

**CSL** *COORDINATED SCIENCE LABORATORY*

*APPLIED COMPUTATION THEORY GROUP*

# **NEW PARALLEL SORTING SCHEMES**

F. P. PREPARATA

REPORT R-782

UILU-ENG 77-2229

**UNIVERSITY OF ILLINOIS – URBANA, ILLINOIS**

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle)  NEW PARALLEL SORTING SCHEMES		5. TYPE OF REPORT & PERIOD COVERED  Technical Report
7. AUTHOR(s)  F. P. Preparata		6. PERFORMING ORG. REPORT NUMBER R-782; UILU-ENG 77-2229
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		8. CONTRACT OR GRANT NUMBER(s) MCS-76-17321 DAAB-07-72-C-0259
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE July, 1977
		13. NUMBER OF PAGES 15
		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report)  Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Parallel Computation Computational Complexity Design of Algorithm Sorting Enumeration Sorting		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) In this paper we describe a family of parallel sorting algorithms for a multi-processor system. These algorithms are enumeration sorts and comprise the following phases: (i) count acquisition: the keys are subdivided into subsets and for each key we determine the number of smaller keys (count) in every subset; (ii) rank determination: the rank of a key is the sum of the previously obtained counts; (iii) data rearrangement: each key is placed in the position specified by its rank. The basic novelty of the algorithms is the use of parallel merging to implement count acquisition. By using Valiant's merging		



## 20. ABSTRACT (continued)

scheme, we show that  $n$  keys can be sorted in parallel with  $n \log_2 n$  processors in time  $C \log_2 n + o(\log_2 n)$ ; in addition, if memory fetch conflicts are not allowed, using a modified version of Batcher's merging algorithm to implement phase (i) we show that  $n$  keys can be sorted with  $n^{1+\alpha}$  processors in time  $(C'/\alpha) \log_2 n + o(\log_2 n)$  thereby matching the performance of Hirschberg's algorithm, which, however, is not free of fetch conflicts.

UILU-ENG 77-2229

NEW PARALLEL SORTING SCHEMES

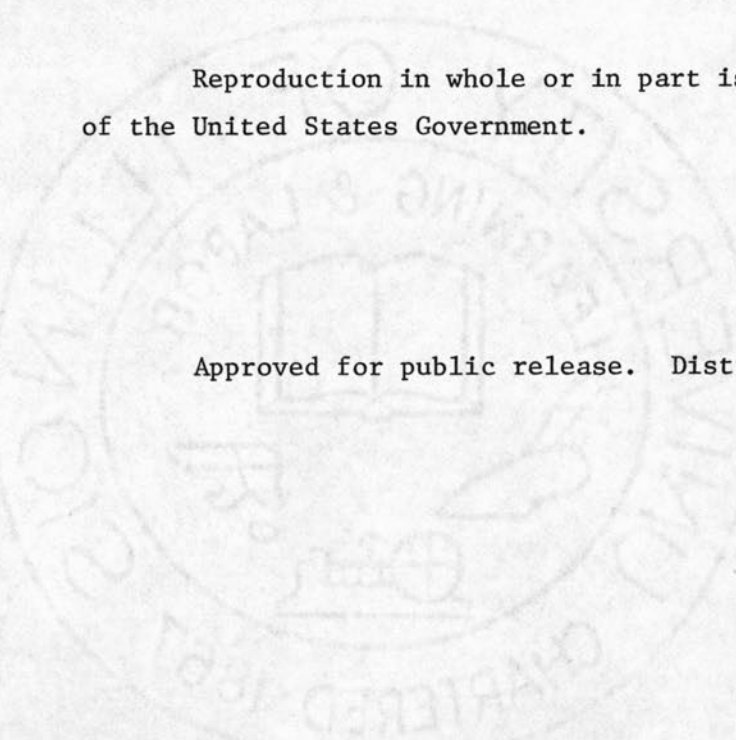
by

F. P. Preparata

This work was supported in part by the National Science Foundation under Grant NSF MCS-76-17321 and in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

Reproduction in whole or in part is permitted for any purpose of the United States Government.

Approved for public release. Distribution unlimited.



## NEW PARALLEL SORTING SCHEMES

F. P. Preparata\*, Senior Member IEEE

University of Illinois at Urbana-Champaign

### Abstract

In this paper we describe a family of parallel sorting algorithms for a multiprocessor system. These algorithms are enumeration sortings and comprise the following phases: (i) count acquisition: the keys are subdivided into subsets and for each key we determine the number of smaller keys (count) in every subset; (ii) rank determination: the rank of a key is the sum of the previously obtained counts; (iii) data rearrangement: each key is placed in the position specified by its rank. The basic novelty of the algorithms is the use of parallel merging to implement count acquisition. By using Valiant's merging scheme, we show that  $n$  keys can be sorted in parallel with  $n \log_2 n$  processors in time  $C \log_2 n + o(\log_2 n)$ ; in addition, if memory fetch conflicts are not allowed, using a modified version of Batcher's merging algorithm to implement phase (i), we show that  $n$  keys can be sorted with  $n^{1+\alpha}$  processors in time  $(C'/\alpha) \log_2 n + o(\log_2 n)$ ; thereby matching the performance of Hirschberg's algorithm, which, however, is not free of fetch conflicts.

---

\* Coordinated Science Laboratory, Department of Electrical Engineering, and Department of Computer Science, University of Illinois, Urbana, IL. 61801  
This work was supported in part by the National Science Foundation under Grant MCS76-17321 and in part by the Joint Services Electronics Program under Contract DAAB-07-72-C-0259.



## NEW PARALLEL SORTING SCHEMES

F. P. Preparata

### 1. Introduction

The efficient implementation of comparison problems, such as merging, sorting, and selection, by means of multiprocessor computing systems has attracted considerable attention in recent years. One of the earliest fundamental results is due to K. E. Batcher [1], who proposed a sorting network consisting of comparators and based on the principle of iterated merging; as is well-known, such scheme sorts  $n$  keys with  $O(n(\log n)^2)$  comparators in time  $O((\log n)^2)$ .<sup>(1)</sup> Batcher's network is readily interpreted, in a more general framework, as a system of  $n/2$  processors with access to a common data memory of  $n$  cells: obviously, the network structure induces a nonadaptive schedule of memory accesses. After the appearance of Batcher's paper, substantial work was aimed at filling the gap between the upper-bound  $O((\log n)^2)$  on the number of steps which is achievable by a network of comparators and the lower-bound  $O(\log n)$ ; the lack of success, however, convinced several workers to look for more flexible forms of parallelism.

The first scheme shown to sort  $n$  keys in time  $O(\log n)$  is due to D. E. Muller and F. P. Preparata [2], but it requires a discouraging number of  $O(n^2)$  processors. Subsequently, new results were obtained on parallel merging by F. Gavril [3], L. G. Valiant [4] must be credited with addressing the fundamental question of the intrinsic parallelism of some

---

This work was supported in part by the National Science Foundation under Grant MCS76-17321 and in part by the Joint Services Electronics Program under Contract DAAB-07-72-C-0259.

(1)  $\log$  means logarithm to the base 2.

comparison problems and with the development of faster algorithms than were previously known. In particular, in [4] he described an algorithm for merging with  $\sqrt{nm}$  processors two sorted sequences of  $n$  and  $m$  keys, respectively, ( $n \leq m$ ), in  $2\log\log n + O(1)$  <sup>(1)</sup> comparison steps; this algorithm can then be applied to sort  $n$  keys with  $n$  processors in  $2\log n \cdot \log\log n + O(\log n)$  steps. His method assumes a computational model in which there is no penalty for memory-processor alignment and the overhead corresponding to the reassignment of sets of processors to subsequences to be merged, is ignored.

A new family of sorting algorithms has been recently discovered by D. Hirschberg [5]. Assuming as a computation model a parallel processing system of the SIMD type (single-instruction stream, multiple-data stream) with random access capabilities to a common memory, Hirschberg shows that  $n$  keys can be sorted in time  $O(k \log n)$  with  $n^{1+1/k}$  processors, where  $k$  is an arbitrary integer  $\geq 2$ . These schemes are not free of memory fetch conflicts (simultaneous reading of the same location by more than one processor) and Hirschberg poses as an open question the possibility of achieving analogous performances without memory fetch conflicts.

In this paper we shall present two results. The first, discussed in Section 2, is an algorithm for sorting  $n$  keys in time  $O(\log n)$  with  $n \log n$  processors: this algorithm combines a number of known techniques, and makes crucial use of Valiant's merging algorithm. The second result (Section 3) is a family of very simple sorting algorithms, which have the same running time as Hirschberg's, but use basically, but different techniques and are entirely free of memory fetch conflicts. As our computation model we adopt a system of several identical

---

(1) Throughout this paper "log" means "logarithm to the base 2."

processors, each capable of random-accessing a common memory with no alignment penalty. Store, fetch, and arithmetic operations have unit costs, and fetch conflicts are disallowed when appropriate.

All of the algorithms described in this paper - as well as Hirschberg's [5] - are instances of enumeration sorting, in Knuth's terminology ([6], p. 73). In these methods each key is compared with all the others and the number of smaller keys determines the given key's final position. Specifically, three distinct tasks are clearly identifiable in enumeration sorting algorithms:

- (i) count acquisition. The set of keys is partitioned into subsets and for each key we determine the number of smaller keys in each subset (this informal description momentarily assumes that all keys are distinct);
- (ii) rank computation. For each key the sum of the counts obtained in (i) gives the final position (rank) of that key in the sorted sequence;
- (iii) data rearrangement. Each key is placed in its final position according to its rank.

Less informally, an enumeration sorting scheme has the following format, where we assume for simplicity that, for some given integer  $r$ ,  $n = kr$ . Data structures to be used are arrays of keys. By  $A[i:j]$  we denote a sequence  $A[i]A[i+1] \dots A[j]$ .

Input:  $A[0:n-1]$ , the array of the keys to be sorted, integer  $r$

Output:  $A[0:n-1]$ , the array of the sorted keys.



1. begin Define  $A_i[0:r-1] \leftarrow A[ir:(i+1)r-1]$ , for  $i=0, \dots, k-1$ .
  2.  $c_\ell^{(ij)} \leftarrow \begin{cases} |\{A_j(h) \mid A_j[h] \leq A_i[\ell]\}| & \text{for } j < i \\ |\{A_j(h) \mid A_j[h] < A_i[\ell]\}| & \text{for } j > i \end{cases}$
  3.  $C_\ell^{(ii)} \leftarrow |\{A_i[h] \mid A_i[h] \leq A_i[\ell], h < \ell\} \cup \{A_i[h] \mid A_i[h] < A_i[\ell], h > \ell\}|$
  3.  $\text{rank}(A_i[\ell]) \leftarrow \sum_{j=0}^{k-1} c_\ell^{(ij)}$
  4.  $A[\text{rank}(A_i[\ell])] \leftarrow A_i[\ell]$
- end

Note that count acquisition, rank computation, and data rearrangement are performed, respectively, in steps 2, 3, and 4. Also, the algorithm must insure that all ranks be distinct, which is a crucial condition for the data rearrangement task (otherwise memory store conflicts would occur). This clearly poses no problem when the keys are all distinct. In the opposite case, some convention must be adopted for the ordering of sets of identical keys. One such convention is that sorting be stable (see [6], p. 4), that is, the initial order of identical keys is preserved in the sorted array. Thus, all of our sorting schemes will be stable. This is reflected in the rules for the computation of the parameters  $C_\ell^{(ij)}$  in Step 2 of the above algorithm.

The simple algorithm proposed by Muller and Preparatá in [2] is a crude example of enumeration sorting, in which the sets  $A_i$  are chosen to be singletons. With this choice, each key is compared with every other key, thereby using  $O(n^2)$  processors; similarly, rank computation uses  $O(n^2)$  processors, since  $O(n)$  processors are assigned to each key. The time bound  $O(\log n)$  is due to Step 3 (counting in parallel the number of 1's in a set of  $n$  binary digits), whereas Steps 2 and 4 run in constant time in our present model.

In the more complex procedures to be later described, the operations of rank computation and data rearrangement are essentially carried out as in the basic scheme described above. The main difference occurs with regard to count acquisition. In the Muller-Preparata method the counts are acquired by comparing each key with every other. The comparison of two keys  $A[i]$  and  $A[j]$  could be viewed as merging  $A[i]$  and  $A[j]$ . If rather than dealing with single keys we now deal with sorted sequences of keys  $A_i[0:r-1]$  and  $A_j[0:r-1]$ , where  $r > 1$  and, say,  $j < i$ , then the number of keys in  $A_j[0:r-1]$  which are no greater than  $A_i[\ell]$  ( $\ell=0, \dots, r-1$ ) as well as the number of keys in  $A_i[0:r-1]$  which are less than  $A_j[h]$  ( $h=0, \dots, r-1$ ), can be obtained by merging the two sequences  $A_i[0:r-1]$  and  $A_j[0:r-1]$ . In fact, let  $B[0:2r-1]$  be the array obtained by merging the two sorted arrays  $A_j[0:r-1]$  and  $A_i[0:r-1]$  with the ordering convention  $A_k[s] \leq A_k[s+1]$  ( $k=i, j$ ) and  $B[s] \leq B[s+1]$ . Suppose also that the merging be stable, that is, the order of identical keys in the concatenated array  $A_j[0:r-1]A_i[0:r-1]$  is preserved in  $B[0:2r-1]$ . If  $B[q] = A_i[\ell]$ , then there are  $(q-\ell)$  entries of  $A_j[0:r-1]$  in  $B[0:q-1]$  which are no greater than  $A_i[\ell]$ ; similarly if  $B[q] = A_j[h]$ , then there are  $(q-h)$  entries of  $A_i[0:r-1]$  in  $B[0:q-1]$  which are strictly less than  $A_j[h]$ . This is the central idea of the algorithms to be described.

## 2. A fast parallel sorting algorithm

In this section we assume that in our computational model memory fetch conflicts are permitted. To provide the feature required by Valiant's merging algorithm, that a key be simultaneously compared with several other keys, we may assume that the processors have broadcast capabilities. The only overhead we shall neglect is the reassignment of processors to the operation of merging pairs of subsequences, as occurs in Valiant's method [4].

Notice that this model of parallel computation coincides with that required by Valiant's merging algorithm.

We assume inductively that the following algorithm, SORT1, for  $p < n$  uses at most  $\lfloor p \log p \rfloor$  processors to sort  $p$  keys. Since SORT1 is recursive, the following presentation constitutes a constructive extension of the inductive step to the integer  $n$ . The induction can be started with  $n \geq 4$ .

#### Algorithm SORT1

begin

1.  $k \leftarrow \lceil \log n \rceil$ ,  $r \leftarrow \lfloor n / \lceil \log n \rceil \rfloor$
2. Define arrays  $S[0:k;0:k;0:2r-1]$  and  $R[0:k;0:k;0:r-1]$  (three-dimensional arrays) and  $A_i[0:r-1] \leftarrow A[ir:(i+1)r-1]$  ( $i=0, \dots, k-1$ ),  $A_k[0:n-kr-1] \leftarrow A[kr:n-1]$  (for  $n > kr$ ).

Comment: When  $n=kr$ , array  $A_k$  is obviously vacuous. Array  $S$  is defined for simplicity as having  $(k+1)^2 2r$  cells, although the algorithm will only make use of the cells  $S[i;j;q]$  for which  $i < j$ .

3.  $A_i[0:r-1] \leftarrow \text{SORT1}(A_i[0:r-1])$  ( $i=0, \dots, k-1$ )  
 $A_k[0:n-kr-1] \leftarrow \text{SORT1}(A_k[0:n-kr-1])$ .

Comment: This step is a parallel recursive call of SORT1 and it involves sorting in parallel  $k$  sets of  $r$  keys each and, possibly, one set of  $(n-kr)$  keys. By the inductive hypothesis, it uses at most  $k \lfloor r \log r \rfloor + \lfloor (n-kr) \log(n-kr) \rfloor \stackrel{\Delta}{=} N$  processors. Since  $n-kr < \lceil \log n \rceil$ , the number of processors used is less than  $\lceil \log n \rceil \cdot \lfloor \lfloor n / \lceil \log n \rceil \rfloor \cdot \log \lfloor n / \lceil \log n \rceil \rfloor \rfloor + \lfloor \lceil \log n \rceil \log \lceil \log n \rceil \rfloor$   
 $\leq n \log(n / \lceil \log n \rceil) + \lceil \log n \rceil \log \lceil \log n \rceil$   
 $= n \log n - \log \lceil \log n \rceil (n - \lceil \log n \rceil) \leq n \log n - 1$   
 $\leq \lfloor n \log n \rfloor$ , for  $n \geq 3$ . For the sake of uniformity, array  $A_k$  is now extended to size  $r$ , where each cell of  $A_k[n-kr:r-1]$  is filled with a dummy sentinel larger than any key.



$$4. \quad S[i;j;0:r-1] \leftarrow A_i[0:r-1] \quad (i=0, \dots, k-1; j=i+1, \dots, k)$$

$$S[i;j;r:2r-1] \leftarrow A_j[0:r-1] \quad (i=0, \dots, j-1; j=1, \dots, k)$$

Comment: This is a copying operation whose objective is to obtain

$$S[i;j;0:2r-1] = A_i[0:r-1]A_j[0:r-1] \quad \text{for all pairs } (i,j) \text{ with } i < j.$$

In our model, this operation could be done with maximal parallelism.

However, using only  $\binom{k+1}{2}r$  processors, the  $\binom{k+1}{2}2r$  elementary

copying operations are completed in two time units. For later

convenience we assume that the record associated with key  $A_i[\ell]$

contains a LABEL consisting of the pair of integers  $(i, \ell)$ .

$$5. \quad S[i;j;0:2r-1] \leftarrow \text{MERGE} (S[i;j;0:r-1], S[i;j;r:2r-1])$$

$$(i=0, \dots, k-1; j=i+1, \dots, k)$$

Comment: This step uses Valiant's merging algorithm and runs in time

$C_1 \log \log r + O(1)$ , for some constant  $C_1$ , using  $\binom{k+1}{2}r$  processors. The

original version of Valiant's merging algorithm can be readily

modified, so that, whenever two keys are identical the indices of

their respective subarrays are compared.

$$6. \quad \text{Let } (x, \ell) = \text{LABEL } S[i;j;q]$$

$$\text{If } x=i \text{ then } R[i;j;\ell] \leftarrow q-\ell \text{ else } R[j;i;\ell] \leftarrow q-\ell$$

$$(i=0, \dots, k-1; j=i+1, \dots, k; q=0, \dots, 2r-1)$$

$$7. \quad R[i;i;\ell] \leftarrow \ell \quad (i=0, \dots, k; \ell=0, \dots, r-1)$$

Comment: Steps 6 and 7 complete the count acquisition task. In

fact after Step 7 the content of  $R[i;j;\ell]$  is  $C_\ell^{(ij)}$ , in the

terminology of Section 1. Step 6 can be executed in two time

units using  $\binom{k+1}{2}r$  processors, whereas Step 7 uses  $\binom{k+1}{2}r$

processors and runs in one time unit.

$$8. \quad \text{rank}(A_1[\ell]) \leftarrow \sum_{j=0}^k R[i;j;\ell] \quad (i=0, \dots, k; \ell=0, \dots, r-1)$$

Comment: This step implements the rank computation. For any pair  $(i, \ell)$  the sum can be computed with  $\lfloor (k+1)/2 \rfloor$  processors in time  $\lceil \log(k+1) \rceil \approx \log \log n$ . The total number of processors used is therefore  $n \lfloor (k+1)/2 \rfloor$ .

$$9. \quad A[\text{rank}(A_1[\ell])] \leftarrow A_1[\ell] \quad (i=0, \dots, k; \ell=0, \dots, r-1)$$

end

To complete the analysis of the algorithm, we observe that none of Steps 4-7 uses more than  $\binom{k+1}{2} r$  processors. But

$$r \frac{k(k+1)}{2} = \lfloor n / \lceil \log n \rceil \rfloor \lceil \log n \rceil \left( \frac{\lceil \log n \rceil + 1}{2} \right) \leq n \frac{\lceil \log n \rceil + 1}{2}$$

where the last inequality is due to the removal of the "floor" sign.

Also, Step 8 uses  $n \lfloor (k+1)/2 \rfloor \leq n(\lceil \log n \rceil + 1)/2$ . Since, for all  $n \geq 4$   $n \geq 4, n(\lceil \log n \rceil + 1)/2 < \lfloor n \log n \rfloor$ , the inductive hypothesis on the number of processors is extended.

Finally, let  $T(n)$  denote the running time of the algorithm for  $n$  keys. Since  $r \approx n/\log n$  we obtain

$$T(n) = T\left(\frac{n}{\log n}\right) + C_2 \log \log n + C_3$$

for some constants  $C_2$  and  $C_3$ . It is easily verified that a function of the form  $C_2(\log n) + o(\log n)$  is a solution of the above recurrence. It is worth noting that for the same number of processors, Valiant proposes a sorting scheme of the merge-sort type ([4], Corollary 8) which runs in time  $2 \log n \cdot \log \log n - o(\log n \cdot \log \log n)$ .

### 3. Parallel sorting algorithms with no memory fetch conflicts

We shall now consider a family of algorithms for sorting  $n$  numbers in parallel with  $n^{1+\alpha}$  processors ( $0 < \alpha \leq 1$ ) in time  $(C'/\alpha) \log n + o(\log n)$ ,

for some constant  $C'$ . Each of these algorithms has the same performance as the corresponding algorithm by Hirschberg [5], although no memory fetch conflict occurs in this case. Again, we make the inductive hypothesis that for  $p < n$ , Algorithm SORT2 uses  $p^{1+\alpha}$  processors to sort  $p$  keys. The format of SORT2 closely parallels that of SORT1, with a few crucial differences to be noted.

Algorithm SORT2

begin

1.  $k \leftarrow \lceil n^\alpha \rceil, r \leftarrow \lfloor n / \lceil n^\alpha \rceil \rfloor$
2. Define arrays  $S[0:k;0:k;0:2r-1]$ ,  $R[0:k;0:k;0:r-1]$  and  $A_i[0:r-1] \leftarrow A[ir:(i+1)r-1]$  ( $i=0, \dots, k-1$ ),  $A_k[0:n-kr-1] \leftarrow A[kr:n-1]$  for  $n > kr$ .
3.  $A_i[0:r-1] \leftarrow \text{SORT2}(A_i[0:r-1])$  ( $i=0, \dots, k-1$ ),  $A_k[0:n-kr-1] \leftarrow \text{SORT2 } A_k[0:n-kr-1]$

Comment: This parallel recursive call of SORT2 sorts  $k$  sets of

$r$  keys each and, possibly, one set of  $n-kr < k$  keys. By the

inductive hypothesis, at most  $kr^{1+\alpha} + (n-kr)^{1+\alpha} \triangleq N$  processors are

used. Since  $n-kr < k$ , then  $N < kr^{1+\alpha} + (n-kr) \cdot k^\alpha = kr(r^\alpha - k^\alpha) + n \cdot k^\alpha$ .

Also  $kr = \lceil n^\alpha \rceil \cdot \lfloor n / \lceil n^\alpha \rceil \rfloor \leq n$ , whence  $N < n(r^\alpha - k^\alpha + k^\alpha) \approx n \cdot n^{(1-\alpha)\alpha^2}$

$= n^{1+\alpha-\alpha^2} < n^{1+\alpha}$ , where we have used the approximation  $r \approx n^{1-\alpha}$ .

Steps 1-3 are analogous to the corresponding ones in SORT1; however,

the copying operation implemented by Step 4 of SORT1 must be

considerably modified, as shown by the following Steps 4-6, to

avoid fetch conflicts. Here again,  $A_k$  is extended to size  $r$

as in SORT1.

4.  $S[i;k;0:r-1] \leftarrow A_i[0:r-1]$  ( $i=0, \dots, k-1$ )  
 $S[0;j;r:2r-1] \leftarrow A_j[0:r-1]$  ( $j=1, \dots, k$ )



5. for  $m \leftarrow 0$  step 1 until  $\lceil \log(k+1) \rceil - 2$  do  
 $S[i; j-2^m; 0:r-1] \leftarrow S[i; j; 0:r-1]$   
      $(j=k-2^m+1, \dots, k; i=0, \dots, j-2^m-1)$   
 $S[i+2^m; j; r:2r-1] \leftarrow S[i; j; r:2r-1]$   
      $(i=0, \dots, 2^m-1; j=i+2^m+1, \dots, k)$

6. Let  $\lceil \log(k+1) \rceil - 1 = v$ .  
 $S[i; j-2^v; 0:r-1] \leftarrow S[i; j; 0:r-1]$   
      $(j=2^v+1, \dots, k; i=0, \dots, j-2^v-1)$   
 $S[i+2^v; j; r:2r-1] \leftarrow S[i; j; r:2r-1]$   
      $(i=0, \dots, k-2^v-1; j=i+2^v+1, \dots, k)$

Comment: Steps 4-6 jointly replicate each  $A_i[0:r-1]$  the required number  $k$  of times. Step 4 is an initial copy; Step 5 consists of  $(\log \lceil k+1 \rceil - 1)$  stages, each of which doubles the ranges of the indices; Step 6 accounts for the fact that  $k$  may not be a power of 2 and completes filling the array  $S$ . Clearly this copying operation is implemented in  $\log \lceil k+1 \rceil + 1 \approx \alpha \log n + 1$  time units. A straightforward analysis shows that that the largest number of processors used in any of these stages is at most  $5/16$  of the total number  $\binom{k+1}{2} 2r$  of cells of  $S$  to be filled. It is also easily shown that  $(5/16) \binom{k+1}{2} 2r \approx (5/16) (n^\alpha + 1) n^\alpha \cdot n^{1-\alpha} < n^{1+\alpha}$  for any  $n \geq 1$  and  $\alpha > 0$ .

7.  $S[i; j; 0:2r-1] \leftarrow \text{MERGE}(S[i; j; 0:r-1], S[i; j; r:2r-1])$   
      $(i=0, \dots, k-1; j=i+1, \dots, k)$ .

Comment: This step uses a stable version of Batcher's merging algorithm [1], which is easily obtained by requiring that whenever two identical keys are encountered their array indices be compared (see Appendix). The following facts about Batcher's merging algorithm are well-known:

(i) no fetch conflict occurs because at any stage (or, time unit) each key is compared with exactly one key; (ii)  $\binom{k+1}{2}r \approx [(n^\alpha+1)n^\alpha/2]$ .  $n^{1-\alpha} < n^{1+\alpha}$  processors are used; (iii) merging is completed in  $\log r \approx (1-\alpha)\log n$  time units.

8. Steps 8, 9, 10, and 11 of this algorithm are respectively identical to Steps 6, 7, 8, and 9 of SORT1 and are therefore omitted. The latter are clearly free of memory fetch conflicts. The analysis of SORT1 showed that at most  $\max\left(\binom{k+1}{2}r, n \lfloor (k+1)/2 \rfloor\right)$  processors were used in any of those steps. In the present case, we have already shown that  $\binom{k+1}{2}r < n^{1+\alpha}$ ; similarly we conclude  $n \lfloor (k+1)/2 \rfloor \leq n(n^\alpha+1)/2 < n^{1+\alpha}$ .

From the performance viewpoint

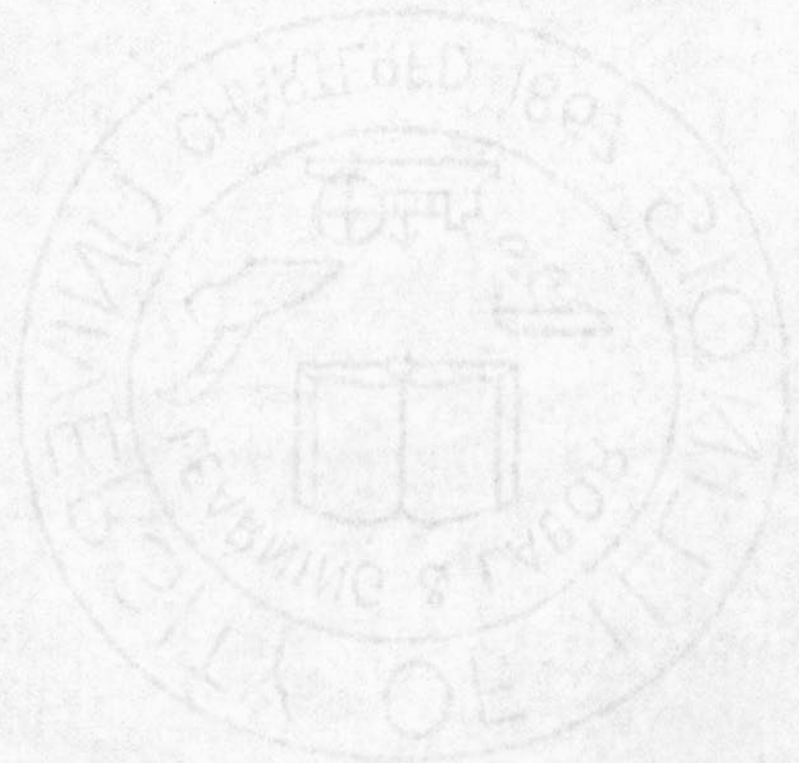
From the performance viewpoint, all steps of the algorithm require at most  $n^{1+\alpha}$  processors, as postulated. This extends the inductive hypothesis on the number of processors used by the algorithm. As to the running time  $T(n)$ , we note the following: Steps 4-6 jointly require  $\alpha \log n + 1$  time units; Step 7 requires  $(1-\alpha)\log n$  time units; Step 10 requires  $\alpha \log n$  time units; Steps 8, 9, and 11 run in constant time. Since Step 3 is a recursive call of SORT2 on sets of  $r \approx n^{1-\alpha}$  elements, we obtain for  $T(n)$  the recurrence equation

$$T(n) = T(n^{1-\alpha}) + (C'_1\alpha + C'_2)\log n + C'_3$$

for some constants  $C'_1$ ,  $C'_2$ , and  $C'_3$ . It is easily verified that a function of the form  $[C'_1\alpha + C'_2]/\alpha \log n + o(\log n)$  is a solution of this equation, whence  $T(n) \leq (C'/\alpha)\log n + o(\log n)$ .

References

1. K. E. Batcher, "Sorting networks and their applications," Proc. AFIPS Spring Joint Computer Conference, Vol. 32, pp. 307-314, April 1968.
2. D. E. Muller and F. P. Preparata, "Bounds to Complexities of Networks for Sorting and for Switching," Journal of the ACM, Vol. 22, No. 2, pp. 195-201, April 1975.
3. F. Gavril, "Merging with parallel processors," Comm. ACM, Vol. 18, 10, pp. 588-591, October 1975.
4. L. G. Valiant, "Parallelism in Comparison Problems," SIAM Journal of Computing, Vol. 4, 3, pp. 348-355, September 1975.
5. D. S. Hirschberg, "Fast Parallel Sorting Algorithms," Tech. Rep., Department of Electr. Eng., Rice University, Houston, Texas, January 1977.
6. D. E. Knuth, The Art of Computer Programming. Vol. III: Sorting and Searching, Addison-Wesley, Reading, Mass., 1972.





## Appendix

A stable version of Batchier's merging algorithm.

The original version of Batchier's odd-even merging algorithm runs as follows (here, for simplicity, we assume that the common length of the sequences to be merged is a power of 2):

$$\text{MERGE}(A[0: 2^{k-1}-1], A[2^{k-1}: 2^k-1])$$

1.  $A'[j] \leftarrow A[2j], A'[2^{k-1} + j] \leftarrow A[2j+1] \quad (j=0,1,\dots,2^{k-1}-1)$
2.  $B[0: 2^{k-1}-1] \leftarrow \text{MERGE}(A'[0: 2^{k-2}-1], A'[2^{k-2}: 2^{k-1}-1])$   
 $B[2^{k-1}: 2^k-1] \leftarrow \text{MERGE}(A'[2^{k-1}: 3 \cdot 2^{k-2}-1], A'[3 \cdot 2^{k-2}: 2^k-1])$
3.  $A[2j-1] \leftarrow \min(B[j], B[2^{k-1} + j-1]), A[2j] \leftarrow \max(B[j], B[2^{k-1} + j-1])$   
 $(j=1,\dots,2^{k-1}-1)$

This algorithm is not stable (see [6], p. 135, exercise 13), because in compliance with the rules of the comparator module, whenever  $B[j] = B[2^{k-1} + j-1]$  in Step 3, the algorithm assigns  $A[2j-1] \leftarrow B[j]$ ,  $A[2j] \leftarrow B[2^{k-1} + j-1]$ . Fortunately, however, with a simple modification, stability can be attained. Specifically, we associate with each key of the initial array  $A[0: 2^k-1]$  a label, and set  $\text{LABEL}(A[j]) \leftarrow j$  (notice that all labels are distinct). We then replace Step 3 above with the following step:

- 3!. If  $B[j] = B[2^{k-1} + j-1]$  then  $A[2j-1] \leftarrow$  key with smaller label  
 $A[2j] \leftarrow$  key with larger label  
else  $A[2j-1] \leftarrow \min(B[j], B[2^{k-1} + j-1]), A[2j] \leftarrow \max(B[j], B[2^{k-1} + j-1])$   
 $(j=1,\dots,2^{k-1}-1)$  .

We now prove that the new version of the algorithm is stable.

Assume that in the original array  $A[0: 2^k-1]$  the subarrays  $A[0: t_1-1]$ ,  $A[t_1: s_1-1]$ , and  $A[s_1: 2^k-1]$  contain keys which are respectively less, equal, and larger than some fixed value  $a$ ; similarly for  $A[2^{k-1}: 2^{k-1}+t_2-1]$ ,

$A[2^{k-1}+t_2: 2^{k-1}+s_2-1]$ , and  $A[2^{k-1}+s_2: 2^k-1]$ . Assume inductively that the merged sequences obtained in Step 2 are stably sorted and consider a key  $A[p]$  for  $p \in [t_1, s_1-1]$ . Assume at first  $p = 2j$ ; then, by Step 1,  $A'[j] = A[2j]$ . Moreover, there are  $\lceil t_2/2 \rceil$  keys in  $A'[2^{k-2}: 2^{k-1}-1]$  strictly smaller than  $A[2j]$ ; whence  $A[2j] = B[j + \lceil t_2/2 \rceil]$ . According to Step 3'  $B[j + \lceil t_2/2 \rceil]$  is compared with  $B[2^{k-1} + j-1 + \lceil t_2/2 \rceil]$ . Suppose  $t_2$  is even: then if  $(2j-1) \in [t_1, s_1-1]$ ,  $B[2^{k-1} + j-1 + \lceil t_2/2 \rceil] = A[2j-1]$  and we compare (the labels of)  $A[2j]$  and  $A[2j-1]$ ; otherwise, i.e., when  $2j-1 = t_1-1$  or, equivalently,  $A[2j] = A[t_1]$ , the latter is compared with a key less than a. Suppose now that  $t_2$  is odd: then if  $(2j+1) \in [t_1, s_1-1]$ , we have  $B[2^{k-1} + j-1 + \lceil t_2/2 \rceil] = A[2j+1]$  and we compare (the labels of)  $A[2j]$  and  $A[2j+1]$ ; otherwise  $A[2j] = A[s_1-1]$  and  $A[s_1-1]$  is compared with a key which is either larger or has a larger label. Clearly, in the given case, stability is ensured, and by analogous arguments we can treat all other cases.