $P = n(\log\log n)^2/\log n$ processors and runs in time $\widetilde{O}(\log n/\log\log n)$.

A shared memory of size $2n$ is used. The main idea is to find unique assignments (in the common memory) for each one of the indices $i \in [n]$ and then to eliminate unused cells of common memory using a prefix sum computation. Processors are partitioned into groups of size $(\log\log n)^2$. Each group of $(\log\log n)^2$ processors gets $\log n$ successive indices. Detailed algorithm follows.

step1

> The $\log n$ indices given to each group of processors are partitioned into groups of size $(\log\log n)^2$. Step1 consists of $\log n/(\log\log n)^2$ phases. In the ith phase $(i = 1, 2, \ldots, \log n/(\log\log n)^2)$ each processor is given a distinct index from the group i of indices. Each processor spends $d\log\log n$ time (for some constant d) to find an assignment for its index (as explained in step2 of section 2.2). After $d\log\log n$ time the ith phase ends.

step2

> P processors perform a prefix sum computation to determine the number (call it N) of indices that do not have an assignment yet. Let $z = \lfloor P/N \rfloor$.

step3

> A distinct group of z processors in parallel work to find an assignment for every index j that remains without an assignment. A group succeeds even if a single processor in the group succeeds. Each group is given $C \log n / \log\log n$ time (for some constant C).
>
> After $C \log n / \log\log n$ time, even if a single index remains without an assignment the whole algorithm is aborted and started anew.
>
> (Grouping of processors in this step can easily be done using the prefix sum of step2).

step4

> Finally, P processors perform a prefix sum computation to eliminate unused cells and read the positions of their indices in the output.

Analysis. Consider the ith phase of step1. The probability that a given processor π succeeds in finding an assignment for its index in a single trial is $\geq 1/2$. Let Y be a random variable equal to the number of processors failing in the jth trial of phase i. Then Y is upperbounded by a binomial random variable with parameters $(N_i^j, 1/2)$ (where N_i^j is the number of processors that have not succeeded until the beginning of jth trial of phase i). (Note $N_i^1 = P$). The Chernoff bounds (equation 3) imply that Y is at the most a constant (< 1) fraction of N_i^j with probability $\geq 1 - 2^{-\epsilon N_i^j}$ (for some fixed $\epsilon < 1$). Therefore the number of unsuccessful processors at the end of phase i is $\widetilde{O}(P/\log n)$. The number of keys without assignments at the end of step1 is $\sum_{i=1}^{\log n/(\log\log n)^2} N_i^{d \log\log n}$. Using additive property of binomial distributions and the Chernoff bounds we conclude that the number of keys without assignments at the end of step1 is $O(n/\log n)$ (and hence $z = \Omega((\log\log n)^2)$) with probability $\geq 1 - n^{-\beta}$ for any $\beta \geq 1$.

Step2 runs in time $O(\log n/\log\log n)$ (lemma 2.2). In step3, probability that a particular group fails in one trial is $\leq (1/2)^{\Omega((\log\log n)^2)}$. This implies that the probability that there is at least one unsuccessful group at the end of step3 can be made $\leq n^{-\alpha}$, for any $\alpha \geq 1$, if we choose a proper C.

Thus we conclude that the whole algorithm will run successfully in time $\widetilde{O}(\log n/\log\log n)$. Clearly, this algorithm can also be used to solve the assignment probelem of section 2.2. Thus we have the following

Lemma 4.2 *The problem of computing a random permutation of n elements (and hence the assignment problem of section 2.2) can be solved in time $\widetilde{O}(\log n/\log\log n)$ using $P = n(\log\log n)^2/\log n$ processors.*

4.3 An Optimal Sub-Logarithmic GENERAL_SORT Algorithm

Given as input $k_1, k_2, \ldots, k_n$, Reischuk's algorithm [25] for GENERAL_SORT samples $\sqrt{n}$ keys at random. If $l_1, l_2, \ldots, l_{\sqrt{n}}$ are the sampled keys in sorted order, these keys divide the input keys into $p \le \sqrt{n}+1$ collections $S_1, S_2, \ldots, S_p$ where $S_1 = \{q | q \le l_1\}, S_i = \{q | l_{i-1} < q \le l_i\}$ for $i = 2, 3, \ldots, (p-1)$, and $S_p = \{q | q > k_{p-1}\}$. With very high likelihood [25], each one of these collections will be of size $O(\sqrt{n}\log n)$. (Reif and Valiant [24] give an algorithm for sampling $\sqrt{n}$ keys that will ensure that each one of these collections will be of size $O(\sqrt{n})$.) Having identified these collections, his algorithm sorts each one of them recursively and merges the results trivially.

As such, [25]'s algorithm requires a computer of word length $\Omega(\sqrt{n}\log n)$. This problem can be circumvented using the assignment algorithm of section 2.2. Moreover such a modified algorithm can be made sub-logarithmic if $n(\log n)^\epsilon$ processors are used. Detailed algorithm follows.

procedure sublogGS($\{k_1, k_2, \ldots, k_n\}$);

step1. If n is a constant sort trivially.

step2. $\sqrt{n}$ processors in parallel each sample a random key.

step3. Sort the $\sqrt{n}$ keys sampled in step2 by comparing every pair of keys and computing the rank of each key. This can be done in $O(\log n/\log\log n)$ time using n processors. Let the sorted sequence be $l_1, l_2, \ldots, l_{\sqrt{n}}$.

step4. Processors are partitioned into groups of size $(\log n)^\epsilon$. Each group gets an index $i \in [n]$. In parallel each group does a $(\log n)^\epsilon$-ary search on $l_1, l_2, \ldots, l_{\sqrt{n}}$ to find out the collection $S_{i'}$ that k_i belongs to.

step5. n processors collectively compute $N(1), N(2), \ldots, N(p)$ such that $\sum_{j=1}^{p} N(j) = O(n)$ and $N(j) \geq |S_j|$ for evey $j \in [p]$. (Recall $p \leq \sqrt{n} + 1$).

step6. n processors use the sub-logarithmic assignment algorithm of section 4.2 to rearrange $k_1, k_2, \ldots, k_n$ such that all the elements of S_1 will appear first, all the elements of S_2 will appear next, and so on.

step7. Recursively sort $S_1, S_2, \ldots, S_p$. Here $O(\sqrt{n}(\log n)^{\epsilon})$ processors work on each sub-problem. Finally output sublogGS(S_1), ..., sublogGS(S_p).

Analysis. If $T'(n)$ is the time sublogGS takes to sort n general keys, step1 and step2 take $O(1)$ time each. Step3, step4, and step6 take $\tilde{O}(\log n/\log\log n)$ time each. Step7 takes time $T'(c\sqrt{n})$ (for some constant c) with probability $\geq 1 - n^{-\alpha}$ (for any $\alpha \geq 1$). This is because no collection will be of size more than $O(\sqrt{n})$ with the same probability (if we employ Reif and Valiant [24]'s sampling algorithm). Computing $N(1), N(2), \ldots, N(p)$ (step5) can be done in time $\tilde{O}(\log n/\log\log n)$ using n processors using a sampling algorithm very similar to the one given in section 3.2. (details in appendix B). Therefore, the recurrence relation for $\overline{T}'(n)$, the expected value of $T'(n)$ can be written as

$$\overline{T}'(n) \leq \overline{T}'(c\sqrt{n}) + \tilde{O}(\log n/\log\log n) + \tilde{O}(n^{-\alpha})\,\overline{T}'(n - \sqrt{n} + 1)$$

By induction we can show that $\overline{T}'(n) \leq \tilde{O}(\log n/\log\log n)$. Thus we have the following

Theorem 4.1 *GENERAL_SORT can be done in time $\tilde{O}(\log n/\log\log n)$ with $n(\log n)^{\epsilon}$ CRCW PRAM processors.*

4.4 A Sub-Logarithmic Algorithm for INTEGER_SORT

In section 3, we presented an INTEGER_SORT algorithm that used $n/\log n$ processors to sort n integer keys in time $\tilde{O}(\log n)$. The same algorithm can be used to sort in time $\tilde{O}(\log n/\log\log n)$ if the number of processors used is $P = n(\log\log n)^2/\log n$. We will indicate here only the modifications needed to be made.

The P processors are partitioned into groups of size $(\log\log n)^2$ and each group is given $\log n$ successive indices. In Fine-Sort step1, each group of $(\log\log n)^2$ processors stable sorts the $\log n$ keys given to it using any of the parallel optimal stable GENERAL_SORT algorithms, in time $O(\log n/\log\log n)$. Step2 runs in time $O(\log n/\log\log n)$. In step3, each group of processors computes the position of each one of its $\log n$ keys in the output using the prefix sum of step2. The time needed for step3 is $\log n/(\log\log n)^2$.

In Coarse-Sort, while computing the $N(i)$'s, steps 1 and 3 run in time $O(1)$. In step2, we need to sort $n/\log^2 n$ keys. The sub-logarithmic algorithm of section 4.2 for GENERAL_SORT can be used to run step2 in time $\tilde{O}(\log n/\log\log n)$ using $< n/\log n$ processors. After computing $N(i)$'s, rearranging of the keys can be done using P processors in time $\tilde{O}(\log n/\log\log n)$ (lemma 4.2). Therefore, both Coarse-Sort and Fine-Sort run in time $\tilde{O}(\log n/\log\log n)$. Thus we have the following

Theorem 4.2 *INTEGER_SORT can be performed in* $\tilde{O}(\log n/\log\log n)$ *time using* $P = n(\log\log n)^2/\log n$ *CRCW PRAM processors.*

5 Conclusions

All the sorting algorithms appearing in this paper are non-stable. It remains an open problem to obtain stable versions of these algorithms. If we have a stable algorithm for INTEGER_SORT then the definition of integer keys can be extended to include integers in the range $[n^{O(1)}]$. Any deterministic algorithm for INTEGER_SORT using a polynomial number of CRCW PRAM processors will take at least $\Omega(\log n/\log\log n)$ time as has been shown by Beam and Hastad [6]. However it is an open question whether there exists a randomized CRCW PRAM algorithm that uses a polynomial number of processors and runs in time $o(\log n/\log\log n)$.

A recent result of Alon and Azar [2] implies that our sub-logarithmic time GENERAL_SORT algorithm is optimal. Their lower bound result is for a more powerful comparison tree model of Valiant and hence readily holds for PRAMs as well. Alon and Azar's theorem is that if P is the number of processors used, then the average time, T, required for sorting n elements by any randomized algorithm is $\Theta(\log n/\log(1+P/n))$ for $P \geq n$ and the average time is $\Theta(\log n/(P/n))$ for $P \leq n$. In particular, if $P = n(\log n)^{\epsilon}$, then $T = \Theta(\log n/\log\log n)$. It remains an open problem to prove or disprove the optimality of our sublogarithmic INTEGER_SORT algorithm.

Acknowledgements

The authors would like to thank Yijie Han, Sandeep Sen, and the referees for their insightful comments.

References

[1] AHO, HOPCROFT AND ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] N. ALON AND Y. AZAR, *The Average Complexity of Deterministic and Randomized Parallel Comparison Sorting Algorithms*, Proc. IEEE Symposium on Foundations of Computer Science, 1987, pp. 489-498.

[3] D. ANGLUIN AND L.G. VALIANT, *Fast Probabilistic Algorithms for Hamiltonian Paths and Matchings*, J. Comp. Syst. Sci., 18 (1979), pp. 155-193.

[4] M. ATAI, J. KOMLÓS AND E. SZEMERÉDI,*An $O(n \log n)$ Sorting Network*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 1-9.

[5] K. BATCHER, *Sorting Networks and Their Applications*, Spring Joint Computer Conf. 32, AFIPS Press, Montvale, N.J., 1968, pp. 307-314.

[6] P. BEAM AND J. HASTAD, *Optimal Bounds for Decision Problems on the CRCW PRAM*, 19th ACM Symposium on Theory Of Computing, 1987, pp. 83-93.

[7] H. CHERNOFF, *A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the Sum of Observations*, Annals of Math. Statistics 23, 1952, pp. 493-507.

[8] R. COLE, *Parallel Merge Sort*, Proc. 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 511-516.

[9] R. COLE AND U. VISHKIN, *Approximate and Exact Parallel Scheduling with Applications to List, Tree, and Graph Problems*, Proc. 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 478-491.

[10] W. FELLER, *An Introduction to Probability Theory and Its Applications*, vol.1, Wiley, New York, 1950.

[11] F.E. FICH, *Two Problems in Concrete Complexity Cycle Detection and Parallel Prefix Computation*, Ph.D. Thesis, Univ. of California, Berkeley,1982.

[12] W. HOEFFDING, *On the Distribution of the Number of Successes in Independent Trials*, Annals of Math. Stat. 27, 1956, pp. 713-721.

[13] J.E. HOPCROFT AND R.E. TARJAN, *Efficient Algorithms for Graph Manipulation*, Comm. ACM 16(6), 1973, pp. 372-378.

[14] N.J. JOHNSON AND S. KATZ, *Discrete Distributions*, Houghton Miffin Company, Boston, MA, 1969.

[15] D.E. KNUTH, *The Art of Computer Programming, Vol.3: Sorting and Searching*, Addison-Wesley Publishing Company, Massachusetts, 1973.

[16] L. KUCERA, *Parallel Computation and Conflicts in Memory Access*, Information Processing Letters 14(2), 1982, pp. 93-96.

[17] T. LEIGHTON, *Tight Bounds on the Complexity of Parallel Sorting*, 16th ACM Symposium on Theory of Computing, Washington, D.C., 1984, pp. 71-80.

[18] R.E. LADNER AND M.J. FISCHER, *Parallel Prefix Computation*, J. ACM 27(4), 1980, pp. 831-838.

[19] G.L. MILLER AND J.H. REIF, *Parallel Tree Contraction and Its Application*, 18th IEEE Symposium on Foundations of Computer Science, 1985, pp. 478-489.

[20] J.H. REIF, *Symmetric Complementation*, J. ACM, 31(2), 1984a, pp. 401-421.

[21] J.H. REIF, *On the Power of Probabilistic Choice in Synchronous Parallel Computations*, SIAM J. Computing 13(1), 1984b, pp. 46-56.

[22] J.H. REIF, *An Optimal Parallel Algorithm for Integer Sorting*, 18th IEEE Symposium on Foundations of Computer Science, 1985, pp. 496-503.

[23] J.H. REIF AND J.D. TYGAR, *Efficient Parallel Pseudo-Random Number Generation*, CRYPTO'85, Santa Barbara, CA, Aug. 1985.

[24] J.H. REIF AND L.G. VALIANT, *A Logarithmic Time Sort for Linear Size Networks*, Proc. 15th ACM Symposium on Theory of Computing, 1983, pp. 10-16. Also in JACM 34(1), 1987, pp. 60-76.

[25] R. REISCHUK, *A Fast Probabilistic Sorting Algorithm*, Proc. 22nd IEEE Symposium on Foundations of Computer Science, 1981, pp.88-102.

[26] Y. SHILOACH AND U. VISHKIN, *Finding the Maximum, Merging, and Sorting in a Parallel Computation Model*, J. Algorithms 2, 1981, pp.212-219.

[27] R.E. TARJAN, *Depth First Search and Linear Graph Algorithms*, SIAM J. of Computing 1(2), 1972, pp.146-160.

[28] U. VISHKIN, *Randomized Speed-Ups in Parallel Computation*, Proc. of the 16th Symp. on Theory of Computing, 1984, pp. 230-239.

APPENDIX A: Probabilistic Bounds

We say a random variable X *upper bounds* another random variable Y (equivalently, Y *lower bounds* X) if for all x such that $0 \leq x \leq 1$, Probability$(X \leq x) \leq$ Probability$(Y \leq x)$.

A *Bernoulli trial* is an experiment with two possible outcomes viz. *success* and *failure*. The probability of success is p.

A *binomial variable* X with parameters (n, p) is the number of successes in n independent Bernoulli trials, the probability of success in each trial being p.

The *distribution function* of X can easily be seen to be

$$\text{Probability}(X \leq x) \;=\; \sum_{k=0}^{x} \binom{n}{k} p^n (1-p)^{n-k}.$$

[Chernoff 52] and [Angluin and Valiant 79] have found ways of approximating the tail ends of a binomial distribution. In particular, they have shown that

Lemma A.1 *If X is binomial with parameters (n, p), and $m > np$ is an integer, then*

$$Probability(X \geq m) \;\leq\; \left(\frac{np}{m}\right)^m \; e^{m-np}. \tag{1}$$

Also,

$$Probability(X \leq \lfloor (1-\epsilon)pn \rfloor) \;\leq\; exp(-\epsilon^2 np/2) \tag{2}$$

and

$$Probability(X \geq \lceil (1+\epsilon)np \rceil) \;\leq\; exp(-\epsilon^2 np/3) \tag{3}$$

for all $0 < \epsilon < 1$.

APPENDIX B: A Sampling Algorithm

Given an index set $Q = \{1, 2, \ldots, n\}$. Each index belongs to exactly one of $\sqrt{n}$ groups $G_1, G_2, \ldots, G_{\sqrt{n}}$. For any index i, in constant time we can find out the group $G_{i'}$ that i belongs to.

Problem. To compute $N(1), N(2), \ldots, N(\sqrt{n})$ such that $\sum_{i=1}^{\sqrt{n}} N(i) = O(n)$ and $N(i) \geq |G_i|$ for each $i \in [\sqrt{n}]$. Given that each $|G_i| \leq \sqrt{n} \log n$.

Lemma B.1 *The above problem can be solved in time $\tilde{O}(\log n / \log\log n)$ using n processors.*

Proof. We provide a sampling algorithm. A shared memory of size n is used. This shared memory is divided into $\sqrt{n}$ blocks $B_1, B_2, \ldots, B_{\sqrt{n}}$ each of size $\sqrt{n}$.

step1

$n/\log n$ processors in parallel each choose a random index (in $[n]$).

step2

Every processor $\pi \in [n/\log n]$ has to find an assignment for its index i in the block $B_{i'}$. It chooses a random cell in $B_{i'}$ and tries to write in it. If it succeeds, it increments the contents of that cell by 1. If it does not succeed in the first trial, it tries a second time to increment the **same** cell. It tries as many times as it takes.

A total of $h \log n/\log\log n$ (for some h to be determined) time is given.

step3

n processors perform a prefix sum computation on the contents of the shared memory and hence compute $L(1), L(2), \ldots, L(\sqrt{n})$ where $L(i)$ is the sum of the contents of block $B_i, i \in [\sqrt{n}]$.

step4

$\sqrt{n}$ processors set in parallel $N(i) = d(\log n)max(1, L(i))$ and output $N(i), i \in [\sqrt{n}]$. d is a constant to be determined.

Analysis. Let $M(i), i \in [\sqrt{n}]$ stand for the number of indices chosen in step1 that belong to G_i and let $R(i) = d(\log n)max(1, M(i))$. Following the proof of lemma 3.4, the $R(i)$'s satisfy the conditions $\sum_{i=1}^{\sqrt{n}} R(i) = O(n)$ and $R(i) \geq |G_i|, i \in [\sqrt{n}]$. The proof will be complete if we can show that $L(i) = M(i)$ with very high probability.

Showing $L(i) = M(i), i \in [\sqrt{n}]$ is the same as showing that no cell in the common memory will be chosen by more than $h \log n/\log\log n$ processors in step1. Let Y be a random variable equal to the number of processors that have chosen a particular cell q. Following the proof of lemma 3.4, no $M(i)$ will be greater than $c\beta\sqrt{n}$ with probability $\geq 1 - n^{-\beta}$ for any $\beta \geq 1$ and some fixed c. Therefore, Y is upperbounded by a binomial variable with parameters $(c\beta\sqrt{n}, 1/\sqrt{n})$. The Chernoff bounds (equation 1) imply that $Y \leq h \log n/\log\log n$ with probability $\geq 1 - n^{-\alpha}$, for any $\alpha \geq 1$ and a proper h. $\Box$