BIT 23 (1983), 170–180

LINEAR SORTING WITH $O(\log n)$ PROCESSORS

J. A. ORENSTEIN *, T. H. MERRETT and L. DEVROYE

School of Computer Science, McGill University, 805 Sherbrooke Street West, Montréal, PQ, Canada H3A 2K6

Abstract.

A file of n records can be sorted in linear time given $O(\log(n))$ processors. Four such algorithms are presented and analyzed. All of them have reasonable hardware requirements; no memory access conflicts are generated; a constant number of communication lines per processor are needed (except for one case); and the space requirements are $O(n)$ or $O(n \log(\log(n)))$.

1. Introduction.

It is well known that sorting requires $\Omega(n \log n)$ comparisons. The time requirement for a single processor is therefore $\Omega(n \log n)$. If parallel processors are used then the time requirement can be reduced. A realistic model of parallel computation has the following properties:

- The number of processors is sub-linear.
- The number of communication lines attached to each processor is sub-linear (and preferably constant).
- Input to and output from the parallel processor is sequential. (This implies that any sorting algorithm must take $\Omega(n)$ time.)
- Memory access conflicts do not occur.

Todd (1978) describes "parallel mergesort" which can be implemented with $O(\log(n))$ processors and $O(n)$ memory to sort n objects in $O(n)$ time without memory conflicts. Other parallel sorting algorithms (Batcher, 1968; Hirschberg, 1978; Preparata, 1978) violate one or more of the properties listed above. In this paper we present four algorithms that can sort n records in $O(n)$ time (as measured by at least one of two criteria) given $O(\log(n))$ processors.

We will measure the running time of each algorithm by two criteria:

1. The required number of "effective passes" of the file. An *effective pass* (abbreviated to "ep") is the time required to read (or write) the file using a single processor. This measurement is of interest when the file resides on secondary storage.
2. The number of parallel comparisons. Each processor is either idle or

Received May 24, 1982. Revised December 17, 1982.

* Current address: COINS Department, University of Massachusetts, Amherst, MA 01003, U.S.A.

performs comparisons, (one comparison per time unit); all the other operations take 0 time units. The number of parallel comparisons is defined to be the number of time units required to sort the file.

In some cases, either the number of elements in each processor is random or the time needed by each processor is random. Because one usually waits until the slowest processor has finished executing before going on to the next phase of the algorithm, the expected time taken by the algorithm can be written as the expected value of a maximum of some random variables. In the appendix, we give some useful probability theoretical machinery for the study of such quantities (we are mainly interested, of course, in upper bounds). The tools given there are applied to several of our algorithms.

We will also discuss the memory requirements of each algorithm.

2. Practical parallel sorting algorithms.

We will use the following conventions:

- All logarithms are base 2.
- The number of records being sorted, n, is a power of 2. The modifications of the algorithms to handle other values of n are trivial (or we give the modification).

2.1. *Parallel bucketsort.*

Bucketsort, a variant of distributive partitioning (Dobosiewicz, 1978), is especially suitable for implementation on a parallel processor. Using an order preserving key-to-address function (Sorensen, Tremblay and Deutscher, 1978) the records can be distributed among m buckets, $B_1, B_2, \ldots, B_m$, so that each record in B_i precedes each record in B_j if $i < j$. Each bucket is sorted by one processor using an $O(n \log n)$ method. The concatenation of the results, $B_1 B_2 \ldots B_m$ is the sorted output.

The processors are arranged serially for parallel bucketsort. P_0 stores the input file and receives the result, while P_i, $i = 1, 2, \ldots, m$, is the processor assigned to B_i. There is a two-way communication line between P_i and P_{i+1}, $i = 0, 1, \ldots, m-1$. During input, each record travels from P_0 towards P_m until its destination is reached. After the records of B_i have been sorted by P_i the records travel in order back to P_0.

2.2. *Merge of sorted subfiles.*

The worst case for parallel bucketsort occurs when the records are all placed in the same bucket. This can be avoided if $\lfloor n/m \rfloor$ or $\lceil n/m \rceil$ records are assigned to each processor in an arbitrary fashion. A merge of the sorted subfiles

(abbreviated to MOSS) will be required; concatenation will no longer be sufficient due to the arbitrary distribution of the records. This merge can be done by a single processor with m input lines, one from each of the processors, $P_1, P_2, \ldots, P_m$.

2.3. *Another version of MOSS.*

MOSS, as presented in section 2.2, requires the use of one processor with m input lines. This processor performs the merge. If we take $m = \log n$, (which will be shown to be a reasonable choice in section 3.1), the hardware requirement is not as realistic as for parallel bucketsort. However, the merge can also be performed by a tree of processors.

Each component processor has two input lines. The m processors which sort the subfiles are the leaves of the tree. Pairs of leaves pass their output to a parent processor which will produce one run of length $\leqq \lceil 2n/m \rceil$. This run in turn gets passed (with another run) to another processor. Eventually, the root will receive two runs and merge them to produce the sorted file. This tree of processors has $\lceil \log(m) \rceil + 1$ levels and the entire machine uses $2m - 1 = O(\log n)$ processors. (If m is not a power of 2 then some processors in the tree are idle).

We will refer to this version of MOSS as MOSS($\log n$) and to the previous version as MOSS(1). The subscript indicates the number of processors used in the merge.

2.4. *Parallel sorting using splitting.*

Another parallel sorting algorithm can be implemented using the same architecture as for MOSS($\log n$). The file will pass from the root to the leaves. We assume again that $m = \log n$. Let the levels of the tree be labelled $0, 1, \ldots, \lceil \log m \rceil$ from the root to the leaves. The algorithm for a processor on level $i, i = 0, 1, \ldots, \lceil \log m \rceil - 1$, is as follows:

> Wait for the parent (on level $i-1$) to supply an unsorted subfile of n_i records.
> Find M, the median record.
> For each record, r:
> Send r to the left child if $r \leqq M$;
> Send r to the right child if $r > M$.

The leaf processors receive subfiles of no more than $\lceil n/m \rceil$ records each. These subfiles are sorted in parallel. One-way communication lines linking the leaves are used (as in parallel bucketsort) to return the output.

3. Time requirement – number of *I/O* operations.

In sorting large files of data resident on secondary storage, it is customary to measure the running time in terms of the number of transfers between primary and secondary memory. We will, for each algorithm, count the number of effective passes required (as defined in section 1).

3.1. *Parallel bucketsort.*

If the records are distributed uniformly (deterministically) across the buckets and if $m = \log n$, then parallel bucketsort runs in less than 4 eps. Input and output take 1 ep each and each processor can sort its records in no more than 2 eps.

Suppose that each processor sorts its records using mergesort, which requires one read and one write per comparison. Then each processor takes, (assuming that n/m is an integer)

$$T(n/m) = \frac{2\log(n/m)}{m} \text{ eps},$$

where $T(N)$ is the time required to sort N records. For $m = \log n$

$$T(n/m) = \frac{2[\log n - \log\log n]}{\log n} \text{ eps} < 2 \text{ eps}.$$

For random distributions across the buckets, the analysis of section 4.1 is applicable. It says that the expected running time of the sort step is $O(1)$ eps in most cases.

In the worst case, one processor received all the records. The running time is then $O(\log n)$ eps.

3.2. *MOSS*(1).

MOSS(1) always runs in 4 eps. Input and the sorting of the subfiles take 1 and 2 eps respectively. The merge step costs 1 ep. In practice the steps of the m-way merge would be overlapped with the output operations. (We discuss the cost of merging in section 4.2.).

3.3. *MOSS(log n).*

As above, the leaf processors require 3 eps to read and sort their subfiles.

For the analysis of the merging time, suppose that n/m is an integer. The tree has $\lceil \log m \rceil + 1$ levels labelled $0, 1, \ldots, \lceil \log m \rceil$ from the leaves to the root. A processor at level 1 reads 2 records (simultaneously) at time 0 and produces a run of length $2n/m$ from time 1 through time $2n/m$. A processor on level 2

can start reading after its children (on level 1) have started writing. Therefore, it starts reading at time 2 and writes from time 3 to time $4n/m+2$. In general, a processor at level i writes from time $2i-1$ to time $2^i n/m+2(i-1)$. The last processor, at level $\log m$, finishes writing at time

$$T(n) = n+2(\log m-1).$$

Thus merging takes about 1 ep.

3.4. *Parallel sorting using splitting.*

Input, output and the sorting done by the leaf processors take 4 eps. Unlike MOSS($\log n$), a processor must wait for its supplier to finish working before it can start. Thus, the time required to send a subfile to a child processor is significant. A processor on level i has to send $n/2^{i+1}$ records to each child. The total communication time is

$$C(n) = \sum_{i=0}^{\log n-1} n/2^{i+1} = n-1 \simeq 1 \text{ ep.}$$

So this algorithm takes 5 eps.

4. Time requirement number of comparisons.

In this section we consider the number of parallel comparisons required by each algorithm. Since each algorithm uses $O(\log n)$ processors, no more than $O(\log n)$ comparisons can occur simultaneously.

4.1. *Parallel bucketsort.*

Let n_i denote the number of records received by $P_i, i = 1,\ldots,m$. Then the running time of parallel bucketsort is

$$T(n) = \max_i (O(n_i \log n_i)).$$

In the worst case $T(n) = O(n \log n)$.

For the average time taken by parallel bucketsort, we assume the convenient model from Devroye and Klincsek (1981): assume that the data consist of independent identically distributed random variables with density f on $[0,1]$, and that the m buckets are defined by the intervals $[(i-1)/m,\ i/m)$ where $m \sim \log n$ and $1 \leqq i \leqq m$. The number of points in the ith bucket is N_i. If $T(n)$

is the average time taken by parallel bucketsort and $X_n = \max_i(N_i \log(N_i+1))$ then $T(n) = \Omega(E_n))$ and $T(n) = O(E(X_n))$ where $E(.)$ denotes "expected value".*)

THEOREM 1. *In any case,* $T(n) = \Omega(n)$. *Also,* $T(n) = O(n)$ *if and only if* f *has a bounded version, i.e. a version such that* $\sup_x f(x) < \infty$.

The proof of Theorem 1 appears in the Appendix. We point out that in this model we count truncation as a constant-time operation. The use of parallel bucketsort for internal sorting is somewhat limited because linear average running times are achieved without the parallelism (Dobosiewicz, 1978; Devroye and Klincsek, 1981) and for a broader class of densities than those given in Theorem 1. Finally we note that Theorem 1 remains valid if we first find the minimal and maximal elements in the data and then divide the obtained sub-interval of $[0, 1]$ into m equal buckets.

4.2. *MOSS*(1).

Each processor has no more than $\lceil n/m \rceil$ records where $m = \log n$ is the number of processors. The sorting of subfiles costs

$$T\left(\frac{n}{m}\right) = O\left(\frac{n}{\log n} \log\left(\frac{n}{\log n}\right)\right) = O(n)$$

time units. The merge of the sorted sub-files is of degree m. Using a heap, the merge requires $\sim n \log m$ comparisons. Thus the time complexity of MOSS(1) is $O(n \log\log n)$ which is suboptimal.

4.3. *MOSS*($\log n$).

If the merge at the end of MOSS is performed by a tree of processors then the time complexity is $O(n)$: the processors at level i take time $cn/2^i$ for some constant c, and

$$\sum_{i=0}^{\log n - 1} cn/2^i = O(n).$$

This very simple analysis does not apply when instead of an $O(n \log n)$ worst case sorting algorithm for the sub-files (of size approximately n/m), we use, say, quicksort. In the worst case, the algorithm will require time proportional to $n^2/m^2 \sim n^2/\log^2 n$. We could ask whether it is still true that the average time is $O(n)$. Since the processors at the bottom level take different times, say, $T_1, \ldots, T_m$,

*) $f(x) = \Omega(g(x))$ means that there exists a constant C such that $f(x) \geq Cg(x)$ for all x.

the overall time of the algorithm is bounded from below by $\max_i(T_i)$, and from above by $\max_i(T_i)+O(n)$. This observation allows us to prove (see Appendix)

THEOREM 2. *If quicksort is used to sort the m sub-files in the MOSS*$(\log n)$ *algorithm, then the average time taken by the algorithm is* $O(n)$.

4.4. *Parallel sorting using splitting.*

The running time of this algorithm is determined by the time required to find the median of a sub-file.

Linear time median finding algorithms (Blum et al., 1973; Schonhage et al., 1976; see also Knuth, 1975) can be used to split sets in two. In that case the total time taken by the algorithm is

$$O\left(\sum_{i=0}^{\log n-1} n/2^i\right) = O(n).$$

In practice, these algorithms are notoriously slow compared to the algorithms of Hoare (1961) (see also Aho, Hopcroft and Ullman (1974)) and Floyd and Rivest (1975) which are linear on the average but super-linear in the worst case. Furthermore, it is not necessary to achieve an exact split in each processor; an approximate split will suffice. Assume for example that we use Hoare's simple bounded workspace algorithm FIND (Hoare, 1961) to split a set of n elements into two sets of sizes n_1 and n_2 where $|n_1-n_2| \leqq c$ for some constant c. All the elements in the first set are smaller in value than those in the second set. If the same c is used regardless of the value of n, we see that at level 1, the sets are of size between $n/2-c$ and $n/2+c$, and at level i, all sets are of size between $n/2^i-2c$ and $n/2^i+2c$. In particular, since there are $\lceil \log m \rceil$ levels, we see that at the bottom level ($\lceil \log m \rceil$), each processor has at most $n/m+2c$ elements. If $c = O(n/\log n)$ then we have $O(n/\log n)$ elements in the processors at the bottom level and these can be sorted in time $O(n)$ in parallel by heapsort, and in average time $O(n)$ in parallel by quicksort (Theorem 2).

We have to establish that the average time taken by the splitting stage of the algorithm is $O(n)$:

THEOREM 3. *If parallel sorting with splitting is implemented in such a way that* (1) *algorithm FIND is used to split any set of size* n_0 *into two sets of sizes contained in* $[n_0/2-c, n_0/2+c]$ *where* $c = O(n/\log n)$; *and* (2) *an* $O(n \log n)$ *average time comparison-based sorting algorithm is used to sort the sub-files at the bottom level of the tree, then the average time taken by the entire algorithm is* $O(n)$.

The proof of Theorem 3 appears in the Appendix.

5. Memory requirements.

For parallel bucketsort a processor requires $O(n)$ storage in the worst case. Thus the total memory requirement for parallel bucketsort is $O(n \log n)$.

The memory requirement for MOSS(1) is $O(n)$ since each of the m processors sorts a sub-file of about n/m records.

The memory requirement of MOSS($\log n$) and of parallel sorting using splitting is $O(n \log \log n)$ since there must be room for the entire file at every level of the tree of processors.

6. Summary.

The average case results of sections 3, 4 and 5 are summarized below.

	Time (eps)	Time (parallel comparisons)	Space
Parallel bucketsort	4*	$O(n)$*	$O(n)$*
MOSS(1)	4	$O(n \log \log n)$	$O(n)$
MOSS(log(n))	4	$O(n)$	$O(n \log \log n)$
Parallel sorting using splitting	5*	$O(n)$*	$O(n \log \log n)$

Appendix.

We will repeatedly use the following lemma:

LEMMA 1. *For any collection $X_1, \ldots, X_n$ of positive random variables, and for all constants $r \geqq 1$,*

$$E(\max_i X_i) \leqq n^{1/r} \max_i E^{1/r}(X_i^r).$$

PROOF. By Jensen's inequality,

$$E^r(\max X_i) \leqq E(\max X_i^r) \leqq E\left(\sum_{i=1}^{n} X_i^r\right) \leqq n \max E(X_i^r).$$

PROOF OF THEOREM 1.

Assume first that f is bounded by c. Since $T(n) = O(n + E(X_n))$, it suffices to consider $E(X_n)$. By Lemma 1,

* See text for more details.

$$E(X_n) \leqq \log(n+1)\, E(\max_i N_i)$$

$$\leqq \log(n+1)(\max_i E(N_i) + E(\max_i (N_i - E(N_i))))$$

$$\leqq \log(n+1)(\sqrt{m}\max_i \sqrt{\mathrm{Var}(N_i)} + \max_i E(N_i)).\ 1 \leqq i \leqq m.$$

Since $E(N_i) = np_i \leqq nc/m$ (where p_i is the integral of f over $[(i-1)/m, i/m)$), and $\mathrm{Var}(N_i) = np_i(1-p_i) \leqq nc/m$, we have

$$E(X_n) \leqq \log(n+1)(\sqrt{(nc)} + nc/m) \sim nc$$

sine $m \sim \log n$. Thus, $T(n) = O(n)$.

Next, we show that when f is unbounded, then $\lim_{n\to\infty} \inf T(n)/n = \infty$. First, we note that $T(n) = \Omega(n + E(X_n))$ and that $E(X_n) \geqq \max_{1 \leqq i \leqq m} E(N_i)\log(E(N_i)+1)$ (this follows from Jensen's inequality). Let $N = \max_{1 \leqq i \leqq m} E(N_i)$. We will first prove that $\lim_{n\to\infty} \inf NM/n = \infty$. Choose a positive number M and let A be the set of all x with $f(x) \geqq M$. If λ is Lebesgue measure, then

$$Nm/n = m \max_{1 \leqq i \leqq m} P_i \geqq mM \max_{1 \leqq i \leqq m} \lambda\left(A\left[\frac{i-1}{m}, \frac{i}{m}\right)\right)$$

$$\geqq mM \sup_{x \in A} \min\left(\lambda\left(A\left[x - \frac{1}{2m}, x\right)\right), \lambda\left(A\left[x, x + \frac{1}{2m}\right)\right)\right)$$

$$\geqq mM\lambda^{-1}(A) \int_A \min\left(\lambda\left(A\left[x - \frac{1}{2m}, x\right)\right), \lambda\left(A\left[x, x + \frac{1}{2m}\right)\right)\right) dx.$$

By the Lebesgue density theorem, for almost all $x \in A$, $2m\lambda(A[x - 1/(2m), x)) \to 1$ as $m \to \infty$ (Wheeden and Zygmund, 1977). Thus, by Fatou's Lemma,

$$\liminf_{n\to\infty} Nm/n \geqq \frac{M/2}{\lambda(A)}\lambda(A) = \frac{M}{2}.$$

Since M was arbitrary, the limit infimum must be ∞. We conclude the proof by noting that

$$E(X_n)/n \geqq (Nm/n)\frac{1}{m}\log\left(1 + \frac{n}{m}\right) = \frac{Nm}{n}(1 + o(1)) \to \infty.$$

PROOF OF THEOREM 2.

We need only show that $E(\max_{1 \leqq i \leqq m} T_i) = O(n)$. By Lemma 1 and the fact that quicksort, when used on a set of n elements, takes time T where $E(T) \sim c_1 n \log n$,

$\mathrm{Var}(T) \sim c_2 n^2$ for some $c_1, c_2 > 0$ (see Sedgewick (1977) and Knuth (1975, pp. 121–122); for the original version of quicksort, see Hoare (1962)), we have

$$E(\max_{1 \leqq i \leqq m} T_i) \leqq \max_{1 \leqq i \leqq m} E(T_i) + \sqrt{m} \max_{1 \leqq i \leqq m} \sqrt{\mathrm{Var}(T_i)}$$

$$\sim c_1 \frac{n}{m} \log\left(\frac{n}{m}\right) + \sqrt{\left(mc_2 \left(\frac{n}{m}\right)^2\right)} \sim c_1 n + n\sqrt{(c_2/m)} \sim c_1 n.$$

Here we used the facts that $m \sim \log n$ and that each processor at the bottom level has $\sim n/m$ elements.

PROOF OF THEOREM 3.

The average time taken by the splits while we move down the tree is the sum of the average times taken at level i, $0 \leqq i \leqq \overline{\log m}$. At level i, we have 2^i processors each working on a set of at most $n/2^i + 2c$ elements, and the execution times are $T_1, \ldots, T_{2^i}$. Thus, at level i, we take time bounded by

$$\max_{1 \leqq j \leqq 2^i} T_j$$

and average time

$$E(\max_{1 \leqq j \leqq 2^i} T_j) \leqq 2^{i/2} \max_{1 \leqq j \leqq 2^i} \sqrt{E(T_j^2)} \leqq 2^i a(n/2^i + 2c) \leqq bn/2^{i/2}$$

for some universal positive constants a, b. The bound on $E(T_j^2)$ is provided by a result of Devroye (1982) on the moments of the running time of Hoare's algorithm FIND (Hoare, 1961). Summing with respect to i gives the bound

$$bn \sum_{i \geqq 0} 2^{-i/2} = bn/(1 - 1/\sqrt{2}) = O(n).$$

This concludes the proof of Theorem 3.

References.

1. A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. K. E. Batcher, *Sorting networks and their applications*, Proceedings of the AFIPS Spring Joint Conference 32, pp. 307–314, 1968.
3. M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest and R. E. Tarjan, *Time bounds for selection*, Journal of Computer and System Sciences 7, pp. 448–461, 1973.

4. L. Devroye, *Exponential bounds for the running time of a selection algorithm*, Manuscript, McGill University, Montreal, 1982.
5. L. Devroye and T. Klincsek, *Average time behaviour of distributive sorting algorithms*, Computing 26, pp. 107, 1981.
6. W. Dobosiewicz, *Sorting by distributive partitioning*, Information Processing Letters 7, pp. 1–6, 1978.
7. R. W. Floyd and R. L. Rivest, *Expected time bounds for selection*, Communications of the ACM 18, pp. 165–173, 1975.
8. D. S. Hirschberg, *Fast parallel sorting algorithms*, Communications of the ACM 21, pp. 657–661, 1978.
9. C. A. R. Hoare, *Quicksort*, Computer Journal 5, pp. 10–15, 1962.
10. C. A. R. Hoare, *FIND* (Algorithm 65), Communications of the ACM 4, pp. 321–322, 1961.
11. D. E. Knuth, *The Art of Computer Programming, Vol. 3. Sorting and Searching*, Addison-Wesley, Reading, Mass., 1975.
12. F. P. Preparata, *New parallel-sorting schemes*, IEEE Transactions on Computers, C-27, pp. 669–673, 1978.
13. A. Schonhage, M. Paterson and N. Pippenger, *Finding the median*, Journal of Computer and System Sciences 13, pp. 184–199, 1976.
14. R. Sedgewick, *The analysis of quicksort programs*, Acta Informatica 7, pp. 327–355, 1977.
15. P. G. Sorenson, J. P. Tremblay and R. F. Deutscher, *Key-to-address transformation techniques*, INFOR 16, pp. 1–34, 1978.
16. S. Todd, *Algorithm and hardware for a mergesort using multiple processors*, IBM Journal of Research and Development 22, pp. 509–517, 1978.
17. R. L. Wheeden and A. Zygmund, *Measure and Integral*, Marcel Dekker, New York, 1977.