EXTERNAL SORTING USING TRANSPUTER NETWORKS

by

Mark Maurice Joseph Mezofenyi

B.Math., University of Waterloo, 1989

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Mark Maurice Joseph Mezofenyi 1992
SIMON FRASER UNIVERSITY
December 1992

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: Mark Maurice Joseph Mezofenyi

Degree: Master of Science

Title of thesis: External Sorting using Transputer Networks

Examining Committee: Dr. Robert D. Cameron
Chair

Dr. M. Stella Atkins, Senior Supervisor

Dr. Wo-Shun Luk, Supervisor

Dr. Ze-Nian Li, S.F.U. Examiner

Date Approved: Dec. 7, 1992

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

External Sorting using Transputer Networks

Author:

(signature)

Mark Mezofenyi

(name)

December 16, 1992

(date)

Abstract

Sorting is an extremely common application for computers. A sort is *external* when the amount of data being sorted is too large to fit entirely within main memory. This thesis begins with a discussion of a well-known external sorting algorithm and enhancements of it. For a random input file stored on a single disk the maximum theoretical sort rate of the enhanced version of this algorithm is half the i/o rate of the disk. However, other factors affect the actual sort rate, such as the speed of the cpu, the amount of main memory, the speed of the i/o subsystem, the number of auxiliary disks available, the size of the records being sorted, and the size of the input file.

We describe an implementation of this external sort algorithm on a single transputer with a fixed amount of main memory. Using a single disk for both the input and output files we explore the effects of varying the software-dependent factors and the number of auxiliary disks on the sort rate. We find that a single transputer is, in most cases, not able to achieve the theoretical limit imposed by the i/o rate of a single disk.

In an attempt to attain this limit we implemented a parallel extension of the above external sort. We found that additional transputers can be used to bring the performance of the sort up to the theoretical single-disk limit for the largest files storable in our environment. We also show that, by using a storage node which contains more than a single disk, the limiting factor for the parallel version of the single-input single-output sort on the largest files in our environment becomes the speed of a transputer link.

The thesis concludes with a discussion of methods to parallelize further the external sort by using multiple storage nodes for both the input and output files, thus surmounting the bottleneck imposed by the inter-transputer link speed.

Acknowledgements

I would especially like to thank my Senior Supervisor, Dr. Stella Atkins, without whose support this thesis would not have been possible. I would also like to thank Timothy Dudra who patiently answered so many of my questions on TASS. Finally, I would like to thank Graham Finlayson for his ceaseless good cheer, Zhaohui Xie for coming to so many movies with me(!), and all the other foreign students who make SFU a vastly richer place to study.

Contents

Abstract iii

Acknowledgements iv

1 Introduction 1

1.1 Transputer Description 2

1.2 System Overview 3

1.3 Definitions and Assumptions 5

1.4 Thesis Overview 6

2 External Sorting With a Single CPU - Theory 8

2.1 Basic Algorithm 9

2.1.1 Overview of the Basic Algorithm 9

2.1.2 Phase I - Sort Phase 10

2.1.3 Phase II - Merge Phase 11

2.2 Theoretical Performance of the Basic Algorithm 12

2.2.1 Phase I - Sort Phase 13

2.2.2 Phase II - Merge Phase 14

2.3 Replacement Selection Sort Algorithm 22

3 TASS "File System" 29

4 External Sorting With a Single Transputer 33

4.1 Preliminary Remarks 33

4.2 Single Node; Single Disk Configuration: Non-Double-Buffered 37
4.2.1 Analysis 37
4.2.2 Results 39
4.3 Single Node; Single Disk Configuration - Double-Buffered 45
4.3.1 Analysis 45
4.3.2 Results 46
4.4 Single Node; Two Disk Configuration 49
4.5 Single Node; Three and More Disk Configurations 50
4.5.1 Using a Single TASS storage node 53
4.5.2 Using Multiple TASS storage nodes 63
4.6 Final Squeeze 66
4.7 Summary 67

5 External Sorting With Multiple CPUs **70**
5.1 Single-Input Single-Output 71
5.1.1 Sort Phase 71
5.1.2 Merge Phase 76
5.2 A Low Communication Sort (Multiple-Input Single-Output) 81
5.3 Multiple-Input Multiple-Output 84
5.3.1 Implementation of a Small Multiple-input Multiple-output External Sort 93
5.4 Summary 99

6 Summary and Future Work **101**
6.1 Summary 101
6.2 Future Work 102

Bibliography **106**

List of Figures

1.1 Our Transputer Hardware Environment 4
1.2 TASS Storage Node Configurations 5

2.1 Single T-Node Single Disk External Sort 10
2.2 Single T-Node Two Disks Configuration 13
2.3 Effect of Double-Buffering in Merge Phase 21
2.4 Snowplow Analogy 25

4.1 Possible Configurations for Storing Sorted Segments on Multiple Disks 52
4.2 Overview of 100MB data 68

5.1 Parallellizing the Sort Phase 71
5.2 Fastsort 77
5.3 Hypercubes 86
5.4 Parallel Shellsort 87
5.5 Parallel Quicksort 89
5.6 Small Scale Multi-input Multi-output External Sort 94
5.7 Final Overview of 100MB data 100

List of Tables

4.1 Sort Times and Throughput Rates for a Single T-Node with a Single Disk - NO double-buffering 40

4.2 Sort Times and Throughput Rates for a Single T-Node with a Single Disk - NO double-buffering 40

4.3 Sort Times and Throughput Rates for a Single T-Node with a Single Disk - WITH double-buffering 47

4.4 Sort Times and Throughput Rates for a Single T-Node with TWO Disks (with double-buffering) 50

4.5 Sort Times and Throughput Rates For a Single T-Node Using One TASS storage node with Multiple Disks for the Sorted Segments (no double-buffering) 55

4.6 Sort Times and Throughput Rates For a Single T-Node Using One TASS storage node with Multiple Disks for the Sorted Segments (with double-buffering) 55

4.7 Sort Times and Throughput Rates For a Single T-Node Using One TASS storage node with Multiple Disks for the Sorted Segments (no double-buffering) 56

4.8 Sort Times and Throughput Rates For a Single T-Node Using One TASS storage node with Multiple Disks for the Sorted Segments (with double-buffering) 56

4.9 Merge Phase, 50MB (16 segments), 1 disk 58

4.10 Merge Phase, 50MB (16 segments), 2 disks 58

4.11 Merge Phase, 50MB (16 segments), 3 disks 59

4.12 Merge Phase, 50MB (16 segments), 4 disks 59
4.13 Merge Phase, 100MB (32 segments), 2 disks 60
4.14 Merge Phase, 100MB (32 segments), 3 disks 60
4.15 Merge Phase, 100MB (32 segments), 4 disks 60
4.16 Sort Times and Throughput Rates for a Single T-Node with Multiple Single-Disk storage nodes for Sorted Segments 64
4.17 Sort Times and Throughput Rates Using One Double-Disk storage node and Two Single-Disk storage nodes 67

5.1 Sort Times and Throughput Rates When Using Multiple T-Nodes for Creation of Sorted Segments 75
5.2 Sort Times and Throughput Rates for a Fastsort Implementation . . 79
5.3 Maximum 2-way and 3-way Merge Rates (integer keys) 80
5.4 Sort Times and Throughput Rates When Using Multiple Storage Nodes for both Unsorted and Sorted Files 95

Chapter 1

Introduction

Sorting data is an extremely common function of computers. It has been estimated that sorting consumes approximately 25% of all computing time [Knu73]. Also, the efficient execution of a variety of other tasks (searching and relational joins, for example), depends upon sorted data. Because of its importance many people have studied sorting and an extensive repertoire of soring algorithms has been developed. There exist many books and surveys on the subject ([Knu73] and [BDHM84]). [BF82] and [Kwa86] are two studies of external sorting (defined next) in particular.

Sorting may be divided into two categories: *internal* and *external*. A sort is internal when all the data being sorted can reside within all available main memory at once. On a single-cpu or shared-memory parallel computer available main memory consists of all the RAM present in the one machine; on a distributed memory multicomputer available main memory is the sum of all the local RAM on each node. If the amount of data to be sorted is larger than can be stored in available RAM at one time, necessitating that some of the data must remain on secondary storage during the sort, then the sort is external. In this thesis we examine external sorting.

For both internal and external sorting algorithms involving comparison-exchange operations the theoretical lower bound on the number of such operations required during the sort is $O(n \log n)$. However, there are still at least two ways in which we may attempt to decrease the actual execution time of a sort program based on such

an optimal algorithm: 1) increase processor speed and/or 2) spread the comparison-exchange operations among multiple processors executing in parallel. We will see that for external sorting with transputers, based on present day technology, the first of these options is of limited value, especially for the largest files that can exist in our hardware environment (described below). This is because a primary bottleneck for external sorts in our environment is the maximum input/output rate achievable for secondary storage for a transputer. Thus, to increase the rate at which an external sort is performed *both* multiple processors *and* multiple disks are required.

1.1 Transputer Description

The *transputer* is a family of RISC processors which were introduced in 1985 by Inmos [Inm89]. Each transputer is equipped with four high speed bi-directional serial links which allow parallel processing networks to be built easily and economically. Each device is also equipped with 4 KBytes of on-chip memory and a memory interface allowing the chip to be augmented with external memory. Through the use of direct memory access circuitry, the transputer allows communication and computation to occur simultaneously and autonomously. There is no hardware support for shared memory between any two nodes in a transputer network, thus the autonomous communication, via the serial links, is required to provide transputer networks with an efficient message passing model.

The transputer additionally provides a two priority level process hierarchy, coupled with a hardware based process scheduler. The scheduler provides rapid context switching of 35 microseconds. This promotes the use of multiple process software environments, often implemented in the form of multi-threaded process groups. As there is no memory management provided by the device, all processes within one transputer share the same memory space.

There are two transputer variants discussed within this thesis, namely the 16 bit T222 and the 32 bit T800. Both provide all the features mentioned above, with the only notable difference being the limited 64 KByte address space of the T222.

As each transputer has four data links with which to pass data to other nodes a

large number of topologies can be easily built. This connection flexibility gives the system designer many choices of topology for exploiting the parallelism of a problem.

1.2 System Overview

The hardware is based on CSA transputer boards [Com89], each with 4 T800 transputers per board. Four transputers are connected to the VME-bus of a SUN4/110 host via serial links, shown in Figure 1.1. These root transputers are then connected via bi-directional transputer links to a network of 64 more transputers and 4 SCSI disks. Each T800 transputer is equipped with 2MB memory.

All software was written using Logical Systems C compiler [Com90]. This C system provides a library supporting the concurrency features of the transputer. The features of the library are similar to those provided by the OCCAM language originally designed for the transputers [Inm89].

The software we use to access a general mass storage system is called TASS (Transputer Auxiliary Storage System) [Dud92]. TASS provides the abstraction of a disk with N chunks rather than a conventional file service.

A TASS *storage node*, (also called a *disk node*) shown in Figure 1.2, is defined as a cache server, coupled with 1 to 3 subservient disk servers. The cache server runs on a T800 and each disk server runs on a T222. Each such storage node can support from 1 to 3 directly connected clients. Multiple disk address spaces are joined into one abstracted disk address space either by interleaving, concatenating or striping. The T800 cache controller is required, even for single disk systems, because each SCSI disk is controlled by a 16-bit T222 transputer having a 64K address space which is insufficient for caching large disk chunks. Each disk has a maximum capacity of approximately 110 MBytes and so the maximum capacity of a single storage node is 330 MBytes. We have a total of four disks with which to create storage nodes.

The random and sequential read access sustained data transfer rates of a storage node with a single disk attached to it are 206 KBytes/sec and 730 KBytes/sec, respectively. When a single storage node has two disks attached to it in an interleaved fashion these rates become 205 KBytes/sec and 1142 KBytes/sec, respectively; with

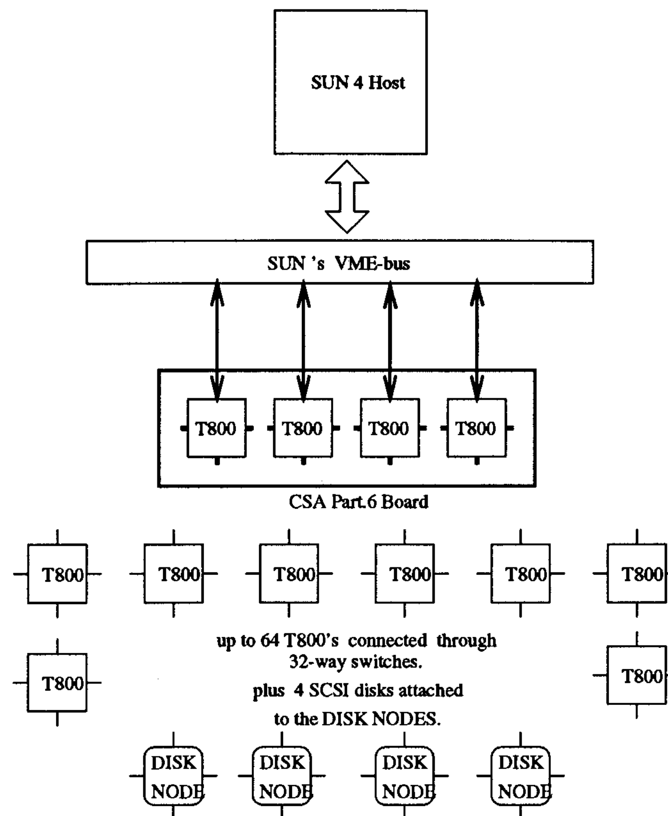


Figure 1.1: Our Transputer Hardware Environment

three interleaved disks the random access rate remains the same as when two disks are used but the sequential rate increases slightly to 1155 KBytes/sec. If two disks are attached to a single storage node in a striped fashion then the random sustained input rate is 341 KBytes/sec and the corresponding sequential rate is 1182 KBytes/sec; with three striped disks these figures are 446 KBytes/sec and 1204 KBytes/sec. Note that striped disks provides better performance than that of interleaved disks, and that this improvement is most pronounced with the random access rates. Finally, joining two or more disks on a single storage node in a concatenated fashion leaves the sequential input rate unchanged from that of a single disk storage node, as expected. The write performance for interleaved, striped, and concatenated disks are slightly less than the respective sequential read figures. These figures are roughly those given in [Dud92] which discusses this data in much more detail and also describes interleaving, striping,

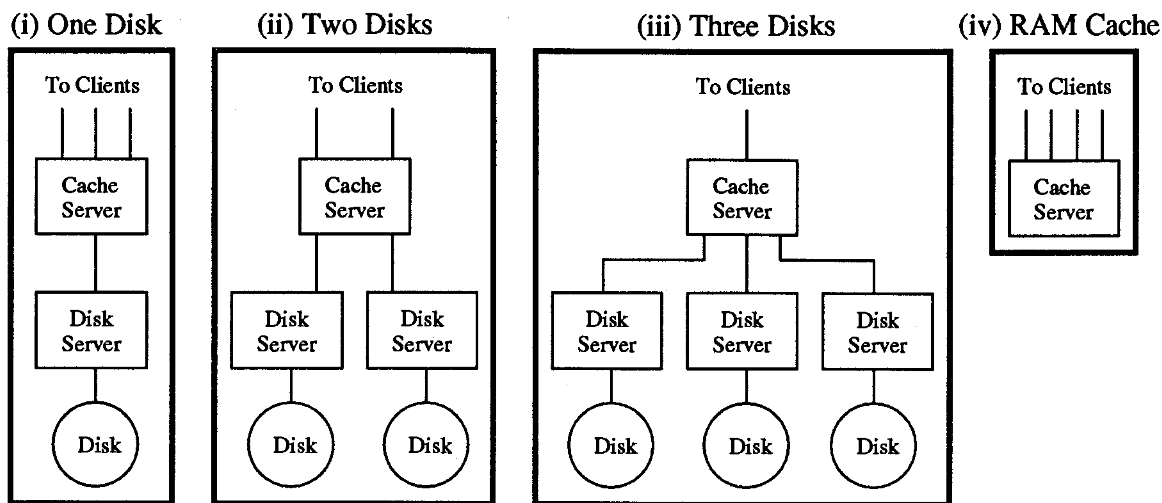


Figure 1.2: TASS Storage Node Configurations

and concatenating of disks.

It is interesting to briefly compare the performance of the various storage nodes described above with those of the example storage organizations described by Hennessy and Patterson [PH90]. In particular, they list practical storage configurations with a capacity of greater than 40GBytes capable of sustained *random* i/o rates between 3.3MBytes/sec and 16.4 MBytes/sec.[1] With transputers the best i/o rates achievable are limited by the transfer rate of a link (about 1.2 MBytes/sec).[2] Care must be taken when applying the conclusions we arrive at for our particular hardware environment to systems such as those described by Hennessy and Patterson.

1.3 Definitions and Assumptions

A *single-input* sort is one in which the input file exists on a single disk (or, in our environment, a single storage node) at the start of the sort. A *single-output* sort is one on which the sorted file produced by the sort is to be placed on a single storage node. A *multi-input* (*multi-output*) sort has its input (output) file spread among multiple storage nodes at the start (end) of the sort.

[1]The cost for these i/o configurations is approximately 400,000 1990 U.S. dollars.

[2]Since each transputer has four links the actual maximum i/o rates achievable are somewhat greater than 1.2 MBytes/sec.

A *block* or *chunk* is the smallest collection of data which can be transferred between a disk and the transputer node functioning as a disk server. It is analogous to a disk sector or track.

We make a number of asssumptions throughout this text. Firstly, we assume that we are the only user of the computer system during execution of the sort. In particular, this means that the only seeking performed by the disks we use is that which is required by the sort algorithm. We assume that we have complete control over the placement of data blocks on the disk. This means that we can choose to store a file contiguously on a disk. Although it can be argued that this is an unrealistic assumption we can suppose that this allows us to simulate with our relatively slow disks a faster random access i/o-subsystem such as the one described by Hennessy and Patterson which we mentioned above.

We assume that the entire contents of the file are to be sorted, ie. we are not satisfied with just a sorted index into the file based on the sort key. We assume that that file to be sorted must be read from and written to the disk at least once. That is, the disk on which the file resides is not "exotic" in any way, such as those mentioned in [BBH78]. Finally, we assume, with no loss of generality, that the records of the file are to be sorted into ascending order.

1.4 Thesis Overview

Chapter 2 presents a detailed examination of an efficient single cpu single-input single-output external sorting algorithm. Chapter 3 briefly discusses a "file system" which was constructed on top of TASS to facilitate implemenation of the external sorts examined in this thesis. In Chapter 4 we present timing results for executions of a single-input single-output external sort executing on a single transputer on files of various sizes. We also discuss the effects of record size and of various storage node configurations on performance.

In Chapter 5 we consider methods for parallellizing the single-input single-output external sort. We first describe a method for parallelizing the cpu activity of the sort and then discuss the results of a transputer-based implementation of this method. We

follow this with a recounting of a multi-input single-output external sort. We then give a brief discussion of multi-input multi-output external sorting. Finally, we discuss the results obtained from a transputer-based implementation of one such multi-input multi-output algorithm.

Chapter 2

External Sorting With a Single CPU - Theory

Because we are not using any type of exotic disk technology we assume that the file to be sorted must be read from disk at least once. If only a single disk is used then the time needed to write the sorted file must also be accounted for. Therefore, a loose lower bound on the time required to perform an external sort is the time it takes to read the input file once and write the sorted file once. In this chapter we discuss an external sort which can theoretically, given a fast enough processor with a large amount of RAM[1] and using a single disk, sort any file in a time close to twice this lower bound. If two disks are being used such that the sorted file can be written to one disk at the same time as the input file is being read from the other then this loose lower bound must be reduced to the time needed only to read the input file once or write the sorted file once, whichever takes longer. Indeed, when two disks are used the algorithm presented in this chapter can, in some cases, theoretically sort a file in the time it takes to read that file.

[1]Though not necessarily enough RAM to contain the whole file to be sorted at one time.

2.1 Basic Algorithm

We assume initially that we have one processor to perform the sort and one disk on which both the file to be sorted and the sorted file will be stored. A simple way to sort a (small) file in such an environment is to read the unsorted file into the main memory of the processor, use an internal sorting algorithm (eg. Quicksort) to sort the records of the file, and then write the sorted file out to disk. The total time taken for this sort will equal the time taken to read the entire file once plus the time required for the internal sort plus the time needed to write the entire file back to disk.

We are interested, however, in sorting files which are too large to fit into main memory all at once, and so this simple method will not suffice. One possible way around the limit imposed by the amount of main memory is to read only the sort keys of the records in the file and sort these internally. Assuming that there exists a (secondary) index into the original file based on these keys, the records of the original file could then be read in sorted order and written to another file in the order in which they are read, thus producing a sorted file. There are two serious drawbacks to this method: 1) main memory still imposes a limit on the size of the file which can be sorted and 2) the cost of the re-arrangement phase would be excessive due to the number of disk seeks required to individually read each record of the original file.

2.1.1 Overview of the Basic Algorithm

To completely avoid any limit imposed by the amount of main memory available an algorithm reminiscent of the internal Mergesort algorithm can be used. Mergesort works by dividing a file into smaller parts, recursively sorting each of these parts, and then merging the sorted parts into a single sorted file. The basic external sorting algorithm, which will be expanded upon throughout the remainder of this text, works much the same way. A large file is divided into parts, called *segments*, none of which is larger than the size of available main memory. Each segment is sorted in main memory using an internal sorting algorithm and then stored in a temporary file on disk. Once the entire file has been written out to disk as sorted segments these segments are merged to produce the sorted file. A pictorial representation of this

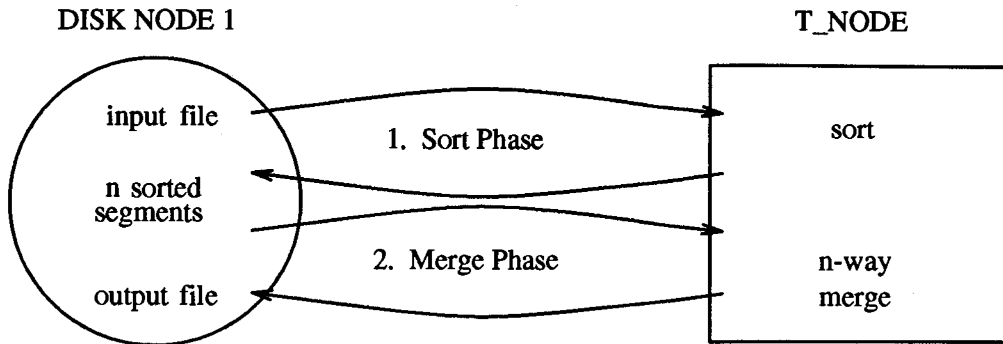


Figure 2.1: Single T-Node Single Disk External Sort

algorithm is given in Figure 2.1.[2] Thus the basic algorithm consists of two phases: an initial phase in which sorted segments are created and a final phase in which sorted segments are merged. The first phase will henceforth be referred to as *Phase I* or the *Sort Phase*; the second phase will be referred to as *Phase II* or the *Merge Phase.*

With this algorithm the main limit imposed by the size of main memory is the maximum size of each of the sorted segments. This, however, does not constrain the size of the file being sorted. Another limit imposed by the size of main memory is the number of segments which can be merged at one time. This limit could constrain the size of the file to be sorted but there is a fairly simple way around this limit which is described in detail in Section 2.1.3.

We now discuss each phase of the basic external sorting algorithm in more detail.

2.1.2 Phase I - Sort Phase

Any internal sorting algorithm can be used to sort an unsorted segment of the original file. Although, on average, Quicksort is faster than Heapsort, we will see when discussing the Replacement Selection algorithm in Section 2.3 that Heapsort provides a number of advantages over Quicksort when used to sort data originating from a disk.

[2]It should be noted that Figure 2.1 presents a logical view of the disk. That is, "DISK NODE 1" actually represents 2 transputers and one disk, as shown in Figure 1.2(i).

2.1.3 Phase II - Merge Phase

Phase II of the basic external sorting algorithm requires some method of performing a *k-way* merge where k will equal the number of sorted segments created during Phase I of the algorithm. k is known as the *order* of the merge. A k-way merge can be performed as follows: A heap data stucture is used and each element of this heap will contain a data record and an indication of from which sorted segment that record was obtained. Every non-empty sorted segment will have exactly one record which belongs to it in the heap. The heap is organized such that the record with smallest key value is at the top of the heap.

The k-way merge begins by reading the first (and thus, smallest) record of each sorted segment and placing it in the heap. Once this heap initialization is completed the top of the heap will contain the smallest record of the original file. Now, the top (smallest) record is repeatedly removed from the heap and a new record is inserted into the heap. This new record is obtained from the sorted segment to which the just removed record belonged, assuming that that segment is not yet empty. If that sorted segment is empty then no new record is inserted into the heap and so the size of the heap shrinks by one. All records removed from the heap are written to a file in the order in which they are removed from the heap. This process is repeated until each of the sorted segments and the heap is empty. At this time the output file will contain a sorted version of the original file.

The k-way merge algorithm as described reads one record at a time from disk. Of course, if the size of a record is smaller than a block, this is not possible as the minimum amount of data obtainable from a disk is a block. In order to avoid reading a block each time a record from that block is required, a *buffer* is allocated in main memory to hold the block. The size of a buffer is usually some integral multiple of the block size. One buffer is needed for each of the segments which are being merged. Thus, the k-way merge algorithm requires enough memory to hold the heap used by the merge and one buffer for each of the segments being merged. If there is not enough memory for the heap and the buffers, we can either make the buffers smaller or the merge can be performed in a number of passes over all the sorted segments. A pass

takes a small number of the original sorted segments and merges them into a larger sorted segment. It does this repeatedly until all of the original sorted segments have been merged into larger sorted segments. Thus, at the end of a pass there is a smaller number of segments and each of these new segments is larger (on average) than those of the previous pass. Passes are performed until only one sorted segment remains - the sorted file.

Thus, the only limit to the size of the file which can be sorted by this basic algorithm is the amount of available disk space.

2.2 Theoretical Performance of the Basic Algorithm

The total time taken for this external sort will equal the time taken to read the entire file twice (one read of the original file and one read of the sorted segments) plus the time taken to write the entire file twice (one write of the sorted segments and one write of the sorted file) plus the time required to create each of the sorted segments plus the time needed to merge all the sorted segments together. Notice that the cost of this algorithm includes two reads and two writes of files equal in size to the original file. This is one more read and write than the simple method originally described required. Thus, we have gained the ability to sort arbirarily large files at the expense of doubling the i/o required.

Let us examine each of these costs in more detail. We assume that the reader is familiar with the components of hard disk drives. There are a number of hard disk drive parameters with which we will be specifically concerned. *Average seek time*, s, is defined as the average amount of time it takes to position the disk drive's arm over the required data. *Effective block transfer time*, ebt, is defined as the time required to transfer a block of data to/from main memory. Effective block transfer time includes the time required for the read/write head to pass over inter-block gaps and block identifiers [Sal88], but is largely independent of the location of the block.

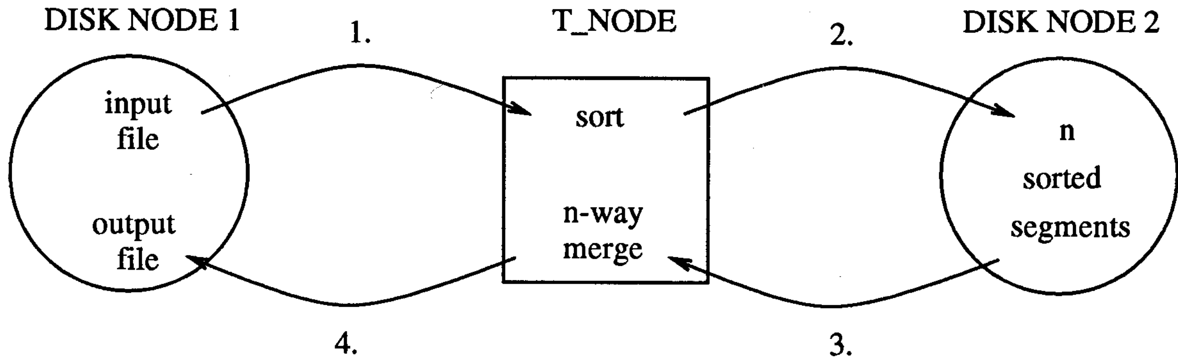


Figure 2.2: Single T-Node Two Disks Configuration

2.2.1 Phase I - Sort Phase

Average seek time, effective block transfer time, and the size of a block are the major factors in determining the cost of i/o. As an initial estimate for the length of time required to read in the unsorted file in the Sort Phase we can use the following formula [Sal88]:

$$(s + ebt) \times b \tag{2.1}$$

where b is the total number of blocks in the file. We can use the same formula to estimate the time required to write all the sorted segments created in the Sort Phase, except in this case b will likely be slightly larger since the last block of each sorted segment is unlikely to be entirely full.

For most current hard disk drives s is significantly larger than ebt. We thus see from formula 2.1 that the seek time has a great impact on the cost of i/o. The seek time can be significantly reduced by using buffers whose size are an integral number (larger than one) of blocks. Since there only one seek would be required for each buffer the total number of seeks would diminish. Note, however, that in our algorithm the file to be sorted is read entirely sequentially from beginning to end and that each sorted segment is written to disk sequentially as it is created. So, if the input file is stored contiguously on disk and each of the sorted segments is also stored contiguously on disk then the only seeking required during the Sort Phase would be between the input file and the sorted segment which is currently being created. This seeking can be completely avoided by using a second disk to store the segments, as shown in

Figure 2.2.[3] In this case only one seek is required to read the input file - the initial seek to the start of the file and so formula 2.1 would be reduced to:

$$s + b \times ebt.$$

For all but the tiniest files the second term of this formula completely dominates the first and so we may eliminate the time for the single seek to arrive at the following estimate for the length of time required to read in the unsorted file when two disks are used:

$$b \times ebt. \quad (2.2)$$

Formula 2.2 may also be used as a rough estimate of the total write time required by the Sort Phase when two disks are used. So, by storing all files contiguously and through the use of an extra disk, we can eliminate virtually all the seeking required by the Sort Phase and thus significantly reduce the cost of i/o for this phase.

The use of double-buffering would allow us to take advantage of available Direct Memory Access to improve the throughput of the Sort Phase. It turns out, however, that optimal use of double buffering in the Sort Phase is somewhat tricky. The Replacement Selection algorithm described in Section 2.3 provides a method for using double-buffering during the Sort Phase of an external sort. We defer the presentation of this algorithm and its complexity to section 2.3.

2.2.2 Phase II - Merge Phase

Let us first briefly discuss the algorithmic complexity of the Merge Phase. Let nsg represents the number of sorted segments created during Phase I. To simpilfy the algorithmic analysis, let us assume that all nsg segments are merged in a single pass. The conclusion we draw holds regardless of the actual number of passes used. The basic data structure used during the Merge Phase is a heap. The total number of elements in the heap at any point during the Merge Phase is nsg.[4] So the cost of a single insertion/deletion operation is $O(lgnsg)$. If there are n records in the file then

[3] Note that, as in Figure 2.1, each "DISK NODE" in Figure 2.2 actually represents a macroscopic view of Figure 1.2(i).

[4] Except at the very start and end of the Merge Phase.

the total cost of the Merge Phase is $O(n \times lgnsg)$. Therefore, since the number of segments grows as the size of the file grows we may conclude that the total cpu cost of the Merge Phase grows faster than the size of the file.

Now let us examine in detail the cost of i/o for the Merge Phase. The 1 and 2 disk formulas for writing in the Merge Phase are identical to the corresponding formulas for writing in the Sort Phase, ie. formulas 2.1 and 2.2. However, reading during the Merge Phase is more problematic. Since only one buffer's worth of a sorted segment is read at a time during a k-way merge as described above we are required to perform a large amount of seeking among the different segments as their corresponding buffers empty. In theory, each sorted segment could be stored on a separate disk. This would eliminate all but the initial seek required to get to the start of the segment on that segment's disk. Unfortunately, the number of disks required would grow with the size of the file to be sorted and it is not reasonable to require such a large and variable number of disks. (We can still, however, use some small fixed number of disks to store the sorted segments and thus allow much of the seeking required in the Merge Phase to occur in parallel.) In short, seeking cannot be dealt with as easily in the Merge Phase as it was in the Sort Phase. It therefore remains an important factor in the cost of input during the Merge Phase.

We now present an analysis of the cost of input in the Merge Phase, based largely on that given in [Sal88]. Let us divide a sorted segment into *pieces* each of which is the size of a buffer in RAM (except, possibly, the last piece of the segment). Let np represent the total number of pieces in all the sorted segments located on one disk. If there is more than one sorted segment on a disk then seeking among the segments on the disk will be required during the merge. We can use the following slight modification of formula 2.1 to get a rough estimate of the time required for one pass of a multi-pass merge to read all the sorted segments on one disk:

$$np \times s + b \times ebt. \tag{2.3}$$

Note, in particular, the significance of the seek time in this formula. We can, however, mitigate its impact by increasing the size of a buffer. Since a piece and a buffer are the same size, increasing the buffer size means decreasing the number of pieces in the

sorted segment. This in turn means a smaller number of seeks required to read all the sorted segments on a disk.

Unfortunately, there is a cost to using arbitrarily large buffers during the Merge Phase: the larger the size of the buffers the smaller the number of them which can be placed in main memory at one time and thus the smaller the order of the merge which is possible. If too large buffers are used this would result in one or more additional passes being required to merge all of the sorted segments into a single sorted file. We are thus faced with the following question: Is it better to perform the Merge Phase in one pass using relatively small buffers or is it better to use multiple passes with relatively large buffers?

We now show how to determine the optimal number of passes to be used for merging the sorted segments in a one-disk-drive sort. Recall that nsg represents the number of sorted segments created during Phase I. Also, let P be the order of the merge to be performed on each pass. We assume that the order of the merge does not change from one pass to another. Thus the number of passes which will be required to merge all the sorted segments into a single sorted file will be

$$\lceil \log_P nsg \rceil .$$

From this formula we can see that, since nsg remains constant for a given file, finding the optimal number of passes for the Merge Phase is equivalent to finding the optimal value for P.

We assume that each of the sorted segments from the Sort Phase is as large as available RAM. During the Merge Phase we divide this same amount of RAM into P buffers, one buffer for each of the segments being merged.[5] Thus, the size of a piece will be $(1/P)$th the size of a segment, ie. there will be P pieces per segment. So

$$np = P \times nsg. \tag{2.4}$$

Substituting this into formula 2.3 we get

$$P \times nsg \times s + b \times ebt. \tag{2.5}$$

[5] In practice, we wouldn't expect to be able to make the size of the buffers precisely $1/P$th the size of RAM. The buffers should be composed of an integral number of disk blocks.

as our estimate for the amount of time required to read all the sorted segments during one pass of the merge. Now, on average, for each input buffer filled there will eventually be one output buffer filled, assuming input and output buffers are the same size. Since a single disk is being used this effectively doubles the total length of time for the i/o in one pass.[6] Thus, the total time spent reading from and writing to disk during the entire Merge Phase will be

$$\lceil \log_P nsg \rceil \times 2 \times [P \times nsg \times s + b \times ebt]. \tag{2.6}$$

We want to find the value of P which minimizes this function. Since

$$\log_P nsg = \ln nsg / \ln P$$

we can substitute into formula 2.6 to obtain

$$\lceil \ln nsg / \ln P \rceil \times 2 \times [P \times nsg \times s + b \times ebt]. \tag{2.7}$$

To simplify further calculations we will ignore the ceiling operation throughout the remainder of the discussion. In addition, note that since the size of the file remains constant the factor $\ln nsg$ also remains constant. Therefore, as we are only looking for the value of P which minimizes formula 2.7, we can eliminate from this formula the factor $\ln nsg \times 2$. We thus arrive at

$$(1/\ln P) \times [P \times nsg \times s + b \times ebt]. \tag{2.8}$$

To find the value of P which minimizes formula 2.8 we differentiate with respect to P and set this derivative equal to 0. We thus obtain

$$\frac{nsg \times s \times \ln P - [P \times nsg \times s + b \times ebt]/P}{(\ln P)^2} = 0.$$

Rearranging, we get

$$P \times (\ln P - 1) = \frac{b \times ebt}{nsg \times s} \tag{2.9}$$

[6]Note, it is important that, when a *single* disk is being used, the output buffers in the Merge Phase be the same size as the input buffers. When using a separate disk for storing the segments then we could make the output buffers smaller and thus, possibly, leaving room for slightly larger input buffers. As a result less seeking would be required which would lead to an improved Merge Phase read time.

Solving equation 2.9 for P and taking its ceiling gives us an approximate value for the optimal order of the merge to be performed by each pass of the Merge Phase.

Given the optimal order for the merge it remains to determine the optimal number of passes to perform during the Merge Phase. Since nsg segments are produced during the Sort Phase the natural choices for the number of passes and corresponding merge orders in the Merge Phase are as follows: one pass which performs an nsg-order merge, two passes each performing $\lceil\sqrt{nsg}\rceil$-order merges, three passes each performing $\lceil\sqrt[3]{nsg}\rceil$-order merges, etc. If one of the merge orders just listed is equal to the optimal merge order then the corresponding number of passes is the optimal number of passes for the Merge Phase. In the more likely occurrence that none of these possible merge orders equals the optimal order then the best choice for the number of passes will be one of the two among those just listed whose corresponding merge order is closest to the optimal value. For example, if the number of segments produced for the file to be sorted is $nsg = 63$ and the optimal merge order for that file is 53 (ie., $P = 53$) then the best choice for the Merge Phase is either one pass which performs a 63-way merge or two passes each consisting of 8-way merges. Which of these two choices is best can be determined by substituting the respective merge orders for P in equation 2.6. For whichever value of P produces the smaller result the corresponding number of passes is the optimal number of passes to perform.

Perhaps suprisingly, it turns out that the optimal order for the merge is independent of the size of the file being sorted. This can be seen as follows: let seg represent the number of blocks in a segment. Then $b = nsg \times seg$ or $seg = b/nsg$. Therefore, equation 2.9 can be re-written as

$$P \times (\ln P - 1) = \frac{seg \times ebt}{s}. \tag{2.10}$$

As each of seg, ebt, and s are independent of the size of the input file so, therefore, is P. However, the independence of optimal merge order from the size of the input file is somewhat misleading as it does not mean that the optimal number of passes is independent of the the size of the input file. This can be seen from the way in which the optimal number of passes is calculated, as discussed in the previous paragraph. nsg increases in direct proportion to the size of the input file while the optimal merge

order remains fixed, so the order, n, of the root required to bring the value of $\sqrt[n]{nsg}$ close to the optimal merge order will increase. From the discussion in the previous paragraph we can see that the optimal number of passes is the same as n.

Equation 2.9 can still be used to determine the optimal number of passes even when two disks are used during the Merge Phase (one for the input and one for the output). This is because the factor of 2 which was introduced to account for the write time, and which would be eliminated when two disks are used, was eliminated anyway during the transformations used to arrive at equation 2.9.

Another way mentioned in [Sal88] to reduce the amount of seeking required during the Merge Phase is to increase the size of main memory. More RAM allows bigger buffers which results in a reduction of the total number of pieces in the sorted segments and thus also less seeking (by equation 2.3). In addition, increasing the size of main memory increases the size of the segments which can be produced during the Sort Phase. For a given file size larger sorted segments means less sorted segments. This in turn means more space available for buffers during the Merge Phase. Again, this allows bigger buffers and the concomitant reduction in the amount of seeking required. Thus, the amount of main memory available has a substantial impact on the time required by the Merge Phase.

Since double-buffering is so successful in speeding up the Sort Phase we could use this same technique in an attempt to speed up the Merge Phase also. However, unlike in the Sort Phase, we need a substantial number of extra buffers to use double-buffering in the Merge Phase - one extra buffer for each of the segments being merged. To see why, let us assume that each sorted segment has records whose keys are uniformly distributed among the entire range of keys in the original file. Thus, if one piece of each segment is read into memory and then the merging begun, all of the pieces in memory will empty at about the same time. At such a time, the processor needs access to the next piece for **each** segment in order to continue the merge processing. If input for each segment is being double-buffered then this next piece will quite likely already be present in main memory.

Even though all the pieces for the sorted segments will empty at approximately the same time, no two pieces will empty at **exactly** the same time. The next piece

that should be read is from the segment whose piece currently in memory will be the first to be emptied. With a little extra effort we can read in pieces of segments in the exact order in which they will be needed. This order can be ascertained by examining the key of the last record of each piece currently in memory. The piece with the smallest such key will be the first to empty and the corresponding segment should be the next to have a piece read from it[Sed90]. By maintaining each of these keys and pointers to their corresponding sorted segments in a heap, we can assure that all of the extra buffers in memory will be filled with pieces in the order in which these pieces will be needed soonest.

Unfortunately, there is a drawback to the use of double-buffering during the Merge Phase. Since we have a fixed amount of memory, doubling the number of buffers will necessitate cutting the size of each buffer in half. As discussed previously, reducing the size of the buffers results in an increase in the number of disk seeks required, and disk seeks are very costly.

Whether or not to use double-buffering in the Merge Phase depends upon the relative speeds of the cpu and the i/o subsystem. Obviously, if the increase in the cost of reading due to cutting the size of the buffers in half is larger than the total time spent by the cpu in merging the segments then double-buffering should not be used. More specifically, let the time spent by the cpu on performing the merge be C, the time required for reading all the segments during the merge without double-buffering be R, and the extra time incurred by the extra seeks required for double-buffering be r. Then the time spent in the Merge Phase when double-buffering is *not* used will be $R + C$. If we assume that DMA allows i/o to be performed perfectly in parallel with cpu merging and that $R > C$ (ie. an i/o bound merge), then the time spent in the Merge Phase when double-buffering *is* used will be $R + r$. Thus, double-buffering is only worthwhile if $r < C$. This is easily seen from Figure 2.3 which illustrates a case in which the use of double-buffering would pay off.

We can derive, as we did in section 2.2.2 for the case in which double-buffering is not used in the Merge Phase, formalae which can be used to calculate how long the Merge Phase will take and what is the optimal number of passes to perform during this phase when double-buffering *is* used. With the use of double-buffering the total

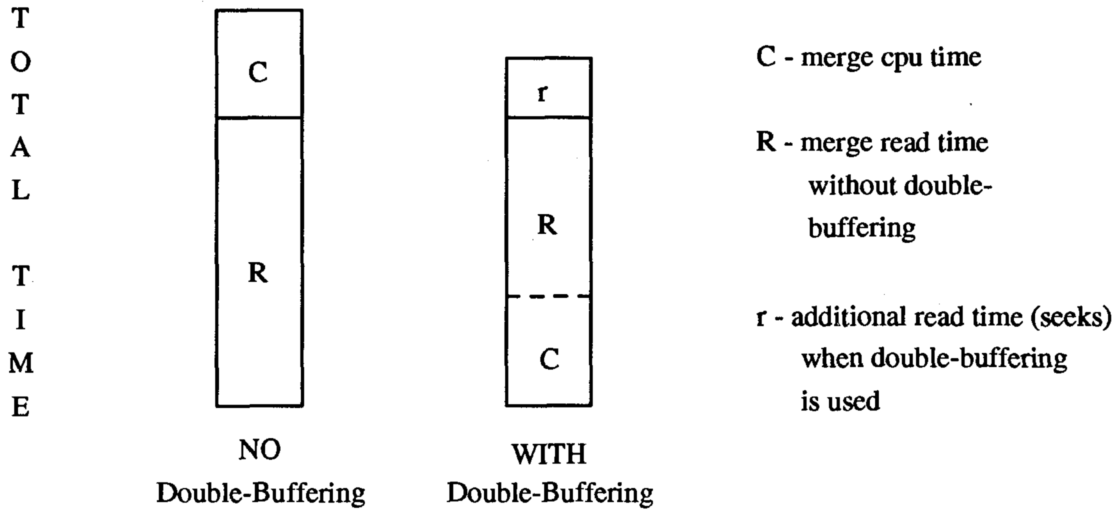


Figure 2.3: Effect of Double-Buffering in Merge Phase

number of pieces to be read in the Merge Phase doubles and so equation 2.4 becomes

$$np = 2 \times P \times nsg. \tag{2.11}$$

Thus, the new estimate for the total time to read all the segments in one pass of the Merge Phase becomes

$$2 \times P \times nsg \times s + b \times ebt. \tag{2.12}$$

This reflects the increased cost incurred by the extra seeking required with the use of double-buffering. Thus, the new equation to be used to predict the total time spent on i/o during the Merge Phase becomes

$$\lceil \log_P nsg \rceil \times 2 \times [2 \times P \times nsg \times s + b \times ebt] \tag{2.13}$$

If this change is filtered through the discussion in section 2.2.2 used to arrive at equation 2.9 we find that the new equation to be used to determine the optimal number of passes becomes

$$P \times (\ln P - 1) = \frac{b \times ebt}{2 \times nsg \times s}. \tag{2.14}$$

Comparing equation 2.14 with equation 2.9 it can be seen that the use of double-buffering in the Merge Phase will result in a smaller optimal order for the merge.

This is because, for a file of a given size, the use of double-buffering does not change any of the values to substitute for the variables in equation 2.14 from those used in equation 2.9, yet the right hand side of equation 2.14 has a factor of two in the denominator whereas equation 2.9 does not. This raises the possibility of requiring an increase in the number of passes needed to achieve an optimal Merge Phase time. The reason for this can be seen from the discussion following equation 2.9 where now the optimal merge order will be smaller while the total number of segments (nsg) will remain unchanged. Thus, we see another way in which the use of double-buffering in the Merge Phase can, possibly, increase the Merge Phase time. Note, however, that formula 2.13 and equation 2.14 cannot, by themselves, allow us to determine whether double-buffering will improve the *overall* time for the Merge Phase. To determine this we must also know how much time the cpu will spend actually performing the merge and how much of this merging can be overlapped with the reading of pieces of sorted segments.

Finally, we mention the possibility of using data compression techniques in speeding up the Merge Phase input. If the speed of the processor is so great relative to the i/o sub-system that it can merge segments far faster than these segments can be read from storage, then it might be useful to have the processor compress these segments during the Sort Phase and decompress them during the Merge Phase. This would decrease the demands made on the i/o subsystem while increasing that made on the processor, thus making the two somewhat more balanced.

2.3 Replacement Selection Sort Algorithm

We now present another method for creating sorted segments during Phase I of the external sort. Earlier, we discussed a straightforward way of creating sorted segments. This simple method required filling main memory with records from the input file, sorting these records using some standard internal sort, and then storing the sorted records on disk as a sorted segment. The method we present now is known as *Replacement Selection* [Knu73] and it has a number of significant advantages over this straightforward method.

The first advantage is that Replacement Selection allows input **and** output to occur in parallel with the actual creation of a sorted segment. In contrast, the simple method allows only parallel input (through the use of double-buffering). Output cannot be performed in the simple method until the entire segment has been sorted because it is not known until all the records for a particular segment have been read in which of these records should be the first to be written out. Replacement Selection allows writing of a sorted segment to begin immediately after one buffer's worth of data for that segment has been read. Thus, Replacement Selection increases the efficiency of Phase I of the external sort.

A second advantage of Replacement Selection is that, as we will see shortly, it allows the creation of sorted segments which are larger than the size of main memory. As discussed in the previous section, increasing the size of the sorted segments decreases the number of sorted segments which in turn allows for quicker merging. (Recall that the fewer the number of segments being merged, the larger the buffers which can be used during the merge.) Thus, the use of Replacement Selection in Phase I also results in an increase in the efficiency of Phase II of the external sort.

We now give a brief summary of the Replacement Selection algorithm. The basic idea is to use Heapsort to create the sorted segments. What sets Replacement Selection apart from Heapsort is the use of **two** heaps at once. Initially, both heaps are empty. We can begin by filling main memory with records and inserting all these records into one heap. We can then proceed by alternately removing one record from the heap and placing it in an output buffer and then replacing this record in the heap with a record from the input buffer. In this way each of getting input from the original file, processing records in the heap, and outputting sorted records to disk can proceed in parallel.

However, for the sorted segment which is currently being created, care must be taken not to output a record whose key has a value which is smaller than the value of the key of the most recently written record. This is avoided through the use of the second heap. Whenever we are about to replace an old record in the heap with a new one we check that the value of the new key is greater than or equal to the value of key of the record just written out. If so, then the new record is inserted into the heap. If

not, then the new record is inserted into the second (backup) heap. If the records of the input file are in random order then the first heap will gradually get smaller and smaller and the second heap will become correspondingly larger and larger. When one heap shrinks to size zero, the roles of the heaps are reversed and a new sorted segment is begun. The total size of the two heaps will equal the size of available main memory throughout most of the Sort Phase, except at the beginning and ending of the phase.[7]

We can now see a third advantage of Replacement Selection: when the input file is already sorted the Sort Phase will produce the sorted file itself and not just some number of sorted segments. This is because, when the input file is already sorted, no new record will ever have a key whose value is less than that of the record it is replacing in a heap. Therefore, no record will ever need to be placed in the backup heap, thus negating any need to start a new sorted segment. So, records from the input file will just stream from the file through the heap and out to a single sorted segment. In fact, the input file need not be entirely sorted in order for the Sort Phase to be able to produce a sorted file. If we let ps be the position[8] of a given record in the sorted version of the input file and tr be the number of records which can fit inside main memory at one time then as long as the position of any record in the input file is less than $ps + tr$ the Sort Phase will produce only one segment (ie., the sorted file). Finally, note that when the use of Replacement Selection in the Sort Phase is combined with the use of two disks (one for input and one for output), the best possible time for the external sort is just the time needed to read the input file.

Thus, the best case behaviour of Replacement Selection is very good indeed. But what of the worst case? If the input file is sorted in the reverse order of that required it might at first seem that every new record will be required to put into the backup heap, since every new record will have a key whose value is smaller than that of the previous record. However, Replacement Selection begins by filling main memory with a heap of records. Only then does output begin. So, an entire main memory's worth

[7] Creation of the first and final sorted segments is a little tricky. The interested reader is referred to [Baa88] for more details.

[8] In terms of number of (equal-sized) records, not in terms of number of bytes.

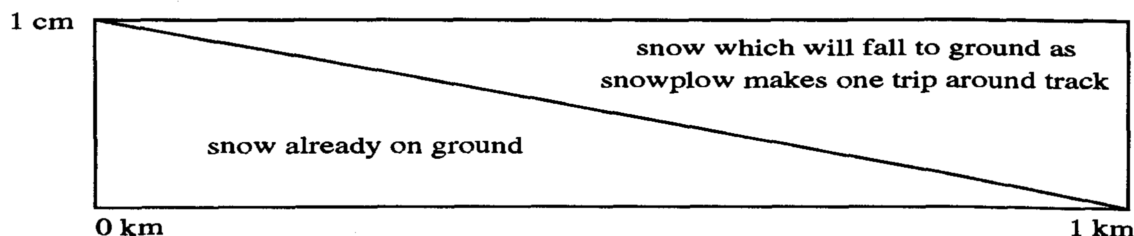


Figure 2.4: Snowplow Analogy

of records can be output before needing to switch to the backup heap and a new sorted segment. Meanwhile, the size of the backup heap will have grown to equal that of main memory. Thus, in the worst case, each sorted segment will be as large as main memory (except, possibly, the last one.) So, we are no worse off using Replacement Selection in the Sort Phase than we are when not using Replacement Selection.

What is the average size of the segments created by Replacement Selection? Perhaps surprisingly, the expected length of the sorted segments is *twice* the size of main memory. Knuth [Knu73] presents the following physical analogy (which he attibutes to E.F. Moore) to give an intuitive explanation for this expected segment length. Suppose there is a snowplow moving around a circular track of circumference 1km. Also, suppose that snow is continuously falling as the snowplow makes its way around the track. If we assume that the plow is removing snow from the track at the same rate at which new snow is falling from the sky, then on each trip around the track the plow will remove twice as much snow as there exists on the track at any instant. This can be seen with the aid of Figure 2.4. If we imagine that the plow is continuously moving from left to right across this figure then the triangle beneath the diagonal line represents snow already on the track and the triangle above the diagonal line represents the snow which will fall onto the track as the plow moves around it. Since the two triangles are of equal area the total snow that the plow removes is twice the amount which is on the track at the start of the plow's trip. In much the same way, data enters and leaves a heap in main memory and the total amount of data which passes through the heap before a new segment is begun is twice the amount which exists in the heap at any instant.

Since Replacement Selection allows sorting and i/o to proceed in parallel then for

an i/o-bound sort the cpu cost of actually creating the sorted segment is subsumed by that of the i/o. However, let us briefly discuss the algorithmic complexity of Replacement Selection. The basic data structure used by this algorithm is a heap. If the average number of records in this heap is y then the average cost of a single insertion/deletion operation on this heap is $O(lgy)$. Therefore, the cost of building one average-sized segment is $O(y \times lgy)$ and, since there are nsg segments in all, the total cost of the Sort Phase is $O(nsg \times y \times lgy)$. However, for a given amount of main memory the cost of building an average sized segment remains constant regardless of the size of the file being sorted. Therefore, the actual cost of the entire Sort Phase is $O(nsg)$. That is, the cost of the Sort Phase is linear with respect to the size of the input file.

Let us examine the effects of the use of Replacement Selection during the Sort Phase on the estimates given in section 2.2.2 for the i/o times of the Merge Phase. Initially, we will assume that double-buffering is *not* used during the Merge Phase. In this case, for a P-way merge each piece of a segment is $1/P$th the size of main memory. Therefore, since the use of Replacement Selection results in segments whose size average twice that of main memory, the number of pieces per segment during a P-way merge is $2 \times P$. So, the equation for the total number of pieces in all the sorted segments becomes

$$np = 2 \times P \times nsg. \tag{2.15}$$

This is identical to equation 2.11 which was obtained when double-buffering is used and Replacement Selection is not. Therefore, the formula to use to estimate the input time for one pass of the Merge Phase would be the same as formula 2.12, namely

$$2 \times P \times nsg \times s + b \times ebt \tag{2.16}$$

Likewise, equation 2.15 would lead to the same equation as 2.14 to use in the determination of the optimal number of passes for the Merge Phase, namely

$$P \times (\ln P - 1) = \frac{b \times ebt}{2 \times nsg \times s}. \tag{2.17}$$

So, using either Replacement Selection in the Sort Phase of double-buffering in the Merge Phase (but not both) produces identical equations for predicting the i/o cost of

the Merge Phase. However, for a file of a given size, the equations for the respective methods will not produce identical results. In particular, the value to substitute for nsg in the equations for the Replacement Selection case will, on average, be half that for the double-buffering case.

Comparing formula 2.16 with formula 2.5 which is used when neither Replacement Selection nor double-buffering are employed, it might at first glance seem that Replacement Selection will double the input time for one pass of the Merge Phase. However, again for a file of a given size, not only will the value to substitute for nsg in 2.16 be, on average, half that to substitute in 2.5, so too will the value to substitute for P be halved, on average. This is obvious since cutting the number of segments in half also cuts in half the order of the merge required to merge these segments. So, formula 2.16 will reflect the fact that Replacement Selection improves the input time for one pass for the Merge Phase.

Perhaps surprisingly, the optimal value obtained for P does not change with the addition of Replacement Selection if we assume that it produces segments which are indeed twice as large as they otherwise would be. If we compare equation 2.17 with equation 2.9 we see that the only difference between the two is the factor of 2 in the denominator of the right hand side of equation 2.17. However, with segments twice as large the number of segments (nsg) will be cut in half and the influence of the factor of 2 will be completely negated. This is not to say, however, that Replacement Selection has no effect on the optimal number of passes to perform during the Merge Phase. We have already seen that Replacement Selection can reduce this optimal number to zero!

If both Replacement Selection is used in the Sort Phase and double-buffering is used in the Merge Phase then the equation for the total number of pieces in all the sorted segments becomes

$$np = 4 \times P \times nsg \qquad (2.18)$$

since double-buffering cuts the size of each piece in half and so doubles the total number of pieces. If we begin the discussion in section 2.2.2 with this equation we find that the formula to use to estimate the input time for one pass of the Merge

Phase using a single disk is

$$4 \times P \times nsg \times s + b \times ebt \tag{2.19}$$

and the new equation to be used to determine the optimal number of passes to be performed during the Merge Phase becomes

$$P \times (\ln P - 1) = \frac{b \times ebt}{4 \times nsg \times s}. \tag{2.20}$$

Just as when double-buffering was used without Replacement Selection, however, these equations by themselves are not enough to determine whether the use of double-buffering in the Merge Phase will improve the overall time for the Merge Phase.

Chapter 3

TASS "File System"

The primary purpose of TASS [Dud92] is to function as a disk cache for SCSI disks attached to a transputer network. It is essentially a block server and as such it provides its clients only a low level (albeit fast) interface to the disks themselves. The routines it supplies allow a client transputer node to read and write part of or an entire TASS chunk.[1] TASS does not provide a file system and, consequently, none of the standard Unix/C file system calls. Thus, to facilitate the implementation of the programs written for this study, rudimentary file system functions at a level higher than TASS chunk storage were created.

Five file system functions were fashioned after the Unix/C file system calls fopen(), fread(), fwrite(), fseek(), and ftell(). The usage of each of the TASS versions of these calls (except fopen) in terms of parameters supplied is identical to the usage of the Unix/C versions. Their functionality (again, excepting fopen) is equivalent to their Unix/C counterparts. The implementation of all of these functions, however, is markedly different from the corresponding Unix/C versions. These calls do not, for example, access any directory tree on any of the disks. Indeed, there is still no actual file system present on the disks. This means that there is no read or write protection of the data in any of the "files" created by these TASS file functions between runs of any client program. Protection during a program run is gauranteed as a consequence of the single-user nature of the transputer network on which the programs are run.

[1] As previously mentioned, a chunk is a virtual disk block of size fixed by TASS.

As in Unix, the first function that must be called before any reading or writing can be done to a TASS file is fopen(). Since there is no directory of files for this function to access, the user must specify with this function a location on the disk (ie. chunk number) at which to begin the file. A TASS file is composed of strictly contiguous TASS chunks so the user is responsible for ensuring that enough room exists on the disk for the maximum size of the file. Also, if more than one file is to be opened during a program run, the user must ensure that no file will overrun any other. These are the ways in which usage of the TASS fopen() function differs from that of its Unix counterpart. The only function of the TASS version of fopen() is to initialize the data structure (of type *FILE, as in the Unix version) which will be used by the remaining TASS file system calls.

The TASS versions of fread(), fwrite(), fseek() and ftell() are each used in exactly the same way as their Unix counterparts, including the type and order of their parameters. The increases in functionality provided by these functions over the unadorned TASS "chunk" functions are the ability to read and write any amount of data (in particular, more than one chunk's worth) and, on sequential accesses, not having to explicity keep track of where the next piece of data is to come from or go to.

The Unix versions of fread() and fwrite() must buffer their accesses to disk as a disk can only access data in discrete page sizes (eg. 512 bytes or 8 Kbytes) whereas a user is free to access an arbitrary number of bytes in a file. In addition, buffering allows these functions to minimize the amount of actual disk accesses they require since reads and writes of a part of a buffer's worth of data can be done entirely within main memory[Mor88]. All accesses the TASS versions of fread() and fwrite() make of the disk, however, will be through the TASS cache. The cache node must read data from the disk in chunk sizes, but once the data is in the cache the client node is able to request any size of data it desires through the transputer link which connects it to the cache node. Therefore, the necessity of buffering is removed from the TASS versions of fread() and fwrite().

Should buffering be used in the TASS versions of fread() and fwrite() even though it's not required? The use of buffering was decided against for the following reasons: Firstly, these programs do their own buffering with buffers the sizes of which change

according to the size of the file being sorted. Even during a single execution of an external sort the size of the buffers used by the program changes from the Sort Phase to the Merge Phase. Since fread() and fwrite() were intended to be library routines whose usage was identical to their Unix counterparts, the size of the buffers these routines used, if any, could not be changed dynamically. Secondly, reading and writing is the bottleneck in external sorting so avoiding the extra memory-to-memory copy incurred by buffering was deemed advantageous.

In an effort to improve the input rate for sequential reads of a file Unix will employ pre-fetching when it detects that such reading is taking place[MK91]. That is, when a user is reading a file sequentially an explicit request for data from one block in the file will, in turn, cause Unix to read the next block of the file into another system buffer. This way, when the user requests the next block in the file that block will already be in or on its way to main memory. TASS also allows a client to invoke pre-fetching in an attempt to speed up the input rate. However, TASS pre-fetches the data from the disk into the Cache node, not into the client node. If the client node wants to pre-fetch data into its local memory it must do so itself. Such automatic pre-fetching could be built into the TASS version of fread() but was not for reasons analogous to those given for not using buffering. In particular, the programs already perform double-buffering, obviating the need for additional pre-fetching.

One technicality remains regarding pre-fetching. In TASS, the client must not only explicitly request that pre-fetching occur, but must also specify relative to the chunk being read the number of the chunk to pre-fetch. According to [Dud92] the best performance on sequential reads is obtained when pre-fetching the fifth chunk following the chunk currently being read. Unfortunately, pre-fetching this number of chunks ahead leads to sub-optimal performance for the reads which occur in the Merge phase of the external sort (we discuss this in more detail in the next chapter). Therefore, a second type of fread(), called fread_npr(), was created for the TASS files system calls. The only difference between fread_npr() and fread() is the addition of a single parameter to fread_npr() which allows the user to specify how many chunks ahead of the chunk being read should the chunk to be pre-fetched into the Cache node occur. The TASS version of fread() itself automatically pre-fetches the chunk which

results in optimal performance of sequential reading.

Almost all calls to the TASS CacheServer are hidden within these TASS file system functions. The only original TASS function which still needs to be called by the program is tass_shutdown(). This function must be called after all usage of the cache by the program is finished in order to provide the cache the opportunity to shut down gracefully. It wasn't placed in an "fclose()" function since the only purpose of this function would have been to call the tass_shutdown() procedure. This procedure can only be called once per program run, unlike fclose() which should be, but is not necessarily(!), called once per open file.

Chapter 4

External Sorting With a Single Transputer

In this chapter we describe an implemenation of the external sort discussed in Chapter 2. We also present and analyze the execution results for various disk and storage node configurations.

4.1 Preliminary Remarks

Replacement Selection is used in both the non-double-buffered and double-buffered implementations of the external sort because this method creates, on average, segments which are twice as large as main memory. That is, whether double-buffering is used or not, Replacement Selection helps improve the expected time for the Merge Phase without any detrimental effects on the times for the Sort Phase.

The size of the i/o buffers used during the Sort Phase is smaller than that of the i/o buffers used during the Merge Phase. If only a single disk is being used then using small buffers during the Sort Phase will result in an increased number of seeks during that phase, substantially reducing i/o performance. However, if bigger buffers are used during the Sort Phase then less space is available in RAM for the segment which is being created. This will result in an increase in the number of segments which will, in turn, slow down the input rate for the Merge Phase. Thus, there is a tradeoff

between the best speed achievable for the Sort Phase and that achievable for the Merge Phase. However, this tradeoff disappears when more than one disk is available for the sort as then virtually all seeking during the Sort Phase can be eliminated. Therefore, we won't concern ourselves further with the preceeding tradeoff.

We now explain how the initial sizes of files and the records to sort were chosen. Recall that the size of each disk used in the experiments is 110MB. Therefore, because an additional amount of space slightly larger than the input file is needed to store the sorted segments created during the Sort Phase, the largest file which could be sorted using one disk is less than 55MB.[1] To help see the effect that increasing the file size has on the times for the sort we will also, initially, report the times for a much smaller file. As previously mentioned the expected size of each sorted segment is 3.6MB. Thus, a randomly ordered 5MB file (1/10th the size of the large file) is close to the smallest such file which we would expect not to be able to sort entirely into one segment during the Sort Phase of the external sort. So, 5MB and 50MB files were initially used as input to the sort. Later, when we start using more than one disk, we will be able to sort 100MB files.

All the records in one file are of uniform size. However, an interesting dynamic occurs when the size of the record is changed while keeping the overall size of the file constant. We expect the length of time to read and write a file to remain unchanged for a given file size. However, the amount of cpu processing required ought to increase as the size of the records decreases since, for a constant file size, the number of records increases as the record size decreases. Thus, we can alter the proportions of the overall sort time spent by the cpu and by the i/o subsystem by changing the size of record while keeping the size of the file constant. Put another way, we can change the "boundedness" of the sort from cpu-bound to i/o-bound by altering the record size. So, initially, for each size of file two different sizes of records were used: 8 bytes and 128 bytes.[2] [A+85] put forth the sorting of 100MB files of 100-byte records as a

[1] We assume here that we are allowed to overwrite the original file with the sorted file.

[2] Record sizes were also chosen so that an integral number of them fit precisely into a chunk. Using arbitrary record sizes would have necessitated either splitting a record across chunk boundaries or leaving each chunk partially empty. The initial version of TASS did not support partial reads or writes of chunks making splitting records across chunk boundaries problematic. Leaving each chunk

benchmark for external sorting. So the data for the 50MB file of 128-byte records are the data in which we will be most interested initially.

To produce the data in all[3] the tables ten randomly ordered files for each combination of file and record size were produced. These files were sorted one at at time and the average of the times for the ten files of a given size and record length was calculated. It should be noted that the times for sorting a given file differed very little from one run to the next. For example, the standard deviation of the total sort times for a given 50MB file composed of 128-byte records when a single disk was used without double-buffering was typically 0.04 seconds. This is because, as mentioned previously, we are not running our programs in a multi-user environment and so there are no extraneous factors which can affect the execution time of the program. There were larger differences among the total sort times for each of the 10 different files used in the experiments. For example, for the same implementation just mentioned, the standard deviation of the total sort times for the ten different 50MB files of 128-byte records was 1.5 seconds.

The times reported for input and output in all of the tables are *virtual* times in that they represent the time taken for these operations to complete *as viewed from the perspective of the sort/merge process.* That is, a clock reading was acquired just before and just after each i/o request by the sort/merge process and the sum of the differences of each pair of these readings was taken as the total i/o time (one sum for input and one for output). Because all i/o requests go through a disk cache (TASS) and not straight to a raw disk, the virtual i/o times will almost always be smaller than the corresponding *real* disk i/o times, where real disk i/o time is defined as the time needed to transfer data from a (non-cached) disk to the user process' RAM. Also, the i/o rates reported are virtual in that they are based on virtual times.

One way in which a TASS CacheServer can be used to make the virtual input

partially empty is feasible but would decrease the efficiency of the i/o, the extent of this decrease depending upon how much space is left empty in each chunk. However, since chunks are large and records are small we would expect less than 1% of a chunk to be left empty on average. Therefore, though we avoided these problems by carefully choosing record sizes, the data we obtain shouldn't differ by more than 1% from what would have been obtained had a more complicated solution been implemented.

[3]Except where noted.

rate greater than the real disk input rate is through the pre-fetching feature that TASS provides. If a client is to process a number of chunks from disk and the client knows the order in which it will process these chunks, it can ask the CacheServer for the chunk it needs first and then request the CacheServer to pre-fetch the chunk(s) needed next. Thus, while the client is processing the first chunk in its main memory, the CacheServer can be pre-fetching the next chunk into its (the CacheServer's) main memory. Then, when the client is finished processing its chunk and asks for the CacheServer for the chunk it requires next, this chunk will already be on its way to the CacheServer's memory, it it isn't already there. Indeed, if the client takes so long to process each chunk that the next chunk is always already in the CacheServer's memory then the virtual input rate can be as high as link speed. We will see examples of such supra-disk input rates in section 4.5.

The use of the CacheServer has an even bigger impact on the virtual output rates. Almost every write appears to the client to occur at link speed since the data being output only needs to be transferred from the client's RAM to the CacheServer's RAM. Once this transfer is complete the client continues its processing and leaves the CacheServer with the responsibility of actually sending the output to disk. There are no complications regarding pre-fetching as is the case for input. We will see, in the tables for the non-double-buffered versions of the sort, virtual output rates which are almost always just under link speed.

If, in addition, the sort uses double-buffering then the virtual input and output rates will be substantially higher since then the sort process has worker processes performing i/o for it in parallel with the sort process itself. The virtual i/o rates can become so high, in fact, that they become somewhat meaningless.

So, we provide the individual i/o throughput rates for each phase of the sort in the tables for the non-double-buffered versions of the sort and these figures will reflect the benefits of using the cache. However, once double-buffering is added, extremely large (ie., meaningless) virtual i/o rates are produced and so these are not reported. Only the total throughput figures for each phase of the sort and for the overall sort will be reported in the tables based on the double-buffered implementations.

When accessing a disk through TASS one must be careful to use the proper amount

of pre-fetching. For strictly sequential reading optimal TASS performance is achieved by, after having read a chunk, requesting that TASS pre-fetch the fifth chunk following that chunk in the file [Dud92]. However, when random chunks are being read from the disk, pre-fetching 5 chunks ahead can result in chunks being read into the Cache Node and then being flushed before any of the data in those chunks is actually needed by the client. For now we state without further explanation that we have found that the Merge Phase achieves optimal performance for random reads by pre-fetching 2 chunks ahead when sorting files of size 50MB and by pre-fetching 1 chunk ahead when sorting 100MB files. We will discuss this in more detail in section 4.5.1. Since the Sort Phase performs only sequential reading, 5 chunk pre-fetch is used during that phase, irrespective of the size of the file.

The numbers of most interest in all the tables which follow are the throughput rates in Kb/sec for both phases of the sort and for the total sort. The best we can hope to achieve for each phase using a single disk is 730 Kb/sec - the maximum sequential read rate from the disk. The best overall throughput we can achieve is half this rate since the whole sort must read the entire file twice.

4.2 Single Node; Single Disk Configuration: Non-Double-Buffered

The initial implementation of the external sort uses a single cpu and a single disk. It does not make use of double-buffering in either the Sort Phase or the Merge Phase. This simplest possible implementation forms the basis against which the effects of all further enhancements can be measured.

4.2.1 Analysis

Let us begin by examining equation 2.17 from section 2.3 which can be used in determining the optimal number of passes to be performed in the Merge Phase within our hardware environment when Replacement Selection is used during the Sort Phase but

double-buffering is not used during the Merge Phase. We first determine the appropriate values to use for ebt and s. The optimal sequential read rate for a storage node containing a single disk is 730 Kb/sec. Since the size of a block (ie., chunk) as defined by TASS is 16896 bytes the effective block transfer rate is $ebt = 0.0231$ seconds. The estimate we use for the seek time is $s = 0.0589$ seconds which was obtained as follows: Subtract the time taken to read 730Kb of data from a TASS CacheServer sequentially (1.00 second) from the time taken to read the same amount of data randomly (3.54 seconds) and divide this number by the number of chunks in 730Kb (43.21). For the random reads one chunk is read for every seek and so there would be approximately 43.21 seeks. Thus, the time per seek is taken to be (3.54 - 1.00) seconds / 43.21 seeks = 0.0589 seconds/seek.

Each of our transputer nodes has 2MB of main memory, but some of this memory is needed for the program object code and for run-time data. Therefore, the size of main memory usable for the creation of sorted segments is about 1.8MB and the expected size of these segments is 3.6MB. So, for a 5MB file our estimate for the number of segments is $nsg = 1.39$.[4] Because there are 16896 bytes per block, for a 5MB file $b = 296$. Substituting these values into equation 2.17 and then solving for P we obtain $P = 21$. Recall that the optimal order for a merge is independent of the size of the file being sorted. So, we estimate that 21-way merges are optimal in our hardware environment when a single disk is used during the sort, no double-buffering takes place, and Replacement Selection is used in the Sort Phase.

Using this optimal merge order we now calculate the optimal number of passes to perform during the Merge Phase as discussed in 2.2.2. As the number of segments produced from a 5MB file (integral value of 2) is substantially less than the optimal merge order of 21 we expect that the optimal Merge Phase time for 5MB files is achieved with a single pass 2-way merge. The expected number of segments produced from a 50MB file is approximately 14 segments which is likewise less than 21 and so we expect that merging the segments for a 50MB file is optimally performed with a

[4]In the analysis of section 2.2.2 we assumed that all segments were of equal size, which in practice they are not. So, while the actual number of segments will always be an integer, we will substitute non-integer values for nsg into the equations presented in Chapter 2 in an effort to obtain more accurate estimates.

single pass 14-way merge.

Finally, let us use formula 2.16 to estimate the total amount of time required for input during the Merge Phase. For a 5MB file, we use $P = 2$, $b = 296$, and $nsg = 1.39$ in formula 2.6; for a 50MB file, we use $P = 14$, $b = 2960$, and $nsg = 13.9$. The values used for s and ebt are those just discussed above. Thus, we estimate that input in the Merge Phase will take 7.2 seconds for a 5MB file and 91.3 seconds for a 50MB file. Recall that when a single disk is used the time required for output during the Merge Phase is the same as that required for input.

4.2.2 Results

In Tables 4.1 and 4.2 we present the execution times and the throughput rates for the external sort using a single T800 transputer and a single disk. Table 4.1 contains data for files of size 5MB with 8 and 128-byte records; Table 4.2 contains the corresponding data for 50MB files together with data for 1056-byte records.

The first thing we point out is how "amazingly" accurate (at least for the 8-byte records) the estimate is that we calculated above for the Merge Phase input time for the 5 MByte file: the actual time given in Table 4.1 is 7.1 seconds whereas the estimate is 7.2 seconds. The match is not as good when the record size is 128 bytes because the actual time has now increased to 11.1 seconds. Unfortunately, the estimated value for the 50 MByte file is not nearly as good no matter what the size of the records being sorted. For example, the actual time given in Table 4.2 for the 8-byte records is 200 seconds whereas the estimated input time is 91.3 seconds.

Why does the accuracy of the estimate decrease so dramatically as the size of the file increases? The apparent accuracy of the Merge Phase input time estimate for the 5MB file with 8-byte records is somewhat of an illusion. This can be seen as follows: notice that the Sort Phase input time for the 8-byte records in Table 4.1 (9.1 seconds) is larger than the Merge Phase input time (7.1 seconds). This is surprising because input during the Sort Phase requires no seeking, whereas seeking does occur during the input of the Merge Phase. This discrepancy between expected and actual behaviour is a result of the cache in the TASS CacheServer transputer node through

Table 4.1: Sort Times and Throughput Rates for a Single T-Node with a Single Disk - NO double-buffering

File Size	5 MBytes			
Record Size	8 Bytes		128 Bytes	
	secs	Kb/sec	secs	Kb/sec
Sort Phase	**182.4**	**27.4**	**26.7**	**187**
cpu	163.5	30.6	10.8	463
input	9.1	549	9.4	532
output	9.7	515	6.5	769
Merge Phase	**47.5**	**105**	**19.4**	**258**
cpu	36.1	139	3.9	1282
input	7.1	704	11.1	450
output	4.3	1163	4.4	1136
Total	**229.9**	**21.7**	**46.1**	**108**
Total cpu	199.6	25.1	14.7	340
Total input	16.2		20.5	
Total output	14.0		10.9	
Total i/o	30.2		31.4	

Table 4.2: Sort Times and Throughput Rates for a Single T-Node with a Single Disk - NO double-buffering

File Size	50 MBytes					
Record Size	8 Bytes		128 Bytes		1056 Bytes	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**1631**	**30.7**	**265**	**189**	**249**	**201**
cpu	1446	34.6	84	595	10	5000
input	88	568	117	429	175	286
output	97	515	64	777	64	781
Merge Phase	**775**	**64.5**	**290**	**173**	**278**	**180**
cpu	531	94.2	67	752	35	1429
input	200	250	180	278	199	251
output	44	1136	44	1136	44	1136
Total	**2406**	**20.8**	**555**	**90**	**527**	**95**
Total cpu	1977	25.3	151	331	45	1111
Total input	288		297		374	
Total output	141		108		108	
Total i/o	429		405		482	

which all data is read. Some of the data (eg. parts of sorted segments) in the cache from the Sort Phase are still present in the cache during the Merge Phase and because the file is so small (5MB) this data does not get flushed by Merge Pase activity before it is actually needed by the Merge Phase. Therefore, not all the data that the Merge Phase requires must be read from disk and so the actual Merge Phase input time decreases. (Unfortunately, this improvement in Merge Phase read time derived from the TASS cache does not hold as the size of the file increases since for larger files any data in the cache present at the end of the Sort Phase is highly likely to be removed from the cache by Merge Phase activity before that data is actually needed by the Merge Phase. This can be seen by comparing Sort and Merge Phase times for a given record size in Table 4.2.) So, due to the cache, our Merge Phase input time estimate is not as accurate as it appears even for the 5MB file with 8-byte records.

In addition, it seems that in Tables 4.1 and 4.2 some of the Merge Phase time that should be attributed to the output is being attributed to the input. This is because the (virtual) output time is only measuring the time required to transfer the output data from the sort node to the CacheServer node and doesn't include the time required to transfer the data from the CacheServer node to the disk. However, this data must be transferred to disk eventually. When a read (or pre-fetch) request comes to the CacheServer space must be made in the cache for the chunk being read from disk (if the requested chunk is not already in the cache). At this time some dirty chunk may be flushed from the cache and the time required for this flush attributed to the read time. Thus, the input times may be somewhat inflated and the output times deflated.

However, there is still one more complication: when the CacheServer is otherwise idle it writes dirty chunks to disk. When the sort implementation does not use double-buffering, as is the case for the implementation we are examining now, there will indeed be times when the CacheServer is otherwise idle. So some of the time the cache spends writing chunks to disk will be attributed to neither the virtual input nor virtual output times. Thus, the i/o times reported in Tables 4.1 and 4.2 are somewhat lower than they would be without the CacheServer.

So, a better, but still not perfect, figure against which to compare our estimated

value for Merge Phase input time would be the average of the input and the output times. Thus, the actual Merge Phase input time for the 50MB file of 8-byte records could be taken as $244/2 = 122$ seconds. This is much closer to the estimated 91 seconds calculated above but still significantly off the mark. So, the conclusion we draw is that a more sophisticated analysis of the time required to perform i/o in the Merge Phase is needed than the one given in Chapter 2. Such an analysis would need to include the effects of the CacheServer on the virtual i/o times.

Continuing now with our examination of the data in Tables 4.1 and 4.2 we can see that, as expected, the total throughput for the sort is much better when sorting large records. There is an almost five-fold increase in total throughput from 21.7 Kb/sec for the 8-byte records to 108 Kb/sec for the 128-byte records when the size of the file is 5MB. For the 50MB files there is more than a four-fold increase from 20.8 Kb/sec for the small records to 90 Kb/sec for the large records. The increase in total sort throughput is due almost entirely to the decreased amount of cpu-processing required to sort the smaller number of records that result from using larger records in a file of a constant size. For example, the total cpu throughput is approximately 13 times greater (331 Kb/sec vs. 25.3 Kb/sec) when sorting 50MB files composed of 128-byte records rather than 8-byte records. Notice that the total i/o times remain relatively constant (compared to the changes in the cpu times) for a given file size as the record size is increased.

To further test this trend, the sort was re-executed with a 50MB file containing 1056-byte records. The results are included in Table 4.2. We see that sorting 1056-byte records again results in improved throughput, this time to 95 Kb/sec. The improvement, however, is not nearly as dramatic as when moving from 8-byte to 128-byte records in the 50MB files. This is because we are running into the limit on improvement of overall throughput imposed by the i/o bandwidth. However, we see that we can indeed alter the proportion of cpu vs. i/o time in the overall execution time by altering the size of the records used.

Notice from Tables 4.1 and 4.2 that the speed of output during the Merge Phase is consistently at or better than 1136 Kb/sec. This is actually approaching measured transputer link speed - approximately 1.2MBytes/sec. As explained above it is TASS

which allows the write throughput to achieve the limit set by the transputer link's data rate. Of course, this can only happen when the Cache Node does not have to flush to disk some of the data which it contains before it can accept more data from the Sort Node. As mentioned above, TASS automatically writes data to disk when it would otherwise be idle, ie. when the Sort Node is not making any request upon it. In the non-double-buffered implementation of the sort the Cache Server has plenty of time to flush data and so the Sort Node often perceives that writes occur near link speed.

The input times for Merge Phase are better than random read times of 230 Kb/sec as reported in [Dud92] because in the Merge Phase we try to read as many chunks as possible after performing a seek. Completely random reads, on the other hand, only read one chunk per seek. This is why, as explained before, it is important to use as large buffers as possible during the Merge Phase.

As expected, since sorting is $O(n \log n)$, the total throughput for a given record size decreases as the file size increases, from 21.7 Kb/sec to 20.8 Kb/sec for the 8-byte records and from 108 Kb/sec to 90 Kb/sec for the 128-byte records.

Perhaps the most striking piece of data from Table 4.2 is the throughput of the cpu in the Sort Phase for the 1056-byte records: 5000 Kb/sec. When sorting large records the processing power of even one single transputer is more than a match for the 730 Kb/sec sequential throughput of our i/o subsystem. That is, the transputer can sort large records into sorted segments far faster than these records can be read from disk. In the Merge Phase, too, the cpu throughput is faster than the maximum disk transfer rate.

Recall from Chapter 2 that there is some question as to the optimal number of passes to be performed during the Merge Phase. We have predicted above that one pass should be optimal for the the files discussed in this section. There is a quick and obvious way to demonstrate that using two passes (and larger buffers) in the Merge Phase cannot reduce the time for this phase from that taken when using one pass (and smaller buffers). When using a single disk without double-buffering the total i/o time for each pass cannot be less than the time required to sequentially read and sequentially write all the data in the file. The actual i/o time will really be somewhat

greater than this sequential time due to the amount of seeking required during the Merge Phase. If the measured i/o time for a one pass Merge Phase is less than the time needed to sequentially perform the i/o for two passes then adding a second pass can only increase the time for this phase. For both the 5 and 50MB file sizes the time taken for i/o during the Merge Phase indicates that using a second pass in the Merge Phase would not be advantageous. For example, it would take 68.5 seconds to read, strictly sequentially, 50Mb of data using TASS with a single disk. It would take approximately the same amount of time to sequentially write the same amount of data. Therefore, since we are using only one disk here, the total i/o time for one pass in the Merge Phase must be at least 137 seconds. Thus, two passes would require a minimum i/o time of $2 \times 137 = 274$ seconds. The Merge Phase i/o times reported for the 50MB file in Table 4.2 are all less than 274 seconds so adding a second pass would be counter-productive.

The most significant data in Table 4.2 is that for the 50MB file with 128-byte records because this most closely approximates a "real-world" scenario for external sorting. The total sort time for this file and record size is 555 seconds of which i/o comprises 405 seconds. Thus, i/o consumes 73% of the total sort time. This is the first concrete evidence we present that input/output is a major bottleneck in external sorting. This i/o bottleneck is further illustrated by the data for the 1056-byte records in Table 4.2 where we see that i/o consumes 91% of the total sort time.

Note that the overall cpu throughput for the 50MB file with 128-byte records is 331 Kb/sec. The optimal disk transfer rate on input that can be achieved through TASS is approximately 730 Kb/sec (for contigous data) and so the absolute best total sort rate we could achieve is 730 Kb/sec, assuming we only needed to read the file once and that we had a second disk. However, with the standard external sorting algorithm that we are using we expect, with a one-pass Merge Phase, to have to read the file being sorted twice, and so the absolute best total sort rate we can achieve is $730/2 = 365$ Kb/sec. Thus, we can infer that if i/o and sorting could be performed perfectly in parallel, then *one* T800 transputer is almost enough to sort a 50MB file of 128-byte records in the minimum possible time. So, we can anticipate that more modern processors will be able to sort 100-byte records into sorted segments and then

merge these segments faster than the records can be supplied by our i/o subsystem. Clearly, to achieve continued increases in the overall throughput of the external sort, increases in the speed of the i/o subsystem are needed.

Finally, note that the fastest total sort throughput actually achieved for any of the file/record size combinations listed in Tables 4.1 and 4.2 is 108 Kb/sec for the 5Mb file with 128-byte records. This is approximately 30% of the maximum 365 Kb/sec rate achievable when the input and output files are stored on a single disk. For the larger file, the one we are more interested in, the best total throughput achieved is somewhat smaller - 95 Kb/sec for the 1056-byte record size. Clearly, we haven't yet reached the limit imposed by the single disk input rate.

4.3 Single Node; Single Disk Configuration - Double-Buffered

The first way we attempt to improve the performance of the basic implementation of the external sort is to use double-buffering in both the Sort and Merge Phases of the algorithm. We expect that double-buffering will improve the total throughput for the Sort Phase. However, as discussed in Chapter 2, there is some question as to whether double-buffering will improve or degrade the performance of the Merge Phase.

4.3.1 Analysis

As we are now using both Replacement Selection in the Sort Phase and double-buffering in both phases the equation to use to determine the input time for one pass of the Merge Phase is equation 2.19 and the equation to use in determining the optimal number of passes is equation 2.20. We will evaluate these equations for our environment for the 50MB file size. However, the value of doing so is somewhat limited since, as we saw in the previous section, the predictions provided are not very accurate.

The values to use for all the variables in equations 2.19 and 2.20 remain unchanged from those used in the previous section, ie. $s = 0.0589$ seconds, $ebt = 0.0231$ seconds,

and, for 50MB files, $P = 14$, $b = 2960$, and $nsg = 13.9$. Therefore, $4 \times P \times nsg \times s + b \times ebt = 4 \times 14 \times 13.9 \times 0.0589 + 2960 \times 0.0231 = 114.2$ seconds. This is longer than the time predicted when double-buffering is not used, as expected.

Using equation 2.20 and substituting the values from above we find that the predicted value for the optimal merge order when double-buffering is used is 13.

4.3.2 Results

The results of running the double-buffered version of the external sort on exactly the same files used to obtain the data in Table 4.1 are presented in Table 4.3. Comparing the results in Table 4.3 with those of tables 4.1 and 4.2 we see that for every file and record size there is a substantial decrease in the total amount of time that the sort/merge process waits for i/o to complete. For example, for the 50MB file of 128-byte records the total i/o time for the non-double-buffered sort is 405 seconds whereas the corresponding time for the double-buffered implementation is 308.2 seconds - a savings of about 24%. This decrease in virtual i/o time is as expected.

Looking at the input times for the 50MB files we see a large difference between the times for the 8-byte records and for the 128-byte records. The 8-byte record sort/merge is completely cpu-bound and so almost all i/o can take place under the care of the worker process at a rate faster than the sort/merge process can deal with the data. For example, on input the reader process can get records from disk faster than they can be merged during the Merge Phase by the cpu. Thus, every time the sort/merge process asks for more records to merge these records are already present in memory and so the the virtual Merge Phase input time is very low (5.9 seconds). The 128-byte record merge, on the other hand, is i/o-bound and so the sort/merge process can merge records faster than they can be brought into memory. Therefore, the virtual time for the Merge Phase input is large (157.8 seconds). For this reason it is now especially hard to compare the Merge Phase input times presented in Table 4.3 with the predicted value of 114.2 seconds. Therefore, from this point on we make no attempt to predict the Merge Phase input time based on the analysis of Chapter 2.

Note that the output times for the Merge Phase are almost negligible - all are less

Table 4.3: Sort Times and Throughput Rates for a Single T-Node with a Single Disk - WITH double-buffering

File Size	5 MBytes				50 MBytes			
Record Size	8 Bytes		128 Bytes		8 Bytes		128 Bytes	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**174**	**28.7**	**23.9**	**209**	**1552**	**32.2**	**243.9**	**205.0**
cpu	163.7		11.5		1449		93.8	
input	5.0		9.4		49.9		128.5	
output	5.3		3.0		53.4		21.6	
Merge Phase	**36.2**	**138**	**22.4**	**223**	**532**	**94**	**258.6**	**193.3**
cpu	35.3		19.6		526		100.5	
input	0.8		2.6		5.9		157.8	
output	0.1		0.2		0.1		0.3	
Total	**210**	**23.8**	**46.3**	**108**	**2084**	**24.0**	**502.5**	**99.5**
Total cpu	199		31.1		1975		194.3	
Total input	5.8		12.0		55.8		286.3	
Total output	5.4		3.2		53.5		21.9	
Total i/o	11.2		15.2		109.3		308.2	

than half a second. One of the reasons for such a low figure is that to ouput a buffer's worth of data the sort/merge process now needs simply to release that buffer. The rest of the work is done by another process and the sort/merge process does not wait for this work to be completed before it returns from its call to the output procedure.

For both the 5MB and 50MB files with 8-byte records there is a slight decrease in the amount of total cpu time. However, for the two files with 128-byte records there is a substantial *increase* in the amount of total cpu time. For the 5MB file the total cpu time more than doubles from 14.7 seconds to 31.1 seconds; for the 50MB file this time increases by 29% from 151 seconds to 194.3 seconds. This was unexpected. Note that most of this increase occurs in the Merge Phase. Possibly this is due to increased processing overhead associated with the double-buffering.

We can, once again, easily see from the data in Table 4.3 that the use of a second pass will not improve the Merge Phase time for the 50MB file composed of 128-byte records - even though we are merging more than the predicted optimal number of

segments (recall that predicted optimal merge order is 13; actual number of segments is 16). Two passes for this size of file will still cost a minimum of 274 seconds because the use of double-buffering does not speed up the disk - only the virtual i/o rate. Therefore, since the average Merge Phase time for the 128-byte record 50MB files is only 258.6 seconds, a second pass would be detrimental. A second pass is not likely to help for the other file and record sizes listed in Table 4.3 as for these combinations the cpu is the primary bottleneck. The formulas we presented in section 2.2.2 will not help us decide if a second pass is worthwhile here as these formulas only account for the cost of i/o during the Merge Phase. We did, however, implement a two-pass Merge Phase and ran it on these files. The resulting times were much worse than for the single-pass version. For example, for the 50MB file of 8-byte records the two-pass Merge Phase took a total of 856.2 seconds, more than 60% longer than the one-pass version (532 seconds).

Comparing Table 4.1 with Table 4.3 we see that the addition of double-buffering does not have a significant impact on the overall throughput of the external sort for the small files. In fact, for 5MB file with 128-byte records the time for the double-buffered sort (46.3 seconds) is actually slightly, though not significantly, worse than that of the non-double-buffered sort (46.1 seconds). However, for the 50MB files double-buffering provides a significant improvement in overall throughput for both the record sizes. The greatest improvement, about 15%, appears when sorting the 50MB file with 8-byte records where the throughput went from 20.8 Kb/sec to 24.0 Kb/sec. For the 50MB file with 128-byte records there is a smaller improvement of approximately 11% (the non-double-buffered rate is 90 Kb/sec; the double-buffered rate is 99.5 Kb/sec).

In short, double-buffering generally improves the overall throughput of a single disk external sort in our environment but still leaves us far short of the limit imposed by the sequential read rate of a single disk. In section 4.5 we will provide data on the effects of double-buffering when more than one disk is used. There we will see that the use of double-buffering significantly improves the over-all performance of the sort. Thus, we will continue to use double-buffering throughout the remaining tests.

4.4 Single Node; Two Disk Configuration

An obvious way to achieve increased throughput of the i/o subsystem is to use extra disks. We have previously discussed ways in which extra disks can be used in a single processor system to increase the speed of the external sort. In the following sections we show the results of using extra disks in various ways in our hardware environment.

We first discuss the effect on performance of adding a second disk which will be used to store the sorted segments created in Phase I. The input file and output file will both still be stored on the original disk. We expect the use of a second disk to reduce the cost of i/o in both phases of the sort. As previously discussed, in the Sort Phase this is because the second disk eliminates the need for virtually all disk seeks. In the Merge Phase substantial seeking will still be required on the disk containing the sorted segments. However, no seeking will be required during the output of the sorted file. Thus, the overall number of seeks in the Merge Phase will decrease, reducing the total i/o time for this phase also.

With the addition of a second disk the maximum size of file which can be sorted becomes 100MB. We defer, however, introduction of data for this size of file until the next section.

Comparing Table 4.4 with Table 4.3 we see that there is essentially no change in any of the times for either the small or the large files with 8-byte records. However, the sorts for both of these files are essentially cpu-bound and thus it isn't surprising that an improvement in the i/o sub-system doesn't have much of an impact on the sort times for these files. Comparing the data for the files with large records, however, shows that the use of a second disk produces a dramatic improvement in total sort times. For the 5MB files the total time is reduced by 44% from 46.3 seconds to 26.1 seconds. For the 50MB files there is an even greater reduction of 56% from 502.5 seconds to 219.6 seconds.

Most significantly, when sorting the 50MB file with 128-byte records the throughput rates for both phases are now more than half way to the limit (365 Kb/sec) imposed by the rate at which the file can be read from a single disk (730 Kb/sec). That is, we are rapidly approaching the point at which the i/o becomes the bottleneck.

Table 4.4: Sort Times and Throughput Rates for a Single T-Node with TWO Disks (with double-buffering)

File Size	5 MBytes				50 MBytes			
Record Size	8 Bytes		128 Bytes		8 Bytes		128 Bytes	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**174**	**28.7**	**16.0**	**312.5**	**1551**	**32.3**	**106.8**	**468.2**
cpu	163.6		10.9		1448		86.5	
input	5.0		4.3		49.6		12.7	
output	5.4		0.8		53.5		7.6	
Merge Phase	**36.2**	**138**	**10.1**	**495.0**	**532**	**94**	**112.8**	**443.3**
cpu	35.3		4.2		526		65.4	
input	0.8		5.7		5.9		47.2	
output	0.1		0.2		0.1		0.2	
Total	**210**	**23.8**	**26.1**	**191.6**	**2083**	**24.0**	**219.6**	**227.7**
Total cpu	199		15.1		1974		151.9	
Total input	5.8		10.0		55.5		59.9	
Total output	5.4		1.0		53.6		7.8	

Clearly, adding a second pass to the Merge Phase cuts the maximum throughput of the Merge Phase down to half the throughput of the disk. Therefore, since in Table 4.4 the Merge Phase throughputs for both the 5MB and 50MB files composed of 128-byte records is already more than half the 730KB/sec throughput limit of a single disk, a two-pass Merge Phase will result in worse times. The Merge Phase times for the files composed of 8-byte records are cpu-bound and so a second pass is not going to help, just as was the case when only one disk was used during the sort.

4.5 Single Node; Three and More Disk Configurations

As explained previously, much of the time consumed by input in the Merge Phase is expended on disk seeks. By storing the sorted segments on multiple disks some of these seeks may be executed in parallel. There are two methods in which we may add extra disks for the storage of the intermediate segments using TASS. The simplest way

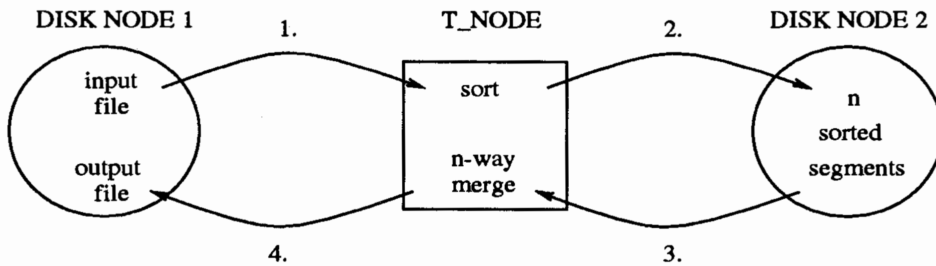
is store the segments on a single TASS storage node of type (ii) or (iii) instead of type (i) as shown in Figure 1.2. This change is virtually transparent to the programmer. The only apparent difference is that of a faster disk with more storage. The logical view in this case, shown in Figure 4.1 (i), remains unchanged from that shown in Figure 2.2. We just substitute for "DISK NODE 2" in Figure 2.2 the TASS storage node shown in Figure 1.2 (ii) or (iii) instead of that shown in Figure 1.2 (i). TASS is responsible for deciding on which disk each chunk of a segment will be stored. The results obtained using this method are given in section 4.5.1.

If two interleaved disks are used to store a single file then [Dud92] reports that the sequential throughput improves to 1142Kb/sec. So, if two disks are used in an interleaved fashion to store the sorted segments then, at first glance, it might seem that we can improve the value of the effective block transfer time, *ebt*, for the sorted segments from 0.0231 seconds to 0.0148 seconds. Unfortunately, a single block is still contained on a single disk and so it can't be read any faster when two interleaved disks are used on a single storage node. The advantage of using two disks in an interleaved fashion only becomes apparent if, after having performed a seek[5], more than one block is read. Then the CacheServer can simultaneously read two blocks in parallel, one block from each disk. Therefore, as long as any buffer we use is large enough to contain at least two blocks, then interleaved disks will improve the value for *ebt*.

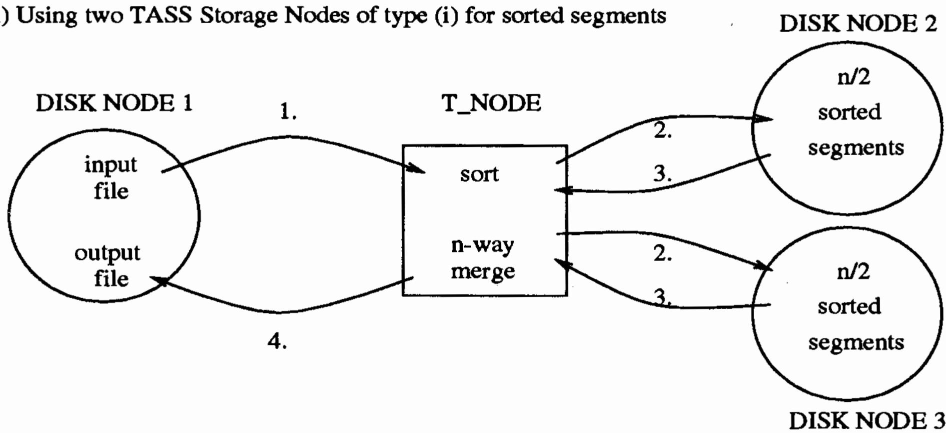
The second way to use extra disks for storing the sorted segments is to add one or two more TASS storage nodes of the type shown in Figure 1.2(i). This method is shown in Figure 4.1 (ii) and (iii). This method is not transparent to the programmer. That is, the program for the Sort Phase must specify on which disk to store the current sorted segment and then make this information available to the Merge Phase so that the appropriate disk for each segment can be read during this phase. The results obtained using this method are given in section 4.5.2. Unlike the first method this method places no limit on the size of the buffers needed to obtain optimal performance.

[5]Actually, two seeks in parallel, one on each disk

(i) Using one TASS Storage Node of type (ii) or (iii) for sorted segments



(ii) Using two TASS Storage Nodes of type (i) for sorted segments



(iii) Using three TASS Storage Nodes of type (i) for sorted segments

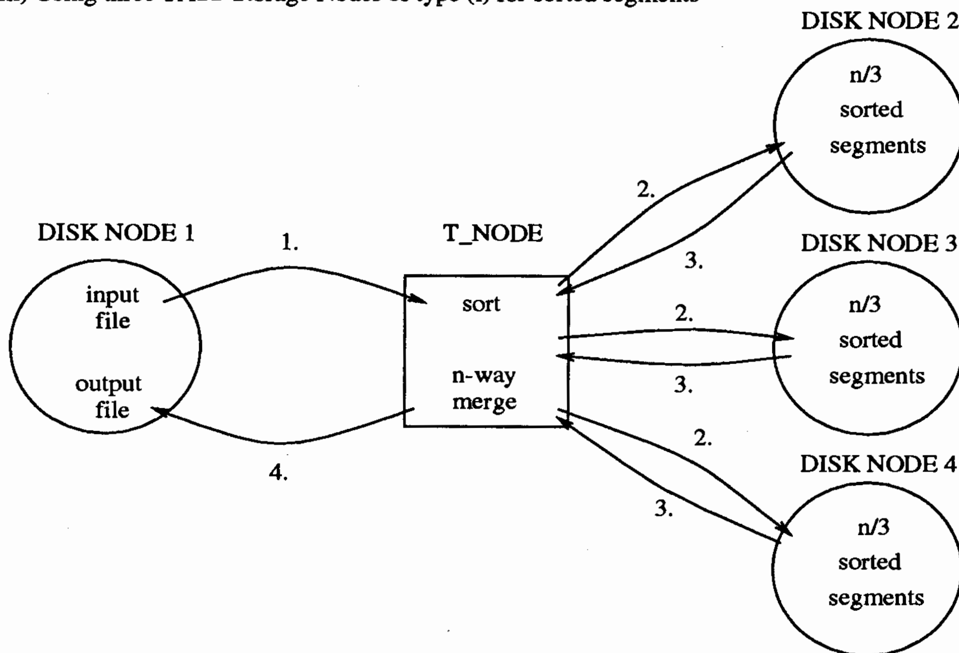


Figure 4.1: Possible Configurations for Storing Sorted Segments on Multiple Disks

4.5.1 Using a Single TASS storage node

In Tables 4.5 to 4.8 we list performance results for when a single TASS storage node with one or more disks attached to it is used for the storage of the sorted segments. Some data from previous sections is also included in Tables 4.5 and 4.6 to facilitate comparisons. Tables 4.5 and 4.6 present data for 50MB files; Tables 4.7 and 4.8 for 100MB files. In all the tables the total number of disks specified includes the one disk on which the original and sorted files are to reside. For example, a total of 3 disks means 1 disk for the original and sorted files and two other disks for the temporary sorted segments. Tables 4.7 and 4.8 do not include data for the one disk case because a single disk is not large enough to sort a 100MB file with. Finally, since there was no change in the sort times for the files with 8-byte records when moving from the use of a single disk to using two disks (that is, it was and will remain cpu-bound), only the data for the 128-byte records is given in these tables.

Before examining the results of adding a second and third disk for storage of the sorted segments, let us briefly return to the effect on performance that double-buffering has. There is little doubt that double-buffering should improve the throughput of the Sort Phase. However, recall from section 2.2.2 that there is the possiblity that the use of double-buffering might have a negative impact on the throughput for the Merge Phase because of the increased amount of seeking double-buffering would entail in this phase. Tables 4.5 and 4.7 show data for the non-double-buffered version of the sort and Tables 4.6 and 4.8 show data for the double-buffered version.

From these tables it can be seen that, in every case listed, the throughput of the Sort Phase does indeed improve with the addition of double-buffering, as expected. For example, when sorting 50MB files using two disks double-buffering increases the Sort Phase throughput from 266 Kb/sec to 468 Kb/sec - a 76% improvement. More importantly, comparing the Merge Phase throughputs of the non-double-buffered and double-buffered versions shows that, in every case, the use of double-buffering definitely is advantageous in this phase. For example, when sorting 50MB files with two disks double-buffering provides a 92% improvement in throughput for the Merge Phase (231 Kb/sec for the non-double-buffered implementation; 443 Kb/sec for the

double-buffered version). Moreover, Tables 4.5 and 4.6 show that double-buffering is especially beneficial when the sorted segments are stored on disks separate from that used to store the original and sorted files. For example, when sorting the 50MB file using only a single disk the increase in throughput due to double-buffering for the Sort and Merge Phases are only 9% and 12%, respectively. The improvements provided by double-buffering when multiple disks are used, as discussed in the previous paragraph, are far greater.

We now examine the effect on performance of increasing the number of disks used to store sorted segments. From the 2 and 3 disks columns of Tables 4.5 through 4.8 it can be seen that the addition of a third disk (that is, a second for the sorted segments) does significantly improve the throughput for the Merge Phase. For example, for the 50MB file the Merge Phase throughput of the double-buffered implementation increases from 443 Kb/sec when using a total of two disks to 523 Kb/sec when using 3 disks, an 18% improvement. Similarly, for the 100MB file there is a 13% improvement from 310 Kb/sec to 349 Kb/sec. The respective improvements for the non-double-buffered implementation are not as great, further demonstrating the utility of double-buffering. Note, however, that even for the double-buffered version, the increase in Merge Phase throughput obtained with the third disk is not nearly as great as that achieved with the addition of the second disk.

Adding a third disk for the sorted segments (four disks in total) is even less advantageous.[6] For the 100MB file, using a total of four disks improved the Merge Phase throughput of the double-buffered implementation by only 9% over the 3 disk case (349 Kb/sec vs. 382 Kb/sec). Surprisingly, for the double-buffered version, the addition of the fourth disk actually produced a *worse* throughput for the 50MB files than was achieved for these files with both the three and two disk cases.

Why did the fourth disk produce a small, but significant, improvement in performance for the 100MB file yet result in a decrease in performance for the 50MB

[6]It needs to be mentionned that at the end of the writing of this thesis it was discovered (by the developer of TASS) that one of our four disks has a worse performance than the other three. For example, its maximum sequential input is approximately 590 Kb/sec as opposed to the 730 Kb/sec for each of the other three disks. Fortunately, the slow disk was only used in our experiments which required all four disks, so only these experiments show performance which is worse than would otherwise have been obtained.

Table 4.5: Sort Times and Throughput Rates For a Single T-Node Using One TASS storage node with Multiple Disks for the Sorted Segments (no double-buffering)

File/Record Size	50 MBytes / 128 bytes (16 segments)							
Tot. No. of Disks	1		2		3		4	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**265.1**	**188.7**	**187.7**	**266.4**	**187.5**	**266.7**	**188.2**	**265.7**
cpu	84.0	595.4	85.7	583.6	85.8	582.9	85.6	584.3
input	116.7	428.6	52.1	959.9	52.0	961.8	53.1	941.8
output	64.4	776.6	49.9	1002.2	49.7	1006.3	49.5	1010.3
Merge Phase	**290.0**	**172.5**	**216.1**	**231.4**	**210.6**	**237.5**	**217.5**	**229.9**
cpu	66.5	752.1	66.5	752.1	66.5	752.1	66.4	753.2
input	179.7	278.3	106.0	471.8	100.4	498.1	107.7	464.4
output	43.8	1141.8	43.6	1147.1	43.7	1144.4	43.4	1152.4
Total	**555.1**	**90.1**	**403.8**	**123.8**	**398.1**	**125.6**	**405.7**	**123.3**
Total cpu	150.5	332.2	152.2	328.5	152.3	328.4	152.0	328.9
Total input	296.4		158.1		152.4		160.8	
Total output	108.2		93.5		93.4		92.9	

Table 4.6: Sort Times and Throughput Rates For a Single T-Node Using One TASS storage node with Multiple Disks for the Sorted Segments (with double-buffering)

File/Record Size	50 MBytes / 128 bytes (16 segments)							
Tot. No. of Disks	1		2		3		4	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**243.9**	**205.0**	**106.8**	**468.2**	**106.6**	**469.0**	**108.4**	**461.2**
cpu	93.8		86.5		86.3		86.1	
input	128.5		12.7		12.7		14.7	
output	21.6		7.6		7.6		7.6	
Merge Phase	**258.6**	**193.3**	**112.8**	**443.3**	**95.6**	**523.0**	**116.7**	**428.4**
cpu	100.5		65.4		65.7		112.0	
input	157.8		47.2		29.7		4.1	
output	0.3		0.2		0.2		0.6	
Total	**502.5**	**99.5**	**219.6**	**227.7**	**202.2**	**247.3**	**225.1**	**222.1**
Total cpu	194.3		151.9		152.0		198.1	
Total input	286.3		59.9		42.4		18.8	
Total output	21.9		7.8		7.8		8.2	

Table 4.7: Sort Times and Throughput Rates For a Single T-Node Using One TASS storage node with Multiple Disks for the Sorted Segments (no double-buffering)

File/Record Size	100 MBytes / 128 bytes (32 segments)					
Tot. No. of Disks	2		3		4	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**367.2**	**272.3**	**366.3**	**273.0**	**366.7**	**272.7**
cpu	165.8	603.1	165.8	603.1	165.8	603.1
input	101.1	988.9	101.2	988.0	102.0	980.2
output	100.3	996.9	99.3	1006.7	98.9	1011.2
Merge Phase	**616.1**	**162.3**	**591.8**	**169.0**	**589.3**	**169.7**
cpu	141.7	705.6	141.7	705.6	141.7	705.6
input	386.9	258.4	362.5	275.8	360.1	277.7
output	87.5	1142.7	87.6	1141.4	87.5	1142.4
Total	**983.3**	**101.7**	**958.1**	**104.4**	**956.0**	**104.6**
Total cpu	307.1	325.6	307.5	325.2	307.5	325.2
Total input	488.0	204.9	463.7	215.6	462.1	216.4
Total output	187.8	532.4	186.9	535.0	186.4	536.4

Table 4.8: Sort Times and Throughput Rates For a Single T-Node Using One TASS storage node with Multiple Disks for the Sorted Segments (with double-buffering)

File/Record Size	100 MBytes / 128 bytes (32 segments)					
Tot. No. of Disks	2		3		4	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**206.1**	**485.2**	**205.5**	**486.6**	**208.6**	**479.3**
cpu	170.0		169.4		169.2	
input	20.9		20.9		24.2	
output	15.2		15.2		15.2	
Merge Phase	**322.5**	**310.1**	**286.8**	**348.7**	**261.6**	**382.3**
cpu	144.5		144.6		144.8	
input	177.9		142.1		116.7	
output	0.1		0.1		0.1	
Total	**528.6**	**189.2**	**492.3**	**203.1**	**470.2**	**212.7**
Total cpu	314.5		314.0		314.0	
Total input	198.8		163.0		140.9	
Total output	15.3		15.3		15.3	

file? There are only two parameters which (may) vary when moving from sorting a 50MB file to sorting a 100MB file: the size of the i/o buffers in the Merge Phase and the pre-fetch degree. Therefore, we now examine the effects of these two factors more closely. In Tables 4.9 through 4.15 we list the times taken using various sizes of buffers and various degrees of pre-fetching for just the Merge Phase of the sort. Tables 4.9, 4.10, 4.11, and 4.12 list the results for 50MB files using one, two, three, and four disks respectively; Tables 4.13, 4.14, and 4.15 list the data 100MB files for two, three, and four disks respectively.

Unfortunately, there doesn't appear to be any consistent pattern applicable to all of these tables. The data in the Tables for the 50MB files are especially unruly. It turns out that the best times for both the one-disk and three-disks 50MB sorts (254.7 seconds in Table 4.9 and 76.5 seconds in Table 4.11, respectively) are achieved when using a buffer size of 2 chunks and a pre-fetch degree of 2 chunks. However, when using two disks the best 50MB time (113.3 seconds in Table 4.10) is achieved when using a buffer size of 2.5 chunks and a pre-fetch degree of 2. The times for the four-disks case (Table 4.12) are almost impervious to any change in the size of the buffers or the pre-fetch degree used, and they are consistently worse than the times for the two and three disk cases. The Merge Phase times for the four-disk case are almost always within one second of 116 seconds and the best time (115.6 seconds) was achieved with a buffer size of one chunk and a pre-fetch degree of one chunk. This is especially suprising because reducing the size of the buffer actually seems to improve the performance, although only very slightly.

Tables 4.13 through 4.15 for the 100MB files are much less variable, though still not consistent. For example, if one looks at the data in each of these Tables for one particular buffer size and pre-fetch degree (eg. a buffer size of one chunk and a pre-fetch degree of two chunks) one can see that adding an extra disk almost always reduces the Merge Phase time, or at least leaves it unaffected. This, however, is not true for the buffer size and pre-fetch degree which produce the best time in each table (buffer size of one chunk and a pre-fetch degree of one chunk). For these times, adding the fourth disk results in a worse time than that obtained with three disks. This is similar to what occurs when sorting the 50MB files. That is, the best times

Table 4.9: Merge Phase, 50MB (16 segments), 1 disk

Buffer Size (in chunks)	Prefetch Degree 1 secs	2 secs	3 secs
1 Total	**272.0**	**274.5**	**483.4**
cpu	118.1	124.7	129.2
input	153.6	149.5	354.0
output	0.3	0.3	0.2
1.5 Total	**273.1**	**302.7**	**441.7**
cpu	107.3	107.9	121.2
input	165.4	194.6	320.3
output	0.4	0.2	0.2
2 Total	**271.4**	**254.7**	**336.9**
cpu	95.3	101.1	105.0
input	175.9	153.3	231.7
output	0.2	0.3	0.2
2.5 Total	**273.1**	**259.9**	**374.7**
cpu	104.1	99.8	115.2
input	168.7	159.8	259.2
output	0.3	0.3	0.3

Table 4.10: Merge Phase, 50MB (16 segments), 2 disks

Buffer Size (in chunks)	Prefetch Degree 1 secs	2 secs	3 secs
1 Total	**155.7**	**155.2**	**158.6**
cpu	67.7	67.5	67.5
input	87.8	87.5	90.9
output	0.2	0.2	0.2
1.5 Total	**158.0**	**128.9**	**135.2**
cpu	66.9	67.7	67.7
input	89.8	61.0	67.3
output	0.2	0.2	0.2
2 Total	**161.5**	**119.1**	**120.4**
cpu	65.9	66.0	66.1
input	95.4	52.9	54.1
output	0.2	0.2	0.2
2.5 Total	**163.8**	**113.3**	**142.7**
cpu	64.9	65.4	65.1
input	98.7	47.7	77.4
output	0.2	0.2	0.2

Table 4.11: Merge Phase, 50MB (16 segments), 3 disks

Buffer Size (in chunks)	Prefetch Degree 1 (secs)	2 (secs)	3 (secs)
1 Total	**78.2**	**79.9**	**86.3**
cpu	68.6	68.6	68.6
input	9.4	11.1	17.6
output	0.2	0.2	0.2
1.5 Total	**137.8**	**80.4**	**117.0**
cpu	68.0	68.6	113
input	69.6	11.6	3.5
output	0.2	0.2	0.5
2 Total	**167.1**	**76.5**	**82.1**
cpu	65.9	66.9	66.8
input	101.0	**9.4**	15.1
output	0.2	**0.2**	0.2
2.5 Total	**178.1**	**96.1**	**108.5**
cpu	64.9	65.7	65.4
input	113.0	30.2	42.9
output	0.2	0.2	0.2

Table 4.12: Merge Phase, 50MB (16 segments), 4 disks

Buffer Size (in chunks)	Prefetch Degree 1 (secs)	2 (secs)	3 (secs)
1 Total	**115.6**	**116.3**	**116.6**
cpu	113.6	113.5	113.0
input	1.5	2.4	3.3
output	0.5	0.4	0.3
1.5 Total	**123.6**	**116.5**	**116.9**
cpu	68.0	113.8	113.1
input	55.3	2.4	3.5
output	0.3	0.3	0.3
2 Total	**157.9**	**116.8**	**117.1**
cpu	65.9	114.0	113.1
input	91.8	2.4	3.5
output	0.2	0.4	0.5
2.5 Total	**169.7**	**116.7**	**117.6**
cpu	64.9	111.9	107.5
input	104.5	4.3	9.7
output	0.3	0.5	0.4

Table 4.13: Merge Phase, 100MB (32 segments), 2 disks

	Prefetch Degree		
Buffer Size	0	1	2
(in chunks)	secs	secs	secs
1 Total	**499.3**	**310.2**	**541.6**
cpu	143.5	144.5	143.9
input	355.5	165.5	397.5
output	0.3	0.2	0.2
1.5 Total	**447.5**	**321.2**	**422.9**
cpu	143.7	144.6	144.0
input	303.6	176.4	278.7
output	0.2	0.2	0.1

Table 4.14: Merge Phase, 100MB (32 segments), 3 disks

	Prefetch Degree		
Buffer Size	0	1	2
(in chunks)	secs	secs	secs
1 Total	**500.5**	**181.1**	**432.7**
cpu	143.6	**145.8**	144.0
input	356.7	**35.0**	288.5
output	0.2	**0.3**	0.2
1.5 Total	**498.7**	**286.7**	**355.8**
cpu	143.5	144.9	144.4
input	355.0	141.6	211.3
output	0.2	0.2	0.1

Table 4.15: Merge Phase, 100MB (32 segments), 4 disks

	Prefetch Degree		
Buffer Size	0	1	2
(in chunks)	secs	secs	secs
1 Total	**497.3**	**234.7**	**400.5**
cpu	143.7	231.3	144.3
input	353.3	2.9	255.9
output	0.3	0.5	0.3
1.5 Total	**497.2**	**264.1**	**320.1**
cpu	143.5	145.0	144.6
input	353.4	118.8	175.4
output	0.3	0.3	0.1

(underlined in the tables) for both the 50MB files and the 100MB files are produced with a total of three disks and not four.

We created Tables 4.9 through 4.15 in an effort to determine why adding a third disk for the sorted segments seemed to improve the performance for the 100MB file but not for the 50MB file. We assumed that the size of buffer and the pre-fetch degree used during the Merge Phase might explain this discrepancy. Having produced these Tables we see that the values chosen for these two parameters can indeed have a large impact upon the performance of the Merge Phase. However, these Tables don't seem to reveal any method which can be used to determine in advance which buffer size and pre-fetch degree will produce the best results for a given file size using a given number of disks for the sorted segments.

From an analytic point of view, however, we have previously determined (section 2.2.2) that the best performance in the Merge Phase ought to be achieved when using as large a buffer size as possible. As for the pre-fetch degree, it would seem, based on [Dud92], that the best pre-fetch degree ought to be as large as is possible (upto 5) without resulting in pre-fetched chunks needing to be flushed from the cache before they are actually read by the application node.

More precisely, let us define a *fetched chunk* as a chunk which has been brought from disk and is currently residing in the cache. A *pre-fetched chunk* is a fetched chunk of which no part has yet been read by the Sort Node. It is the responsibility of the Sort Node to explicitly request that a chunk be pre-fetched. A fetched chunk turns into a *used chunk* once the sort/merge process has actually read all the data in that chunk.

Let the maximum possible number of fetched chunks in the TASS node be MC. Obviously, the total number of pre-fetched chunks requested at any instant must never exceed MC, otherwise one of the currently pre-fetched chunks will have to be removed from the cache to make room for the most recently pre-fetched chunk. Such an action would cause a degradation in performance as any fetched chunk which is so removed will eventually have to be re-read from disk when the merge process actually reads this chunk (each chunk is used precisely once). This extra read would very likely require an extra seek increasing the read time still further. Since, among all

the fetched chunks only one can actually be in the process of being read by the client node, the maximum number of chunks which can profitably be pre-fetched is $MC-1$. Therefore, the maximum pre-fetch degree is $\lfloor (MC-1)/nsg \rfloor$, where nsg is the total number of segments produced during the Sort Phase. With this pre-fetch degree each segment will have the maximum possible number of chunks belonging to it pre-fetched into the Cache Node without necessitating the removal of other pre-fetched chunks before they are used.

Is the maximum pre-fetch degree the optimal pre-fetch degree? In our environment $MC = 116$ and a 50MB file, for example, will typically result in 16 sorted segments. So the maximum pre-fetch degree for a 50MB file is $\lfloor (116-1)/16 \rfloor = 7$. However, Tables 4.9 through 4.12 indicate that much smaller pre-fetch degrees produce better results. Similar remarks can be made for the 100MB files. So, the maximum pre-fetch degree is not the optimal pre-fetch degree.

The main reason for this discrepancy seems to be the general purpose replacement algorithm which the TASS cache uses in deciding which chunk in the cache to replace when it needs room for a chunk currently being fetched. This replacement policy is a variation of the Least Recently Used heuristic. If the maximum pre-fetch degree is used by the merge process then the most recently used chunk must be the chunk which is replaced. However, with the TASS cache replacement policy pre-fetch degrees at or close to the maximum seem to cause pre-fetched and not used chunks to be replaced. Therefore, with the TASS cache a pre-fetch degree substantially lower than the maximum produces the best results. Exactly which pre-fetch degree results in the best performance for a given size of file needs to be determined experimentally. Tables 4.9 through 4.12 show that, for 50MB files, the optimum pre-fetch degree is 2; for 100MB files, Tables 4.13 through 4.15 show that the optimum pre-fetch degree is 1.

The actual optimum buffer size seems to depend on the interaction of the buffer size, pre-fetch degree, and number of disks on which the sorted segments are being stored. All of these appear to impact upon the cache's decision as to which chunk to replace when pre-fetching another chunk - sometimes a pre-fetched chunk and sometimes a used chunk. Therefore, to stay in line with the analysis of section 2.2.2,

we choose to use buffers which are as large as possible during the Merge Phase even though Tables 4.9 through 4.15 indicate that using smaller buffers can produce better results.

Finally, attaching extra disks to a TASS CacheServer is intended to be invisible to the user except in that they provide greater storage capacity and (hopefully) better performance. So, in producing our sort data, we don't alter the buffer size or pre-fetch degree when changing the number of disks we use for the sorted segments.

To summarize, the data in our sort Tables are based upon buffer sizes which are predicted optimal through theoretical analyses and pre-fetch degrees determined experimentally. The data we present are underestimations in that better performance can usually be achieved for a given file size and number of disks if various buffer sizes and pre-fetch degrees are experimented with for the given file size and number of disks used.

In conclusion, using a TASS storage node of type (ii) (ie., one node with two disks attached to it) for the segments provides better throughput in the Merge Phase than a TASS storage node of type (i) (ie., one node with one disk). Adding a third disk to the storage node for the segments is more problematic. In some cases, including the optimum achievable through experimentation, the third disk actually results in reduced performance. In any case, adding extra disks provides diminishing returns.

4.5.2 Using Multiple TASS storage nodes

In this section we discuss the results obtained when using multiple TASS storage *nodes* each with one disk attached, as shown in Figure 4.1(ii) and (iii). The data for this implementation is given in Table 4.16. The same 50MB and 100MB files and the same buffer size and pre-fetch degree in the Merge Phase were used to produce this table as were used to produce the analogous Tables in previous sections.

It can immediately be seen from Table 4.16 that, for both the 50MB and 100MB files, the addition of the fourth storage node has almost no impact on total throughput from that obtained with three storage nodes. For both sizes of files the Sort Phase throughput actually seems to decrease slightly, from 466.0 Kb/sec to 463.8 Kb/sec

Table 4.16: Sort Times and Throughput Rates for a Single T-Node with Multiple Single-Disk storage nodes for Sorted Segments

File Size	50 MBytes				100 MBytes			
Tot. No. of Disks	3		4		3		4	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**107.3**	**466.0**	**107.8**	**463.8**	**207.2**	**482.6**	**208.1**	**480.5**
cpu	85.9		85.8		168.9		169.0	
input	12.5		12.8		20.5		20.3	
output	8.9		9.2		17.8		18.8	
Merge Phase	**71.0**	**704.2**	**70.9**	**705.2**	**152.2**	**657.0**	**151.0**	**662.3**
cpu	67.9		67.8		146.9		146.7	
input	2.8		2.9		5.2		4.2	
output	0.3		0.3		0.1		0.1	
Total	**178.3**	**280.4**	**178.7**	**279.8**	**359.4**	**278.2**	**359.1**	**278.5**
Total cpu	153.8		153.6		315.8		315.7	
Total input	15.3		15.7		25.7		24.5	
Total output	9.2		9.5		17.9		18.9	

for the 50MB file, and from 482.6 Kb/sec to 480.5 Kb/sec. Neither of these decreases is significant, however. The Merge Phase throughput increases slightly for both sizes of files, as is expected. However, both increases are negligible. The Merge Phase throughput for the 50MB files goes from 704.2 Kb/sec using 3 storage nodes to 705.2 Kb/sec using 4 storage nodes; the corresponding figures for the 100MB files are 657.0 Kb/sec and 662.3 Kb/sec.

However, if the data in Table 4.16 is compared to the corresponding data in Tables 4.6 and 4.8 it can be seen that there is a big improvement in the Merge Phase throughputs when using multiple single-disk storage nodes vs. when a single multi-disk storage node is used. For example, for the 50MB files the best Merge Phase throughput in Table 4.6 is 523.0 Kb/sec. But when using multiple storage nodes the corresponding best throughput is 705.2 Kb/sec. This throughput is even better than the best achieved in all the experiments with varying buffer sizes and pre-fetch degrees presented in Tables 4.11 and 4.12. The best Merge Phase time in these Tables is 76.5 seconds which translates into a throughput of 653.6 Kb/sec.

Similar remarks hold for the 100MB files. The best Merge Phase throughput in Table 4.16 is 662.3 Kb/sec obtained when using three single-disk storage nodes for the sorted segments (4 disks in total). The best Merge Phase throughput obtained in Table 4.8 is 382.3 Kb/sec, also using a total of 4 disks. Even with experimentation, the best Merge Phase throughput that can be achieved using a multi-disk storage node is 552.2 Kb/sec, based on Tables 4.13 through 4.15.

There are a number of reasons why using multiple single-disk storage nodes provides better performance than using a single multi-disk storage node. Firstly, each storage node has 2MB of main memory so there is less likelihood that pre-fetched chunks will be pushed from the cache by other pre-fetched chunks before their contents are read by the Sort/Merge Node. Secondly, each additional storage node is attached to the Sort/Merge Node through its own separate link. Therefore, data from each storage node may be read into the Sort/Merge Node in parallel. Lastly, seeks occur on each of the disks in parallel, unlike when interleaved disks are used on a single storage node, as in the previous sections.

The throughput for the Merge Phase is now very close to the limit imposed by the throughput of a single disk. This limit is 730 Kb/sec and for the 50MB file the Merge Phase throughput is over 700 Kb/sec for both the three and four disk cases. There is, however, more room for improvement when sorting the larger 100MB file.

Finally, note that the throughputs of the Sort Phase have not changed siginificantly from the corresponding rates in Tables 4.6 and 4.8. Indeed, Table 4.16 is the first Table in which the throughput for the Merge Phase is consistently better, and much more so, than that of the Merge Phase. This is not too surprising as all the improvements we have made up to now have been intended to better the Merge Phase times. The Sort Phase throughputs have so far all been under 500 Kb/sec. This is much less than the 730 Kb/sec maximum sequential throughput of a single disk and not even half of the 1200 Kb/sec maximum throughput of a link. The bottleneck in the Sort Phase is the cpu.

4.6 Final Squeeze

Since using the fourth disk as a third disk on which to store the sorted segments provides such little improvement in performance, we now try using one two-disk storage node for the original and sorted files and two single-disk storage nodes for the sorted segments. The physical appearance of this arrangement is as in Figure4.1(ii). The data obtained with this arrangement is shown in Table 4.17. This arrangement also allows us, for the first time, to sort 200MB files and so we include data for files of this size in Table 4.17.

Notice from this Table that the Sort Phase throughput is steadily increasing as the size of the file increases. For example, the Sort Phase throughput of the 50MB file is 477.6 Kb/sec whereas the corresponding throughput for the 200MB file is 498.5 Kb/sec. While this is nice to see, it is unexpected. As explained in Chapter 2, the Sort Phase time is proportional to the size of the input file, and so we expect the Sort Phase throughput to remain constant regardless of the size of the file. The only explanation that we can think of is that some fixed cost of the Sort Phase is being amortized against the size of the file. Both the Merge Phase throughput and the total throughput behave as expected as the size of the file increases: that is, they decrease.

Comparing the total throughput figures in Table 4.17 with the total throughput figures in Table 4.16 we see that using four disks in this way (two disks on each side of the Sort/Merge Node) provides slightly better throughput than using one disk for the input and output files and three for the sorted segments. In fact, this arrangement provides the best throughput of all the arrangements we have looked at so far.

However, by adding the second disk for the original and sorted files we have increased the maximum sequential throughput of the storage node to approximately 1140 KBytes/sec [Dud92]. Using two disks for the sorted segments also means the limit on i/o throughput for the segments ought to be close to link speed. Unfortunately, Table 4.17 shows that the throughput for both phases of the sort is not even greater than the limit imposed by the bandwidth of a single disk. We have reached a cpu-bottleneck. This is confirmed by the times for all the file sizes in Table 4.17, where the times for both phases are consistently dominated by the cpu time.

Table 4.17: Sort Times and Throughput Rates Using One Double-Disk storage node and Two Single-Disk storage nodes

File Size	50MB		100MB		200MB	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**104.7**	**477.6**	**203.3**	**491.9**	**401.2**	**498.5**
cpu	86.1		168.6		334.1	
input	9.8		17.0		31.9	
output	8.8		17.7		35.2	
Merge Phase	**69.2**	**722.5**	**152.3**	**656.6**	**322.3**	**620.5**
cpu	66.2		146.9		316.8	
input	2.8		5.3		4.9	
output	0.2		0.1		0.6	
Total	**173.9**	**287.5**	**355.6**	**281.2**	**723.5**	**276.4**
Total cpu	152.3		315.5		650.9	
Total input	12.6		22.3		36.8	
Total output	9.0		17.8		35.8	

4.7 Summary

In Figure 4.2 we summarize the results presented in this chapter for the 100MB files consisting of 128-byte records. This Figure shows two histograms: one for the times taken by the various configurations and one for the corresponding rates. In this Figure $a \times b + c \times d$ refers to the number and type of storage nodes used in a particular configuration. The factor on the left of these expressions ($a \times b$) refers to the storage node used for the input and sorted files; the factor on the right ($c \times d$) refers to the storage node(s) used for the sorted segments. The leftmost operand of these factors refers to the number of TASS storage nodes being used whereas the righthand operand refers to the number of disks present in the storage node(s). So, for example, $1 \times 2 + 2 \times 1$ refers to the configuration in which one storage node with 2 disks was used for both the input and sorted files and 2 storage nodes each with one disk attached were used for the sorted segments. The TIMES histogram shows the total time taken by each sub-phase of the external sort for each of the configurations discussed in this chapter. The three bars on the left show the cpu, input, and output

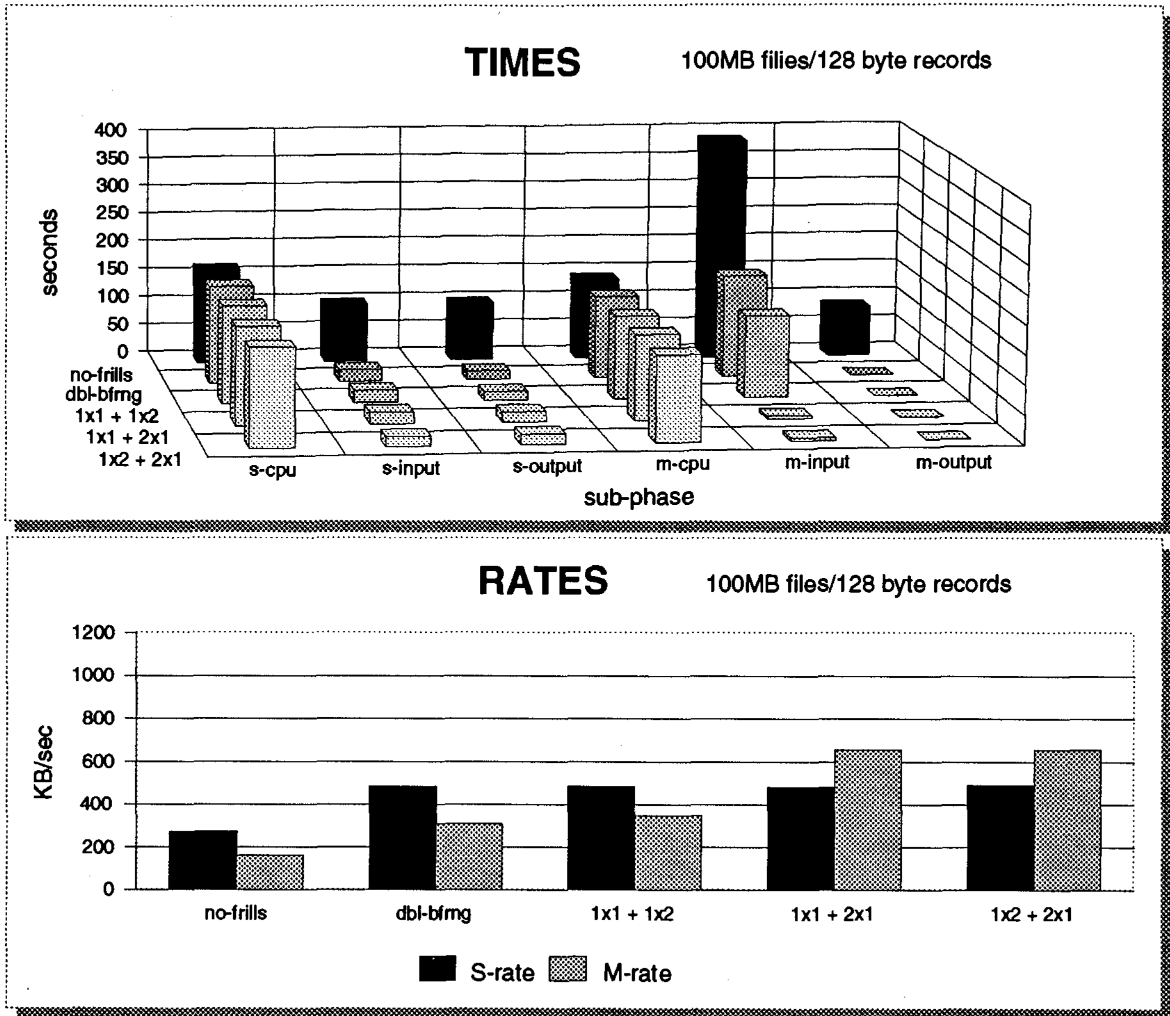


Figure 4.2: Overview of 100MB data

times for the Sort Phase, and the three bars on the right show the corresponding times for the Merge Phase. The RATES histogram shows the overall Sort Phase and Merge Phase rates for the same configurations.

From the TIMES histogram in Figure 4.2 it can be seen that the first implementations discussed in this chapter are i/o-bound but that by the end of the chapter the external sort has become cpu-bound. Indeed, the RATES histogram illustrates once again that we have not yet achieved, for either phase, the ceiling (approximately 1140 KB/sec) imposed by the 2-disk storage node on which the input and sorted files are being stored.

There are a number of ways to ameliorate the cpu-bottleneck we have now encountered. The most obvious, and simplest, way is to increase the speed of the cpu. This is not unreasonable as there are now many cpu's which are much faster than the T800 transputer - which performs at approximately 10 MIPS. Modern risc chips commercially available today perform at 60 to 100 MIPS. Given such fast cpus it is easy to see that the bottleneck will once again become the bandwidth of the i/o subsystem.

The next generation of transputers, the T9000, will be ten times faster than the T800, both in cpu and in link speed [Pou91].[7] It's not unreasonable to speculate that this kind of performance would allow a single T9000 to perform an external sort on a 200MB file in 1/10 the time given in Table 4.17. Even if the sequential i/o throughput of a single disk does not increase significantly over that of the disks we have used it ought always to be possible, using multiple transputers and disks, to make the throughput of a single storage node reach the limit imposed by the throughput of a link. For example, if the throughput of a single disk were 1/9 that of a link we could attach three disks to one transputer to form a "Sub-"storage node. The sequential throughput of such a Sub-storage node would very likely be close to three times that of a single disk. We could then attach three of these Sub-storage nodes to another transputer thus forming a single storage node with nearly nine times the sequential throughput of a single disk. Such a storage node ought to enable a T9000 to perform a 200MB external sort at nearly ten times the speed given in Table 4.17.

At this point we would once again be faced with a cpu-bottleneck and we would again have two choices. We could either use a still faster cpu or we could perform the external sort in parallel. This second choice is discussed in the next chapter.

[7]The number of links on a T9000 will still be four.

Chapter 5

External Sorting With Multiple CPUs

When the cpu is the bottleneck in an external sort additional cpus may be used to speed up the sort. Based on the results of the previous chapter, however, for files composed of 128-byte records not very many transputers can be used before the bottleneck becomes either the speed of the disks or the speed of the links on the transputers. At this point the only way to further improve the sort times is to split the file to be sorted and/or the sorted file onto multiple disks. That is, the i/o must also be parallelized.

In this chapter we examine external sorts which are increasingly parallel. We begin with a discussion of ways to parallelize single-input single-output external sorts in which both the original and the sorted files reside on a single disk. We then recount a low-communication algorithm in which the input file is to begin on multiple disks but where the sorted file is to be sent to a single location. Finally, we discuss fully parallel external sorts in which both the original and the sorted files are stored on multiple disks.

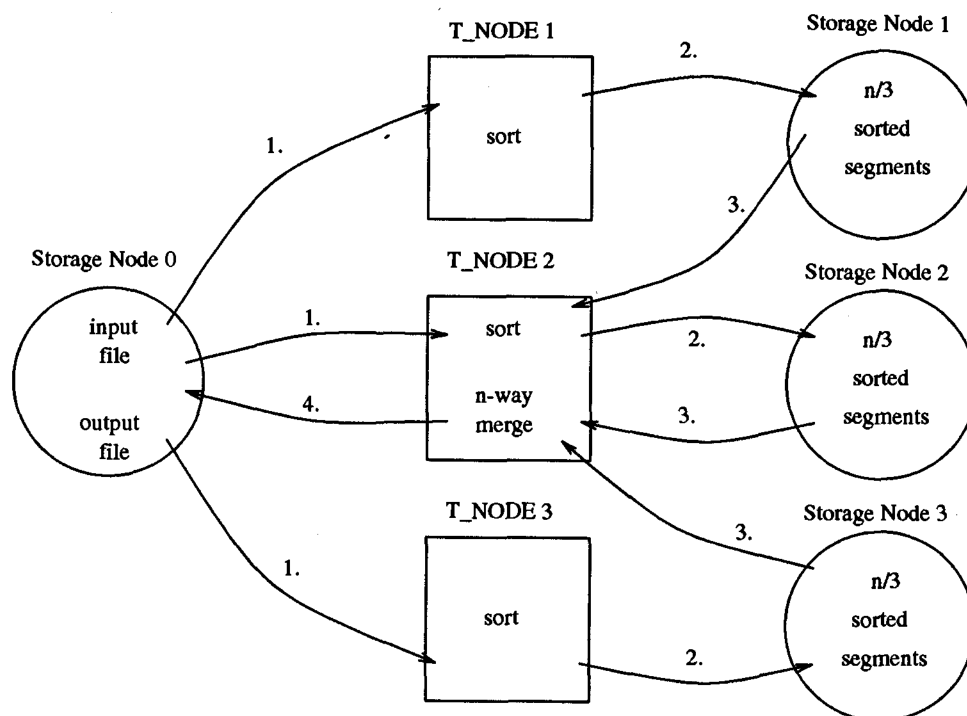


Figure 5.1: Parallellizing the Sort Phase

5.1 Single-Input Single-Output

5.1.1 Sort Phase

We first look at a method for overcoming the cpu-bottleneck in the Sort Phase. We saw in the previous chapter that, for 128-byte records, a single transputer node cannot create sorted segments as fast as the data for these segments can be read from a single disk. A simple way to use extra transputer nodes to speed up the creation of sorted segments is shown in Figure 5.1, which illustrates the use of three T_NODEs for this purpose. The same two phase external sort is used here as has been used in all the experiments discussed in this thesis so far. T_NODEs 1 and 3 are only active during the Sort Phase and so will henceforth be referred to as Sort Nodes. T_NODE 2 is active during both the Sort and the Merge Phases and so will alternately be referred to as a Sort Node or the Merge Node.

Each Sort Node creates, in parallel with the other Sort Nodes, sorted segments

from its input using the Replacement Selection algorithm discussed in Section 2.3. When a Sort Node has finished creating its segments it notifies the Merge Node of this. In our implementation this is accomplished by having the Sort Node tell the Merge Node the number and disk location of the segments that the Sort Node has just created. After having received this notification from all the Sort Nodes the Merge Node merges all the segments into a sorted file - just as in section 4.5.2.

The file to be sorted is stored on storage node 0 and during the Sort Phase each of the T_NODEs attached to storage node 0 reads part of this file. The ability provided by TASS to attach upto three application nodes to a single storage node provides an easy way of allowing this. However, there is one complication encountered when using this simple method. The input file must be read sequentially[1] from beginning to end to obtain the maximum throughput from the disk. Care must be taken when using multiple T_NODEs for the Sort Phase to avoid introducing extraneous seeks while reading the file to be sorted. We can't partition the input file into multiple contiguous portions and assign one portion to each of the Sort Nodes because all of these nodes will be reading parts of their portions concurrently resulting in unnecessary seeking between the portions.

There are a number of ways to overcome this problem, including explicit synchronization among the Sort Nodes. We chose to insert a "pipe" T_NODE between the storage node which contains the input file and the Sort Nodes. This pipe T_NODE implicitly forces the requests by the Sort Nodes to be satisfied in sequential order. With this method, during the Sort Phase the pipe transputer reads the unsorted file in sequential order and pipes this data through to each of the Sort Nodes. One reason for choosing this implicit method is that the Merge Node will not have enough links to support explicit synchronization among the Sort Nodes when three Sort Nodes are used, as can be seen in Figure 5.1 in which all four links of the Merge Node are already in use. A second reason will become apparent shortly.

We have chosen to supply each of the Sort Nodes with its own storage node for storage of the sorted segments in part because we saw in the last chapter that using multiple disks for segments results in improved performance for the Merge Phase.

[1]As has been the case in all the tests to this point.

Additionally, it avoids the extraneous seek problem for output in the Sort Phase that we have just discussed for input in this phase.

Based on the data from section 4.5.2, where we saw Sort Phase throughput rates of well over 365 Kb/sec when sorting 128-byte records, we can reasonably conjecture that using just 2 Sort Nodes in this way would enable the creation of sorted segments with this size of record at or near 730KB/sec - the peak input rate achievable when using a single storage node with a single disk. Indeed, with two Sort Nodes we expect that the rate at which the input file could be read from a single disk would again become the bottleneck. Therefore, we use a storage node with 2 disks attached to it to allow the read rate of the input file to keep up with the sort rate of the Sort Nodes. This leaves us with two disks for the storage of the sorted segments. Thus, we implemented an external sort which uses two Sort Nodes in the Sort Phase, each with its own storage node for storing the segments it creates.

The results of the runs on our standard sample input files are presented in Table 5.1. The Table to which the data in Table 5.1 are most comparable is Table 4.17 which was obtained using a configuration almost identical to that used for Table 5.1. The only difference between the configurations for the two Tables is the addition of the single extra transputer during the Sort Phase in obtaining Table 5.1. As expected, the throughput figures when two transputers are used during the Sort Phase are almost twice as high as the corresponding figures when only one Sort Node is used. For example, the Sort Phase throughput for the 200MB files in Table 4.17 is 498.5 Kb/sec whereas the addition of the second Sort Node increases this throughput to 968.1 Kb/sec.

The Sort Phase throughput figures in Table 5.1 indicate that there is still room for the use of a third Sort Node, since none of the Sort Phase throughputs are as large as the rate at which the input file can be read from the single storage node with two disks. However, the addition of a third node was not attempted for the following reasons. Ideally, each Sort Node would have its own disk on which to store the sorted segments it creates. As all four of the disks available to us are already being used in the configuration above this precludes assigning another disk to another Sort Node. Attaching two Sort Nodes to a single disk for segments would cause the introduction

of extraneous seeks with an accompanying decrease in performance of this disk. This could have been circumvented with the addition of another "pipe" node between the two Sort Nodes and the single disk for the segments. This pipe node would have to merge pairs of segments as it received them from the two Sort Nodes to which it was attached. The pipe node could then write segments out to the single disk in a strictly sequential manner. This was deemed to be more trouble than it was worth.[2]

There is a small, but significant, degradation in the throughput of the Merge Phase from Table 4.17 to Table 5.1. For example, the throughput of the Merge Phase for the 200MB file in Table 4.17 is 620.5 Kb/sec whereas the corresponding throughput in Table 5.1 is 582.9 Kb/sec. The explanation for this is twofold. Firstly, while producing Table 5.1 it was found that two Sort Nodes sorting a given file create, on average, slightly more sorted segments than a single Sort Node would. Secondly, the introduction of a second Sort Node required new object code in the Merge Node. Taken together, these two factors decreased the size of the largest buffers usable during the Merge Phase and we have previously discussed how large buffers are instrumental to Merge Phase performance.

So, it can be seen from Table 5.1 that it is possible to match the rate at which the input file can be read with the rate at which sorted segments can be created. It is not unreasonable to surmise that even if the throughput of the cpu in the Sort Phase is much slower than the input rate of the file to be sorted (as would occur in our environment if smaller than 128-byte records were being sorted) still more Sort Nodes could efficiently be used to improve the effective cpu throughput of the Sort Phase. A larger binary or ternary tree of transputer nodes could be constructed and used in the following way: the single storage node for the input file would be placed at the root of the tree, the interior nodes would all act as "pipe" nodes, and the leaves of the tree would be the Sort Nodes. The interior (pipe) nodes in the tree would pass data from the single input storage node at the root node throught to the leaf (Sort) nodes. Each Sort Node would create sorted segments from the data it recieves from its parent and store these segments on its own local storage node, as discussed above.

[2]Programming a transputer (the root node in this case) in which all four links are being used, leaving none available for debugging, is a formidable task!

Table 5.1: Sort Times and Throughput Rates When Using Multiple T-Nodes for Creation of Sorted Segments

File Size	50MB		100MB		200MB	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**54.3**	**920.8**	**105.1**	**951.5**	**206.6**	**968.1**
cpu	45.3		87.3		171.7	
input	4.8		9.7		18.5	
output	4.2		8.1		16.4	
Merge Phase	**71.8**	**696.4**	**156.4**	**639.4**	**343.1**	**582.9**
cpu	68.5		152.5		335.6	
input	3.2		3.3		7.0	
output	0.1		0.2		0.5	
Total	**126.1**	**396.5**	**261.5**	**382.4**	**549.7**	**363.8**
Total cpu	113.8		239.8		507.3	
Total input	8.0		13.0		25.5	
Total output	4.3		8.3		16.9	

Note that, in this configuration there would be a minor delay between the first request for data by a leaf node and the actual arrival of this data since the data must filter down through the tree from the root. However, each path from the root to a leaf would behave as a pipeline and so this delay would only happen on the first input request.

The total number of nodes required in the tree would be determined by the input rate of the file from the storage node at the root and the size of the records in the file being sorted. Since the maximum input rate is essentially fixed at the speed of a link, the only factor affecting the number of Sort Nodes required would be the size of the records being sorted. Enough nodes would be needed to make the total cpu throughput of the Sort Phase match the input rate through the link. The data in Table 5.1 shows that, when 128-byte records are being sorted, certainly not more than three leaf nodes would be required to be able to create sorted segments at link speed. When the file is composed of 8-byte records the data in Table 4.4 shows us that we can create sorted segments at a rate of 32 Kb/sec with a single transputer and so no more than 38 leaf nodes would be required to create sorted segments at link speed

(approximately 1200 Kb/sec). A ternary tree with this number of leaves could be constructed with as little as 13 internal nodes. Any number of leaf nodes beyond this maximum (38) could not improve the throughput of the Sort Phase.

Note that when the appropriate number (as described above) of Sort Nodes are used then the link connecting the root to the input storage node is utilized to capacity and the cpu's at the leaves are utilized to 100% capacity. However, the cpu's of the interior nodes are essentially idle and the disks used for storing the sorted segments are underutilized. Thiese underutilizations are in direct proportion to the number of Sort (leaf) Nodes. Both of these underutilizations can be alleviated to some degree - though we do not attempt these modifications in our implementations.

The underutilization of the interior cpu's could be alleviated by making all the interior nodes and the leaf nodes behave in a way which is slightly different from the behaviour of the leaf nodes described above. All the nodes in the tree could be made to sort some of the unsorted data which passes through to them and merge the data they sort with the sorted data received from their parent.[3]

The underutilization of the disks at the leaves could be alleviated through the addition of merger "pipes" as described above. In this way we would need less disks for the storage of the sorted segments. However, reducing the number of disks for the segments can seriously undermine the throughput of the Merge Phase input. Therefore, one would have to balance the efficiency of the disk utilization in the Sort Phase with maximum input throughput achievable during the Merge Phase.

Finally, let us point out again that the throughput of the Sort Phase does not decrease as the size of the file increases. So the maximum numbers of transputers utilizable as discussed above will not increase, no matter how large the file being sorted.

5.1.2 Merge Phase

We still have not done anything to improve the throughput of the Merge Phase since in the configuration of the previous section there is still only a single transputer

[3]The root node has no parent from which to receive sorted data and so it would not do any merging.

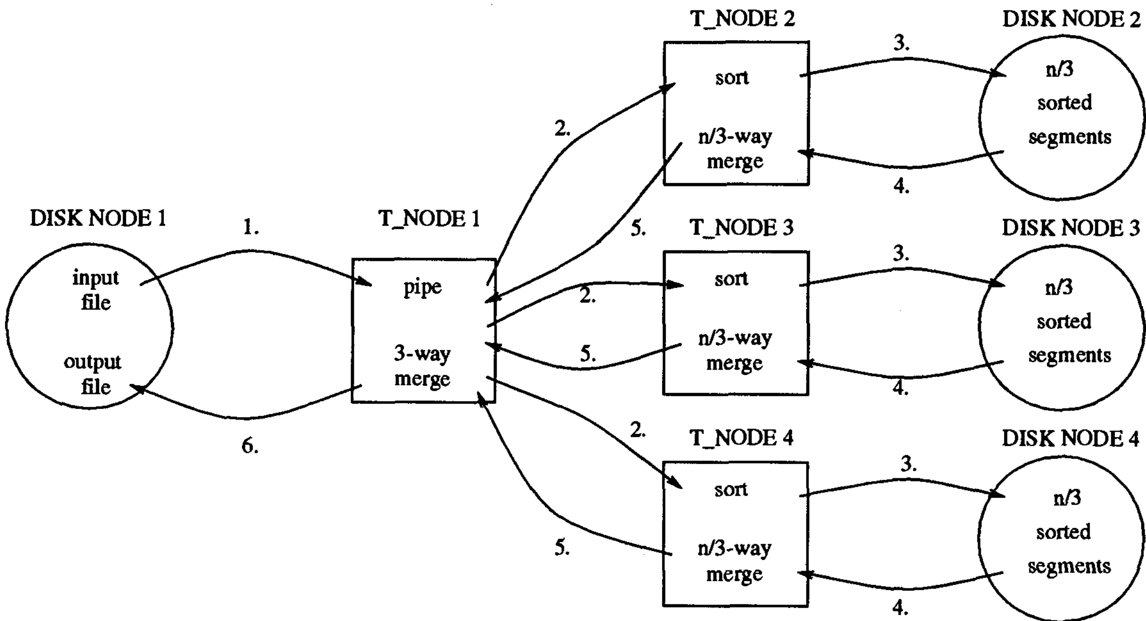


Figure 5.2: Fastsort

node performing the merge. We can overcome this disadvantage by using during the Merge Phase the "pipe" transputer(s) we introduced in the previous section. This configuration is illustrated in Figure 5.2. Now, during the Merge Phase each of T_NODEs 2, 3, and 4 merges the segments on its local disk and then each of these nodes sends one large sorted segment back to T_NODE 1. If multi-pass merges are required to optimally merge the segments on any local disk then the initial passes are performed in parallel on each of the disks requiring them before the final single segment created for each auxiliary storage node is passed to T_NODE 1. T_NODE 1 merges the 3 segments it receives from the other nodes into the sorted file which it stores on DISK NODE 1. Notice that, by having each of T_NODES 2, 3, and 4 send their final merged segments to T_NODE 1 as those segments are created, a pipeline is formed increasing still further the amount of parallelism during in the Merge Phase. Thus, all the transputer nodes are active during the Merge Phase. This single-input single-output parallel external sort is known as "Fastsort"[STG+90]. Using multiple nodes in this way during the Merge Phase is especially important since, as can be seen from Table 5.1, the Merge Phase cpu time is now the dominant contributor to

the total sort time.

Unfortunately, we cannot present data for a complete implementation of Fastsort which is directly comparable with the data presented in Table 5.1. This is because one of the four disks which we were using to obtain data broke before we could complete the experiments. The data in Table 5.1 was obtained using one node with two disks for the input and sorted files, and 2 nodes each with a single disk for storing the sorted segments. Given that one of these four disks must disappear the best choice is to remove one of the two that were used for storing the input and sorted files because removing one of the disks used for the segments would impair the performance of the Merge Phase input - the most expense of the i/o-costs for the external sort.

The disappearance of one of the two disks for the input file and sorted file has several ramifications. Firstly, the largest file that can be sorted is now once again approximately 100MB. More importantly, the maximum i/o rate of the disk for the input and sorted files is now approximately 700KB/sec. Actually, the situation is even worse than this. The actual maximum throughput for both phases of the sort is now limited to that of the slowest of our three remaining disks: approximately 590KB/sec. Since from Table 5.1 we can see that the Merge Phase for the 100MB files is already operating past this limit we would actually expect to see a decrease in the Merge Phase throughput in our implementation of Fastsort when operating on 128-byte records. However, by decreasing the size of the records we can make the sort more cpu-bound and so we will attempt to illustrate the effects of a complete implementation of Fastsort by sorting records of 64-bytes instead of 128-bytes.

In Table 5.2 we present data for a full implementation of Fastsort using a binary tree with two leaf nodes. The data in this Table are for files of 50MB and 100MB composed of 64-byte records. We also include in this Table comparable data for the fastest configuration from Chapter 4 still available to us with only three working disks. This configuration is that discussed in section 4.5.2 with one disk for the input and sorted files and two storage nodes for the sorted segments; this configuration is illustrated in Figure 4.1(ii). As in Table 5.1 we see from Table 5.2 that Fastsort has nearly doubled the Sort Phase throughput, though with the smaller record size there is need for substantially more than two transputer nodes to be able to create sorted

Table 5.2: Sort Times and Throughput Rates for a Fastsort Implementation

File Size	50 MBytes				100 MBytes			
Technique	1x1 + 2x1		Fastsort		1x1 + 2x1		Fastsort	
	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec	secs	Kb/sec
Sort Phase	**200**	**250**	**107**	**467**	**391**	**256**	**207**	**483**
cpu	175		90		344		175	
input	13.1		11.4		23.4		17.5	
output	11.7		5.8		23.8		11.6	
Merge Phase	**104**	**481**	**86.1**	**581**	**232**	**431**	**174**	**575**
cpu	101		81.6		226		136	
input	2.9		4.4		5.5		36.9	
output	0.2		0.1		0.2		0.2	
Total	**304**	**164**	**193**	**259**	**623**	**161**	**381**	**262**
Total cpu	276		171.6		570		311	
Total input	16.0		15.8		28.9		54.4	
Total output	11.9		5.9		24.0		11.8	

segments at link speeds. More importantly, notice that for both sizes of files Fastsort improves the throughput of the Merge Phase almost to the limit imposed by our slowest disk. Unfortunately, for the 100MB file size the Fastsort Merge Phase input time goes up significantly over the $1 \times 1 + 2 \times 1$ implementation. The explanation for this is that each merging transputer node now has only one disk for sorted segments attached to it and we saw from Chapter 4 that at least two are needed for optimal performance.

What would be the maximum limit to the throughput of the Merge Phase of Fastsort if this algorithm were implemented using T800 transputers? There are (at least) two possible bottlenecks to the Merge Phase: one is the rate at which the sorted file can be written to the disk(s) on which the file is to be stored (ie. the Merge Phase is i/o-bound); the other potential bottleneck is the rate at which the root node of the tree on which the Fastsort is implemented can merge the segments it receives from its children (ie. the Merge Phase is cpu-bound). So the maximum throughput of the Merge Phase of a Fastsort on transputers will be limited to the lower of these two rates.

Table 5.3: Maximum 2-way and 3-way Merge Rates (integer keys)

Record Size (bytes)	2-way Merge (Kb/sec)	3-way Merge (Kb/sec)
8	1234	848
16	2136	1458
32	3588	2563
64	5548	4201
128	7702	6272
256	9524	8375

We know that if the sorted file is to be stored on a single storage node then the maximum rate for ouput of this file is slightly less than link speed (ie. 1200 Kb/sec). But what is the rate at which the root node can merge the segments it receives from its children? Because a transputer has four links Fastsort can be implemented on either a binary or a ternary tree. (The configuration in Figure 5.2 illustrates a ternary tree.) Thus, the root node will have to perform either a 2-way or a 3-way merge.[4] In Table 5.3 we list, for records of various sizes, the maximum 2-way and 3-way merge rates achievable by a single transputer. The merge is based on an integer key, consistent with the keys we have been using so far. For this experiment the records being merged began and ended the merge in RAM. In an actual implementation of Fastsort the records being merged would arrive through links from the children of the root. However, with double-buffering the merge in a Fastsort ought to be able to be performed in parallel with this input. Thus, we anticipate that there would be only a slight degradation in the merge performance in an actual Fastsort from the data presented in Table 5.3.

From Table 5.3 it can be seen that for 2-way merges, even for 8-byte records, a single transputer can merge records faster than they can be written out through a single link. A T800 cannot perform a 3-way merge on 8-byte records as fast as these records could be output to disk and so for optimal Fastsort performance on 8-byte records a binary tree would have to be used. However, for larger records even a 3-way merge can be performed by a T800 more quickly than the maximum output rate. Therefore, we may surmise that, given a sufficiently deep merge tree, the maximum

[4]In general, each interior node must perform the same order merge as the root node.

rate at which the Merge Phase of a Fastsort can be performed is limited not by the speed of the cpu but by the speed of a link.

For the Sort Phase the number of nodes needed to enable Fastsort to create sorted segments at link speed is independent of the size of the file. However, as mentioned in Chapter 2, the Merge Phase is $O(N \log nsg)$ (where N is the total number of records in the file and nsg is the total number of sorted segments) and so the number of nodes needed to merge at link speed will grow as the size of the file grows. As the number of nodes required to maintain maximum Merge Phase speed grows beyond the number needed to maintain maximum Sort Phase speed some of the leaf nodes will become underutilized during the Sort Phase. This could be offset by increasing the amount of RAM at each leaf node.[5] Recall that the time needed to create a single sorted segment is $O(y \log y)$ where y is the number of records which can fit into available RAM at one time. Therefore, increasing the size of RAM slows down the throughput of the Sort Phase.

Finally, note that since both the cpu speed and the link speed of the T9000 are expected to increase by a factor of 10 over the corresponding speeds in the T800 the number of transputers required to maintain a balance between the link speed and the sort speed will be unaffected.

5.2 A Low Communication Sort (Multiple-Input Single-Output)

Lorie and Young [LY89] describe a backend database machine and a low communication parallel sort in which the unsorted file begins distributed across sub-nodes but where the sorted file ends up at one particular node (the host). Thus, their algorithm is multiple-input single-output. The algorithm they give avoids the explicit merging bottleneck of Fastsort and the architecture of their machine is such that the return of the sorted file to the host imposes no bottleneck - up to a point limited by

[5]No increase would be needed for the interior nodes. In fact, very little memory is needed for the interior nodes.

the hardware. Thus their algorithm is "fully" parallel. A more detailed description follows.

The backend database machine Lorie and Young use is composed of one host site and n subsites (nodes). Each node has its own local processor, memory, and disk. A network connects each node to every other node, thus providing n-to-n communication among nodes. In addition, each node has a separate communication link to the host site (n-to-1 network). This n-to-1 network is practically identical to the Hennessy and Patterson example discussed in [PH90]. It is this last feature of the architecture which enables the sorted file (which will be spread among the nodes near the end of the sort) to be passed back to the host from all the nodes "in parallel" - limited only by the input bandwidth of the host. Lorie and Young's experimental results indicate that the return of the sorted records to the host is typically responsible for most of the response time in parallel external sorts. Therefore, the input bandwidth of the host is crucial for high performance in parallel single-output external sorting.

The algorithm itself consists of four phases: 1) local sort, 2) synchronization, 3) merge, and 4) return of sorted records to host. The unsorted file is initally spread across the disks of the nodes. In the first phase of the algorithm each node sorts the part of the file which resides on its local disk. Simultaneously, it creates an index list and a distribution table. The index list indicates the value of the lowest key on a sorted disk block and is used to speed up the merge phase. The distribution table contains information on the distribution of keys at that node.

In the second phase, each node sends its distribution table to a synchronization site (some previously agreed upon node) which merges these tables into a global distribution table. This table is then returned to each of the nodes. Each node is assigned ownership of a unique range of key values from the global distribution table and the next phase can begin.

At this point each node possesses a number of records which are owned by other nodes. A node can determine to which node each of its records belongs through the global distribution table. The merge phase consists of each node sending the *key* of every record which it doesn't own to the node which does own that record. Along with each of these keys the identity of the sender is also sent. Concurrently with

sending <key, id> pairs each node receives and merges the pairs being sent to it from the other nodes, producing a stream of node ids. The stream on each node is thus ordered such that reading records from nodes in the order in which node ids appear in the stream would produce, in sorted order, the records which belong to that stream's node.

Since it is assumed that keys are much smaller than the record to which they belong, the amount of communication is reduced by sending only keys - hence this is a 'low communication' algorithm. Another optimization in the merge phase is to stagger the sending of keys to the nodes so that not all keys are sent to the first node at once, and then all keys sent to the second node at once, etc.

Finally, in the last phase, each node first sends to the host site the stream of *node ids* it created in the merge phase. The host site concatenates these streams together in the proper order to form a final stream. By proper order, we mean that the stream from the node which owns the group of smallest keys is placed first into the final stream, the stream from the node which owns the group of second smallest keys is placed second in the final stream, and so on. After all the nodes have sent their streams to the host each node begins sending its records to the host in the order the records appear on its local disk. Recall that these records have not been moved since the end of the initial sort phase. The host uses the final stream of node ids to decide from which node it should read the next record, thus producing the sorted file.

The novel feature of this algorithm is the method used to reduce the communication costs. We haven't implemented it because it is still a single-output sort, and as such limited to the maximum rate at which data can be output to a single storage node. However, we have discussed it because it provides an example of an external sort which avoids the input bottleneck by starting with a file which is distributed across all of the subsites. Lorie and Young do not provide a straightforward measurement of their algorithm's performance and so it is hard to compare the performance of their algorithm with that of Faststort.

5.3 Multiple-Input Multiple-Output

We have seen that, given a small number of transputers and auxiliary disks, the speed of a link becomes a bottleneck when trying to create sorted segments from data stored on a single storage node. Similarly, the link speed can become a bottleneck when trying to write merged segments (ie. the sorted file) to a single storage node. Clearly, to overcome the link speed bottleneck we must spread both the input and the sorted files among multiple storage nodes thus enabling multiple links to be used in parallel. There has been much study on parallel and distributed *internal* sorts in which the data to be sorted begins and ends spread out among the RAM attached to each processor used in the sort. In this section we briefly list some of these methods and comment on their suitability towards adaptation for external sorting.

Let us define a *processing node* as a collection of transputers and storage nodes which together can perform a single-input single-output external sort at a rate limited by the speed of a single transputer link. Let T_s be the total time required for the sort on a single processing node and T_p be the total time required for the sort using pn processing nodes. The *speedup*, S, obtained through the use of multiple processing nodes is defined as the ratio T_s/T_p. Ideally, by dividing the input and output files among pn processing nodes we would like to see a speedup of pn. In other words, we would like the *efficiency* of the external sort on the parallel machine to be as close to 100% as possible, where efficiency is defined as S/pn. However, a number of factors can conspire to prevent this, not the least of which is the cost of communication required among the multiple storage nodes. Another limiting factor which will be of prime concern to us while examining the applicability of distributed and parallel internal sorting algorithms towards external sorting will be the number of passes (ie. reads or writes) a particular sort must make over the data stored local to each processing node. If the input and output files are divided among pn processing nodes but an algorithm requires pn passes over all the data at each of the nodes then we have gained very little, if anything at all. So, we will be most interested in those internal algorithms which use the fewest number of passes.

Luk *et al* have performed empirical studies on the performance of a variety of

distributed internal sorting methods in a LAN environment and dicuss the results in [LPS87], [LPW88], and [LL89]. The methods they studied included a Block Odd-Even Sort based on a two-way merge-split discussed in [BDHM84], a Block Bitonic Sort, a Distributed Quicksort, and a Selection Sort (adapted from the Distributed Sorting Algorithm given in [RSS85]) which is very similar to the synchronization and merge phases of Lorie and Young's algorithm. We will not discuss the first of these two algorithms in detail here for the following reasons: The Block Odd-Even internal Sort requires each processing node to perform $O(pn)$ passes over the data local to that node. As all of these passes access the local data sequentially we could access the data on a disk in much the same way as if all this data were in RAM, albeit much more slowly. If we were to use the Block Odd-Even algorithm to perform an external sort $O(pn)$ passes would negate much of the benefit of splitting the input file across pn processing nodes, as discussed in the previous paragraph. So the Block Odd-Even algorithm does not appear to be a good candidate for external sorting. The Block Bitonic Sort requires $O((\log pn)(\log pn))$ sequential passes over the data at each processing node. While this is better than the Block Odd-Even Sort the Distributed Quicksort is better still as it requires only $O(\log pn)$ sequential passes. Finally, strictly from the point of view of the number of passes required, the Selection Sort is virtually ideal since it requires only one more sequential pass than the number required to sort the data locally at each node. The Distributed Quicksort and the Selection Sort are essentially identical to their parallel counterparts; we will examine the parallel versions of these algorithms in more detail shortly.

For these distributed algorithms performing internal sorts Luk *et al* concluded that neither the cost of the local processing nor of the communication between nodes should be ignored in theoretical analyses of the various sorting algorithms' performance. It seems reasonable that the cost of communication should not be ignored when any of these algorithms is adapted for external sorting. However, the theoretical analyses they provide in their studies are applicable towards the hardware environment they used for their experiments, namely a local area network using an ethernet. This is substantially different than the hardware environment we are examining, namely networks of transputer statically reconfigurable through hardware crossbar switches. So,

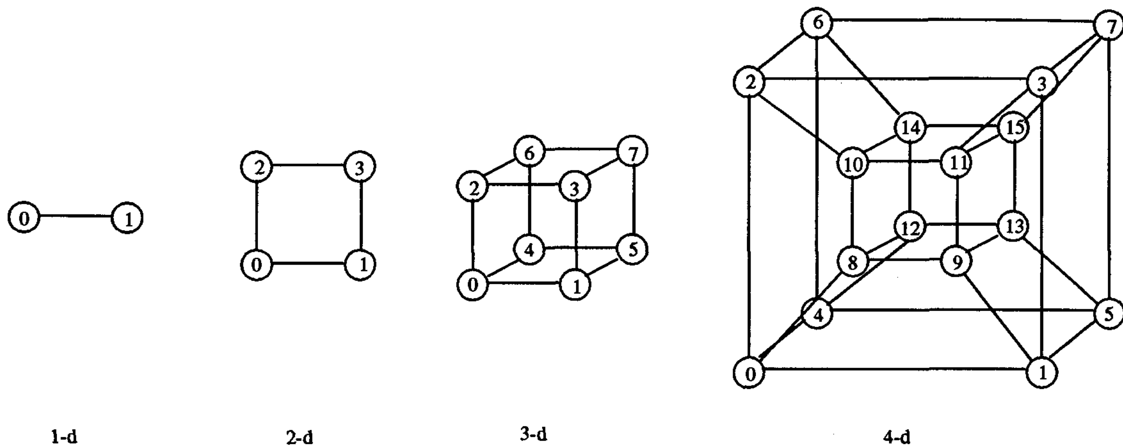


Figure 5.3: Hypercubes

we cannot simply transfer Luk *et al*'s communication cost analyses to our environment. Finally, for large numbers of nodes they found that the algorithm with the best performance was the distributed quicksort.

[Fox88] discusses methods for performing parallel internal sorts on a hypercube. We will assume that the reader is already acquainted with hypercubes, some examples of which are shown in Figure 5.3. Note, in particular, that a hypercube of dimension d has 2^d processing nodes and that every node is connected to d other nodes. When using transputers to construct a hypercube with $d > 3$ each processing node must have more than one transputer. This is because each transputer has only four links, at least one of which will be used to access a storage node. This leaves three links on a single transputer for connecting a node to other nodes in the hypercube. The extra links needed for higher dimensions can be obtained by putting extra transputers at each node.

The three parallel internal sorts which [Fox88] discusses are Bitonic, Parallel Shellsort, and Parallel Quicksort. Again, we are interested in the adaptability of these algorithms towards external sorting. We can immediately dismiss the Bitonic algorithm given in [Fox88] because of its extreme inefficiency - that is the hypercube must be allowed to grow without bound to work very well on small lists.

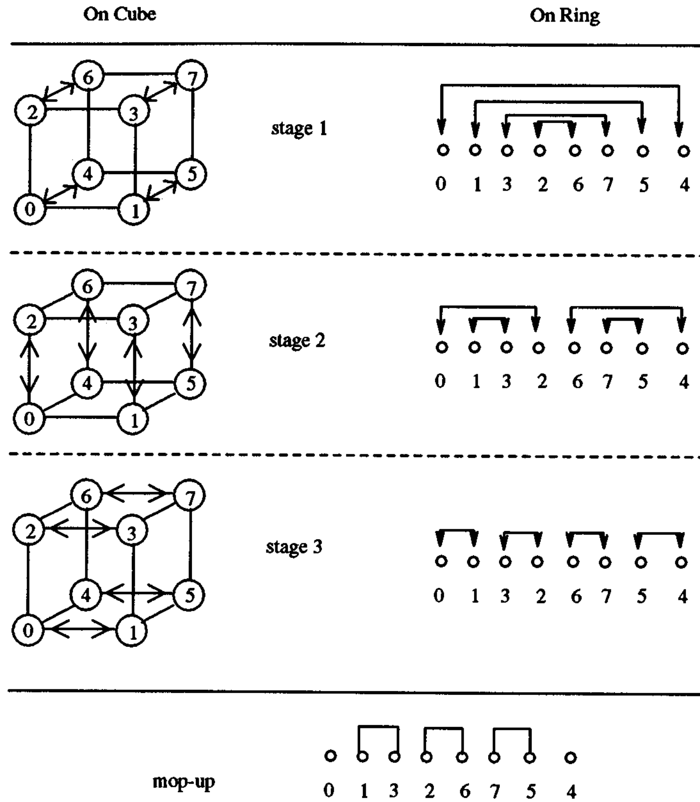


Figure 5.4: Parallel Shellsort

The Parallel Shellsort shows more promise for use with large files. For this algorithm we will follow the convention used by Fox *et al* and map the nodes of a hypercube onto a ring using the *gridmap* utility given in [Fox88].[6] Through this utility each node in the hypercube is given an integer ring position. The file will be sorted across the ring when all the records at ring position 0 are in sorted order locally and they precede the records at ring position 1, which in turn are in sorted order locally and precede the records at ring position 2, etc.

The Parallel Shellsort is composed of two phases: an initial phase reminiscent of the serial Shellsort followed by a mopping-up phase. There are as many stages in the initial phase as there are dimensions in the hypercube being used for the sort. In early stages of the initial phase data make large jumps towards their final

[6] The actual details of this procedure are not needed to understand the remainder of this discussion.

destination and as the initial phase progresses the jumps become smaller and smaller. This is illustrated for a 3-dimensional hypercube in Figure 5.4 which is essentially the same figure used by [Fox88] to illustrate this sort. Each stage of the initial phase requires one pass over all the data in each node of the hypercube. Therefore, since the dimension d of the hypercube equals $\log pn$, the initial phase requires $\log pn$ passes. Unfortunately, [Fox88] shows that the mopping-up phase may require as many as $2^d - 1$ steps, the length of the longest path through the ring that a record of data may have to travel during the mop-up phase. However, [Fox88] also shows that the probability of requiring such a large number of passes in the mopping-up phase decreases as the amount of data stored on each node grows. This is good news for external sorts as in this case we would expect large amounts of data on each node. Their experimental results on internal sorts showed that, for the largest lists (2048Kb), the mop-up phase consumed 10% to 20% of the total time.

Most interestingly, from an external sorting point of view, Fox *et al*'s experimental results showed that the efficiency of the Parallel Shellsort increased as the amount of data to be sorted grew - at least on internal sorts. It should also be noted that their experimental results show a decrease in efficiency for this algorithm as the dimension of the hypercube increases. A big advantage of the Parallel Shellsort is that it doesn't require *a priori* knowledge of the statistical distribution of the data in the file - a major shortcoming of the Parallel Quicksort which we describe next.

For the Parallel Quicksort the file will be sorted across the hypercube when all the records at node 0 are in sorted order locally and precede the records at node 1, which in turn are in sorted order locally and precede the records at node 2, etc. The Parallel Quicksort is a straightforward extension of the serial Quicksort. The only difference between the serial and the parallel version is that in the parallel version the first few partitions partition the data among equal sized groups of processing nodes. In the case of a hypercube these equal sized groups can be conveniently split across dimensions of the hypercube, so that the first partition splits the file across one dimension of the hypercube (the nodes on either side of this split form two "sub"-hypercubes), the next partition splits the file across another dimension (thus forming four "sub-sub"-hypercubes), and so on for each dimension of the hypercube (until

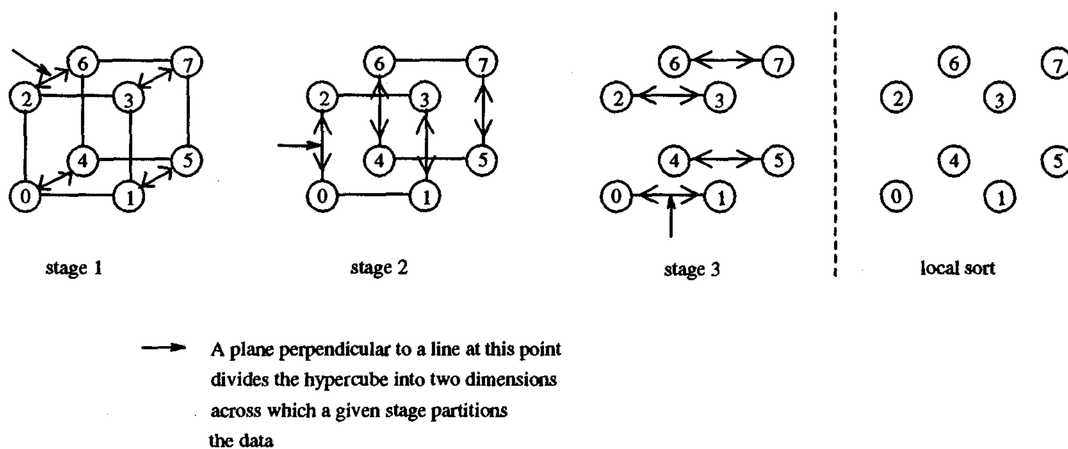


Figure 5.5: Parallel Quicksort

we have 2^d 0-dimensional hypercubes). Note that all the sub-hypercubes which exist (logically) after each stage act independently of all the others for the remainder of the sort - that is, they each perform a Parallel Quicksort on their own data. Also note that each of these partition stages requires one sequential pass over all the data stored at each node. Finally, after d partitions, when all the data stored at a particular processing node belongs to that node with respect to the global sort order, each node performs a local external sort on its data. If the Parallel Quicksort is being used to sort data internally then this local sort would be a serial Quicksort. However, in our case this local sort is external and so one of the algorithms from the previous chapters would be more efficient. The Parallel Quicksort is illustrated in Figure 5.5 for a $d = 3$ hypercube.

The two main reasons that Fox *et al* give for any inefficiency in the Parallel Quicksort both hinge on the determination of the "split key" - the key used during the partitions to split the data between sub-hypercubes. Firstly, a significant amount of time must be spent in determining this key. Secondly, it is difficult to choose a split key which evenly divides the data among the sub-cubes and any error in choice of split key results in load imbalancing. Both of these sources of inefficiency will also come into play when using the Parallel Quicksort to perform external sorts.

The experimental results obtained by Fox *et al* show that for large (*internal*) files

on small dimensional hypercubes the efficiency of the Parallel Quicksort is quite high (80% to 90%) and is better than that of the Parallel Shellsort. However, the efficiency of the internal Parallel Quicksort falls off much more rapidly than does that of the internal Parallel Shellsort as the dimension of the hypercube grows.

What kind of efficiency could we expect from an external Parallel Quicksort or an external Parallel Shellsort as the size of the hypercube grows? Unfortunately, if the external sort is truly i/o bound, we cannot expect very high efficiency. This can be seen for the Parallel Quicksort as follows: Let the total time required to read the input file if the entire file is stored on a *single* storage node be R_s. If the single-input single-output sort can be performed in two passes then the total cost of an i/o-bound sort is $2 \times R_s$. If instead that file is spread evenly across 2^d processing nodes, then the time needed to read the portion of the file on a single processing node will be $R_s/2^d$. This is the minimum time that each stage of the Parallel Quicksort will take, assuming that the file remains evenly spread among the multiple nodes throughout the sort and ignoring the processing and communication costs. Since, on a d-dimensional hypercube, the Parallel Quicksort requires (at least) d passes the minimum time for the entire sort will be $d \times R_s/2^d$. Therefore, the maximum speedup is $(2 \times R_s)/(d \times R_s/2^d) = 2 \times 2^d/d = 2^{d+1}/d$ and the maximum efficiency is $(2^{d+1}/d)/2^d = 2/d$. So, on a 2-dimensional hypercube (ie. four processing nodes) the maximum efficiency is 100%, but on a 10-dimensional hypercube (ie. 1024 nodes) the maximum efficiency is 20%. Once the costs of inter-processing-node communication and the final local sort stage of the Parallel Quicksort are included the actual efficiency will be somewhat less. For analogous reasons, the Parallel Shellsort is limited to a similar maximum efficiency for each dimension of hypercube. So we can see that Parallel Quicksort and Parallel Shellsort will not be cost effective for external sorting with large dimensional hypercubes.

What is limiting the efficiency of the parallel external sorts discussed above as the number of processing nodes increases is the accompanying increase in the number of passes required. We see that, when the sort is i/o bound, any algorithm which uses even a small number of passes over all the data at each node will suffer poor efficiency. For example, a hypercube with dimension $d = 3$ (only 8 processing nodes)

has a maximum efficiency of 66%. There is at least one algorithm in which, regardless of the number of processing nodes being used, the number of passes can be kept to a minimum of one pass for distribution and two passes for a subsequent local sort: Bucket Sort.[7] In this sort as each record is read from disk it is sent to the node to which it belongs without ever being written out to disk in the interim. Unfortunately, this method requires knowing, before the distribution can begin, the $pn-1$ keys which split the input file into pn equal parts - one for each of the pn processing nodes. If these split keys are not already known then approximations to them must be determined. The act of performing this determination would somewhat decrease the efficiency of the sort, though hopefully not as much as the multiple passes of the Parallel Quicksort and Shellsort do. However, the efficiency of the Bucket Sort is also highly sensitive to the accuracy of the split keys used. The entire Bucket Sort will take at least as long as the local external sort on the processing node with the greatest proportion of records.

There exist a number of distributed and parallel selection algorithms. For example, [QA89] describes a method for use in an optimal parallel internal sort on a hypercube. Unfortunately, this method requires a large number of passes over the data in each node and as such is inappropriate for use in an external sort. [LPS87] refers to a modification of the selection algorithms described in [SS82] and [SS83] for use in a distributed internal sort. However, this algorithm again requires a number of iterations which would significantly reduce the efficiency of a parallel external sort. [Kwa86], however, discusses a number of methods which are particularly suited to external sorting.

The preceding analysis of the effect of i/o on multi-input multi-output external sorts pre-supposes that the single-input single-output external sort which we want to parallellize is i/o-bound (ie. it can already be performed at the limit imposed by the speed of a link). One reason that we are so concerned with i/o-bound external sorts is that, in general, it appears that the speed of new cpu's is increasing at a

[7]Actually, as we will see in the next section, we can combine the actions of some of these passes together, reducing still further the total number of passes.

greater annual rate than the speed of i/o sub-systems.[8] If such technological trends continue then larger and larger files will be able to be sorted before the cpu becomes the primary bottleneck. Already, in our environment, as can be inferred from the data given in section 5.1, files composed of 128-byte records can be as large as 200MB and still only require four T800's to make the sort i/o-bound.

But what if the file size grows large enough (or the record size small enough) that substantial numbers of transputers are required to bring the Fastsort rate upto the i/o limit of a single storage node? As the external sort becomes more cpu-bound a parallel external sort could conceivably obtain greater efficiency than the limits predicted above. The question then becomes which is more (cost) efficient: extending the number of nodes used in the Fastsort implementation until it again becomes i/o-bound, or using a multi-input multi-output sort?

For example, suppose an external sort for a given file and record size can be performed at 75 Kb/sec using a single transputer and two storage nodes. Suppose, then, that a ternary tree consisting of 13 transputer nodes (nine leaf nodes and four interior nodes) can be used to enable a Fastsort to sort the same file at a rate of 600 Kb/sec (approximately the limit imposed by the link speed). This Fastsort would use a total of 10 storage nodes (one storage node for the root and one for each of the nine leaf nodes) and 13 transputer nodes to produce an 8-fold increase in the sort throughput for the given file. However, if a parallel external sort on a 3-dimensional hypercube were used in order to achieve the same 8-fold increase in throughput it would require 8 transputer nodes and 16 storage nodes (we would need at least two storage nodes for each processing node to eliminate the expensive seeks that a single storage node would require). Note that an 8-fold increase in total sort throughput assumes that we can obtain 100% efficiency from the hypercube sort. Thus, the hypercube based sort would require less transputers but incur the cost of extra storage nodes. Which method is more cost effective depends on the relative expense of transputer nodes vs. storage nodes.

Any model used to compare the performance/cost ratio of a single-input single-output Fastsort implementation vs. a multi-input multi-output parallel sort must

[8]This does not appear to be the case with the T9000.

include the size of the file being sorted, the size of the records in the file, the amount of RAM available on each processor node, the number and cost of transputers needed, and the number and cost of storage nodes.

5.3.1 Implementation of a Small Multiple-input Multiple-output External Sort

We have implemented one (very) small scale multiple-input multiple-output external sort. We are limited to such small scale i/o parallelism because we only have four disks, though from the previous sections we can see that this may not be such a bad thing. The algorithm we have implemented is basically a two bucket Bucket Sort, and is illustrated in Figure 5.6. We assume that before the sort begins the input file is stored on two separate storage nodes (Storage Nodes 1 and 2 in Figure 5.6). We also assume that we already know the median key value of the input file. If the median is not known then an approximation of the median would have to be determined in a preliminary step.

There are again two phases to this sort: a Sort Phase and a Merge Phase. During the Sort Phase T_NODEs 1 and 2 read that part of the input file which is stored on their local disks (Storage Nodes 1 and 2, respectively). They create, using Replacement Selection, sorted segments from their input. Ideally, T_NODE 1 would create *two* sorted segments concurrently, one containing records with keys smaller than the median key (call this a *low* segment) and another containing records with keys larger than the median (a *high* segment); similarly for T_NODE 2. Since two segments are being created concurrently the average size of each of these segments would then both be equal to the size of available RAM. When portions of the low segments are ready to be written out from T_NODEs 1 and 2 they are sent to T_NODE 3; analogously, the high segments are sent to T_NODE 4. T_NODEs 3 and 4 both merge the portions of the segments they receive and then write them out to storage nodes 3 and 4, respectively. The average size of segments written out by T_NODEs 3 and 4 are both twice the size of the segments created by T_NODEs 1 and 2. With appropriate use of double-buffering all four T_NODEs can operate in parallel during the Sort Phase.

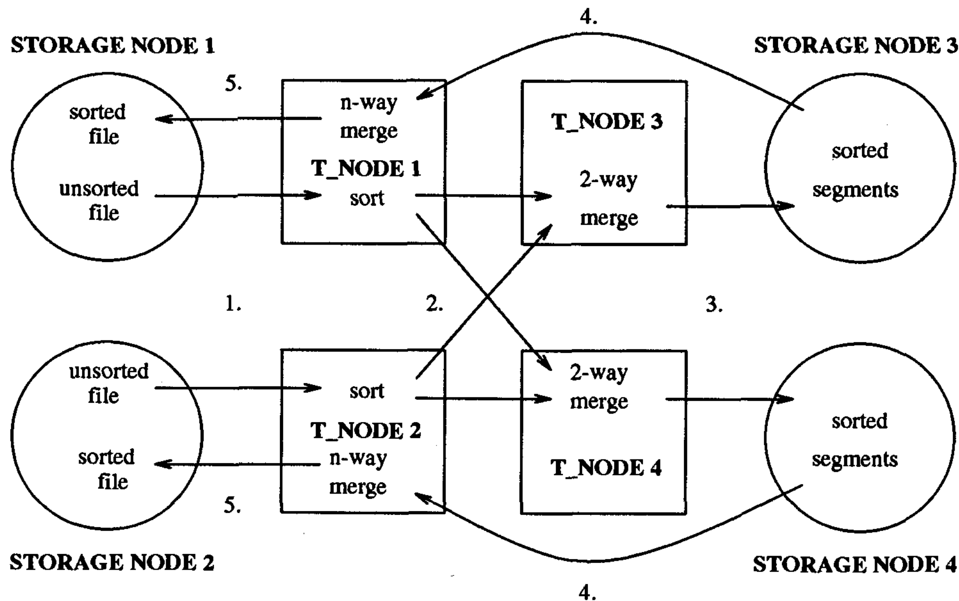


Figure 5.6: Small Scale Multi-input Multi-output External Sort

During the Merge Phase T_NODEs 1 and 2 merge, in parallel, the sorted segments stored on storage nodes 3 and 4 respectively.

Our implementation deviates from this ideal in one respect: each of T_NODEs 1 and 2 creates only a single segment containing both high and low records. The entire "low portion" of the segment is written out first and then the entire "high portion" of the segment is written out, and the same occurs for the next segment, and so on. This means that T_NODEs 3 and 4 will not operate in parallel as much as they would in the ideal implementation as only one of these nodes is receiving input at any one time.

The data obtained for this sort is given in Table 5.4.[9] Since for this implemenation there are essentially two sub-"sorts" running in parallel, one on T_NODEs 1 and 3 and another on T_NODEs 2 and 4, we obtained two sets of data - one for each sub-sorts. The data in the table reflects the larger of the two sets, since that is how long any user would have to wait for the entire sort to finish.

[9]Data is not given for 200MB files because one of our four disks broke before this data could be obtained.

Table 5.4: Sort Times and Throughput Rates When Using Multiple Storage Nodes for both Unsorted and Sorted Files

File Size	50 MBytes		100 MBytes	
	secs	Kb/sec	secs	Kb/sec
Sort Phase	**70.8**	**706.2**	**133.0**	**751.9**
cpu	50.3		87.3	
input	9.1		12.1	
output	11.4		33.6	
Merge Phase	**64.8**	**771.6**	**125.0**	**800.0**
cpu	29.6		62.4	
input	35.0		62.4	
output	0.2		0.2	
Total	**135.6**	**368.7**	**258.0**	**387.6**
Total cpu	79.9		149.7	
Total input	44.1		74.5	
Total output	11.6		33.8	

Two of the Tables with the most appropriate data against which to compare the data in Table 5.4 are Tables 4.6 and 4.8. Recall that these two Tables list the results obtained for a sort using a single transputer node, a single disk for the input and output files, and various numbers of disks for the sorted segments. More specifically, we are interested in the columns from these Tables which list the results for when a total of two disks are used - one for the input and output files and one for the sorted segments. The configuration which produced the data in this column closely matches that of one of the sub-sorts in our bucket sort. We are also interested in comparing the data in Table 5.4 with the data in Table 5.1. Recall that this Table lists data for a sort where two transputer nodes are used to create sorted segments, each of these transputer nodes has a separate disk attached to it for storing sorted segments, and a single storage node with two disks is used for both the unsorted and sorted files. (This is a configuration similar to that shown in Figure 5.1.) Table 5.1 lists data for an implementation in which only the Sort Phase has been parallized, whereas Table 5.4 lists data for an implementation in which both the Sort and the Merge Phases have been parallized.

Note that for the 100MB file in Table 5.4 the Sort Phase cpu time (87.3 seconds) and input time (12.1 seconds) are almost exactly the same as those for the two disk sort on 50MB files listed in Table 4.6 (86.5 and 12.7 seconds, respectively). This is exactly as expected. However, the 100MB Sort Phase output time in Table 5.4 (33.6 seconds) is more than 4 times larger than the corresponding time for the 50MB file in Table 4.6 (7.6 seconds). This is at least in part due to the slowness of the disk on storage node 4.[10] It likely also due in part to our deviation from the ideal implementation, which can be seen as follows: Both T_NODEs 1 and 2 begin by sending data to T_NODE 3 at a rate of approximately 468 Kb/sec (the Sort Phase rate from Table 4.6), so the total rate at which data is sent to T_NODE 3 (when it is sent) is 2×468 Kb/sec = 936 Kb/sec. However, even if T_NODE 3 can merge this data instantaneously, it can only send this data to storage node 3 at a rate of 730 Kb/sec (ie. the maximum rate at which data can be written to a single disk storage node). That is, data is being sent to T_NODE 3 faster than it possibly can be consumed by that node. The same imbalance of input and output will occur with T_NODE 4 when it receives its data.

There are a number of possible ways around this problem. One method would be to use extra buffers in the main memories of T_NODEs 3 and 4 and to allow the data from T_NODEs 1 and 2 to fill these buffers irrespective of how fast they can be merged and written to disk by the merging processes running on T_NODEs 3 and 4. Assuming the amount of memory used on T_NODE 3 for buffers is twice[11] the amount used on T_NODEs 1 and 2 for segment creation then there will be enough buffer space on T_NODE 3 to completely contain the low portions of the segments currently being created by each of T_NODEs 1 and 2. Once T_NODEs 1 and 2 have finished sending the low portions of their segments to T_NODE 3 they can start sending the high portions to T_NODE 4. Meanwhile, the merging process on T_NODE 3 can keep keep on merging all the buffers which remain in its RAM. As long as T_NODE 3 finishes this merging before T_NODEs 1 and 2 start sending the low portions of their

[10]Recall that one of our four disks has a peak sequential capacity of 590 Kb/sec.

[11]Actually, since T_NODE 3 is merging at the same time as it is receiving from T_NODEs 1 and 2, and thus emptying buffers at the same time as they are being filled, somewhat less than twice the amount of RAM would be necessary.

next segments to T_NODE 3 then T_NODEs 1 and 2 should never have to wait to send data to T_NODE 3. A similar argument can be made for T_NODE 4. Thus, the Sort Phase write times in Table 5.4 should improve. Unfortunately, this method requires extra memory on T_NODEs 3 and 4.

Another method would be to implement the Sort Phase in the ideal way described above. This would result in data being sent to T_NODEs 3 and 4 concurrently, instead of alternately as in the present implementation. So, the total rate of input to each of T_NODEs 3 and 4 would be $2 \times 468/2$ Kb/sec = 468 Kb/sec[12]. Therefore, the total input rate to each of T_NODEs 3 and 4 would be less than these nodes' maximum possible output rates (ie. 730 Kb/sec), thus removing the bottleneck encountered in our implementation. However, there is still another bottleneck possible: the rate at which the merge processes on T_NODEs 3 and 4 perform their merges. As long as T_NODEs 3 and 4 can merge the two segments they are receiving at a rate greater than or equal to the rate at which they are receiving these segments from T_NODEs 1 and 2 then the latter will never have to wait to output their data. Recall from Table 5.3 that a simple 2-way merge of 128-byte records with integer keys can be performed by a T800 transputer at approximately 7,700 Kb/sec. As this rate is far greater than the maximum rate at which data can be written to a single storage node the merge itself should not impose a bottleneck.

Finally, we mention that better performance in the Sort Phase could be achieved using a similar technique to that discussed in section 5.1.

Let us now look at the data for the Merge Phase in Table 5.4. We can see that, for the 100MB file, the time for this phase (125 seconds) is significantly less than half the time required for this phase by the single-cpu 2-disk implementation as listed in Table 4.8 (322.5 seconds). This is not too surprising since in the bucket sort there are two 50MB merges going on in parallel. As the theoretical length of the cpu time required for the Merge Phase grows faster than the size of the file being sorted we would expect that the time for one 50MB merge would be less than half the time

[12]Actually, since the heaps for the low and high segments would each contain (approximately) half the number of records in main memory at any one time, insertions and deletions into these heaps would occur slightly faster than if all the records were in one heap. So the actual rate at which T_NODEs 3 and 4 receive data from T_NODEs 1 and 2 would be slightly greater than 468 Kb/sec

needed for one 100MB merge. Indeed the 100MB Merge Phase *cpu* time in Table 5.4 (62.4 seconds) is less than half that shown in the 2-disks column of Table 4.8 (144.5 seconds). In other words, this bucket sort cuts the cpu cost for 100MB files by approximately 82 seconds, or 57%.

Moreover, the bucket sort has an even bigger savings in terms of input time. The bucket sort Merge Phase for the 100MB file takes 62.4 seconds for input whereas the Merge Phase for single-cpu 2-disk implementation takes 177.9 seconds for a 100MB file, as listed in Table 4.8. That is, the bucket sort cuts the input cost of the Merge Phase for 100MB files by approximately 115.5 seconds, or 65%. This is as a result of the larger buffer sizes that may be used during the Merge Phase when 50MB merges are performed instead of 100MB merges.

Unfortunately, if we compare the total Merge Phase time of the bucket sort for the 100MB file (again, 125 seconds) with the total Merge Phase time for the single-node 2-disk 50MB sort given in Table 4.6 (112.8 seconds) we see that the 100MB Merge Phase in the bucket sort is taking significantly longer, even though it is essentially two 50MB merges executing in parallel. As stated above, the times reported in Table 5.4 are for the longest of the two sub-sorts comprising the bucket sort (one sub-sort on T_NODEs 1 and 3, the other on T_NODEs 2 and 4). It turns out that the merge occuring on T_NODE 1 was always (ie. for every file) quicker than that occuring on T_NODE 2. The times for the 50MB merge on T_NODE 1 closely matched those for the 2-disk 50MB Merge Phase listed in Table 4.6. The reason why T_NODE 2 always performs a slower merge is that, as stated previously, the disk on storage node 4 has significantly poorer performance than any of the other three disks in our hardware environment. Indeed, if we compare the Merge Phase times for the 100MB bucket sort and the 50MB sort given in Table 4.6 we can see that all the extra cost occurs during the input.

To get better performance during the Merge Phase we could use extra storage nodes for the storage of the sorted segments, as discussed in Chapters 2 and 4.[13] This would allow the performance of the Merge Phase of the bucket sort on 100MB

[13]We would need more than the four disks available to us to do this.

files to approach that listed in Table 4.16 for the 50MB files. Furthermore, an additional transputer node could be attached to each storage node allowing the merges on T_NODEs 1 and 2 to be executed in the style of the Fastsort, as discussed in section 5.1.

What about increasing the number of buckets in this bucket sort? The main difficulty is determining the split keys in advance of the actual bucket sort. An efficient way of doing this accurately is crucial in maintaining the overall efficiency of the sort. Having accomplished this (possibly using one of the methods discussed in [Kwa86]) a good architecture to use for the Sort Phase would again be a hypercube. In fact, if the processes which execute on T_NODEs 3 and 4 in Figure 5.6 were moved to T_NODEs 1 and 2, respectively, then we can see that the bucket sort we have implemented is capable of executing on a 1-dimensional hypercube, albeit with some reduction in throughput.

5.4 Summary

In this section we give a brief overview of the experimental results of this thesis. Figure 5.7 shows two histograms which compare the times and rates for the various configurations we have examined. Data for the complete implementation of Fastsort is not included in this Figure because only three disks were available for the Fastsort experiments rendering data from these experiments incomparable with the rest of the data for 100MB files. From the Times histogram in Figure 5.7 it can be seen that we have started with an i/o-bound sort (no-frills), moved to a cpu-bound sort ($1 \times 2 + 2 \times 1$), alleviated the cpu-bottleneck (par-s and Fastsort, if Fastsort were shown), and have begun to encounter another i/o-bottleneck with our parallel i/o sort from the preceding section (par-i/o). The reason for this encroaching i/o-bottleneck is the limited number of disks available for our experiments (four). From the Rates histogram we can see that the throughputs of both phases of the sort have steadily improved except seemingly for the parallel i/o sort of the previous section. However, in some sense this parallel i/o sort has the best throughput of all as it is the only implementation whose phases have throughput rates (approximately 800KB/sec) which

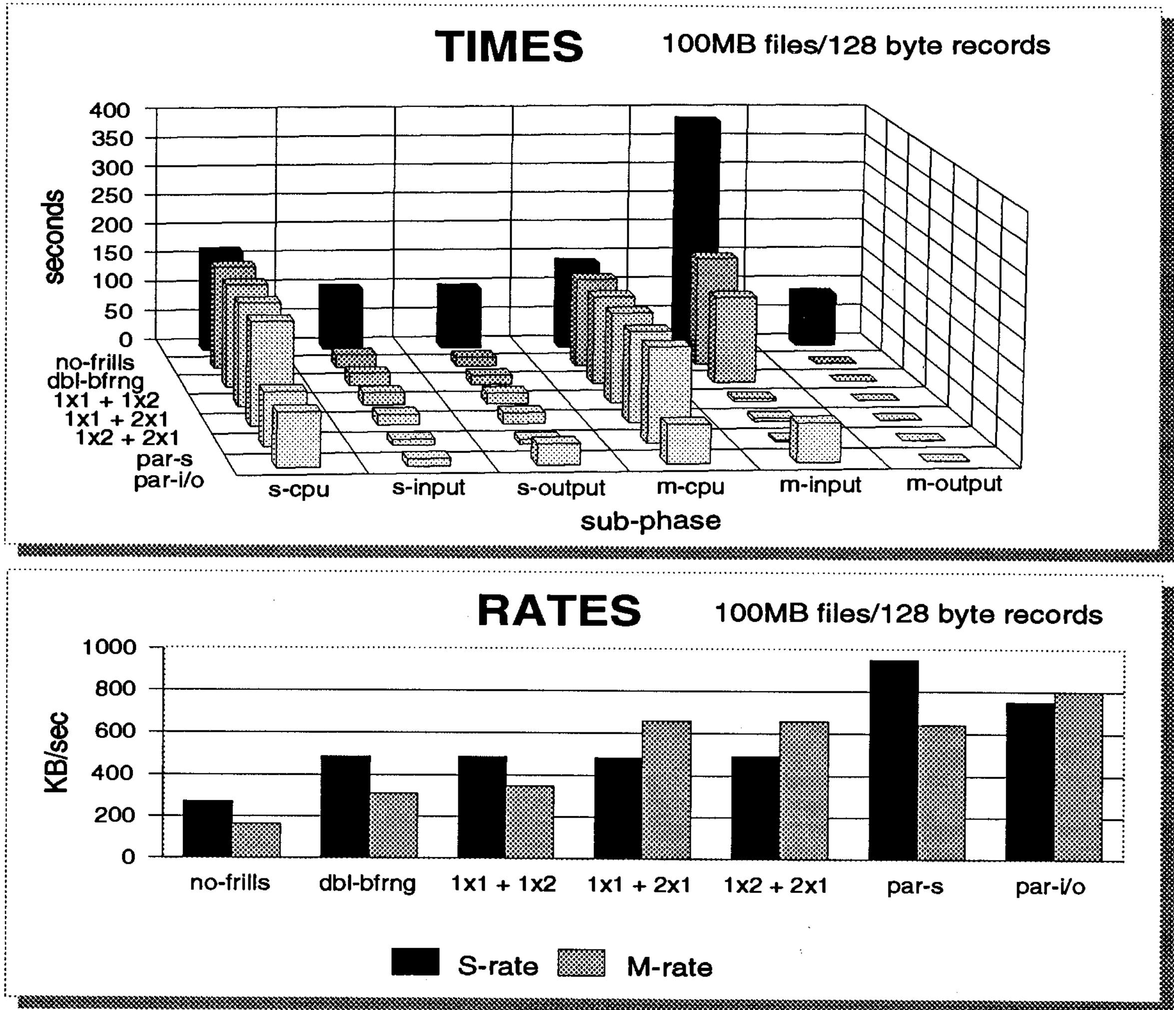


Figure 5.7: Final Overview of 100MB data

are faster than that of a single storage node (approximately 700KB/sec when the single storage node has a single disk).

Chapter 6

Summary and Future Work

6.1 Summary

The single-cpu external sorting algorithm we have studied is composed of two phases: a Sort Phase and a Merge Phase. For small records (eg. 8-bytes) the bottleneck for both phases in our environment is the cpu. For medium (eg. 128-byte) and large (eg. 1056-byte) sized records the bottleneck in the single-disk implementation is the i/o. With the addition of a single auxiliary disk (for storing sorted segments) the cpu becomes the bottleneck (for both phases of the sort) for the medium sized records as well as for the small records.

It is possible to reduce the percentage of time taken by i/o still further through the use of double-buffering and even more auxiliary disks. Double-buffering was found to be highly effective in the Merge Phase as well as in the Sort Phase. Pre-fetching is necessary for optimal input performance in both phases but too aggressive pre-fetching in the Merge Phase can result in performance degradation. The degree of pre-fetching in the Merge Phase which is too agressive was found, in many cases, to be surprisingly low.

The use of a second auxiliary disk for the sorted segments significantly improved the input time for the Merge Phase. The best performance that could be achieved with two auxiliary disks (through a trial and error process of changing the pre-fetch degree) was better than that achievable with three auxiliary disks. Thus, the use

of more than two disks on a single auxiliary storage node is questionable. It was also found to be more effective to place auxiliary disks on separate storage nodes. Using separate auxiliary nodes gives the best Merge Phase input performance for a single-cpu external sort in our environment.

It is fairly straightforward to get around the cpu-bottleneck which exists during the Sort Phase through the addition of more transputers, though care must be taken not to let the extra transputers introduce extraneous seeks during the input for this phase. With a small number of transputers it is possible to create sorted segments at the limit imposed by the throughput of a storage node (ie. link speed).

From the results of the bucket sort implementation we can infer that adding more cpu's would also be highly effective in reducing the cpu-bottleneck in the Merge Phase. In fact, with a deep enough tree, it is likely that a Fastsort can merge sorted segments at a rate again limited by the throughput rate of a link.

We have found that, for the largest files storable in our environment (ie. 200MB) there is not much room for cpu-parallelism before the i/o-bottleneck is encountered. For files composed of 128-byte records this bottleneck is encountered with only three or four transputers; somewhat more could be used for the smallest possible records.

If an external sort is i/o-bound one seemingly attractive way around the i/o bottleneck is to split both the input and output files among multiple disks. However, we have seen that, for i/o bound sorts, multi-input multi-output algorithms which require even a small number of passes over the data at each node can have very poor efficiency, even for a small number of processing nodes. Overall sort efficiency could be maintained with an efficient and accurate method for determining the split keys for a file.

If an external sort is sufficiently cpu-bound then adapted versions of parallel internal sorts may be more efficiently used in external sorting.

6.2 Future Work

A more sophisticated analysis of the effects of the TASS CacheServer on the performance of the external sort needs to be done. Times for external merges using a cache

which uses a replacement algorithm specifically suited to the needs of merges (eg. Most Recently Used) should be obtained. Such times should be compared with those presented in this thesis which arise from a cache which uses the more general Least Recently Used replacement algorithm.

Performance figures for transputer external sorts using non-integer keys (eg. character strings) should be obtained and analyzed.

A model should be created which, for a given file and record size, allows the prediction of the number of transputers and disks that are needed to enable Fastsort to work at the limit imposed by the speed of a link.

More sophisticated models are needed to predict the efficiency of internal parallel sorts (eg. Parallel Quicksort) when they are adapted for external sorting. Such models would need to include the cost of inter-node communication.

With more disks large transputer implementations of Fastsort and, for example, Parallel Quicksort could be made. Efficiencies and cost/performance tradeoffs for these large scale implementations could be compared.

Efficient ways of finding split keys in large files need to be found.

Bibliography

[A+85] Anon. et al. A measure of transaction processing power. *Datamation*, 31(7), April 1985.

[Baa88] Sara Baase. *Computer Algorithms : Introduction to Design and Analysis.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1988.

[BBH78] J. Bannerjee, R. I. Baum, and D. K. Hsiao. Concepts and capabilities of a database computer. *ACM Trans. Database Syst.*, 3(4), December 1978.

[BDHM84] Dina Bitton, David J. DeWitt, David K. Hsiao, and Jaishankar Menon. A taxonomy of parallel sorting. *Computing Surveys*, 16(3), September 1984.

[BF82] D. Bitton-Friedland. *Design, analysis and implementation of parallel external sorting algorithms.* PhD thesis, University of Wisconsin, 1982.

[Com89] Computer System Architects, 950 North University Avenue, Provo, Utah, 84604, U.S.A. *Part.6, Part.7, Part.8, and Part.12 Users' Manuals*, june 1989.

[Com90] Computer Systems Architects, 950 North University Avenue, Provo, Utah, 84604, U.S.A. *Logical Systems C for the Transputer: Version 89.1 User Manual*, 1990.

[Dud92] Timothy Dudra. An auxiliary storage system for transputer-based multi-computers. In *Proceedings of the 5th North American Transputer Users Group*, 1992.

[Fox88] Geoffrey C. Fox. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, N.J., 1988.

[Inm89] Inmos Ltd., 1000 Aztec West, Almondsbury, Bristol, BS12 4SQ, U.K. *Transputer Data Book, 2nd Edition*, 1989.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, Reading, Massachusetts, 1973.

[Kwa86] Sai Choi Kwan. *External Sorting: I/O Analysis and Parallel Processing Techniques.* PhD thesis, University of Washington, 1986.

[LL89] W.S. Luk and F. Ling. An analytic/empirical study of distributed sorting on a local area network. *IEEE Transactions on Software Engineering*, 15(5), May 1989.

[LPS87] W.S. Luk, J. G. Peters, and M. Salehmohammond. Performance evaluation of lan sorting algorithms. In *Proc. 1987 Conf. Measurement Modeling Comput. Syst.* ACM-SIGMETRICS, May 1987.

[LPW88] W.S. Luk, J. G. Peters, and Xiao Wang. On the communication cost of distributed database processing. In *The 8th International Conference on Distributed Computing Systems*, June 1988.

[LY89] R. A. Lorie and H.C. Young. A low communication sort algorithm for a parallel database machine. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, 1989.

[MK91] L. W. McVoy and S. R. Kleiman. Extent-like performance from a unix file system. In *USENIX proceedings*, 1991.

[Mor88] Joseph P. Moran. Sunos virtual memory implementation. In *Procs. European UNIX User's Group*, April 1988.

[PH90] David A. Patterson and John L. Hennessy. *Computer Architeture A Quantitative Approach.* Morgan Kaufmann, San Mateo, California, 1990.

[Pou91] Dick Pountain. The transputer strikes back. *Byte*, 16(8), August 1991.

[QA89] Ke Qiu and Selim G. Akl. Optimal sorting on a hypercube. Technical Report 89-255, Queen's University at Kingston, 1989.

[RSS85] D. Rotem, N. Santoro, and J. Sidney. Distributed sorting. *IEEE Trans. on Computers*, 34(4), April 1985.

[Sal88] B. Salzberg. *File Structures: An Analytic Approach.* Prentice Hall, Englewood, New Jersey, 1988.

[Sed90] Robert Sedgewick. *Algorithms in C.* Addison-Wesley, Reading, Massachusetts, 1990.

[SS82] N. Santoro and J.B. Sidney. Communication bounds for selection in distributed sets. Technical Report SCS-TR-10, Carleton University, 1982.

[SS83] N. Santoro and J.B. Sidney. A reduction technique for selection in distributed files: I. Technical Report SCS-TR-23, Carleton University, 1983.

[STG+90] B. Salzberg, Alex Tsukerman, Jim Gray, Michael Stewart, Susan Uren, and Bonnie Vaughan. Fastsort: A distributed single-input single-output external sort. *ACM SIGMOD Record*, 19(2), June 1990.