Adaptive Parallel Sorting

Vladimir Estivill-Castro* Derick Wood†

March 4, 1992

Abstract

The capacity of an algorithm to modify its behavior to the size, difficulty or structure of the problem instance has been named adaptivity. The literature on parallel algorithms qualifies as adaptive those algorithms that work with a fixed amount of processors on input of arbitrary size while the literature on sorting algorithms qualifies as adaptive those algorithms that perform work proportional to the amount of disorder in the input. We combine these two concepts and present the first two parallel sorting algorithms that

- are designed for a fixed number of processors and reconfigure according to the size of the input
- adapt to the amount of disorder in the input, and
- are cost optimal.

1 Introduction

Over the last few years, it has become increasingly evident that parallelism will provide an opportunity for faster, more efficient and more cost-effective computation. Because of its simplicity and universality, the *P-RAM* (Parallel Random Access Machine) model has resulted in a theoretically convenient parallel model of computation [11,18]. The *P-RAM* offers a flexible starting point for the designer of parallel algorithms. In most situations, however, when a problem involving n data items is to be solved in parallel, it is unrealistic on the part of the algorithm designer to assume that the number of available processors is bigger than or equal to n, especially when n is extremely large. Therefore, algorithms that *adapt* to a sublinear number of available processors and whose running time is a function of n as well as of the number of processors are desirable [2, page 159]. For the problem of sorting n elements, another property of the algorithm is desirable. The algorithms should *adapt* to the amount of disorder in the data; that is, its *cost* (the product of the number of processors and its running time) is a smooth non-decreasing function of the size and the disorder in

*Department of Computer Science, York University, North York, Ontario M3J 1P3, Canada.

†Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada

the input. Adaptive sorting algorithms for sequential computation are well-studied [9,10,17]. This paper presents the first parallel sorting algorithms that:

- are designed for a fixed number of procesors p and modify their behavior and running time according to the size of the input, (also referred as *self-reconfiguring* [4])
- adapt to the amount of disorder in the input,
- their worst-case running time varies inversely with p and is significantly smaller than the best sequential algorithm, and
- the cost of the parallel algorithm is optimal.

Our model of parallel computation consists of a *P-RAM* where there are $p \geq 1$ processors working synchronously and communicating though a random access memory. Each processor is a uniform-cost Random Access Machine (*RAM*) with usual operations and instructions (i.e. the cost of comparisons and arithmetic operations is constant). The processors are indexed by the natural numbers $P_0, P_1, \ldots, P_{p-1}$. During a given time unit a selected number of processors are active and execute the same instruction, each on a different data set; the remaining processors are inactive. Since the processors execute the same program such a model is also called a single-instruction, multiple-data-stream (*SIMD*) model [3].

In one step each processor can access (either reading from it or writing to it) one memory location. In the literature, *P-RAM* models differ in regard to simultaneous access of the same memory location by more than one processor. We use the following convention:

- *CRCW-P-RAM*: two processors are allowed to simultaneously read from or write into the same memory cell.
- *CREW-P-RAM*: two processors are allowed to simultaneously read from the same memory cell but concurrent writing in not allowed.
- *EREW-P-RAM*: no two processors are allowed to simultaneously read from or write into the same memory cell.

For the *CREW-P-RAM* and *EREW-P-RAM* models it is assumed that there is an initial threshold time of $\log p$ to activate p processors at the beginning of every computation [11,18].

To evaluate parallel algorithms we use the following well-established metrics [2,3]. Let X be a sequence of n elements from a total order, the parallel running time of a sorting algorithm A is the number of steps performed by A to sort X and is denoted by $T_A(X)$. The cost of a parallel sorting algorithm A is defined as the produc $p \cdot T_A(X)$ and is denoted by $C_A(X)$. A parallel algorithm is said to be *cost-optimal* if, for all X, $C_A(X)$ is within a constant factor of the lower bound. In this paper we adopt the common convention that the running time of the optimal sequential algorithm is a lower bound for the cost of a parallel algorithm. Moreover, we also adopt the convention that,

for any algorithm A on a *CRCW-P-RAM* with p processors the cost is at least p, since A will take one time step to activate the processors. Similarly, for any algorithm A on a *CREW-P-RAM* or *EREW-P-RAM* with p processors, the cost of A is at least $p \log p$ since A will take $\log p$ time to activate the processors (or broadcast n, the size of the input).

Our goal is to obtain a cost-optimal parallel sorting algorithms for all values of p and $n \geq p$.

In a comparison-based model of computation $\Omega(n \log n)$ comparisons are necessary in the worst-case to sort a sequence of length n. Frequently, the input is nearly-sorted and it can be sorted using less comparisons. Traditional sorting algorithms like *Quicksort* or *Mergesort* are oblivious to the existing order in the sequence (they even sort again an already sorted sequence). Algorithms that profit from the order in the input sequence are said to be *adaptive* [16, page 224]. The adaptive behavior of a sorting algorithm A is demonstrated by showing that the time required by A to sort a sequence X is a non-decreasing function of the size of X and the disorder in X. The following are common measures of disorder. The largest distance, $Dis(X)$, determined by an inversion in $X = \langle x_1, \ldots, x_n \rangle$, where (i, j) is an *inversion* if $i < j$ and $x_i > x_j$. The largest distance, $Max(X)$, an element must travel to reach its sorted position. The number, $Runs(X)$, of ascending runs.

A sequential sorting algorithm is optimally adaptive with respect to a measure of disorder M (or M-optimal) if it achieves, within a constant factor, maximal adaptivity for every input sequence [14].

Sequential adaptive sorting has been investigated extensively [9,10,17]; however, the literature on adaptive sorting for parallel computation consists of only a few special cases. Altman and Igarashi [6] present parallel algorithms requiring n processors that adapt to a previously known Dis value. Altman and Chlebus [5] present an algorithm that is cost-optimal for Dis on a *CRCW-P-RAM* with n processors (note that the number of processors must equal the size of the input). Carlsson and Chen [8] present an algorithm that is cost-optimal for $Runs$ on a *EREW-P-RAM* with $n/\log n$ processors. Levcopoulos and Petersson [13] claim to obtain a cost-optimal algorithm for Rem. However, when sorting a sequence X with n elements the algorithm uses $n/\log n$ processors during two phases that last $O(\log n)$ time and $O([n + Rem(X) \log Rem(X)]/logn)$ processors during one phase that last $O(\log n)$. The cost of their algorithm is at least $\Omega(n \log n)$ and not $O([n + Rem(X) \log Rem(X)]/logn)$ as they claim. Levcopoulos and Petersson [12] also propose a different model of computation. At any step the parallel algorithm may decide to declare several processors as idle and not be charged for them. The algorithm has full power to request more processors and they are supplied with no delay. Thus, the cost of the algorithm is redefined [12] as the sum over all time steps t_i of the number of processors active at step t_i. In this model, a parallel sorting algorithm is described. This algorithm estimates the number of inversions, (requesting a convenient number of processors in several phases), and sorts optimally with respect to Inv. Note that the *P-RAM* differs drastically from this model in that the number of processors is fixed through the computation and the algorithm can not avoid to be charged for those processors that declares idle. McGlinn [15] has studied adaptive sorting empirically.

Adaptive Parallel Merge: Merges s sorted sequences using $p = n^{1-\epsilon}$ processors.

for $i := 1$ to $\log s$
- Form $\lfloor s/2^i \rfloor$ pairs of sequences.
- Distribute the processors among the pairs and assign $\max\{1, (l'_i + l'_j)p/n\}$ processors to each pairs of sequences of lengths l'_i and l'_j.
- In parallel, merge each pair into a sorted sequence using Bilardi and Nicolau's algorithm.

Figure 1: Pseudo-code for *Adaptive Parallel Merge*

In this paper we present adaptive parallel algorithms that sort optimally with respect to a measure of disorder and use a fixed number of processors. In Section 2 we introduce *Parallel Mod Sort*, an adaptive parallel algorithm that is cost-optimal with respect to the measure Dis, and the measure Max, for all positive values of n and $p = n^{1-\epsilon}$ ($0 < \epsilon < 1$). Moreover, *Parallel Mod Sort* is cost-optimal for all variants of the *P-RAM*. In Section 3 we introduce *Parallel Block Sort*, an adaptive parallel algorithm that is cost-optimal with respect to the measure $Runs$, for all positive values of n and $p = n^{1-\epsilon}$ ($0 < \epsilon < 1$). Moreover, *Parallel Mod Sort* is cost-optimal for all variants of the *P-RAM*.

2 A cost-optimal algorithm for Dis and Max

In this section we present a parallel sorting algorithm that is cost-optimal with respect to Dis and adapts to the size of the input. The algorithm can be implemented in all *P-RAM* models. In order to present our sorting algorithm we must present *Adaptive Parallel Merge* (see Figure 1), an adaptive algorithm for merging $s(\leq p)$ sorted sequences, using p processors in a *EREW-P-RAM*. The algorithm is a generalization of Bilardi and Nicolau's [7] algorithm for merging two sorted sequences of total length n in $O(n/p + \log n)$ time using p processors.

Lemma 2.1 *Let S be a set of s sorted sequnces of lengths $l_1, l_2, \ldots, l_s$ (with $\sum_{i=1}^{s} l_i = n$). Assume the values s and l_i, for $i = 1, \ldots, s$ are known to all processors. Adaptive-Merge uses $p = n^{1-\epsilon}$ ($0 < \epsilon < 1$) processors to merge these s sorted sequences in $O([n/p]\log(s+1))$ time on a P-RAM.*

Proof: The algorithm is the same for all variants of the *P-RAM*. Each step is an application of Bilardi and Nicolau merging algorithm, thus each step takes

$$O\left(\frac{l'_i + l'_j}{(l'_i + l'_j)n^{-\epsilon}} + \log(l'_i + l'_j)\right)$$

time, for some l'_i, l'_j with $l'_i + l'_j \leq n$. Thus, each step requires $O(n^\epsilon + \log n) = O(n^\epsilon)$ time. Since the total number of step is $\lceil \log s \rceil$, the lemma follows. □

We use the following notation. Given an array $A = A[0, \ldots, n-1]$, we denote the subarray of A consisting of all elements in position l to position r by $A[l, \ldots, r]$. A subsequence of A is the sequence of elements $A[i(0)], A[i(1)], \ldots, A[i(s)]$ where $i : \{1, 2, \ldots, s\} \rightarrow \{0, \ldots, n-1\}$ is injective and monotonically increasing. For positive m, we denote by $\{A_k\}_m$ the subsequence determined by $i(r) = m \cdot r + k$, for $k = \{0, \ldots, n-1\}$. For example, for $m = 2$, $\{A_0\}_2$ denotes the subsequence of elements in even positions. We present now the sorting algorithm.

Parallel Mod Sort: Sorts an array $A[0, \ldots, n-1]$ with processors $P_0, \ldots, P_{p-1}$.

1.- *Divide the input into p subarrays.* Let $p-1$ of these subarrays consist of $\lceil n/p \rceil$ elements and the remaining subarray contains $n + 1 - (p-1)\lceil n/p \rceil (\leq \lceil n/p \rceil)$ elements. Assign processor P_i to the subarray $A[i\lceil n/p \rceil, \ldots, i\lceil n/p \rceil + \lceil n/p \rceil - 1]$, for $i = 0, \ldots, p-2$, and assign P_{p-1} to the subarray $A[(p-1)\lceil n/p \rceil, \ldots, n]$.

2.- *Sort each subarray.* In parallel, each processor P_i, for $i = 0, \ldots, p-1$, sorts its subarray using a sequential *Dis*-optimal sorting algorithm.

3.- *Test for sortedness.* In parallel, processor P_i, for $i = 0, \ldots, p-2$, checks if $A[i\lceil n/p \rceil + \lceil n/p \rceil - 1] > A[(i+1)\lceil n/p \rceil]$ and the "or" of these boolean values is computed. If A is sorted, terminate.

4.- *Divide the input into $p-1$ subarrays shifted* $\lceil n/2p \rceil$. Let these $p-1$ subarrays consist of $\lceil n/p \rceil$ elements. Assign processor P_i to the subarray $A[\lceil n/2p \rceil + i\lceil n/p \rceil, \ldots, \lceil n/2p \rceil + (i+1)\lceil n/p \rceil - 1]$, for $i = 0, \ldots, p-2$.

5.- *Sort each subarray.* In parallel, each processor P_i, for $i = 0, \ldots, p-2$, sorts its subarray using a sequential *Dis*-optimal algorithm.

6.- *Test for sortedness.* In parallel, processor P_i, for $i = 0, \ldots, p-2$, checks if $A[\lceil n/2p \rceil + (i+1)\lceil n/p \rceil - 1] > A[\lceil n/2p \rceil + (i-1)\lceil n/p \rceil]$ and the "or" of these boolean values is

computed. If A is sorted, terminate.

7.- *Compute k an estimate for $Dis(A)$.* Let R be the subsequence of A consisting of the elements in the subsequences $\{A_{\lceil n/2p\rceil+1}\}_{\lceil n/p\rceil}$ and $\{A_{\lceil n/2p\rceil}\}_{\lceil n/p\rceil}$. Use Altamn and Chlebus parallel algorithm to compute $Dis(R)$. Let $k = \lceil Dis(R)/2\rceil \lceil n/p\rceil$.

8.- *Merge subsequences formed by elements $k+1$ positions apart.*

8.1 If $p \leq k+1$ then,

8.1.1 For $i = 0, \ldots, k$, let $A_i = \{A_i\}_{k+1}$. Assign

$$A_{j\lceil (k+1)/p\rceil}, A_{j\lceil (k+1)/p\rceil+1}, \cdots A_{(j+1)\lceil (k+1)/p\rceil-1}$$

to processor P_j for $j = 0, \ldots, p-2$ and $A_{(p-1)\lceil (k+1)/p\rceil}, \ldots, A_{k+1}$ to processor P_{p-1}. In parallel, each processor merges its subsequences into a sorted subsequence.

8.1.2 Use *Adaptive Parallel Merge* to merge the resulting p subsequences, all of length no more than $\lceil n/p\rceil$ using p processors.

8.2 If $p > k+1$, use *Adaptive Parallel Merge* to merge $\{A_0\}_{k+1}, \ldots, \{A_k\}_{k+1}$ (each of length no more than $\lceil n/(k+1)\rceil$) using p processors.

The merit of the algorithm is expressed in the following theorem.

Theorem 2.2 *Let X be a sequence of n elements from a total order.*

- *In a CRCW-P-RAM with $p = n^{1-\epsilon}$ processors ($0 < \epsilon < 1$) Parallel Mod Sort sorts X in $O((n/p)[1+\log(Dis(X)+1)])$ time. The cost of Parallel Mod Sort is $O(n[1+\log(Dis(X)+1)]+p)$ units which is Max and Dis-optimal.*
- *In a EREW-P-RAM or CREW-P-RAM with $p = n^{1-\epsilon}$ processors ($0 < \epsilon < 1$) Parallel Mod Sort sorts X in $O((n/p)[1+\log(Dis(X)+1)]+\log p)$ time. The cost of Parallel Mod Sort is $O(n[1+\log(Dis(X)+1)]+p\log p)$ units which is Max and Dis-optimal.*

Proof: Let X be a sequence of length n. The correctness of the algorithm follows from the fact that it terminates at steps 3 or 6 when it has verified that the array is sorted. If the algorithm reaches step 7, it can be verified that

$$\left\lceil \frac{Dis(R)+2}{2} \right\rceil \left\lceil \frac{n}{p} \right\rceil \geq Dis(X) \geq \left\lfloor \frac{Dis(R)-1}{2} \right\rfloor \left\lceil \frac{n}{p} \right\rceil .$$

Thus, the algorithm terminates at step 8 when it merges a partition of the array consisting of $k+1$ sorted subsequences.

We now analyze the time used by the algorithm. Clearly, steps 1, 3, 4, and 6 require constant time in a *CRCW-P-RAM* while they require $O(\log p)$ time in a *EREW-P-RAM* or *CREW-P-RAM*. Since a sequential *Dis*-optimal algorithm is used in steps 2 and 5, the time used in these steps is

$$O((n/p)[1+\log(1+\min\{n/p, Dis(X)\})]) = O((n/p)[1+\log[1+Dis(X)]).$$

Thus, if the algorithm finishes at steps 3 or 5, the theorem follows. It can be verified that Altman and Chlebus's algorithm for computing *Dis* can be extended to an algorithm with exclusive writing provided that the paralllel computation of prefixes for associative operations is performed in $\log p$ time. Thus step 7 can be performed in

$$O(\log[Dis(R)+1]) = O(\log[Dis(X)+1])$$

time on a *CRCW-P-RAM* and in

$$O(\log[Dis(R)+1]+\log p) = O(\log[Dis(X)+1]+\log p)$$

time on a *EREW-P-RAM* or on a *CREW-P-RAM*.

If $p < k+1$, then $p \leq Dis(X)$ and the algorithm performs step 8.1. During step 8.1.1 each processor merges $\lceil (k+1)/p \rceil$ sequences each of length no more than $\lceil n/(k+1) \rceil$. Thus, this takes $O((n/p)\log\lceil (k+1)/p \rceil) = O((n/p)\log[1+Dis(X)])$ time. Now, by Lemma 2.1, step 8.1.2 takes $O((n/p)\log(p+1)) = O((n/p)\log[Dis(X)+1])$ time ($\log p$ additional time is required on a *CREW-P-RAM* or *EREW-P-RAM* to broadcast the parameters for the merge).

If $p \geq k+1$, by Lemma 2.1, step 8.2 on a *CRCW-P-RAM* takes $O((n/p)\log(k+1))$ for $p = n^{1-\epsilon}$, $0 < \epsilon < 1$. Similarly, on a *EREW-P-RAM* and on an *CREW-P-RAM* this step last $O((n/p)\log(k+1)+\log p)$ time.

From this analysis and the fact that a sequential *Dis*-optimal algorithm sorts in $\Omega(n[1+\log(Dis(X)+1)])$ [9] the theorem follows. □

3 A cost-optimal algorithm for *Runs*

The ideas of the previous section allow us to easily generalize Carlsson and Chen's algorithm [8] to a parallel sorting algorithm that is cost-optimal with respect to *Runs* and adapts to the size of the input. The algorithm can be implemented in all *P-RAM* models.

Parallel Block Sort. Sorts and array $A[0,\ldots,n-1]$ with processors $P_0,\ldots,P_{p-1}$.

1.- *Divide the input into p subarrays.* Let $p-1$ of these subarrays consist of $\lceil n/p \rceil$ elements and the remaining subarray contains $n+1-(p-1)\lceil n/p \rceil (\leq \lceil n/p \rceil)$ elements.

Assign processor P_i to the subarray $A[i\lceil n/p\rceil, \ldots, i\lceil n/p\rceil + \lceil n/p\rceil - 1]$, for $i = 0, \ldots, p-2$, and assign P_{p-1} to the subarray $A[(p-1)\lceil n/p\rceil, \ldots, n]$.

2.- *Sort each subarray.* In parallel, each processor P_i, for $i = 0, \ldots, p-1$, sorts its subarray using a sequential *Runs*-optimal sorting algorithm.

3.- *Compute Runs*(A). In parallel, processor P_i, for $i = 0, \ldots, p-2$, checks if $A[i\lceil n/p\rceil + \lceil n/p\rceil - 1] > A[(i+1)\lceil n/p\rceil]$ and the "prefix sum" of these boolean values is computed. If A is sorted, terminate.

4.- *Merge the runs found in the previous step.* Use *Adaptive Parallel Merge* to merge the runs resulting from steps 1 to 3.

The merit of the algorithm is expressed in the following theorem.

Theorem 3.1 *Let X be a sequence of n elements from a total order.*

- *In a CRCW-P-RAM with $p = n^{1-\epsilon}$ processors $(0 < \epsilon < 1)$ Parallel Block Sort sorts X in $O((n/p)[1 + \log(Runs(X) + 1)] + \log p)$ time. The cost of Parallel Block Sort is $O(n[1 + \log(Runs(X) + 1)] + p\log p)$ units which is Runs-optimal if $Runs = \Omega([p/n]\log p)$.*
- *In a EREW-P-RAM or CREW-P-RAM with $p = n^{1-\epsilon}$ processors $(0 < \epsilon < 1)$ Parallel Block Sort sorts X in $O((n/p)[1 + \log(Runs(X) + 1)] + \log p)$ time. The cost of Parallel Block Sort is $O(n[1 + \log(Runs(X) + 1)] + p\log p)$ units which is Runs-optimal.*

Proof: Let X be a sequence of length n. The correctness of the algorithm follows from the fact that it stops after step 3 when A is sorted or it merges the resulting runs in step 4.

We now analyze the time used by the algorithm. Clearly, step 1 takes constant time on a *CRCW-P-RAM*, while it requires $\log p$ time on a *EREW-P-RAM* or on a *CRCW-P-RAM*. Since a sequential *Runs*-optimal algorithm is used in step 2 and each block $A[i\lceil n/p\rceil, \ldots, i\lceil n/p\rceil + \lceil n/p\rceil - 1]$, for $i = 0, \ldots, p-2$, and $A[(p-1)\lceil n/p\rceil, \ldots, n]$ is a subsequence of the original array, the time used in this step is $O((n/p)[1 + \log[1 + Runs(X)]])$. Now, let X' be the sequence in array A after step 2. A pair of contiguous elements $A[i], A[i+1]$ that delimit a run because $A[i] > A[i+1]$ will be called a "break-point". It is not hard to see that, after step 2, a block $A[i\lceil n/p\rceil, \ldots, i\lceil n/p\rceil + \lceil n/p\rceil - 1]$ has no break-point. All break-points in a block are removed, except for the first and the last which may be translated to positions between blocks. This implies that $Runs(X') \leq 2Runs(X)$. In step 3, the algorithm takes $\log p$ on a *CRCW-P-RAM* and $\log p$ on a *EREW-P-RAM* or on a *CREW-P-RAM*. If the algorithm stops at step 3, the theorem follows. In step 4, *Adaptive Parallel Merge* works with $Runs(X')$ sequences, thus it takes $O((n/p)\log[1 + Runs(X')]) =$

$O((n/p)\log[1 + Runs(X)])$ time. From this analysis and the fact that s sequential *Runs*-optimal algorithm sorts in $\Theta(n(1 + \log[1 + Runs(X)]))$ time [14], the theorem follows. □

4 Final remarks

We have presented adaptive parallel algorithms for optimal sorting with respect to *Dis*, *Max* and *Runs*. Our algorithms are designed for all types of *P-RAMs* with a fixed number of processors $p = n^{1-\epsilon}$ $(0 < \epsilon < 1)$ and can easily be adapted to handle the case $p = n$ (where n is the size of the input). This paper leaves three main lines for further research. First, to our knowledge no adaptive parallel algorithms for optimal sorting with respect to the measure *Inv* (the total number of inversions) and designed for a *P-RAM* with a fixed number p of processors has been presented yet. Second, it would be interesting to design adaptive parallel algorithms for optimal sorting with respect to several measures of disorder. Finally, we have discussed adaptivity when we have a sublinear number of processors. However, the following problem suggested by Akl [1] remains open: are there pallelel algorithm for sorting which, for some $0 \leq u \leq 1$, use $O(n/\log^{u-1} n)$ processors and run in $O(\log^u n)$ time?. We would be interested to know if such algorithms could profit from existing order in the input. If in this case the cost of discovering that the input is sorted is the same as sorting, then there is little hope for adaptive algorithms.

References

[1] S. G. Akl. Optimal parallel algorithms for computing convex hulls and for sorting. *Computing*, 33(1):1–11, 1984.

[2] S. G. Akl. *Parallel Sorting Algorithms.* Notes and Reports in Computer Science and Applied Mathematics. Academic Press, Orlando, Florida, 1985.

[3] S. G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1989.

[4] S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, C-36(11):1367–1369, November 1987.

[5] T. Altman and B.S. Chlebus. Sorting roughly sorted sequences in parallel. *Information Processing Letters*, 33:297–300, 1989/90.

[6] T. Altman and Y. Igarashi. Roughly sorting: Sequential and parallel approach. *J. of Information Processing*, 12:154–158, 1989.

[7] G. Bilardi and A. Nicolau. Adaptive biotonic sorting: an optimal parallel algorithm for shared-memory machines. *SIAM Journal on Computing*, 18(2):216–228, 1989.

[8] S. Carlsson and J. Chen. An optimal parallel adaptive sorting algorithm. *Information Processing Letters*, 39:195–200, 1991.

[9] V. Estivill-Castro. *Sorting and Measures of Disorder.* PhD thesis, University of Waterloo, 1991. Available as Department of Computer Science Research Report CS-91-07.

[10] V. Estivill-Castro and D. Wood. A survey of adaptive sorting algorithms. Technical Report CS-91-64, Department of Computer Science, University of Waterloo, 1991.

[11] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, Cambridge, 1988.

[12] C. Levcopoulos and O. Petersson. An optimal parallel algorithm for sorting presorted files. In *In proceedings of the 8th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 154–160. Springer-Verlag Lecture Notes in Computer Science 338, 1988.

[13] C. Levcopoulos and O. Petersson. A note on adaptive parallel sorting. *Information Processing Letters*, 33:187–191, 1989.

[14] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34:318–325, 1985.

[15] R. McGlinn. A parallel version of Cook and Kim's algorithm for presorted lists. *Software — Practice and Experience*, 19(10):917–930, October 1989.

[16] K. Mehlhorn. *Data Structures and Algorithms, Vol 1: Sorting and Searching.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin/Heidelberg, 1984.

[17] O. Petersson. *Adaptive Sorting.* PhD thesis, Lund University, Department of Computer Science, 1990.

[18] C.C. Ribeiro. Parallel computer models and combinatorial algorithms. *Annals of Discrete Mathematics*, 31:325–364, 1987.