Chapter 1

SORTING AND SELECTION ON PARALLEL DISK MODELS

Sanguthevar Rajasekaran  
Department of CISE, Univ. of Florida  
raj@cise.ufl.edu

Abstract  
Data explosion is an increasingly prevalent problem in every field of science. Traditional out-of-core models that assume a single disk have been found inadequate to handle voluminous data. As a result, models that employ multiple disks have been proposed in the literature. For example, the Parallel Disk Systems (PDS) model assumes \( D \) disks and a single computer. It is also assumed that a block of data from each of the \( D \) disks can be fetched into the main memory in one parallel I/O operation.

In this article, we survey sorting and selection algorithms that have been devised for out-of-core models assuming multiple disks. We also consider practical implementations of parallel disk models.

1. INTRODUCTION

Computing applications have advanced to a stage where the data involved is humongous. The volume of data calls for the use of secondary storage devices such as disks. Even a single disk may not be enough to handle I/O operations efficiently. Thus researchers have proposed models with multiple disks.

One of the models (which refines prior models) that has been studied well is the Parallel Disk Systems (PDS) [25]. In this model there are \( D \) disks and a single computer (sequential or parallel). In one parallel I/O operation, a block of data from each of the \( D \) disks can be brought into the main memory of the computer. A block consists of \( B \) records. We usually require that \( M \geq 2DB \), where \( M \) is the internal memory size. At least \( DB \) amount of memory is needed to store the data fetched from the disks and the remaining part of the main memory can be used to overlap
local computations with I/O operations. Algorithmists have designed algorithms for various fundamental problems on the PDS model. In the analysis of these algorithms, only I/O operations are counted since the local computations are usually very fast.

In this article, we survey sorting and selection algorithms that have been proposed for models with multiple disks. We also investigate the issue of implementing such models in practice. In Section 2 we present details of the PDS model. Sections 3 and 4 are devoted to a survey of sorting and selection algorithms, respectively. In Section 5 we study the problem of implementing parallel disk models in practice. Section 6 concludes the article.

2. PARALLEL DISK SYSTEMS

Here we give more details of the PDS model. A PDS consists of a computer (this could be sequential or parallel) together with \( D \) disks. For any given problem, the input will be given in the disks and the output also is expected to be written in the disks. In one I/O operation, a block of \( B \) records can be brought into the core memory of the computer from each one of the \( D \) disks. In analyzing the time complexity of any algorithm on the PDS model, we consider only the number of parallel I/O operations and neglect the local computation time since the later is usually much smaller.

If \( M \) is the internal memory size of the computer, then one usually requires that \( M \geq 2DB \). A portion of this memory is used to store operational data whereas the other portion is used for storing prefetched data that enables overlap of local computations with I/O operations. From hereon, \( M \) is used to refer to only \( DB \).

The sorting problem on the PDS can be defined as follows. There are a total of \( N \) records to begin with so that there are \( \frac{N}{D} \) records in each disk. The problem is to rearrange the records such that they are in either ascending order or descending order with \( \frac{N}{D} \) records ending up in each disk. For the selection problem also, each disk will have \( \frac{N}{D} \) input keys to begin with. The output is through any standard I/O device associated with the computer.

3. SORTING RESULTS ON THE PDS MODEL

The problem of external sorting has been widely explored owing to its paramount importance. Given a sequence of \( n \) keys, the problem of sorting is to rearrange this sequence in either ascending or descending order. It is easy to show that a lower bound on the number of
I/O read steps for parallel disk sorting is \( \Omega \left( \frac{N}{DB} \left[ \frac{\log(N/B)}{\log(M/B)} \right] \right) \). Here \( N \) is the number of records to be sorted and \( M \) is the internal memory size of the computer. \( B \) is the block size and \( D \) is the number of parallel disks used. Numerous asymptotically optimal algorithms that make \( O \left( \frac{N}{DB} \left[ \frac{\log(N/B)}{\log(M/B)} \right] \right) \) I/O read steps have been proposed (see e.g., [16, 1, 3]).

In the model of Aggarwal and Vitter [2], each I/O operation results in the transfer of \( D \) blocks each block having \( B \) records. A refinement of this model was envisioned in [25]. Many asymptotically optimal algorithms have been developed for sorting on this model. Nodine and Vitter’s optimal algorithm [15] involves the solution of certain matching problems. Aggarwal and Plaxton’s optimal algorithm [1] is based on the Sharesort algorithm of Cypher and Plaxton (which was originally offered for the hypercube model). An optimal randomized algorithm was given by Vitter and Shriver for disk sorting [25]. Though these algorithms are highly nontrivial and theoretically interesting, the underlying constants in their time bounds are high.

The algorithm that people use in practice is the simple disk-striped mergesort (DSM) [4], even though it is not asymptotically optimal. DSM is simple and the underlying constant is small. In any I/O operation, DSM accesses the same portions of the \( D \) disks. This has the effect of having a single disk which can transfer \( DB \) records in a single I/O operation. DSM is basically an \( MDB \)-way mergesort. To start with, initial runs are formed in one pass through the data. After this, the disks have \( N/M \) runs each of length \( M \). Next, \( MDB \) runs are merged at a time. Blocks of any run are uniformly striped across the disks so that in future they can be accessed in parallel utilizing the full bandwidth.

Each phase of merging can be done with one pass through the data. There are \( \log(N/M) \) phases and hence the total number of passes made by DSM is \( \frac{\log(N/M)}{\log(M/DB)} \), i.e., the total number of I/O read operations called for by DSM is \( \frac{N}{DB} \left( 1 + \frac{\log(N/M)}{\log(M/DB)} \right) \). Note that the constant here is just 1.

If \( N \) is a polynomial in \( M \) and \( B \) is small (which are readily satisfied in practice), the lower bound simply yields \( \Omega(1) \) passes. All the optimal algorithms mentioned above make only \( O(1) \) passes. Thus the challenge in the design of sorting algorithms lies in reducing this constant. If \( M = 2DB \), the number of passes made by DSM is \( 1 + \log(N/M) \), which is \( \omega(1) \).

Some recent works specifically focus on the design of practical sorting algorithms. For example, Pai, Schaffer, and Varman [17] analyzed the average case performance of a simple merging algorithm, with the help
of an approximate model of average case inputs. Barve, Grove, and Vitter [4] have proposed a simple randomized algorithm (SRM) and analyzed its performance. The analysis of SRM involves the solution of certain occupancy problems. The expected number $\text{Read}_{SRM}$ of I/O read operations needed in their algorithm is such that

$$\text{Read}_{SRM} \leq \frac{N}{DB} + \frac{N}{DB} \frac{\log(N/M)}{\log kD} \frac{\log D}{k \log \log D} \left( 1 + \frac{\log \log \log D}{\log \log D} + \frac{1 + \log k}{\log \log D} + O(1) \right).$$  

(1.1)

SRM merges $R = kD$ runs at a time, for some integer $k$. The expected performance of SRM is optimal when $R = \Omega(D \log D)$. However, in this case, the internal memory needed is $\Omega(BD \log D)$. They have also compared SRM with DSM through simulations and shown that SRM performs better than DSM.

Recently, Rajasekaran [20] has presented an algorithm (called $(\ell, m)$-merge sort (LMM)) that is asymptotically optimal when $N$ is a polynomial in $M$ and $B$ is small. The algorithm is as simple as DSM. LMM makes less number of passes through the data than DSM when $D$ is large.

Problems such as FFT computations (see e.g., [6]), selection (see e.g., [21]), etc. have also been studied on the PDS model.

Next we present details of the LMM algorithm.

### 3.1 THE $(\ell, M)$-MERGE SORT (LMM)

Most of the sorting algorithms that have been developed for the PDS are based on merging. To begin with, these algorithms form $\frac{N}{M}$ runs each of length $M$. A run refers to a sorted subsequence. These initial runs can be formed in one pass through the data (or equivalently $\frac{N}{DB}$ parallel I/O operations). Thereafter the algorithms merge $R$ runs at a time. Let a phase of merges stand for the task of scanning through the input once and performing $R$-way merges. Note that each phase of merges will reduce the number of remaining runs by a factor of $R$. For instance, the DSM algorithm employs $R = \frac{M}{DB}$. The difference among the above sorting algorithms lies in how each phase of merges is done.

LMM of [20] is also based on merging. It fixes $R = \ell$, for some appropriate $\ell$. The LMM generalizes such algorithms as the odd-even merge sort, the $s^2$-way merge sort of Thompson and Kung [24], and the columnsort algorithm of Leighton [12].

The well known odd-even mergesort algorithm has $R = 2$. It repeatedly merges two sequences at a time. There are $n$ sorted runs each of length 1 to begin with. Thereafter the number of runs decreases by a factor of 2 with each phase of merges. The odd-even merge algorithm
is used to merge any two sequences. A description of odd-even merge follows.

1) Let \( U = u_1, u_2, \ldots, u_q \) and \( V = v_1, v_2, \ldots, v_q \) be the two sorted sequences to be merged. Unshuffle \( U \) into two: \( U_{\text{odd}} = u_1, u_3, \ldots, u_{q-1} \) and \( U_{\text{even}} = u_2, u_4, \ldots, u_q \). Likewise unshuffle \( V \) into \( V_{\text{odd}} \) and \( V_{\text{even}} \).

2) Recursively merge \( U_{\text{odd}} \) with \( V_{\text{odd}} \). Let \( X = x_1, x_2, \ldots, x_q \) be the resultant sequence. Also merge \( U_{\text{even}} \) with \( V_{\text{even}} \) to get \( Y = y_1, y_2, \ldots, y_q \).

3) Shuffle \( X \) and \( Y \) to form the sequence: \( Z = x_1, y_1, x_2, y_2, \ldots, x_q, y_q \).

4) Do one step of compare-exchange operation, i.e., sort successive subsequences of length two in \( Z \). In particular, sort \( y_1, x_2 \); sort \( y_2, x_3 \); and so on. The resultant sequence is the merge of \( U \) and \( V \).

The zero-one principle can be used to prove the correctness of this algorithm. Thompson and Kung’s algorithm [24] is a generalization of the above algorithm. In this algorithm, \( R \) takes on the value \( s^2 \) for some appropriate function \( s \) of \( n \). At any given time \( s^2 \) runs are merged using an algorithm similar to odd-even merge.

LMM generalizes \( s^2 \)-way merge sort. LMM employs \( R = \ell \). The number of runs is reduced by a factor of \( \ell \) by each phase of merges.

At any given time, \( \ell \) runs are merged using the \((\ell, m)\)-merge algorithm. This merging algorithm is similar to the odd-even merge except that in Step 2, the runs are \( m \)-way unshuffled (instead of 2-way unshuffling). In Step 3, \( m \) sequences are shuffled and also in Step 4, the local sorting is done differently. A description of the merging algorithm follows.

**Algorithm \((\ell, m)\)-merge**

1) Let \( U_i = u_{1i}, u_{2i}, \ldots, u_{ri} \), for \( 1 \leq i \leq \ell \), be the sequences to be merged. When \( r \) is small use a base case algorithm. Otherwise, unshuffle each \( U_i \) into \( m \) parts. I.e., partition \( U_i \) into \( U_{i1}, U_{i2}, \ldots, U_{im} \), where \( U_{i1} = u_{1i}, u_{1+i}, \ldots, U_{i2} = u_{2i}, u_{2+i}, \ldots \); and so on.

2) Merge \( U_{i1}, U_{i2}, \ldots, U_{im} \), for \( 1 \leq j \leq m \), recursively. Let the merged sequences be \( X_j = x_{1j}, x_{2j}, \ldots, x_{rjm} \), for \( 1 \leq j \leq m \).

3) Shuffle \( X_1, X_2, \ldots, X_m \). In particular, form the sequence \( Z = x_{11}, x_{21}, \ldots, x_{1m}, x_{22}, x_{22}, \ldots, x_{2m}, \ldots, x_{1rjm}, x_{2rjm}, \ldots, x_{mrjm} \).

4) At this point the length of the ‘dirty sequence’ (i.e., unsorted portion) can be shown to be no more than \( lm \). We don’t know where the dirty sequence is located. There are many ways to clean up the dirty sequence. One such way is given below.

Let \( Z_1 \) denote the sequence of the first \( lm \) elements of \( Z \); Let \( Z_2 \) denote the next \( lm \) elements as \( Z_2 \); and so on. Thus \( Z \) is partitioned into
$Z_1, Z_2, \ldots, Z_{r/m}$. Sort each one of the $Z_i$’s. Then 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.

Figure 1.1 illustrates this algorithm.

Figure 1.1 The $(\ell, m)$-merge algorithm

The above algorithm is not specific to any architecture. (The same can be said about any algorithm). Rajasekaran [20] gives an implementation of LMM on PDS. The number of I/O operations used in this implemen-
tation is $\frac{N}{DB} \left[ \frac{\log(N/M)}{\log(\min(\sqrt{M},M/B))} + 1 \right]^2$. This number is a constant when $N$ is a polynomial in $M$ and $M$ is a polynomial in $B$. In this case LMM is optimal. It has been demonstrated that LMM can be faster than the DSM when $D$ is large [20]. Recent implementation results of Cormen and Pearson [7, 18] indicate that LMM is competitive in practice.

3.2 LMM ON PDS

In this section we give details of the implementation of LMM on the PDS model. The implementation merges $R$ runs at a time, for some appropriate $R$. We have to specify how the intermediate runs are stored across the $D$ disks. Number the disks as well as the runs from zero. Each run will be striped across the disks. If $R \geq D$, the starting disk for the $i$th run is $i \mod D$, i.e., the zeroth block of the $i$th run will be in disk $i \mod D$; its first block will be in disk $(i + 1) \mod D$; and so on. This strategy yields perfect disk parallelism since in one I/O read operation, one block each from $D$ distinct runs can be accessed. If $R < D$, the starting disk for the $i$th run is $iD_R$. (Assume without loss of generality that $D$ divides $R$.) Even now, we can obtain $D_R$ blocks from each of the runs in one I/O operation and hence achieve perfect disk parallelism. See Figure 1.2.

The value of $B$ will be much less than $M$ in practice. For instance, when $\frac{M}{B} > \sqrt{M}$ the number of read passes made by LMM is no more than $\left(2\frac{\log(N/M)}{\log M} + 1\right)^2$. However, the case $\frac{M}{B} \leq \sqrt{M}$ is also considered. The number of read passes made by LMM is upper bounded by $\left[\frac{\log(N/M)}{\log(\min(\sqrt{M},M/B))} + 1\right]^2$ in either case. LMM forms initial runs of length $\sqrt{M}$ each in one read pass through the data. After this, the runs will be merged $R$ at a time. Throughout, we use $T(u,v)$ to denote the number of read passes needed to merge $u$ sequences of length $v$ each.

Some special cases. Some special cases will be considered first. The first case is the problem of merging $\sqrt{M}$ runs each of length $M$, when $\frac{M}{B} \geq \sqrt{M}$. In this case use $R = \sqrt{M}$. This merging can be done using Algorithm $(l,m)$-merge with $l = m = \sqrt{M}$.

Let the sequences to be merged be $U_1, U_2, \ldots, U_{\sqrt{M}}$. In Step 1, each $U_i$ gets unshuffled into $\sqrt{M}$ parts. Each part is of length $\sqrt{M}$. This unshuffling takes one pass. In Step 2, $\sqrt{M}$ merges have to be done. Each merge involves $\sqrt{M}$ sequences of length $\sqrt{M}$ each. There are only $M$ records in each merge and hence all the merges can be done in one pass through the data. Step 3 involves shuffling and Step 4 involves...
cleaning up. The length of the dirty sequence is $(\sqrt{M})^2 = M$. We can combine these two steps to complete them in one pass through the data. This can be done as follows. Have two successive $Z_i$'s (c.f. Algorithm $(l,m)$-merge) (call these $Z_i$ and $Z_{i+1}$) at any time in the main memory. Sort $Z_i$ and $Z_{i+1}$ and merge them. Ship $Z_i$ to the disks. Bring in $Z_{i+2}$, sort it, and merge it with $Z_{i+1}$. Ship out $Z_{i+1}$; and so on.

Observe that perfect disk parallelism is maintained throughout. Thus we get:

**Lemma 3..1** $T(\sqrt{M}, M) = 3$, if $\frac{M}{B} \geq \sqrt{M}$.

The second special case considered is that of merging $\frac{M}{B}$ runs each of length $M$, when $\frac{M}{B} < \sqrt{M}$. Employ Algorithm $(l,m)$-merge with $l = m = \frac{M}{B}$. Along similar lines, we can prove the following Lemma.
Lemma 3.2 \( T\left( \frac{M}{B}, M \right) = 3, \text{ if } \frac{M}{B} < \sqrt{M}. \)

The general algorithm. The general algorithm utilizes the above special cases. The general algorithm is also presented in two cases, one for \( \frac{M}{B} \geq \sqrt{M} \) and the other for \( \frac{M}{B} < \sqrt{M}. \) As usual, initial runs are formed in one pass. After this pass, \( N/M \) sorted sequences of length \( M \) each remain to be merged.

If \( \frac{M}{B} \geq \sqrt{M} \), employ Algorithm \((l,m)\)-merge with \( l = m = \sqrt{M} \) and \( R = \sqrt{M}. \) Let \( K \) denote \( \sqrt{M} \) and let \( N_M = K^{2c}. \) As a result, \( c = \frac{\log(N/M)}{\log M}. \) The following relation is easy to see.

\[
T(K^{2c}, M) = T(K, M) + T(K, KM) + \cdots + T(K, K^{2c-1}M). \tag{1.2}
\]

This relation means that we start with \( K^{2c} \) sequences of length \( M \) each; we merge \( K \) at a time to end up with \( K^{2c-1} \) sequences of length \( KM \) each; again merge \( K \) at a time to end up with \( K^{2c-2} \) sequences of length \( K^2M \) each; and so on. Finally there will remain \( K \) sequences of length \( K^{2c-1}M \) each which are merged. Each phase of merges is done using the Algorithm \((l,m)\)-merge with \( l = m = \sqrt{M}. \)

\( T(K, K^i M) \) for any \( i \) can be computed as follows. There are \( K \) sequences to be merged each of length \( K^i M \). Let these sequences be \( U_1, U_2, \ldots, U_K. \) In Step 1, each \( U_j \) is unshuffled into \( K \) parts each of size \( K^{i-1}M. \) This can be done in one pass. Now there are \( K \) merging problems, where each merging problem involves \( K \) sequences of length \( K^{i-1}M \) each. The number of passes needed is \( T(K, K^{i-1}M). \) The length of the dirty sequence in Steps 3 and 4 is \( \leq K^2 = M. \) Thus Steps 3 and 4 can be completed in one pass. Therefore,

\[
T(K, K^i M) = T(K, K^{i-1} M) + 2.
\]

Expanding this out we see,

\[
T(K, K^i M) = 2i + T(K, M) = 2i + 3.
\]

We have made use of the fact that \( T(K, M) = 3 \) (c.f. Lemma 3.1).

Upon substitution of this into Equation 1.2, we get

\[
T(K^{2c}, M) = \sum_{i=0}^{2c-1} (2i + 3) = 4c^2 + 4c
\]

where \( c = \frac{\log(N/M)}{\log M}. \) If \( N \leq M^3, \) the above merging cost is \( \leq 24 \) passes.

We have the following
Theorem 1. The number of read passes needed to sort \( N \) records is 
\[
1 + 4 \left( \frac{\log(N/M)}{\log M} \right)^2 + 4 \frac{\log(N/M)}{\log M} \cdot \text{if } \frac{M}{B} \geq \sqrt{M}.
\]
This number of passes is no more than 
\[
\left[ \frac{\log(N/M)}{\log(\min\{\sqrt{M},M/B\})} + 1 \right]^2.
\]

Now consider the case \( \frac{M}{B} < \sqrt{M} \). Algorithm \((l,m)\)-merge can be used with \( l = m = \frac{M}{B} \) and \( R = \frac{M}{B} \). The following Theorem is proven in a similar fashion.

Theorem 2. The number of read passes needed to sort \( N \) records is upper bounded by 
\[
\left[ \frac{\log(N/M)}{\log(\min\{\sqrt{M},M/B\})} + 1 \right]^2 \cdot \text{if } \frac{M}{B} < \sqrt{M}.
\]

Theorems 1 and 2 yield

Theorem 3. We can sort \( N \) records in 
\[
\leq \left[ \frac{\log(N/M)}{\log(\min\{\sqrt{M},M/B\})} + 1 \right]^2 \text{read passes over the data, maintaining perfect disk parallelism.}
\]
In other words, the total number of I/O read operations needed is 
\[
\leq \frac{N}{DB} \left[ \frac{\log(N/M)}{\log(\min\{\sqrt{M},M/B\})} + 1 \right]^2.
\]

Observation. In Algorithm \((l,m)\)-merge, both \( l \) and \( m \) have to be \( \leq \frac{M}{B} \) in order to achieve perfect disk parallelism.

4. SELECTION ALGORITHMS FOR THE PDS

Given a sequence of \( n \) keys and an integer \( i, 1 \leq i \leq n \), the problem of selection is to identify the \( i \)th smallest of the \( n \) keys. This important comparison problem has been extensively studied. Numerous asymptotically optimal sequential algorithms have been discovered. Asymptotically optimal algorithms have been presented for varying parallel models as well.

Recently, Rajasekaran [21] has given two asymptotically optimal algorithms for selection for the PDS model. The first algorithm is deterministic and the second is randomized. The deterministic algorithm has been implemented and has been shown to perform better in practice than the algorithm of Manku et. al. [13].

In this section a summary of these algorithms is presented.
4.1 A RANDOMIZED ALGORITHM

In this section we describe the randomized selection algorithm of [21]. The number of I/O read operations made by the algorithm is \( O\left(\frac{N}{DB}\right) \) with high probability.

Numerous selection algorithms proposed in the literature, be they deterministic or randomized, sequential or parallel, are based on sampling. For instance, Floyd and Rivest’s randomized algorithm [8] is based on random sampling. The algorithm has the following steps. 1) Select a random sample \( S \) of \( s \) elements from the input set \( X \); 2) Sort the sample \( S \) and find two elements \( \ell_1 \) and \( \ell_2 \) from \( S \) such that the \( i \)th smallest element of \( X \) will have a value in between \( \ell_1 \) and \( \ell_2 \) and also the number of keys from \( X \) that have a value in between \( \ell_1 \) and \( \ell_2 \) is ‘small’; 3) Eliminate keys of \( X \) that do not have a value in the range \([\ell_1, \ell_2]\); and 4) Perform an appropriate selection in the set of remaining keys.

Sampling techniques have also been used to develop selection algorithms for numerous parallel models of computing. Though sampling based algorithms have sampling as the common theme, they have model-dependent innovations and might employ additional techniques.

A Sampling Lemma. Let \( Y \) be a sequence of \( n \) numbers from a linear order and let \( S = \{k_1, k_2, \ldots, k_s\} \) be a random sample from \( Y \). Also let \( k'_1, k'_2, \ldots, k'_s \) be the sorted order of this sample. If \( r_i \) is the rank of \( k'_i \) in \( Y \), the following lemma provides a high probability confidence interval for \( r_i \). (The rank of any element \( k \) in \( Y \) is one plus the number of elements \( < k \) in \( Y \).)

Lemma 4.1 For every \( \alpha > 0 \), \( \text{Prob.}\ (|r_i - \frac{i}{s}| > \sqrt{\frac{3\alpha n}{s}}\sqrt{\log n}) < n^{-\alpha}. \)

For a proof of this Lemma see [22].

We say a randomized algorithm uses \( \tilde{O}(g(n)) \) amount of any resource (like time, space, etc.) if there exists a constant \( c \) such that the amount of resource used is no more than \( c\alpha g(n) \) with probability \( \geq 1 - n^{-\alpha} \) on any input of length \( n \) and for any \( \alpha \) (see e.g., [9]). Similar definitions apply to \( \tilde{o}(g(n)) \) and other such ‘asymptotic’ functions. By high probability we mean a probability \( \geq 1 - n^{-\alpha} \) for any fixed \( \alpha \geq 1 \) (\( n \) being the input size of the problem at hand). Let \( B(n, p) \) denote a Binomial distribution with parameters \( n \) and \( p \).

Let the input size for the selection problem be \( N \) and let \( N = M^c \). In practice \( c \) can be assumed to be a constant. In today’s PC market, \( M \) is of the order of megabytes and the disk space is of the order of gigabytes.
So it is perhaps safe to assume that $c$ is no more than $3$. To begin with each key is *alive* and $n = N$.

**Algorithm RSelect**

*repeat*

**Step 1.** Let $n$ denote the number of alive keys. If $n \leq M$ then goto Step 6. Each alive key gets included in the sample $S$ with probability $\frac{M}{n}$. Thus the expected number of sample keys is $M$. The actual number of keys in $S$ can be shown to be $M + \tilde{o}(M)$. Count the number $s$ of sample keys.

**Step 2.** Sort $S$ and pick two keys $\ell_1$ and $\ell_2$ from $S$ whose ranks in $S$ are $\ell_1^S - \delta$ and $\ell_2^S + \delta$, respectively, for $\delta \geq \sqrt{3c \alpha \log n}$, for any fixed $\alpha \geq 1$.

**Step 3.** Count the number $n_1$ of alive keys that are less than $\ell_1$ and the number $n_2$ of alive keys that have a value in the range $[\ell_1, \ell_2]$.

**Step 4.** If $i < n_1$, or $i > n_1 + n_2$, or $n_2 > \frac{n}{M^{0.4}}$, goto Step 1.

**Step 5.** Any alive key whose value lies outside the range $[\ell_1, \ell_2]$ dies. Set $i = i - n_1$ and $n = n_2$.

*forever*

**Step 6.** Sort the alive keys and output the $i$th smallest key.

**Analysis.** The number of sample keys in Step 1 has a binomial distribution $B(n, M/n)$. Applying Chernoff bounds we see that $s = M + \tilde{o}(M)$. Step 1 takes $O\left(\frac{n \log n}{MB}\right)$ read operations.

Step 2 does not involve any I/O operations since $S$ is kept in the internal memory.

In Step 3, an application of Lemma 4.1 implies that the number $n_2$ of keys surviving Step 5 is $\tilde{O}\left(\frac{n}{\sqrt{s} \sqrt{\log n}}\right) = \tilde{O}\left(\frac{n}{\sqrt{M}}\right)$. It follows that, the number of iterations of the *repeat* loop is $\tilde{O}(c)$.

In Step 4, we can show that the probability of executing the goto statement is very small.

Step 5 takes $O\left(\frac{n}{MB}\right)$ I/O read operations.

The number of surviving keys from one iteration of the *repeat* loop to the next decreases by a factor of $M^{0.4}$ with high probability. As a result, the total number of I/O read operations made by the entire algorithm is $\tilde{O}\left(\frac{N}{MB}\right)$.

Thus the following Theorem follows.

**Theorem 4** Selection from out of $N$ keys can be performed on the PDS using $O\left(\frac{N}{MB}\right)$ I/O read operations.  □
4.2 A DETERMINISTIC ALGORITHM

In this section we summarize the deterministic selection algorithm of [21]. The number of I/O read operations performed by the algorithm is $O\left(\frac{N}{DB}\right)$. The constant in this time bound is small and hence this algorithm has the potential of being practical. Recent implementation results indicate that this algorithm is faster than the algorithm of Manku et al. [13].

In the development of deterministic selection algorithms also sampling has dominated. For instance, Blum et al.’s algorithm [5] partitions the input such that there are 5 elements in each part, finds the median of each part, finds the median $M$ of these medians, splits the input into two groups (those that are $\leq M$ and those that are greater than $M$), identifies the group that has the key to be selected, and finally performs a selection in the group that contains the key to be selected. We can think of the medians of the 5-element parts as a sample of the input keys and hence the median $M$ of these medians can be expected to be an approximate median of the input keys.

Let $X$ be a collection of $n$ keys whose $i$th smallest key we are interested in finding. We identify two elements of $X$ such that they will bracket the $i$th smallest element and also the number of keys of $X$ that have a value in between these two keys is not large.

Partition $X = R_0$ into $M$-element parts. Sort each part. In each part retain those keys that are at a distance of $\sqrt{M}$ from each other. I.e., keep the keys whose ranks are $\sqrt{M}, 2\sqrt{M}, 3\sqrt{M}, \ldots$. The number of keys in the retained set $R_1$ is $\frac{n}{\sqrt{M}}$. Now group the elements of $R_1$ such that there are $M$ elements in each part, sort each part, and retain only every $\sqrt{M}$th element in each part. Call the retained set $R_2$. Proceed to obtain $R_i$’s in a similar fashion (for $i \geq 3$) until we reach a stage when $|R_j| \leq M$. If $n = M^c$, then clearly, $j = 2c - 2$.

We can use a tree of degree $\sqrt{M}$ to represent this process. There are $M$ input elements in each leaf. $R_0$ comprises all the elements in the leaves. The root has $R_0$. Let the root be in level $j$. Let its children be in level $j - 1$, and so on. The leaves are at level 0. The root has $\sqrt{M}$ children. Each such child has $M$ elements. $\sqrt{M}$ elements are passed on from each child to its parent. In general each node in the tree has $M$ elements. $\sqrt{M}$ elements from out of these will go to its parent. Each node has $\sqrt{M}$ children.

Pick from $R_j$ two elements $\ell_1$ and $\ell_2$ whose ranks are $i \frac{|R_j|}{n} - \delta$ and $i \frac{|R_j|}{n} + \delta$, respectively.
Assume without loss of generality that $|R_j| = M$. In this case, $|R_i| = M(\sqrt{M})^{j-i}$. Consider an element $x$ whose rank in $R_j$ is $q$. The rank of $q$ in $R_{j-1}$ will be in the range $[q\sqrt{M}, q\sqrt{M} + \sqrt{M}(\sqrt{M} - 1)]$. This rank is also in the interval $[q\sqrt{M}, q\sqrt{M} + M]$. In other words, there is an uncertainty of $M$ in the rank of $x$ in $R_{j-1}$. Each child of the root contributes $\sqrt{M} - 1 \approx \sqrt{M}$ to this uncertainty. In general the uncertainty in the rank of $x$ in $R_i$ is contributed to by each node in level $i$. There are $M^{(j-i)/2}$ nodes at level $i$. Every such node contributes $\sqrt{M} - 1$ to the uncertainty. As a result, if $U(i)$ is the maximum possible rank of $x$ in $R_i$, then $U(i)$ satisfies:

$$U(i) \leq \sqrt{M} U(i + 1) + M^{(j-i+1)/2}$$

which solves to $U(i) \leq M^{(j-i)/2} U(j) + (j - i) M^{(j-i+1)/2}$. When $i = 0$, we get $U(0) \leq M^{j/2} U(j) + j M^{(j+1)/2}$. In other words, $U(0) \leq qM^{c-1} + (2c - 2) \frac{n}{\sqrt{M}}$. Note that $c = \frac{\log n}{\log M}$.

Thus if we pick $\delta$ to be $(2c - 2 + \epsilon)\sqrt{M}$, for any $\epsilon > 0$, the rank of $\ell_1$ in $R_0$ will be in the interval

$$\left[ i - (2c - 2 + \epsilon) \frac{n}{\sqrt{M}}, i - \epsilon \frac{n}{\sqrt{M}} \right].$$

The rank of $\ell_2$ in $R_0$ will be in the interval

$$\left[ i + (2c - 2 + \epsilon) \frac{n}{\sqrt{M}}, i + (4c - 4 + \epsilon) \frac{n}{\sqrt{M}} \right].$$

In summary, we realize that the $i$th smallest element of $R_0$ will have a value in the interval $[\ell_1, \ell_2]$ and also the number of keys of $R_0$ that have a value in the interval $[\ell_1, \ell_2]$ is no more than $(6c - 6 + 2\epsilon) \frac{n}{\sqrt{M}}$.

Let the above process of starting from $R_0$ and obtaining $R_1, R_2, \ldots, R_j$ be called a stage of sampling. The number of I/O read operations needed for a stage is $O \left( \frac{n}{M^{j/2}} \right)$ where $|R_0| = n$. A stage of sampling corresponds to one iteration of the repeat loop of \text{RSelect}. Similar sampling techniques have been employed by Munro and Paterson [14].

Let $K = k_1, k_2, \ldots, k_N$ be the input. Say we are interested in finding the $i$th smallest key. To begin with each key is alive and $n = N$. A detailed description of the selection algorithm follows.

\textbf{Algorithm DSelect}

\textbf{repeat}

\textbf{Step 1.} If $n \leq M$ goto Step 3. Perform one stage of sampling in the collection of alive keys. Obtain two keys $\ell_1$ and $\ell_2$ that will bracket the key to be selected.
Step 2. Kill the alive keys that have a value outside the range $[\ell_1, \ell_2]$. Count the number $n$ of keys surviving this step.

forever

Step 3. Sort the alive keys and output the $i$th smallest element.

Analysis. The number of alive keys reduces by a factor of $\Omega\left(\frac{\sqrt{M \log M}}{\log n}\right)$ from one iteration to the next of the repeat loop. As a result, if $N = M^c$, the number of iterations of the repeat loop is $O(c)$.

The number of I/O read operations needed in any iteration of the repeat loop is $O\left(\frac{n}{DB}\right)$ as has been established before. Since the number of alive keys decreases by a factor of $\Omega\left(\frac{\sqrt{M \log M}}{\log n}\right)$ from one iteration to the next, the total number of I/O read operations is only $O\left(\frac{N}{DB}\right)$. We arrive at the following Theorem.

Theorem 5 We can perform selection from $N$ given keys on the PDS using $O\left(\frac{N}{DB}\right)$ I/O read operations. $\square$

5. A PRACTICAL REALIZATION OF PARALLEL DISKS

Though the existing models with multiple disks address the problem of data explosion, it’s not clear how these models can be realized in practice. The assumption of bringing $D$ blocks of data in one I/O operation may not be practical. A new model called a Parallel Machine with Disks (PMD) is proposed in [23] that is a step in the direction of practical realization. A PMD is nothing but a parallel machine where each processor has an associated disk. The parallel machine can be structured or unstructured. The underlying topology of a structured parallel machine could be a mesh, a hypercube, a star graph, etc. Examples of unstructured parallel computers include SMP, NOW, a cluster of workstations (employing PVM or MPI), etc. The PMD is nothing but a parallel machine where we study out-of-core algorithms. In the PMD model we not only count the I/O operations but also the communication steps. One can think of a PMD as a realization of the PDS model.

Every processor in a PMD has an internal memory of size $M$. In one parallel I/O operation, a block of $B$ records can be brought into the core memory of each processor from its own disk. There are a total of $D = P$ disks in the PMD, where $P$ is the number of processors. Records from one disk can be sent to another via the communication mechanism available for the parallel machine after bringing the records into the main memory of the origin processor. The communication time
could potentially be comparable to the time for I/O on the PMD. It is essential therefore to not only account for the I/O operations but also for the communication steps, in analyzing any algorithm’s run time on the PMD.

We can state the sorting problem on the PMD as follows. There are a total of $N$ records to begin with. There are $\frac{N}{D}$ records in each disk. The problem is to rearrange the records so that they are in either ascending order or descending order with $\frac{N}{D}$ records ending up in each disk. We assume that the processors themselves are ordered so that the smallest $\frac{N}{D}$ records will be output in the first processor’s disk, the next smallest $\frac{N}{D}$ records will be output in the second processor’s disk, and so on.

We can apply LMM on a general PMD in which case the number of I/O operations will remain the same, i.e., $\frac{N}{DB} \left\lceil \frac{\log(N/M)}{\log(\min{\sqrt{MD}, MD/B})} + 1 \right\rceil^2$. In particular, the number of passes through the data will be $\left\lceil \frac{\log(N/M)}{\log(\min{\sqrt{MD}, MD/B})} + 1 \right\rceil^2$. Such an application needs mechanisms for $(k - k)$ routing and $(k - k)$ sorting.

The problem of $(k - k)$ routing in the context of a parallel machine is the problem of packet routing where there at most $k$ packets of information originating from each processor and at most $k$ packets are destined for each processor. We are required to send all the packets to their correct destinations as quickly as possible. The problem of $(k - k)$ sorting is one where there are $k$ keys to begin with at each processor and we are required to sort all the keys and send the smallest $k$ keys to processor 1, the next smallest $k$ keys to processor 2, and so on.

Let $R_M$ and $S_M$ denote the time needed for performing one $(M - M)$ routing and one $(M - M)$ sorting on the parallel machine, respectively. Then, in each pass through the data, the total communication time will be $\frac{N}{DM}(R_M + S_M)$, implying that the total communication time for the entire algorithm will be $\leq \frac{N}{DM}(R_M + S_M) \left\lceil \frac{\log(N/MD)}{\log(\min{\sqrt{MD}, MD/B})} + 1 \right\rceil^2$.

The following Theorem is proven in [23].

**Theorem 6** Sorting on a PMD can be performed in $\frac{N}{DB} \left\lceil \frac{\log(N/MD)}{\log(\min{\sqrt{MD}, MD/B})} + 1 \right\rceil^2$ I/O operations. The total communication time is $\leq \frac{N}{DM}(R_M + S_M) \left\lceil \frac{\log(N/MD)}{\log(\min{\sqrt{MD}, MD/B})} + 1 \right\rceil^2$. $\blacksquare$
6. CONCLUSIONS

In this article we have provided a survey of sorting and selection algorithms that can be found in the literature. In particular, we have summarized the DSM, SRM, and LMM algorithms for sorting. DSM is a simple algorithm that does well in practice but is not asymptotically optimal. SRM is a randomized algorithm. LMM algorithm is asymptotically optimal and is promising to be practical. In addition we have discussed asymptotically optimal (deterministic and randomized) algorithms for selection on the PDS model. We have also studied the problem of realizing parallel disk models in practice.

Notes
1. Throughout this article we use log to denote logarithms to the base 2 and ln to denote natural logarithms.

References


