Portland State University

PDXScholar

Dissertations and Theses

Dissertations and Theses

2-8-1995

PSUsort: A Parallel External Sort for a Shared Memory Multiprocessor System

Sujata V. Ramamoorthy
Portland State University

Follow this and additional works at: https://pdxscholar.library.pdx.edu/open_access_etds

Part of the [Computer Sciences Commons](#)

Let us know how access to this document benefits you.

Recommended Citation

Ramamoorthy, Sujata V., "PSUsort: A Parallel External Sort for a Shared Memory Multiprocessor System" (1995). *Dissertations and Theses*. Paper 4948.

<https://doi.org/10.15760/etd.6824>

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. Please contact us if we can make this document more accessible: pdxscholar@pdx.edu.

Thesis Approval

The abstract and thesis of Sujata V. Ramamoorthy for the Master of Science degree in Computer Science were presented February 8, 1995 and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

Leonard Shapiro, Chair

Yingke Li

Marc Feldesman
Representative of the office
of Graduate Studies

DEPARTMENT APPROVAL:

Warren Harrison, Chair
Department of Computer Science

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by

on

20 March 1995

Abstract

An abstract of the thesis of Sujata V. Ramamoorthy for the Master of Science in Computer Science, presented February 8, 1995.

Title : PSUsort : A Parallel External Sort for a Shared Memory Multiprocessor System.

A method to parallelize external sorts on a shared memory multiprocessing system is presented in this thesis. The main goal of the thesis is to develop a sorting package that is scaleable and efficient. No prior knowledge of the nature, source or size of the data is assumed for this work. A dynamic load-balancing architecture is used with no static allocation of tasks to processes.

The package consists of an interface and a kernel. The interface provides the sort with the following - the sort input, output and temporary work spaces as abstract data types (ADTs), memory available, number of processes available, compare routine to compare records, etc. Only the interface needs to be changed to suit different environments.

The kernel implements the parallel sort algorithm. The traditional sort-merge technique is used for the external sort as opposed to a distributive sorting technique. Memory-sized runs are first generated and later merged. Parallel binary merges is the technique used for both the run generation and the merge phase. A forecasting table is used to read ahead in the merge phase.

PSUsort:
A PARALLEL EXTERNAL SORT FOR
A SHARED MEMORY MULTIPROCESSOR SYSTEM

by
SUJATA V. RAMAMOORTHY

**A thesis submitted in partial fulfillment of the
requirements for the degree of**

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
1995

Acknowledgments

Many thanks to my advisor, Dr. Leonard Shapiro. This work would not have been possible without his superb guidance. I would also like to thank Gary Graunke, from Sequent Computer Systems, Inc. for his invaluable insight and industry perspective on ideas presented in this thesis. This work was sponsored by Sequent Computer Systems, Inc. and special thanks to them for the grant and the equipment resources provided for this study. Thanks also to the members of the thesis committee for their time and effort in reviewing the thesis. I would also like to thank Goetz Graefe for his contribution towards my understanding of DBMS implementation techniques. Finally, I would like to thank my husband Sai for his tremendous support and encouragement during the course of this work.

Contents

Acknowledgments	<i>ii</i>
1 Introduction	1
2 Related Work	6
3 Interface	13
3.1 SortInputStream	15
3.2 SortOutputStream	16
3.3 SortRandomStorage	17
3.4 DiskBlock	19
3.5 SortInterface	19
4 Kernel	21
4.1 Parallelization Strategy	21
4.2 Implementation	23
4.2.1 Run generation phase	25
4.2.2 Merge Phase	35
4.2.2.1 Reading from the SortRandomStorage	40
4.2.2.2 Merges in the Merge Phase	42
4.3 Block Sizes in the Sort	47

5	Performance	52
5.1	System Configuration	52
5.2	Profiling	52
5.3	Experimental Results	57
5.3.1	Scale-up	57
5.3.2	Speed-up	60
5.3.3	Size-up	61
5.3.4	Comparison with /bin/sort Utility	63
6	Summary	65
	Bibliography	71

List of Tables

2.1	Duality between the sort-merge and the distributive methods	7
3.1	Methods provided by ADT SortInputStream	16
3.2	Methods provided by ADT SortOutputStream	17
3.3	Methods provided by ADT SortRandomStorage	18
3.4	Methods provided by ADT DiskBlock	19
5.1	Scale-up results	58
5.2	Speed-up results with cardinality fixed at 350,000 records ...	61
5.3	Size-up results with 7 processes and 7MB of memory	62
5.4	Comparison with /bin/sort utility	63

List of Figures

3.1	Sample (DFSORT) format accepted by the interface	14
3.2	Block diagram of PSUsort	15
4.1	How much of data can be sorted in two passes ?	24
4.2	Run generation phase	26
4.3	Pseudocode for the run generation phase	27
4.4	Pseudocode for inserting a SE into the merge tree	28
4.5	A merge node	29
4.6	Pseudocode for the allocation of records for a merge	31
4.7	Pseudocode for performing merges from top to bottom	32
4.8	Pseudocode for merging and generating a run	33
4.9	Merge phase	36
4.10	Pseudocode for the merge phase	37
4.11	Pseudocode for reading from the SortRandomStorage	41
4.12	Pseudocode for merges from bottom to top	43
4.13	Deadlock scenario 1	45
4.14	Deadlock scenario 2	46
4.15	Pseudocode for generating the sorted stream of records and writing to SortOutputStream	47

5.1	Execution time distribution	53
5.2	Scale-up graph	59
5.3	Speed-up graph	61
5.4	Size-up graph	62
5.5	PSUsort vs. /bin/sort	64

CHAPTER 1

INTRODUCTION

Sorting is an important operation performed by a Database Management System (DBMS). In a DBMS, sorting is used for explicit order-by clauses, sort-merge joins, duplicate elimination, index builds, sub-queries, grouping and aggregation [7].

The volume of data managed by DBMSs is ever increasing and is on the order of gigabytes and terabytes for many large organizations at present. The typical data-set is so large that the main memory is insufficient to hold the data. With the advent of parallel query processing, multiple queries often run in parallel and main memory is divided between them. Virtual memory does not provide optimal performance and hence explicit I/O operations to temporary work space is desirable [15,20].

With the advent of multi-processor architectures, methods are required to parallelize the sort operation. There are two traditional ways of doing this - splitting the data physically amongst processors followed by a logical merge, and splitting the data logically followed by physical concatenation of results at each processor [1,2,9,14]. The first is

referred to as the sort-merge method and the second as the distributive method.

A distributive sort lends itself easily to parallelization. It can be faster as there is no merge step involved. The challenge lies in splitting the data evenly amongst processors to achieve maximum parallelism. This requires some advance knowledge of the nature of the data. There has been much research done to find algorithms that can estimate the quantiles for distribution based on sampling or some kind of pre-processing of the data-set [1,4,18]. When the source of the data is a pipe, one needs to wait for all of the data to be read in to get a good sampling. Sampling the first N blocks of data will not be a good representative of the complete dataset and can result in an inaccurate estimation of quantiles. One might argue that a sort cannot produce its output until it consumes all of its input. But then waiting for all of the data to be available for sampling can be expensive. Distribution of data cannot begin until sampling is completed and quantiles are determined. Also an incorrect estimation by the partitioning function could be very costly. An uneven distribution could cause a severe overflow at one processor and require repartitioning.

The sort-merge method has the advantage that no pre-processing of data is required. It is not affected by skews due to the physical partitioning of data. PSUsort uses this sort-merge model and is explained in detail in chapters 3 and 4. The process that performs the merge step in this method can become a bottleneck as a complete pass over the data is required [9]. To improve parallelism, the merge may be constructed of a tree of binary merges. A process may be assigned to each binary merge. Each level of the resulting merge tree must have the same throughput. The root merge becomes the bottleneck. In this thesis, a parallelization strategy is presented that alleviates this merge bottleneck.

PSUsort is implemented on a shared memory multi-processor system. These systems offer a limited parallelization as compared to distributed systems. Nevertheless they can support up to 30 processors before the bus actually becomes a bottleneck. Such systems are expected to be the nodes of distributed systems in a hierarchical-memory architecture [15]. Our choice of such a system was partly due to its availability for research.

PSUsort can be described using the taxonomy proposed by Graefe for external sorting [9]. According to the taxonomy -

- Does the sort input and output reside on a single or multiple disks ?

The sort package expects the interface to handle these issues. The interface provides ADTs for input and output streams. The input and output data could reside on a single or multiple disks, could be coming through a pipe, a tape, etc. The source and destination of data are not assumed by the sort and are handled by the interface.

- How often does a data item migrate between sites ?

With the shared memory architecture there are no sites to begin with. We could consider caches to be sites. In that case our data may move between sites several times. Cache sensitivity [5] has not been explored in this study.

- Are whole records or only key and record ids moved ?

Record ids (pointers to records) are used for sorting. Whole records are written to disk and then read back in the case of external sorting.

- What main memory sorting method is used for run creation ?

Each input block is sorted using quick-sort and then merged with other such blocks using parallel binary merges to create runs.

- How are sorted runs merged ?

Again using parallel binary merges. This is explained in detail in chapter 4.

The main goal is to develop a sort that exploits parallelism and is independent of the nature of the data. We hope to achieve at least a near-linear speed-up and scale-up empirically.

Chapter 2 discusses related work in the area of sorting and compares the sort-merge model with the distributive model. It also discusses the motivation behind our implementation. Chapter 3 describes the various ADTs implemented in the interface. Chapter 4 describes the implementation of the kernel in detail along with pseudocode. Chapter 5 discusses the performance of PSUsort on a Sequent Symmetry. Chapter 6 provides the summary.

CHAPTER 2

RELATED WORK

There has been much research done in the areas of both internal and external sorting. Most of the research on parallel sorts has been based on a distributive sort in a shared-nothing environment [1,4]. In a distributive sort, records are first split logically into buckets that fit in memory and then each bucket is sorted internally. The partitioning of the input data set into buckets is a recursive process and continues until the buckets fit in memory. The result is just the concatenation of the individual sorted buckets. In a sort-merge based method, the complete dataset is physically divided into runs. The individual runs are sorted in memory and later merged to generate larger runs. The merging of runs continues until one final run of sorted records is formed.

There is a duality between the distributive sort and the merge sort [1,14]. The amount of work done by the two sorts is also of the same order of complexity. Table 2.1 illustrates the duality, where M is the memory size in bytes, R is the record size in bytes, N is the number of records and B is the run size (or bucket size) in bytes.

	SORT-MERGE	DISTRIBUTIVE SORT
	Run generation phase	Bucket generation phase
I/O	Sequential reads/writes	Sequential reads, Random writes
CPU	$O(N \log (M/R))$ comparisons	$O(N \log (M/B))$ comparisons
	Merge phase	Bucket sorting phase
I/O	Random reads, Sequential writes	Sequential reads/writes
CPU	$O(N \log (M/B))$ comparisons	$O(N \log (M/R))$ comparisons

Table 2.1 : Duality between the sort-merge and the distributive methods

Thus, there is a correspondence between the run generation phase of the sort-merge and the bucket sorting phase of the distributive sort. A similar correspondence exists between the merge phase of the sort-merge and the bucket generation phase of the distributive sort.

In the case of distributive sorts, data need to be evenly distributed among the buckets to achieve optimal performance. There has been some research in the area of finding a partitioning function for even distribution of records. All these papers assume the availability of all of the data or are based on some kind of a sampling technique [1,4,17,18]. They involve a pre-processing step to determine the ranges. In cases where such pre-processing is not possible, data can be split in some static way based on a range that the key can assume and the distribution

can be changed dynamically by splitting or merging the buckets, also called *bucket tuning*. Therefore, a distributive sort could have an overhead of pre-processing the data or correcting an imbalance in distribution on the fly to balance the load amongst the workers or processes. Another reason for an even distribution is to avoid overflows of buckets. Without an even distribution, some buckets can get larger than others resulting in unnecessary overflows to intermediate storage. This involves recursive partitioning of the overflow buckets with an overhead of finding a partitioning function again for the overflow buckets. The recursive partitioning in a distributive sort corresponds to multiple levels of merges in the merge-sort.

Parallel distributive sorts are straight-forward. Each worker reads a portion of the input data set, applies the partitioning function to re-distribute the data, and after the data exchange step sorts the local set of records. Parallel sort-merges have not been explored in great detail in the sort literature. Newberg et al. [5] and Anderson [8] discuss an implementation of a sort-merge based algorithm on an SMP (shared memory multi-processor) and Fastsort [11] is an implementation of a sort-merge on a shared-nothing architecture. Both Alphasort [5] and Fastsort [11] use function partitioning as the parallelization strategy (see

section 4.1 for a description of the parallelization strategies). In the case of Alphasort all of the merging and I/O is done by the root process while others do the copying of records into buffers. The root process could be a bottleneck and it is not clear if such an implementation is scaleable. The paper does not provide any results on scaleability. Fastsort distributes records to nodes, sorts and performs a local merge at each node followed by a super-merge of the data streams from all nodes. The super-merge could become a bottleneck. Anderson [8] discusses the implementation of a static task graph for scheduling tasks to workers. Our implementation of an *everybody does everything* paradigm with data-partitioning as the parallelization strategy is simple, scaleable and requires no such task graphs.

Parallelizing the merge phase in merge-sort has been a topic of research to avoid the bottleneck of a single process merge. Anderson [8] discusses ways of parallelizing merges by splitting the runs into parts using a binary search that can be merged in parallel by multiple processes. This is similar to the percentile method described by B.K.Iyer et al. [7] and the quick-merge algorithm described by Quinn [3]. In quick-merge the records in the first run are used as dividers to split the rest of the runs. This method does not exhibit exact load balancing and

can result in a loss in speed-up in the cases of skewed data. The percentile method [7,8] uses a binary search to divide the runs into equal sized partitions for a parallel merge. This method performs well even in cases with skew. The percentile finding algorithm needs to be executed in a critical section before the merge and can get fairly complicated. For example, with 8 processors, each one doing a merge, each run (the records that are available in memory from a run) needs to be partitioned 8 ways to perform merges in parallel. The binary search algorithm to split the runs 8 ways can get very complicated, especially when the number of runs is large and could be of unequal sizes. Our implementation is a simple alternative to the percentile-finding algorithm, where the merges are binary and the processes choose a certain number of records to merge from each data stream based on a binary search. This way two processes can be merging two different pairs concurrently or can be performing merges of disjoint sets of records from the same pair of runs. This is much simpler to implement as merges are binary and runs are not statically partitioned, rather each process chooses to merge a certain portion of the run. For example, in a tight situation where each run is represented by only a block's worth of records in memory, splitting each block 8 or so ways could be a significant overhead as compared to the parallel merge itself.

Forecasting is a method for improving performance by reading-ahead data in the merge phase. Woodrum discusses forecasting in detail [12]. Betty Salzberg discusses double buffering while merging runs when a large amount of main memory is available [6]. This reduces the fan-in of the merge by half. Also, when the data are not uniformly distributed, all of the data read in may not be immediately useful. Goetz Graefe discusses forecasting, where one buffer's worth of records is read in advance from a run determined by comparing the last record from each run [15]. PSUsort uses a forecasting table sorted in the order in which reads need to be performed. Thus, the amount of reading ahead depends on the memory that is available. Also, all of the data read into the buffers is immediately useful. So, in the case of a large main memory, having a forecasting table in memory along with read-ahead buffers can be more useful than double buffering. By having a sufficient amount of read-ahead, CPU and I/O bandwidth can be matched. Forecasting table is absolutely necessary for a scaleable parallel implementation to deal with severe data skew. Without a forecasting table, only one buffer's worth of records can be read ahead as it's last key is used for predicting the next buffer for read ahead.

Verkamo [2] does a comparison between the performance of distributive and merge sorts. The conclusion drawn in the paper is that for small records, the optimal performance of distributive sorts is somewhat better (10-20%) than the merge sort and for large records, the optimal performance of the merge sort is better (20-30%) and at its optimal point, distributive sorting was slower and required more space. It does not discuss any parallelism.

There are numerous tricks that various papers talk about to achieve optimal performance. For example, Goetz Graefe [15] discusses various optimizations for uniprocessor sorts like saving the last run in memory, eager versus lazy merging, smaller runs with more merge levels for optimized I/O, writing runs in reverse order, etc. These optimizations can be applied to multiprocessor sorts as well. We did not explore these as the idea was mainly to develop a method to parallelize sorts rather than to do performance tuning.

CHAPTER 3

INTERFACE

The sort package consists of an interface and the kernel. The interface encapsulates the sort-independent details and the kernel comprises the actual implementation of the parallel algorithm. In this chapter we describe the components of the interface.

The user of the sort can provide the interface with all the details in a specific format required by the interface. An example of the format accepted by the interface is shown in Figure 3.1. The format is based on DFSORT [19]. The interface provides the sort package with various ADTs to handle the input, output and intermediate data streams. It can be modified to suit different environments. For example, the data might be read from a network and sent out to another machine, or the amount of memory to be used by the sort could be granted by a resource manager, or the number of processes to be used by the sort could be altered depending on the number of on-line processors. These issues are external to the kernel and are controlled by the interface. The interface could also provide the kernel with information like I/O channel

bandwidth or system bus bandwidth to let the kernel make appropriate judgments on different block sizes for a complete overlap of CPU and I/O work. Currently the kernel does not make use of such information.

SORT FIELDS = (0, 4, BI, A)	-- key offset, key length, data type, ascending/descending
RECORD TYPE = F, LENGTH = (128)	-- F for fixed length and length is in bytes
OPTION SORTIN = in,	-- file for input dataset
SORTOUT = out,	-- file for sort output
SORTDD = work,	-- file for intermediate storage
MAINSIZE = 2000k	-- Main memory available to sort
END	

Figure 3.1 : Sample (DFSORT) format accepted by the interface

The kernel reads input records from the interface, sorts them and writes the sorted stream of records to the output device through the interface. A two-pass sort is performed when the main memory is insufficient to sort the data in a single pass. Memory-sized sorted runs are generated and then later merged to generate a single stream of sorted records.

The interface provides the kernel with a set of ADTs described below.

Figure 3.2 illustrates the block diagram of PSUsort.

Figure 3.2 : Block diagram of PSU sort

3.1 SortInputStream

This ADT provides methods to handle the input stream of data to the sort package. The kernel provides the interface with pages and the interface returns them filled. The interface can allocate pages on its own if the kernel does not, or the number of pages required to read an **InputBlock** exceeds the number of pages provided. The interface also provides an array of record addresses called the **RecordAddressVector** that can be used directly by the kernel. The kernel assumes that records do not span pages. Reading is done in two phases - processes reserve an **InputBlock** in a critical section and then do the actual read from the offset they reserved in parallel with other processes. It is the

responsibility of the kernel to reserve the InputBlock in a critical section. Table 3.1 lists the methods provided by this ADT.

Method	Purpose	Return Value
nextInputBlock()	Reserve the next input block for reading	Offset reserved in input device
getBlock()	Read an InputBlock	Pages filled with data
maxBlkAttributes()	To allocate data structures	Maximum records and maximum bytes per InputBlock
inputRandomOk()	To see if data are coming through a tape or pipe	Yes or No
The remaining methods are used by the kernel to make appropriate decisions. They may all return a NULL value.		
systemBusBW()		Bus BW in bytes/sec
inBlkCPUEstimate()		Block read CPU time in usecs
inBlkChanlEstimate()		Block read channel busy time in usecs
inBlkSeekEstimate()		Block read disk seek/rotational latency in usecs
getTotalRecords()		Cardinality if available
getTotalBytes()		Size of the dataset

Table 3.1 : Methods provided by ADT SortInputStream

3.2 SortOutputStream

This ADT provides methods to write sorted data to the output device. The kernel provides the interface with a stream of record addresses sorted by the key. The interface can choose to either copy the records into a buffer and write to the output device, or pass the record addresses

themselves, or make use of operating system scatter/gather read/write calls to output the data. Writing is again in two phases - reserving the output offset in a critical section by the kernel followed by writes in parallel with other processes. Table 3.2 lists the methods provided by this ADT.

Method	Purpose	Return Value
reserveOutputBlock()	Allocate space in the output stream	Offset reserved in output device
putBlock()	Copy records to output buffers and do I/O	None
outputRandomOK()	To see if the output is a tape or pipe	Yes or No
The remaining methods are used by the kernel to make appropriate decisions. They may all return a NULL value.		
outBlkCPUEstimate()		Block write CPU time in usecs
outBlkChanlEstimate()		Block write channel busy time in usecs
outBlkSeekEstimate()		Block write seek/rotational latency time in usecs

Table 3.2 : Methods provided by ADT SortOutputStream

3.3 SortRandomStorage

This ADT provides methods to read and write runs to intermediate storage. The unit of writes is called LargeBlocks and the unit of reads SmallBlocks (See section 4.3 for the description of SmallBlocks and LargeBlocks). All of the block sizes - LargeBlocks, SmallBlocks,

InputBlocks and OutputBlocks are in terms of pages and the page size can be configured. The interface does the copying of records into pages for both the SortInputStream and the SortOutputStream, but the kernel is responsible for copying records for the SortRandomStorage. Letting the kernel do the copying allows it to fill the pages it deems appropriate. For example, the kernel might append record lengths to records or it may mark the end of last record in a page with a special character.

The interface decides where to write the LargeBlocks in the intermediate storage. The user can specify multiple files or devices for the intermediate storage for a higher I/O bandwidth. In such a case the interface can spread the LargeBlocks in a random manner to avoid reads from being on the same device for longer periods of time in the case of skewed data. Table 3.3 lists the methods provided by this ADT.

Method	Purpose	Return Value
minBlockSize()	Minimum allowable disk block	Size in bytes
maxBlockSize()	Maximum allowable disk block	Size in bytes
blockIncrement()	Disk block is a multiple of this	Integral number
setBlockSizes()	Fix small and large block sizes	None
allocLargeBlock()	To allocate disk space	DiskBlock (see below)
dropLargeBlock()	Deallocate LargeBlocks	None
writeBlocks()	Write LargeBlocks to disk	None
readBlocks()	Read SmallBlocks from disk	Data filled pages

Table 3.3 : Methods provided by ADT SortRandomStorage

3.4 DiskBlock

This ADT defines the address of a block of data in the intermediate storage. It is used by the kernel to perform reads and writes to the intermediate storage. Since the kernel chooses the order in which to perform the reads from the intermediate storage, it needs to keep track of disk addresses containing the blocks. The disk address could be as simple as a file offset into a UNIX file. Table 3.4 lists the methods provided by this ADT.

Method	Purpose	Return Value
addOffset()	To get to an offset within a DiskBlock	DiskBlock
eqDiskBlock()	To compare two DiskBlock addresses	Yes or No

Table 3.4 : Methods provided by ADT DiskBlock

3.5 SortInterface

This is a data structure filled in by the interface, containing the following fields -

- SortInputStream, SortOutputStream and SortRandomStorage ADTs.
- Number of processes that the kernel will use.
- Memory in bytes available to the kernel.

- Pointer to a memory allocation and deallocation routine that the kernel should use. This enables the implementation of an independent memory manager by the interface.
- Duplicate elimination flag which indicates to the kernel to eliminate duplicate records.
- Key length. For variable keylengths, this is the maximum length that a key can assume¹.
- Pointer to a compare routine to compare two records.
- Pointer to a routine to extract keys from records. This is used by the kernel to maintain the forecast table.

¹ Variable record lengths have been implemented but not tested in the current implementation.

CHAPTER 4

KERNEL

4.1 Parallelization strategy

There are two techniques to parallelizing programs - data partitioning and function partitioning [16]. With function partitioning, there is an allocation of functions to processes. The functions are executed in parallel by the processes with appropriate synchronization between them. With data partitioning, each and every process performs all of the functions but with a different set of data.

Sorting can be accomplished with either approach. For example, with the function partitioning approach, some processes could be doing I/O, some copying, some doing the actual sort, some merging different data streams, etc. I/O work by one process can be overlapped with sorting work by another. This results in a static scheduling of tasks as the order and the set of functions that can be executed concurrently are pre-determined. Load balancing is difficult to achieve as some functions may take longer than others. Also the number of functions may be limited as compared to the number of data partitions. This method is not easily scaleable. For example, the single process that does the merge can be a

bottleneck [9]. It is also a challenge in itself to decide the number of processes to do I/O, sort, etc. Alphasort [5] and Fastsort [11] use function partitioning as the parallelization strategy.

With the data partitioning approach, the processes are self-scheduling and load balancing is easier to achieve. Each process performs all of the functions but with a different set of data. For example, all of them do I/O, copying, sorting, merging records, etc. There is no single bottleneck as the time spent on executing expensive functions is equally shared by all the processes. Load balancing is easier since the data can be partitioned evenly amongst the processes. Each process works with a subset of data and coordinates with others while choosing the subset. Scheduling is automatic as the processes move on from one task to another. This method adapts automatically to the number of processes in the system. PSUsort uses the data partitioning approach because of the above mentioned reasons. Most distributive sorts [4,9] follow this *everybody does everything* paradigm. They have a data distribution step where all the processors are involved in data exchanges as per the partitioning function followed by a local sort at each processor. Since each processor works on a subset of data, we can say that these methods use the data partitioning approach.

4.2 Implementation

In this sub-section we describe the algorithm used in the implementation of the kernel and the rationale behind it. We present pseudocode and comment on it. Our goal is to familiarize the reader with our implementation details at a low level.

Each of the N processes spawned executes the following -

RUN GENERATION PHASE BARRIER (N) MERGE PHASE	-- all N processes meet
--	-------------------------

The kernel consists of two phases - a run generation phase and a merge phase. In the run generation phase data are read in from the SortInputStream and memory-sized runs are created and stored in SortRandomStorage. In the merge phase the intermediate runs are read in from the SortRandomStorage, merged, and then written to SortOutputStream. This is called a two-pass sort.

In general more than two passes over the data may be required for an external sort. The implementation is restricted to a two-pass sort for the current study as it is sufficient for most cases [5,6] and exhibits all the

characteristics of an external sort. It can be generalized to a multi-pass sort easily.

Figure 4.1 How much of data can be sorted in two passes ?

Figure 4.1 illustrates the amount of data that can be sorted in two passes given the size of main memory. Let M be the memory size in bytes and B be the size of blocks in bytes. Then each run is of size M and contains M/B blocks. A two-pass sort is then sufficient to sort $(M/B) \cdot (M/B) \cdot B$

bytes of data, where M/B is the size of each run in blocks and is also the number of runs for a two-pass sort (See Figure 4.1 above). For example, with a B of 4096 bytes and memory of 1 megabyte, 256 megabytes of data can be sorted. With the same block size and 10 megabytes of memory, 25 gigabytes of data can be sorted.

4.2.1 Run generation phase

In this sub-section, we present the pseudocode of the run generation phase and comment on it. The purpose of the run generation phase is to read all of the data from `SortInputStream` and generate and store memory-sized runs to `SortRandomStorage`. `SortInputStream`, `SortOutputStream` and `SortRandomStorage` are ADTs defined in Chapter 3. Figure 4.2 shows the merge tree for the run generation phase. A process could be working in any part of this figure at a given point in time, with appropriate synchronization with other processes. All of the data structures are in shared memory. The pseudocode for the run generation phase is shown in Figure 4.3.

SE : Stream element
RAV : RecordAddressVector
M : Merge node
The arrows indicate data flow.

Figure 4.2 : Run generation phase

Each process (UNIX process) spawned executes the following in the run generation phase - reads an InputBlock worth of data, forms an array of record addresses called the RecordAddressVector (RAV), sorts the RAV using quicksort, forms a stream element (SE) with the sorted RAV and some data structures for synchronization purposes, inserts the SE as the input to a leaf level merge node. The merge nodes form a binary tree connected by SEs, i.e. the output of a leaf merge node is a stream of SEs, that form the input for the higher level merge node.

RUN GENERATION

PURPOSE

Transform data from SortInputStream into sorted runs stored in SortRandomStorage

PSEUDOCODE

Initialize data structures (1 process only)

While there is an InputBlock to be read from SortInputStream do

 If sufficient memory available and not EOF then

 Read an InputBlock.

 Form a RAV and then SE.

 Pair-up the SE into the merge tree at level 0

 Else

 Consolidate merge tree into one tree (1 process only).

 Set up the root of the tree etc. (1 process only).

 Merge and generate a run

Endwhile

Figure 4.3 : Pseudocode for the run generation phase

The pseudocode for inserting a stream element into the merge tree is shown in Figure 4.4. While inserting a stream element at the leaf merge

node, the inserting process checks within a critical section whether another such stream exists at that level. If one exists then it performs a merge of the two input streams to generate a stream at the output of the merge node. This stream is then paired with another such stream at the higher level. Pairing goes on until a level is reached where there is no other single stream to pair with. In that case, the process creates a merge node, attaches the single stream to it and returns to begin another cycle of reading an InputBlock and inserting a stream into the tree etc. Each process follows the same algorithm and one could be doing I/O while the other is quicksorting or merging records.

PAIR-UP SE AT LEVEL L **PURPOSE**

To insert the *SE* at level L of the merge tree

PSEUDOCODE

For I from level L to *maxlevel* do

 If no unpaired merge node exists at level I then

 Create a merge node at level I .

 Insert *SE* at I as child1

 return

 Else

 Insert *SE* as child2

 Perform a merge at level I producing a *SE* for higher level merge node.

Endfor

Figure 4.4 : Pseudocode for inserting a SE into the merge tree

As each InputBlock is read it is added to the merge tree, performing merges from the leaf node at which it is added to all the way up to the top. Every new InputBlock forms a new stream to one of the leaf merge nodes. This way, by the time the last InputBlock is read in, the merge tree is almost complete and records can be output as soon as the last block is read in. Since records cannot be output by a sort before exhausting its input, the time between the completion of reading of the input and the start of the output should be minimal for the sort to be efficient.

Figure 4.5 : A Merge node

Figure 4.5 shows a merge node of the tree along with the two input and one output streams. Merging at every merge node is performed in two

phases. A process reserves a certain number of records from the two input streams of a merge node in a critical section (exclusive access through locking) and then does the copying and merging in parallel with other processes. Different processes could be performing merges in parallel at the same merge node on the sections of the input streams they reserved. This is particularly useful when a sub-tree is hot due to skew in the data. Here the copying refers to the copying of record pointers and not whole records. This is a critical portion of our algorithm. Since multiple processes perform the merge, they need to reserve the records to be merged in a critical section and this critical section should be small for performance reasons. Thus, for the merge to be efficient the amount of time spent reserving records should be much smaller than the copying time. Processes allocate records from the two input streams using a binary search and thus the time for the reservation of records is $O(\log N)$. The time for merging and copying is $O(N)$. In general, allocating thousands of records per merge will be optimal since the logarithm of a thousand is much less than a thousand. Also, the amount of records chosen to merge should not be too large so that maximum parallelism can be exploited. Making N too large will result in a load imbalance. For example, the last process to finish up with a run generation will hold the

other processes at the barrier before next run for too long. Pseudocode for the allocations of records at a merge node is shown in Figure 4.6.

ALLOCATE RECORDS FOR A MERGE

PURPOSE

To allocate records from the two input streams of a merge node using binary search.

PSEUDOCODE

Limit number of records from each stream to be RECS_PER_MERGE.

If both streams are empty then -- used only by the merge phase

Mark the smaller of the first unread keys from the two streams as the first unread key of this node. (See section 4.2.2.2)

Else If either one of the streams is empty then -- used only by the merge phase

Choose records from the other stream using binary search such that they are smaller than the first unread key in the empty stream. (See section 4.2.2.2)

Else if both streams are not empty then

Choose $r1$ and $r2$ records from the two streams using binary search such that $(r1-1)$ th rec in stream1 $<$ $r2$ th rec in stream2 and $(r2-1)$ th rec in stream 2 $<$ $r1$ th rec in stream1.

Walk through the left stream and the right stream reserving $r1$ and $r2$ records respectively.

Figure 4.6 : Pseudocode for the allocation of records for a merge

In the case of external sorting, InputBlocks are read in until memory is exhausted, after which a sorted run is generated by performing merges from the root merge node to the bottom of the merge tree. The merge tree to perform the next merge while traversing top-down is based on the last keys that the sub-trees produced. The one that produced the smaller

one in the collating sequence is chosen. Pseudocode for performing merges from top to bottom is shown in Figure 4.7.

MERGE FROM TOP TO BOTTOM AT NODE M

PURPOSE

To start from node M in the merge tree and perform merges top to bottom.

PSEUDOCODE

While true do

 Allocate records for a merge at node M

 If records available for merge then

 Merge and copy records reserved

 Else if last stream elements (dummy) from both input streams reached then

 Generate last stream element (dummy) at the output of node M

 return

 If no children exist then

 return

 Else

 Pick M to be the child that generated the smaller last record.

Endwhile

Figure 4.7 : Pseudocode for performing merges from top to bottom.

Records from the root node of the tree are written to the SortRandomStorage or SortOutputStream depending on whether the sorting is external or internal. Pseudocode for merging and generating a run is shown in Figure 4.8. The RAVs from the output stream of the root node are used to copy records to a pool of LargeBlocks. This is the granularity at which data are written to the SortRandomStorage and SmallBlocks is the granularity at which data are read in from the

intermediate storage. LargeBlocks are multiples of SmallBlocks and both are further made up of pages which is the granularity of recycling memory. See section 4.3 for the description of the various block sizes.

MERGE AND GENERATE A RUN

PURPOSE

To produce SEs at the root node and store the records corresponding to the SEs into SortRandomStorage/SortOutputStream.

PSEUDOCODE

```
While root SE not the last stream element (dummy) do -- root stream could be empty
  If root stream empty then
    Generate SEs at root node by performing merges top to bottom
  Else
    Select the top SE from the root stream.
    If internal sort (EOF reached and run number is 0) then
      Call interface to write to SortOutputStream
    Else
      Copy records to LargeBlocks saving forecasting table entrees.
      Flush LargeBlocks to SortRandomStorage if full.
      Generate more SEs at root node by performing merges.
Endwhile
```

Figure 4.8 : Pseudocode for merging and generating a run

The processes alternate between copying and merging from top to bottom until all of the input is exhausted. The process that is last to fill up a LargeBlock flushes it to the SortRandomStorage. In the case of internal sorting (reaching end of file (EOF) on input before exhausting the memory), the SEs from the root node of the merge tree are directly

written to the `SortOutputStream`. RAVs are directly passed to the interface that is responsible for copying.

In the case of external sorting, all the processes synchronize at a barrier after every run generation. The same algorithm continues until EOF is reached at the input, after which the last run is generated and flushed to the intermediate storage.

During the run generation phase, the following is done to facilitate the merge phase (described in the following section) -

- 1) A list of addresses of first `SmallBlocks` of each run is generated to help in building the merge tree in the merge phase.
- 2) A forecast table is generated with an entry for each `SmallBlock` (except for the first `SmallBlocks` of each run). Each entry contains the key of the first record in the `SmallBlock`, the address of the `SmallBlock` in disk and the run number. The address in the disk is an abstraction provided by the interface and is called the `DiskBlock`.

4.2.2 Merge phase

In this sub-section we present the pseudocode of the merge phase and comment on it. We also describe the rationale behind the choice of different block sizes for reading/writing from/to `SortInputStream`, `SortOutputStream` and `SortRandomStorage`.

The purpose of the merge phase is to merge runs from the `SortRandomStorage` and write the single stream of sorted data to `SortOutputStream`. The merge phase comes into the picture only in the case of external sorting. There is a barrier between the run generation and the merge phases where all the processes synchronize. After some initial set-up, one process sorts the forecast table while others build the merge tree. The merge tree is similar to that of the run generation phase and now each run forms an input stream to one of the leaf merge nodes. Figure 4.9 shows the merge tree for the merge phase. Pseudocode for the merge phase is given in Figure 4.10. The building of the merge tree is also similar to that in the run generation phase, each process reads a `SmallBlock` from the list of first blocks, generates a `RecordAddressVector`, forms a stream element SE from the RAV, inserts the SE into the merge tree at a leaf merge node, performs merges from the leaf merge node to the level at which there is no stream to pair with,

SE : Stream element
 RAV : RecordAddressVector
 M : Merge node
 The arrows indicate the data flow.

Figure 4.9 : Merge phase

and then starts over the whole cycle until the list of first blocks is exhausted. The tree is then consolidated into one single merge tree as there could be merge nodes at multiple levels without the two input streams (consider an odd number of leaf merge nodes).

MERGE RUNS

PURPOSE

To merge the runs created in the run generation phase and write sorted output to SortOutputStream.

PSEUDOCODE

Initialize data structures (1 process only)

If last process then

 Sort the forecasting table using quicksort

Else

 Build a merge tree using list of first blocks generated in run generation phase.

Read SmallBlocks from intermediate storage

Perform merges to generate sorted output stream

Figure 4.10 : Pseudocode for the merge phase

The kernel uses a forecast table to perform read aheads in the merge phase. It is superior to the other two methods of reading ahead - double buffering [6] and a single forecast buffer [12,14,15]. With double buffering, two input buffers are allocated per run. All of the data read ahead may not be immediately useful (consider skewed data) and also the number of runs that can be merged at a time reduces to half. With a single forecast buffer, read ahead is limited to just one buffer. A forecast

table contains a list of blocks in the order in which they need to be read. Also the memory requirements of the merge phase may be much less than the run generation phase and the additional memory may be well utilized with the help of a forecast table. Parallel sorts require parallel input capabilities even when the data are badly skewed and a forecast table allows for this. Without a forecast table, only one buffer's worth of records can be read ahead as it's last key is used for predicting the next buffer for read ahead.

The forecast table contains one entry per SmallBlock, excluding the first SmallBlock from each run. Each entry contains the first key in the SmallBlock, the run number and the DiskBlock information corresponding to the SmallBlock. The table is sorted by the first key in the entries using quicksort. In the current implementation, the forecast table resides in memory and could use considerable amounts of memory in the worst case when the key length is large. A simple alternative could be to write the entries to SortRandomStorage during the run generation phase and read them back in for sorting. After sorting, the table could either reside in memory, as the memory requirements of the merge phase is smaller than the run generation phase, or can be written out to the intermediate storage. In our implementation it was decided to keep the

forecast table in memory for simplicity. Also, for cases where the record lengths are much larger than key lengths, the size of the forecast table is not very large. For example, say we have a two-pass sort with record length 100 and key length 10 and cardinality 1,000,000. Then the size of the dataset is 100MB. Lets say that a SmallBlock is of size 16KB and can accommodate 163 records. Therefore the forecast table will contain approximately 6135 entries (we have included the first SmallBlocks of the runs also for simplicity). If each entry occupies 24B of memory, then the size of the forecast table will be 147KB, much smaller than the 100MB dataset.

After the merge tree is built and the forecast table is sorted, the processes synchronize and execute the following - perform reads from the SortRandomStorage using the information in the forecast table and write a sorted stream of records from the root of the merge tree while performing merges from the bottom to the top of the tree. The processes alternate between the two steps until all of the input from the runs are exhausted.

4.2.2.1 Reading from the SortRandomStorage

The amount of memory left after the tree building phase, which consumes one SmallBlock worth of memory per run, is called the free staging area. This free memory could be used for reading ahead useful data from the SortRandomStorage. Figure 4.11 describes the pseudocode for the reads from the SortRandomStorage.

There is a list of block information attached to each run. Each process reads the forecast table in a critical section and updates the block list for the appropriate run. They continue to do so until pages are available in the free staging area. On exhausting the same, they perform the reads from the SortRandomStorage using the list of blocks associated with a run. The processes choose a run to read based on a shared index on the runs. On choosing a run to read, the process reads all of the blocks currently in the list before moving on to the next run.

The kernel optimizes the reads from SortRandomStorage by combining contiguous blocks into a single read. The interface provides the kernel with a routine to check for contiguity. Since LargeBlocks are a multiple of SmallBlocks, reads can be in units of LargeBlocks in the best case

when there is enough memory available to read from the run. If LargeBlocks are KMAX times SmallBlocks, then reads can be combined up to a maximum of KMAX SmallBlocks or however many are contiguous and available for reading depending on the free memory. Having a shared index and exhausting all of the forecasted reads for a run before moving on to the next run allows the forecast length to be maximal, giving higher average blocking factors for the reads.

READ SmallBlocks FROM SortRandomStorage

PURPOSE

To read runs from SortRandomStorage as per the forecast table efficiently.

PSEUDOCODE

While not end of forecasting table and memory available > SmallBlock do

 Get the next entry from the forecasting table

 Add DiskBlock info to the read list of the corresponding run.

Endwhile

While runs visited < total number of runs or enough blocks not read do

 Choose a run for reading using a shared index

 While the read list for the run is not exhausted do

 Combine contiguous SmallBlocks from the read list up to KMAX

 Perform a single read for the contiguous blocks.

 Form a stream element and add to the merge tree.

 Get the first unread key info from the next entry in the forecast table for the run.

 Endwhile

Endwhile

Figure 4.11 : Pseudocode for reading from the SortRandomStorage

When multiple devices are used for the SortRandomStorage, the interface can allocate LargeBlocks in a random fashion to avoid worst case merge patterns from being on the same device for long periods of time. Also, striping data in this fashion can increase read bandwidth. With combining of reads and careful disk block allocation, the forecast table allows extra memory to be used to increase the blocking factor and speed up random reads.

4.2.2.2 Merges in the merge phase

The algorithm for performing parallel binary merges in the merge phase is similar to that of the run generation phase as the structure of the merge tree is similar. In the run generation phase, while building the tree, merges are performed from bottom to top. It is along the branch where the data are read in. Once the tree is complete, i.e. all the reading for a run is complete, merges are performed from top to bottom. The choice of the sub-tree to perform the next merge is made on the basis of last key produced so far by the sub-tree. Since all of the data are available in the tree, an incorrect decision gets corrected in the next iteration.

The building of the tree in the merge phase is similar to that of the run generation phase. Once the tree is complete, the merges are performed from bottom to top choosing the run to start with in a round-robin fashion. This is intuitive since data are constantly read in at the leaf merge nodes and can be propagated to the top of the tree. This is similar to the merges performed during the tree building phase in the both run generation and the merge phases. Pseudocode for the merges from bottom to top is presented in Figure 4.12.

MERGE FROM BOTTOM TO TOP

PURPOSE

To perform merges from a leaf node to the top of the merge tree.

PSEUDOCODE

Pick a run for merge in a round-robin fashion

While true do

 Allocate records for merging at node M

 If records available for merge then

 Merge and copy records reserved

 Else if last stream elements from both input streams then

 Generate last stream element for the output of node M

 If parent of node M available then

 let M point to the parent node.

 else

 return

Endwhile

Figure 4.12 : Pseudocode for merges from bottom to top

In the merge phase, since all of the data are not available in the merge tree while performing merges, care needs to be taken to avoid deadlocks. For example, in the run generation phase it was decided to allocate a minimum number of records to perform merges at the merge nodes, so that the overhead of allocation is much less than the actual merge and copy. In the merge phase, such a decision could result in a deadlock when memory conditions are tight (just enough for a two-pass sort). For example, the data could be staggered in different sections of the tree not allowing for a merge at any node due to the non-availability of a minimum number of records and at the same time no more data can be read in due to the non-availability of memory. Figure 4.13 shows a simple case of deadlock. Say the memory can hold 2000 records and a minimum of 500 records is required for a merge. Then merging cannot proceed at node B for the lack of a minimum number of records in its inputs and no further records can be read because the memory is full. So the decision was made to merge whatever is available rather than waiting for a certain minimum. This did not make much of a difference in the average number of records reserved per merge.

Figure 4.13 : Deadlock scenario 1

A classic problem with merging ordered data is that the data needs to be available at both the left and the right streams before any merging can proceed. Since data are read in as per the forecast table, with skewed data one of the runs could become dry. A whole sub-tree could become dry. This can result in a deadlock situation, where one cannot merge due to the non-availability of data in the required streams and one cannot read due to the non-availability of memory. Figure 4.14 illustrates a simple such deadlock scenario. Say the memory can hold 2000 records. Merging cannot proceed at node B and consequently at node C as both input streams are not available and no further records can be read because the memory is full. Such a deadlock situation is avoided by the use of the forecast table. Since it has the first key information for each

SmallBlock, each run can be marked with the first key that is unread so far (See Figure 4.6). This information can be propagated up the tree and merges can proceed depending on this information when streams become dry.

Figure 4.14 : Deadlock scenario 2

The output stream from the root of the merge node is provided to the interface, again in a two-staged approach, reserving in a critical section followed by writing to the `SortOutputStream`. The interface can choose to copy data into a buffer before writing, or use scatter/gather writes/reads system calls to write to the output device. It can also choose to distribute data between multiple devices resulting in a range-partitioning of data. Figure 4.15 describes the pseudocode for the writing of sorted data to `SortOutputStream`.

GENERATING SORTED STREAM OF DATA AND WRITING TO SortOutputStream

PURPOSE

To produce a sorted stream at the root node by performing merges and writing to SortOutputStream.

PSEUDOCODE

```
While root SE not the last stream element (dummy) do    -- root stream can be empty
  If root stream from the merge tree is empty then
    Generate stream element at root by performing merges bottom to top
  Else
    Pick the top stream element from the root stream
    Write to SortOutputStream.
    For each record address in the stream element do
      If record address points to last record in the page then
        mark the page as free for recycling
    Endfor
  If reading from SortRandomStorage not done yet then
    Read SmallBlocks from SortRandomStorage
  Generate SEs at root by performing merges bottom to top
Endwhile
```

Figure 4.15 : Pseudocode for generating the sorted stream of records and writing to SortOutputStream

4.3 Block sizes in the sort

The InputBlock is the unit of reads from the SortInputStream and the OutputBlock is the unit of writes to the SortOutputStream. InputBlock and OutputBlock sizes are fixed by the interface. The sizes are in units of pages. The interface may set the block sizes depending on the input and the output devices.

Each InputBlock forms an input stream to a leaf node of the merge tree. If the size of the InputBlock is too small, then the input streams at the leaf nodes contain few records, increasing the height of the tree. Merges near the leaves of the tree may not be efficient. On the other hand, if the size of the InputBlock is too large and I/O is sequential, the end effects may become significant. For example, while one process is reading the first InputBlock, the other processes cannot do anything useful. The size of the blocks should also be such that the I/O is efficient. For disks, generally a block size between 8KB and 64KB results in a good I/O performance. Most systems have a limit on the maximum bytes that can be transferred with a single I/O call, beyond which two or more calls are issued anyways.

Both InputBlocks and OutputBlocks were chosen to be 16KB in our implementation. Since the input and the output devices were disks in our experiments, the choice of 16KB was primarily for I/O efficiency.

SmallBlock is the unit of reads and LargeBlocks is the unit of writes to SortRandomStorage. The kernel chooses the SmallBlock and the LargeBlock sizes and the interface provides the minimum and maximum

block sizes that can be used for reads (writes) to the SortRandomStorage.

SmallBlocks determine the entry of a forecast table. Reads during the merge phase are performed by referring to the forecast table, the unit of reads being SmallBlocks. So, the amount of data read each time depends on the number of entries in the forecast table. For a smaller table, more data are read each time and vice-versa. Reading more data each time requires more memory and all of the data read may not be immediately useful. Consider the case where the first record from the first SmallBlock is the first record in the sorted output stream and all the rest from the same SmallBlock belong to the end of sorted output stream. Thus having more entries in the forecast table reduces the memory requirements per read and also reduces the wastage of space resulting from reading records not immediately useful. On the other hand, having too many entries in the forecast table may result in a very small block size for reads causing the I/O to be inefficient. The reads being random during the merge phase, reading in smaller block sizes will result in larger number of seeks causing an inefficient I/O. This is mostly solved by combining reads while reading from SortRandomStorage as mentioned in section 4.2.2.1. With forecasting and combining reads, we always read

the optimal size block (within the limits of the operating system and hardware). We still have the overhead of the large forecasting and CPU processing to set up the reads as SmallBlock size decreases.

In our implementation, the size of a SmallBlock was fixed at 16KB. Since disks were used for SortRandomStorage this size of 16KB resulted in an efficient I/O. Having a larger block size would decrease the number of entries in the forecast table increasing the wastage of memory by reading records that may not get consumed immediately.

LargeBlocks is the granularity of writes to SortRandomStorage. As mentioned above, I/O is efficient for larger block sizes. By allocating the disk space in large chunks, we greatly increase the chances that the disk blocks will be contiguous. This enables optimization during reads, where multiple SmallBlocks can be combined into a single read when contiguous in the disk. The data in the SortRandomStorage can also be striped across multiple physical devices. In such a case the block size for writing can be same as the stripe width. Having a small block size for writing to SortRandomStorage does not have any advantages and hence it is best to use a large block size.

In our implementation the size of a LargeBlock was fixed at 64KB, the maximum allowed by the interface. Again, since disks were used for the SortRandomStorage, this size resulted in efficient I/O. The SortRandomStorage was striped across multiple disks in our experiments and the stripe width was 64KB. Thus by setting the sizes of LargeBlocks and SmallBlocks to be 64KB and 16KB, reads of 4 SmallBlocks can be combined into a single read in the best case.

CHAPTER 5

PERFORMANCE

5.1 System Configuration

Performance tests were conducted on a Sequent Symmetry running PTX 4.1 operating system (OS). This system had eight 486-25mhz processors and 200MB of main memory. The input and output data streams were stored as UNIX files and the intermediate storage was a raw device. Data in both the UNIX files and the intermediate storage was striped across eight disks, four disks on each of the two channels using SVM (Sequent Volume Manager). The first 1/8th of each physical disk was part of logical disk 1, and so on. The stripe factor was 64KB. SVM is transparent to the application and is a layer between the OS and the disk sub-system.

5.2 Profiling

A tool *ggprof* [21] that could profile multiple processes simultaneously was used for profiling. This tool uses the Sequent microsecond clock. Figure 5.1 depicts the results of profiling a sort of 32MB of data with 7 processes and 4MB of main memory. The times are wall-times.

Figure 5.1 : Execution time distribution

Each process spent approximately -

- 50% of the execution time on I/O.
- 14% of the execution time on various activities like parsing, memory management, initialization, building the merge tree, building RecordAddressVectors, reserving blocks in the input, output and random storage devices for I/O and various other book-keeping purposes.
- 10% of the execution time making comparisons.
- 10% of the execution time copying record pointers or records themselves.
- 10% of the execution time waiting on a barrier.
- 6% of the execution time spinning on a lock.

- 0.14% of the execution time allocating records for a merge in the tree (critical section).

The following paragraphs present an analysis of the data obtained by profiling. The results fall in line with our expectation.

On an average, 50% of the time was spent performing I/O. Since synchronous reads/write were used and 7 processes were running on an 8 processor machine, there was no overlap of CPU and I/O work. During the initial design phase of the project, using an asynchronous I/O subsystem was considered. Though PTX provides asynchronous system calls for I/O, some operating systems like HPUX do not provide such calls. So, it was decided against using asynchronous calls for portability reasons. Also, since the model used for parallelization follows the *everybody does everything paradigm*, one could over-decompose and use more processes than processors for overlapping CPU and I/O work. But, with over-decomposition the system runs into convoy problems with spin locks, where a process that is holding a spin lock gets timed out and others simply wait on the same lock during their time-slice. Also, having more processes than processors introduces extra context switches worsening the locality of data. So, we restricted ourselves to at most 7

processes in our experiments leaving out one processor for the OS to use.

The run generation phase consists of two steps - 1. Reading data from the SortInputStream and inserting the same into the merge tree. 2. After building the tree, merging records and writing the sorted stream from the root node to the SortRandomStorage. There are two barriers, one before and one after step 2, that consume about 10% of the execution time. All processes wait at the barrier till the last process checks in. These barriers were introduced for two reasons - simplicity and isolation of different sections. Separating the tree-building phase from the merge helped in memory management as well. All of the memory was used by a single run at any point during the run generation phase. This made memory management simple as each run relinquished all of the memory in one shot for use by the next run. These barriers are not mandatory and can be avoided at the cost of extra complexity. Another barrier that consumed about 1% of the execution time is in the merge phase. In this phase, after initial set-up, one process sorts the forecast table while others build the merge tree. There is a barrier between sorting of the forecast table and the merging of the runs. This is essential as the single process that does the sorting of the forecast table cannot begin merging

until the tree is complete and the processes that build the tree cannot do any further reading from SortRandomStorage until the forecast table is sorted.

Each process spent about 6% of the execution time spinning on a lock. This is not much of an overhead and is mainly due to the fine granularity of parallelization used. Spin locks are better when the critical sections are small as it is better to spin than to switch context from one process to another. Since all of our critical sections were fairly small, spin locks did not take away much of the execution time.

Making comparisons typically takes up much of the sorting time. As mentioned in section 4.5, allocating records for a merge can take a significant amount of time unless a binary search is used and the number of records allocated is such that the time for allocation is much less than the time for actual merge and copy. As shown by the profiler, each process spent on an average 0.14% of the execution time allocating records for merges in the merge tree and about 10% comparing and copying record pointers along the tree. This is as we expected and allocation of records using binary search in a critical section is not much of an overhead as compared to merging itself.

5.3 Experimental results

All experiments were performed with 128-byte records and 4-byte keylengths. Keys were of type character (i.e. comparisons were performed using `memcmp()`). We present the results of scale-up, speed-up and size-up experiments below. We also provide a comparison of PSUsort with the `/bin/sort` UNIX utility. All timings are wall-times.

5.3.1 Scale-up

Linear scale-up is achieved when N times as many resources can solve a problem with N times as much data in the same amount of time [15]. The resources were increased in steps of 1MB of memory and 1 process and the cardinality of the sort was increased in units of 50,000 records. Tables 5.1 show the results of scale-up experiments. Figure 5.2 shows the scale-up graph. The ideal behavior is shown to be a constant line considering the dominating I/O cost, which is linear. As seen from the graph our implementation does not scale linearly. The efficiency of the sort drops down to about 50% with 7 processes. The reason for the deviation from the ideal behavior is the following -

- Parallelism could not be achieved at the I/O level. The UNIX file system was used for the `SortInputStream` and the `SortOutputStream`.

It held a lock (the inode lock to be specific) on the file during I/O. Thus, in spite of having the data striped across multiple disks, the I/O was essentially sequential. Thus, as the number of records increased, so did the time for I/O regardless of the increase in resources. The current trend in the database world is to fragment the data across multiple devices so that reads can be in parallel. This can be done with our implementation by having the input data spread across multiple devices and modifying the interface to read from them in a round-robin fashion.

Memory in MB	# of Procs	Cardinality	Time in secs	actual/ideal
1	1	50,000	62	1.00
2	2	100,000	70	1.12
3	3	150,000	80	1.29
4	4	200,000	97	1.56
5	5	250,000	120	1.93
6	6	300,000	126	2.03
7	7	350,000	138	2.22

Table 5.1 : Scale-up results

Figure 5.2 : Scale-up Graph

The behavior of the sort is a step function, i.e. the time to sort does not differ much as long as two passes through the data are required. Only when the amount of memory is increased to an amount that is sufficient for a single pass will the timing differ drastically. This is because the I/O time depends mainly on the number of passes through the data. Hence, increasing the memory in steps of 1MB did not make much difference as 2 passes were required through the data anyway. There are optimizations, like saving the last run [15], that change the step function nature of the sort. We did not explore such optimizations in our study.

5.3.2 Speed-up

Speed-up results indicate the increase in speed with increase in resources. N times as many resources should solve a constant-size problem in $1/N$ of the time [15]. Again, memory was increased in units of 1MB along with the increase in number of processes from 1 to 7. The cardinality was kept fixed at 350,000 records. Table 5.2 show the results of speed-up experiments. Figure 5.3 shows the speed-up graph. The curve again indicates a loss in efficiency as more resources are added, the efficiency dropping down to about 50% with 7 processes and 7MB of memory. The reason for a non-linear speed-up is the following -

- Linear speed-up could not be achieved due to the use of synchronization primitives and sequential I/O. According to the results produced by the profiler (see section 5.2) for a sort using 7 processes, each process spent on an average 15% of the execution time waiting on a synchronization primitive. The contention for locks increases with increase in number of processes. Also, since the I/O was essentially serial, adding more processes did not contribute much towards reducing the I/O time.

Memory MB	in	# of Procs	Time in secs	speed-up	% efficiency
1		1	456	1	100
2		2	242	1.88	94
3		3	197	2.31	77
4		4	165	2.76	69
5		5	157	2.90	58
6		6	145	3.14	52
7		7	136	3.35	48

Table 5.2 : Speed-up results with cardinality fixed at 350,000 records

Figure 5.3 : Speed-up Graph

5.3.3 Size-up

Linear size-up is achieved when N times the amount of data can be sorted in N times the amount of time when resources are unchanged. Cardinality was increased from 50,000 to 350,000 in steps of 50,000 with resources fixed at 7MB memory and 7 processes. Table 5.3 shows the results of size-up experiments. Figure 5.4 shows the size-up graph.

The graph is not linear from cardinality 50,000 to 100,000 and is almost linear from there onwards. The reason for the initial non-linearity is that with cardinality 50,000 the sort is internal and for the other cardinalities in the plot it is external and requires two passes through the data. The graph also shows a slight non-linearity at cardinality 300,000 which we believe is due to an anomaly in the system.

Cardinality	Time in secs	Ratio
50,000	15	1.0
100,000	40	2.6
150,000	59	4.0
200,000	81	5.4
250,000	102	6.8
300,000	112	7.5
350,000	140	9.3

Table 5.3 : Size-up results with 7 processes and 7MB of memory

Figure 5.4 : Size-up Graph

5.3.4 Comparison with /bin/sort utility

Table 5.4 and Figure 5.5 show the comparison between PSUsort and the standard UNIX utility /bin/sort. PSUsort is significantly faster than /bin/sort. With similar resources (100MB, one process) it is 3 times faster. With more resources (7 processes) it is 7-8 times faster. With significantly fewer resources (10MB) its performance is similar to that of /bin/sort.

		16MB data size	32MB data size	64MB data size
	Memory available	Time in secs	Time in secs	Time in secs
7 processes	100MB	24	49	98
	10MB	52	95	180
3 processes	100MB	29	54	115
	10MB	88	161	315
1 process	100MB	75	152	315
	10MB	172	329	665
/bin/sort	All available	164	388	827

Table 5.4 : Comparison with /bin/sort utility

Figure 5.5 : PSUsort vs. /bin/sort

We conclude the following from our experiments - PSUsort exhibits good, but not perfect, speed-up (50% of ideal) and scale-up (50% of ideal). It can be made to scale linearly if parallelism can be achieved at the I/O front and some of the barriers are avoided. The spin locks consumed only 6% of the execution time in the particular experimental case and should not be a concern. Locking is at a fine granularity and the critical sections are small. Also, with more tuning we believe that our results can be improved.

CHAPTER 6

SUMMARY

Sorting is an important operation performed by a DBMS. The volume of data managed by a typical DBMS is approaching gigabytes and terabytes. The amount of main memory available for the sort operation is typically in megabytes, hence the need for an external sort. Also, with multi-processor machines being available, most of the operations performed by a DBMS need to be parallelized for performance reasons.

There are two basic methods of performing external sorts - the sort-merge model and the distributive model. There exists a duality between the two models in terms of CPU and I/O work as shown in chapter 2. For the distributive sort to be optimal, the data needs to be evenly distributed to the workers. This requires pre-processing of data to determine the quantiles received by each worker. An inaccurate estimation can cause multiple levels of overflow at one worker as compared to others leading to an adverse performance. The sort-merge model, on the other hand, involves a merge step that needs a complete pass over the data and may become a bottleneck. This bottleneck can be

alleviated by performing the merge operation in parallel and we present one such method in this thesis called parallel binary merges.

There are two techniques to parallelizing programs - the function partitioning approach and the data partitioning approach. With function partitioning, functions are assigned to workers and multiple functions could be performed in parallel by different workers. Since the order in which the functions need to be performed and the set of functions that can go on concurrently are known in advance, the scheduling of tasks is static. Load balancing is difficult to achieve since one function may take longer than another function. With the data partitioning approach every worker performs all of the functions but with a different set of data. The workers move from one function to another and follow the *everybody does everything* paradigm. Load balancing is easier as the workers can work on equal amounts of data. Also, the number of data points can be more than the number of functions in a given program. Due to these reasons our implementation uses the data partitioning approach.

The sort package consists of an interface and a kernel, separating the sort-independent issues into the interface. The interface provides ADTs for the input, output and intermediate storage along with details like

memory available, processes available, a compare function to compare two records, etc. The interface can be modified to suit different environments. The interface in turn reads from the user different parameters like I/O files, memory size, etc.

The implementation of the kernel is based on the sort-merge model. InputBlock sized blocks are read, and an array of pointers to the records in the InputBlock is formed which is then sorted using quick-sort and inserted into a binary merge tree as inputs to leaf nodes. These pointers are propagated up the tree by performing merges. On exhausting the memory, records from the top of the merge tree are written to the intermediate storage in units of LargeBlocks to form a run. Once all of the runs are generated, merging of the runs is performed by reading from the intermediate storage in units of SmallBlocks and forming a binary merge tree with blocks from each run attached to leaf nodes of the tree. The sorted stream from the top of the tree is handed off to the interface in units of OutputBlocks. See section 4.3 for the explanation of the different block sizes. Each worker goes through the same cycle - reading, sorting, merging, writing and so on. Different workers could be performing different or similar functions with a different set of data.

Data structures are locked at a fine granularity resulting in small critical sections.

Some of the optimizations incorporated in the design and implementation of PSUsort are the following -

- Using a forecast table for reading ahead useful data from the intermediate storage during the merging of runs.
- Making LargeBlocks a multiple of SmallBlocks so that reads from the intermediate storage can be combined when the blocks are contiguous. The algorithm for reading from SortRandomStorage using the forecast table is optimized for maximal forecast lengths and higher average blocking factors.
- Choosing the various block sizes for optimized disk I/O.
- Using a binary merge tree and parallelizing the merges themselves. A binary search is used to divide up the records in the two input streams of a merge node for parallel merges. By choosing the number of records reserved to be large, the overhead of a binary search in a critical section is reduced.

Experiments were run on a Sequent Symmetry with eight 486-25mhz processors running PTX 4.1 operating system. The profiling on the sort showed results as expected - 0.14% of the execution time was spent on

the binary search for allocating records for a merge as opposed to 10% of the time spent on copying and merging records in the merge tree. It was also observed that only 6% of the time was spent spinning on a lock due to the fine granularity of locking. 10% of the time was spent on barriers, which we believe can be reduced, but only by increasing the complexity of the program. The speed-up and scale-up results indicated a drop in efficiency of about 50% when the number of processes was increased from 1 to 7. The main reason for this drop is that our input and output streams were from one UNIX file each and the UNIX file system held a lock on a file during I/O making the same practically serial. Since about 50% of the time was spent on I/O, which did not get parallelized, the speed-up and scale-up curves were sub-linear. We believe that solving the I/O bottleneck by having multiple input and output files can make the program scaleable. A comparison with the `/bin/sort` UNIX utility showed that PSUsort was significantly faster.

Due to the data partitioning approach and the fine granularity of parallelism used, we believe that PSUsort can achieve near linear scale-up and speed-up. The complexity of the implementation is mainly due to the parallelism aspects and the algorithm is otherwise straight-forward.

Also, there is a heavy code re-use as the algorithms for the run generation and the merge phases are very similar.

FUTURE WORK

The program being complex due to the fine granularity of parallelism involved, a lot of time and effort was spent on the design of the algorithms and ensuring the correctness of the implementation. We could not spend as much time on the performance tuning as we would have liked to and it would be interesting to do the same in the future. We would also like to solve the I/O bottleneck to get the program to a near linear scale-up and speed-up. It would also be interesting to see when the bus in the shared memory architecture actually becomes a bottleneck. The program currently implements a two-pass sort and it should be fairly easy to extend the same to a multi-pass sort in future for generality. The code for handling variable length records is implemented and needs to be tested. It would be interesting to see the usage of PSUsort for different applications by varying the interfaces to the kernel. Other optimizations that could be tried with the implementation are - use of processor affinity to improve cache effects, other scheduling of merges in the merge tree to improve cache effects and conditional locking to remove lock waiting.

BIBLIOGRAPHY

1. "Parallel Tag-Distribution Sort" by J.L.Baer et al., *Computer Science Technical Report*, University of Washington, Seattle, January 1985.
2. "Performance comparison of Distributive sort and mergesort as external sorting algorithm" by A.I.Verkamo, *The Journal of Systems and Software* 10, 187-200 (1989).
3. "Parallel sorting algorithm for tightly coupled multiprocessors" by M.J.Quinn, *Parallel computing* 6 (1988) 349-357, North-Holland.
4. "Parallel sorting on a shared-nothing architecture using probabilistic splitting" by Dewitt et al., *Proceedings of the international Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, December 1991.
5. "Alphasort : A RISC machine sort" by Newberg et al., April 1993, revised October 1993, *SFSC Technical report* 93.2.
6. "Merging sorted runs using large main memory" by Betty Salzberg, *Acta Informatica* 27 (1989), 195-215, Springer International.
7. "Percentile finding algorithm for multiple sorted runs" by B.K.Iyer et al., *Proceedings of the 15th international conference on VLDB*, Amsterdam, 1989.

8. "An experimental study of parallel merge sort" by Richard J. Anderson, *Dept. of CS Technical Report 88-05-01*, University of Washington, Seattle, 1988.
9. "Parallel External Sorting in Volcano" by Goetz Graefe, *CU Boulder Computer Science Technical Report 459*, June 1990.
10. "Tuning a parallel database algorithm on a shared-memory multiprocessor" by Goetz Graefe et al., *Software Practice and Experience*, Vol 22 (7), 495-517 (July 1992).
11. "Fastsort : A distributed single-input single-output external sort" by Betty Salzberg et al., *Proceeding of the ACM SIGMOD Conference*, Atlantic City, NJ, May-June 1990.
12. "Model of floating buffering" by L.J.Woodrum, *IBM systems journal*, Vol 9, 1970.
13. "System issues in parallel sorting for database systems" by B.K.Iyer et al., *Technical Report RJ 6585* (November 30, 1988), *IBM Almaden Research Lab*.
14. "The Art of Computer Programming, Vol 3: Sorting and Searching", D. Knuth, Addison Wesley, Reading, MA., 1973.
15. "Query evaluation techniques for large databases" by G.Graefe, *ACM computing surveys*, June 1993.

16. "Guide to parallel programming" on Sequent computer systems, Second Edition by Anita Osterhaug, Editor, *Sequent Technical Publications*, 1989.
17. "An adaptive method for unknown distribution in distributive partitioned sorting" by Philip J. Janus and Edmund A. Lamagna, *IEEE Transaction on Computers*, Vol C-34, No. 4, April 1985.
18. "Practical Selectivity Estimation through Adaptive Sampling" by Naughton et al., *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 1990.
19. "DFSORT Application Programming Guide", *IBM SC33-4035-15*, Feb 1991.
20. "Comparison of External Sorting and Internal Sorting in Virtual Memory" by A.I. Verkamo, *Performance Evaluation* 6, 135-145, July 1986.
21. "*ggprof* ", code developed by Gary Graunke as an internal profiling tool for Sequent Computers, Inc.