Sorting Large Files on a Backend Multiprocessor*

Micah Beck

Dina Bitton

W. Kevin Wilkinson†

TR 86-741
March 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* Support for this work was provided by the National Science Foundation, under grant DCR-8410889, a Hewlett-Packard faculty development fellowship, and Bell Communication Research.

† Bell Communications Research, Morristown, NJ 07960

SORTING LARGE FILES ON A BACKEND MULTIPROCESSOR

Micah Beck
Dina Bitton

Computer Science Department, Cornell University, Ithaca, NY 14853

W. Kevin Wilkinson

Bell Communications Research, Morristown, NJ 07960

ABSTRACT. A fundamental measure of processing power in a database management system is the performance of the sort utility it provides. When sorting a large data file on a serial computer, performance is limited by factors involving processor speed, memory capacity and I/O bandwidth. In this paper, we investigate the feasibility and efficiency of a parallel sort-merge algorithm through implementation on the *JASMIN* prototype, a backend multiprocessor built around a fast packet bus. We describe the design and implementation of a parallel sort utility that may become a building block for query processing in a database system that runs on *JASMIN*. We present and analyze the results of measurements corresponding to a range of file sizes and processor configurations. Our results show that using current, off-the shelf technology coupled with a streamlined distributed operating system, three and five microprocessor configurations provide a very cost-effective sort of large files. The three processor configuration sorts a 100 megabyte file in one hour, which compares well with commercial sort packages available on high-performance mainframes. In additional experiments, we investigate a model to tune our sort software, and scale our results to higher processor and network capabilities.

Support for this work was provided by the National Science Foundation, under grant DCR-8410889, a Hewlett-Packard faculty development fellowship, and Bell Communications Research.

TABLE OF CONTENTS

1. Introduction

Sorting of large data files is one of the most important and frequently performed operations in database management. Output files generated by a report are usually sorted with respect to an attribute or a combination of attributes that are of interest to the users of the report. In mass-storage devices, files are often maintained sorted with respect to a key attribute in order to facilitate subsequent searching and processing. Database management systems also pre-sort files in order to eliminate duplicate records [SEL79, BIT83] or to process complex join and aggregate queries [SEL79]. Thus, one of the fundamental measures of processing power in a transaction management system is the performance of the sort utility it provides [ANO85].

In most systems, sort performance is not satisfactory. For large files, a sort may require hours of processing time during which it saturates the CPU and I/O devices. As a consequence, even in high-performance transaction management systems, a large sort will often be deferred to after-hours processing in order not to interfere with the execution of short interactive transactions.

What limits the performance of a file sorting operation, and how can this performance be enhanced? A theoretical machine with a 100 megabyte memory and a 50 nanosecond Compare instruction could sort one million 100 byte records in 1.5 minutes, including 30 seconds to read the file from a disk with a 3 million-byte per second transfer rate and 30 seconds to write it back to the same disk [ANO85]. In practice, file sizes exceed main memory capacity by several orders of magnitude, comparisons and record moves are slower, and the effective I/O bandwidth is well below the maximum bandwidth of the disk channel. In addition, general purpose operating systems impose processor overhead and limit the utilization of other resources. The designer of a sort utility must strive to minimize record moves in memory, overlap I/O latency with computation, and reduce disk seek times. Even then, the efficiency of file sorting on a serial computer with conventional mass-storage

remains severely limited. Parallel computation combined with high I/O transfer rates is a natural approach to overcoming these limits. Indeed, a number of recent database machine designs use parallel sorting as a fundamental building block for query processing [TAN83, SZH83, KIT84, KAM85].

In this paper, we investigate the feasibility of parallel file sorting on a backend multiprocessor machine. We describe the design of a sort utility that exploits parallel computation and high I/O bandwidth, and its implementation on the *JASMIN* multiprocessor. We consider two schemes which differ in the starting location of the unsorted data (Figure 1), and may lead to significant differences in performance. The first scheme corresponds to a *Backend Sort*, where a file initially located on the host is downloaded to a backend multiprocessor for sorting. The file is distributed to the the backend processors, which sort it and then return the sorted file to the host. The second scheme, a *Distributed Sort*, assumes that the source file is initially distributed across a number of disks attached to backend processors. The processors sort the file in parallel, then send it to the host or write it to a backend disk. Basically, the parallel sorting algorithm used in both schemes is the same, but the design goal of overlaping computation with the distribution of the data in one case (*Backend Sort*), is replaced with overlaping computation with disk access in the other (*Distributed Sort*).

The remainder of this paper is organized as follows. In Section 2, we describe the hardware and software architecture of the *JASMIN* system. In Section 3, we describe the parallel external sort algorithm, and the implementation choices that we have made – in particular the process structure and the buffering scheme. In Section 4, we present the performance of our algorithm as implemented on *JASMIN*. We analyze the execution time, processor utilization, and network traffic as functions of the file size and the multiprocessor configuration. In Section 5, we investigate a model for tuning our sort utility, and scale our results to faster hardware and higher degrees of parallelism. Finally, Section 6 contains our conclusions.

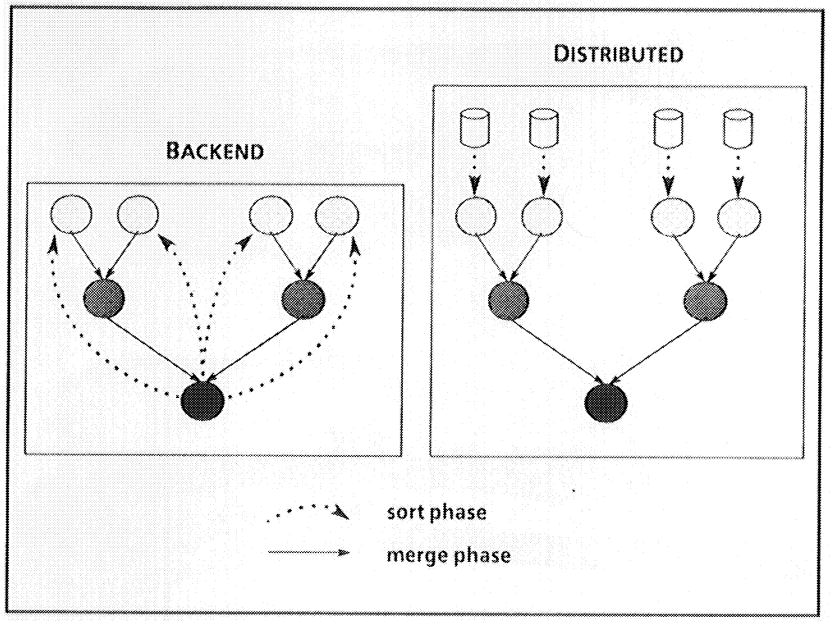

Figure 1: Backend and Distributed Sorting

2. A Backend Multiprocessor

Our parallel external sort algorithm was implemented on a prototype AT&T Bell Laboratories multiprocessor [AHU83]. This multiprocessor may be configured with up to 12 microcomputers communicating through a fast packet bus. It also has an umbilical connection to a DEC VAX host running UNIX† System V. The backend processors run the *JASMIN* distributed operating system kernel, which is under development at Bell Communications Research. *JASMIN* is designed to efficiently support distributed applications. In this section, we briefly describe the hardware characteristics of the multiprocessor and process management and communication in *JASMIN*. The reader is referred to [LEE84] for a complete description of the operating system and to [FIS84] for a description of a distributed database management system which runs on *JASMIN*.

†UNIX is a trademark of AT&T Bell Laboratories.

2.1. The S/Net

The multiprocessor used in *JASMIN* is built around the S/Net, a fast (80 Mbit/s) packet bus developed at AT&T Bell Laboratories. The multiprocessor consists of a set of Multibus-based microcomputers, each with an S/Net Processor Interface Board. The S/Net itself consists of a set of Buffer Interface Boards, tied together on a single backplane, each connected to a Processor Interface Board. A designated Buffer Interface Board is connected to the VAX host. The VAX downloads the microcomputers via the S/Net and provides some support, such as file access and debug output, for the execution environment. The microcomputers consist of a 10 MHz Motorola 68000 CPU and 1 Megabyte of main memory. Some of the microcomputers also have an SMD disk controller and a Fujitsu Eagle local disk.

While the total bandwidth of the S/Net is 80 Mbit/s, no single processor can achieve that speed. The maximum kernel level data transfer rate that a single processor can attain on the S/Net is approximately 300 Kbyte/sec, or 3% of the network's total capacity. Operating system overhead further reduces the effective data transfer rate between processes to 100 Kbyte/sec.

2.2. The *JASMIN* Operating System

The *JASMIN* distributed operating system is modeled after DEMOS [BAS77]. The basic object implemented by the system is an interprocess communication capability called a *path*, over which small, fixed length messages can be sent. Holding a path enables a process to send a message to the creator of the path. Additionally, bulk data transfers between processes may be accomplished by attaching a buffer to a path in the creator's address space and using kernel primitives to move data into or out of the buffer. The system allows any process to create paths and to pass them to other processes via messages.

A set of processes which share text and data address space, but have separate execution stacks, are said to form a *load module*. Since memory is shared by these processes, a

load module cannot span processors. While processes in the same load module share text and data space, their execution stacks inhabit different memory segments, they hold different sets of paths, and they receive messages from separate queues. In other words, paths and messages are exchanged between processes, not load modules. The load module notion is a convenience to permit cooperating processes to share memory.

JASMIN process scheduling is particularly simple. When created, each process is assigned a static priority level. Within each priority level, scheduling is strictly non-preemptive. A running process will execute until it performs a blocking system call, or until a higher priority process becomes ready-to-run. This guarantees mutual exclusion among a set of processes operating on shared data structures within a load module, as long as these processes are run at the same priority level.

3. A Parallel File Sorting Algorithm

Large files cannot usually be sorted in main memory, and so file sorting requires an *external sort algorithm*. Most external sort algorithms are based on iterative merging. They partition the file into subfiles that separately fit in memory, internally sort the individual subfiles, and then iteratively merge them into a single sorted file. During each iteration, every record is read and written once to mass-storage. These serial algorithms can be parallelized in a number of ways [EVE74, BIT82, KWA86].

3.1. The Algorithm

We have implemented a parallel N-way merge-sort algorithm adapted from [BIT82]. The algorithm assumes a logical tree connection between the processors (Figure 2). Each of the leaf processors has a disk attached to it, and thus can perform an external sort, independently of the host and the other processors in the backend. The parallel algorithm has two phases. We will refer to the first phase as the *sort phase*, and to the second as the *merge phase*. During the *sort phase*, each leaf processor creates fixed-length sorted subfiles from

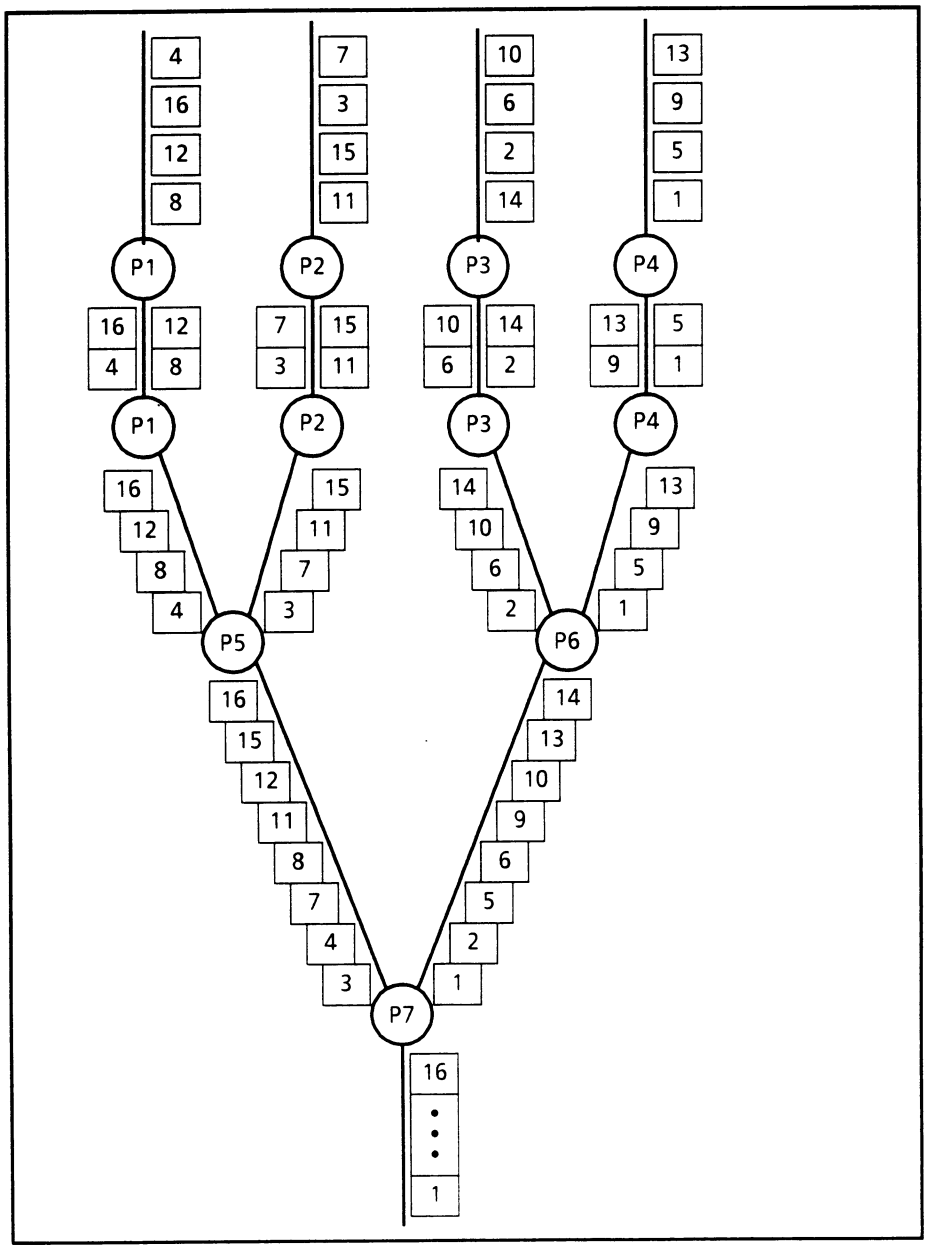

Figure 2: A Parallel Merge-Sort

its partition of the file; during the *merge phase*, these subfiles are merged in parallel. Parallelism in merging is exploited both through pipelining merge steps between levels of the tree, and through concurrent merging performed by processors on the same level of the tree. The two phases of the algorithm are described in more detail below.

During the sort phase, fixed-length sorted subfiles are created by the leaves as in a serial sort. Partitions of the file are sorted in memory using Quicksort, and then written to disk. Available memory is divided between a subfile and an array of pointers to records in that subfile. During Quicksort, data records are not moved in memory, only pointers are changed.

After the sort phase completes, each leaf merges its subfiles, producing a single sorted partition of the file. As it is produced, the partition is transferred to the leaf's parent in the merge tree. The parent processor merges the partitions it receives from its children and sends the result one level up in the tree. In our implementation, merge streams are transferred in blocks containing many records. While pipelining of successive merge phases *could* be at the record granularity, communication overhead makes such a design prohibitively inefficient. The complete sorted file is produced at the root of the merge tree.

It is interesting to note that, given a fixed topology for the merge tree, most of the steps in our sort algorithm are linear in terms of the number of comparisons performed. The partitioning of n records into n/b fixed-size sorted blocks of b records each takes $O(nlog(b))$ comparisons. Similarly, merging d sorted streams of n/d records takes $O(nlog(d))$ comparisons. Thus, at non-leaf nodes, where the degree of the merge is fixed by the topology of the tree, the merge is linear. At the leaves, the degree of the merge is equal to the number of temporary files, which is in turn proportional to the size of input. The merge phase at the leaves is thus the only point where the number of comparisons is non-linear. Because comparisons are such a small portion of the total work done in sorting a large file, even the uniprocessor configuration of our sort only shows significant non-linear behavior in file sizes over 25 megabytes (see Section 4.2).

3.2. Implementation on *JASMIN*

In order to concentrate on designing the parallel part of the sort utility, and have a good baseline for evaluating our implementation, we chose to start with a robust serial sort

3.2.3. Disk File Layout

The layout of files on disk is central to minimizing access time. We experimented with two approaches: contiguous files and interleaved files. In contiguous file layout, the logical disk blocks of a file lie in consecutive physical disk addresses. This means they will be physically contiguous unless they cross a cylinder boundary. In interleaved file layout, the first logical block of all files lie in one consecutive set of disk addresses, followed by the second block of all files, etc. (Figure 5).

Contiguous file layout minimizes seeking when reading or writing an entire sequential file. It maximizes seeking when multiple sequential files are to be accessed concurrently. Interleaved layout achieves the opposite: maximal seeking when accessing a single file sequentially; minimal seeking when accessing many files concurrently. Intermediate schemes which interleave files in segments larger than one logical block are possible, but we did not implement any. In our sort algorithm, temporary files are written one at a time during the sort phase, and then all are read concurrently during the merge phase. Thus, contiguous layout performs best during the sort phase, while interleaved layout wins in the merge phase.

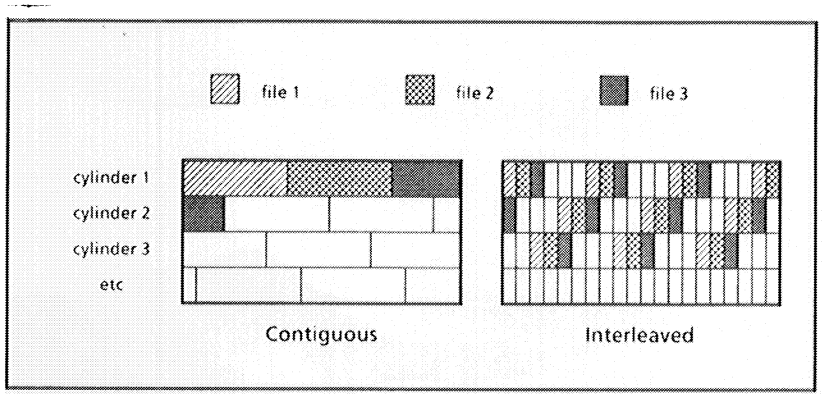

Figure 5: Contiguous vs Interleaved File Layout

program. We considered the *Linderman Sort*, a program recently developed at Bell Laboratories [LIN84]. The program has been extensively tuned, and achieves very good performance on UNIX. The algorithm implemented in Linderman's program is a polymerge sort [KNU73], where initial runs are generated with Quicksort. We started with adapting this code for sorting the subfiles in the sort phase of our parallel algorithm, and for merging subfiles on one processor. Then, we turned our attention to design issues that are critical to the performance of the parallel sort program:

(1) the process configuration and synchronization protocol

(2) the memory configuration (including buffer allocation),

(3) the layout of files on the local disks.

3.2.1. Process Configuration

On each backend processor, processes were configured so that delays due to data transfer and processor synchronization would be minimized. In particular, data transfers (processor-to-disk, or processor-to-processor) had to be overlapped with computation whenever possible. To avoid copying data within a processor, we adopted a structure consisting of a single *JASMIN* load module per processor. Multiple processes within each load module share access to a single buffer pool, but communicate and synchronize using messages.

Data transfers are organized as *streams* of record blocks. Each stream consists of a series of fixed size blocks containing a number of records. There are three types of streams, each with its own block size. *Input streams* carry unsorted record blocks from the root to the leaves. *Disk streams* carry sorted temporary files between a leaf and its disk. *Merge streams* carry sorted record blocks between processors in the merge tree.

During the sort phase, the records to be sorted are sent as input streams from the root to the leaf processors. In order to overlap communication latency, leaf processors interact with multiple communications processes in the root. In the leaf processors, an *input spooler* process reads the *input stream* from the the root, and builds *sort buffers*, which consist of

data records and an array of pointers to these records (Figure 3).

Once a sort buffer is built, it is passed to the main *sort/merge process*. This process performs a Quicksort on the pointer array, and passes the buffer to the *file spooler* process. The file spooler moves records for the first time from the input blocks, in which they were received, to disk blocks which are then written to disk. Each sorted buffer forms one *temporary file* on the disk. The use of two sort buffers per processor ensures a high degree of overlap between input, sorting and writing to disk.

During the merge phase, the sort buffers are unused and so are returned to free memory for use as disk or merge block buffers. The sort/merge process is central to the merge phase on both leaf and non-leaf processors (Figure 4). In the leaves, it merges sorted disk streams from the temporary files, one stream per file. In a non-leaf node, merge streams from the node's children in the tree are merged. In the former case, disk streams are read through the file spooler, which performs double buffering (disk read-ahead). In the latter case, a special *network spooler* process synchronizes data transfer from child processes

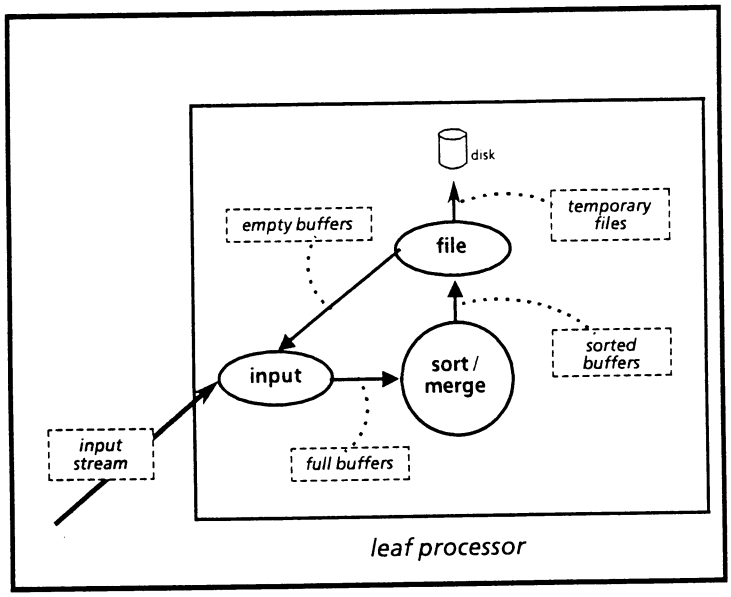

Figure 3: Sort Phase Leaf Process Structure

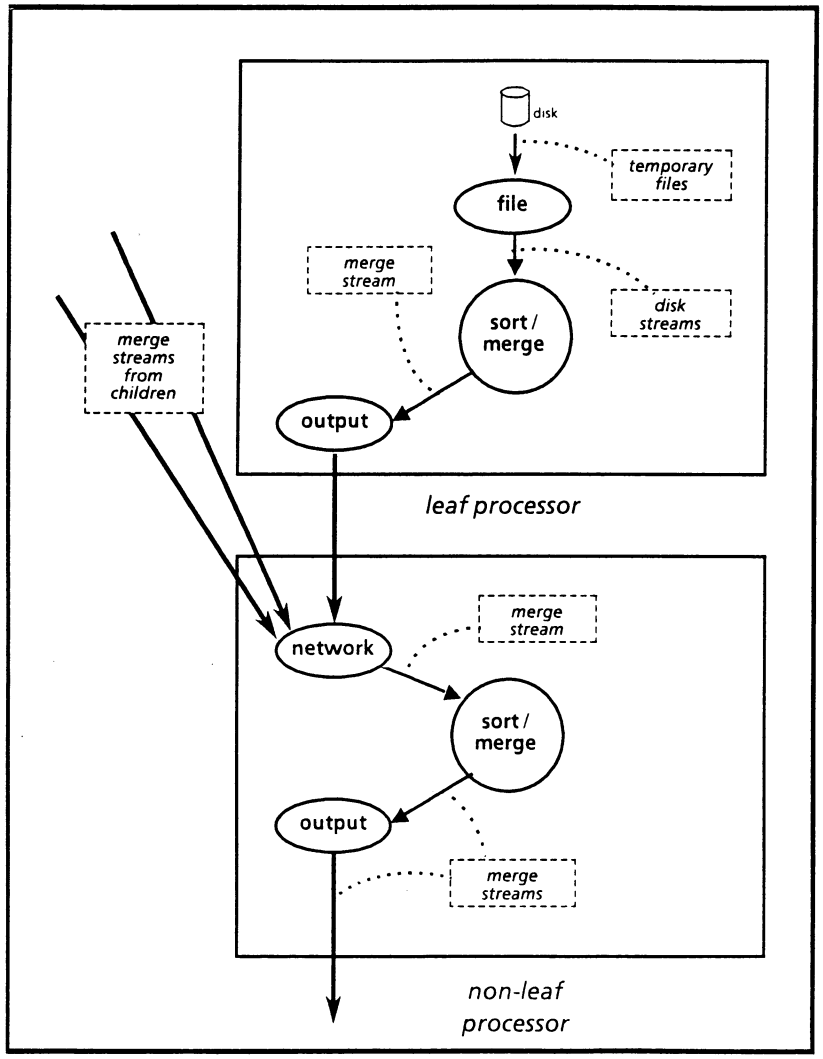

Figure 4: Merge Phase Process Structure

and implements double buffering of merge streams. On all non-root processors, the output spooler process implements double buffering of merge streams being sent to the parent in the merge tree.

3.2.2. Memory Configuration

Having only one megabyte of main memory attached to each processor restricts the maximum size of data buffers. After loading the operating system, utility tasks, the text segment for the sorting program, a stack segment for each sorting process, and

miscellaneous sort program variables, about 576K remains for buffers. This led us to a configuration of two 256K sort buffers, resulting in temporary files somewhat smaller than that (the buffer must hold not only the records, but also an array of pointers to them). In contrast, a mainframe sort routine will typically use a 1 megabyte sort buffer.

During the merge phase, each leaf processor must read a potentially large number of disk streams, and merge them to produce a single merge stream. Each stream is double buffered so the total memory used when merging n files will be

$$M = 2nB_{disk} + 2B_{merge}$$

where the B_x are block sizes. Now, the memory used as sort buffers during the sort phase can be reused as disk and merge blocks during the merge phase. Thus, all 576K are available for allocation. If we are to sort a 100M file using two leaf nodes, then each leaf must handle 50M, or 200 temporary files. Rearranging our equation, we see that

$$B_{disk} = (M - 2B_{merge}) / 2n \leq (576K / 400) \approx 1.4K$$

So, choosing $B_{disk} = 1K$, we get $B_{merge} \leq 88K$.

The constraint on the size of a merge block does not arise at the leaves, but at non-leaf nodes. If a non-leaf node merges m merge streams to form a single sorted merge stream, then it requires $2m + 2$ merge blocks for buffering. Since non-leaf nodes use no memory for disk blocks, if we want to handle values of m up to 8, we get

$$B_{merge} = M / (2m + 2) \leq 576K / 18 \approx 32K$$

Thus, the demands of the problem we posed – sorting a 100Mbyte file in configurations ranging from two to eight leaf processors using varying tree structures – leads to a rather narrow choice of buffer sizes. We used the closest powers of two:

$$(B_{disk}, B_{merge}) = (1K, 32K)$$

3.2.3. Disk File Layout

The layout of files on disk is central to minimizing access time. We experimented with two approaches: contiguous files and interleaved files. In contiguous file layout, the logical disk blocks of a file lie in consecutive physical disk addresses. This means they will be physically contiguous unless they cross a cylinder boundary. In interleaved file layout, the first logical block of all files lie in one consecutive set of disk addresses, followed by the second block of all files, etc. (Figure 5).

Contiguous file layout minimizes seeking when reading or writing an entire sequential file. It maximizes seeking when multiple sequential files are to be accessed concurrently. Interleaved layout achieves the opposite: maximal seeking when accessing a single file sequentially; minimal seeking when accessing many files concurrently. Intermediate schemes which interleave files in segments larger than one logical block are possible, but we did not implement any. In our sort algorithm, temporary files are written one at a time during the sort phase, and then all are read concurrently during the merge phase. Thus, contiguous layout performs best during the sort phase, while interleaved layout wins in the merge phase.

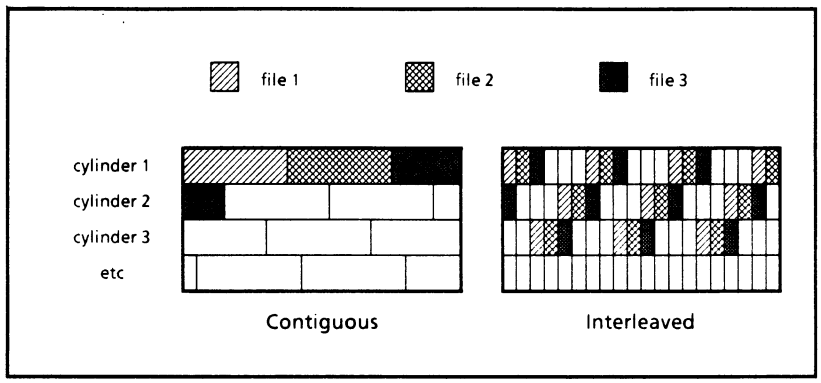

Figure 5: Contiguous vs Interleaved File Layout

4. Measurements

In this section, we present and analyze our measurements for a range of backend configurations and file sizes. In order to establish a performance baseline, we start with our uniprocessor sort (*JS1*) on *JASMIN*, and compare it to the Linderman Sort on a fast UNIX machine (Section 4.2). We then analyze in detail the performance of the parallel *Backend Sort*, for different multiprocessor configurations (Section 4.3). Finally, we estimate elapsed times for the *Distributed Sort* (Section 4.4).

4.1. Parameters Varied

In our experiments, we varied the following parameters:

(1) Number of processors in the backend:

We configured the backend as a single processor, as a two-level tree with 3 and 5 processors (2 and 4 leaves), and as a three-level binary tree with 7 processors (4 leaves, 2 internal nodes). In subsequent tables, these configurations are labeled as *JS1*, *JS3*, *JS5*, and *JS7*, respectively. In all the tree configurations, each leaf processor had a local disk.

(2) Structure and size of the data file:

We varied the size of our data file from 1 to 100 megabytes. The data was randomly generated as a stream of 100 byte records, with a 10-byte ascii-numeric sort key. The record format included a header consisting of a 2-byte record-length field and a 2-byte key-length field, which were interpreted by the sort program. Thus, our implementation can also sort variable length records. In addition to the data records, we used two types of control records, *end of block* and *end of file*, in order to form streams of records. These control records are four bytes long.

We also varied the memory configuration and the file layout scheme. However, all the measurements presented below correspond to experiments with double buffering and interleaved file layout, since these appeared to be more efficient.

4.2. Uniprocessor Efficiency

In order to establish a baseline performance measure for our implementation, we compared a single-processor configuration to the UNIX sort [LIN84] on a faster machine. Figure 6 shows the times recorded for sorting files of 1 to 50 megabytes on a *JASMIN* 68000-based microcomputer and on a CCI Power-6 computer running UNIX 4.2 BSD. In both cases the disk was a Fujitsu Eagle. The *JASMIN* figures constitute a baseline that we will use to evaluate the parallel speedup achievable in multiprocessor configurations. We will use the UNIX-CCI numbers as a baseline for comparing the performance of a fast, general-purpose serial machine with that of a backend multiprocessor.

The 68000-based microcomputer is rated at 0.6 MIPS, while the CCI processor is rated at 5 to 6 MIPS. Considering the difference in processor capabilities, the comparison greatly favors the *JASMIN* sort. The UNIX-CCI sort is 1.6 times faster for small files, but only 1.2 times faster for large files. From Figure 6, we observe that, up to 35M, the rate of the *JASMIN* sort is almost constant at 18 Kbyte/sec. On the other hand, the rate of the

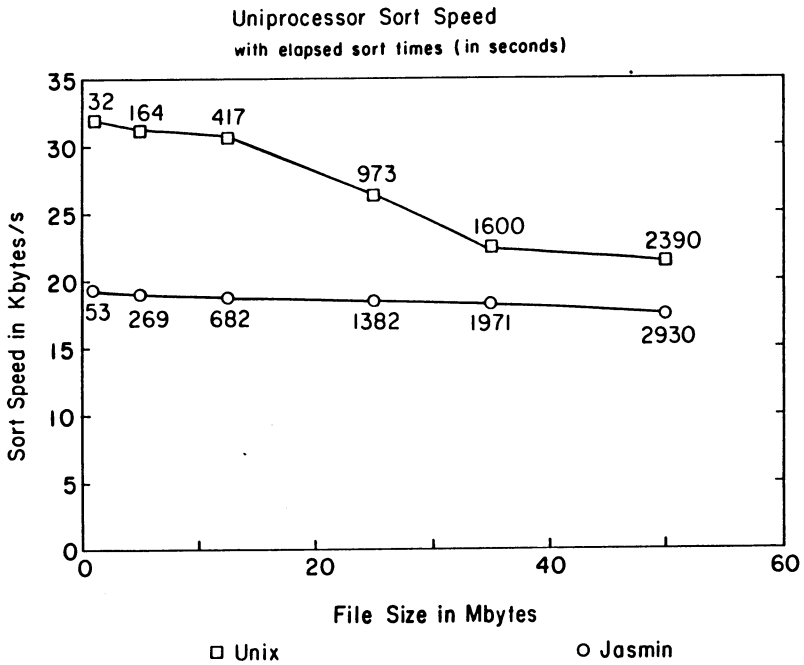

Figure 6: Serial Sort Times in Seconds on *JASMIN* and UNIX

CCI-UNIX sort degrades rapidly for file sizes above 20M. This degradation is at least partly due to the limit of 20 on the number of open files in a UNIX process, which necessitates multiple N-way merge passes in the merge phase. The comparison illustrates the limits imposed by a general-purpose operating system (UNIX) compared to *JASMIN*. The *JASMIN* kernel essentially gives an application the full power of the underlying computer. In this application, we observed a processor utilization rate of 90%.

4.3. Backend Sort

In Table 1, we show the total execution time of the Backend Sort for 4 processor configurations, and for file sizes of 5, 12.5, 25, 50 and 100 megabytes. The times shown correspond to the root's elapsed time, which was only slightly higher than the leaves' or internal nodes'. Figure 7 compares the sorting speed achieved in these experiments to the UNIX-CCI sort.

The most striking observation from this table is how fast the 100M sort is, for both JS3 and JS5. The backend multiprocessor sorts 100M in 1 hour with 3 processors, and in 52 minutes with 5 processors. This performance is comparable to that of highly-tuned commercial sort packages such as the IBM SYNC-SORT on a high-performance mainframe (e.g. an IBM 3081, rated at approximately 7 MIPS). It is thus clear that our parallel Backend Sort provides a very cost-effective alternative to main-frame, serial sorting of large files.

File Size	JS1	JS3	JS5	JS7
5M	269	178	178	173
12.5M	682	439	419	427
25M	1384	884	836	869
50M	2930	1782	1584	1682
100M	*	3625	3131	3305

Table 1: Total Sort Times In Seconds
1, 3, 5 and 7 Processor Configurations

* This number is unavailable due to processor memory limitations.

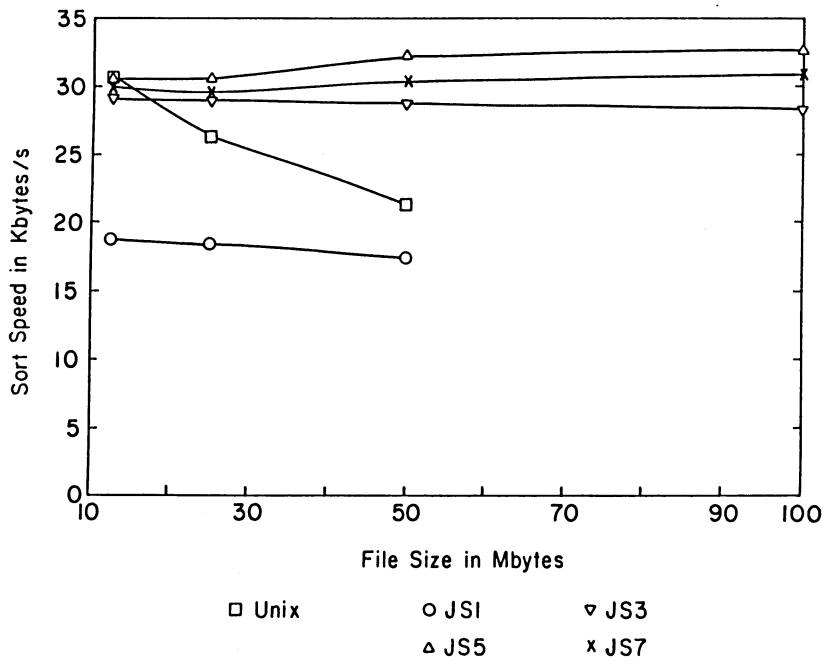

Figure 7: Parallel and Uniprocessor Sort Speeds

Further analysis of the data in Table 1 leads us to the following observations on parallel speedup:

(1) The 3 processor configuration is faster than the 1 processor configuration by a factor of 1.5 for small files to 1.6 for a 50M file.

(2) Using two additional leaves (i.e. the 5 processor configuration) further improved the sort time by up to 16%.

(3) Using a two-level merge tree with 7 processors does not improve performance. In fact JS7 shows slightly higher overall times than JS5.

The added processing power and I/O bandwidth in JS3, with 3 processors and 2 disks, proved very effective in reducing up the elapsed time of the sort. However, as more processors and disks were added, the additional performance enhancement was limited. These observations suggest that the parallel sort is limited by network data transfer rather than by computation or disk activity. We tested this hypothesis by instrumenting the sort program to record idle and elapsed times for both the sort and merge phases. Table 2 shows these measurements for a 25 megabyte sort with 3 processors, and a 50 megabyte sort with 5 processors.

		JS3-25M		JS5-50M	
		leaf	root	leaf	root
Sort Phase	elapsed time	638	639	1004	1005
	disk busy	2	0	0	0
	idle time	320	162	680	26
Merge Phase	elapsed time	937	938	1871	1873
	disk busy	53	-	55	-
	idle time	598	629	1534	1274
Total	elapsed time	1575	1577	2875	2878
(Both Phases)	disk busy	55	-	55	-
	idle time	918	791	2217	1300

Table 2: Sort, Merge and Idle Time In Seconds
3 and 5 Processor Backend Sort

The times in Table 2 are for the root, and one of the leaves. (Since all leaves had similar elapsed and idle times, we only present our measurements for one.) Discrepancies between the elapsed times shown in Table 1 and Table 2 for the same configurations are due to distortions introduced by measuring idle time. The additional data in Table 2 supports our hypothesis in the following ways:

(1) The sort phase of the 50M sort lasts almost twice as long as the sort phase of the 25M sort, although the leaves do exactly the same work in both. The additional time shows up as idle time at the leaves, while they wait for data from the root. In both configurations, the root shows little idle time during the sort phase. I/O delays (shown as *Disk Busy* time) are almost null during the sort phase.

(2) The merge phase of the 50M sort lasts almost exactly twice as long as that of the 25M sort, and substantial idle time accumulates at both the leaves and the root. Unlike the sort phase, where the root processor is saturated, both root and leaves appear to be waiting on the network. Disk waiting during the merge phase accounted for less than 10% of the idle time, in both configurations. Furthermore, it did not decrease with the use of two additional disks (JS5 versus JS3).

(3) The non-idle time at the leaves during the sort and merge phases, is approximately the same on both configurations: sort phase 320 seconds, merge phase 340 seconds. The non-idle time at the root varied linearly with the amount of data in both phases. The sort phase might be speeded up by starting with a distributed file (see Section 4.4), or increasing the power of the root processor. However, it is clear that a higher network transfer rate is necessary in order to speed up the merge phase.

We conclude that for the Backend Sort, the limitation of a low network transfer rate means that the use of more than 4 processors as leaves can speed the sort phase only marginally, and the extra processor power is not used efficiently. Similarly, use of additional pro-

cessors as interior merge nodes in the 7 processor configuration only reduces non-idle time at the root, which does not improve the overall merge speed. A network which delivered data at the faster rate would be able to increase the utilization of the leaves and root during the merge phase, and so could effectively utilize more processors.

4.4. Distributed Sort

The above discussion of Table 2 indicates that the distribution of the file from the root to the leaf processors created a bottleneck. In a *Distributed Sort*, where the file is initially distributed across the local disks at the leaves, the sort phase would be much faster. In fact, since, during this phase, the computation is fully partitioned and every leaf processor has its own disk, the sort phase time at the leaves is simply the uniprocessor sort phase time for one partition. Thus, a good estimate of the sort phase time in the Distributed case is the sort phase time of one partition in the JS1 configuration. Merge phase time in the Distributed Sort is identical to that in the Backend case. By combining these estimates of the sort and merge phase times, we obtained the total times for the Distributed Sort. The results obtained are shown in Table 3.

		Sort	Merge	Total
JS3	10M	147	158	305
	50M	738	828	1566
JS5	10M	74	157	231
	50M	370	696	1066

Table 3: Distributed Sort Estimate
Sort, Merge and Total Times in Seconds

While in the Backend Sort, the bottleneck at the root made the use of more than 3 processors inefficient, we now observe a substantial speedup with 5 processors. For a file of 50M, the elapsed time is 1566 seconds for a distributed 3 processor sort and 1066 seconds for a 5 processor sort. This is a speed-up of 1.5 compared to 1.1 for the Backend Sort.

5. Tuning and Scaling

In the experiments reported above, we adopted the buffer configuration and file layout that appeared to be most efficient. In this section, we will describe our experience with tuning these parameters. Based on the results of our analysis and on additional experiments, we propose a model for *tuning* and *scaling* a sort utility that could exploit higher processor utilization, more parallelism and higher network transfer rates.

5.1. Choosing Parameter Values

We experimented with various ways to partition the 576K of memory that were available for buffer configuration in each processor. The parameter to which the sort time was most sensitive was merge block size. Larger merge blocks decreased the overhead during the merge phase, and reduced the number of interprocessor synchronization points. With double buffering, the maximum size of a merge block that we could choose was 32K (see Section 3.2). To a lesser extent, increasing the input block size also increased sort speed and we settled on an input block size of 8K. Larger input blocks caused wasteful fragmentation of sort buffers.

The large amount of idle time in the leaves during the merge phase suggests that double buffering of disk blocks may not be advantageous. It may be better to consolidate the two buffers into one large buffer to reduce the number of disk accesses. Additional enhancement could also come from improving the scheduling of network data transfers. In the current implementation, the network spooler in a parent node transfers merge blocks from the node's children in the order in which they are requested. However, at the same time, double buffering may cause network traffic that blocks the transfer of data which is needed for merging to proceed. Children are serviced in approximately round-robin order, giving no priority to a transfer which is holding up the merge.

We compared the performance of our sort with the two alternative file layouts: *contiguous* and *interleaved*. Using the interleaved scheme provided an overall improvement of

5% in elapsed time.

5.2. Scaling Processor Speed

Processor speed affects the performance of the sort in two ways. One is by determining the time for comparing and moving records in memory. The other is by determining the time needed to handle the communication tasks. Furthermore, because of the tree structure of the merge, different CPU loads are placed on different nodes in the tree. We designed an experiment to determine which components of the sort utilized the most processing power in merging and eventually, constituted a bottleneck.

We varied the merge speed by artificially increasing the time of the comparison routine. This was achieved by inserting a do-nothing loop before each Compare statement. In an initial experiment, we quantified the effect of the comparison delay on the merge speed of a leaf processor. We configured a 2-processor sort, with one leaf and the root, and varied the comparison delay from 0 to 300 loop iterations. We observed that the non-idle merge time at the leaf increased linearly and varying the delay from 0 to 300 decreased the merge speed from 100% to 20% of the undelayed speed. Once we established this rate, we proceeded to measure the effect of varying the merge speed at the leaves, in the 3-processor (JS3) and the 5-processor (JS5) configurations. In both cases, we kept the amount of data merged by a single leaf equal to 2.5M. The results of this experiment are shown in Table 4.

From the numbers in the Idle column for JS3 and JS5, we observe that the idle time at the leaves decreases steadily, up to a delay of 100 for JS3 and 300 for JS5, while the root merge time remains constant. This confirms our expectation that, without any performance degradation, the leaf processor could be slower. This experiment shows that the merge time would not be affected by using leaf processors that are 50% as fast in JS3, or only 20% as fast in JS5.

In JS3, we also observe that for delays greater than 100 (thus less than half the merge speed), the leaf idle times remain small while the root idle time starts to increase. This

		JS3		JS5	
		Merge	Idle	Merge	Idle
d=0	leaf	178	116	377	313
	root	179	120	377	246
d=10	leaf	180	110	364	290
	root	181	123	364	242
d=100	leaf	184	40	379	240
	root	178	107	379	249
d=200	leaf	243	36	368	145
	root	231	162	373	247
d=300	leaf	319	40	398	115
	root	306	237	378	243
d=400	leaf	375	6	452	105
	root	376	310	412	272

Table 4: Merge Time in Seconds
No Root Delay, Leaf Delay From 0 to 400

indicates that the root begins to wait, not only for the network but also for the leaves.

We performed another experiment to determine the effect of decreasing the root merge speed. In the JS3 configuration, we held the leaf merge delay at 200 and varied the root delay from 0 to 500. The corresponding measurements of one leaf and the root merge and idle times are shown in Table 5.

		JS3	
		Merge	Idle
	leaf	230	6
d=0	root	231	162
	leaf	230	5
d=100	root	232	122
	leaf	228	6
d=200	root	230	73
	leaf	228	6
d=300	root	230	31
	leaf	238	22
d=400	root	240	7
	leaf	279	61
d=500	root	284	6

Table 5: Merge Times in Seconds
Leaf Delay Fixed at 200, Root Delay From 0 to 500

Interestingly, the delay at the root had no effect until it increased to approximately 400, which is twice the delay at the leaf. However, the root idle time steadily decreased, indicating a higher processor utilization. This constitutes further evidence that the bottleneck at the root is caused by communication overhead, rather by the merge operation. The processor load due to record comparisons is actually lower on the root than on the leaves.

5.3. Scaling Network Configuration

In our implementation, the efficient exploitation of parallelism was limited by an operating system maximum of eight processors, and by high latency in data transfer between processors. It is interesting to project how much faster our sort could be, were these constraints lifted. We propose a simplified model to determine, given a file size and a target sort time, how many processors and what tree configuration would be required.

From our measurements of idle time (Table 2, above), we observe that the absolute rate of merging is 27 Kbyte/sec, while 80K bytes are merged for every second of non-idle time. Thus, even if network latency were reduced to zero, the merge rate would be limited to 80 Kbyte/sec, about three times its present rate. However our experiments with scaling processor speed demonstrated that most of the CPU time at the root is spent in network overhead, rather than on merging. Faster merge rates would be possible if the burden on processors due to network communication were reduced. In that case, additional parallelism could be exploited, because increasing the numbers of leaves would decrease the sort time, rather than merely increasing idle time at each leaf.

If we assume that the merge phase is not bound by network latency, then we may derive an "efficient" processor topology for a given target sort goal. By "efficient", we mean that the configuration would guarantee that the times for the local sort of temporary files, the serial merge at the leaves, and the concurrent merge at the internal nodes and the root, all remain a linear function of the size of the data processed by each node.

When we assume that no component of our configuration exhibits non-linear behavior, we can characterize each component by two parameters: its linear rate, and the constraints within which it functions at that rate before non-linear behavior appears. From our experiments we derived the following parameters:

$R_{lsort} = 20Kbyte/sec$, linear rate of leaves during the sort phase
$R_{lmerge} = 35Kbyte/sec$, linear rate of leaves during the merge phase
$N_{lmerge} = 10Mbyte$, maximum data a leaf can merge at that rate
$R_{merge} = 80Kbyte/sec$, linear rate of non-leaves during the merge phase
$D_{merge} = 16$, maximum degree of a merge at that rate

We used this model to compute how many processors and which tree topology would be required to achieve higher sort rates. A straightforward computation leads to the tree configurations shown in Table 6, for 50Kbyte/sec and 75Kbyte/sec target sort rates, and file sizes equal to 100 and 500 megabytes.

In configurations for the lower sort target rate (50 Kbyte/sec), the limiting factor is the need to keep the leaves linear in the merge phase. Hence, when the file size increases from 50 to 100 megabytes, a larger merge tree (50 leaves and 2 levels) is required. For the 75 Kbyte/sec sort target rate, the speed of the inherently linear sort phase is the limiting factor. Consequently, the optimal configuration is less dependent on the file size. Our computation leads to the same tree configuration (60 leaves, 4 processors at level 1) for both file sizes.

Target		Processor Config.		
Sort rate	File size	level 0	level 1	level 2
50KB/s	100M	10	1	-
	500M	50	4	1
75KB/s	100M	60	4	1
	500M	60	4	1

Table 6: Efficient Processor Configurations (Scaling)

6. Conclusions

We have described our experience with developing and evaluating a parallel sort utility. Our goal was to explore the feasibility of fast file sorting algorithms under limited parallelism, limited inter-processor communication bandwidth and the constraints of conventional I/O devices. Our testbed was a multiprocessor that can be configured with up to 12 microprocessors, communicating through a fast packet-switched bus and running a distributed operating system kernel. Multiple disk drives, each attached to one microprocessor, provided high I/O bandwidth.

We implemented a parallel external merge-sort algorithm, and measured its performance for a range of file sizes and processor configurations. After detailed analysis of our measurements, and scaling of our results, we reach a number of conclusions:

(1) Using current off-the-shelf technology, 3 and 5 processor configurations provide a very cost-effective solution to sorting a large file. The 3 processor configuration sorts a 100 megabyte file in 3625 seconds (1 hour), which compares well with commercial sort packages available on high-performance mainframes. The 5 processor sort is 16% faster for large files. This level of performance, at a low hardware cost, results from a design which makes effective use of a streamlined, distributed operating system to exploit a limited amount of parallelism in computation and I/O.

(2) Latency in network data transfer prevents the effective use of higher levels of parallelism in the backend sort model, where the data file is not distributed. Interestingly, the 10 Mbyte/sec physical bandwidth of our network was not the limiting factor: Only 300 Kbyte/sec of that is available to any one processor, and operating system overhead reduces that to approximately 100 Kbyte/sec for the actual process-to-process data transfer rate. At this rate, network latency allows even a microprocessor (rated at 0.6 MIPS) to sort and merge data as quickly as it can be moved between machines. Adding processors to the backend configuration simply increased idle time.

(3) We observed that distributed storage of files provides a substantial benefit in sorting. For instance, with the 5-processor configuration, a 50-megabyte distributed file could be sorted in 17 minutes, compared to 26 minutes needed for a non-distributed file. Our distributed sort model makes efficient use of additional leaf processors to accelerate the sort phase, which the backend model is unable to do.

(4) Additional tuning of the sort program, and scaling to higher processor and network capabilities would result in substantial improvement of the performance. We have presented experiments and a simple analytical model of this enhancement.

This study demonstrates that, with current microprocessor and communication technology, parallel sorting of large files is a viable, cost-effective alternative to highly tuned sort utilities on mainframes. Our analysis indicates that its efficiency can be expected to increase dramatically due to improvements in local area network technology in the next few years.

7. Acknowledgements

We would like to acknowledge the early work of Bill Burnette and Susan Fontaine at AT&T-Bell Laboratories in developing the JASMIN operating system (then called MAX). U. Premkumar was responsible for developing the network interface which distributes JASMIN across processors. His work with the JASMIN and S/Net software and hardware made this project possible. Hikyu Lee also made significant contributions to the JASMIN S/Net implementation. We would also like to thank Ming-Yee Lai for graciously allowing us to disrupt his use of the S/Net, and for helping us with an early draft of this paper.

We are grateful to two managers who have been especially supportive of our work. Dan Fishman saw JASMIN through its early days at Bell Laboratories, and, although now at HP Labs, he continues to encourage our efforts. Mike Lesk, of Bell Communications Research, has generously supported the collaboration with Cornell University which has made this research possible. Finally, we thank Jim Gray from TANDEM Computers for his helpful comments and suggestions.

References

[ANO85] Anon et al., A Measure of Transaction Processing Power, Tandem TR 85-2, February, 1985.

[AHU83] Ahuja, S.R., S/NET: A High Speed Interconnect for Multiple Computers, IEEE J on Selected Areas in Communications, SAC-1, 5, November, 1983.

[BAS77] Baskett, FH, Howard, JH, Montague, JT, Task Communication in DEMOS. Proceedings of the 6th ACM Symposium on Operating System Principles, November, 1977, pp. 23-31.

[BIT82] Bitton, D, Design, Analysis and Implementation of Parallel External Sorting Algorithms, Ph.D. Thesis, University of Wisconsin-Madison, TR 464, January, 1982.

[BIT83] Bitton, D, and DeWitt D.J., Duplicate Record Elimination in Large Data Files, ACM Trans. on Database Systems, June, 1983, pp.255-265.

[EVE74] Even S., Parallelism in Tape Sorting, Comm. ACM, Vol.17, No. 4, April, 1974, pp.202-204.

[FIS84] Fishman, DH, Lai, MY, Wilkinson, WK, An Overview of the *JASMIN* Database Machine. Proceedings of the ACM SIGMOD Conference, Boston, MA, June, 1984, pp.234-239.

[KIT84] Kitsuregawa, M., Tanaka, H., Moto-oka, T., "Architecture and Performance of Relational Algebra Machine GRACE," Proceedings of the 1984 International Conference on Parallel Processing, 1984.

[KAM85] Kamiya, S., Matsuda, S., Iwata, K., Shibayama, S., Sakai, H., Murakami, K, "A Hardware Pipeline Algorithm for Relational Database Operation," The 12th International Symposium on Computer Architecture, June, 1985.

[KWA86] Kwan S.C., External Sorting: I/O Analysis and Parallel Processing Techniques, Ph.D. Thesis, University of Washington, TR 86-01-01.

[KNU73] Knuth D., *The Art of Computer Programming III, Searching and Sorting*, Addison-Wesley 1973.

[LEE84] Lee, H, Premkumar, U, The Architecture and Implementation of Distributed *JASMIN* Kernel. Bell Communications Research Technical Memo, TM-ARH-000324, October, 1984, Morristown, N.J.

[LIN84] Linderman, JL, Theory and Practice in the Construction of a Working Sort Routine. AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8, Part 2, pp. 1827-1845, October, 1984.

[SEL79] Selinger P.G. et al., Access Path Selection in a Relational Database Management System, Proceedings of SIGMOD, pp. 23-34, May, 1979.

[SZH83] Schweppe, H, Zeidler, H.Ch., Hell, W., Leilich, H.-O., Stiege, G., Teich, W., "RDBM - A Dedicated Multiprocessor System for Database Management," in Advanced Database Machine Architecture, (D.K. Hsiao, ed.), Prentice-Hall, 1983.

[TAN83] Tanaka, Y., "A Data-Stream Database Machine with Large Capacity," in Advanced Database Machine Architecture, (D.K. Hsiao, ed.), Prentice-Hall, 1983.