Tandem TR 86.3

# FASTSORT: AN EXTERNAL SORT USING PARALLEL PROCESSING

Alex Tsukerman
Jim Gray
Michael Stewart
Susan Uren
Bonnie Vaughan
Spring 1986

## ABSTRACT

FastSort is an external sort that uses parallel processing, large main memories and parallel disc accesses to obtain high performance. FastSort can sort a file as quickly as it can read the input and it can produce the result as quickly as it can write the target file -- that is it sorts in linear time rather than the traditional *Nlog(N)* time of serial sort systems. This paper describes the design and implementation of FastSort. It presents performance measurements of FastSort on various Tandem Nonstop processors, with particular emphasis on the speedup obtained by using parallelism to sort large files.

TABLE OF CONTENTS

## INTRODUCTION

Loosely coupled multi-processors can give linear growth in transaction throughput for on-line transaction processing -- by doubling the number of processors, discs and communications lines, Tandem systems can process twice as many transactions per second [Chmeil and Houy]

This linear growth of throughput as resources are added does not apply to batch transaction processing. A batch Cobol program will not run much faster as more processors are added because the program executes on a single processor. Tandem is exploring ways to apply parallelism to both batch and online transaction processing.

FastSort is an example of the effort to apply parallelism to batch processing. FastSort breaks a large sort job into several smaller ones that are done in parallel by subsort processes. These subsorts can use multiple processors, multiple channels, and multiple discs. The result is a high performance sorting system. When sorting a file of one million 100-byte records, FastSort is competitive with the industry leader in single-processor sorting and can outperform other sort programs by using the Tandem architecture for parallel sorting. It is also four to eight times faster than Tandem's standard SortMerge product. With larger files, FastSort's advantages are even more dramatic.

FastSort speed is proportional to the size of the input file, *N*, rather than the traditional *Nlog(N)* speed of conventional sorting products. This linearity is achieved by distributing the processor and disc load among several processes if the load exceeds the capacity of a single processor or disc. FastSort can sort records as fast as it can read them. Once it has read the file, it can produce the output as fast as it can write the output file.

This article presents the history and design of FastSort. It also explains how to estimate FastSort execution times on NonStop II, NonStop TXP, and Nonstop VLX processors.

## EVOLUTION OF FastSort

Many programs and products use SortMerge on Tandem systems. User programs and batch-oriented job control files invoke it explicitly. The File Utility Program (FUP) invokes sort to create key-sequenced files and indices for structured files. The ENFORM query processor uses sort to evaluate queries.

SortMerge is a mature and functional product. It sorts records based on multiple-typed key fields, allows user-defined collating sequence, eliminates duplicates, projects out fields, merges sorted files, and produces statistics on the sort run. It accepts input from devices, processes, and all file types. The sorted output can be sent to any destination, although SortMerge does not directly produce key- sequenced or Edit files.

Performance is SortMerge's only serious flaw. Originally written for Tandem's 16-bit NonStop 1+ system, SortMerge does not take advantage of the 32-bit addressing or the parallel architecture of modern Tandem NonStop systems. SortMerge runs as a single process that defaults to 34 Kbytes of main memory and uses a maximum of 128 Kbytes of memory. Consequently, its performance is not impressive for large batch applications or for loading or reorganizing large files.

Because of SortMerge's poor performance, other SortMerge products were developed in the field for NonStop systems. Roland Ashbaugh created SuperSort [Ashbaugh], and Eric Rosenberg developed and marketed Qsort [Rosenberg]. Both of these sort programs use multiple processors executing in parallel to sort large files -- this is known as a parallel sorting. In addition, Qsort uses the large main memory provided by the NonStop architecture.

FastSort is a compatible evolution of SortMerge -- they share a common manual and any program using SortMerge will work with FastSort. External improvements in FastSort include the ability to build key-sequenced files in addition to all other file types, automatic selection of an efficient configuration which the user can override, and generation of better diagnostic messages in error cases. But speed is the main advantage of FastSort. Internally, FastSort uses improved algorithms, extended memory, double buffering, streamlined comparison logic, streamlined structured file access, bulk I/O, and multiple processors and discs.

As a result, when sorting a file of one million 100-byte records, FastSort is four times faster than SortMerge, and eight times faster if multiple processors are used.

## HOW FastSort WORKS

FastSort takes one or more input files and produces an output file containing the input records ordered by up to 32 key fields.

Sorting is done in passes. During the first pass FastSort reads the input records and adds them to a binary tournament tree arranged much like the winners in a tennis tournament. The maximum record is at the root of the tree, and the winner of each subtree is the root of that subtree (see Figure 1).

Initially the tournament tree is full of "maximum" null records. FastSort adds input records to the leaves of the tree, gradually displacing null records, which are removed at the root of the tree. The tree minimizes the number of times FastSort compares a record with other records. A tree of height 14 can hold 16,385 records ( $2^{14}$+l) and FastSort compares each record with only 14 others in sorting the whole tree. If the records are 100-bytes each, such a tree occupies about 16K*l00 ~ 1.7 Mbytes. Even with great attention to minimizing and streamlining compares, about 75% of FastSort's time is devoted to comparing records. This is because the compare work for a file of N records rises as *Nlog(N).* For *N* beyond 10,000, the *Nlog(N)* term dominates the costs of reading the input and writing the output. For more details, see Knuth's discussion of replacement selection [pp. 251-266, pp. 328-351].

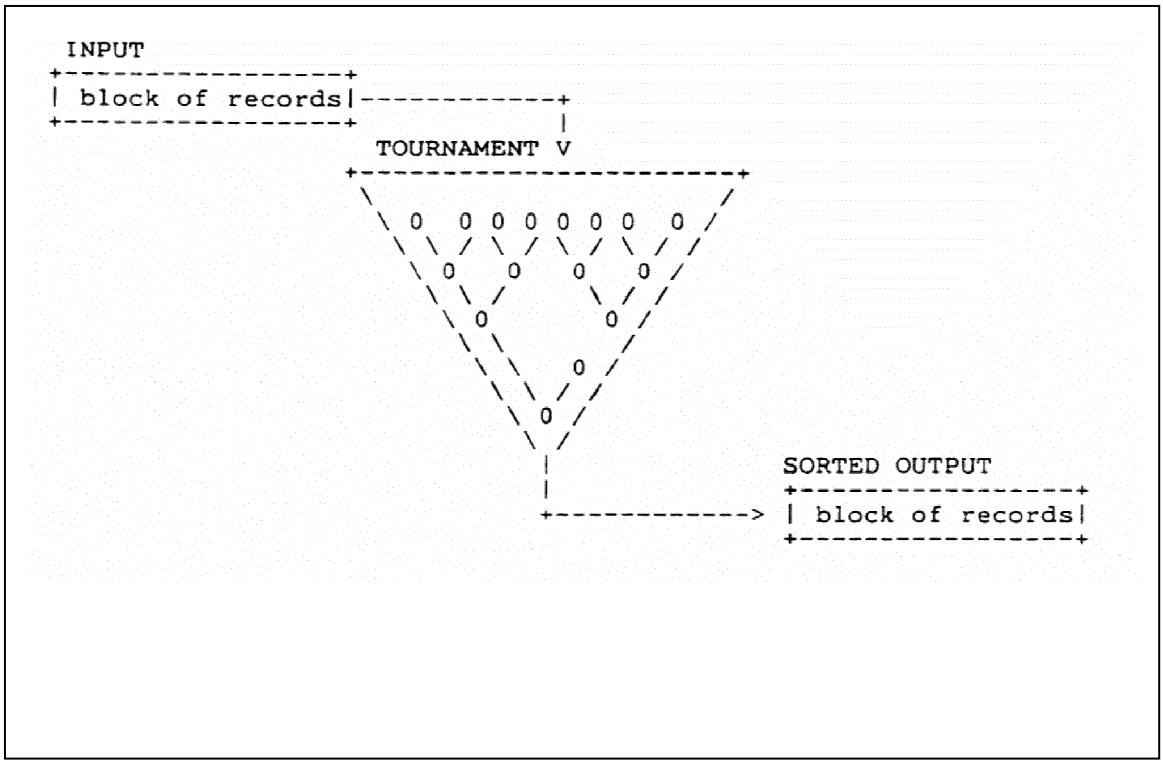


**Figure 1**. The structure of a tournament. The input arrives as double-buffered blocks of records. The sorted output is produced in double-buffered blocks of records. Records move from the leaves (top) of the tournament tree to the root (bottom). The "winner" record is at the root.

## ONE-PASS SORTS

If the input file is less than the size of main memory, then the sort can be done in one pass. As records are read in, they are added to the leaves of the tournament tree. By the time the last record is read, the records are completely sorted in the tree and ready to be written to the output file.

Currently, Tandem Nonstop systems can attach 16 Mbytes of main memory per processor. Such a processor can sort large files entirely in memory. On a full system, Parallel FastSort can apply 16 processors to the problem and sort even larger files in main memory. In this parallel one-pass approach a distributor-collector process starts a sub-sort process in each CPU. The subsorts allocate memory sufficient to hold their part of the job. The distributor then reads the input stream (tape, process, or disc file) and distributes the records in round-robin fashion to the subsorts. When the distributor-collector comes to the end of the input file, it sends an end-of-file to the subsorts. The distributor-collector process now becomes a collector. It reads the output runs from the subsorts, merges (sorts) these runs into a single run, and writes the resulting run to the output stream.

FastSort reads input records in large blocks and writes output records in large blocks to minimize message overhead and disc usage. Block sizes can be up to 30 Kbytes but 16 Kbyte blocks provide most of the benefits. In addition, FastSort uses double buffering; it always gets the next input block while sorting the current block. During output, it always writes the previous block to the output stream while filling the current block in memory. During the first pass, reading the input file and writing the output file is purely sequential access to the disc (almost no seeks), so parallel FastSort is limited by the speed at which discs can be sequentially read or written.

## MULTI-PASS SORTS

One-pass main-memory sorts are the fastest, but not always the cheapest, way to sort. For larger files, or for a less memory-intensive approach, a two-pass or multi-pass algorithm is appropriate. The first pass produces a set of "runs" stored in a scratch file, each run is a sorted sequence of records. Later passes merge these runs into a single sorted sequence which is written to the output file.

If the file is bigger than the tournament, as new records arrive, non-null winners are selected from the root and written out to a scratch file. The tournament is then recomputed to calculate the new root. The result is a "hole" in a leaf of the tournament. A new input record replaces this hole and the cycle repeats. Hence the name replacement-selection sort. This process produces a run of output records.

If an input record bigger than the previous winner arrives, it "breaks" the run -- the new record cannot be added to the end of the current run and still keep the run sorted. In this case, the sorted tournament is written to the scratch file and a new tournament is begun with the new record. The actual implementation is a little fancier than this [Knuth].

If the file arrives sorted, or almost sorted, then only one run is generated. This is the best case. If the file arrives sorted in reverse order, then each run is the size of the tournament. This is the worst case. On average, if the file is random, the average run is twice the size of the tournament [Knuth pp. 40].

If only one run is produced, it is copied to the output file and FastSort is done. If multiple runs are produced, then FastSort merges them to form a single run. These latter passes over the data are collectively called merging. If multiple merge passes are necessary, the intermediate results are kept in the scratch file (Figure 2).

During merging, a tournament is used to combine multiple runs into a single larger run. The following example shows how multiple merge passes may be required. Suppose that the first pass generates fourteen runs, and that the tournament size is ten. Then only ten runs can be merged at once. During first merge, five runs will be merged, which reduces the number of runs to ten -- one run produced from of merging and nine runs from pass one. During a third pass, these ten runs are merged into the output file.

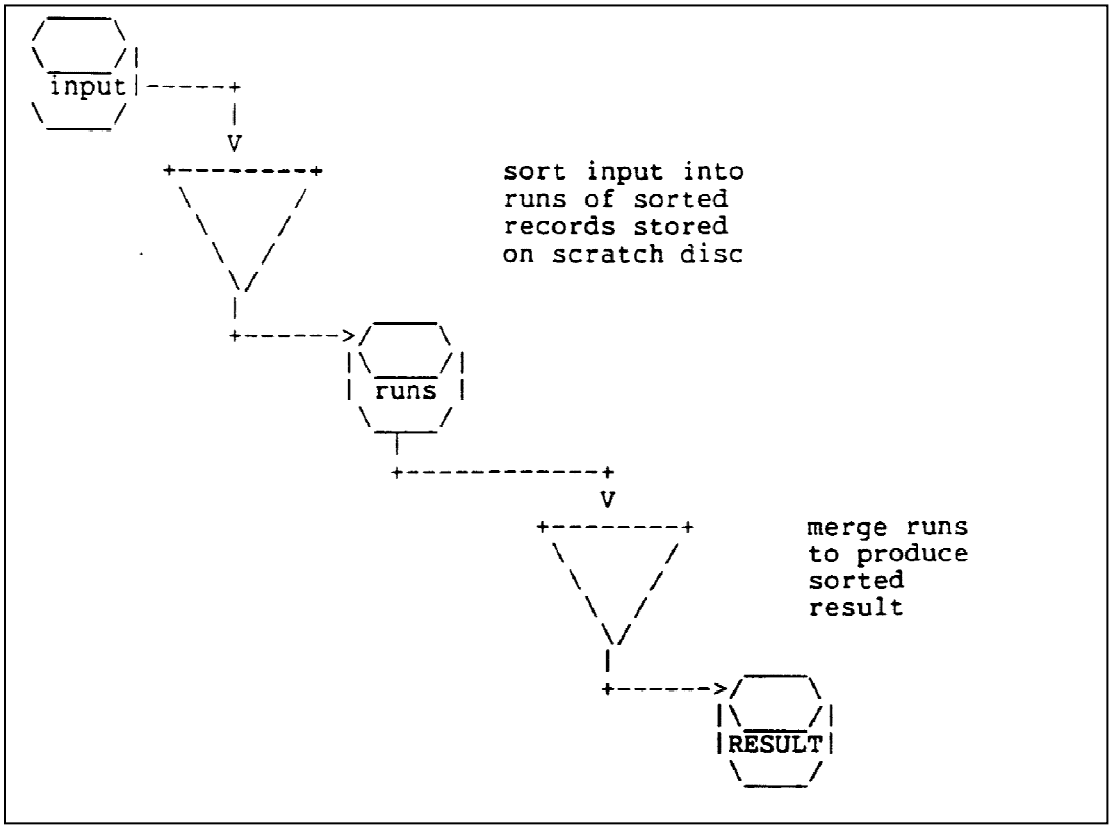


**Figure 2**: A two-pass sort. The first pass over the data produces runs of sorted records stored on a scratch disc. A new run starts whenever an input record is bigger than the tournament winner. During the second pass, the runs are merged together to form a single run, the sorted output. If there are many runs, then more than one merge pass over the data may be required.

Each pass over the data costs disc time and CPU time. So, it is very desirable to have at most one merge pass —- an initial pass to produce runs and a second pass to merge them into the sorted output. A sufficiently large tournament gives a two-pass sort. Surprisingly, not much memory is required for a two—pass sort. The memory requirement rises as the square root of the *(FileSize*BlockSize)/2*.

Assuming a 16 Kbyte blocking factor, the approximate memory requirements for a two-pass sort are shown in Table 1.

| Tournament Required for a Two Pass Sort | File Size (Mbytes) | | | | |
|---|---|---|---|---|---|
| | 1M | 10M | 100M | 1G | 10G |
| | .1M | .3M | 1M | 3M | 10M |

**Table 1.** Tournament size needed for a two-pass sort.

Table 1 shows that a file can get 10,000 times bigger and need only 100 times as much memory. If you balk at 10 Mbytes of memory for a 10 Gbyte sort, observe that 10 Mbytes of main memory costs 50K$ while the disc cost is 1M$ unmirrored and 2M$ mirrored (30Gbyte total for input, scratch and output files). Memory cost is only 5% of the disc cost.

If the user selects the AUTOMATIC option, FastSort tries to allocate enough memory to give a one-pass sort for small files (less than 100Kb), and a two-pass sort otherwise. In general, it uses the equation for Table 1 to estimate the memory size and then adds 30% as a fudge factor. The AUTOMATIC option limits itself to 50% of available main memory while the MINTIME option uses at most 70% of the processor's available main memory. FastSort determines the available memory by asking the operating system memory manager how many pages are not locked down.

The user can explicitly specify how much memory FastSort should use instead of letting FastSort estimate memory requirements.

FastSort also tries to optimize its I/O by using a large block size (l6Kbytes is the default) in reading and writing files. It also double buffers reads and writes so that sorting overlaps I/O requests. Using these features, FastSort reduces I/Os by a factor of four, and eliminates half of the data moves.

By combining all these improvements, serial FastSort runs about four times faster than standard SortMerge in sorting a file of one million 100-byte records -- cutting the time from 115 minutes down to 29 minutes on a NonStop VLX processor. This compares to 21 minutes for SyncSort, the industry leader, on an IBM 4381

To beat SyncSort, FastSort needs to use parallel processing.

## PARALLEL SORTS

The speed of a single processor sort is limited by the speed of one Cpu and the speed of one scratch disc. Parallel sorting uses multiple subsort processes to sort parts of the input file, and a special process, called a distributor-collector, to allocate work to the subsort processes and merge their output runs into the final result. Parallel FastSort operates as follows (see Figure 3):

- The distributor-collector accepts user parameters and starts subsort processes. Every subsort has its own scratch file.
- The distributor-collector reads the input file(s) and distributes records among subsort processes on a round-robin basis. Each subsort sorts its part and produces a sorted stream of records.
- The distributor-collector merges output from the subsorts and writes the output file.

To minimize Cpu and memory contention, each sort process should run in a different CPU. To minimize disc contention, each subsort should have a different scratch disc. Also, the distributor-collector should run in a lightly loaded CPU because it can be CPU bound. In addition, each subsort should run in the CPU containing its primary scratch disc so that inter-processor message traffic does not rise as *Nlog(N).*

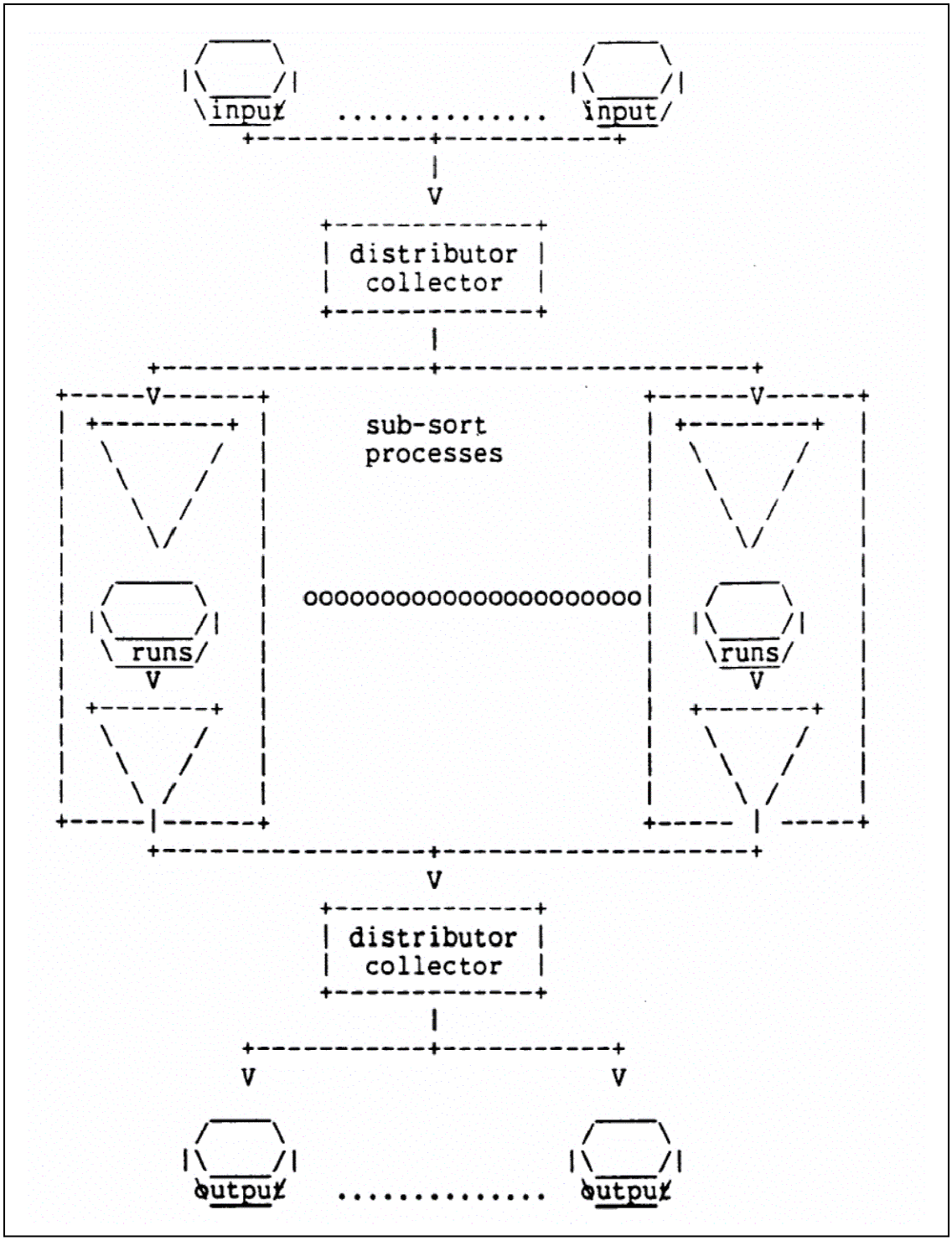


**Figure 3**. The structure of a parallel sort. The distributor-collector process drives two or more subsorts which in turn write their runs to their scratch files. When input is complete, the subsorts merge their runs and return their sorted data to the distributor-collector which merges them into the output file.

FastSort automatically configures parallel sorts to satisfy these configuration rules. The user can configure a three subsort parallel sort by simply naming the scratch discs ($DATA1, $DATA2, $DATA3 in the following example):

```
FROM        infile
TO          outfile
SUBSORT     $datal
SUBSORT     $data2
SUBSORT      $data3
RUN   ,     AUTOMATIC
```

Of course, the user can override FastSort's decisions by specifying the CPU, priority, memory size, block size, and other attributes of the subsort processes. The user can also prohibit use of certain CPUs or restrict use to certain CPUS

When properly configured, parallel sorting is faster than serial sorting because

- The first pass is CPU bound for tournaments containing more than 10,000 records. Parallel sorting spreads this load among multiple CPUs so that the first pass remains bound by the speed at which the distributor-collector can read the input file.
- The second (merge) pass is disc-seek bound while merging the runs. By applying multiple discs to the merge pass, merging can run at the speed of the distributor-collector writing the output file.

In consequence, once started, parallel sort runs as fast as the distributor-collector can read and write the input and output files.

The fact that sort does *Nlog(N)* work to sort a file of size *N* is completely hidden in the subsorts. Hence, parallel FastSort is a linear time sorting algorithm, running at the rate of about 30kb/sec on a NonStop II processor, 80Kb/sec on a NonStop TXP processor, and 110Kb/sec on a NonStop VLX processor.

As a rule of thumb, files less than 1 Mbyte should be sorted in one pass in memory. For such files, the parallel sort setup time dominates any savings in speed, so parallel sort is slower than serial sort for files under .5Mbyte. But for files larger than 1 Mbyte, parallel sort begins to pay off. Table 2 shows the speedups from parallel sorting on a NonStop TXP processor.

| File Size (bytes) | TXP sort time (seconds) | | | SpeedUp |
|---|---|---|---|---|
| | Serial Time | Parallel Time | Parallel cpus | |
| **.1M** | 5 E0 | 8 E0 | 2 | 0.6 |
| **1M** | 2.7 E1 | 2.8 E1 | 2 | 1.0 |
| **10M** | 2.1 E2 | 1.3 E2 | 2 | 1.6 |
| **100M** | 2.5 E3 | 1.3 E3 | 3 | 2.0 |
| **1G** | 3.7 E4 | 1.2 E4 | 4 | 3.0 |

**Table 2**. TXP elapsed time to sort various file sizes using single-processor sort or multi-processor sort. The rightmost column shows the speedup of parallel sorting over serial sorting.

At our design point of a million 100-byte records, parallel FastSort gives a speedup of about two over a single-processor sort and it outperforms SyncSort on 4381 by a factor of 1.4 on the NonStop VLX processor.

## PERFORMANCE MEASUREMENTS OF FASTSORT

A specific benchmark is needed to discuss the performance of sort. Tandem uses the sort benchmark described in Datamation as a standard [Anon]. It describes an entry sequenced file of one million records. Each record is one hundred bytes long and has a ten character key. The input records are in random order.

In our tests, all discs are mirrored. We scale the input file to be one thousand, ten thousand, one hundred thousand, one million and ten million records. For each of these file sizes, we measure the elapsed time for the sort on the best configuration for a specific processor type. This best configuration includes Guardian 90, Disc Process 2 (DP2), 3107 disc controllers and 4314 discs. For the ten million record case, partitioned files are used, six mirrored input discs, six mirrored scratch discs, and six mirrored output discs. Figure 4 gives the resulting log-log plots of elapsed times.

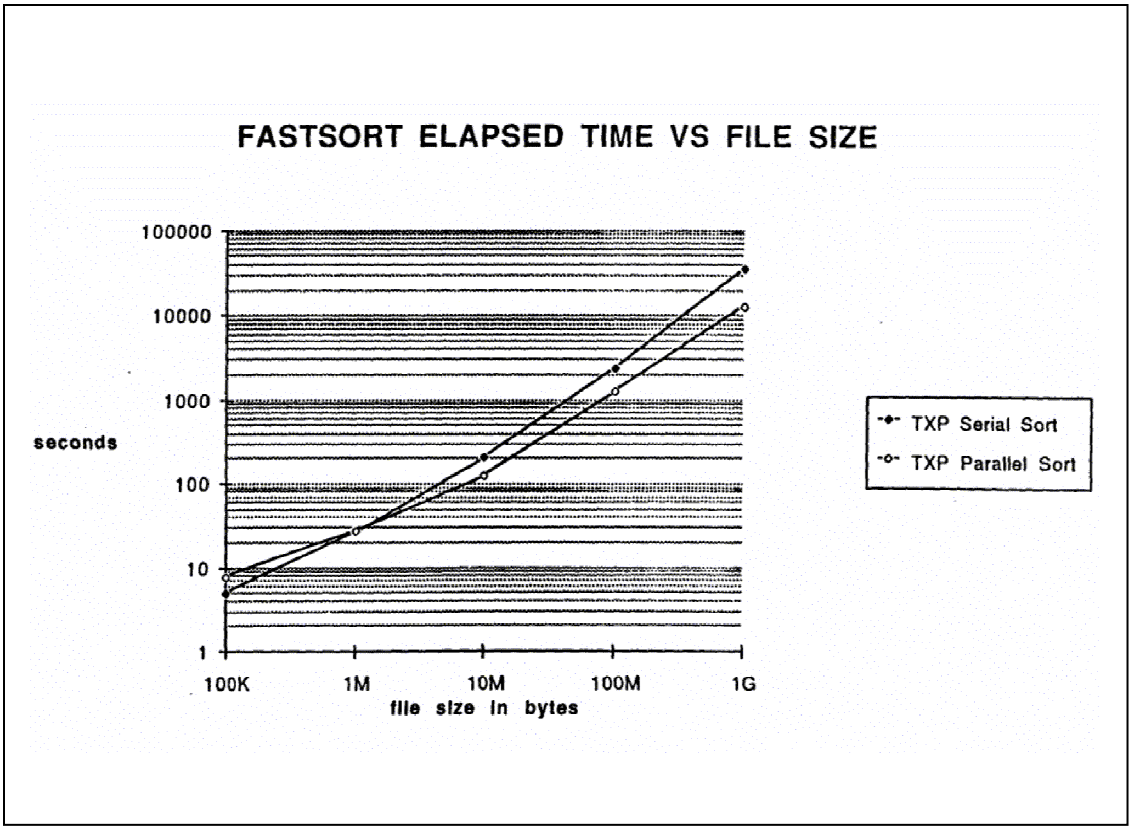


**Figure 4a**. FastSort elapsed time vs file size (in bytes) when run on a TXP processor. Note that parallel sort time increases linearly with file size beyond 10MB and that parallel sort is faster than serial sort for files larger than 1MB.

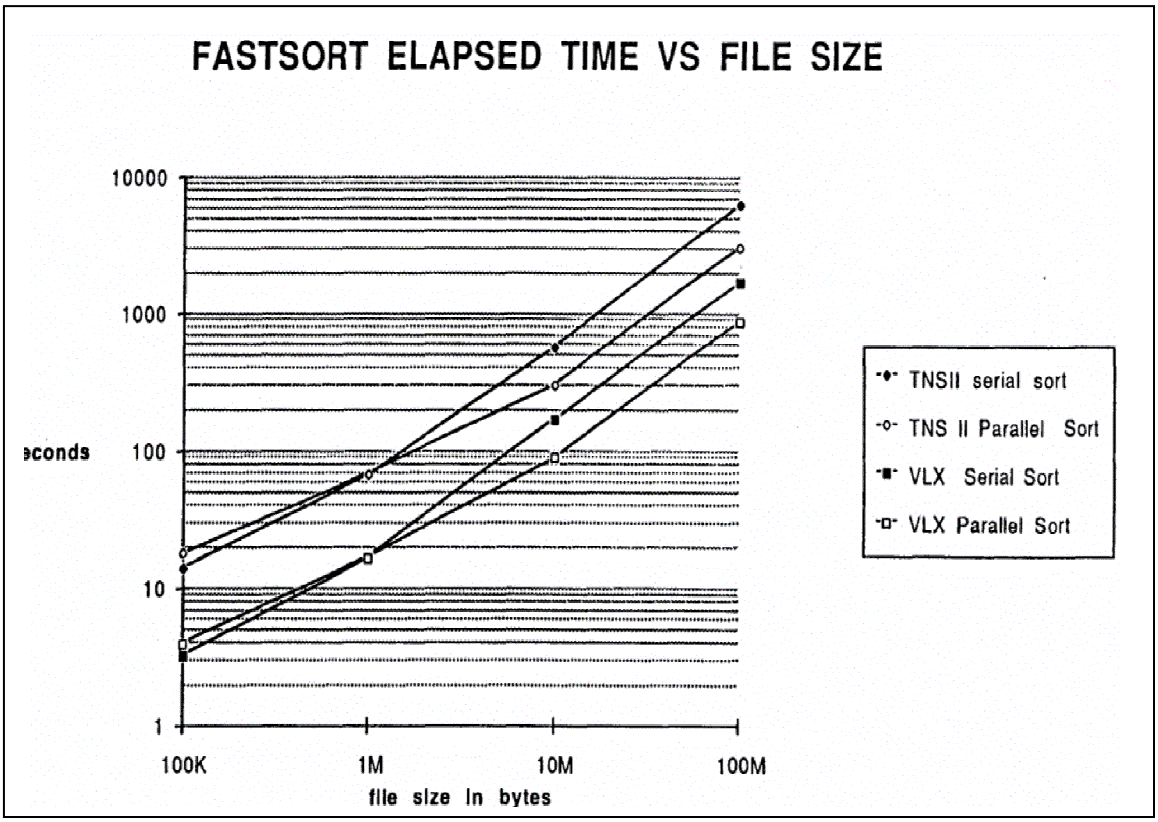


**Figure 4a**. FastSort elapsed time vs file size (in bytes) when run on a TNS II or TXP processor. Note that parallel sort is faster than serial sort and that CPU speed is the major determinant of sort speed.

Figure 4 shows that parallel sorting improves throughput by a factor of two over serial sort for one million records and a factor of three for ten million records. As the file sizes get even larger, parallel sorting becomes even more attractive. On the other hand, at the low end, below 1 Mbyte, parallel sorting is more expensive than serial sort because the process setup time dominates. The break-even point is about 10,000 records or about one Mbyte. Files smaller than this are best sorted entirely in main memory using a single processor.

Figure 5 shows FastSort's speed when sorting a million records, measured in records per second.

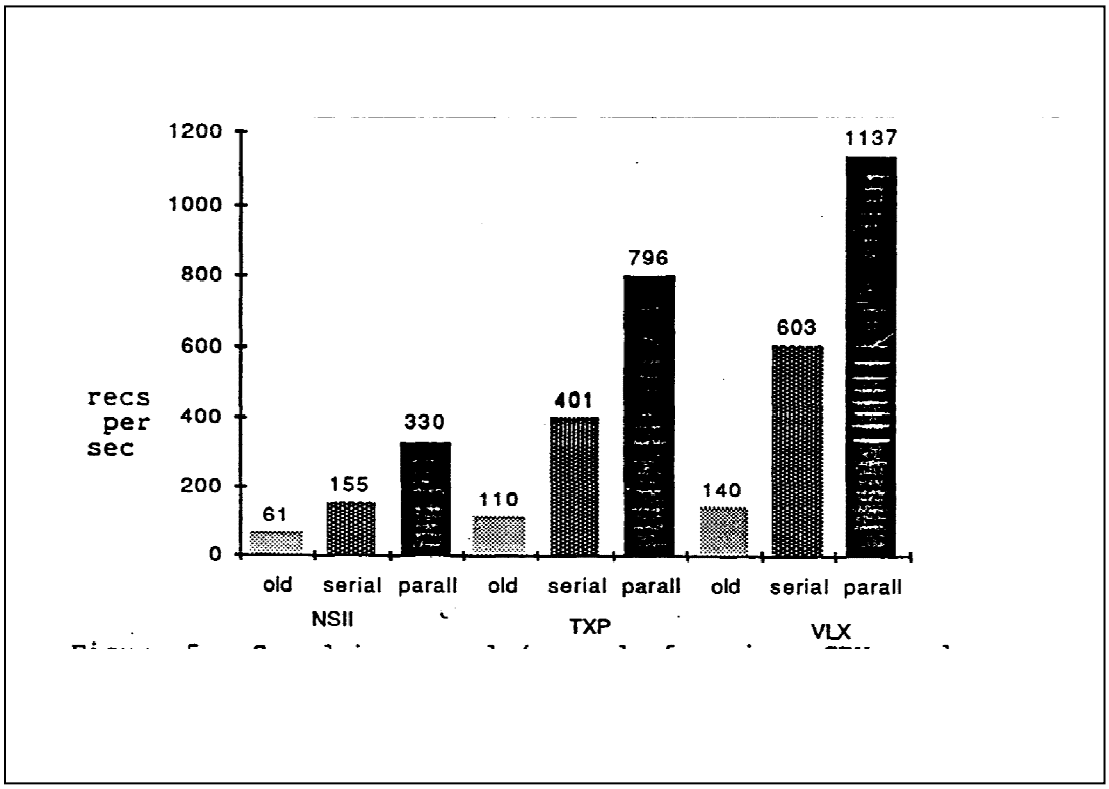


**Figure 5**. Speed in records/seconds of various CPUs and methods in sorting one million records.

Using the numbers in Figure 5, the time to sort "N" 100-byte records on the various processors can be roughly computed by dividing by the rate shown in Figure 5. For example, the NonStop VLX processor would take about N/603 = 166sec ~ 3 minutes to sort N = 100,000 records. Such calculations are very rough, but are helpful in giving estimates. Remember -- as the Environmental Protection Agency says about gasoline mileage rates for automobiles, "your actual mileage will vary depending on driving habits and road conditions". Your actual sorting speed may vary depending on your input stream, your configuration, and your concurrent workload.

The speed of FastSort varies very little with file type if the files are dense. Figure 6 gives the FastSort rate vs file type assuming the files are dense. The 5% difference between structured and unstructured files is caused by the differences between the routines used to deblock records. If the files are sparse, e.g. an almost empty relative file or a B-tree with slack, then FastSort will appear to run slower because it is reading some useless data. For

example, we observed that if the input and output files are B-trees with the typical 30% slack, then FastSort runs about 10% slower on a per-record basis.

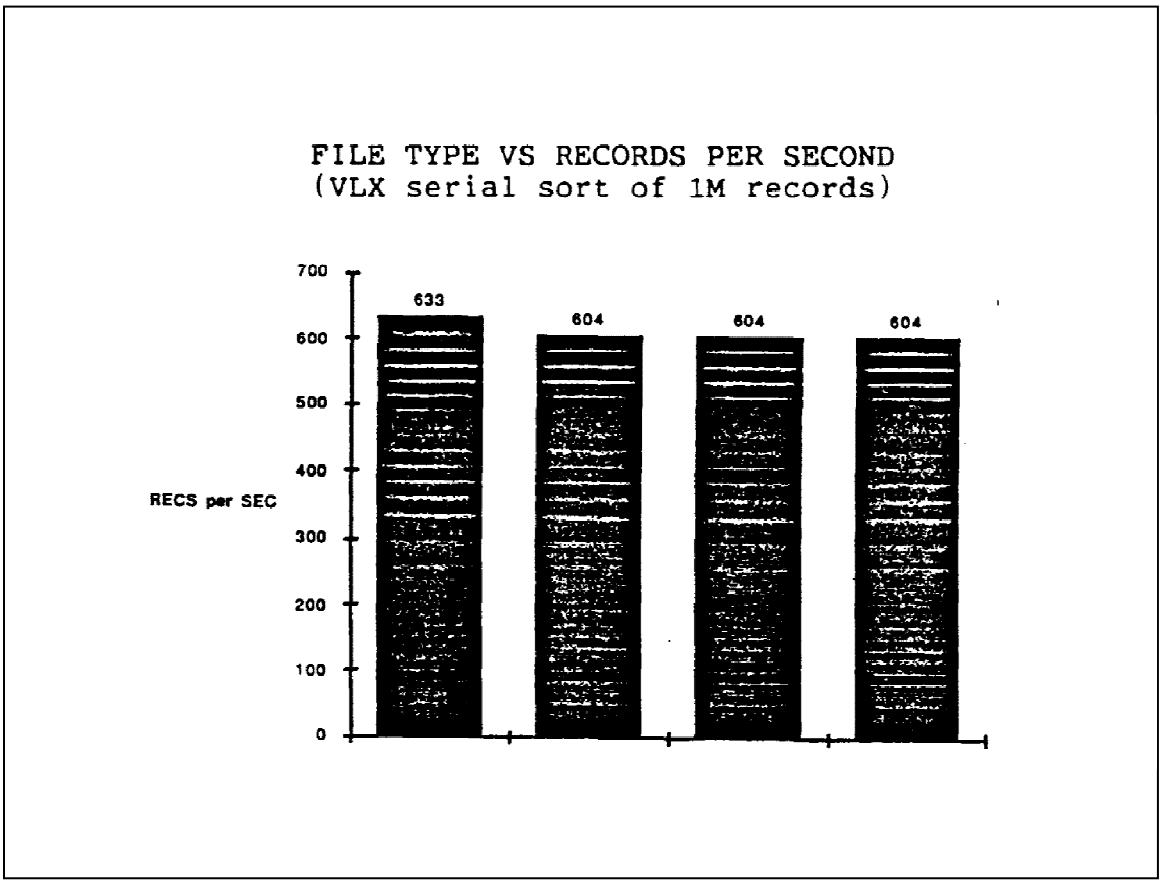


**Figure 6**: Speed single processor FastSort of a million records of various file types on a NonStop VLX processor.

Speed also varies with disc process type and block size. The old disc process, DP1, supports at most a 4 Kbyte file transfer size. The new disc process, DP2, supports up to a 30 Kbyte transfer size. Large transfer sizes reduce the number of disc and message transfers. Figure 7 shows the effect of block size on sorting speed. It indicates that using 16 Kbyte blocks improves throughput by about 30% over 4 Kbyte blocks. Using larger blocks (say 28K) does not provide additional savings.

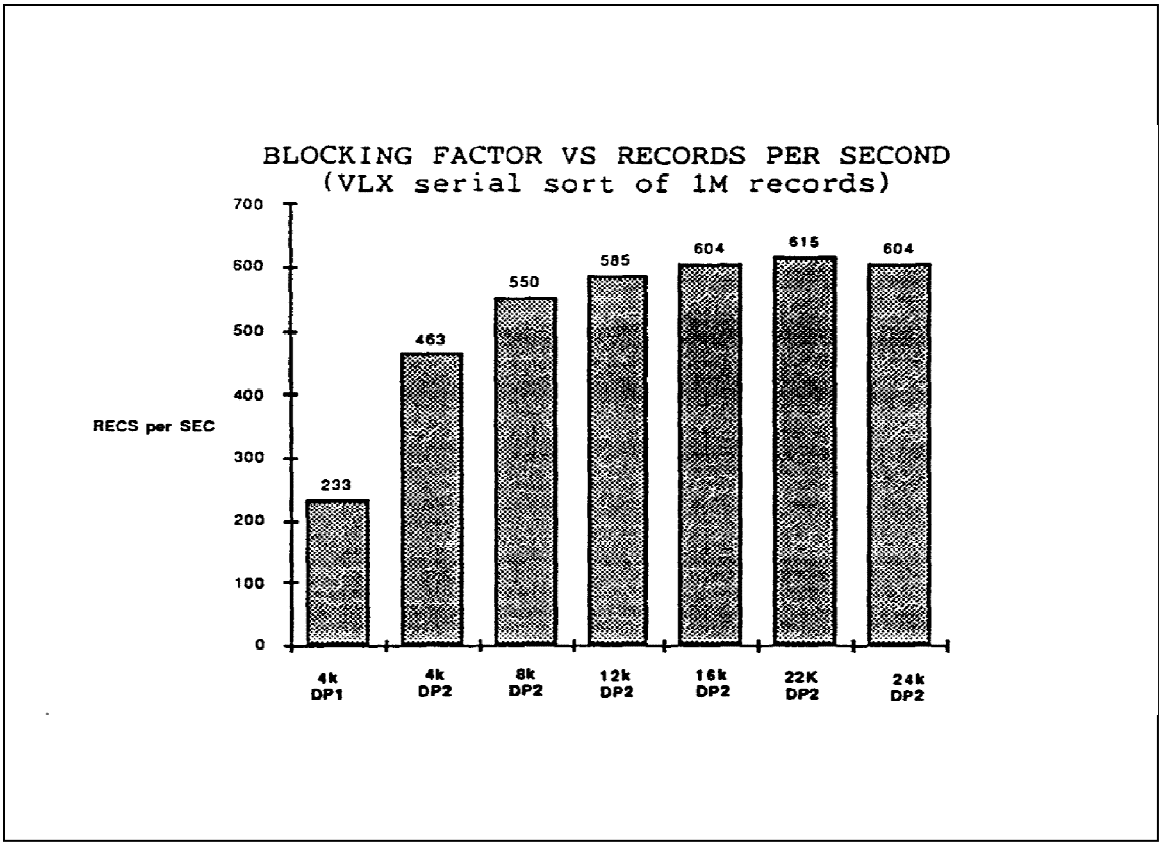


**Figure 7**: Affect of block size on speed of single processor FastSort of a million records on a NonStop VLX processor.

## SUMMARY AND CONCLUSIONS

While working on FastSort, we learned one lesson very well: when in doubt use brute force. Sophisticated algorithms gave small gains. But brute force techniques like faster processors, multiple processors, large memories, large block sizes, and double buffering yielded high payoffs.

By using all these techniques, FastSort is competitive with the fastest sort products. The novel feature about FastSort is that by using parallel processors and discs, it can sort in time proportional to the size of the input file, rather than the traditional nLog(n) time of conventional sorts. On a VLX for example, FastSort can sort at the rate of 110 Kbytes/second, independent of file size.

**ACKNOWLEDGMENTS**

John Shipman wrote the original Tandem SortMerge product. It was enhanced and refined by Bob Wells. The idea for parallel sorting is not new, but the work of Ed "Ash" Ashbaugh on SuperSort inspired us. During the implementation, we benefited from the sequential I/O improvements of DP2, and from the utilities which use these improvements provided to us by Joan Pearson, Janardhana Jawahar, and Charles Levine. David Hatala and Larry Watson provided us with performance numbers for SyncSort. Kevin Coughlin and Art Sheehan gave us valuable review comments on this document.

**REFERENCES**

[Anon] Anon Et. Al., "A Measure of Transaction Processing Power", *Datamation*, Vol.31, No. 7, 1 April 1985.

[Ashbaugh] E.R. Ashbaugh, "Large Scale Sorting Using Multiple Processing", Focus, Vol. 4, No. 2, Tandem Computers, Cupertino, CA, June 1984.

[Chmeil and Houy] Chmeil, T., Houy, T., "Credit-authorization Benchmark for High Performance and Linear Growth", Tandem Systems Review Vol. 2, No. 1, Cupertino, CA., Feb. 1986.

[Knuth] Don Knuth, The Art of Computer Programming, Vol. 3., Addison Wesley, 1973.

[Rosenberg] Eric Rosenberg, QSort Reference Manual, Dedalus Systems, Greenwich, CT., Sept. 1984.

[Tandem] Sort and FastSort Manual, Tandem Part Num: 82442-A00, Tandem Computers Inc., Cupertino, CA., June 1985.