Optimal and Practical Algorithms for Sorting on the PDM[1]

Sanguthevar Rajasekaran
Department of CSE, Univ. of Connecticut, U.S.A.

Sandeep Sen[2]
Department of CSE, IIT New Delhi, India

Abstract. *The Parallel Disks Model (PDM) has been proposed to alleviate the I/O bottleneck that arises in the processing of massive data sets. Sorting has been extensively studied on the PDM model due to the fundamental nature of the problem - several asymptotically optimal algorithms are known for sorting. Although randomization has been frequently exploited, most of the prior algorithms suffer from complications in memory layouts, implementation, restrictions in range of parameters and laborious analysis. In this paper, we present a randomized mergesort algorithm based on a simple idea that sorts using an asymptotically optimal number of I/O operations with* ***high probability*** *and has all the desirable features for practical implementation.*

In the second part of the paper, we also present several novel algorithms for sorting on the PDM that take only a small number of passes through the data. Recently, a considerable interest has been shown by researchers in developing algorithms for problem sizes of practical interest and we are able to obtain several improvements and simplification, in particular for random input.

1 Introduction

When the amount of data an application has to deal with is enormous, out-of-core computing techniques have to be invoked. In this case, the I/O bottleneck has to be dealt with. The PDM has been proposed to alleviate this I/O bottleneck [2]. In a PDM, there is a (sequential or parallel) computer that has access to $D(\geq 1)$ disks. In one I/O operation, it is assumed that a block of size B can be fetched into the main memory. One typically assumes that the main memory has size M where M is a (small) constant multiple of DB[3].

Efficient algorithms have been devised for the PDM for numerous fundamental problems. In the analysis of these algorithms, typically, the number of I/O operations needed are optimized. Since local computations take much less time than the time needed for the I/O operations, these analyzes are reasonable. Since sorting is a fundamental and highly ubiquitous problem, a lot of effort has been spent on developing sorting algorithms for the PDM. It has been shown by Aggarwal and Vitter [2] that $\Omega\left(\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)}\right)$[4] I/O operations are needed to sort N keys (residing in D disks) where each block contains B keys. The internal memory has a capacity of M keys. (Each key is $\Omega(\log N)$ bits). This lower bound has been proved from the well-known lower bound of $\Omega(N \log N)$ comparisons for any sequential

[1]This research has been supported in part by the NSF Grants CCR-9912395 and ITR-0326155. Preliminary versions of the results in this paper have been presented in either [21] or [22].

[2]This research was conducted when this author was visiting the University of Connecticut.

[3]This is the hardest case as we will show later

[4]In this paper we use log to denote logarithms to the base 2 and ln to denote logarithms to the base e.

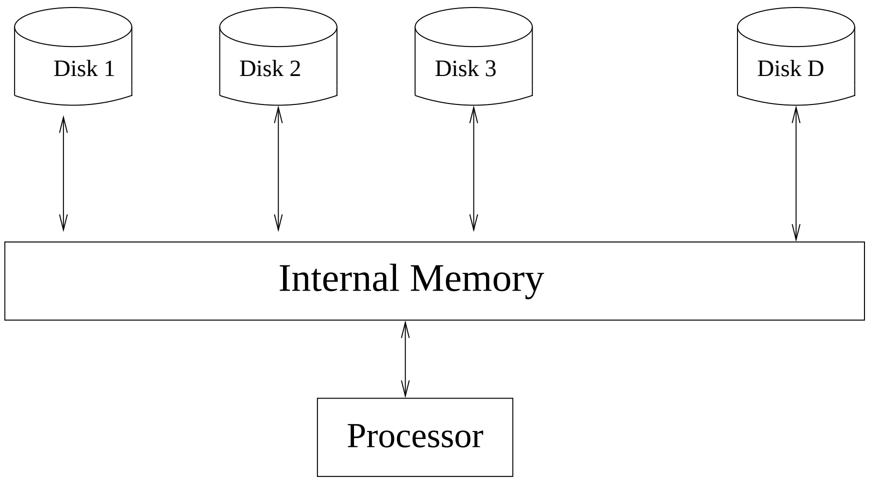

Figure 1: A Machine with Parallel Disks

comparison-based sorting algorithm for N keys (see e.g., [12]). Many asymptotically optimal algorithms have been devised as well (see e.g., Arge [3], Nodine and Vitter [18], and Vitter and Hutchinson [28]). The LMM sort of Rajasekaran [20] is optimal when N, B, and M are polynomially related and is a generalization of Batcher's odd-even merge sort [7], Thompson and Kung's s^2-way merge sort [26], and Leighton's columnsort [14].

Notation. We say the amount of resource (like time, space, etc.) used by a randomized algorithm is $\widetilde{O}(f(N))$ if the amount of resource used is no more than $c\alpha f(N)$ with probability $\geq (1 - N^{-\alpha})$ for any $N \geq n_0$, where c and n_0 are constants and α is a constant ≥ 1. We could also define the asymptotic functions $\widetilde{\Theta}(.)$, $\widetilde{o}(.)$, etc. in a similar manner.

One of the main results of this paper is a simple randomized algorithm for sorting on the PDM that takes only $\widetilde{O}\left(\frac{N}{DB}\frac{\log(N/B)}{\log(M/B)}\right)$ I/O operations. We also present sorting algorithms that take a small number of passes for problem sizes of practical interest.

2 Prior Algorithms and our results

Most of the previous PDM sorting algorithms can be categorized under two families - ones based on bucketsort and the others on mergesort. The first kind is based on distribution sort [29, 28, 17] where keys are classified into buckets depending on their values and this is repeated recursively within each bucket till each bucket reaches a manageable size (corresponding to the base case). The randomized versions of distribution sort (like quicksort) are often simpler than their deterministic counterparts. The basic idea is sampling and due to Frazer and McKellar. Given a sequence X of n keys to sort: 1) a random sample of s keys are picked from X; 2) these sample keys are sorted to get the sequence $l_1, l_2, \ldots, l_s$; 3) X is partitioned into $s+1$ parts $X_0, X_1, \ldots, X_s$ using the sample keys as splitters. In particular, $X_0 = \{q \in X : q \leq l_1\}$, $X_i = \{q \in X : l_i < q \leq l_{i+1}\}$ for $1 \leq i \leq (s-1)$, and $X_s = \{q \in X : q > l_s\}$; and 4) the parts $X_0, X_1, \ldots, X_s$ are sorted recursively and independently.

The second kind of sorting algorithms on the PDM are based on R-way merging for some suitable value of R that minimizes the number of passes through the data for the given size of internal memory [1, 6, 11, 18, 20].

The primary difficulty in the case of distribution sort is in ensuring full parallelism in the case of writing. Likewise, the primary challenge in merge-based sorts is in obtaining optimal read parallelism. By striping across disks, and using internal buffers of size $\Omega(D^2)$, full parallelism can be achieved by a straightforward greedy approach. For $D = \Theta(M/B)$, we cannot afford so much space. More precisely, how do we come up with *balanced* read/write schedules across the D disks when the data is arbitrarily distributed at the beginning and the internal memory size is $O(D)$?

In this context, the algorithm of Barve, Grove, and Vitter [6] deserves special mention. It uses a value of $R = M/B$. This algorithm stripes the runs across the disks such that for each run the first block is stored in a random disk and the other blocks are stored in a cyclic fashion starting from the random disk. They only analyze the expected performance of the algorithm (and no high probability bounds have been derived). Their algorithm called *Simple Randomized Mergesort* (SRM), has an optimal expected performance only when the internal memory size M is $\Omega(BD \log D)$. However, the standard assumption on M is that $M = O(DB)$. The reason for the (slightly) sub-optimal performance is a consequence of an occupancy based analysis that yields a bound of $\omega(1)$ bound on maximum number of blocks lying in a single disk that must be read in a single phase.

This problem has been redressed by the algorithm of Hutchinson, Sanders and Vitter [13]. Their approach is based on an algorithm of Sanders, Egner and Korst [23] who use lazy writing at the expense of an internal buffer. By using Fully Randomized (FR) scheduling to allocate blocks of each stream to disks, they show that an *expected* parallelism off $\Omega(D)$ can be achieved using an internal buffer of size $O(D)$. They used asymptotic queuing theoretic analysis to bound the expected number of writes in a batched arrival queuing system with a bounded buffer. A batch corresponds to a memory-load of keys that we are trying to classify into buckets and the bounded buffer is a part of the memory. Vitter and Hutchinson extended it to a scheduling scheme called Random Cycling (RC) which is easy to stripe across disks. The FR schedule is more complicated to implement and is not read-optimal for $M = o(BD \log D)$. The RC scheduling resulted in optimal distributed sort (RCD) and optimal mergesort (RCM) for $M/B \gg D$ [13] [5] that has generated practical interest [11].

In this paper we present a simple randomized algorithm for sorting on the PDM that makes only $\widetilde{O}\left(\frac{\log(N/M)}{\log(M/B)}\right)$ passes through the data. Our algorithm uses techniques like staggering of the leading blocks (of streams being merged) and periodic rearrangement of input blocks to prevent clustering of the blocks on any single disk. Note that our bound holds **with high probability** for any value of N unlike the previous randomized algorithms for which only expected bounds have been proved. In a sense, we are able to retain the advantages of the simplicity of SRM with minimal modification and obtain optimal parallelism for the entire range of the parameters. In our analysis we rely only on standard tools that do not rely on asymptotic convergence. In addition, we are able to adhere to desirable properties like striping and simplicity. The underlying approach in our algorithm is to first generate a random permutation and subsequently sort the random permutation using a simple mergesort. In the first phase, an efficient radix sorting algorithm is designed to generate a random permutation. The above strategy bears resemblance to the approach of Valiant and Brebner [27] although requiring very different techniques in the context of PDM. In fact, we develop a novel strategy to obtain full parallelism ($\Theta(D)$) for merging random sequences. This is likely to find further applications.

[5]The exact value of the constant is dependent on the buffer size but it is one aspect where our algorithm possibly holds an advantage.

2.1 Fixed passes algorithms

In this paper we also focus on developing sorting algorithms with a small number of passes. We note that for most of the applications of practical interest $N \leq M^2$. For instance, if $M = 10^8$ (integers), then M^2 is 10^{16} (integers) (which is around 100,000 tera bytes). Thus we focus on input sizes of $\leq M^2$ in this paper.

Recently, many algorithms have been devised for problem sizes of practical interest. For instance, Dementiev and Sanders [11] have developed a sorting algorithm based on multi-way merge that overlaps I/O and computation optimally. Their implementation sorts gigabytes of data and competes with the best practical implementations. Chaudhry, Cormen, and Wisniewski [10] have developed a novel variant of columnsort that sorts $M\sqrt{M}$ keys in three passes over the data (assuming that $B = M^{1/3}$). Their implementation is competitive with NOW-Sort. (By a pass we mean $\frac{N}{DB}$ read I/O operations and the same number of write operations.) In [8], Chaudhry and Cormen introduce some sophisticated engineering tools to speedup the algorithm of [10] in practice. They also report a three pass algorithm that sorts $M\sqrt{M}$ keys in this paper (assuming that $B = \Theta(M^{1/3})$). In [9], Chaudhry, Cormen, and Hamon present an algorithm that sorts $M^{5/3}/4^{2/3}$ keys (when $B = \Theta(M^{2/5})$). They combine columnsort and Revsort of Schnorr and Shamir [25] in a clever way. This paper also promotes the need for oblivious algorithms and the usefulness of mesh-based techniques in the context of out-of-core sorting. In fact, the LMM sort of Rajasekaran [20] and all the algorithms in this paper (except for the integer sorting algorithm) are oblivious.

Another important thrust of this paper is on algorithms that have good expected performance. In particular, we are interested in algorithms that take only a small number of passes on an **overwhelming fraction** of all possible inputs. As an example, consider an algorithm $\mathcal{A}$ that takes two passes on at least $(1 - M^{-\alpha})$ fraction of all possible inputs and three passes on at most $M^{-\alpha}$ fraction of all possible inputs. If $M = 10^8$ and $\alpha = 2$, only on at most 10^{-14} % of all possible inputs, $\mathcal{A}$ will take more than two passes. Thus algorithms of this kind will be of great practical importance.

We make the following contributions in this direction: 1) We bring out the need for algorithms that have good expected performance in the context of PDM sorting. A saving of even one pass could make a big difference if the input size is large. Especially, algorithms that run in a small number of passes on an overwhelming fraction of all possible inputs will be highly desirable in practice. As a part of this effort we prove a Lemma on random permutations that should be of independent interest. 2) The second main contribution is in the development of algorithms for input sizes $\leq M^2$. This input size seems to cover most of the applications of practical interest.

The above two thrusts of our interest have yielded several specific algorithms for PDM sorting. All of these algorithms use a block size of $\sqrt{M}$. Here is a list: 1) A three pass algorithm for sorting $M\sqrt{M}$ keys. This algorithm is based on LMM sort and assumes that $B = \sqrt{M}$. In contrast, the algorithm of Chaudhry and Cormen [8] uses a block size of $M^{1/3}$ and sorts $M\sqrt{M}/\sqrt{2}$ keys in three passes. 2) We also present another mesh-based algorithm that sorts $M\sqrt{M}$ keys; 3) An expected two pass algorithm that sorts nearly $M\sqrt{M}$ keys. In this paper, all the expected algorithms are such that they take the specified number of passes on an overwhelming fraction (i.e., $\geq (1 - M^{-\alpha})$ for any fixed $\alpha \geq 1$) of all possible inputs; 4) An expected three pass algorithm that sorts nearly $M^{1.75}$ keys; 5) A seven pass algorithm (based on LMM sort) that sorts M^2 keys. 6) We also present a mesh-based algorithm that sorts M^2 keys; and 7) An expected six pass algorithm that sorts nearly M^2 keys.

2.2 Organization

In Section 3 we present our permutation algorithm and in Sections 4 and 5 we provide details of our optimal randomized sorting algorithm. Section 6 presents our algorithms that take a small number of passes. Section 7 concludes the paper.

3 Integer sorting and Random Permutation

Often, the keys to be sorted are integers in some range $[1, R]$. Numerous sequential and parallel algorithms have been devised for sorting integers. These integer sorting algorithms can be of two types, namely, forward radix sorting (or Most Significant Bit (MSB) first sorting) and backward radix sorting (or Least Significant Bit (LSB) first sorting). In either case the keys are sorted in phases where in each phase the keys are sorted only with respect to some number of bits. In any phase of the LSB first sorting we sort all the inputs keys with respect to the same set of bits. On the other hand forward radix sorting works differently. In the first phase of this algorithm we sort all the input keys with respect some number (say ℓ) of bits. This sorting partitions the input keys such that each part has keys whose most significant ℓ bits are the same. Thus at the end of the first phase each bucket becomes an independent input set that has to be sorted. In the second phase each bucket will be sorted with respect to some number of the next most significant bits. As a result, the bucket will get split into many buckets, and so on. This process of splitting buckets continues until the buckets are of small size. When this happens the buckets can be sorted using a base-case algorithm.

Several efficient out-of-core algorithms have been devised by Arge, Ferragina, Grossi, and Vitter [4] for sorting strings. For instance, three of their algorithms have the I/O bounds of $O\left(\frac{N}{B}\log_{M/B}\frac{N}{B}\right)$, $O\left(\frac{N}{FB}\log_{M/B}\frac{N}{F}+\frac{N}{B}\right)$, and $O\left(\frac{K}{B}\log_{M/B}\frac{K}{B}+\frac{N}{B}\log_{M/B}|\Sigma|\right)$, respectively. These algorithms sort K strings with a total of N characters from the alphabet Σ. Here F is a positive integer such that $F|\Sigma|^F \leq M$ and $|\Sigma|^F \leq N$. These algorithms could be employed on the PDM to sort integers. For a suitable choice of F, the second algorithm (for example) is asymptotically optimal.

In this section we analyze LSB first radix sort (see e.g., [12]) in the context of PDM sorting. This algorithm sorts an arbitrary number of keys. We assume that each key fits in one word of the computer. We believe that for applications of practical interest radix sort applies to run in no more than 4 passes for most of the inputs.

The range of interest in practice seems to be $[1, M^c]$ for some constant c. For example, weather data, market data, etc. are such that the key size is no more than 32 bits. The same is true for personal data kept by governments. As another example, if the key is social security number, then 32 bits are enough. However, one of the algorithms given in this section applies for keys from an arbitrary range as long as each key fits in one word of the computer.

The first case we consider is one where the keys are integers in the range $[1, M/B]$. Also assume that each key has a random value in this interval. If the internal memory of a computer is M, then it is reasonable to assume that the word size of the computer is $\Theta(\log M)$. Thus each key of interest fits in one word of the computer. M and B are used to denote the internal memory size and the block size, respectively, in words.

The idea can be described as follows. We build M/B runs one for each possible value that the keys can take. From every I/O read operation, M keys are brought into the core memory. From out of all the keys in the memory, blocks are formed. These blocks are written to the disks in a striped manner. The striping method suggested in [20] is used. Some of the blocks could be non full. All the blocks in the memory are written to the disks using as few parallel write steps as possible. We assume that $M = CDB$ for some constant C. Let $R = M/CB$. The details of the algorithm IntegerSort follow.

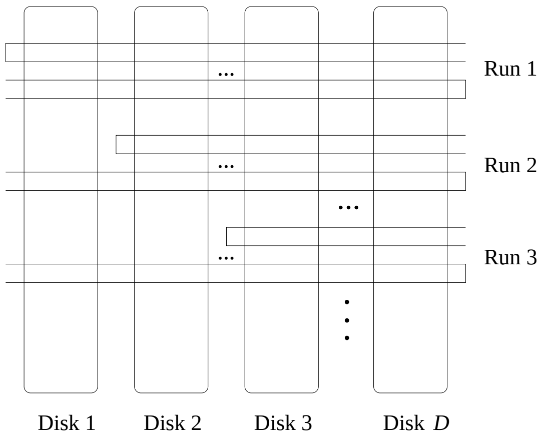

Figure 2: The runs are striped across disks in a staggered fashion. In this example the leading blocks are staggered by one disk - more generally they can be staggered by s disks.

```
Algorithm IntegerSort

for $i := 1$ to $N/M$ do

    1. In $C$ parallel read operations, bring into the core memory $M = CDB$ keys.

    2. Sort the keys in the internal memory and form blocks according to the values of
       keys. Keep a bucket for each possible value in the range $[1, R]$. Let the buckets be
       $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_R$. If there are $N_i$ keys in $\mathcal{B}_i$, then, $\lceil N_i/B \rceil$ blocks will be formed out of
       $\mathcal{B}_i$ (for $1 \leq i \leq R$).

    3. Send all the blocks to the disks using as few parallel write steps as possible. The runs
       are striped across the disks (in the same manner as in [20]) (see Figure 2).

       The number of write steps needed is $\max_i\{\lceil N_i/B \rceil\}$. 4. Read the keys written to the
       disks and write them back so that the keys are placed contiguously across the disks.
```

Theorem 3.1 *Algorithm* IntegerSort *runs in $O(1)$ passes through the data for a large fraction ($\geq (1 - N^{-\alpha})$ for any fixed $\alpha \geq 1$) of all possible inputs. If step 4 is not needed, the number of passes is $(1 + \mu)$ and if step 4 is included, then the number of passes is $2(1 + \mu)$ for some fixed $\mu < 1$.*

Proof: Call each run of the **for** loop as a *phase* of the algorithm. The expected number of keys in any bucket is CB. Using Chernoff bounds, the number of keys in any bucket is in the interval $[(1 - \epsilon)CB,\ (1 + \epsilon)CB]$ with probability $\geq [1 - 2\ \exp(-\epsilon^2 CB/3)]$. Thus, the number of keys in every bucket is in this interval with probability $\geq \left(1 - \exp\left[\frac{-\epsilon^2 CB}{3} + \ln(2R)\right]\right)$. This probability will be $\geq (1 - N^{-(\alpha+1)})$ as long as $B \geq \frac{3}{C\epsilon^2}(\alpha + 1)\ln N$. This is readily satisfied in practice (since the typical assumption on B is that it is $\Omega(M^\delta)$ for some fixed $\delta > 1/3$).

As a result, each phase will take at most $\lceil (1+\epsilon)C \rceil$ write steps with high probability. This is equivalent to $\frac{\lceil (1+\epsilon)C \rceil}{C}$ passes through the data. This number of passes is $1+\mu$ for some constant $\mu < 1$.

Thus, with probability $\geq (1 - N^{-\alpha})$, IntegerSort takes $(1+\mu)$ passes excluding step 4 and $2(1+\mu)$ passes including step 4. □

As an example, if $\epsilon = 1/C$, the value of μ is $1/C$.

Remark 3.1 *The sorting algorithms of [28] have been analyzed using asymptotic analysis. The bounds derived hold only in the limit. In comparison, our analysis is simpler and applies for any N.*

We extend the range of the keys using the following algorithm. This algorithm employs forward radix sorting. In each stage of sorting, the keys are sorted with respect to some number of their MSBs. Keys that have the same value with respect to all the bits that have been processed up to some stage are said to form a *bucket* in that stage. In the following algorithm, δ is any constant < 1.

```
Algorithm RadixSort

for $i := 1$ to $(1+\delta)\frac{\log(N/M)}{\log(M/CB)}$ do
    1. Employ IntegerSort to sort the keys with respect to their $i$th most
       significant $\log(M/CB)$ bits.
2. Now the size of each bucket is $\leq M$. Read and sort the buckets.
```

Theorem 3.2 *N random integers in the range $[1, R]$ (for any R) can be sorted in an expected $(1+\nu)\frac{\log(N/M)}{\log(M/B)} + 1$ passes through the data, where ν is a constant < 1. In fact, this bound holds for a large fraction ($\geq 1 - N^{-\alpha}$ for any fixed $\alpha \geq 1$) of all possible inputs.*

Proof. In accordance with Theorem 3.1, each run of step 1 takes $(1+\mu)$ passes. Thus RadixSort takes $(1+\mu)(1+\delta)\frac{\log(N/M)}{\log(M/B)}$ passes. This number is $(1+\nu)\frac{\log(N/M)}{\log(M/B)}$ for some fixed $\nu < 1$.

It remains to show that after $(1+\delta)\frac{\log(N/M)}{\log(M/B)}$ runs of step 1, the size of each bucket will be $\leq M$. At the end of the first run of step 1, the size of each bucket is expected to be $\frac{NBC}{M}$. Using Chernoff bounds, this size is $\leq (1+\epsilon)\frac{NBC}{M}$ with high probability, for any fixed $\epsilon < 1$. After k (for any integer k) runs of step 1, the size of each bucket is $\leq N(1+\epsilon)^k (CB/M)^k$ with high probability. This size will be $\leq M$ for $k \geq \frac{\log(N/M)}{\log\left[\frac{M}{(1+\epsilon)CB}\right]}$. The RHS of the above expression is $\leq (1+\delta)\frac{\log(N/M)}{\log(M/B)}$ for any fixed $\delta < 1$ as long as $M > C^2 B$ which is true for $D \geq C$.

Step 2 takes one pass. □

Remark 3.2 *As an example, consider the case $N = M^2$, $B = \sqrt{M}$ and $C = 4$. In this case,* RadixSort *takes no more than 3.6 passes through the data.*

We show how to randomly permute N given keys such that each permutation is equally likely. We employ RadixSort for this purpose. The idea is to assign a random label with each key in the range $[1, N^{1+\beta}]$ (for any fixed $0 < \beta < 1$) and sort the keys with respect to their labels. This can be done in $(1+\mu)\frac{\log(N/M)}{\log(M/B)} + 1$ passes through the data with probability $\geq 1 - N^{-\alpha}$ for any fixed $\alpha \geq 1$. Here μ is a constant < 1. For many applications, this permutation may suffice. But we can ensure that each permutation is equally likely with one more pass through the data.

When each key gets a random label in the range $[1, N^{1+\beta}]$, the labels may not be unique. The maximum number of times any label is repeated is $\widetilde{O}(1)$ from the observation that the number of keys falling in a bucket is binomially distributed with mean $1/n$ and applying Chernoff bounds (equation 1 in Appendix B). We have to randomly permute keys with equal labels which can be done in one more pass through the data as follows. We think of the sequence of N input keys as $S_1, S_2, \ldots, S_{N/DB}$ where each S_i $(1 \leq i \leq N/(DB))$ is a subsequence of length DB. Note that keys with the same label can only span two such subsequences. We bring in DB keys at a time into the main memory. We assume a main memory of size $2DB$. There will be two subsequences at any time in the main memory. Required permutations of keys with equal labels are done and DB keys are shipped out to the disks. The above process is repeated until all the keys are processed.

Remark 3.3 *With more care we can eliminate this extra pass by combining it with the last stage of radix sort.*

Thus we get the following:

Theorem 3.3 *We can permute N keys randomly in $O(\frac{\log(N/M)}{\log(M/B)})$ passes through the data with probability $\geq 1 - N^{-\alpha}$ for any fixed $\alpha \geq 1$, where μ is a constant < 1, provided $B = \Omega(\log N)$.*

Remark 3.4 *In the above Theorem, we assume that $B = \Omega(\log N)$. This is a very benign assumption that readily holds in practice. However, we later show how to eliminate this assumption without sacrificing the asymptotic performance.*

4 Randomized Sorting

In this section we present a randomized sorting algorithm that sorts N given keys in $\widetilde{O}\left(\frac{\log(N/M)}{\log(M/B)}\right)$ passes through the data. Our algorithm employs the permutation algorithm from Section 3.

4.1 A simple algorithm

We start with a simple version of the algorithm (called RSort1) which will be useful for exposition of the basic ideas. Unfortunately, RSort1 may not run in an optimal number of I/O's. This algorithm is modified in the next subsection to achieve theoretical optimality. In the following algorithm, $R = M/B$.

```
                         Algorithm RSort1

1. Randomly permute the input $N$ keys using the algorithm of Theorem 3.3.
2. In one pass through the data form runs each of length $M = DB$.
3. for $i := 1$ to $\frac{\log(N/M)}{\log R}$ do
       while there are more runs do
           Merge the next $R$ runs as follows.

           Start by bringing in two blocks from each run. Assume that $R = D = M/B$
           and the main memory is of size $2DB$. Merge the runs to ship $M$ keys out to the
           disks. This run becomes an input run for the next iteration. If one of the
           runs becomes empty before $M$ output keys are formed, start all over again.
           (We will show that the probability of this happening is negligible).

           From here on, maintain the invariant that for each run we have two leading blocks
           in the memory. Form $BD$ output keys and ship them. Repeat this until the $R$
           runs are merged.
```

Analysis. Let a *phase* refer to one run of step 3 of RSort1. In the following discussion ignore the number of parallel I/Os needed to do one scan through the data while merging the runs.

Consider the problem of merging any R runs in some phase of RSort1. Consider some point in time when there are $2D$ blocks in the main memory with 2 blocks per run. We merge these blocks to form M output keys. Note that the M output keys are such that each key is equally likely to have come out of the R runs - the number of keys from each run in a phase is binomially distributed with success probability $\frac{1}{R}$ and expectation B. Using Chernoff bounds, this number lies in the interval $[(1-\epsilon)B,\ (1+\epsilon)B]$ with probability $\geq 2[1 - \exp(-\epsilon^2 B)/3]$, ϵ being any constant > 0. In other words, each run gets consumed at the rate of at least $(1-\epsilon)$ blocks per $(1+\epsilon)$ blocks brought in (from each run). For $B \geq \log N$, this holds with high probability. Note that the probability of ϵ being very close to 1 is very low and hence the event of two blocks getting consumed before M output keys are formed is very low.

In summary, with high probability, it takes at most $1/(1-\epsilon)$ scans through a run before it gets consumed completely. As a result, RSort1 makes $\widetilde{O}\left(\frac{\log(N/M)}{\log(M/B)}\right)$ scans through the input.

Even though RSort1 makes an optimal number of scans through the input, each scan may take more than an optimal number of I/Os. This can be seen as follows. At the beginning of the algorithm, the runs are striped in a cyclic fashion. Let the runs be $R_1, R_2, \ldots, R_q$. The first block of run i will be in disk $(i-1) \bmod D + 1$; the second block of run i will be in disk $i \bmod D + 1$; and so on (for $1 \leq i \leq q$). If whenever blocks are accessed from different runs these blocks come from different disks, then it will mean that the number of I/O operations is optimal as well. For instance if each run gets consumed at the rate of one block per block brought in, then this will hold.

However, the runs get consumed at different rates. For instance, there could come a time when we need a block from each run and all of these blocks are in the same disk. An occupancy analysis similar to the one in Barve Grove and Vitter [6] will imply that the expected number of I/O operations in the worst case could be non-optimal unless M/B is $\Omega(D \log D)$.

In the next subsection we modify RSort1 to make it optimal and still retain the simplicity.

4.2 A Second Algorithm: Periodic Resetting

The key ideas to make RSort1 optimal are: 1) Let $Q_1, Q_2, \ldots, Q_R$ be the runs to be merged at some point in time. Let a *stage* refer to the step of bringing in required keys, merging the $2DB$ keys in memory and forming M output keys. We keep the R runs such that the leading blocks for the runs are in successive disks (or very nearly so); and 2) When there are many blocks in every run, the above property may be difficult to maintain since as time progresses, the leading blocks deviate more and more from the expected disk locations. We periodically rearrange the leading M keys of each run so that the above property is reinstated after the rearrangement. Again we assume that $R = M/B$.

In the description that follows, we use M to denote DB and we assume that the actual internal memory has size $2DB$.

```
                         Algorithm RSort2

1. Permute the input $N$ keys using the algorithm of Theorem 3.3.
2. In one pass through the data form runs of length $M = DB$ each.
3. for $i := 1$ to $\frac{\log(N/M)}{\log R}$ do
     while there are more runs do
        Merge the next $R$ runs as follows.

        Begin by bringing in $\frac{2M}{RB} = 2$ blocks from each run. Merge the runs
        to ship $M$ keys out to the disks to be used as an input run for the next iteration.
        If one of the runs becomes empty before $M$ output keys are formed, start all over
        again. We shall show later that the probability of this event is low.

        Maintain the property that there are exactly $\frac{2M}{RB} = 2$ leading blocks per run.
        Form $BD$ output keys and ship them. Call the step of bringing in
        enough keys to have $2M/(RB)$ blocks per run, merging them and outputting $M$
        keys as a stage of the algorithm. After every $R(= M/B)$ stages perform
        a rearrangement of runs. In particular, read the leading $M$ keys of each run
        and write them back so that the leading blocks of the runs are in successive disks.
        Use separate areas in the disks for the purpose of rewriting.

        Repeat the above step until the $R$ runs are merged.
```

Theorem 4.1 RSort2 *takes* $\widetilde{O}\left(\frac{\log(N/M)}{\log(M/B)}\right)$ *read passes through the data provided* $B = \Omega(\sqrt{M \log N})$.

Proof. Let a *phase* of the algorithm refer to one run of step 3 and let a *stage* of the algorithm refer to bringing in enough keys per run (i.e. $2M/R$ keys per run), merging the runs, and shipping M keys out to the disks.

It suffices to prove that each phase of the algorithm takes $\widetilde{O}(N/DB)$ I/Os.

In each stage of the algorithm the expected number of blocks consumed from each run is 1. If the starting block of a run is i then after q stages, the leading block of this run is expected to be in disk $(i + q - 1) \bmod D + 1$. Assume that the leading block of each run continues to be within one disk of its expected disk. Consider the task of bringing into main memory at most K leading blocks of each run.

How many I/Os will be needed? It is easy to see that in the worst case, $3 \cdot K$ I/Os will suffice as each disk can have at most 3 of the leading blocks. Thus when $K = 1$, three I/O's suffice.

We can actually obtain a stronger result:

Lemma 4.1 *Consider D disks and R runs with $R \leq D$. The runs are striped in the usual way. Let the leading block of each run stray away from its expected disk by at most q disks. The problem of bringing in K blocks from each run can be accomplished in at most $K + 2q$ I/Os.*

Proof: Assume that $R = D$ since this corresponds to the worst case. Also assume that $K \leq D$ for simplicity (though the result is general). Consider any disk $\mathcal{D}$. From out of the blocks we want to fetch, how many blocks will reside in $\mathcal{D}$? If the starting blocks of the runs are equidistant, then K blocks will reside in $\mathcal{D}$. The starting blocks of runs can be at most q disks away from their expected starting disks. The starting disks of some of the runs could be to the right of their expected disks and the starting blocks of some others could be to the left of their starting disks. The number of blocks in $\mathcal{D}$ (excluding those K runs that are expected to have a block each in $\mathcal{D}$) from out of the first kind of runs is at most q and the number of blocks from the second kind is at most q. (We assume that there is sufficient buffer, i.e. $(K + 2q)DB$ for this purpose.) $\Box$

When we perform R stages, the expected number of keys coming out of each run is M. This number will stray away from its expected value by at most $\sqrt{\alpha M \ln N}$ with probability exceeding $(1 - N^{-\alpha})$. Thus the leading block of each run will stray away by at most one disk provided $B \geq \sqrt{\alpha M \ln N}$. This condition is readily satisfied in practice. In this case, each stage of the algorithm can be completed in three I/Os with high probability.

Also, all the rearrangement of keys take an additional $(1+\nu)\log(N/M)/\log(M/B)$ read passes and the same number of write passes, where ν is any constant > 0.

In summary, the number of read passes taken by the algorithm is $4(1+\nu)\frac{\log(N/M)}{\log(M/B)} + (1+\mu)\frac{\log(N/M)}{\log(M/B)} + 2$ with probability $\geq (1 - N^{-\alpha})$ for any fixed $\alpha \geq 1$. Here ν, μ are constants between 0 and 1. Note that the number write passes are only $2(1+\nu)\log(N/M)/\log(M/B) + (1+\mu)\frac{\log(N/M)}{\log(M/B)}$. *end of proof Theorem 4.1* $\Box$

4.3 Relaxing the Constraint on B

RSort2 assumes that $B = \Omega(\sqrt{M \log N})$. This assumption can be relaxed with the following idea. Employ a value of $R = (M/B)^{\epsilon}$ for any constant $1 > \epsilon > 0$.

Note that when $R = (M/B)^{\epsilon}$, we have $R < D$. If $Q_1, Q_2, \ldots, Q_R$ are the runs, we stripe Q_1 starting from disk 1, Q_2 starting from disk $1 + D/R$, Q_3 starting from $1 + 2D/R$, and so on. (Assume w.l.o.g. that D is an integral multiple of R). In other words, the leading blocks of the runs are D/R disks apart. Therefore, even if the leading blocks of the runs stray by D/R blocks each stage can be performed in three I/Os per disk. Further, if we can ensure that the deviation is less than half this quantity, i.e. $D/2R$, then $q = 0$ and each stage can be performed in one I/O per disk. Each run contributes $(M/B)^{1-\epsilon}$ (expected) blocks in each stage.

The resultant algorithm RSort is the same as RSort2 except that rearrangements are done every $(M/B)^{\epsilon}$ stages and we use $R = (M/B)^{\epsilon}$.

When $(M/B)^{\epsilon}$ stages are performed, the expected number of keys consumed from each run is M. The actual value for any run can stray away from its expected value by $\sqrt{\alpha M \ln N}$, with probability greater than $(1 - N^{-\alpha})$. If we can bound the deviation by $0.5 \cdot DB/R$ the maximum number of blocks required from any disk is 1 (i.e. $q = 0$). This happens when $\sqrt{\alpha M \ln N} \leq 0.5 \cdot M^{1-\epsilon}B^{\epsilon}$. This implies that $B \geq 2M^{(\epsilon-1/2)/\epsilon}(\alpha \ln N)^{1/(2\epsilon)}$.

When $\epsilon = 1/2$, the above condition becomes: $B \geq 2\alpha \ln N$. This is a benign condition and readily holds in practice. For a value of $\epsilon = \frac{1}{2} - \delta$ (for any fixed $\delta > 0$), the above condition becomes, $B \geq 1$, provided $N \leq e^{M^{2\delta/\alpha}}$.

An Alternative Analysis for large N. Assume that N is $\Omega(M^C)$, for some constant C. The expected number of I-Os for R stages is cR for some constant c from the properties of even distribution of keys. Call this an *epoch* and because of rearrangements the epochs are independent. Therefore the probability that more than $\sqrt{\alpha(N/(MR)) \ln N}$ epochs exceed cR I/Os is less than $1/N$. This can be argued as follows. Let $X_i = 1$ if the i-th epoch requires in excess of $2cR$ I/O's (a *bad* epoch) and 0 otherwise. Clearly $\Pr[X_i = 1] \leq 1/2$ and X_i's are independent. By applying Chernoff bounds to $\sum_i X_i$, we obtain the required bound on the number of *bad* epochs. The total number of I/Os in a *bad* epoch cannot exceed $R^{1+\zeta}$ (for any constant $\zeta > 0$) from Lemma 4.1. Therefore, if $N = \Omega(M^C)$, for an appropriate C, the total number of I/O's over all bad epochs is $o(N/B)$ with high probability.

Thus we obtain the following result:

Theorem 4.2 RSort *takes* $\widetilde{O}\left(\frac{\log(N/M)}{\log(M/B)}\right)$ *read passes for (uniformly distributed) random input data.*

Bounding the leading constant

For choice of $\epsilon = 1/2$, the number of iterations made by the algorithm is no more than $2(1+\nu) \cdot \frac{\log(N/M)}{\log(M/B)}$ where each iteration involves $2(1+\nu)$ read passes including rearrangement. The number of write passes is same. This gives a total of $8(1+\nu)\frac{\log(N/M)}{\log(M/B)}$ passes with probability $\geq (1 - N^{-\alpha})$ for any constant $\alpha \geq 1$.[6] Here ν, μ are constants between 0 and 1. To this, we must add the time for generating the initial random permutation.

When B is large, we can decrease the number of read passes made by RSort as follows. Let $B = M^\beta$. The condition on B becomes: $M^\beta \geq M^{(\epsilon-1/2)/\epsilon}(\alpha \ln N)^{1/(2\epsilon)}$. This condition is satisfied when $\epsilon < \frac{1}{2(1-\beta)}$. For this choice of ϵ, the number of read passes made by RSort is $(4-4\beta)(1+\nu)\frac{\log(N/M)}{\log(M/B)} + (1+\mu)\frac{\log(N/M)}{\log(M/B)} + 2$. This follows from : 1) The permutation takes $(1+\mu)\frac{\log(N/M)}{\log(M/B)}$ passes; 2) Initial runs formation takes one pass; 3) Merging of runs takes $2(1-\beta) \times (1+\nu)\frac{\log(N/M)}{\log(M/B)}$ passes; and 4) the rearrangements take $2(1-\beta) \times (1+\nu)\frac{\log(N/M)}{\log(M/B)}$ read passes. When $\beta = 1/2$, the number of passes is $4(1+\nu)\frac{\log(N/M)}{\log(M/B)} + (1+\mu)\frac{\log(N/M)}{\log(M/B)} + 2$, which is the same as what RSort2 takes. As another example, when $\beta = 3/4$, the number of read passes is $2(1+\nu)\frac{\log(N/M)}{\log(M/B)} + (1+\mu)\frac{\log(N/M)}{\log(M/B)} + 2$.

4.4 Relaxing the constraint in RadixSort

The algorithms IntegerSort and RadixSort assume that $B = \Omega(\ln N)$. As a consequence, the algorithm of Theorem 3.3 also makes this assumption. We can relax this constraint in exactly the same manner as in Section 4.3. In particular, the algorithm IntegerSort is modified as follows. Instead of sorting N keys in the range $[1, M/B]$, IntegerSort now sorts N keys in the range $[1, R = (M/B)^\epsilon]$ where ϵ is defined in Section 4.3. The algorithm runs in stages. In any stage we bring in M keys. Reading doesn't pose any difficulties in parallel access. There could be potential problems in writing. We address this problem in exactly the same manner as in Section 4.3. Periodically, we start writing the runs so that the leading blocks are equally apart.

[6]The mergesort algorithm in [13] has a constant $2 + D/m$ for the expected running time, where m is the size of internal memory. However, it is not clear how one can obtain high probability bounds.

4.5 The general case – $M > DB$

Thus far we have assumed that the size of internal memory is $2BD$. But in practice, the internal memory could be larger than a constant multiple of BD. All the results we have derived so far extend to this case in a straight forward way. Let $M = 2qBD$ for some integer q.

Theorem 3.3 now becomes:

Theorem 4.3 *We can permute N keys randomly in $(1+\mu)\frac{\log(N/(qBD))}{\log(qD)} + 1$ passes through the data with probability $\geq 1 - N^{-\alpha}$ for any fixed $\alpha \geq 1$, where μ is any constant > 0.*

In the algorithm of Theorem 3.3, we sorted keys in phases where in each phase the range was $[1, D]$. There are several stages in any phase. A stage involves bringing into main memory around DB keys, sorting them, and shipping around a block per key value (i.e., around DB keys) to the disks. In the new algorithm, we have to sort the keys in phases where in each phase we sort keys in the range $[1, qD]$. Here again, there are many stages per phase. In any stage we bring in qDB keys, sort them, and ship out around qDB keys. The only difference is that a stage previously involved one parallel read I/O and in the new algorithm there are q parallel read I/Os per stage.

In a similar fashion, we obtain our final result:

Theorem 4.4 *For $M = 2qDB$, RSort can be easily modified to run in $\widetilde{O}\left(\frac{\log(N/(qBD))}{\log(qD)}\right)$ passes through the data.*

5 Sorting Algorithms that Take a Small Number of Passes

We now shift our focus to PDM sorting algorithms that that take a small number of passes for problem sizes of practical interest. More specifically, we assume that $N \leq M^2$ - the randomized algorithm of the previous section will take expected $\Theta(1)$ passes. Here we explore what we can do in small number of passes like three to five. We present several simple deterministic algorithms. Further we also explore their performance for random inputs in terms of expected number of passes.

For this section we assume that the block size B is $\Omega(M^\alpha)$ where α is a constant between 0 and 1 (typically $\alpha = 0.5$). The reason for this is that it is consistent with current technology and larger block sizes also favour parallelism.

5.1 A Lower Bound

The following lower bound result will help us judge the optimality of algorithms presented in this paper.

Lemma 5.1 *At least two passes are needed to sort $M\sqrt{M}$ elements when the block size is $\sqrt{M}$. At least three passes are needed to sort M^2 elements when the block size is $\sqrt{M}$. These lower bounds hold on the average as well.*

Proof. The bounds stated above follow from the lower bound theorem proved in [5]. In particular, it has been shown in [5] that $\log(N!) \leq N \log B + I \times (B \log((M-B)/B) + 3B)$, where I is the number of I/O operations taken by any algorithm that sorts N keys residing in a single disk. Substituting $N = M\sqrt{M}, B = \sqrt{M}$, we see that $I \geq \frac{2M\left(1-\frac{1.45}{\log M}\right)}{\left(1+\frac{6}{\log M}\right)}$. The RHS is very nearly equal to $2M$. In other words, to sort $M\sqrt{M}$ keys, at least two passes are needed. It is easy to see that the same is true for the PDM also. In a similar fashion, one could see that at least three passes are needed to sort M^2 elements. □

5.2 Two Three-Pass Algorithms

In this section we present two three-pass algorithms for sorting on the PDM. Both of these algorithms assume a block size of $\sqrt{M}$. The first algorithm is based on LMM sort and the second algorithm is mesh-based. Both of them sort $M\sqrt{M}$ keys.

5.2.1 LMM Based Algorithm

We adapt the (l, m)-merge sort (LMM sort) algorithm of Rajasekaran [20]. The LMM sort partitions the input sequence of N keys into l subsequences, sorts them recursively and merges the l sorted subsequences using the (l, m)-merge algorithm.

The (l, m)-merge algorithm takes as input l sorted sequences $X_1, X_2, \ldots, X_l$ and merges them as follows. Unshuffle each input sequence into m parts. In particular, X_i $(1 \leq i \leq l)$ gets unshuffled into $X_i^1, X_i^2, \ldots, X_i^m$. Recursively merge $X_1^1, X_2^1, \ldots, X_l^1$ to get L_1; Recursively merge $X_1^2, X_2^2, \ldots, X_l^2$ to get L_2; $\cdots$; Recursively merge $X_1^m, X_2^m, \ldots, X_l^m$ to get L_m. Now shuffle $L_1, L_2, \ldots, L_m$. At this point, it can be shown that each key is within a distance $\leq lm$ from its final sorted position. Perform local sorting to move each key to its sorted position.

The analysis of many of the sorting algorithms on a rectangular array is based on 0-1 principle so that we only have to analyze inputs consisting arbitrary number of 0's and 1's. Central to our analysis as well as others like [25, 15, 16] is the notion of *dirty* rows/columns/blocks.

Definition A row/column is *dirty* if it contains a mixture of 0's and 1's. Similarly a (rectangular) block is dirty if it contains a mixture of 0's and 1's. Otherwise it is *clean*, i.e., it contains only 0's or only 1's. Note that when an array is sorted in a row-major indexing scheme, at most one row is dirty.

Columnsort algorithm [14], odd-even merge sort [7], and the s^2-way merge sort algorithms are all special cases of LMM sort [20]. For the case of $B = \sqrt{M}$, and $N = M\sqrt{M}$, LMM sort can be specialized as follows to run in three passes.

Algorithm ThreePass1

1. Form $l = \sqrt{M}$ runs each of length M. These runs have to be merged using (l, m)-merge. The steps involved are listed next. Let $X_1, X_2, \ldots, X_{\sqrt{M}}$ be the sequences to be merged.

2. Unshuffle each X_i into $\sqrt{M}$ parts so that each part is of length $\sqrt{M}$. This unshuffling can be combined with the initial runs formation task and hence can be completed in one pass.

3. In this step, we have $\sqrt{M}$ merges to do, where each merge involves $\sqrt{M}$ sequences of length $\sqrt{M}$ each. Observe that there are only M records in each merge and hence all the mergings can be done in one pass through the data.

4. This step involves shuffling and local sorting. The length of the dirty sequence is $(\sqrt{M})^2 = M$. Shuffling and local sorting can be combined and finished in one pass through the data as showm in [20].

We get the following:

Lemma 5.2 *LMM sort sorts $M\sqrt{M}$ keys in three passes through the data when the block size is $\sqrt{M}$.*

Observation 5.1 *Chaudhry and Cormen [8] have shown that Leighton's columnsort algorithm [14] can be adapted for the PDM to sort $\sqrt{M^{1.5}/2}$ keys in three passes. In contrast, the three pass algorithm of Lemma 5.2 (based on LMM sort) sorts $M^{1.5}$ keys in three passes.*

5.2.2 A Mesh-Based Algorithm

In this section we describe a simple algorithm that can sort $M^{3/2}$ elements using M internal memory. Consider the input arranged as an $M \times \sqrt{M}$ array. We will often refer to *sub-meshes* $r \times c$ where such a submesh contains a subset of r consecutive rows and c consecutive columns beginning from a multiple of r rows and c columns respectively.

Algorithm ThreePass2

1. Sort all the $\sqrt{M} \times \sqrt{M}$ sub-meshes.
   The sorting order is row major such that every consecutive submeshes have their rows sorted in reverse directions.
2. Sort all columns vertically.
3. Sort every consecutive pair of $\sqrt{M}/2 \times \sqrt{M}$ submesh by bringing them one after the other (in a top to down ordering) into the internal memory. After sorting, the smallest $\sqrt{M}/2$ elements are written out and the next one is brought in till all sub-meshes are exhausted.

Let us first prove the correctness before we show that the algorithm makes exactly three passes.

Theorem 5.1 *Algorithm* ThreePass2 *sorts the $M \times \sqrt{M}$ data items correctly for all inputs.*

Proof: Our proof is based on 0-1 principle and the notion *dirty* rows defined earlier. After Step 1, every $\sqrt{M} \times \sqrt{M}$ sub-mesh has at most 1 dirty row. After Step 2, there can be at most $\sqrt{M}$ dirty rows which can be further restricted to $\sqrt{M}/2$ from the principle of Shearsort [24]. This implies at most two dirty $\sqrt{M}/2$ sub-meshes. Step 3 cleans up these in a manner similar to [15].

Now we proceed to bound the number of passes. Assume that the initial data is striped row wise, in blocks of size $\sqrt{M}$. Therefore the entire submesh can be read using one parallel read. After sorting them these are written out in a column major (striped across columns). This also achieves full parallelism as each submesh contains $\sqrt{M}$ blocks - one from each column. Therefore in the next phase sorting columns can be done by reading one column at a time. While writing these back we do the reverse of Step 1. Step 3 is clearly one pass through the data reading $\sqrt{M} \times \sqrt{M}$ element sub-mesh at a time. □

5.3 A Useful Lemma

In this section we prove a lemma that will be useful in the analysis of expected performance of sorting algorithms.

Consider a set $X = \{1, 2, \ldots, n\}$. Let $X_1, X_2, \ldots, X_m$ be a random partition of X into equal sized parts. Let $X_1 = x_1^1, x_1^2, \ldots, x_1^q$; $X_2 = x_2^1, x_2^2, \ldots, x_2^q$; $\cdots$; $X_m = x_m^1, x_m^2, \ldots, x_m^q$ in sorted order. Here $mq = n$.

We define the rank of any element y in a sequence of keys Y as $|\{z \in Y : z < y\}| + 1$. Let r be any element of X and let X_i be the part in which r is found. If $r = x_i^k$ (i.e., the rank of r in X_i is k) what can we say about the value of k?

Probability that r has a rank of k in X_i is given by

$$P = \frac{\binom{r-1}{k-1}\binom{n-r}{q-k}}{\binom{n-1}{q-1}}.$$

Using the fact that $\binom{a}{b} \le \left(\frac{ae}{b}\right)^b$, we get

$$P \le \frac{\left(\frac{r-1}{k-1}\right)^{k-1}\left(\frac{n-r}{q-k}\right)^{q-k}}{\left(\frac{n-1}{q-1}\right)^{q-1}}$$

Ignoring the -1's and using the fact that $(1-u)^{1/u} \le (1/e)$, we arrive at:

$$P \le \left(\frac{rq/n}{k}\right)^k e^{-(q-k)[r/n-k/q]}.$$

When $k = \frac{rq}{n} + \sqrt{(\alpha+2)q\log_e n + 1}$ (for any fixed α), we get, $P \le n^{-\alpha-2}/e$. Thus the probability that $k \ge \frac{rq}{n} + \sqrt{(\alpha+2)q\log_e n + 1}$ is $\le n^{-\alpha-1}/e$.

In a similar fashion, we can show that the probability that $k \le \frac{rq}{n} - \sqrt{(\alpha+2)q\log_e n + 1}$ is $\le n^{-\alpha-1}/e$. This can be shown by proving that the number of elements in X_i that are greater than r cannot be higher than $\frac{(n-r)q}{n} + \sqrt{(\alpha+2)q\log_e n + 1}$ with the same probability.

Thus, the probability that k is not in the interval

$$\left[\frac{rq}{n} - \sqrt{(\alpha+2)q\log_e n + 1},\quad \frac{rq}{n} + \sqrt{(\alpha+2)q\log_e n + 1}\right]$$

is $\le n^{-\alpha-1}$.

As a consequence, probability that for any r the corresponding k will not be in the above interval is $\le n^{-\alpha}$.

Now consider shuffling the sequences $X_1, X_2, \ldots, X_m$ to get the sequence Z. The position of r in Z will be $(k-1)m + i$. Thus the position of r in Z will be in the interval:

$$\left[r - \frac{n}{\sqrt{q}}\sqrt{(\alpha+2)\log_e n + 1} - \frac{n}{q},\quad r + \frac{n}{\sqrt{q}}\sqrt{(\alpha+2)\log_e n + 1}\right]$$

We get the following Lemma:

Lemma 5.3 *Let X be a set of n arbitrary keys. Partiton X into $m = \frac{n}{q}$ equal sized parts randomly (or equivalently if X is a random permutation of n keys, the first part is the first q elements of X, the second part is the next q elements of X, and so on). Sort each part. Let $X_1, X_2, \ldots, X_m$ be the sorted parts. Shuffle the X_i's to get the sequence Z. At this time, each key in Z will be at most $\frac{n}{\sqrt{q}}\sqrt{(\alpha+2)\log_e n + 1} + \frac{n}{q} \le \frac{n}{\sqrt{q}}\sqrt{(\alpha+2)\log_e n + 2}$ positions away from its final sorted position.* □

Observation 5.2 *Let Z be a sequence of n keys in which every key is at a distance of at most d from its sorted position. Then one way of sorting Z is as follows: Partition Z into subsequences $Z_1, Z_2, \ldots, Z_{n/d}$ where $|Z_i| = d, 1 \le i \le n/d$. Sort each $Z_i (1 \le i \le n/d)$. Merge Z_1 with Z_2, merge Z_3 with Z_4, $\cdots$, merge $Z_{n/d-1}$ with $Z_{n/d}$ (assuming that n/d is even; the case of n/d being odd is handled similarly); Followed by this, merge Z_2 with Z_3, merge Z_4 with Z_5, $\cdots$, and merge $Z_{n/d-2}$ with $Z_{n/d-1}$. Now it can be seen that Z is in sorted order.*

Observation 5.3 *The above discussion suggests a way of sorting n given keys. Assuming that the input permutation is random, one could employ Lemma 5.3 to analyze the expected performance of the algorithm. In fact, the above algorithm is very similar to the LMM sort [20]. Moreover, we need roughly $O(fracn\sqrt{q})$ internal memory to complete each phase of the sorting in one pass.*

5.4 An Expected Two-Pass Algorithm

In this section we present an algorithm that sorts nearly $M\sqrt{M}$ keys when the block size is $\sqrt{M}$. The expectation is over the space of all possible inputs. In particular, this algorithm takes two passes for a large fraction of all possible inputs. Specifically, this algorithm sorts $N = \frac{M\sqrt{M}}{c\sqrt{\log M}}$ keys, where c is a constant to be fixed in the analysis. This algorithm is similar to the one in Section 5.3. Let $N_1 = N/M$.

Algorithm ExpectedTwoPass

1. Form N_1 runs each of length M. Let these runs be $L_1, L_2, \ldots, L_{N_1}$. This takes one pass.

2. In the second pass shuffle these N_1 runs to get the sequence Z (of length N). Perform local sorting as depicted in Section 5.3. Here are the details: Call the sequence of the first M elements of Z as Z_1; the next M elements as Z_2; and so on. In other words, Z is partitioned into $Z_1, Z_2, \ldots, Z_{N_1}$. Sort each one of the Z_i's. Followed by this merge Z_1 and Z_2; merge Z_3 and Z_4; etc. Finally merge Z_2 and Z_3; merge Z_4 and Z_5; and so on.

   Shuffling and the two steps of local sorting can be combined and finished in one pass through the data. The idea is to have two successive Z_i's (call these Z_i and Z_{i+1}) at any time in the main memory. We can sort Z_i and Z_{i+1} and merge them. After this Z_i is ready to be shipped to the disks. Z_{i+2} will then be brought in, sorted, and merged with Z_{i+1}. At this point Z_{i+1} will be shipped out; and so on.

   It is easy to check if the output is correct or not (by keeping track of the largest key shipped out in the previous I/O). As soon as a problem is detected (i.e., when the smallest key currently being shipped out is smaller than the largest key shipped out in the previous I/O), the algorithm is aborted and the algorithm of Lemma 5.2 is used to sort the keys (in an additional three passes).

Theorem 5.2 *The expected number of passes made by Algorithm* ExpectedTwoPass *is very nearly two. The number of keys sorted is* $M\sqrt{\frac{M}{(\alpha+2)\log_e M+2}}$.

Proof. Using Lemma 5.3, every key in Z is at a distance of at most $\leq N_1\sqrt{M}\sqrt{(\alpha+2)\log_e M + 2}$ from its sorted position with probability greater than $(1 - M^{-\alpha}$. We want this distance to be $\leq M$. This happens when $N_1 \leq \sqrt{\frac{M}{(\alpha+2)\log_e M+2}}$.

For this choice of N_1, the expected number of passes made by ExpectedTwoPass is $2(1-M^{-\alpha})+5M^{-\alpha}$ which is very nearly 2. □

As an example, when $M = 10^8$ and $\alpha = 2$, the expected number of passes is $2 + 3 \times 10^{-16}$. Only on at most 10^{-14} % of all possible inputs, ExpectedTwoPass will take more than two passes. Thus this algorithm is of practical importance. Please also note that we match the lower bound of Lemma 5.1 closely.

Observation 5.4 ThreePass2 *can also be modified in a similar manner to obtain an expected two pass algorithm. The columnsort algorithm [14] has eight steps. Steps 1, 3, 5, and 7 involve sorting the columns. In steps 2, 4, 6, and 8 some well-defined permutations are applied on the keys. Chaudhry and Cormen [8] show how to combine the steps appropriately, so that only three passes are needed to sort $M\sqrt{M/2}$ keys on a PDM (with $B = \Theta(M^{1/3})$). Here we point out that this variant of columnsort can be modified to run in an expected two passes. The idea is to skip steps 1 and 2. Using Lemma 5.3, one can show that modified columnsort sorts $M\sqrt{\frac{M}{4(\alpha+2)\log_e M+2}}$ keys in an expected two passes. Contrast this number with the one given in Theorem 5.2.*

5.5 An Expected Three Pass Algorithm

In this section we show how to extend the ideas of the previous section to increase the number of keys to be sorted. In particular, we focus on an expected three pass algorithm. Let N be the total number of keys to be sorted and let $N_2 = N\sqrt{(\alpha+2)\log_e M + 2}/(M\sqrt{M})$.

Algorithm ExpectedThreePass

1. Using ExpectedTwoPass, form runs of length $M\sqrt{\frac{M}{(\alpha+2)\log_e M+2}}$ each. This will take an expected two passes. Now we have N_2 runs to be merged. Let these runs be $L_1, L_2, \ldots, L_{N_2}$.

2. This step is similar to Step 2 of ExpectedTwoPass. In this step we shuffle the N_2 runs formed in Step 1 to get the sequence Z (of length N). Perform local sorting as depicted in ExpectedTwoPass.

   Shuffling and the two steps of local sorting can be combined and finished in one pass through the data (as described in ExpectedTwoPass).

   It is easy to check if the output is correct or not (by keeping track of the largest key shipped out in the previous I/O). As soon as a problem is detected (i.e., when the smallest key currently being shipped out is smaller than the largest key shipped out in the previous I/O), the algorithm is aborted and another algorithm is used to sort the keys. One choice for this alternate algorithm is the seven pass algorithm presented in the next section.

Theorem 5.3 *The expected number of passes made by Algorithm* ExpectedThreePass *is very nearly three. The number of keys sorted is $\frac{M^{1.75}}{[(\alpha+2)\log_e M+2]^{3/4}}$.*

Proof. Here again we make use of Lemma 5.3.

In this case $q = M\sqrt{\frac{M}{(\alpha+2)\log_e M+2}}$. In the sequence Z, each key will be at a distance of at most $N_2 M^{3/4}[(\alpha+2)\log_e M + 2]^{1/4}$ from its sorted position (with probability $\geq (1 - M^{-\alpha})$). We want this distance to be less than M. This happens when $N_2 \leq \frac{M^{1/4}}{[(\alpha+2)\log_e M+2]^{1/4}}$.

For this choice of N_2, the expected number of passes made by ExpectedTwoPass is $3(1-M^{-\alpha})+7M^{-\alpha}$ which is very nearly 3. $\Box$

Observation 5.5 *Chaudhry and Cormen [8] have recently developed a sophisticated variant of columnsort called* subblock columnsort *that can sort $M^{5/3}/4^{2/3}$ keys in four passes (when $B = \Theta(M^{1/3})$). This algorithm has been inspired by the Revsort of Schnorr and Shamir [25]. Subblock columnsort introduces the following step between steps 3 and 4 of columnsort: Partition the $r \times s$ matrix into subblocks of size $\sqrt{s} \times \sqrt{s}$ each; Convert each subblock into a column; and sort the columns of the matrix. At the end of step 3, there could be at most s dirty rows. With the absence of the new step, the value of s will be constrained by $s \leq \sqrt{r/2}$. At the end of the new step, the number of dirty rows is shown to be at most $2\sqrt{s}$. This is in turn because of the fact there could be at most $2\sqrt{s}$ dirty blocks. The reason for this is that the boundary between the zeros and ones in the matrix is monotonous (see Figure 5 in [8]). The monotonicity is ensured by steps 1 through 3 of columnsort. With the new step in place, the constraint on s is given by $r \geq 4s^{3/2}$ and hence a total of $M^{5/3}/4^{2/3}$ keys can be sorted. If one attempts to convert subblock columnsort into a probabilistic algorithm by skipping steps 1 and 2 (as was done in Observation 5.4), it won't work since the monotonicity is not guaranteed. So, converting subblock columnsort into an expected three pass algorithm (that sorts close to $M^{5/3}$ keys) is not feasible. In other words, the new step of forming subblocks (and the associated permutation and sorting) does not seem to help in expectation. On the other hand,* ExpectedThreePass *sorts $\Omega\left(\frac{M^{1.75}}{\log M}\right)$ keys in three passes with high probability.*

5.6 A Seven-Pass Algorithm

In this section we show how to adapt LMM sort to sort M^2 keys on a PDM with $B = \sqrt{M}$. This adaptation runs in seven passes. Let $N = M^2$ be the total number of keys to be sorted.

Algorithm SevenPass

1. Use LMM sort (c.f. Lemma 5.2) to form runs of length $M\sqrt{M}$ each. Now there are $\sqrt{M}$ runs that have to be merged. Let these runs be $L_1, L_2, \ldots, L_{\sqrt{M}}$.

   Use (l, m)-merge to merge these runs, with $l = m = \sqrt{M}$. The tasks involved are listed below.

2. Unshuffle each L_i $(1 \leq i \leq \sqrt{M})$ into $\sqrt{M}$ subsequences $L_i^1, L_i^2, \ldots, L_i^{\sqrt{M}}$.

3. Merge $L_1^1, L_2^1, L_3^1, \ldots, L_{\sqrt{M}}^1$; Let Q_1 be the resultant sequence; Merge $L_1^2, L_2^2, \ldots, L_{\sqrt{M}}^2$; Let Q_2 be the resultant sequence; $\cdots$; and merge $L_1^{\sqrt{M}}, L_2^{\sqrt{M}}, \ldots, L_{\sqrt{M}}^{\sqrt{M}}$; Let $Q_{\sqrt{M}}$ be the resultant sequence. Note that each $Q_i (1 \leq i \leq \sqrt{M})$ is of length $M\sqrt{M}$.

4. Shuffle $Q_1, Q_2, \ldots, Q_{\sqrt{M}}$. Let Z be the shuffled sequence.

5. It can be shown that the length of the dirty sequence in Z is at most M. Clean up the dirty sequence as illustrated in ExpectedTwoPass.

Theorem 5.4 *Algorithm* SevenPass *runs in seven passes and sorts M^2 keys.*

Proof: In accordance with Lemma 5.2, step 1 takes three passes. Step 2 can be combined with step 1. Instead of writing M keys of a run directly into the disks, do the unshuffling and write the unshuffled runs. In step 3 there are $\sqrt{M}$ subproblems each one being that of merging $\sqrt{M}$ sequences of length M each. These mergings can be done in three passes (c.f. Lemma 5.2). Finally, steps 4 and 5 together need only one pass (c.f. ExpectedTwoPass). $\square$

5.7 An Expected Six Pass Algorithm

In this section we show how to adapt SevenPass to get an algorithm that sorts nearly M^2 elements in an expected six passes. Call the new algorithm ExpectedSixPass. This algorithm is the same as SevenPass except that in step 1, we use ExpectedTwoPass to form runs of length $M\sqrt{\frac{M}{(\alpha+2)\log_e M+2}}$ each. This will take an expected two passes. There are $\sqrt{M}$ such runs. The rest of the steps are the same. Of course now the lengths of L_i^j's and Q_i's will be less. We get the following:

Theorem 5.5 ExpectedSixPass *runs in an expected six passes and sorts $\frac{M^2}{\sqrt{(\alpha+2)\log_e M+2}}$ keys.*

6 Conclusions

In this paper we present a simple randomized algorithm for sorting on the PDM that makes only $\widetilde{O}\left(\frac{\log(N/M)}{\log(M/B)}\right)$ passes through the data. Our algorithm uses techniques like staggering of the leading blocks (of streams being merged) and periodic rearrangement of input blocks to prevent clustering of the blocks on any single disk. Note that our bound holds **with high probability** for any value of N

unlike the previous randomized algorithms for which only expected bounds have been proved. In a sense, we are able to retain the advantages of the simplicity of SRM with minimal modification and obtain optimal parallelism for the entire range of the parameters. In our analysis we rely only on standard tools that do not rely on asymptotic convergence. In addition, we are able to adhere to desirable properties like striping and simplicity. The underlying approach in our algorithm is to first generate a random permutation and subsequently sort the random permutation using a simple mergesort.

To the best of our knowledge, the techniques of staggered striping and periodic rearrangement have not been used previously and have the potential for further applications. Implementation of the algorithms and experimental comparison with previous algorithms should be a topic of future investigations. As pointed out earlier, theoretical comparison of the associated constant factors can be misleading since our bounds hold with high probability. Simulations results will not suffice, since it is hard to capture the many advantages of the simplicity of SRM in an actual PDM environment.

We have also presented several algorithms that take a small number of passes for problem sizes of practical interest. In addition, we have presented algorithms with good expected performance. In practice, these could turn out to be the more efficient than the theoretically optimal algorithms.

References

[1] A. Aggarwal and G. Plaxton, Optimal parallel sorting in multi-level storage, *Proc of the ACM-SIAM SODA 1994*, pp. 659 – 668.

[2] A. Aggarwal and J. S. Vitter, The Input/Output Complexity of Sorting and Related Problems, *Communications of the ACM* 31(9), 1988, pp. 1116-1127.

[3] L. Arge, The Buffer Tree: A New Technique for Optimal I/O-Algorithms, *Proc. 4th International Workshop on Algorithms and Data Structures (WADS)*, 1995, pp. 334-345.

[4] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter, On Sorting Strings in External Memory, *Proc. ACM Symposium on Theory of Computing*, 1995.

[5] L. Arge, M. Knudsen, and K. Larsen, A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms, *Proc. Third Workshop on Algorithms and Data Structures (WADS)*, 1993.

[6] R. Barve, E. F. Grove, and J. S. Vitter, Simple Randomized Mergesort on Parallel Disks, *Parallel Computing* 23(4-5), 1997, pp. 601-631.

[7] K. Batcher, Sorting Networks and their Applications, *Proc. AFIPS Spring Joint Computing Conference* 32, 1968, pp. 307-314.

[8] G. Chaudhry and T. H. Cormen, Getting More From Out-of-Core Columnsort, *Proc. 4th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2002, pp. 143-154.

[9] G. Chaudhry, T. H. Cormen, and E. A. Hamon, Parallel Out-of-Core Sorting: The Third Way, to appear in *Cluster Computing*.

[10] G. Chaudhry, T. H. Cormen, and L. F. Wisniewski, Columnsort Lives! An Efficient Out-of-Core Sorting Program, *Proc. 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, 2001, pp. 169-178.

[11] R. Dementiev and P. Sanders, Asynchronous Parallel Disk Sorting, *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 2003, pp. 138-148.

[12] E. Horowitz, S. Sahni, and S. Rajasekaran, *Computer Algorithms*, W. H. Freeman Press, 1998.

[13] D. Hutchinson, P. Sanders and J. Vitter, Duality between prefetching and queued writing with parallel disks, *9th European Symposium on Algorithms 2001*, LNCS 2161, pp. 62 – 73.

[14] T. Leighton, Tight Bounds on the Complexity of Parallel Sorting, *IEEE Transactions on Computers* C34(4), 1985, pp. 344-354.

[15] Y. Ma, S. Sen, D. Scherson, The Distance Bound for Sorting on Mesh Connected Processor Arrays is Tight, *Proc. 27th Symposium on Foundations of Computer Science*, 1986, pp. 255-263.

[16] J.M. Marberg and E. Gafni. Sorting in constant number of row and column phases on a mesh. Algorithmica, 3, 4 (1988), pp. 561–572.

[17] M. Nodine and J. Vitter, Deterministic distribution sort in shared and distributed memory multirocessors, *Proc. of the ACM SPAA 1993*, pp. 120 – 129. Full version available online from *http://www.cs.duke.edu/ jsv/Papers/catalog/node16.html.*

[18] M. H. Nodine and J. S. Vitter, Greed Sort: Optimal Deterministic Sorting on Parallel Disks, *Journal of the ACM* 42(4), 1995, pp. 919-933.

[19] S. Rajasekaran, Sorting and selection on interconnection networks, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science 21*, 1995, pp. 275-296.

[20] S. Rajasekaran, A Framework for Simple Sorting Algorithms on Parallel Disk Systems, *Theory of Computing Systems*, 34(2), 2001, pp. 101-114.

[21] S. Rajasekaran and S. Sen, PDM Sorting Algorithms That Take A Small Number Of Passes, *Proc. International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.

[22] S. Rajasekaran and S. Sen, A Simple Optimal Randomized Sorting Algorithm for the PDM, *Proc. International Symposium on Algorithms and Computation (ISAAC)*, 2005, Springer-Verlag Lecture Notes in Computer Science 3827, 2005, pp. 543-552.

[23] P. Sanders, S. Enger and J. Korst, Fast concurrent access to parallel disks, *Proc of the ACM-SIAM SODA 2000*, pp. 849 – 858.

[24] I. D. Scherson, S. Sen, A. Shamir, "Shear Sort: A True Two-dimensional Sorting Technique for VLSI Networks," Proc. International Conf. on Parallel Processing, pp. 903-908, 1986.

[25] C. P. Schnorr and and A. Shamir, An optimal sorting algorithm for mesh connected computers, *Proc. 18th Annual ACM Symposium on Theory of Computing*, 1986, pp. 255-263.

[26] C.D. Thompson and H.T. Kung, Sorting on a Mesh Connected Parallel Computer, *Communications of the ACM* 20(4), 1977, pp. 263-271.

[27] L. Valiant and G. Brebner. Universal schemes for parallel communication, *Proc. of Thirteenth ACM STOC*, 1981, pp. 263 – 277.

[28] J. S. Vitter and D. A. Hutchinson, Distribution Sort with Randomized Cycling, *Proc. 12th Annual SIAM/ACM Symposium on Discrete Algorithms*, 2001.

[29] J. S. Vitter and E. A. M. Shriver, Algorithms for Parallel Memory I: Two-Level Memories, *Algorithmica* 12(2-3), 1994, pp. 110-147.

Appendix A: Chernoff bounds

If a random variable X is the sum of n iid Bernoulli trials with a success probability of p in each trial, the following equations give us concentration bounds of deviation of X from the expected value of np. The first equation is more useful for large deviations whereas the other two are useful for small deviations from a large expected value.

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

$$Prob(X \leq (1-\epsilon)pn) \leq exp(-\epsilon^2 np/2) \tag{2}$$

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

for all $0 < \epsilon < 1$.