# A PRACTICAL REALIZATION OF PARALLEL DISKS FOR A DISTRIBUTED PARALLEL COMPUTING SYSTEM

By

XIAOMING JIN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2000

To my  son, my wife Ping Zhang, my teacher Sanguthevar Rajasekaran

ACKNOWLEDGMENTS

I would like to express my appreciation and gratitude to my advisor, Dr. Sanguthevar Rajasekaran, for his guidance during this study. Dr. Sanguthevar Rajasekaran has been very helpful for my work. It is because of his LMM sort algorithm, my work can be completed. I would also like to thank Drs. Doug Dankel and Paul Fishwick, my supervisory committee, who also make my work be possible.

I owe much gratitude to my mother and to my wife, Ping Zhang. Without their taking care of the family, I would not have time to complete my work.

TABLE OF CONTENTS

page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

# A PRACTICAL REALIZATION OF PARALLEL DISKS FOR A DISTRIBUTED PARALLEL COMPUTING SYSTEM

By

Xiaoming Jin

December 2000

Chair: Dr. Sanguthevar Rajasekaran
Major Department: Computer and Information Science and Engineering

Several models of parallel sorting are found in the literature. Among these models, the parallel disk models are proposed to alleviate the I/O bottleneck when handling large amounts of data. These models have the general theme of assuming multiple disks. For instance, the Parallel Disk Systems (PDS) model assumes D disks which are disposed on 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 thesis we present a new model for multiple disks and evaluate its performance. This model is called a Parallel Machine with Multiple Disks (PMD). A PMD model has multiple computers each of which is connected with one disk. A PMD model can be thought of as a realization of the PDS model. In this thesis, we also present a more practical model which is called Parallel Machines with multiple Files (PMF). A PMF model has multiple computers connected on a central file system. We investigate the

sorting problem on this new model. Our analysis demonstrates the practicality of the PMF. We also present experimental confirmation of this assertion with data from our implementation.

# CHAPTER 1
# INTRODUCTION

## 1.1 Scope and Objective

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 [17] . 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) [8,9] have been proposed in the literature to address this issue. Realizations of HMMs using PRAMs and hypercube have been explored [9]. Sorting algorithms on these realizations have been investigated.

In this thesis 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 [9]. 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 Rajasekaran's [14] PMD (where the underlying parallel machine is a mesh, a hypercube, and a cluster of workstations).

In particular, we present a new model which is called Parallel Machine with Multiple Files (PMF). A PMF model has multiple computers which are managed by a network file system. As in modern days, network file system has been a popular distributed file system, so this model can be more practical in real life. In this model, input data will be partitioned into several files which are stored in the file system. All computers can read and write data from these files. We show why this model has more appealing properties than other models. We compute the run times for sorting in this model. Also we compute the speed ups of using multiple computers in verse of using single computer. These analyses demonstrate the practicality of the PMF.

## 1.2 Thesis Outline

This thesis consists of 5 chapters. In addition to this introduction, the rest of the thesis is organized as follows.

In Chapter 2, we provide a summary of known algorithms for the PDS model.

In Chapter 3, we present details of the PMD model. To make our discussion concrete, we use the mesh as the topology of the underlying parallel machine. However, the discussion applies to any parallel machine. Also in this chapter, we state some routing and sorting algorithms which are applied on the PMD model. Especially, we give detail description of the LMM algorithm which played a vital role in both PMD and PMF models.

In Chapter 4, we show a more practical model, PMF model. We stated its structure, and give detail description of its implementation. Also, we show our experimental results of the PMF model.

Chapter 5 concludes the thesis.

# CHAPTER 2
# SORTING ON THE PDS MODEL

In this chapter, we present an overview sorting results on the PDS model which has the structure of multiple disks with a single computer.

## 2.1 The PDS Model

Sorting has been studied well on the PDS model (see Fig. 2.1). A known lower bound for the number of I/O read steps for parallel disk sorting is $\Omega(\frac{N}{DB}[\frac{\log(N/B)}{\log(M/B)}])$. 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 $O(\frac{N}{DB}[\frac{\log(N/B)}{\log(M/B)}])$ I/O read steps (see e.g., references 10, 1, and 3).

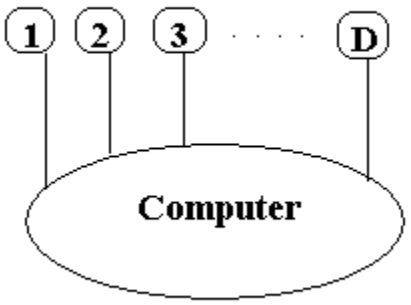


Figure 2.1. Architecture of a PDS Model

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 by Vitter and Shriver [17]. Several asymptotically optimal algorithms have been given for sorting on this model. Nodine and Vitter's optimal algorithm [8] 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 [17]. All of these results are highly nontrivial and theoretically interesting. However, the underlying constants in their time bounds are high.

## 2.2 Sorting on The PDS Model

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 algorithm is $\frac{N}{DB}(1+\frac{\log(N/M)}{\log(M/DB)})$. 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 above mentioned 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, research has been performed dealing with the practical aspects. Pai, Schaffer, and Varman [11] 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 $\text{Read}_{\text{SRM}}$ of I/O read operations made by their algorithm is such that

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

The algorithm merges R=kD runs at a time, for some integer k. When R = $\Omega$(DlogD), the expected performance of their algorithm is optimal. However, in this case, the internal memory needed is $\Omega$(BDlog D). They have also compared SRM with DSM through simulations and shown that SRM performs better than DSM. Recently, Rajasekaran [14] has presented an algorithm (called ($\ell$,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.

# CHAPTER 2
# A PARALLEL MACHINE WITH DISKS (PMD)

## 3.1 The PMD Model

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 through 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.

PMD can be thought of as a special case of the HMM [9]. Realization of HMM using PRAMs and hypercubes have already been studied [9].

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 (see Fig. 3.1) 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., Rajasekaran [13]). Some popular indexing schemes are column major, row major, snake-like row, blockwise row-major, etc. For the algorithm to be presented in this thesis, any of these schemes can be employed.

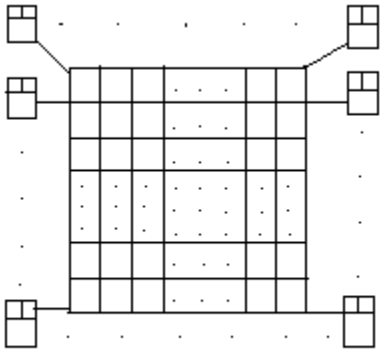

Figure. 3.1. A PMD Model of $n \times n$ Mesh

The algorithm to be presented in this thesis 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 $c\alpha f(n)$ with probability $\geq (1-n^{-\alpha})$, where c is a constant and $\alpha$ is a constant $\geq 1$. We can also define other asymptotic functions in a similar fashion.

## 3.2 Sorting Algorithms

Parallel sorting algorithm have been widely studied due to its classical importance and fundamentals. A lot of sorting algorithms have been presented. In this section, we talk about the problem of packet routing which plays a vital role in the design of algorithm on any parallel machine. We present the algorithms of k-k routing and k-k sorting. Also we give detail explanation of ($l$,m)-merge sort (LMM) algorithm which is the central spirit of sorting on the PMD model and PMF model.

### 3.2.1 Algorithm of k-k Routing and k-k Sorting

The packet routing problem can be defined as following. 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., Rajasekaran [13]).

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 Kaufmann, Rajasekaran, and Sibeyn [5].

[Lemma 3.2.1.1]

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., Rajasekaran [13]) sorting.

[Lemma 3.2.1.2]

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., [6]) for k-k routing and k-k sorting whose run times match those stated in Lemmas 3.2.1.1 and 3.2.1.2. However, we believe that the use of randomized algorithms will result in better performance in practice.

### 3.2.2 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 merging* refer to the task of scanning through the input once and performing R-way merging. Note that each phase of merging 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 Rajasekaran [14] is also based on merging. It employs R=l, for some appropriate l. The LMM is a generalization of the odd-even merge sort, the s2-way merge sort of Thompson and Kung [16], and the columnsort algorithm of Leighton [7].

[3.2.2.1 Algorithm Odd-Even Mergesort]

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 = $u_1,u_2,\ldots,u_q$ and V = $v_1,v_2,\ldots,v_q$ be the two sorted sequences to be merged. *Unshuffle* U into two, i.e., partition U into two: $U_{odd}$ = $u_1,u_3,\ldots,u_{q-1}$ and $U_{even}$= $u_2,u_4,\ldots,u_q$. Similarly partition V into $V_{odd}$ and $V_{even}$.
2. Now recursively merge $U_{odd}$ with $V_{odd.}$ Let X = $x_1,x_2,\ldots,x_q$ be the result. Also merge $U_{even}$ with $V_{even}$. Let Y = $y_1,y_2,\ldots,y_q$ be the result.
3. *Shuffle* X and Y, i.e., form the sequence: Z = $x_1$, $y_1$, $x_2$, $y_2,\ldots$, $x_q,y_q$.
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 [16] 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.

[3.2.2.2 Algorithm ($l$,m)-Merge]

LMM is a generalization of $s^2$-way merge sort algorithm. It uses R = $l$. 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.

1. Let the sequences to be merged be $U_i = u_i^1, u_i^2, \ldots, u_i^r$, for $1 \le i \le l$. If r is small use a base case algorithm. Otherwise, unshuffle each $U_i$ into m parts. In particular, partition $U_i$ into $U_i^1, U_i^2, \ldots, U_i^m$, where $U_i^1 = u_i^1, u_i^{1+m}, \ldots$; $U_i^2 = u_i^2, u_i^{2+m}, \ldots$; and so on.
2. Recursively merge $U_1^j, U_2^j, \ldots, U_l^j$, for $1 \le j \le m$. Let the merged sequences be $X_j = x_j^1, x_j^2, \ldots, x_j^{lr/m}$, for $1 \le j \le m$.
3. Shuffle $X_1, X_2, \ldots, X_m$, i.e., form the sequence $Z = x_1^1, x_2^1, \ldots, x_m^1, x_1^2, x_2^2, \ldots, x_m^2, \ldots, x_1^{lr/m}, x_2^{lr/m}, \ldots, x_m^{lr/m}$.
4. It can be shown that at this point the length of the "dirty sequence" (i.e., unsorted portion) is no more than $l$m. 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 $l$m elements of Z as $Z_1$; the next $l$m elements as $Z_2$; and so on. In other words, Z is partitioned into $Z_1$, $Z_2$,…, $Z_{r/m}$. Sort each ne of the $Z_i$'s. Followed by this merge $Z_1$, $Z_2$; 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 Rajasekaran [14]. The number of I/O operations needed in this implementation has been shown to be $\frac{N}{DB}[\frac{\log(N/M)}{\log(\min\{\sqrt{M}, M/B\})}+1]^2$. 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 Rajasekaran [14] it has been demonstrated that LMM can be faster than the DSM when D is large. Recent implementation results of Pearson [12] 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.

## 3.3 Sorting on the PMD Model

We begin by considering the sorting problem on the mesh. The result can be generalized to any parallel machine.

3.3.1 Sorting on 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 3.2.1.1 and 3.2.1.2) 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 $\frac{N}{DB}$ since each processor need to have $\frac{M}{B}$ I/O for each run. 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 sorting (with k = M) and each such sort takes kn + $\tilde{O}$(kn) steps.

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 Rajasekaran [14]. Each run will be striped across the disks. If R ≥ 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 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.

### 3.3.2 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 $l$ = m = $\sqrt{DM}$.

Let $U_1, U_2, \ldots, U_{\sqrt{DM}}$ 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 [14] for details). Thus we get:

[Lemma 3.3.2.1]

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 $l$ = m = $\frac{DM}{B}$.

In this case we can obtain:

[Lemma 3.3.2.2]

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

3.3.3 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 $l$ = 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(,) can be expressed as follows.

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

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 $K^2$DM 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 $l$ = 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 3.3.2.1) 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 = $\frac{\log(N/DM)}{\log(DM)}$. Now, we have the following:

[Theorem 3.3.3.1]

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

The number of communication steps is no more than

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

The second case to be considered is when $\frac{DM}{B} < \sqrt{DM}$. Here ($l$,m)-merge will be used with $l$ = 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,

$$T(Q^d,DM) = T(Q,DM) + T(Q,QDM) + \cdots + T(Q,Q^{d-1}DM). \qquad (3)$$

Also, we can get,

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

Here the fact T(Q,DM) = 3 (c.f. Lemma 3.3.2.2) 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 3.3.3.2]

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

Theorem 3.3.3.1 and theorem 3.3.3.2 readily yield:

[Theorem 3.3.3.3]

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

3.3.4 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., $\frac{N}{DB}[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})}+1]^2$. 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 $\frac{N}{DB}(R_M + S_M)$, implying that the total communication time for the entire algorithm will be

$$\leq \frac{N}{DB}(R_M + S_M)[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})}+1]^2.$$

Thus we get the following general Theorem:

[Theorem 3.3.4.1]

Sorting on a general PMD model 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 $\leq \frac{N}{DB}(R_M + S_M)\left[\frac{\log(N/DM)}{\log(\min\{\sqrt{DM}, DM/B\})}+1\right]^2$.

# CHAPTER 4
# PARALLEL MACHINE WITH MULTIPLE FILES (PMF)

In this chapter, we present a more practical parallel computing model called PMF. We show why this model has some advantages and report our experimental evaluation of the PMF.

## 4.1 Introduction of the PMF Model

A PMF model is nothing but multiple computers managed in a network file system. The underlying parallel machine is a network of workstations. In this model, input data will be partitioned into several files which are stored as source data (see Fig. 4.1). Computers can read and write data from these files.

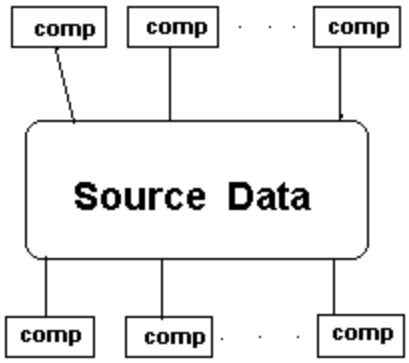


Figure. 4.1. Logical PMF Model.

The sorting problem on the PMF can be defined as follows. There are a total of N records to be sorted, and there are P number of processors. The problem is to sort the N records into descending or ascending order. We partition input data into P parts, and

stored them in P files each with size of N/P. We also index each file and processor as $F_1,F_2,\ldots,F_p$ and $P_1,P_2,\ldots,P_p$ respectively. After sorting, the smallest $\frac{N}{P}$ records will be output in $F_1$, the next smallest $\frac{N}{P}$ records will be output in $F_2$, and so on.

The LMM sort played a vital role in PMF Model. LMM algorithm consists of shuffling, unshuffling and local sorting steps. In each step, the data are partitioned into many parts, and each processor can pick any part from the partition and do the sorting, unshuffling, shuffling and cleaning. This property is essential in PMF model because processors do not need to communicate to get data from each other.

Since there is no communication between processors, the I/O time become critical. But in PMF model, if we increase the number of processors, the I/O time will be reduced. Using 2n processors will reduce the I/O time by half comparing with using n processors.

All of the advantages of this parallel computing model is that it sorts data in perfect parallel without any communications between each processor.

## 4.2 Sorting on the PMF Model

In this section, we show in detail about how to implement LMM sort on the PMF model, and how to partition data so that processors can read and write data in parallel without any communications between processors. LMM sort is a recursive algorithm. Here, we do not do the merge recursively, but proceed it just one level. Meanwhile, we use Quick sort or Heap sort for our local sorting. The detailed description of sorting follows.

The input data size is N, and there are q processors. We create q number of files each of size N/q. Also suppose the memory size is M and it is much less than N/q. The

chosen value of m can be random. We have varied the value of m to see its influence on the run time, but it seems to be nearly the same. Here, for easy implementation, we choose m to be a divisor of N/q, and also m can be divided by q. Without losing any generality, we suppose M is a divisor of N/q and (N/q)/M=r.

Step 1 We mark the files as F1, F2, …, Fq. Also we mark processors as P1, P2,…, Pq. At first, processor Pi will input the first M data from Fi, sort it, unshuffle it into m parts with each part of size a=M/m, and then put it back to its original place (row 1) in Fi, for $1 \le i \le q$. Then Pi input the second M data from Fi, sort it, unshuffle it into m part, and the put it back to its original place (row 2) in Fi. This procedure will continue until r times. We have each file Fi with r unshuffled sequences (see Fig. 4.2), for $1 \le i \le q$. So after step 1, there will be total r*q unshuffled sequences.

|← ---------- unshuffle M into m parts each with size of a ----------→|

|← a →|

| | | | | | |
|---|---|---|---|---|---|
| Row 1: part1 | part 2 | part 3 | part 4 | … | part m |
| Row 2: part1 | part 2 | part 3 | part 4 | … | part m |
| Row 3: part1 | part 2 | part 3 | part 4 | … | part m |
| **x** | **x** | **x** | **x** | **x** | **x** |
| Row r: part1 | part2 | part 3 | part 4 | … | part m |

Figure. 4.2. Unshuffle Result

As we notice, the sorting and unshuffling are done in parallel. There is no communication between each processor since each processor only deal with its corresponding files. Meanwhile, if we increase the number of processors, we also increase the number of files respectively which cause less data in each file. So if the data size is fixed, increasing processors will reduce the number of I/O .

**Step 2** In this step, we want to merge part 1 from all q files, part2 from all q files,…, and part m from all q files (see Fig. 4.2). This can also be done in parallel. But we need to create another q number of files to contain the merged results in order to read data and output data in perfect parallel. Let's call these additional q files as D1, D2, …, Dq. As we mentioned above, our choice of m can be divided by q. Suppose k=m/q. The scheme is as following:

First step: Processor j read part j from file F(j mod q).

Processor j reads part j from file F((j+1) mod q).

Processor j reads part j from file F((j+2) mod q).

**x**

Processor j reads part j from file F((j+q-1) mod q).

Notice that from each file, processor j will read in r number of part j, so there are total r*q number of part j to merge. After processor j merged these r*q sequences, it outputs the merged result to the first place (row 1) of Dj (see Figure. 4.1.3). We also notice that the read data, merge data, and output data are all done in parallel: when processor j is reading data from file F(j mod q), processor (j+1) is reading data from file F((j+1) mod q); when processor j is reading data from F((j+1) mod q), processor (j+1) is

read data from file F((j+2) mod q). Meanwhile, different processor output merged data into different files.

Second step: Processor j reads part (q+j) from file F(j mod q).

Processor j reads part (q+j) from file F((j+1) mod q).

**x**

Processor j reads part (q+j) from file F((j+q-1) mod q).

After reading in the r*q sequences, processor j merges them and put the merged result to the second place of Dj (see Fig. 4.1.3).

**x**

**x**

$k^{th}$ step: Processor j reads part ((k-1)*q+j) from file F(j mod q).

Processor j reads part ((k-1)*q+j) from file F((j+1) mod q).

**x**

Processor j reads part ((k-1)*q+j) from file F((j+q-1) mod q).

After reading in the r*q sequences, processor j merges them and put the merged result to the $k^{th}$ place of Dj (see Fig. 4.3).

After as many as k merging steps, processor j will generate k merged sequences which are outputted in file Dj. The size of each merged sequence can be easily computed as mergeSize=r*a*q=N/m, and we must make sure that mergeSize ≤ M (our choice of m must meet this requirement).

File Dj

|←---------- mergeSize ----------→|

| Row 1: merge result of part j by processor j |
|---|
| Row 2: merge result of part (q+j) by processor j |
| **x** |
| Row k: merge result of part ((k-1)*q+j) by processor j |

Figure. 4.3. Merging Result

From the analysis above, we see each merging step in Step 2 is done in parallel. We can also compute the I/O time for each processor. The logical I/O time will be as following.

I/O time = k*(input data time for each step + output data time for each step)

= k*( r*q + 1)

= m*(r+1/q).

If data size is fixed, and if we double the number of processors, it is obvious that the I/O time will be reduced by ½ from the above equation.

Step 3 In this step, we try to shuffle the m merged sequences. We want to read data, shuffle data, and output data in parallel. The scheme is as following:

We partition file Dj ($0 \le j \le q$) into q part (see Fig. 4.4).

File Dj

|← part 1 →|← part 2 →|← part 3 →| … |← part q →|

| | | | | |
|---|---|---|---|---|
| Row 1 | | | … | |
| Row 2 | | | … | |
| **x** | **x** | **x** | … | |
| Row k | | | … | |

Figure. 4.4. Partition File into q Parts

Each processor will be responsible for its own part. So processor j will shuffle only part j. By partition the data in this way, processors can read data, shuffle data, and output data in parallel.

First step: Processor j reads part j from file D(j mod q).

Second step: Processor j reads part j from file D((j+1) mod q).

**x**

qth step: Processor j reads part j from file D((j+q-1) mod q).

We notice that each processor reads data in parallel: when processor j is reading part j from file D(j mod q), processor (j+1) is reading part (j+1) from file D((j+1) mod q); when processor j is reading part j from file D((j+1) mod q), processor (j+1) is read part (j+1) from file D((j+2) mod q). After each processor Dj read its data, it will perform shuffling, and then output the result to file Fj.

The above analysis is just for easy understanding of how to shuffle data in parallel. In actual practice, we also need to partition each of the q parts into r cells with each cell having data of size a, where r = (N/q)/M and a = M/m which we already mentioned in Step 1. This partition is necessary because if each processor Dj input part j from all q files, the size of the input data is definitely larger than memory size.

Now, we show the detail implementation of Step 3. We need to partition each of the q parts into r cells (see Fig. 4.5).

File Dj (ci represent cell i)

|← ---- part 1 --- →|← ---- part 2 --- →| . … . . . |← --- part q ---- →|

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| c1 | c2 | … | cr | c1 | c2 | … | cr | . . . | c1 | c2 | … | cr |
| c1 | c2 | … | cr | c1 | c2 | … | cr | . . . | c1 | c2 | … | cr |
| : | : | : | : | : | : | : | : | : | : | : | : | : |
| : | : | : | : | : | : | : | : | : | : | : | : | : |
| c1 | c2 | … | cr | c1 | c2 | … | cr | …. | c1 | c2 | … | cr |

Figure. 4.5. Partition Parts into r Cells

First step: Processor j reads c1 from part j of file D(j mod q).

Processor j reads c1 from part j of file D((j+1) mod q).

Processor j reads c1 from part j of file D((j+2) mod q).

**x**

Processor j reads c1 from part j of file D((j+q-1) mod q).

After processor j reads in the data, it will shuffle them, and then output result to the first place of file Fj.

Second step: Processor j reads c2 from part j of file D(j mod q).

Processor j reads c2 from part j of file D((j+1) mod q).

Processor j reads c2 from part j of file D((j+2) mod q).

x

Processor j reads c2 from part j of file D((j+q-1) mod q).

After processor j reads in the data, it will shuffle them, and then output result to the second place of file Fj.

x

x

rth step: Processor j reads cr from part j of file D(j mod q).

Processor j reads cr from part j of file D((j+1) mod q).

Processor j reads cr from part j of file D((j+2) mod q).

x

Processor j reads cr from part j of file D((j+q-1) mod q).

After processor j reads in the data, it will shuffle them, and then output result to the $r^{th}$ place of file Fj.

After as many as r steps, we finished shuffling the data. From the analysis above, Each processor read data, shuffle data, and output data in parallel.

Logical I/O time = r*(input data time for one step + output data time for one step)

= r*(k*q+1)

= r*((m/q)*q+1)

= r*(m+1).

If we double the number of processors, r will be reduced by ½. So doubling the number of processors will also reduce the I/O time by ½.

**Step 4** In this step, we will try to clean the dirty sequences. We know the size of the dirty sequence is no more than l*m = (N/M)*m. The cleaning can be very simple. Each processor j clean the dirty sequence of file Fj. Since each processor clean different file, parallel can easily proceed.

Logical I/O time = 2*(N/q)/(l*m)

= (2*M)/(q*m).

If we double the number of processors, the I/O time will be reduced by ½.

After **Step 4**, we finish the sorting. As we can see from the above analysis, there is no communication between each processor. All the sorting procedures are done in parallel. Meanwhile, If we double the number of processors, the I/O time will be reduced by ½.

## 4.3 Computing the Speed Up

In this section, we report our experimental evaluation of the PMF. We employ 2, 4, and 8 processors to sort different size of data and compute the real time speed up by comparing the result of using only 1 processor. In this simulate application, we fix the memory size to be 138240 byte. The value of m is 120. The data we generate are random floating numbers. We use two different sorting algorithms Quick sort and Heap sort, to sort the local data.

### 4.3.1 Using Quick Sort

Here, we use quick sort for our local sorting. The testing results are shown in the following tables. Also, we show a chart of the data to illustrate the appealing speed up.

Table 4.1. Quick Sort Results Of Using 2 Processors

| Input Data Size | 1 Processor | 2 Processor | Speed Up |
|---|---|---|---|
| 3317760 bytes | 41.2 seconds | 22.1 seconds | 1.86 |
| 6635520 bytes | 66.4 seconds | 34.1 seconds | 1.94 |
| 9953280 bytes | 87.2 seconds | 44.5 seconds | 1.97 |
| 13271040 bytes | 104.8 seconds | 53.2 seconds | 1.97 |

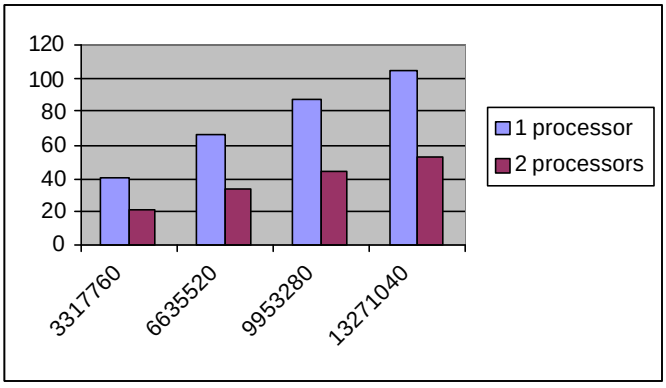


Figure. 4.6. Quick Sort Speed Up Chart Using 2 Processors

Table 4.2. Quick Sort Results Of Using 4 Processors

| Input Data Size | 1 Processor | 4 Processor | Speed Up |
|---|---|---|---|
| 3317760 bytes | 41.2 seconds | 13.4 seconds | 3.07 |
| 6635520 bytes | 66.4 seconds | 21.6 seconds | 3.07 |
| 9953280 bytes | 87.2 seconds | 27.2 seconds | 3.20 |
| 13271040 bytes | 104.8 seconds | 33.1 seconds | 3.16 |

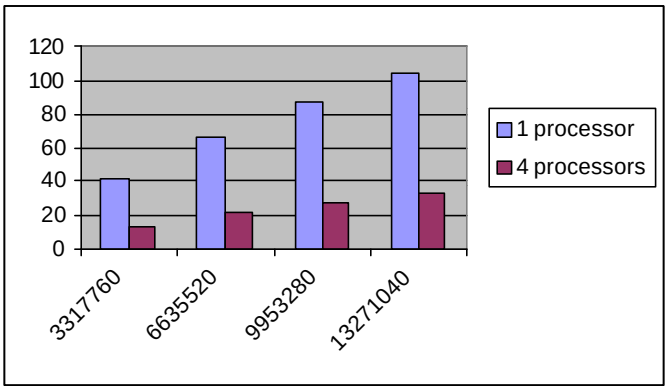


Figure. 4.7. Quick Sort Speed Up Chart Using 4 Processors

Table 4.3. Quick Sort Results Of Using 8 Processors

| Input Data Size | 1 Processor | 8 Processor | Speed Up |
|---|---|---|---|
| 3317760 bytes | 41.2 seconds | 9.70 seconds | 4.24 |
| 6635520 bytes | 66.4 seconds | 14.2 seconds | 4.67 |
| 9953280 bytes | 87.2 seconds | 18.4 seconds | 4.74 |
| 13271040 bytes | 104.8 seconds | 23.1 seconds | 4.53 |

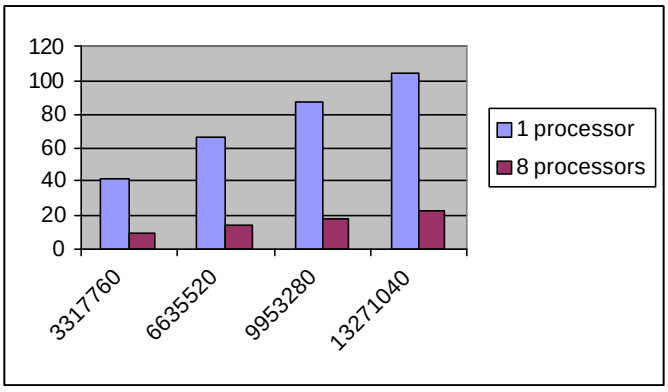


Figure. 4.8. Quick Sort Speed Up Chart Using 8 Processors

4.3.2 Using Heap Sort

Here, we use heap sort for our local sorting. The testing results are shown in the following tables. Also, we show the charts to illustrate the appealing speed up.

Table 4.4. Heap Sort Results Of Using 2 Processors

| **Input Data Size** | **1 Processor** | **2 Processor** | **Speed Up** |
|---|---|---|---|
| 3317760 bytes | 44.6 seconds | 23.2 seconds | 1.92 |
| 6635520 bytes | 68.1 seconds | 35.8 seconds | 1.90 |
| 9953280 bytes | 90.2 seconds | 46.8 seconds | 1.92 |
| 13271040 bytes | 113.1 seconds | 57.9 seconds | 1.95 |

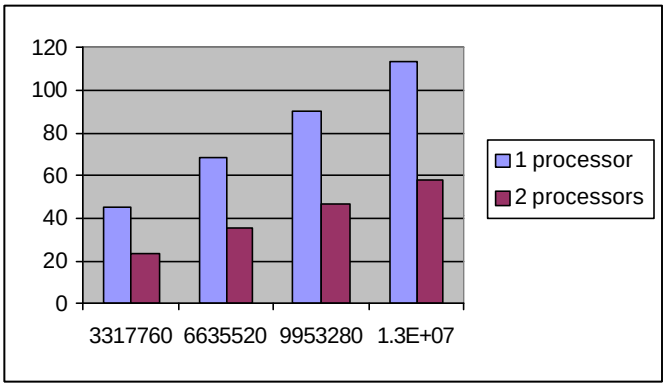


Figure. 4.9. Heap Sort Speed Up Chart Using 2 Processors

Table 4.5. Heap Sort Results Of Using 4 Processors

| Input Data Size | 1 Processor | 4 Processor | Speed Up |
|---|---|---|---|
| 3317760 bytes | 44.6 seconds | 14.2 seconds | 3.14 |
| 6635520 bytes | 68.1 seconds | 21.8 seconds | 3.11 |
| 9953280 bytes | 90.2 seconds | 28.1 seconds | 3.21 |
| 13271040 bytes | 113.1 seconds | 34.8 seconds | 3.25 |

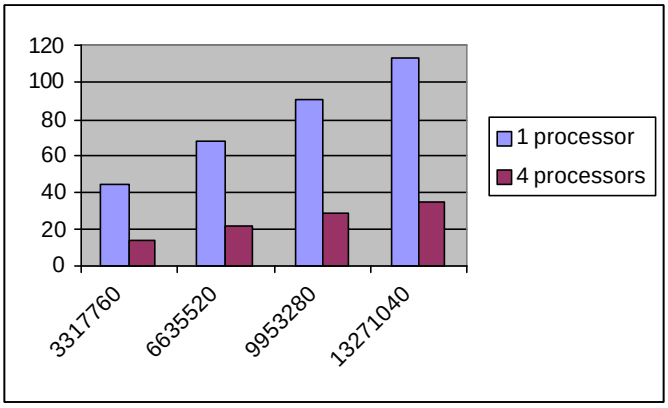


Figure. 4.10. Heap Sort Speed Up Chart Using 4 Processors

Table 4.6. Heap Sort Results Of Using 8 Processors

| Input Data Size | 1 Processor | 8 Processor | Speed Up |
|---|---|---|---|
| 3317760 bytes | 44.6 seconds | 9.79 seconds | 4.56 |
| 6635520 bytes | 68.1 seconds | 14.2 seconds | 4.79 |
| 9953280 bytes | 90.2 seconds | 18.9 seconds | 4.76 |
| 13271040 bytes | 113.1 seconds | 23..5 seconds | 4.81 |

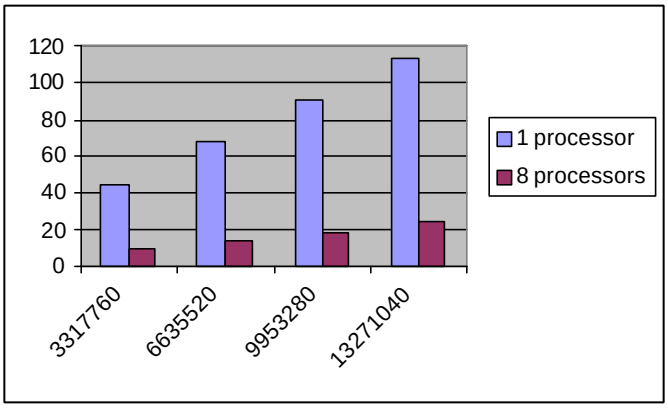


Figure. 4.11. Heap Sort Speed Up Chart Using 8 Processors

# CHAPTER 5
# CONCLUSION

## 5.1 Major Results

Parallel sorting algorithm have been widely studied due to its classical importance and fundamentals. Many sequential sorting algorithms have been presented. Even though there are many sorting algorithms, many of them are less practical due to the large constants in their run time.

In this thesis, 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. Here, we use LMM sort algorithm on the PMD. But for local sorting, we have to use k-k routing and k-k sorting algorithm due to the communication concern. The I/O time and communication time are evaluated on the PMD. From the analysis of the result, the communication time is still significant.

On our research, reducing communication time and I/O time are the major concerns. Here, we presented the PMF model. The underlying parallel machine is a network of workstations. The data are partitioned and stored in several files which are managed by a network file system. The LMM sort algorithm plays a vital role on the PMF model. The LMM sort consists of sorting, shuffling and unshuffling. These require to partition data into many parts. This property make it possible to let different processors read and write different parts of data from different files in parallel. Because of this, there

is no communications between processors, and hence, we overcome the communication overhead. Meanwhile, if we increase the number of processors, I/O time will also be improved. Our experimental results for sorting indicate that we can get decent speedups in practice using the PMF model.

## 5.2 Future Work

From our testing results, we can see that when we use two processors, the speed up is almost 2. But as we use four processors, the speed up is close to 3. Especially, when we use 8 processors, the speed up is only close to 5. The results are not as optimal as we analyzed. In our research, we believe this problem is because of initial start up communication delay. The more processors we use will cause more delays. Meanwhile, different processors' speed can be different, and this will also bad effects on the speed up.

Our future research will be concerning with these problems.

REFERENCES


[1] A. Aggarwal and C. G. Plaxton, Optimal Parallel Sorting in Multi-Level Storage, *Proc. Fifth Annual ACM Symposium on Discrete Algorithms*, New York, 1994, pp. 659-668.

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

[3] L. Arge, The Buffer Tree: A New Technique for Optimal I/O-Algorithms, *Proc. 4th International Workshop on Algorithms and Data Structures (WADS)*, New York, 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, Durham, NC, October 1996.

[5] 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*, New York, 1992, pp. 31-40.

[6] 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, New York, 1993, pp. 272-283.

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

[8] M. H. Nodine, J. S. Vitter, Large Scale Sorting in Paralle Memories, *Proc. Third Annual ACM Symposium on Parallel Algorithms and Architectures*, New York, 1991, pp. 29-39.

[9] M. H. Nodine, and J. S. Vitter, Deterministic Distribution Sort in Shared and Distributed Memory Multiprocessors, *Proc. Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, New York, 1993, pp. 120-129.

[10] 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.

[11] V. S. Pai, A. A. Schaffer, and P. J. Varman, Markov Analysis of Multiple-Disk Prefetching Strategies for External Merging, *Theoretical Computer Science*, 128(2), 1994, pp. 211-239.

[12] 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.

[13] S. Rajasekaran, Sorting and Selection on Interconnection Networks, *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 21, 1995, pp. 275-296.

[14] S. Rajasekaran, A Framework For Simple Sorting Algorithms On Parallel Disk Systems, *Proc. 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, New York, 1998, pp. 88-97.

[15] S. Rajasekaran, Selection Algorithms for the Parallel Disk Systems, *Proc. International Conference on High Performance Computing*, New York, 1998.

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

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

# BIOGRAPHICAL SKETCH

Xiaoming Jin was born in Nanjing, P.R.China. He is a Master of Science degree student in the Department of Computer and Information Science and Engineering at the University of Florida. Also, he is a Master of Science degree student in the Department of Mathematics at the University of Florida. He received his B.S. degree in mathematical science from Liberty University at Virginia.