

A Practical Realization of Parallel Disks

Sanguthevar Rajasekaran

Dept. of CISE, University of Florida
Gainesville FL 32611
raj@cise.ufl.edu

Xiaoming Jin

Dept. of CISE, University of Florida
Gainesville FL 32611
xjin@cise.ufl.edu

Abstract

Several models of parallel disks are found in the literature. These models have been proposed to alleviate the I/O bottleneck arising in handling voluminous data. These models have the general theme of assuming multiple disks. For instance 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 paper we present a more practical model for multiple disks and evaluate it experimentally. This model is called a Parallel Machine with Disks (PMD). A PMD can be thought of as a realization of the PDS model. A PMD can also be considered as a special case of the hierarchical memory models proposed in the literature. We investigate the sorting problem on the new model. Our analysis demonstrates the practicality of the PMD. We also present experimental confirmation of this assertion with data from our implementations.

1 Introduction

Computing applications have advanced to a stage where voluminous data is the norm. The volume of data dictates the use of secondary storage devices such as disks. Even the use of just a single disk may not be sufficient to handle I/O operations efficiently. Thus researchers have introduced models with multiple disks.

A model that has been studied extensively (which is a refinement of prior models) is the Parallel Disk Systems (PDS) model [19]. In this model there is a single computer and D disks. In one parallel I/O, a block of data from each of the D disks can be brought into the main memory. A block consists of B records. If M is the internal memory size, then one usually requires that $M \geq 2DB$. Algorithm designers have proposed algorithms for numerous fundamental problems on the PDS model. In the analysis of these algorithms they counted only the I/O operations since

the local computations can be assumed to be very fast.

The practical realization of this model is an important research issue. Models such as Hierarchical Memory Models (HMMs) [10, 11] have been proposed in the literature to address this issue. Realizations of HMMs using PRAMs and hypercubes have been explored [11]. Sorting algorithms on these realizations have been investigated.

In this paper we propose a straight forward model called a Parallel Machine with Disks (PMD). A PMD can be thought of as a special case of the HMM. A PMD is nothing but a parallel machine where each processor has an associated disk. The parallel machine can be structured or unstructured. If the parallel machine is structured, the underlying topology could be a mesh, a hypercube, a star graph, etc. Examples of unstructured parallel computers include SMP, a cluster of workstations (employing PVM or MPI), etc. In some sense, 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. Given the connection between HMMs and PDSs, we can state that prior works have considered variants of the PMD where the underlying parallel machine is either a PRAM or a hypercube [11].

We begin the study of PMDs with the sorting problem. Sorting is an important problem of computing that finds applications in all walks of life. We analyze the performance of the LMM sort algorithm of [16] on PMD (where the underlying parallel machine is a mesh, a hypercube, and a cluster of workstations). In particular, we compute the theoretical run times for sorting. These analyses demonstrate the practicality of the PMD. To confirm our assertion we have also experimented with the PMD where the underlying parallel machine is a cluster of workstations (using PVM for interprocessor communications). In this paper we also present preliminary data from our experiments.

In Section 2 we provide a summary of known algorithms for sorting on the PDS model. In Section 3 we present details of the PMD model. To make our discussions concrete we use the mesh as the topology of the underlying paral-

lel machine. However, the discussions apply to any parallel machine (with appropriate changes in some parameters). In Section 3 we also state some known theorems for routing and sorting on the mesh. Section 4 carries a brief description of the LMM sort algorithm of [16]. In Section 5 we show how LMM can be applied on the PMD model and analyze the resulting run time. In Section 6 we present our experimental results. Section 7 concludes the paper.

2 Sorting Results on the PDS Model

Sorting has been studied well on the PDS model. A known lower bound for the number of I/O read steps for parallel disk sorting is¹ $\Omega\left(\frac{N}{DB} \left\lceil \frac{\log(N/B)}{\log(M/B)} \right\rceil\right)$. Here N is the number of records to be sorted and M is the internal memory size. Also, B is the block size and D is the number of parallel disks used. There exist several asymptotically optimal algorithms that make $O\left(\frac{N}{DB} \left\lceil \frac{\log(N/B)}{\log(M/B)} \right\rceil\right)$ I/O read steps (see e.g., [12, 1, 3]).

One of the early papers on disk sorting was by Aggarwal and Vitter [2]. In the model they considered, each I/O operation results in the transfer of D blocks each block having B records. A more realistic model was envisioned in [19]. Several asymptotically optimal algorithms have been given for sorting on this model. Nodine and Vitter's optimal algorithm [10] involves solving certain matching problems. Aggarwal and Plaxton's optimal algorithm [1] is based on the Sharesort algorithm of Cypher and Plaxton. Vitter and Shriver gave an optimal randomized algorithm for disk sorting [19]. All of these results are highly nontrivial and theoretically interesting. However, the underlying constants in their time bounds are high.

In practice the simple disk-striped mergesort (DSM) is used [4], even though it is not asymptotically optimal. DSM has the advantages of simplicity and a small constant. Data accesses made by DSM is such that in any I/O operation, the same portions of the D disks are accessed. This has the effect of having a single disk which can transfer DB records in a single I/O operation. An $\frac{M}{DB}$ -way mergesort is employed by this algorithm. To start with, initial runs are formed in one pass through the data. At the end the disk has N/M runs each of length M . Next, $\frac{M}{DB}$ 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 involves one pass through the data. There are $\frac{\log(N/M)}{\log(M/DB)}$ phases and hence the total number of passes made by DSM is $\frac{\log(N/M)}{\log(M/DB)}$. In other words, the total number of I/O read operations performed by the al-

gorithm is $\frac{N}{DB} \left(1 + \frac{\log(N/M)}{\log(M/DB)}\right)$. The constant here is just 1.

If one assumes that N is a polynomial in M and that B is small (which are readily satisfied in practice), the lower bound simply yields $\Omega(1)$ passes. All the abovementioned optimal algorithms make only $O(1)$ passes. So, the challenge in the design of parallel disk sorting algorithms is in reducing this constant. If $M = 2DB$, the number of passes made by DSM is $1 + \log(N/M)$, which indeed can be very high.

Recently, several works have been done that deal with the practical aspects. Pai, Schaffer, and Varman [13] analyzed the average case performance of a simple merging algorithm, employing an approximate model of average case inputs. Barve, Grove, and Vitter [4] have presented a simple randomized algorithm (SRM) and analyzed its performance. The analysis involves the solution of certain occupancy problems. The expected number $Read_{SRM}$ of I/O read operations made by their algorithm is such that

$$Read_{SRM} \leq \frac{N}{DB} + \frac{N}{DB} \frac{\log(N/M)}{\log kD} \frac{\log D}{k \log \log D} \left(1 + \frac{\log \log \log D}{\log \log D} + \frac{1 + \log k}{\log \log D} + o(1)\right) \quad (1)$$

The algorithm merges $R = kD$ runs at a time, for some integer k . When $R = \Omega(D \log D)$, the expected performance of their algorithm is optimal. 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 [16] has presented an algorithm (called (l, m) -merge sort (LMM)) which is asymptotically optimal under the assumptions that 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.

Other problems such as FFT computations (see e.g., [5]), selection (see e.g., [17]), etc. have also been studied on the PDS model.

3 A Parallel Machine with Disks (PMD)

In this section we give more details of the PMD model. A PMD is nothing but a parallel machine where each processor has a disk. Each processor has a core memory of size M . In one I/O operation, a block of B records can be brought into the core memory of each processor from its own disk. Thus 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. It is conceivable that the communication time is considerable on the PMD. Thus it is essential 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.

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

PMD can be thought of as a special case of the HMM [11]. Realization of HMM using PRAMs and hypercubes have already been studied [11].

The sorting problem on the PMD 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. It is assumed that the processors themselves have been 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. This indexing scheme is in line with the usual indexing scheme used in a parallel machine. However any other indexing scheme can also be used.

To make our discussions concrete, we will use the mesh as an example. Let the mesh be of size $n \times n$. Then we have $D = n^2$ disks. An indexing scheme is called for in sorting on a mesh (see e.g., [15]). Some popular indexing schemes are column major, row major, snake-like row, blockwise row-major, etc. For the algorithm to be presented in this paper, any of these schemes can be employed.

The algorithm to be presented in this paper employs as subroutines some randomized algorithms. We say a randomized algorithm uses $\tilde{O}(f(n))$ amount of any resource (such as time, space, etc.) if the amount of resource used is no more than $caf(n)$ with probability $\geq (1 - n^{-\alpha})$, where c is a constant and α is a constant ≥ 1 . We can also define other asymptotic functions such as $\tilde{\Omega}(\cdot)$, $\tilde{o}(\cdot)$, etc. in a similar fashion.

$k - k$ routing and $k - k$ sorting. The problem of *packet routing* plays a vital role in the design of algorithms on any parallel machine. The packet routing problem can be defined as follows. There is a packet of information to start with at each processor that is destined for some other processor. The problem is to send all the packets to their correct destinations as quickly as possible. In any interconnection network, one requires that at most one packet traverses through any edge at any time. The problem of *partial permutation routing* refers to packet routing when at most one packet originates from any processor and at most one packet is destined for any processor. Packet routing problems have been explored thoroughly on interconnection networks (see e.g., [15]).

The problem of $k - k$ routing is the problem of routing where at most k packets originate from any processor and at most k packets are destined for any processor.

In the case of an $n \times n$ mesh, it is easy to prove a lower bound of $\frac{kn}{2}$ on the routing time for this problem based on bisection considerations. There are algorithms whose run times match this bound closely as stated in the following Lemma. A proof of this Lemma can be found e.g., in [7].

Lemma 3.1 *The $k - k$ routing problem can be solved in $\frac{kn}{2} + \tilde{o}(kn)$ time on an $n \times n$ mesh.*

The problem of $k - k$ sorting is defined as follows. There are k keys at each processor of a parallel machine. The problem is to rearrange the keys in either ascending or descending order according to some indexing scheme.

In an $n \times n$ mesh, this problem also has a lower bound of $\frac{kn}{2}$ on the run time. The following Lemma promises a closely optimal algorithm (see e.g., [15]).

Lemma 3.2 *$k - k$ sorting can be solved on an $n \times n$ mesh in $\frac{kn}{2} + \tilde{o}(kn)$ steps.*

The above two Lemmas will be employed by our sorting algorithm on the mesh. It should be noted here that there exist deterministic algorithms (see e.g., [8]) for $k - k$ routing and $k - k$ sorting whose run times match those stated in Lemmas 3.1 and 3.2. However, we believe that the use of randomized algorithms will result in better performance in practice.

4 The (ℓ, m) -Merge Sort (LMM)

Many of the sorting algorithms that have been proposed for the PDS are based on merging. These algorithms start by forming $\frac{N}{M}$ runs each of length M . A run is nothing but a sorted subsequence. Forming these initial runs takes only one pass through the data (or equivalently $\frac{N}{DB}$ parallel I/O operations). After this, the algorithms will merge R runs at a time. Let a *phase of mergings* refer to the task of scanning through the input once and performing R -way mergings. Note that each phase of mergings will reduce the number of remaining runs by a factor of R . For example, the DSM algorithm employs $R = \frac{M}{DB}$. The various sorting algorithms differ in how each phase of mergings is done.

The (ℓ, m) -merge sort algorithm of [16] is also based on merging. It employs $R = \ell$, for some appropriate ℓ . The LMM is a generalization of the odd-even merge sort, the s^2 -way merge sort of Thompson and Kung [18], and the columnsort algorithm of Leighton [9].

The odd-even mergesort algorithm employs $R = 2$. It repeatedly merges two sequences at a time. To begin with there are n sorted runs each of length 1. From thereon the number of runs is decreased by a factor of 2 with each phase of mergings. Two runs are merged using the odd-even merge algorithm that is described below.

1) Let $U = u_1, u_2, \dots, u_q$ and $V = v_1, v_2, \dots, v_q$ be the two sorted sequences to be merged. *Unshuffle* U into two, i.e., partition U into two: $U_{odd} = u_1, u_3, \dots, u_{q-1}$ and $U_{even} = u_2, u_4, \dots, u_q$. Similarly partition V into V_{odd} and V_{even} .

2) Now recursively merge U_{odd} with V_{odd} . Let $X = x_1, x_2, \dots, x_q$ be the result. Also merge U_{even} with V_{even} . Let $Y = y_1, y_2, \dots, y_q$ be the result.

3) Shuffle X and Y , i.e., form the sequence: $Z = x_1, y_1, x_2, y_2, \dots, x_q, y_q$.

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

The correctness of this algorithm can be established using the zero-one principle. The algorithm of Thompson and Kung [18] is a generalization of the above algorithm where R is taken to be s^2 for some appropriate function s of n . At any given time s^2 runs are merged using an algorithm similar to the above.

LMM is a generalization of s^2 -way merge sort algorithm. It uses $R = \ell$. Each phase of mergings thus reduces the number of runs by a factor of ℓ . At any time, ℓ runs are merged using the (ℓ, m) -merge algorithm. This merging algorithm is similar to the odd-even merge except that in Step 1, 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 detailed description of the merging algorithm follows.

Algorithm (ℓ, m) -merge

1) Let the sequences to be merged be $U_i = u_i^1, u_i^2, \dots, u_i^r$, for $1 \leq i \leq \ell$. If r is small use a base case algorithm. Otherwise, unshuffle each U_i into m parts. In particular, partition U_i into $U_i^1, U_i^2, \dots, U_i^m$, where $U_i^1 = u_i^1, u_i^{1+m}, \dots$; $U_i^2 = u_i^2, u_i^{2+m}, \dots$; and so on.

2) Recursively merge $U_1^j, U_2^j, \dots, U_\ell^j$, for $1 \leq j \leq m$. Let the merged sequences be $X_j = x_j^1, x_j^2, \dots, x_j^{lr/m}$, for $1 \leq j \leq m$.

3) Shuffle X_1, X_2, \dots, X_m , i.e., form the sequence $Z = x_1^1, x_2^1, \dots, x_m^1, x_1^2, x_2^2, \dots, x_m^2, \dots, x_1^{lr/m}, x_2^{lr/m}, \dots, x_m^{lr/m}$.

4) It can be shown that at this point the length of the 'dirty sequence' (i.e., unsorted portion) is no more than lm . But we don't know where the dirty sequence is located. We can cleanup the dirty sequence in many different ways. One way is described below.

Call the sequence of the first lm elements of Z as Z_1 ; the next lm elements as Z_2 ; and so on. In other words, Z is partitioned into $Z_1, Z_2, \dots, Z_{r/m}$. 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.

The above algorithm is not specific to any architecture. (The same can be said about any algorithm). An implementation of LMM on PDS has been given in [16]. The number of I/O operations needed in this implementation has been shown to be $\frac{N}{DB} \left[\frac{\log(N/M)}{\log(\min\{\sqrt{M}, M/B\})} + 1 \right]^2$. When N is a polynomial in M and M is a polynomial in B this reduces to a constant number of passes through the data and hence LMM is optimal. In [16] it has been demonstrated that LMM can be faster than the DSM when D is large. Recent implementation results of Cormen and Pearson [6, 14] indicate that LMM is competitive in practice. Thus a natural choice of sorting algorithm for PMD is LMM. In the next Section we implement LMM on a PMD and analyze the resultant I/O and communication steps.

5 Sorting on the PMD

We begin by considering the sorting problem on the mesh. Then we generalize the derived result to any parallel machine.

5.1 The Mesh

Consider a PMD where the underlying machine is an $n \times n$ mesh. The number of disks is $D = n^2$. Each node in the mesh is a processor with a core memory of size M . In one I/O operation, a processor can bring a block of B records into its main memory. Thus the PMD as a whole can bring in DB records in one I/O operation. I.e., we can relate a PMD with a PDS whose main memory capacity is DM and that has D disks.

Let the number of records to be sorted be N . To begin with, there are $\frac{N}{D}$ records at each disk of the PMD. The goal is to rearrange the records in either ascending order or descending order such that each disk gets $\frac{N}{D}$ records at the end. An indexing scheme has to be assumed. For the algorithm to be presented any of the following schemes will be acceptable: row-major, column-major, snake-like row-major, snake-like column-major, blockwise row-major, blockwise column-major, blockwise snake-like row-major, and blockwise snake-like column-major. We assume the blockwise snake-like row-major order for the following presentations. The block size is $\frac{N}{D}$. I.e., the first (in the snake-like row-major order) processor will store the smallest $\frac{N}{D}$ records, the second processor will store the next smallest $\frac{N}{D}$ records, and so on.

As one can easily see, the entire LMM algorithm consists of shuffling, unshuffling and local sorting steps. We use the $k-k$ routing and $k-k$ sorting algorithms (Lemmas 3.1 and 3.2) to perform these steps. Typically, we bring records from the disks until the local memories are filled. Processing on these records is done using $k-k$ routing and

$k - k$ sorting algorithms. The queue length of $k - k$ sorting and $k - k$ routing algorithms is $k + \tilde{o}(k)$. So we do not fill M completely. We only half-fill the local memories so as to run the randomized algorithms. Also in order to overlap I/O with local computations, only half of this memory can be used to store operational data. We refer to this portion of the core memory as M . I.e., M is one-fourth of the core memory size available for each processor.

To begin with we form $\frac{N}{DM}$ sorted runs each of length DM . The number of I/O operations performed is $\frac{N}{DB}$. Also, the number of communication steps is $\tilde{O}(\frac{N}{D}n)$. This is so because, we perform $\frac{N}{DM}$ number of $k - k$ sortings (with $k = M$) and each such sort takes $kn + \tilde{o}(kn)$ steps. In this paper we have chosen to express the communication steps using $\tilde{O}(\cdot)$ instead of calculating the underlying constants explicitly. There are two reasons: 1) Calculating this constant is easy – we prefer to keep the discussion simple, and 2) The permutations we achieve using $k - k$ sorting and $k - k$ routing algorithms are often regular and perhaps the full power of these algorithms are not called for. I.e., it may be possible to devise more efficient algorithms for these permutations implying an improvement in the underlying constants. We plan to investigate this in our future work.

Since LMM is based on merging in phases, we have to specify how the runs in a phase are stored across the D disks. Let the disks as well as the runs be numbered from zero. We use the same scheme as the one given in [16]. Each run will be striped across the disks. If $R \geq D$, the starting disk for the i th run is $i \bmod D$, i.e., the zeroth block of the i th run will be in disk $i \bmod D$; its first block will be in disk $(i + 1) \bmod D$; and so on. This will enable us to access, in one I/O read operation, one block each from D distinct runs and hence obtain perfect disk parallelism. If $R < D$, the starting disk for the i th run is $i \frac{D}{R}$. (Assume without loss of generality that D divides R .) Even now, we can obtain $\frac{D}{R}$ blocks from each of the runs in one I/O operation and hence achieve perfect disk parallelism.

5.1.1 Base Cases

LMM is a recursive algorithm whose base cases are handled efficiently. We now discuss two base cases.

Base Case 1. Consider the problem of merging \sqrt{DM} runs each of length DM , when $\frac{DM}{B} \geq \sqrt{DM}$. This merging is done using (ℓ, m) -merge with $\ell = m = \sqrt{DM}$.

Let $U_1, U_2, \dots, U_{\sqrt{DM}}$ be the sequences to be merged. In Step 1, each U_i gets unshuffled into \sqrt{DM} parts so that each part is of length \sqrt{DM} . This unshuffling can be done in one pass through the data. Thus the number of I/O operations is $\frac{N}{DB}$. The communication time is $\tilde{O}(\frac{N}{D}n)$.

Note. Throughout the algorithm, each pass through the data will involve $\frac{N}{DB}$ I/O operations and $\frac{N}{D}n$ communication steps. Also, we use $T(u, v)$ to denote the number of read passes needed to merge u sequences of length v each.

In Step 2, we have \sqrt{DM} merges to do, each merge involving \sqrt{DM} sequences of length \sqrt{DM} each. Since there are only DM records in each merge, all the mergings can be done in one pass through the data. Steps 3 and 4 perform shuffling and cleaning up, respectively. The length of the dirty sequence is $(\sqrt{DM})^2 = DM$. These two steps can be combined and finished in one pass through the data (see [16] for details). Thus we get:

Lemma 5.1 $T(\sqrt{DM}, DM) = 3$, if $\frac{DM}{B} \geq \sqrt{DM}$.

Base Case 2. This is the case of merging $\frac{DM}{B}$ runs each of length DM , when $\frac{DM}{B} < \sqrt{DM}$. This problem can be solved using (ℓ, m) -merge with $\ell = m = \frac{DM}{B}$.

In this case we can obtain:

Lemma 5.2 $T(\frac{DM}{B}, DM) = 3$, if $\frac{DM}{B} < \sqrt{DM}$.

5.1.2 The Sorting Algorithm

LMM algorithm has been presented in two cases. In our implementation the two cases will be when $\frac{DM}{B} \geq \sqrt{DM}$ and when $\frac{DM}{B} < \sqrt{DM}$. In either case, initial runs are formed in one pass at the end of which $\frac{N}{DM}$ sorted sequences of length DM each remain to be merged.

When $\frac{DM}{B} \geq \sqrt{DM}$, (ℓ, m) -merge is employed with $\ell = m = \sqrt{DM}$. Let K denote \sqrt{DM} and let $\frac{N}{DM} = K^{2c}$. In other words, $c = \frac{\log(N/DM)}{\log(DM)}$.

$T(\cdot)$ can be expressed as follows.

$$T(K^{2c}, DM) = T(K, DM) + T(K, KDM) + \dots + T(K, K^{2c-1}DM) \quad (2)$$

The above relation basically means that there are K^{2c} sequences of length DM each to begin with; we merge K at a time to end up with K^{2c-1} sequences of length KDM each; again merge K at a time to end up with K^{2c-2} sequences of length K^2DM each; and so on. Finally there will be K sequences of length $K^{2c-1}DM$ each which are merged. Each of these mergings is done using (ℓ, m) -merge with $\ell = m = \sqrt{DM}$.

It can also be shown that

$$T(K, K^iDM) = 2i + T(K, DM) = 2i + 3.$$

The fact that $T(K, DM) = 3$ (c.f. Lemma 5.1) has been used.

Upon substituting this into Equation 2, we get

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

where $c = \frac{\log(N/DM)}{\log(DM)}$.

We have the following

Theorem 5.1 *The number of read passes needed to sort N records is $1 + 4 \left(\frac{\log(N/DM)}{\log(DM)} \right)^2 + 4 \frac{\log(N/DM)}{\log(DM)}$, if $\frac{DM}{B} \geq \sqrt{DM}$. This number of passes is no more than $\left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})} + 1 \right]^2$. This means that the number of I/O read operations is no more than*

$$\frac{N}{DB} \left[1 + 4 \left(\frac{\log(N/DM)}{\log(DM)} \right)^2 + 4 \frac{\log(N/DM)}{\log(DM)} \right].$$

The number of communication steps is no more than

$$\tilde{O} \left(\frac{N}{D} n \left[1 + 4 \left(\frac{\log(N/DM)}{\log(DM)} \right)^2 + 4 \frac{\log(N/DM)}{\log(DM)} \right] \right) \bullet$$

The second case to be considered is when $\frac{DM}{B} < \sqrt{DM}$. Here (ℓ, m) -merge will be used with $\ell = m = \frac{DM}{B}$. Let Q denote $\frac{DM}{B}$ and let $\frac{N}{DM} = Q^d$. That is, $d = \frac{\log(N/DM)}{\log(DM/B)}$. Like in case 1 we can get

$$T(Q^d, DM) = T(Q, DM) + T(Q, QDM) + \dots + T(Q, Q^{d-1}DM) \quad (3)$$

Also, we can get,

$$T(Q, Q^i DM) = 2i + T(Q, DM) = 2i + 3.$$

Here the fact $T(Q, DM) = 3$ (c.f. Lemma 5.2) has been used.

Equation 3 now becomes

$$T(Q^d, DM) = \sum_{i=0}^{d-1} (2i + 3) = d^2 + 2d$$

where $d = \frac{\log(N/DM)}{\log(DM/B)}$.

Theorem 5.2 *The number of read passes needed to sort N records on the PMD is upper bounded by $\left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})} + 1 \right]^2$, if $\frac{DM}{B} < \sqrt{DM}$.*

Theorems 5.1 and 5.2 readily yield

Theorem 5.3 *We can sort N records in $\leq \left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})} + 1 \right]^2$ read passes over the data. The total number of I/O read operations needed is $\leq \frac{N}{DB} \left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})} + 1 \right]^2$. Also, the total number of communication steps needed is $\tilde{O} \left(\frac{N}{D} n \left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})} + 1 \right]^2 \right)$.* \bullet

5.2 Sorting on a General PMD

In this section we consider a general PMD where the underlying parallel machine can either be structured (e.g., the mesh, the hypercube, etc.) or unstructured (e.g., SMP, a cluster of workstations, etc.).

We can apply LMM on a general PMD in which case the number of I/O operations will remain the same, i.e., $\frac{N}{DB} \left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})} + 1 \right]^2$. As has become clear from our discussion on the mesh, we need mechanisms for $k - k$ routing and $k - k$ sorting. 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 $\leq \frac{N}{DM}(R_M + S_M) \left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})} + 1 \right]^2$.

Thus we get the following general Theorem.

Theorem 5.4 *Sorting on a PMD can be performed in $\frac{N}{DB} \left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})} + 1 \right]^2$ I/O operations. The total communication time is $\leq \frac{N}{DM}(R_M + S_M) \left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})} + 1 \right]^2$.* \bullet

6 Experimental Results

In this section we report our experimental evaluation of the PMD when the underlying parallel machine is a network of workstations. The problem considered is sorting. PVM has been employed to achieve interprocessor communications. We assume that each processor has a disk associated with it and has a core memory of size M .

Let P be the number of processors. In our implementation, we had $N = PM$, where N is the input size. Disks are realized using files. Let F_i be the file associated with processor i , for $i = 1, 2, \dots, P$. To begin with each processor has M keys. At the end, the first processor will have the smallest M keys, the second processor will have the next smallest M keys, and so on.

LMM sort is a recursive algorithm. It may not help to run several levels of recursion in practice since the communication cost will be prohibitive. Thus we have used a simplified version of LMM sort. To begin with, each processor sorts its M keys. Let X_i be the sorted sequence at processor i , for $i = 1, 2, \dots, P$. Then we use the (ℓ, m) -merge algorithm (with $\ell = P$) to merge the P sorted sequences except that we don't do the merge recursively. The value of m has been taken to be 240. We have varied the value of m to see its influence on the run time. But the speedup we get seems to be nearly the same over a variety of values for m .

The files associated with the different processors have been stored in the same directory. In Step 1 of (ℓ, m) -merge, the unshuffle is done in parallel. The resultant parts of processor i (call them $X_i^1, X_i^2, \dots, X_i^m$) are written back in file F_i .

In Step 2, we need to merge $X_1^i, X_2^i, \dots, X_P^i$ for $i = 1, 2, \dots, m$. This is done in parallel (without recursion). Note that file F_j has $F_j^1, F_j^2, \dots, F_j^m$, for $j = 1, 2, \dots, P$. Each processor reads M relevant keys, merges them locally (by sorting them), and outputs them back in its file.

For example, processor 1 reads $X_1^1, X_2^1, \dots, X_P^1$. At the same time processor 2 reads $X_1^2, X_2^2, \dots, X_P^2$, and so on. After the processors read the relevant keys, they perform merging in parallel.

In Step 3 also we perform shuffling in parallel along the same lines as in Step 2. In Step 4 we clean the dirty sequence. The length of the dirty sequence is at most Pm . For simplicity we choose the number of keys in each file to be a divisor of $2Pm$. By doing this our program can achieve perfect parallelism even in Step 4.

Our goal was to compare the performance of the PMD algorithm with a sequential algorithm. The sequential algorithm is the same as the parallel algorithm (i.e., LMM sort) with the same values for ℓ and m . We assumed a single processor with a core memory of size M and a single disk.

In the past we have implemented various in-core sorting algorithms (both deterministic and randomized) on a network of workstations to see if we can achieve decent speedups. But we have failed in each attempt. The reason is that the run time of the sorting algorithms we tried were $O(N \log N)$ (N being the input size). For this low a run time, speedups are difficult to achieve since the communication costs will be dominating. Thus we were wondering if it was possible to achieve speedups using the PMD model.

Our experimental results indicate that it is possible to obtain impressive speedups using the PMD model. The table below summarizes the case of 4 processors. The times shown are seconds.

Input Size	Sequential Time	PMD Time	Speedup
1,310,720	86	23	3.74
655,360	44.8	15.1	2.98
327,680	22.4	8.8	2.55

When the input size decreases, the speedup also decreases. This is because of the fact that the communication time becomes considerable when the input size decreases.

We are conducting more experiments varying the number of processors and the value of m , etc. More data will be provided in the final version of this paper.

7 Conclusions

We have investigated a straight forward model of computing with multiple disks (which can be thought of as a special case of the HMM). This model, PMD, can be thought of as a realization of prior models such as the PDS. We have also presented a sorting algorithm for the PMD. An interesting open problem is if we can avoid the use of $k - k$ routing and $k - k$ sorting algorithms, instead use some algorithms specific to the permutations under concern (i.e., unshuffle and shuffle), and hence improve the underlying constant in the communication complexity. Investigating other problems on the PMD is also open. Our experimental results for sorting indicate that we can get decent speedups in practice using the PMD model.

Acknowledgements

The first author thanks Tom Cormen and Matt Pearson for many fruitful discussions.

References

- [1] A. Aggarwal and C. G. Plaxton, Optimal Parallel Sorting in Multi-Level Storage, *Proc. Fifth Annual ACM Symposium on Discrete Algorithms*, 1994, pp. 659-668.
- [2] A. Aggarwal and J. S. Vitter, The Input/Output Complexity of Sorting and Related Problems, *Communications of the ACM*, 1988, 31(9):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] R. Barve, E. F. Grove, and J. S. Vitter, Simple Randomized Mergesort on Parallel Disks, Technical Report CS-1996-15, Department of Computer Science, Duke University, October 1996.
- [5] T. Cormen, Determining an Out-Of-Core FFT Decomposition Strategy for Parallel Disks by Dynamic Programming, in *Algorithms for Parallel Processing*, IMA Volumes in Mathematics and its Applications, Vol. 105, Springer-Verlag, 1999, pp. 307-320.
- [6] T. Cormen and M. D. Pearson, *Personal Communication*.
- [7] M. Kaufmann, S. Rajasekaran, and J. F. Sibeyn, Matching the Bisection Bound for Routing and Sorting on the Mesh, *Proc. 4th Annual ACM Symposium*

- on Parallel Algorithms and Architectures, 1992, pp. 31-40.*
- [8] M. Kunde, Block Gossiping on Grids and Tori: Deterministic Sorting and Routing Match the Bisection Bound, *Proc. First Annual European Symposium on Algorithms*, Springer-Verlag Lecture Notes in Computer Science 726, 1993, pp. 272-283.
- [9] T. Leighton, Tight Bounds on the Complexity of Parallel Sorting, *IEEE Transactions on Computers* C34(4), 1985, pp. 344-354.
- [10] M. H. Nodine, J. S. Vitter, Large Scale Sorting in Parallel Memories, *Proc. Third Annual ACM Symposium on Parallel Algorithms and Architectures*, 1991, pp. 29-39.
- [11] M. H. Nodine, J. S. Vitter, Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors, *Proc. Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, 1993, pp. 120-129.
- [12] 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.
- [13] V. S. Pai, A. A. Schaffer, and P. J. Varman, Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging, *Theoretical Computer Science*, 1994, 128(2):211-239.
- [14] M. D. Pearson, Fast Out-of-Core Sorting on Parallel Disk Systems, Technical Report PCS-TR99-351, Dartmouth College, Computer Science, Hanover, NH, June 1999, <ftp://ftp.cs.dartmouth.edu/TR/TR99-351.ps.Z>.
- [15] S. Rajasekaran, Sorting and Selection on Interconnection Networks, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* 21, 1995, pp. 275-296.
- [16] S. Rajasekaran, A Framework For Simple Sorting Algorithms On Parallel Disk Systems, *Proc. 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, 1998, pp. 88-97.
- [17] S. Rajasekaran, Selection Algorithms for the Parallel Disk Systems, *Proc. International Conference on High Performance Computing*, 1998.
- [18] C. D. Thompson and H. T. Kung, Sorting on a Mesh Connected Parallel Computer, *Communications of the ACM* 20(4), 1977, pp. 263-271.
- [19] J. S. Vitter and E. A. M. Shriver, Algorithms for Parallel Memory I: Two-Level Memories, *Algorithmica* 12(2-3), 1994, pp. 110-147.