# FastSort:
# A Distributed Single-Input Single-Output External Sort

Betty Salzberg[1]
Northeastern University, Boston, Massachusetts, 02115

Alex Tsukerman, Jim Gray, Michael Stewart, Susan Uren, Bonnie Vaughan
Tandem Computers, Cupertino, CA 95014-2599

**Abstract:** External single-input single-output sorts can use multiple processors each with a large tournament replacement-selection in memory, and each with private disks to sort an input stream in linear elapsed time Of course, increased numbers of processors, memories, and disks are required as the input file size grows This paper analyzes the algorithm and reports the performance of an implementation

## 1. Introduction

Although external sorting algorithms have been studied extensively, recent advances in computer architecture have made many of these algorithms obsolete For example, the leading source for state-of-the-art in 1973 [Knuth] devotes over one hundred pages to sorting data kept on tapes, while only fifteen pages discuss the use of disks and drums Knuth's discussion of disk sorting assumes no overlap between reading, writing and computing, and it assumes main memory is about 100 Kilobytes No parallelism is considered Today, external data is typically stored on magnetic disks rather than tape and large main memories (tens of megabytes) are common

Parallel algorithms can sort very large files in linear elapsed time by using multiple disks and multiple processors connected via an efficient local area network Such methods *scale up*, that is, their processing rate does not degrade as files get larger -- although they will require more processors and disks to store and process the larger files

FastSort is an example of a parallel external sort It executes on a loosely-coupled (shared-nothing) network of processors Each processor, here called a *site*, has its own main memory of up to 128 Megabytes, and its own disk drives, which are dedicated to sorting for the purpose of this discussion The processors are connected by a local area network [Uren]

FastSort assumes that the unsorted source file is at one site and sorts that file to another site Partitions of the file may be shipped to intermediate sites for processing The intermediate sites sort their parts in parallel

1 This research was partially supported by NSF Research Grant IRI-88-15707



The next section outlines *FastSort* and classifies it using Graefe's taxonomy Section 3 analyzes the method used to produce sorted runs, and Section 4 looks in detail at the merging process Section 5 gives experimental results

## 2. Outline of FastSort

FastSort can be described using the taxonomy proposed by Graefe for distributed external sorting [Graefe] He uses six factors to classify sorting methods

1 Whether the input is read from one source (single input) or is distributed over several sites (multiple-input) and whether the output is written to one destination (single-output) or to many (multiple-output)
2 How many times the data is shipped from site to site
3 Whether whole records or only keys and record IDs are moved
4 What main memory sorting method is used to create sorted runs
5 How sorted runs are merged
6 Whether the sorted runs are created before or after the data is moved between sites (This only applies to multiple-input sorting algorithms )

### 2.1 FastSort: Single-Input Single-Output Sort

FastSort sequentially reads the input file to be sorted This access is done as a single stream Similarly, FastSort writes a single output steam So, it is a single-input and single output sort.

FastSort's input and output are typically disk-resident files, but may be any devices (tape, network line, etc ), or any processes (typically the input or output of a relational operator) FastSort can consume and produce files and tables which are partitioned among sites, but it does this by sequentially accessing a single logical table which the underlying system transparently maps onto partitions distributed in the network

It is sometimes best to partition a file among several sites In this case, the partitions of the file can be read and processed in parallel For example, to build a B-tree index on a large file partitioned among many disks, several processes might be started to sort each partition of the file and then a parallel merge could combine these sorts into a partitioned index Similar ideas apply to using sorts for parallel query exaction GAMMA [DeWitt] and Volcano [Graefe] implement this approach FastSort does not yet have these optimizations

### 2.2 Records are Shipped Twice

FastSort ships each record to another site twice -- first the source is sent to the sub-sort sites and then the sub-sorts each send a sorted stream to the target site This data movement and the creation of sorted runs at the subsort sites is concurrent with the processing of the file at the source and sub-sort sites

The input file is generally stored in a format unsuitable for sorting The input is read into the main memory at the source site and reformatted into a message buffer, discarding unwanted records and record fields The resulting buffer is then sent across the network to the next round-robin subsort site The source site processing is the limiting factor during the first phase It does not exploit parallel processing The reading and reformatting work takes about ten instructions per byte and so, on the 3MIPS processors discussed here, limits FastSort to processing the input file at 300KB/sec (a rate of 270KB/sec has been observed)

Each subsort site sorts the data as it arrives from the source If the sorted data will not fit in main memory, the data overflows to disk as a set of sorted *runs*

The output phase begins after the input has all been read, compressed, distributed to the subsorts, and sorted by them into one or more runs The sorted runs at a given site are merged in one pass and the result is shipped to the destination site The destination site performs a supermerge of the streams from the subsort sites to produce a sorted file which is stored on the destination disk(s) Again, in this merging phase, the data transfer and the merging at the subsort sites are concurrent with the processing at the destination site Figure 1 gives an overview of the process

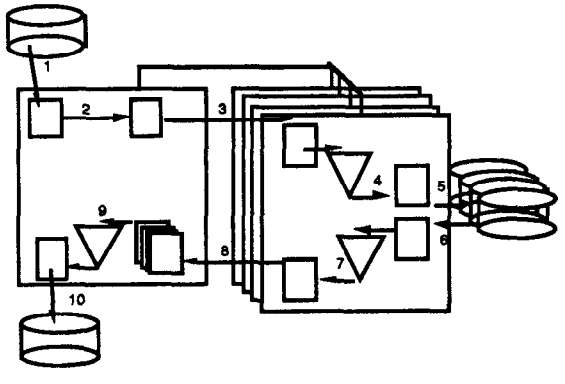


**Figure 1.** The data flow of FastSort The data moves from disk to memory (1), is compressed (select and project) as it is moved to network output buffers of each subsort (2), is sent over the network to the subsort sites (3), the subsorts use a replacement-selection sort to produce sorted runs (4), which are sequentially written to disk (5) After all runs have been created, they are read from disk by the subsorts (6), merged into a single run (7), and sent across the network to the destination (8) The destination does a supermerge of the subsort runs (9) and writes the sorted result to external media (10)

FastSort allocates enough memory so at most two passes over the data are needed, a pass to generate runs and a pass to merge them As shown later (Section 4 1), the required sorting memory is proportional to the square root of the file size Table 1 gives typical sorting memory sizes for various file sizes for single-site processing

**Table 1:** Typical memory requirements vs file size for a 2-pass sort

| Input File Size | Memory Size |
|---|---|
| 1MB | 0 1MB |
| 100MB | 1MB |
| 10GB | 10MB |
| 1TB | 100MB |

### 2.3 Record versus Key Sort

FastSort is a record sort, in the sense that whole records (after input projection and selection) are moved about the network for processing, rather than just moving the record keys

To give an example of a key sort, Lorie [Lorie] after sorted runs have been made at the input sites, merges *key-and-site identifier* pairs in parallel at several sites A list of site identifiers (indicating, for example, that the first record is to come from site 35 and the second from site 12 and the third from site 24 and so forth) is sent to the destination site so that no key comparisons are done at the destination site

For FastSort, key comparisons made at the destination site are not a bottleneck because, in most cases, it need only merge a small number (2 to 16) of runs (see Table 2 in section 4 3). The merge needs at most four compares per record These comparisons take less time than the network and disk overhead for moving the record

### 2.4 Main Memory Sorting Method

Replacement selection [Knuth] is used to create the sorted runs The basic advantages of replacement selection are twofold.

1 Sorted runs are twice memory size on average
2 Sorting is done incrementally and concurrent with input and output That is, by the time all the input records have been read and shipped to subsort sites, all the sorted runs except the last in memory have been created and stored on local disks at the subsort sites It is not necessary to wait to fill memory before the sorting can begin

Main memory sorting is further explained in Section 3

### 2.5 Intermediate Files

If a subsort site can sort all its data entirely in main memory, it does so If not, subsort sites store sorted runs on their local disks The sorted runs are then read off the local disk at each subsort site and merged at the subsort site before being shipped to the destination site

As explained in Table 1, FastSort acquires enough memory and adds enough sites so that only one merge pass is needed at each subsort site

### 2.6 Summary

FastSort is a single-input single-output external parallel sort Records are moved from the input site to several subsort sites Replacement selection with at most one merge pass is performed at each subsort site The sorted streams from each subsort site are sent to a destination site, where a supermerge takes place, creating the sorted file

## 3. Replacement Selection

Replacement selection (based on a tournament or heapsort algorithm) produces runs that are twice memory size on average [Knuth] This ability to produce long runs is a principle reason for using replacement selection

A second benefit of replacement selection is that it can execute concurrent with input and output Most internal sorting methods do not allow the overlap of processing with input and output For example, if Quicksort is used to create sorted runs, all the records must be read into memory before the sort can begin Then sorting takes place in memory and the sorted run is written to disk Then the records for the next sorted run are read into memory, and so forth Sorting cannot be concurrent with input and output if runs are to be memory-sized If shorter runs are acceptable, concurrency is possible Beck et al [Beck] manage to "overlap" Quicksort, by using only half of the memory for the sort while reading in the next run to be sorted to the other half of memory This produces runs of half-memory size and so four times as many runs to merge when compared to replacement-selection sort. This in turn affects the merge memory and computation requirements (see Section 4 1)

### 3.1 Replacement Selection Sorting

As input records arrive, they are added to *heap storage* which is arranged to accomodate variable length records These new records are conceptually "placed" in the empty leaves of a binary tree arranged as a tournament The interior nodes of the binary tree each represent a record whose key is the max of the keys of its children. The leaf nodes do not consume storage space, their parents point directly at the records The interior nodes contain pointers to the two children nodes/records, and a pointer to the "winning" record at that tournament node The record with maximum key is pointed to by the root of the tree, and the winner of each subtree is addressed by the root of that subtree (see Figure 2)

Keys of new records are moved up by comparisons with siblings until they are on the correct level of the tree With $n$ records, the *height* of the tree is $\lceil \log n \rceil$ So at most $\lceil \log n \rceil$comparisons are made at each cycle For example, a tree of height 16 can hold up to $2^{16}$ records and each new arrival requires at most 16 comparisons during tree construction and at most 16 comparisons during replacement

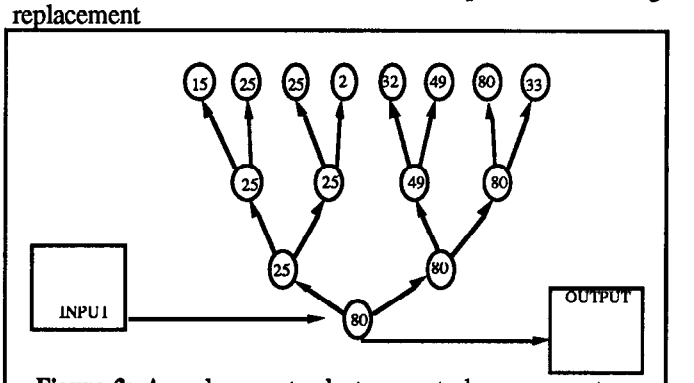


**Figure 2:** A replacement-selection sort showing input records entering the root of a sorting tree and the record with maximum key at the root of the tree Each node has a pointer to its two children and a pointer to the record "at" that node of the tournament The records themselves are stored in a separate area (not shown) until they move to the output buffer. The illustration shows the keys at the nodes, actually the nodes only store pointers to the records, and the keys are found inside the records Note that two records have key 25

### 3.2 Concurrent Processing and Input Output

After the largest possible tree is constructed in memory, records begin to overflow to disk-resident runs The runs are written sequentially by moving the record corresponding to the key in the root of the tree to the output buffer This creates a hole at the leaf and holes on the path from the leaf to the root A new record is inserted from the input at this leaf hole, and the algorithm recursively fills the parent of this leaf with a pointer to the maximum of the two siblings This is illustrated in Figure 3

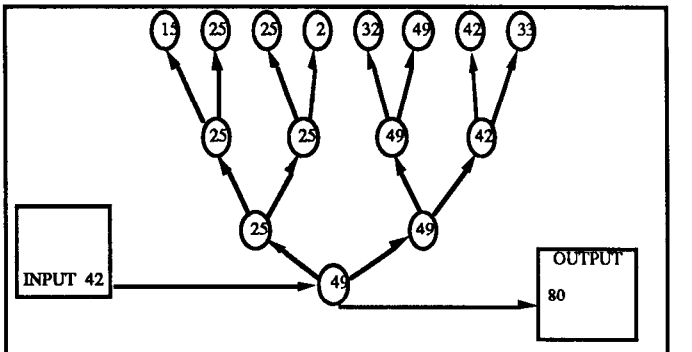


**Figure 3** The new state of the sorting tournament after a record with key 42 has arrived to fill the vacancy left by moving the root record to the output buffer

If the key of the new record is larger than the root it replaced, the new record "breaks the run", that is the new record starts a new run Such records are marked with a new run identifier (one bit only is needed) and do not participate in the current tournament Only nodes of the current tournament occupy interior nodes of the tree This is illustrated in Figure 4

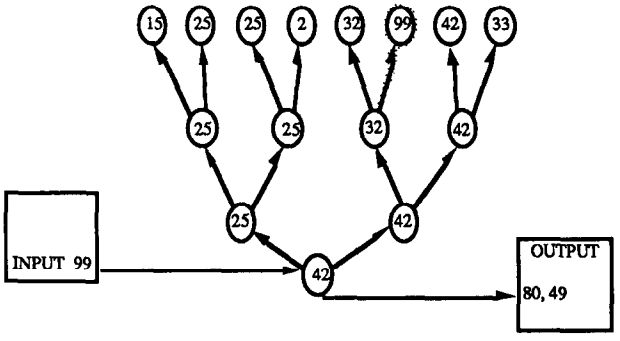


**Figure 4** When a new record arrives which is larger than the root, it "breaks" the run The new record starts a new tournament For example a record with key 99 arrived, and since it was larger than the key of the minimum key in the output buffer, this new record does not participate in the current tournament

As the process continues, the second run takes up more and more space in the sorting heap Finally, all the records in the tree are for the second run Then the process begins again

Reading the input and writing the output is a purely sequential operation (no disk seeks) and so can proceed at disk device speed Overlapping of input, output and sorting can be accomplished by double buffering both the input and the output While the records from one input buffer are being processed, the other input buffer is used to read the next set of records While records move from the root of the sorting heap to one output buffer, the previous output buffer is written to disk The next question to ask is whether or not there is enough *time* to process the records of one input buffer while the other input buffer is being read

### 3.3 Analysis of Subsort Run Generation Performance

In order to analyze the performance of FastSort and show how the timing works out, we need to establish some notation.

NOTATION

| | |
|---|---|
| $F$ | size of file in bytes |
| $R$ | record size in bytes |
| $n_{records}$ | number of records in the file |
| $M$ | memory space used for sorting at each site (bytes) |
| $\rho_{source}$ | rate in bytes/second of moving data from disk to network at source site (steps 1, 2, and half of 3 in Figure 1) |
| $\rho_{subsort}$ | rate in bytes/second of moving data from network to disk or from disk to network at subsort site excluding the management of the sorting heap (half of step 3, all of step 5, or conversely all of step 6 and half of step 8 in Figure 1) |
| $\rho_{target}$ | rate in bytes/second of moving data from network to disk at target site excluding the management of the sorting heap for the supermerge (half of step 8 and all of step 10 in Figure 1). |
| $\rho_{disk}$ | sequential disk I/O rate (bytes per second) |
| $t_{seek}$ | disk seek time (ms ) |
| $n_{sites}$ | number of sites |
| $t_{compare}$ | time to compare keys and copy one pointer (seconds) |
| $n_{runs}$ | number of runs at each site |
| $B$ | the blocksize, the number of bytes transferred in a disk io or message |

The previous section explained how the subsort tournament is is filled from the input buffers If the processor is not fast enough to move the contents of an input buffer through the tournament, before the next input buffer arrives, there are two alternatives·

1. Reduce the tournament size, thereby reducing the number of compares
2. Reduce the input rate by increasing the number of processors and subsorts

Consider these alternatives in turn.

#### 3.3.1 Reducing the Size of the Tournament

Reducing the tournament size reduces the processing time for inserting a new record Reducing the tournament size by a factor of 2 saves one compare Since the height of the tournament is typically between 15 and 20, this is a small savings for a big change (7% benefit for a 100% change) Decreasing the memory used also decreases the average size of the sorted run, and thus increases the number of sorted runs to be merged at the subsort site. Conversely, the tournament can be made much larger and still not be cpu bound if the record arrival rate is slow enough To discuss the tradeoff for optimal tournament size, the cost for inserting records in the tournament is computed as follows

Each tournament node has two 4-byte child pointers and a 4-byte pointer to the record "at" that node So each node occupies 12 bytes Each site has $M$ bytes of tournament memory The height of the tree is $\lceil log(\frac{M}{R+12}) \rceil$ The amount of time it takes to place the key-and-pointer pairs for a new record in the tournament is

$$insert\ time = t_{compare} \lceil log\ (\frac{M}{R+12}) \rceil \text{ seconds} \qquad (1)$$

#### 3.3.2 Increasing the Number of Subsort Sites

Doubling the number of subsort sites cuts the input rate in half This slow-down gives each subsort more time to process each record In particular·

$$local\ input\ rate = \frac{\rho_{source}}{n_{sites}} \text{ bytes/second} \qquad (2)$$

is the input rate from the point of view of a subsort site, and the record arrival interval at a subsort site is

$$subsort\ record\ arrival\ interval = \frac{R}{\rho_{source}/n_{sites}}$$

$$= \frac{R\, n_{sites}}{\rho_{source}} \text{ seconds/record} \quad (3)$$

To avoid the subsorts being cpu bound, the time between record arrivals must exceed the time needed to insert that record in the tournament and move another record to the output buffer and disk (see Figure 1) That is, using (1) and (3)

$$\frac{R\, n_{sites}}{\rho_{source}} \geq t_{compare}\lceil \log\left(\frac{M}{R+12}\right)\rceil + \frac{R}{\rho_{subsort}} \quad (4)$$

equivalently,

$$n_{site} \geq \rho_{source}\left(\frac{t_{compare}\lceil \log\left(\frac{M}{R+12}\right)\rceil}{R} + \frac{1}{\rho_{subsort}}\right) \quad (5)$$

Given $M$, this inequality yields the minimum number of sites needed to prevent the subsorts from being the bottleneck Note that $n_{sites}$ does not depend on the size of the file Sites can be added to slow the subsort input speed arbitrarily This allows time to process the records in huge tournaments (huge memories) All the available memory can be used for the tournament without being cpu limited if enough subsorting sites are used

Adding subsort sites is more effective than adjusting the tournament size when balancing subsort processing time against the rate of the source and target sites -- going from 3 to 4 subsorts cuts the per-site processing load by 33% One would have to go from a 16-level tournament (~6MB)to an 11-level tournament (~200KB) to get a similar reduction by reducing the tournament size. Adding subsort sites also adds memory to the sorting and merging process This contributes to the ability to sort the file in two passes (see Table 1) If massive amounts of memory (number of sites) are added, the data can be sorted in one pass

### 3.3.3 A Numerical Example

One of the experiments described in Section 5 used 3MIPS processors with one-megabyte sorting memories The record size was 100 bytes It took ten instructions per byte to process the records at the source site This means that FastSort processed 300 bytes in 3,000 instructions, which took one millisecond That is, $\rho_{source}$ = 300 bytes/ms Each tournament key comparison and exchange used 85 instructions At three mips this is 28 microseconds per compare Summarizing

| | |
|---|---|
| $R$ | 100 bytes |
| $\rho_{source}$ | 300,000 bytes/s |
| $\rho_{subsort}$ | 300,000 bytes/s |
| $\frac{\rho_{source}}{\rho_{subsort}}$ | 1 |
| $M$ | 1,000,000 bytes |
| $t_{compare}$ | 28 µs |
| $\frac{M}{R+12}$ | about 8,928 records in memory |
| $\lceil \log\left(\frac{M}{R+12}\right)\rceil$ | ~14 tournament levels in memory |
| $t_{compare}\lceil \log\left(\frac{M}{R+12}\right)\rceil$ | 392 µs |
| $\rho_{source}/R$ | 3000 records/s sent by source site |

Using these parameters to evaluate equation (5) gives a value slightly larger than 2 This means three subsort sites are enough to be able to process the incoming data each using a one megabyte tournament, no matter how large the file size Of course the larger file may have to add tournament memory (in the form of more subsorts) in order to sort the file in two passes

The needed memory is in a logarithmic term of equation (5), so substantial changes can be made without affecting the number of sites needed In the example above, if $M$ increases to 10MB, $\lceil \log\left(\frac{M}{R+12}\right)\rceil$ goes from 14 to 17 and still, only three subsort sites are needed If $M$ is increased to exploit massive main memory, more subsort sites may be added to decrease the record arrival rate at each subsort and avoid being cpu limited at any one subsort

### 3.4 Summary

The advantages of replacement selection are that input, output, and processing can be concurrent, and that sorted runs are twice memory size on average If the speed at which data arrives is too fast to have the sorting overlapped by input, a smaller tournament can be used or the number of subsort sites can be increased. Increasing the number of subsort sites is more effective since the number of sites is inversely proportional to the arrival rate, whereas the processing time is only proportional to the *log* of the size of the tournament.

## 4. The Merging Phase

The merge phase combines all the runs, merging them into a single large run that is sent to the target site If the input to a subsort site is less than the size of main memory no merging is done at the subsort site, the records are entirely sorted in memory and then passed to the destination site If the file partition sent to the subsort site is bigger than memory then merging of multiple runs is required The subsort merge is a small replacement-selection sort of the $n_{runs}$

FastSort tries to make at most one merge pass at each site Reading the runs and writing the output are concurrent with merging. This means that the merging time is the *max* of the input time, the send time, and the in-memory merge processing time

### 4.1 Space for Single Merging Pass

The goal is to make only one merge pass During the merge phase, there is one buffer in memory for each run and double buffers for output and input (This is unlike [Salzberg 89] which suggests using two buffers for each sorted run in order to make overlap of calculation with input more likely ) Each buffer is $B$ bytes This means that the memory at each subsort must be large enough to have one buffer for each sorted run and two buffers each for the input and output streams Ignoring the fixed cost of the four extra buffers

$$\frac{M}{B} \geq n_{runs} \tag{6}$$

Each sorted run has size $2M$ on average The total number of sorted runs depends on the file size The number of runs at each subsort site is approximately

$$n_{runs} = \frac{F/n_{sites}}{2M} \tag{7}$$

Combining (6) and (7)

$$\frac{M}{B} \geq \frac{F/n_{sites}}{2M} \tag{8}$$

equivalently,

$$n_{sites} \geq \frac{FB}{2M^2} \quad \text{and also} \quad M > \sqrt{\frac{BF}{2n_{sites}}} \tag{9}$$

If, for example $F$ is 100MB (one million 100-byte records), $M$ is 1MB, and $B$ is 16KB one site is sufficient for a one-pass merge Table 1 was computed from the second equation using $n_{sites} = 1$ More total memory will be required for the same file and blocksize as $n_{sites}$ increases, however, the memory requirement at each site becomes smaller (resulting in smaller run sizes)

### 4.2 Merge Time

The merge processing at the subsort sites should be concurrent with the merge processing at the destination site The processing rate at the subsort and at the destination are approximately the same -- they both have small tournaments

The merge time at each subsort site consists of the disk seek time and disk read time for all the runs stored at that site, the processing time to merge the runs into output buffers, and the time to send the data over the network to the destination The disk time is concurrent with the processing times

On average, the total number of bytes at each site is $F/n_{sites}$ There are B bytes in each buffer So the number of seeks is

$$seeks = \frac{F}{B\ n_{sites}} \tag{10}$$

Thus the time for merging at one site is

$$\textit{disk merge time} = \frac{F\ t_{seek}}{B\ n_{sites}} + \frac{F}{n_{sites}\ \rho_{disk}}$$

and

$$\textit{cpu merge time} = t_{compare}\lceil \log(n_{runs}) \rceil \frac{F}{n_{sites}\ R} + \frac{F}{n_{sites}\ \rho_{subsort}}$$

$$\textit{subsort merge time} = \max(\textit{disk merge time, cpu merge time}) \tag{11}$$

The tournament for the merge is in general much smaller than the tournament used for creating the sorted runs, which fills memory The number of nodes in the merge tournament is only equal to the number of sorted runs at the subsort site

To avoid having the subsort sites be a bottleneck during the merge phase, both the cpu merge time and the disk merge time in (11) must be less than the processing time for the whole file at the destination site The time for processing the whole file at the destination site is

$$\textit{destination merge time} = \frac{F}{\rho_{target}} + t_{compare}\lceil \log(n_{sites}) \rceil \frac{F}{R} \tag{12}$$

Typically, the second term of (12), which is the time for the supermerge in memory, is about one seventh of the total destination merge time To simplify calculations, we require the stronger condition that the cpu merge time at the subsort sites and the disk merge time in (11) be less than the first term of (12) only This will imply that they are less than the total destination merge time

Combining (11) and the first term of (12):

$$\frac{t_{seek}}{B\ n_{sites}} + \frac{1}{n_{sites}\ \rho_{disk}} \leq \frac{1}{\rho_{target}} \quad \text{and}$$

$$\frac{t_{compare}\lceil \log(n_{runs}) \rceil}{n_{sites}\ R} + \frac{1}{n_{sites}\ \rho_{subsort}} \leq \frac{1}{\rho_{target}} \tag{13}$$

equivalently,

$$n_{sites} \geq \rho_{target}\left(\frac{t_{seek}}{B} + \frac{1}{\rho_{disk}}\right) \tag{14}$$

and using (7)

$$n_{sites} \geq \rho_{target}\left(\frac{t_{compare}\lceil \log(F/(n_{sites}\ 2M)) \rceil}{R} + \frac{1}{\rho_{subsort}}\right) \tag{15}$$

One could find $n_{sites}$ to satisfy (15) by successively trying increasing positive integers As $n_{sites}$ becomes larger, the right-hand side of (15) becomes smaller Thus a solution must exist. For a quicker back-of-the-envelope approach, let $n_{sites}$ in the right hand side of (15) be two (If $\rho_{target} = \rho_{subsort}$, as in the examples, at least two subsort sites are needed, larger values of $n_{sites}$ will make the right hand side of (15) smaller, so this will be stronger than the original inequality ) The back-of-the-envelope inequality is thus

$$n_{site} \geq \rho_{target}\left(\frac{t_{compare}\lceil \log(F/(4\ M)) \rceil}{R} + \frac{1}{\rho_{subsort}}\right) \tag{16}$$

### 4.3 Numerical Example

Assume the same parameters as in Section 3 3 3 with a file size F of 100MB and 24 milliseconds for $t_{seek}$, 2 7MB/s for the sequential read rate $\rho_{disk}$, and 3MB/s for the target processing rate $\rho_{target}$ Applying equation (14) after converting all units to milliseconds and bytes

$$n_{sites} \geq 300\left(\frac{24}{16000} + \frac{1}{2700}\right)$$

This inequality says that for these parameters, one site is enough so that the disk I/O time at the subsort sites is concurrent with processing at the target site Here the large (16KB) buffers make the difference. Note that the size of the file and the memory size at the subsort sites is not involved

Applying (16) after converting all units to milliseconds and bytes

$$n_{sites} \geq 300\left(\frac{0\ 028\ \lceil log(25) \rceil}{100} + \frac{1}{300}\right)$$

For these parameters, two subsort sites are enough, but one is not This inequality compares the time needed for CPU processing (merging and sending to the network) at the subsort sites with the time for processing at the target site. The tournament for merging the sorted runs is much smaller than the tournament used in creation of the runs, so it is less likely to be the determining factor for choosing the number of sites

Combining equations (5), (9), (14), and (16), and using our other test parameters, one can compute the number of sites needed given various input file sizes $F$, and site memory sizes $M$ Table 2 tabulates these computations Equation (9), space for a single merging pass, is the determining factor for number of sites when the file is large and the subsort memories are relatively small In this case, increasing memory size at the subsort sites is recommended

**Table 2** Required number of sites for a two-pass parallel sort given a file size F of hundred-byte records, and given M bytes of memory at each site

| | site memory size = M | | |
|---|---|---|---|
| file size | 1MB | 16MB | 128MB |
| 100MB | 3 | 3 | 3 |
| 10GB | 80 | 3 | 3 |
| 1TB | 8,000 | 32 | 3 |

## 5 Experimental Results

There is a standard sort benchmark [Anon] sort an entry sequence file of one million records The records are one hundred bytes each, begin with a ten byte key, and are in random order To show the effects of scaling on FastSort's performance, the input file was modeled on this schema, but was scaled from a thousand records, to ten million records in powers of ten Various processor types and degrees of parallelism were tried The elapsed time was recorded in each case A complete description of the experiments and results is given in [Tsukerman] The results for one processor type (TXP) are shown in Figure 5 Not surprisingly, parallel sorting was quicker than serial sorting for files of more than 1MB For smaller files, the longer startup time of parallel sort makes it slower than a single-process sort The curves also show the linearity of FastSort for files of 10MB or more

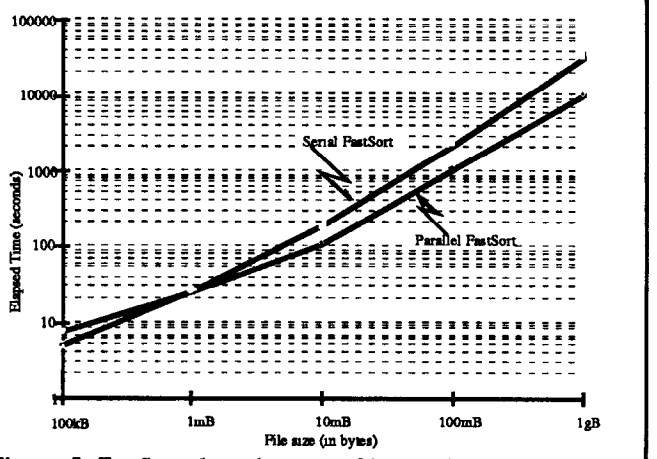


**Figure 5.** FastSort elapsed time vs file size (in bytes) when run on a TXP processor Note that parallel soft time increases only linearly with file size beyond 10MB, and that parallel sort is faster than serial sort for files larger than 1MB

## 6. Conclusions

Parallel FastSort is a linear elapsed-time sorting algorithm, running at the rate of about 130 KB/sec (~1300 records per second) on a Tandem VLX processor Simple equations allow FastSort to pick the amount of memory, number of processors, and number of disks to get linear-time performance Parallel sorting used in FastSort is faster than serial sorting because

1 The speed of the creation of sorted runs is cpu limited -- the cpu must make many comparisons in managing large tournaments Parallel sorting spreads this work among multiple cpus and so gets a speedup
2. The speed of the merge pass is I/O bound (disk seek time bound) Parallel sorting spreads this work over multiple disk drives on multiple subsort processors and so gets a speedup during the second phase of sorting

Both parallel and serial FastSort use double buffers with a tournament sort Consequently, replacement selection is concurrent with output and produces runs twice memory size The use of large buffers for I/O and the use of large main memory are also important Just these improvements, without using parallelism, gave a factor of four speedup over a previous serial algorithm

The current limiting factor of FastSort is the processing speed at the source and destination sites This bottleneck is

inherent in a single-input single-output sorts Obviously, a sort where the file begins distributed (unsorted) and ends distributed in sort-order ranges, each of which is sorted, will be faster and will have other bottlenecks Future work on FastSort will try to allow FastSort to take advantage of this possibility

## 7. References


[Anon] Anon et al , "A Measure of Transaction Processing Power," *Datamation* , Vol 31, No 7, April, 1985

[Beck] Beck M , Bitton D , Wilkinson, W K , "Sorting Large Files on a Backend Multiprocessor," *IEEE Trans on Computers*, C-37, (7) pp 769-778, July 1988

[DeWitt] DeWitt, D , Gerber, R , Graefe, G , Kumar, K , Heytens, M , and Muralikrishna, M , "GAMMA· A High Performance Dataflow Database Machine," Proc Twelfth VLDB, Japan 1986

[Graefe] Graefe, G , "Parallel external sorting in Volcano," Oregon Graduate Center TR No CS/E 89-008, June 1989

[Knuth] Knuth, D , *The Art of Computer Programming*, Addison Wesley, Reading, Ma , (1973)

[Lorie] Lorie, R A , and Young, H C , "A Low Communications Sort Algorithm for a Parallel Database Machine," Proc Fifteenth VLDB, Amsterdam, pp 125-134, 1989

[Salzberg 88] Salzberg, B , *File Structures An Analytic Approach*, Prentice Hall, Englewood, New Jersey, 1988

[Salzberg 89] Salzberg, B , "Merging Sorted Runs Using Large Main Memory," *Acta Informatica*, 27, pp 195-215, 1989

[Tsukerman] Tsukerman, A , et al, *FastSort, An External Sort Using Parallel Processing*, Tandem Technical Report TR 86 3, Part No PN87617, Tandem Computers, Cupertino CA April 1986

[Uren] Uren, S *Message System Performance Tests*, Tandem Systems Review, Tandem Computers In, Cupertino, CA , V 2 3, pp 27-31, Dec 1986