S. Todd

Algorithm and Hardware for a Merge Sort Using Multiple Processors

Abstract: An algorithm is described that allows log (n) processors to sort n records in just over $2n$ write cycles, together with suitable hardware to support the algorithm. The algorithm is a parallel version of the straight merge sort. The passes of the merge sort are run overlapped, with each pass supported by a separate processor. The intermediate files of a serial merge sort are replaced by first-in first-out queues. The processors and queues may be implemented in conventional solid logic technology or in bubble technology. A hybrid technology is also appropriate.

Introduction

Most conventional sorting algorithms operate on a single processor and require of order $n \cdot \log(n)$ cycles to sort n records. Examples are the merge sort [1, pp. 163–165] and quicksort [1, pp. 114–116]. There are single processor algorithms with sort times proportional to n, but these are only effective in certain circumstances. Address sorting [1, pp. 99–102] requires the spread of sort key values to be known and fairly random. Digital sorting [1, p. 170] is very good for main storage sorts of files with short keys. When secondary storage is used, the digit length has to be small to reduce the number of open files; the key then consists of many digits and the constant of proportionality of the sort is high.

A variety of multiple processor sorts exists, most of which require a very large number of processors, proportional to n or more. These are the network sorts [1, pp. 220–243], in particular Batcher's merge exchange sort [1, pp. 111–114], Thompson and Kung's mesh sorts [2], and Chen's parallel bubble sort [3]. Some of these sorts are very fast, but all require very special hardware and are impracticable for large files with current technology. Even proposed a sort using $\lceil (\log_2 n)$ processors and $4 \cdot \lceil(\log_2 n)$ tape units to sort in $3.2\lceil(\log_2 n)$ write cycles [4]. This sort is made very complicated by the necessity of rewinding tapes before they can be read.

We present a sort that is similar to Even's. It uses more sophisticated hardware, which makes it both faster and simpler. The basic algorithm permits $\lceil(\log_2 n) + 1$ processors to sort n records in $2n + \log_2 n - 1$ write cycles. This requires the storage of $2 \log_2 n$ intermediate queues of variable length and maximum total length n records. These can be implemented using conventional main storage or shift register (e.g., bubble) storage. Our queues differ from Even's tapes in that they can be read before they have been fully written, and no rewind is needed. There are variations on our basic algorithm requiring fewer resources.

The proposed sort is suitable for use when several processors are available, but not order n or more. Very simple processors, which are only required to do a merge, can be used. Our sort is faster for sorting general files than single processor sorts, but not as fast as the network sorts.

Our sort could be used in a low cost special purpose sorting machine. Sorting is traditionally used in batch processing and also now in efficient implementations of relational query systems (e.g., [5]). Our sort would form a natural part of a relational data base machine [6].

The algorithm is a variant of a straight merge sort [1, pp. 163–165]. The passes are run overlapped rather than serially. Each pass is supported by a separate processor. Reading from the output of one pass begins before the writing of that output is complete, so the intermediate structures are first-in first-out queues rather than files. When the number of records to be sorted is not an exact power of 2, the normal serial algorithm deals with the remainder at the end of each pass; our algorithm deals with it first.

There are several variations of the algorithm that are more suitable in certain circumstances. A multi-way merge sort reduces the number of processors. Small sections of data can be sorted before being introduced to the

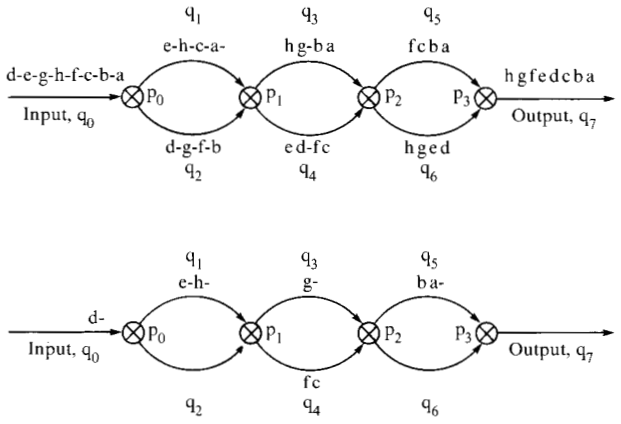

Figure 1 Four processors connected by six queues sort eight records. An overview of the sorting process is shown at (a), and a snapshot of the sort after seven cycles is shown at (b). The - indicates a break between strings of records.

Input		*Comments*
d e g h f c b a	front of file (input)	Initial file.
e h c a d g f b	pass 0 (first pair of intermediate files)	File is split into two files, each containing strings of length 1.
h g b a e d f c	pass 1 (second pair of intermediate files)	Length 1 strings are merged in pairs, e.g., a with b to create b a.
f c b a h g e d	pass 2 (third pair of intermediate files)	Length 2 strings are merged to make length 4 strings, e.g., f c b a
h g f e d c b a	pass 3 (output)	Sort ends by merging two length 4 strings to create a string of all 8 records.

Figure 2 A serial merge sort of eight records. Underscores mark the start of each string.

merge sort, which reduces the number of processors and the handling of very short queues. Blocking may be used to store queued records. Sort processors may be used to steer records directly from input to output queues, rather than to read them into buffers and later write them into queues. It may be best to let the processors run asynchronously. Various hardware techniques can support the algorithm, including solid logic and bubble technologies. Either can support both the processing and queue storage, or solid logic processors can be used with bubble queues.

In the next section we discuss the basic algorithm and analyze its processor and queue requirements. We show how multi-way sorting and presorting affect these requirements. The following section discusses more detailed variations of the algorithm that relate to the supporting hardware. Finally, hardware alternatives are covered, with general comments and three specific implementations.

The algorithm

The algorithm is a variation of a straight two-way merge sort [1, pp. 163–165]. A serial two-way merge sort operates in several passes, with each pass creating sorted sequences (strings) of records. The first pass creates strings of two records; the second pass merges these pairwise into four-record strings. After i passes, the strings have length 2^i. After $\lceil(\log_2 n)$ passes, all n records are in one sorted string.

In our variation the passes are run overlapped. We consider first the special case where the number n of records to be sorted is equal to 2^r for integer r and where there are $r + 1$ processors, 0 through r. The output from the ith processor consists of sorted sequences of 2^i records, created by merging two output strings from the $(i - 1)$th processor. Figure 1 shows the general setup for our algorithm, Fig. 2, the operation of a serial merge sort, and Fig. 3, the operation of an overlapped merge sort. We assume the processors run synchronously and can both read and write one record per cycle. Each processor starts when the previous processor has written one complete string and the first record of a second string.

The first process is finished before the last one starts; thus a single processor can be used for both. Alternatively, a sorter can be run almost as a continuous process. As soon as input of one file is finished, input of the next starts. The later processors act on the end of one file while the early ones act on the start of the next file.

With serial processing the natural merge sort reduces the number of passes needed when the file is already partially sorted. This is of less value with overlapped passes. The processors are usually preallocated, and only a few cycles are saved. Also, the storage requirements for the intermediate queues are less predictable.

In the remainder of this section we analyze the algorithm. We see how it must be adapted when n is not a power of 2, and we discuss the use of multi-way merge sorts and the presorting of blocks of records.

• *Analysis of the algorithm*

We establish sorting time and bounds on queue lengths. There are $r + 1$ processors $(p_0, \cdots, p_r)$ to sort $n = 2^r$ records. The output from p_i consists of 2^{r-i} sorted sequences of 2^i records each, $s_{i,1}, \cdots, s_{i,2^{r-i}}$. Processor p_0 breaks the input into separate records; $s_{0,j}$ consists of the jth input record. Sequence $s_{i,j} (j > 0)$ is created by merging $s_{i-1,2j-1}$ with $s_{i-1,2j}$. Processors p_i and p_{i+1} are connected by two queues, q_{2i+1} and q_{2i+2}. Output sequence $s_{i,j}$ from p_i is written to q_{2i+1} for j odd or q_{2i+2} for j even. Thus p_{i+1} always merges a sequence from q_{2i+1} with one from q_{2i+2}.

Cycles	Input	p_0	p_1	p_2	p_3	Output	Comments
0	d e g h f c b a	→	→	→	→		Input consists of length 1 strings.
		→	→	→	→		
1	d e g h f c b	→	a →	→	→		p_0 switches input to alternate queues.
		→	→	→	→		
2	d e g h f c	→	a →	→	→		a to upper queue, b to lower queue.
		→	b →	→	→		
3	d e g h f	→	c →	a →	→		p_1 begins to merge strings a and b.
		→	b →	→	→		
4	d e g h	→	c →	b a →	→		String b a is finished.
		→	f →	→	→		
5	d e g	→	h →	b a →	→		p_1 begins to merge strings c and f.
		→	f →	c →	→		
6	d e	→	h →	b →	a →		p_2 now starts on b a and f c.
		→	g →	f c →	→		
7	d	→	e h →	g →	b a →		p_0, p_1, and p_2 continue processing.
		→	→	f c →	→		
8		→	e →	h g →	c b a →		p_0 has passed the last record.
		→	d →	f →	→		
9		→	e →	h g →	f c b a →		p_2 completes f c b a.
		→	→	d →	→		
10		→	→	h g →	f c b a →		p_2 starts its 2nd output. p_1 ends.
		→	→	e →	d →		
11		→	→	h g →	f c b →	a	p_3 now starts.
		→	→	→	e d →		
12		→	→	h →	f c →	b a	p_2, p_3 still going.
		→	→	→	g e d →		
13		→	→	→	f →	c b a	p_2 ends.
		→	→	→	h g e d →		
14		→	→	→	f →	d c b a	p_3 is now operating exactly like pass 3 of the serial case.
		→	→	→	h g e →		
15		→	→	→	f →	e d c b a	
		→	→	→	h g →		
16		→	→	→	→	f e d c b a	
		→	→	→	h g →		
17		→	→	→	→	g f e d c b a	
		→	→	→	h →		
18		→	→	→	→	h g f e d c b a	p_3 ends. SORT COMPLETE
		→	→	→	→		

Figure 3 Overlapped merge sort of eight records.

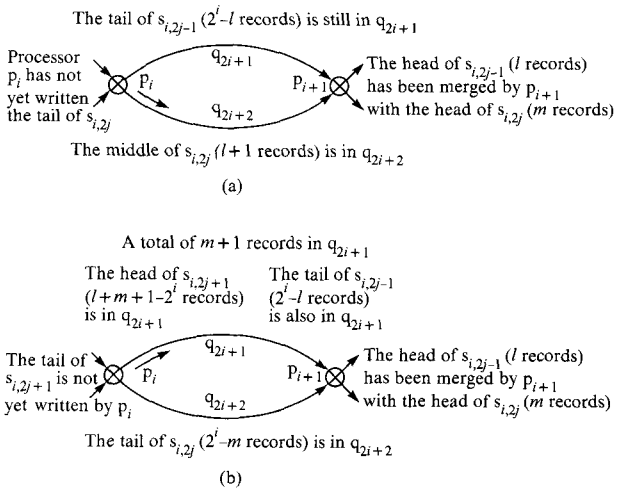

Figure 4 Lengths of intermediate queues. For case 1 at (a), p_i has not yet started writing into $s_{i,2j+1}$; for case 2 at (b), p_i is writing into $s_{i,2j+1}$.

Processor p_{i+1} starts operation as soon as there is input in both q_{2i+1} and q_{2i+2}, that is, $2^i + 1$ cycles after p_i. Processor p_0 starts in cycle 1; thus p_i starts in cycle

$$1 + \sum_{j=0}^{i-1} (2^j + 1) = \text{cycle } 2^i + i.$$

Processor p_i operates for n cycles and completes in cycle $n + 2^i + i - 1$. The sort ends with processor r in cycle $2n + \log_2 n - 1$.

A different way to find sorting time is to consider the case when the last record in is to be the first out. It passes p_0 in cycle n, p_1 in cycle $n + 1$, and so on. It emerges from the sort out of p_r in cycle $n + r$. The final record emerges from the sort $n - 1$ cycles later in cycle $2n + r - 1$.

We establish bounds on queue lengths. Consider a time when p_i and p_{i+1} are both operating. Processor p_{i+1} is about to write into $s_{i+1,j}$, into which it has already read the head (l records) of $s_{i,2j-1}$ and the head (m records) of $s_{i,2j}$. Processor p_i started writing into $s_{i,2j}$ one cycle before p_{i+1} started writing into $s_{i+1,j}$. Either

1. Processor p_i has not yet started writing into $s_{i,2j+1}$, or
2. Processor p_i has started writing into $s_{i,2j+1}$.

Figure 4 illustrates these cases.

In case (1) p_i has written $l + m + 1$ records into $s_{i,2j}$ and none into $s_{i,2j+1}$. The upper queue, q_{2i+1}, contains the tail of $s_{i,2j-1}$ ($2^i - l$ records). The lower queue, q_{2i+2}, contains the middle of $s_{i,2j}$ ($l + 1$ records, all those $l + m + 1$ records written by p_i except those m records already read by p_{i+1}).

In case (2), p_i has written all 2^i records into $s_{i,2j}$ and $l + m + 1 - 2^i$ records into $s_{i,2j+1}$. The upper queue contains the tail of $s_{i,2j-1}$ ($2^i - l$ records) and the head of $s_{i,2j+1}$ ($l + m + 1 - 2^i$ records), a total of $m + 1$ records. The lower queue contains the tail of $s_{i,2j}$ ($2^i - m$ records).

Since $l \le 2^i$ and $m \le 2^i$, in either case:

$$1 \le \text{length}(q_{2i+1}) \le 2^i + 1,$$

$$0 \le \text{length}(q_{2i+2}) \le 2^i,$$

$$\text{length}(q_{2i+1}) + \text{length}(q_{2i+2}) = 2^i + 1.$$

During the period before p_{i+1} starts and after p_i stops, the queues may be shorter than these rules imply.

The equation for the sum of the queue lengths is easily derived from the fact that p_{i+1} starts after p_i has written $2^i + 1$ records into these queues, and that from then until p_i finishes each write cycle sees one record written by p_i and one read by p_{i+1}.

• *Number of records not a power of two*

Consider the case where the number (n) of records is not a power of two. The serial straight two-way merge sort deals with the remaining "short" sequences at the end of each pass. This can be improved in the parallel case by taking the short sequences first. This does not upset the convenient property that merge sort preserves the order of records with equal sort key.

Let $r = \lceil(\log_2 n)$. We still need $r + 1$ processors. If the short sequences are taken at the end in the parallel case, p_r starts at cycle $2^r + r$. The sort ends after $n + 2^r + r - 1$ cycles.

Taking the short sequences at the front is more complicated. We initialize all processors as if they had already operated on $2^r - n$ very small pseudo-records. For $i = 1$ to $r - 1$, p_i still starts operation $2^{i-1} + 1$ cycles after p_{i-1}. Processor p_r starts as soon as there is a record in both its input queues. String $s_{r-1,1}$ contains all the pseudo-records, and thus only $2^{r-1} - 2^r + n$ real records. So p_r starts $2^{r-1} - 2^r + n + 1$ cycles after p_{r-1}; that is, in cycle $2^{r-1} + r - 1 + 2^{r-1} - 2^r + n + 1 =$ cycle $n + r$. The sort finishes in $2n + r - 1$ cycles.

We illustrate this with $n = 5$, $r = 3$ (Fig. 5). The operation of p_0, p_1, and p_2 is similar to their operation in the case $n = 8$ (Fig. 2), only three cycles earlier and with "a," "b," and "c" replaced by pseudo-records. Processor p_2 could have started in cycle 5 rather than cycle 6, but it would have been held up in the next cycle waiting for the "d." Processor p_3 is able to start in cycle 8 without risk of being held up. It is amusing when the last record is the smallest to watch the lower queues clear themselves to let it through to the front.

• *Multi-way merge sort*

A multi-way merge sort can be operated with all passes in parallel in the same fashion as a two-way merge sort. Fewer, more powerful processors are needed.

For a k-way merge sort $r + 1 = \lceil(\log_k n) + 1$ processors sort n records in $2n + r - 1$ cycles. The output of p_i consists of strings of length k^i, which are merged k at a time

Cycles	*Input*	p_0	p_1	p_2	p_3	*Output*	*Comments*
0	d e g h f	→ →	± → ±	→ →	→ →		At start of sort, three pseudo-records are already processed.
1	d e g h	→ →	± → f	→ →	→ →		Sort begins exactly as in Fig. 3, cycle 3, with a, b, c replaced by pseudo-records (+).
2	d e g	→ →	h → f	± → ±	→ →		
3	d e	→ →	h → g	± → f ±	± → →		
4	d	→ →	e h →	g → f ± →	± ± → →		All continues as in Fig. 3.
5		→ →	e → d →	h g → f →	± ± → →		
6		→ →	e →	h g → d →	f ± ± → →		
7		→ →	→ →	h g → e →	f ± ± → d →		p_2 starts its second string.
8		→ →	→ →	h g → →	f c b → e →	d	p_3 writes d because p_3 does not need to wait to process the pseudo-records.
9		→ →	→ →	h → →	f c → g →	e d	
10		→ →	→ →	→ →	→ h g →	f e d	
11		→ →	→ →	→ →	→ h →	g f e d	
12		→	→	→	→	h g f e d	

Figure 5 Overlapped merge sort of five records. The pseudo-records are shown by +.

by p_{i+1} into strings of length k^{i+1}. Each cycle involves each processor in of order $\log_2 k$ comparisons.

The intermediate strings between p_i and p_{i+1} are held in k queues, $q_{ki+1}, \cdots, q_{ki+k}$. These are known as queue set $i + 1$. They have total length $(k - 1)k^i + 1$. Processor p_{r-1} produces strings of length k^{r-1}, so it only requires $\lceil (n/k^{r-1}) \rceil$ output queues. Thus the total number of queues is $(r - 1)k + \lceil (n/k^{r-1}) \rceil$.

Table 1 shows the number of processors and queues for various values of k and n. The best choice of k depends on the comparative cost and speed of processors and queues for a particular implementation.

Table 1 Effects of multi-way merges. Each entry gives number of processors:number of queues.

Total no. of records	*Number of queues merged by each processor*						
	2	*3*	*4*	*5*	*6*	*8*	*16*
100	8:14	6:14	5:14	4:14	4:15	4:18	3:23
1000	11:20	8:20	6:20	6:22	4:23	5:26	4:36
10 000	15:28	10:26	8:27	7:29	7:32	6:36	5:51
100 000	18:34	12:32	10:34	9:37	8:39	7:44	6:66
1 000 000	21:40	14:38	11:40	10:43	9:46	8:52	6:80

Table 2 Each entry gives the number of processors required for a two-way merge sort using presorting.

Total no. of records	*Number of records in each presort*					
	1	*2*	*4*	*8*	*16*	*32*
100	8	7	6	5	4	3
1000	11	10	9	8	7	6
10 000	11	12	13	15	11	10
100 000	18	17	16	15	14	13
1 000 000	21	20	19	18	17	16

• *Presorting*

The first few processors deal with very short strings. They can be eliminated if p_0 carries out an "in core" presort of sets of s input records. The output from p_0 consists of strings of s records, p_i produces strings of $s \cdot k^i$ records, and only $\lceil[\log_k (n/s)] + 1$ processors are needed to sort n records.

The number s should be chosen so that p_0 can carry out the sort ($s \cdot \log_2 s$ comparisons) as the records are read in. It can then write the smallest record of the first string as it reads the first record of the second string. This gives a delay of s read cycles before the first write cycle. The processor p_0 requires storage for s records. Research at the University of Strathclyde on the LEECH processor [7] suggests that a very cheap special purpose processor should be able to handle values of s up to 50 or more. Table 2 shows the effect of presorting on the number of processors used.

When presorting is used, sort time is $2n + \lceil[\log_k (n/s)] - s$ write cycles plus s read cycles before the first write cycle. For k and s much smaller than n, this is very close to sort time without presorting.

Presorting can make the total lengths of the queue sets dependent on record length. The total length of queue sets $i + 1$ becomes $(k - 1)s \cdot k^i + 1$ records. Suppose s be limited by the storage B available to p_0 to $s = \lfloor(B/l)$ for record length l bytes. The total queue length $\{[(k - 1)sk^i + 1]\, l\}$ bytes is slightly less than $B(k - 1)k^i + B$ bytes. This simplifies the design of hardware for the intermediate queues. Space can be efficiently allocated to the queue sets at an early stage, which reduces allocation problems at execution time.

Variations of the algorithm

We discuss in this section variations of the algorithm that are made to suit particular implementations. They are the blocking of the records held in queues; overlapping input, comparison, and output; and synchronous operation. These are not important in the general behavior of the algorithm but must be considered for a specific implementation.

• *Blocking*

In some implementations of queues the designer may prefer to deal with records in blocks rather than individually. Blocking delays the transfer of a record from p_i to p_{i+1} and slightly slows down the sort.

If presorting is not used, the effect of blocking on the early processors is complicated as several strings fit into one block. We only consider the case where the number of records in a block is equal to the length s of strings produced by the presort, which is generally the most convenient in implementation. The sort time becomes $2n + s\lceil[\log_k (n/s)] - 1$. If s records are blocked into block size B, the size of queue set $i + 1$ becomes exactly $B(k - 1)k^i + B$. Thus if a fixed block size is used independently of the record length, there is a fixed requirement for intermediate storage.

As long as no block holds records from several strings, processors complete a block for one output queue before starting a block for another. Thus a single buffering store and write channel can be shared by all output queues.

• *Overlapping input, comparison, and output*

For some implementations the processors see the records as a stream of bits. Rather than reading, comparing, and writing records, a bit serial comparison is made and the data steered to the correct place. This applies only with fixed length records with the key at their head.

Figure 6 shows the setup for overlapped input, comparison, and output when merging two strings. Data are read one bit at a time into a processor, which controls the crossover switch and inhibitors. In a normal cycle the processor is comparing the loser record (that with the larger key) from the previous cycle (which is in the buffer loop) with the next record from the winner record's queue. The loser record's input queue is inhibited. As long as the records are identical, the status of the crossover switch is not important: one stream of bits flows into the output queue and an identical stream into the buffer

loop. As soon as a difference is recorded, the switch is set to steer the smaller (winner) record to the output queue and the loser into the buffer loop.

During the copy sequences, no comparisons are needed. To keep a steady stream of data the output is directed from the input around the buffer loop. In the last cycle of a copy sequence, the first record from the next string of input 1 is steered into the buffer loop. The processor is then ready to start work on its new output string as soon as the first input 2 record arrives. The output queue control switch is flipped, but there is no break in the flow of input data.

When $k > 2$ strings are to be merged, $k - 1$ comparators must be cascaded.

• *Timing and asynchronous operation*

In the analyses so far, we have assumed the processors to be synchronized by their output cycles. This may prove inconvenient because of the different number of reads to be carried out between the writes, because of different comparison costs in each cycle or because of a queue access being held up for implementation reasons (e.g., storage conflict). Reads and writes can be balanced by buffering, but the other problems make asynchronous operation desirable.

No data comparisons are necessary during a copy sequence. Thus the amount of work involved in an output cycle, particularly in the multi-way merge, is not constant. Also the queues may not be implemented completely independently, and storage access may delay a processor. For these reasons it may be required to run the processors asynchronously. If p_{i+1} operates too fast for p_i, p_{i+1} will attempt to read an intermediate queue and find it empty. Thus p_{i+1} must then wait until the required record is written by p_i. If p_i operates too fast for p_{i+1}, p_i may find that the storage available for the intermediate queues becomes full; then p_i must wait for p_{i+1} to read some records.

Analysis of asynchronous operation involves the comparative speed of the processors and queue storage, the degree of independence of the queues, and the details of the file being sorted. With processors appreciably faster than storage and reasonably independent queues, the performance is comparable to that with synchronous operation.

Hardware

The algorithm is suitable for implementation on a wide range of hardware. We discuss in general terms the implementation using solid logic technology and bubble technology, with a specific implementation in each and a hybrid implementation. We show how the variants of the algorithm are used in different circumstances. Other technologies (e.g., multi-channel disks) are not discussed.

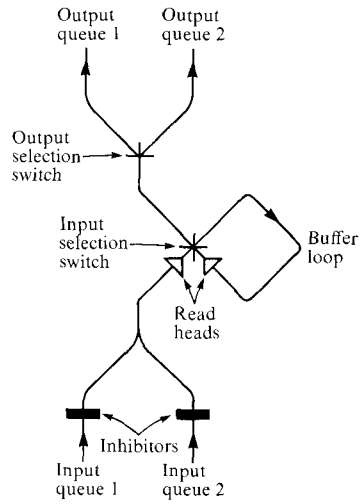

Figure 6 Setup for overlapped input, comparison, and output. Data are read from the read heads into a processor, which controls the selection switches and the inhibitors.

• *Solid logic technology*

Solid logic technology is good at providing random access and powerful processors but less good with independent access to several queues. In an implementation of the algorithm in this technology we would use presorting and a multi-way merge sort to minimize queue traffic.

Simultaneous accesses to a single module of main storage are very complicated (or impossible). Thus we allocate one storage module for each queue set. The queues are implemented by chaining. The bounds on queue set size determine the size of the storage modules.

With suitable buffering each processor requires one read and one write access per write cycle. All processors can be made to read during the first part of a cycle and to write during the second part. This avoids contention of one processor writing a queue while the following processor is reading it. Each processor requires a small local store for the buffered values.

If the sorting rate is limited by the speed with which data can be fed to a device, a very fast store is not appropriate except for the storage used for the presort. With fairly fast processors a possible design carries out a presort to produce strings of records to a total of 1 Kbyte, where K = 1024, and subsequent processors handle a 16-way merge sort. A device with a sort capacity of 256 Kbytes is shown in Fig. 7.

• *Bubble technology*

Magnetic bubble devices are particularly suited to handling streams of data, but not to random access. The bit serial merge of Fig. 6 complete with bubble comparator could be implemented in bubble technology, but presorting is difficult.

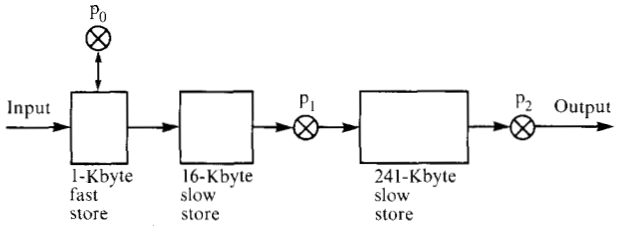

Figure 7 Three-processor solid logic technology sorter for files up to 256 Kbytes. A 16-way merge is used.

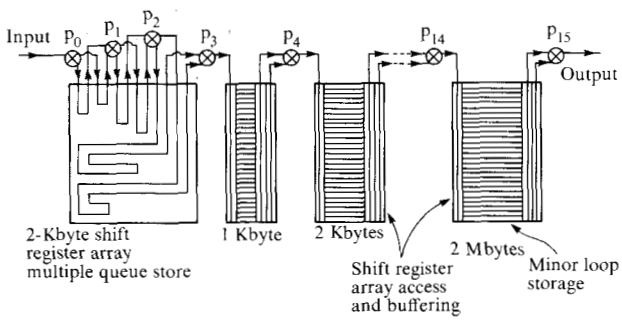

Figure 8 A 16-processor bubble technology sorter for files up to 4 Mbytes. A two-way merge is used.

Several bubble register organizations can support multiple queues with simultaneous access. It is too early to predict which would be the most economical. The most complicated (the shift register array) handles records independently; the others use blocking.

The shift register array [6] was designed for the storage of multiple queues. The operation is analogous to the use of many tapes with independent read and write heads; the problem is the neat stacking of the tape between the heads. The bounds on queue lengths simplify the dynamic allocation of space between them.

Two other mechanisms use blocking. Each can be seen as an array of data, with each column representing a block and several active columns from which data can be read or written. The columns can be moved to transfer them to an active column position when they are to be accessed. The control of the blocks can be handled by some external processor, which remembers which is the next block in each queue. Alternatively, the blocks can be self-identifying and retrieved associatively.

The first of the blocking mechanisms stores data using a major-minor loop scheme [8]. Because simultaneous access to several blocks is needed, the data are retrieved via a small section of shift register array rather than via a single major loop [9]. This allows several blocks to be accessed together, with buffering and unblocking handled in the store.

The second blocking mechanism makes use of much more tightly packed bubbles in an array [10]. This scheme operates faster but does not permit one block to be accessed while another is being moved towards an access column. Thus some additional support is needed to buffer and unblock, or there will be interference between queues.

Of the above, the shift register array is the only scheme which effectively deals with the very short queues encountered between the early processors. The other mechanisms are more efficient for holding long queues. The shift register array can handle many queues with little interference, the other mechanisms only a few. Thus queues produced from several processors can be implemented in one shift register array, but for the blocking schemes independent modules are needed for the different sets of queues.

Figure 8 gives a schematic diagram of one implementation of the algorithm using bubble technology. All processors use bit serial merge. The early queues are implemented in a shift register array. The later queues use minor loops accessed via a shift register array.

The first section is organized to produce strings of records to fit a 1-Kbyte buffer. The programming depends on the record length. There are sufficient queues and processors to support the smallest record length; for longer records some of them are not used. This section is effectively a presorter.

The later sections each have shift register arrays to block and buffer two input queues and buffer and unblock two output queues. The block size is 1 Kbyte. The ith section has minor loops long enough to hold 2^{i-1} blocks. A total of 12 of these sections gives the last a capacity of 2 megabytes. The total storage required is just over 4 Mbytes (where $M = K^2$), which is the sorting capacity of the device.

- *Hybrid implementation*

Many combinations of hardware can be used. Figure 9 gives an example. A solid logic front end is combined with a back end using solid logic processors and bubble queues. Four-way merge sort is used to make reasonable use of the processor without overcomplicating queue interaction in the store.

Summary

We have discussed the application of multiple processors to a merge sort. Files are replaced as intermediate storage structures by first-in first-out queues. For a k-way merge sort, $\lceil(\log_k n) + 1$ processors sort n records in just over $2n$ cycles. The algorithm has wide application where several processors, but not order n processors, are available.

Implementation involves considering several variations of the algorithm and the hardware to support it. Solid

logic and magnetic bubble technologies can be used to implement the hardware, or a hybrid of these technologies can be used.

References

1. D. E. Knuth, "Sorting and Searching," *The Art of Computer Programming*, Vol. 3, Addison-Wesley Publishing Company, Reading, MA, 1973.
2. C. D. Thompson and H. T. Kung, "Sorting on a Mesh-Connected Parallel Computer," *Commun. ACM* **20,** 263 (1977).
3. T. C. Chen, K. P. Eswaran, V. Y. Lum, and C. Tung, "Apparatus for Transposition Sorting of Equal-Length Records in Overlap Relation with Record Loading and Extraction," U.S. Patent Application Serial Number 685,859, 1977.
4. S. Even, "Parallelism in Tape Sorting," *Commun. ACM* **17,** 202 (1974).
5. S. J. P. Todd, "The Peterlee Relational Test Vehicle—A System Overview," *IBM Syst. J.* **15,** 285 (1976).
6. S. J. P. Todd, "Bubble Memory for High Level Databases," *Proceedings of IERE Conference on Computer Systems and Technology*, London, 1977.
7. D. R. McGregor, R. G. Thomson, and W. N. Dawson, "High Performance Hardware for Database Systems," *Systems for Large Databases*, Preprint, P. Lockemann and E. J. Neuhold, eds., North Holland Publishing Co., Amsterdam, 1976.
8. *Magnetic Bubble Technology*, H. Chang, ed., IEEE Press, New York, 1975.

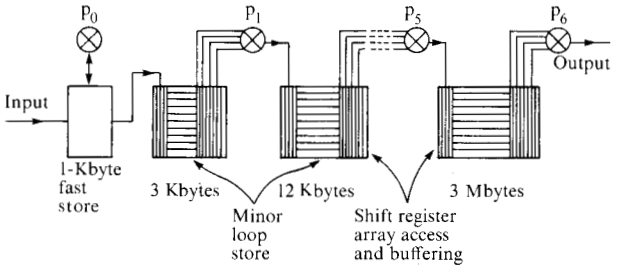

Figure 9 A seven-processor hybrid technology sorter for files up to 4 Mbytes. A four-way merge is used.

9. S. J. P. Todd, "Major Minor Loops Accessed via Shift Register Arrays," *IBM United Kingdom Scientific Centre Technical Note 67* (in preparation).
10. C. K. Wong and P. C. Yue, "Data Organization in Magnetic Bubble Lattice Files," *IBM J. Res. Develop.* **20,** 576 (1976).

Received April 1, 1977; revised November 2, 1977

The author is located at the IBM United Kingdom Scientific Centre, Neville Road, Peterlee, Durham SR8 1BY, England.