PDM Sorting Algorithms That Take A Small Number Of Passes

Sanguthevar Rajasekaran
Dept. of Comp. Sci. and Engg.
University of Connecticut, Storrs, CT
rajasek@engr.uconn.edu

Sandeep Sen
Dept. of Comp. Science
IIT, New Delhi
ssen@cse.iitd.ernet.in

Abstract

We live in an era of data explosion that necessitates the discovery of novel out-of-core techniques. The I/O bottleneck has to be dealt with in developing out-of-core methods. The Parallel Disk Model (PDM) has been proposed to alleviate the I/O bottleneck. Sorting is an important problem that has ubiquitous applications. Several asymptotically optimal PDM sorting algorithms are known and now the focus has shifted to developing algorithms for problem sizes of practical interest. In this paper we present several novel algorithms for sorting on the PDM that take only a small number of passes through the data. We also present a generalization of the zero-one principle for sorting. A shuffling lemma is presented as well. These lemmas should be of independent interest for average case analysis of sorting algorithms as well as for the analysis of randomized sorting algorithms.

1. Introduction

The Parallel Disk Model (PDM) has been proposed to deal with the problem of developing effective algorithms for processing voluminous data. In a PDM, there is a (sequential or parallel) computer $C$ 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 of $C$. One typically assumes that the main memory of $C$ is sized $M$ where $M$ is a small constant multiple of $DB$.

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 [1] that $\Omega \left( \frac{N}{DB} \log(\frac{N}{B}) \right)$ I/O operations are needed to sort $N$ keys (residing in $D$ disks) when the block size is $B$. Here $M$ is the size of the internal memory. Many asymptotically optimal algorithms have been devised as well (see e.g., Arge [2], Nodine and Vitter [21], and Vitter and Hutchinson [29]). The LMM sort of Rajasekaran [23] is optimal when $N$, $B$, and $M$ are polynomially related and is a generalization of Batchter’s odd-even merge sort [6], Thompson and Kung’s $s^2$-way merge sort [28], and Leighton’s columnsort [15].

Recently, many algorithms have been devised for problem sizes of practical interest. For instance, Demetie 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 [9] 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 $\frac{DB}{B}$ read I/O operations and the same number of write operations.) A typical assumption made in developing PDM algorithms is that the internal memory size $M$ is a small multiple of $DB$. In [7], Chaudhry and Cormen introduce some sophisticated engineering tools to speedup the algorithm of [9] 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 [8], Chaudhry, Cormen, and Hamon present an algorithm that sorts $M^{5/3}A^{2/3}$ keys (when $B = \Theta(M^{2/3}))$. They combine columnsort and Revsort of Schnorr and Shamir [26] 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 [23] and all the algorithms in this paper (except for the integer sorting algorithm) are oblivious.

* This research has been supported in part by the NSF Grants CCR-9912395 and ITR-0326155.
† This research was conducted when this author was visiting the University of Connecticut.
In this paper we focus on developing sorting algorithms that take 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 $M^2 = 10^{16}$ (integers) (which is around 100,000 terabytes). Thus we focus on input sizes of $\leq M^2$ in this paper. 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 $A$ 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, $A$ will take more than two passes. Thus algorithms of this kind will be of great practical importance.

**New Results:** There are two main contributions in this paper: 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 two lemmas 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. In this paper we also point out that radix sort can be useful in PDM sorting of integer keys. And hence it should be of independent interest. In this paper we focus on developing sorting algorithms based on LMM sort that also assumes that $B = \sqrt{M}$. In contrast, the algorithm of Chaudhry and Cormen [7] uses a block size of $M^{1/3}$ and sorts $M \sqrt{M}/\sqrt{2}$ keys in three passes. 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) An expected six pass algorithm that sorts nearly $M^2$ keys; and 7) A simple integer sorting algorithm that sorts integers in the range $[1, M^c]$ (for any constant $c$) in a constant number of passes (for any input size). Unlike the bundle-sorting algorithm of Matias, Segal and Vitter [20] that is efficient only for a single disk model, our algorithm achieves full parallelism under very mild assumptions.

**Notation.** We say the amount of resource (like time, space, etc.) used by a randomized or probabilistic algorithm is $\tilde{O}(f(n))$ 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 $\alpha$ is a constant $\geq 1$. We could also define the asymptotic functions $\Theta(\cdot), \tilde{\Theta}(\cdot), \text{etc.}$ in a similar manner.

### 2. A Lower Bound

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

**Lemma 2.1** 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 proven in [4]. In particular, it has been shown in [4] that $\log(N!)$ is $N \log B + I \times (B \log((M - B)/B) + 3B)$, 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

$$I \geq \frac{2M(1 + \frac{1}{16})}{(1 + \log M)}.$$ 

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. □
3. A Generalized Zero-One Principle with application to PDM sorting

3.1. A Mesh-Based Algorithm

In this section we describe a very 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 ThreePass1

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 in the worst case.

Theorem 3.1 Algorithm ThreePass1 sorts the $M \times \sqrt{M}$ data items correctly for all inputs.

Proof: Our proof is based on 0-1 principle so that we only have to analyze inputs consisting of 0’s or 1’s. Central to this algorithm as well as others like [26, 18, 19] is the notion of dirty rows/columns/blocks. Definition Dirty A row/column is dirty if it contains a mixture of 0’s and 1’s. Similarly a block is dirty if it contains a mixture of 0’s and 1’s.

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 [27]. This implies at most two dirty $\sqrt{M}/2$ sub-meshes. Step 3 cleans up these in a manner similar to [18].

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. □

3.2. Average Case analysis

We now modify the algorithm by eliminating the first step and reanalyze it. This version of the algorithm called ExpThreePass1 takes 2 passes for a large fraction of the input permutations. More precisely.

Theorem 3.2 Algorithm ExpThreePass2 sorts $M \sqrt{\frac{M}{\alpha \log_{\alpha} M}}$ data items correctly in expected 2 passes, for some constant $\alpha$.

We prove and use a generalization of the 0-1 principle for this purpose. The traditional zero-one principle for sorting networks states that “if a network with $n$ input lines sorts all $2^n$ binary sequences into nondecreasing order, then it will sort any arbitrary sequence of $n$ numbers into nondecreasing order.” We prove a generalization of the 0-1 principle to sorting networks that sort almost all possible binary sequences. This generalization will be useful in the average case analysis of sorting algorithms as well as in the analysis of randomized sorting algorithms. It may be noted that the 0-1 principle extends to oblivious sorting algorithms [16] and although our results are stated in the context of sorting networks, they are applicable to oblivious sorting algorithms also.

Theorem 3.3 (Generalized 0-1 principle) Let $S_k$ denote the set of length $n$ binary strings with exactly $k$ 0’s or 1’s. Central to this algorithm as well as others like [26, 18, 19] is the notion of dirty rows/columns/blocks. Definition Dirty A row/column is dirty if it contains a mixture of 0’s and 1’s. Similarly a block is dirty if it contains a mixture of 0’s and 1’s.

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 [27]. This implies at most two dirty $\sqrt{M}/2$ sub-meshes. Step 3 cleans up these in a manner similar to [18].

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. □

4. LMM based algorithms and a shuffling lemma

In this section we present another three-pass algorithm for sorting on the PDM based on the $(l, m)$-merge sort (LMM sort) algorithm of Rajasekaran [23]. 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 at a distance of \( \leq lm \) from its final sorted position. Perform local sorting to move each key to its sorted position.

Columnsort algorithm [15], odd-even merge sort [6], and the \( s^2 \)-way merge sort algorithms are all special cases of LMM sort [23].

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 ThreePass2**

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 shown in [23].

We get the following:

**Lemma 4.1** LMM sort sorts \( M \sqrt{M} \) keys in three passes through the data when the block size is \( \sqrt{M} \).

**Observation 4.1** Chaudhry and Cormen [7] have shown that Leighton’s columnsort algorithm [15] 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 4.1 (based on LMM sort) sorts \( M^{1.5} \) keys in three passes.

### 4.1. A Shuffling Lemma

In this section we prove a lemma (we call the shuffling lemma) that will be useful in the analysis of expected performance of sorting algorithms. Though the generalized zero-one principle can be employed in its place, this lemma yields better constants than the generalized zero-one principle.

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 \( X_1 = x_1^1, x_2^1, \ldots, x_i^1 \); \( X_2 = x_1^2, x_2^2, \ldots, x_i^2 \); \( \cdots \); \( X_m = x_1^m, x_2^m, \ldots, x_i^m \) in sorted order. Here \( mq = n \).

We define the rank of any element \( y \) in a sequence of keys \( Y \) as \( \left\lfloor \{ z \in Y : z < y \} \right\rfloor + 1 \). 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

\[
P = \frac{(r-1)(n-r)}{(q-1)}.
\]

Using the fact that \( \binom{a}{b} \leq \left( \frac{ea}{b} \right)^b \), we get

\[
P \leq \frac{(r-1)^{k-1}}{(q-1)} \left( \frac{n-r}{q-k} \right)^{q-k}.
\]

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

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

When \( k = \frac{rn}{n} + \sqrt{(\alpha + 2)q \log_e n + 1} \) (for any fixed \( \alpha \)), we get, \( P \leq n^{-\alpha - 2}/e \). Thus the probability that \( k \geq \frac{rn}{n} + \sqrt{(\alpha + 2)q \log_e n + 1} \leq n^{-\alpha - 1}/e \).

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

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

is \( \leq n^{-\alpha - 1} \).

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

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 \( (k - 1) + i \). Thus the position of \( r \) in \( Z \) will be in the interval:
We get the following Lemma:

Lemma 4.2 Let \( X \) be a set of \( n \) arbitrary keys. Partition \( X \) 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

\[
\frac{n}{q} \sqrt{(\alpha + 2) \log_c n + 1} - \frac{n}{q} r + \frac{n}{q} \sqrt{(\alpha + 2) \log_c n + 1}
\]

distance away from its sorted position.

Observation 4.2 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 \( |Z_i| = d, 1 \leq i \leq n/d \). Sort each \( Z_i \); merge \( Z_1 \) with \( Z_2 \), merge \( Z_3 \) with \( Z_4 \), \ldots, 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 \( Z_3 \) with \( Z_4 \), merge \( Z_5 \) with \( Z_6 \), \ldots, and merge \( Z_{n/d-2} \) with \( Z_{n/d-1} \). Now it can be seen that \( Z \) is in sorted order.

Observation 4.3 The above discussion suggests a way of sorting \( n \) keys given. Assuming that the input permutation is random, one could employ Lemma 4.2 to analyze the expected performance of the algorithm. In fact, the above algorithm is very similar to the LMM sort [23].

5. An Expected Two-Pass Algorithm

In this section we present an algorithm that sorts nearly \( M \sqrt{M} \) keys when the block size is \( \sqrt{M} \) in an expected two passes. 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 \( N = M^{\frac{c}{\log_c M}} \) keys, where \( c \) is a constant to be fixed in the analysis. This algorithm is similar to the one in Section 4.1. 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 4.1. 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 4.1 is used to sort the keys (in an additional three passes).

Theorem 5.1 The expected number of passes made by Algorithm ExpectedTwoPass is very nearly two. The number of keys sorted is \( M \sqrt{(\alpha + 2) \log_c M + 2} \).

Proof. Using Lemma 4.2, every key in \( Z \) is at a distance of at most \( \leq N_1 \sqrt{M} \sqrt{(\alpha + 2) \log_c M + 2} \) from its sorted position with probability \( \geq 1 - M^{-\alpha} \). We want this distance to be \( \leq M \). This happens when \( N_1 \leq \sqrt{M} (\alpha + 2) \log_c M + 2 \).

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 \( 2 + 3 \times 10^{-10} \). 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 2.1 closely.

Observation 5.1 The columnsort algorithm [15] 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 [7] 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 4.2, one can show that modified columnsort sorts \( M \sqrt{4(\alpha + 2) \log_c M + 2} \) keys in an expected two passes. Contrast this number with the one given in Theorem 5.1.
6. Increasing the data size

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

\[ \text{Theorem 6.1} \quad \text{The expected number of passes made by Algorithm ExpectedThreePass is very nearly three. The number of keys sorted is} \quad \frac{M^{1.75}}{\log(\alpha + 2) + 2}. \]

\[ \text{Proof.} \quad \text{Here again we make use of Lemma 4.2.} \]

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

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. \( \Box \)

\[ \text{Observation 6.1} \quad \text{Chaudhry and Cormen [7] have recently developed a sophisticated variant of columnsort called subblock columnsort that can sort} \quad M^{5/3} \quad \text{keys in four passes (when} \quad B = \Theta(M^{1/3}) \). \text{This algorithm has been inspired by the Revsort of Schmorr and Shamir [26]. Subblock columnsort introduces the following step between steps 3 and 4 of columnsort: Partition the} \quad r \times s \quad \text{matrix into subblocks of size} \quad \sqrt{s} \times \sqrt{s} \quad \text{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} \quad s \quad \text{dirty rows. With the absence of the new step, the value of} \quad s \quad \text{will be constrained by} \quad s \leq \sqrt{r/2}. \quad \text{At the end of the new step, the number of dirty rows is shown to be at most} \quad 2\sqrt{5}. \quad \text{This is in turn because of the fact there could be at most} \quad 2\sqrt{5} \quad \text{dirty blocks. The reason for this is that the boundary between the zeros and ones in the matrix is monotonous (see Figure 5 in [7]).} \quad \text{The monotonicity is ensured by steps 1 through 3 of columnsort. With the new step in place, the constraint on} \quad s \quad \text{is given by} \quad r \geq 5s^3/2 \quad \text{and hence a total of} \quad M^{5/3}/A^{2/3} \quad \text{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 5.1), it won't work since the monotonicity is not guaranteed. So, converting subblock columnsort into an expected three pass algorithm (that sorts close to} \quad M^{5/3} \quad \text{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} \quad \Omega \left( \frac{M^{1.75}}{\log M} \right) \quad \text{keys in three passes with high probability.} \]

6.1. Sorting \( M^2 \) elements

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.

\[ \text{Algorithm SevenPass} \]

1. Use LMM sort (c.f. Lemma 4.1) to form runs of length \( M \sqrt{M} \) each. Now there are \( \sqrt{M} \) runs that have to be merged. Let these runs be \( L_1, L_2, \ldots, L_{\sqrt{M}} \).

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

3. Merge \( L_1, L_2, L_3, \ldots, L_{\sqrt{M}} \). Let \( Q_1 \) be the resultant sequence; Merge \( L_1, L_2, \ldots, L_{\sqrt{M}} \). 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.

\[ \text{Theorem 6.2} \quad \text{Algorithm SevenPass runs in seven passes and sorts} \quad M^2 \quad \text{keys.} \]
6.2. 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 $M \sqrt{\frac{M}{(\alpha + 2) \log M + 2}}$ each. This will take an expected two passes. There are \( M \) such runs. The rest of the steps are the same. Of course now the lengths of $L_i$'s and $Q_i$'s will be less. We get the following:

**Theorem 6.3** ExpectedSixPass runs in an expected six passes and sorts $M^2$ keys.

**Remark** We have designed matching mesh-based algorithms. At this point it is not clear if they offer any advantage. We will provide details in the full version.

7. Optimal Integer Sorting

Often, the keys to be sorted are integers in some range $[1, R]$. Numerous sequential and parallel algorithms have been devised for sorting integers. Several efficient out-of-core algorithms have been devised by Arge, Ferragina, Grossi, and Vitter [3] for sorting strings. For instance, three of their algorithms have the I/O bounds of 
\[ O\left(\frac{N}{F} \log \frac{M}{B} \right), \quad O\left(\frac{N}{F_{\mathbb{B}}} \log \frac{M}{B} \frac{N}{F_{\mathbb{B}}} + \frac{N}{F_{\mathbb{B}}} \right),\]
and 
\[ O\left(\frac{N}{F_{\mathbb{B}}} \log \frac{M}{B} \frac{N}{F_{\mathbb{B}}} + \frac{N}{F_{\mathbb{B}}} \log \frac{M}{B} |\Sigma| \right),\]
respectively. These algorithms sort $K$ strings with a total of $N$ characters from the alphabet $\Sigma$. 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 radix sort (see e.g., [13]) 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. For 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 bundle-sorting algorithm of Matias, Segal and Vitter [20] can be applied to this scenario but it is efficient only for a single disk model.

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 stripping method suggested in [23] is used. Some of the blocks could be nonfull. 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 $R = M/B$. More details follow.

**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 $B_1, B_2, \ldots, B_R$. If there are $N_i$ keys in $B_i$, then, $\lceil N_i/B \rceil$ blocks will be formed out of $B_i$ (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 [23]). The number of write steps needed is $\max\{\lceil N_i/B \rceil\}$.

A. Read the keys written to the disks and write them back so that the keys are placed contiguously across the disks.

**Theorem 7.1** 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 assuming that $B = \Omega(\log N)$. If step A is not needed, the number of passes is $(1 + \mu)$ and if step A 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 $[(1−ε)CB, (1+ε)CB]$ with probability $≥ 1 − 2 \exp(−ε^2CB/3)$. Thus, the number of keys in every bucket is in this interval with probability $≥ 1 − \exp(−ε^2CB/3 + \ln(2R))$. This probability will be $≥ (1 − N^{−(α+1)})$ as long as $B ≥ \frac{3}{Cε^2}(α + 1)\ln N$. This is readily satisfied in practice (since the typical assumption on $B$ is that it is $Ω(M^δ)$ for some fixed $δ > 1/3$).

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

Thus, with probability $≥ (1 − N^{−α})$, IntegerSort takes $(1 + μ)$ passes excluding step A and $2(1 + μ)$ passes including step A. □

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

**Observation 7.1** The sorting algorithms of [29] 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 in that 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 $(1 + δ)\frac{\log(N/M)}{\log(M/B)}$ do

1. Employ IntegerSort to sort the keys with respect to their $ith$ most significant $\log(M/B)$ bits.

A. Now the size of each bucket is $≤ M$.

Read and sort the buckets.

**Theorem 7.2** $N$ random integers in the range $[1, R]$ (for any $R$) can be sorted in an expected $(1 + ν)\frac{\log(N/M)}{\log(M/B)} + 1$ passes through the data, where $ν$ is a constant $< 1$ provided that $B = Ω(\log N)$. In fact, this bound holds for a large fraction $≥ 1 − N^{−α}$ for any fixed $α ≥ 1$ of all possible inputs.

**Proof.** In accordance with Theorem 7.1, each run of step 1 takes $(1 + μ)$ passes. Thus RadixSort takes $(1 + μ)(1 + δ)\frac{\log(N/M)}{\log(M/B)}$ passes. This number is $(1 + ν)\frac{\log(N/M)}{\log(M/B)}$ for some fixed $ν < 1$.

It remains to show that after $(1 + δ)\frac{\log(N/M)}{\log(M/B)}$ runs of step 1, the size of each bucket will be $≤ M$. At the end of the first run of step 1, the size of each bucket is expected to be $\frac{NB}{M}$. Using Chernoff bounds, this size is $≤ (1 + ε)\frac{NB}{M}$ with high probability, for any fixed $ε < 1$. After $k$ (for any integer $k$) runs of step 1, the size of each bucket is $≤ N(1 + ε)^k(B/M)^k$ with high probability. This size will be $≤ M$ for $k ≥ \frac{\log(N/M)}{\log(1+ε)}$. The RHS is $≤ (1 + δ)\frac{\log(N/M)}{\log(M/B)}$ for any fixed $δ < 1$.

Step A takes nearly one pass. □

**Observation 7.2** 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.

8. Conclusions

In this paper we have presented several novel algorithms for sorting on the PDM. We believe that these algorithms will perform well in practice. For sorting $M\sqrt{M}$ keys, both ThreePass1 and ThreePass2 seem to have similar performance. They can sort slightly more keys than that of [7]. Both ThreePass1 and ThreePass2 use a block size of $\sqrt{M}$ and the algorithm of [7] uses a block size of $M^{1/3}$. Lemma 2.1 yields a lower bound of 1.75 passes when $B = M^{1/3}$ and 2 passes when $B = \sqrt{M}$ (when $N = M\sqrt{M}$).

For sorting more than $M\sqrt{M}$ keys, LMM sort seems to be the best. For instance, SevenPass sorts $M^2$ keys whereas our mesh based algorithm sorts $M^2/4$ keys. We believe that combining mesh-based techniques with those of [23] and [7] will yield even better results.

References

A. Proof of the generalized 0-1 principle

Definition: A string \( s \in \{0,1\}^n \) is a k-string if it has exactly \( k \) 0’s, \( 0 \leq k \leq n \).

The set of all \( k \)-strings for a fixed \( k \) is called a \( k \)-set and will be denoted by \( S_k \).

Observation A.1 \( S_k \)'s are pairwise disjoint and their union consists of all possible \( 2^n \) strings over \( \{0,1\} \).

Let \( A^{(n)} \), \( B^{(n)} \) be two totally ordered multisets of \( n \) elements each. A bijection \( f : A^{(n)} \rightarrow B^{(n)} \) is monotone if for all \( x, y \in A^{(n)} \) and \( x \leq y \), \( f(x) \leq f(y) \).

Observation A.2 If \( f : A^{(n)} \rightarrow B^{(n)} \) is monotone then the inverse of \( f \) is also monotone.

The correctness of the standard zero-one principle is based on the following elegant result (see Knuth[14] for a proof).

Theorem A.1 For any sorting circuit \( C \) and a monotone function \( f \), \( f(C(a)) = C(f(a)) \).

Given \( I_n = \{1, 2, \ldots, n\} \), the only monotone function between \( I_n \) and strings in \( S_k \) is given by \( f_k(j) = 0 \) for \( j \leq k \) and 1 otherwise. The extension of \( f \) to a sequence \( a = (a_1, a_2, \ldots) \) is given by \( f(a_1), f(a_2), \ldots \).

From our previous observation \( f_k^{-1} \) is also monotone. From the previous theorem it follows that

Lemma A.1 If a sorting circuit does not sort some \( a \in S_k \) then it does not sort (the permutations corresponding to) \( f_k^{-1}(a) \). Conversely, if the circuit correctly sorts \( f_k(\sigma) \) for all \( k \), for an input permutation \( \sigma \), then it correctly sorts \( \sigma \).

Consider the mapping between permutations of \( I_n \) and strings \( a \in S_k \) such that \( a_i = f_k(\Pi(i)) \) for a permutation \( \Pi \). This can be represented as a bipartite graph \( G_k \) with \( n! \) elements in one set and \( |S_k| \) on the other (for each \( k \)). Note that a single permutation can map to exactly one string in \( S_k \). Using simple symmetry arguments, it follows that
Lemma A.2 Each set in the bipartite graph $G_k$ has has vertices with equal degree. In particular, the vertices representing $S_k$ have degrees equal to $\frac{n!}{|S_k|}$.

If the circuit does not sort some permutation $\Pi$ of $I_n$ then it does not sort one (or more) string in $\{0, 1\}^n$, say $a \in S_k$ for some $k$. Conversely, if the circuit does not sort some string $a \in S_k$ then from Lemma A.1 it does not sort any of the permutations in the inverse map. Consider the graph $G_k$ where we mark all the nodes corresponding to the permutations that are not sorted correctly and likewise we mark the nodes of $S_k$ that are not sorted correctly. For a fixed $k$, if the circuit does not sort $\beta |S_k|$ ($\beta < 0$) strings it does not sort $\beta |S_k| \cdot \frac{n!}{|S_k|}$ permutations. Therefore the total fraction of permutations (over all values of $k$) that may not get sorted correctly is $\beta \cdot (n + 1)$. Setting $\alpha = 1 - \beta$ completes the proof of Theorem 3.3.

Corollary: If for some $k$, the sorting circuit does not sort any string in $S_k$, then it does not work correctly on any permutation.

Remark: Note that the number of strings in the set $S_p$, $0.49n \leq p \leq 0.51n$ form an overwhelming fraction of all length $n$ binary strings. One can design many sorting algorithms that work correctly for $S_p$ but which sort only a negligible fraction of $S_{\log n}$ (for example). The above corollary rules out strengthening the main result in the following obvious way -

If a sorting network sorts most binary strings then it sorts most arbitrary inputs.

Chlebus[10] claimed an ad-hoc version of Lemma A.1 but didn’t follow it up with any of the latter arguments thus leaving the connection imprecise and incomplete.