JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **11,** 332–337 (1991)

Parallel Quicksort

BOGDAN S. CHLEBUS*

Instytut Informatyki, Uniwersytet Warszawski, PKiN, p. 850, 00-901 Warszawa, Poland

AND

IMRICH VRŤO

Institute of Technical Cybernetics, Slovac Academy of Sciences, Dubravska cesta 9, 842-37 Bratislava, Czechoslovakia

A parallel version of *quicksort* on a *CRCW PRAM* is developed. The algorithm uses n processors and a linear space to sort n keys in the expected time $O(\log n)$ with large probability. © 1991 Academic Press, Inc.

1. INTRODUCTION

Sorting is an important problem with practical applications. Among the known efficient sequential sorting algorithms is *quicksort* invented by Hoare [18]. In this paper we give a parallel version of *quicksort* implemented on a *CRCW PRAM*.

The *PRAM* is a model of a parallel computing device with a shared global memory (cf. [14, 22]). It consists of a number of processors, each being a sequential *RAM* with its own local memory, communicating via the shared memory. The processors execute the same program in a synchronized manner. There are some variants of the basic model, differing in whether or not the concurrent access conflicts to shared memory locations are allowed. In this paper we assume the *concurrent-read concurrent-write* (*CRCW*) *PRAM* which allows concurrent access for both reading and writing (see [7] for a comprehensive comparison of various *CRCW PRAM* submodels).

There are a number of parallel sorting algorithms which can be effectively implemented on a *PRAM* (cf. [4, 5, 26]). The best achieved simultaneous resource bounds are time $O(\log n)$ on n processors. The first *PRAM* algorithm attaining these time bounds was developed by Reischuk [25]. Next Ajtai *et al.* [2] discovered depth $O(\log n)$ sorting networks, and Reif and Valiant [24] developed a fixed-interconnection-networks algorithm sorting in $O(\log n)$ time. Both these constructions can be implemented on a *PRAM* in time $O(\log n)$ using n processors. Cole [9] gave a *PRAM* implementation of *mergesort* running in time $O(\log n)$ on n processors.

* Current address: Department of Computer Science, University of California, Riverside, CA 92521.

If the input is a sequence of n integers of polynomial magnitude then there are sorting algorithms on *PRAM* which have a better performance measured as the product of time and the number of processors, while preserving time $O(\log n)$. Among them are the algorithms of Rajasekaran and Reif [23], Hagerup [15, 16], and Chlebus [6].

Some of the algorithms mentioned above are deterministic and some are probabilistic, that is, in the latter case the specified time bounds are expected with large probability. Namely, the algorithms of Ajtai *et al.* [2], Cole [9], and Hagerup [16] are deterministic, and the algorithms of Chlebus [6], Hagerup [15], Rajasekaran and Reif [23], Reif and Valiant [24], and Reischuk [25] are probabilistic.

In this paper we develop a parallel version of *quicksort* running on an n processor *CRCW PRAM*, using linear space and completing sorting in time $O(\log n)$ with large probability. The sequential *quicksort* may use randomness in two ways: either the input is assumed to be randomly ordered or the partitioning element is being selected randomly. The same approach is possible for the algorithm presented in this paper.

Suppose first that the input is randomly ordered. Then the algorithm can be implemented on any *CRCW PRAM* satisfying the following two general requirements:

(1) concurrent writing is always successful;

(2) if a number of processors attempt to write to the same memory location then the winner is selected depending on the set of indices of the processors only (and not, for instance, on the values that they try to write).

An example of such a *CRCW PRAM* is the *PRIORITY* model in which the processor with the smallest index succeeds in writing (cf. [7]).

In the general case, with no assumption on the input, the writing conflicts need to be resolved randomly. This means that if a number of processors attempt to write to the same memory cell then exactly one of them succeeds and each is equally likely to be selected. To facilitate the exposition of the algorithm we assume that the random write-conflict resolution is a part of the *CRCW PRAM* architecture. Such a variant of a *CRCW PRAM* is usually referred to as *Random Write PRAM*. It can be efficiently simulated by deterministic *CRCW PRAM* if each of its processors has a source of random bits. Martel and Gusfield [21] describe such a simulation. If their method is applied to our algorithm then the resulting randomized algorithm has the same asymptotic $O(\log n)$ expected time behavior when run on a deterministic *CRCW PRAM*.

Various parallelizations of *quicksort* have already been described in the literature. Wiedermann [28] announced that the parallel version of *quicksort* on the parallel decision-tree model of Valiant [27] has expected $O(\log n)$-time performance. Horowitz and Zorat [19] developed an algorithm sorting n numbers on N processors in expected time of $O(n(1 - 1/N) + (n/N)\log(n/N)$. Akl [3] described an algorithm using n^{1-e} processors, where $0 < e < 1$, and running in $O(n^e \log n)$time. The algorithm is an adaptation of the "deterministic" *quicksort* in which the partitioning element is the median. Asynchronous implementation of this algorithm is also described by Akl [4]. Heidelberg *et al.* [17] describe a parallelization of quicksort for a shared memory multiprocessor using a fetch-and-add synchronization primitive; the paper contains a comprehensive list of references to other papers describing practical parallel sorting algorithms based on the paradigm of quicksort. Martel and Gusfield [21] give a version of *quicksort* sorting n numbers in $O(\log n)$ expected time using n processors in the *CRCW PRAM* model. The high-level ideas on which their and our algorithms are based are similar. The difference is in the implementation of the binary search tree. We represent the tree by pointers stored in an array, whereas Martel and Gusfield [21] use the array directly with no intermediate addressing. Their implementation requires almost $O(n^3)$ space on the average, while in the worst case the space usage may be even exponential. Our representation needs a linear space.

The main contributions of this paper are twofold. First we develop a version of *quicksort* on the *CRCW PRAM* using n processors and $O(n)$ space, and running in the expected time $O(\log n)$. Second, we analyze the time performance of this algorithm. Both the algorithm and its analysis are exceptionally simple, and in this respect compare favorably with all the known *PRAM* sorting algorithms operating in time $O(\log n)$ and using n processors. The analysis is basically that of the height of the random binary search trees. A comprehensive treatment of this topic was given by Devroye [10, 11]. Here we develop a different approach to this problem and using only elementary means show how to obtain a satisfactory analysis.

2. THE ALGORITHM

The sequential *quicksort* sorts by selecting a *partitioning element* and then dividing the remaining keys into two parts: the smaller elements to be placed into the left part of the array and the larger ones to be placed in the right one. After regrouping the keys accordingly the two remaining parts are sorted recursively. The sequential *quicksort* can also be visualized as constructing a binary search tree: the partitioning element plays the role of the root, the smaller elements going to the left subtree and the larger ones to the right subtree. The algorithm we present follows this very interpretation and constructs a binary search tree. First a partitioning element is selected and then, instead of processing one part of the keys after the other, the next partitioning elements are selected for the smaller and greater keys in parallel. The algorithm does not perform any regrouping of keys; instead, since there is a processor for each key, the processors get to know in which part of the tree they are by reading the value of the partitioning element. This is done level by level, starting from the root and then going down toward the leaves. Having constructed the tree, the inorder of the elements is found by scanning the tree again in the level-by-level manner, this time however working up from the leaves toward the root. This completes the task since the inorder is the sorted order.

The algorithm sorts n keys stored in the global memory in the array $KEY[1..n]$. The following one-dimensional global arrays of length n: *LEFT_SON*, *RIGHT_SON*, *LEFT_NUMBER*, *RIGHT_NUMBER*, and *ORDER* are also used. There are n processors, and we write the small letters p and q to denote processors with the respective identifying numbers p and q. Each of the processors has its own local memory. We adhere to the convention to write the number of a processor as a subscript to its variable to distinguish the local variables from the global ones, for instance, as in the local variable X_p.

We use the following parallel-loop instruction, written:

for each processor p **in parallel repeat** I **endrepeat**

where I is an instruction. This parallel loop is performed in a synchronized manner as follows. All the processors which have not completed the loop yet start executing I in parallel. A processor completes its loop if it encounters the instruction **exit** occurring in I. If this has not happened during a particular execution of I then the processor waits for the remaining processors (which have not exited yet) and all of them simultaneously resume performing I for the next time. The whole loop is completed after all the processors have exited. It is clear that this instruction can be implemented on a *PRAM* with both the concurrent read and write capabilities.

The algorithm consists of three calls of the procedures described below.

```
Algorithm parallel quicksort
begin
  build_tree;
```

```
    count_descendants;
    find_order
end.
```

The procedure *build_tree* constructs a binary tree with the nodes labeled by processor numbers. The tree has the property that if a node storing p is a left descendant of a node storing q, where p and q are processor numbers, then either $KEY[p] < KEY[q]$ or $KEY[p] = KEY[q]$ and $p < q$, and if a node storing p is a right descendant of a node storing q then either $KEY[p] > KEY[q]$ or $KEY[p] = KEY[q]$ and $p > q$. Therefore, having built such a tree, sorting can be completed by finding the inorder of the nodes of the tree.

Below we give a detailed description of the procedures. The local variable $FATHER_p$ generally stores the deepest known ancestor of p. The sons of $FATHER_p$ are stored in $LEFT\_SON[FATHER_p]$ and $RIGHT\_SON[FATHER_p]$. If processor p manages to write its number into one of these memory locations, then p exits the main loop, else it tries to attain this goal in the next execution of the loop, until it eventually succeeds. Note that there is no matching **until** for **repeat** because the loop terminates when all the processors have exited.

```
procedure build_tree
  for each processor $p$ in parallel do
    write($p$) into $ROOT$;
    $FATHER_p$ := read($ROOT$);
    $LEFT\_SON[p] := RIGHT\_SON[p] := n + 1$
  od;
  for each processor $p \neq ROOT$ in parallel repeat
    if ($KEY[p] < KEY[FATHER_p]$) or
       ($KEY[p] = KEY[FATHER_p]$ and $p < FATHER_p$)
    then
      write($p$) into $LEFT\_SON[FATHER_p]$;
      $LS_p$ := true;
      comment if $p$ exits then $LS_p$ will store
      the information that $p$ is a left son endcomment
      $X_p$ := read($LEFT\_SON[FATHER_p]$);
      if $p = X_p$ then exit
        else $FATHER_p := X_p$ fi
    else
      write($p$) into $RIGHT\_SON[FATHER_p]$;
      $LS_p$ := false;
      $X_p$ := read($RIGHT\_SON[FATHER_p]$);
      if $p = X_p$ then exit
        else $FATHER_p := X_p$ fi
    fi
  endrepeat
endprocedure {build_tree}
```

Procedure *count_descendants* fills the arrays *LEFT_NUMBER* and *RIGHT_NUMBER* in such a way that when it is completed then $LEFT\_NUMBER[p]$ and $RIGHT\_NUMBER[p]$ are the numbers of left and right descendants of p.

```
procedure count_descendants
  for each processor $p$ in parallel do
    $LEFT\_NUMBER[p] := RIGHT\_NUMBER[p] := Y_p := 0$;
  for each processor $p \neq ROOT$ in parallel repeat
    case
      $LEFT\_SON[p] = RIGHT\_SON[p] = n + 1$ :
      $Y_p := 1$;
      comment $p$ is a leaf endcomment
      $LEFT\_SON[p] = n + 1$ and $RIGHT\_SON[p] \leq n$
      and $RIGHT\_NUMBER[p] > 0$ :
        $Y_p := RIGHT\_NUMBER[p] + 1$;
      comment $p$ has only a right son endcomment
      $LEFT\_SON[p] \leq n$ and $LEFT\_NUMBER[p] > 0$
      and $RIGHT\_SON[p] = n + 1$ :
        $Y_p := LEFT\_NUMBER[p] + 1$;
      $LEFT\_SON[p] \leq n$ and $LEFT\_NUMBER[p] > 0$
      and $RIGHT\_SON[p] \leq n$ and $RIGHT\_NUMBER[p] > 0$ :
        $Y_p := LEFT\_NUMBER[p] + RIGHT\_NUMBER[p] + 1$
    endcase;
    if $p \neq ROOT$ and $Y_p \neq 0$ then
      if $LS_p$
        then $LEFT\_NUMBER[FATHER_p] := Y_p$
        else $RIGHT\_NUMBER[FATHER_p] := Y_p$ fi
        comment $Y_p$ stores the value of the number of de-
        scendants of $p$ including $p$; $LS_p$ is true iff $p$ is a left
        son of $FATHER_p$ endcomment

      exit
    fi
  endrepeat
endprocedure {count_descendants}
```

The last procedure fills the array *ORDER* with the ordinal numbers of processors ordered by $KEY[p]$.

```
procedure find_order
  for each processor $p$ in parallel do $ORDER[p] := 0$ od;
  $ORDER[ROOT] := LEFT\_NUMBER[ROOT] + 1$;
  for each processor $p \neq ROOT$ in parallel repeat
    $Z_p$ := read($ORDER[FATHER_p]$);
    if $Z_p \neq 0$ then
      if $LS_p$
        then $ORDER[p]$ :=
        $Z_p - RIGHT\_NUMBER[p] - 1$
        else $ORDER[p]$ :=
        $Z_p + LEFT\_NUMBER[p] + 1$ fi
```

```
            exit
        fi
    endrepeat
endprocedure {find_order}
```

The correctness of procedures *build_tree*, *count_descendants*, and *find_order* follows directly from the previous discussion. Note that procedures *count_descendants* and *find_order* could be implemented to run in $O(\log n)$ time on $n/\log n$ processors using the optimal expression tree evaluation algorithm (cf. [1, 20, 13]). This completes the description of *parallel quicksort*.

Below we discuss an example that shows interrelations between a hypothetical binary search tree associated with the execution of the algorithm and data stored in the arrays. Figure 1 shows the input keys in array *KEYS*. There is a processor associated with each memory cell and identified by the address. Figure 2 depicts a binary search tree associated with some execution of the algorithm. The first number identifies a processor, and the second one is the corresponding key. Figure 3 shows the data stored in arrays *FATHER*, *LEFT_SON* and *RIGHT_SON* after the steps corresponding to the levels of the tree depicted on Fig. 2.

3. ANALYSIS OF THE ALGORITHM

Algorithm *parallel quicksort* consists of three procedures. Procedure *build_tree* constructs a binary tree in a level-by-level manner, starting from the root. The procedure is completed within time proportional to the height of the tree. Similarly the two remaining procedures traverse the tree in parallel in the level-by-level manner. Therefore the overall time of the algorithm is proportional to the height of the tree.

Denote by H_n the random variable equal to the height of the tree built by procedure *build_tree* on an input of n keys; it is equal to the height of a random binary search tree with n nodes. We use the following notations for integers n and x:

$$P_n(x) = Prob(H_n < x),$$

$$R_n(x) = Prob(H_n \geqslant x) = 1 - P_n(x).$$

LEMMA 1. *The numbers $R_n(x)$ satisfy the following inequality:*

$$R_n(x) \leqslant \frac{2}{n} \sum_{i=0}^{n-1} R_i(x-1),$$

for all positive integers n and x.

Address	1	2	3	4	5	6	7	8	9	10
KEYS	30	24	27	23	28	21	26	29	22	25

FIGURE 1

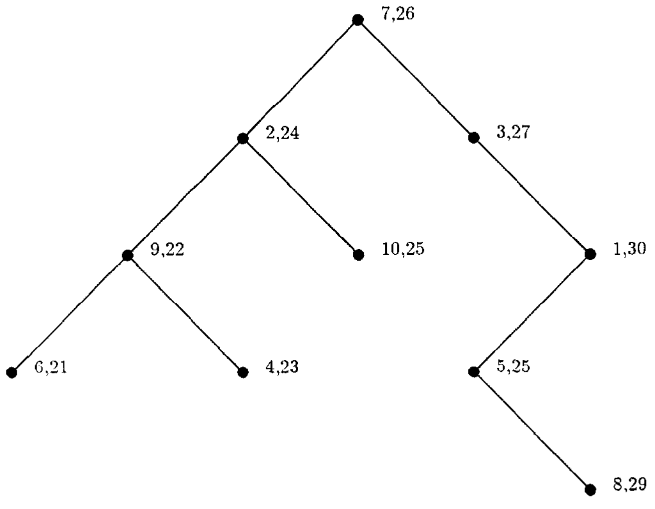

FIGURE 2

Proof. It follows from the definition of H_n that numbers P_n satisfy the recurrence relation

$$P_n(x) = \frac{1}{n} \sum_{i=0}^{n-1} P_i(x-1) \cdot P_{n-i-1}(x-1)$$

for $n, x \geqslant 1$, with the initial conditions $P_0(x) = 1$, for $x = 0, 1, 2, \cdots$ and $P_n(0) = 0$, for $n = 1, 2, 3, \ldots$. Substitute $1 - R_n(x)$ for $P_n(x)$ to obtain

$$1 - R_n(x) = \frac{1}{n} \sum_{i=0}^{n-1} (1 - R_i(x-1)) \cdot (1 - R_{n-i-1}(x-1)).$$

Address	1	2	3	4	5	6	7	8	9	10
FATHER	7	7	7	7	7	7	R	7	7	7
LEFT_SON	0	0	0	0	0	0	0	0	0	0
RIGHT_SON	0	0	0	0	0	0	0	0	0	0

Level 1

Address	1	2	3	4	5	6	7	8	9	10
FATHER	3	7	7	2	3	2	R	3	2	2
LEFT_SON	0	0	0	0	0	0	2	0	0	0
RIGHT_SON	0	0	0	0	0	0	3	0	0	0

Level 2

Address	1	2	3	4	5	6	7	8	9	10
FATHER	3	7	7	9	1	9	R	1	2	2
LEFT_SON	0	9	0	0	0	0	2	0	0	0
RIGHT_SON	0	10	1	0	0	0	3	0	0	0

Level 3

Address	1	2	3	4	5	6	7	8	9	10
FATHER	3	7	7	9	1	9	R	5	2	2
LEFT_SON	5	9	0	0	0	0	2	0	6	0
RIGHT_SON	0	10	1	0	8	0	3	0	4	0

Level 5

FIGURE 3

Regrouping the terms we get

$$R_n(x) = \frac{2}{n}\sum_{i=0}^{n-1} R_i(x-1)$$

$$- \frac{1}{n}\sum_{i=0}^{n-1} R_i(x-1)\cdot R_{n-i-1}(x-1).$$

Now, since $R_i(x) \geq 0$, it is enough to discard the right-hand sum to obtain the required inequality. ■

A combinatorial interpretation of numbers satisfying the inequality of Lemma 1 in terms of the number of cycles in random permutations is studied in [8]. The next lemma shows that $R_n(x)$ converges exponentially to 0 with x converging to infinity.

LEMMA 2. *The numbers* $R_n(x)$ *satisfy the inequality*

$$R_n(x) \leq n^3 2^{-x+1}$$

for all positive integers n and x.

Proof. The proof is by induction on n. For $n = 1$ we have $R_1(1) = 1$ and $R_1(x) = 0$ for $x > 1$. Suppose that the inductive assumption $R_i(x) \leq i^3 2^{-x+1}$ holds for all i such that $0 \leq i < n$ and all positive integers x. Substituting this inequality into the inequality of Lemma 1 we obtain

$$R_n(x) \leq \frac{2}{n}\sum_{i=0}^{n-1} i^3 2^{-x+2} = n^{-1}2^{-x+3}\sum_{i=0}^{n-1} i^3$$

$$= n^{-1}2^{-x+3}\left(\frac{(n-1)n}{2}\right)^2 = (n-1)^2 n 2^{-x+1} < n^3 2^{-x+1},$$

which completes the proof. ■

As an example of how this lemma can be applied we shall show that the expectancy $E(H_n)$ of the random variable H_n equals $O(\log n)$.

COROLLARY 1. *The following inequality holds:* $E(H_n) \leq 3 \log n + 6$.

Proof. Use the equality $E(H_n) = \sum_{x=0}^{\infty} R_n(x)$ (cf. [12]). Substitute either 1 or $n^3 2^{-x+1}$ for $R_n(x)$, depending on whether $x \leq 3\lceil \log n\rceil$ or $x > 3\lceil \log n \rceil$, respectively. This gives

$$E(H_n) \leq \sum_{x=0}^{3\lceil \log n\rceil} 1 + \sum_{x=3\lceil \log n\rceil+1}^{\infty} R_n(x)$$

$$\leq 3\lceil \log n\rceil + 1 + \sum_{x=3\lceil \log n\rceil+1}^{\infty} n^3 2^{-x+1}$$

$$= 3\lceil \log n\rceil + 1 + 2n^3 \sum_{x=3\lceil \log n\rceil+1}^{\infty} 2^{-x}$$

$$= 3\lceil \log n\rceil + 1 + 2n^3 2^{-3\lceil \log n\rceil}$$

$$\leq 3\lceil \log n\rceil + 3 \leq 3\lceil \log n\rceil + 6,$$

which is the required inequality. ■

Devroye [10] showed that $E(H_n) = 4.31107\cdots\log_e(n) + o(\log n)$.

THEOREM 1. *There is a constant* $a > 0$ *such that* parallel quicksort *sorts n keys in time* $a(1 + (3 + b))\log n$ *with probability at least* $1 - n^{-b}$, *for each positive b.*

Proof. Substitute $1 + (3 + b)\log n$ for x in the inequality proved in Lemma 2 to obtain

$$R_n(1 + (3 + b)\log n) \leq n^3 n^{-3} n^{-b} = n^{-b}.$$

The number of steps performed by *parallel quicksort* on n keys is proportional to the height of the constructed tree. This completes the proof. ■

REFERENCES

1. Abrahamson, K., Dadoun, N., Kirkpatrick, D. G., and Przytycka, T. A simple parallel tree contraction algorithm. *J. Algorithms* **10** (1989), 287–302; *Proc. 25th Allerton Conference on Communication, Control and Computing,* 1987.
2. Ajtai, M., Komlos, J., and Szemeredi, E. An $O(n \log n)$ sorting network. *Combinatorica* **3** (1983), 1–19; An $O(n \log n)$ sorting network. *Proc. 15th Annual ACM Symposium on Theory of Computing,* 1983, pp. 1–9.
3. Akl, S. G. Optimal parallel algorithms for computing convex hulls and for sorting. *Computing* **33** (1984), 1–11.
4. Akl, S. G. *Parallel Sorting Algorithms.* Academic Press, San Diego, 1985.
5. Borodin, A., and Hopcroft, J. E. Routing, merging and sorting on parallel models of computation. *J. Comput. System Sci.* **30** (1985), 130–145; Routing, merging and sorting on parallel-models of computation. *Proc. 14th Annual ACM Symposium on Theory of Computing,* 1982, pp. 338–344.
6. Chlebus, B. S. Parallel iterated bucket sort. *Inform. Process. Lett.* **31** (1989), 181–183.
7. Chlebus, B. S., Diks, K., Hagerup, T., and Radzik, T. New simulations between CRCW PRAMs. *Proc. 7th International Conference on Fundamentals of Computation Theory,* 1989, Springer Lecture Notes in Computer Science, Vol. 380, pp. 95–104. Berlin/New York.
8. Chlebus, B. S., and Vrťo. Unifying binary search trees and permutations. Submitted for publication.
9. Cole, R. Parallel merge sort. *SIAM J. Comput.* **17** (1988), 770–785; *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science,* 1986, pp. 511–516.
10. Devroye, L. A note on the height of binary search trees. *J. Assoc. Comput. Mach.* **33** (1986), 489–498.
11. Devroye, L. Branching processes in the analysis of the heights of trees. *Acta Inform.* **24** (1987), 277–298.
12. Feller, W. *An Introduction to Probability Theory and Its Applications,* Vol. 1. Wiley, New York, 1950.
13. Gibbons, A., and Rytter, W. An optimal parallel algorithm for dynamic tree expression evaluation and its applications. *Proc. Symposium on*

Foundations of Software Technology and Theoretical Computer Science, 1986, pp. 453–469.

14. Gibbons, A., and Rytter, W. *Efficient Parallel Algorithms*. Cambridge Univ. Press, Cambridge, 1988.
15. Hagerup, T. Hybridsort revisited and parallelized. *Inform. Process. Lett.* **32** (1989).
16. Hagerup, T. Towards optimal parallel bucket sorting. *Inform. and Comput.* **75** (1987), 39–51.
17. Heidelberg, P., Norton, A., and Robinson, J. T. Parallel quicksort using fetch-and-add. *IEEE Trans. Comput.* **39** (1990), 133–138.
18. Hoare, C. A. R. Quicksort. *Comput J.* **5** (1962), 10–15.
19. Horowitz, E., and Zorat, A. Divide-and-conquer for parallel processing. *IEEE Trans. Comput.* **C-32** (1983), 582–585.
20. Kosaraju, S. Rao, and Delcher, A. L. Optimal parallel evaluation of tree structured computation by raking. *Proc. 3rd Aegean Workshop on Computing*, 1988.
21. Martel, C. U., and Gusfield, D. A fast parallel quicksort algorithm. *Inform. Process. Lett.* **30** (1989), 97–102.
22. Parberry, I. *Parallel Complexity Theory*. Pitman, London, 1987.
23. Rajasekaran, S., and Reif, J. H. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.* **18** (1989), 594–607; An optimal parallel algorithm for integer sorting. *Proc. 26th Annual IEEE Symposium on Foundations of Computer Science*, 1985, pp. 496–503.
24. Reif, J. H., and Valiant, L. A logarithmic time sort for linear size networks. *J. Assoc. Comput. Mach.* **34** (1987), 60–76; *Proc. 15th Annual ACM Symposium on Theory of Computing*, 1983, pp. 10–16.
25. Reischuk, R. Probabilistic algorithms for sorting and selection. *SIAM J. Comput.* **14** (1985), 396–409; A fast probabilistic parallel sorting algorithm. *Proc. 22nd Annual IEEE Symposium on Foundations of Computer Science*, 1981, pp. 212–219.
26. Shiloach, Y., and Vishkin, U. Finding the maximum, merging, and sorting in a parallel computation model. *J. Algorithms* **2** (1981), 88–102.
27. Valiant, L. G. Parallelism in comparison problems. *SIAM J. Comput.* **4** (1975), 348–355.
28. Wiedermann, J. Asymptotically optimal parallel sorting algorithm. *Proc. Symposium Algoritmy, Smolenice*, 1977. [in Slovac]

Received May 1, 1990; accepted July 31, 1990

BOGDAN S. CHLEBUS was born on May 24, 1954, at Suwalki in Poland. He received his M.S. in mathematics in 1977 and his Ph.D. in computer science in 1982 from the Warsaw University. From 1982 to 1989 he was with the Institute of Informatics of the Warsaw University. Since 1989 he has been visiting the Department of Computer Science of the University of California at Riverside. His current research interests include parallel algorithms and architectures, and theory of computation.

IMRICH VRŤO was born at Rimanska Sobota in CSFR on December 17, 1953. He received his B.S. and M.S. in mathematics from the Comenius University, Bratislava, in 1978 and 1980, respectively, and his Ph.D. in computer science from the Slovac Academy of Sciences, Bratislava, in 1988. From 1978 to 1989 he was with the Institute of Technical Cybernetics. Since 1990 he has been a research fellow at Computing Center of the Slovac Academy of Sciences at Bratislava. His current research interests include parallel algorithms, VLSI complexity theory, and graph theory.