アルゴリズム 53－4
（1996. 9. 13）

仕事・時間量について最適なPRAM上のkマージアルゴリズム

林達也 中野浩嗣
名古屋工業大学 電気情報工学科

ステファン・オラリウ
オールドドミニオン大学 計算機科学科

要素数が合計がnのソートされたk個の列をマージして新しいソート列を求める問題をkマージ問題と呼ぶ。本論文では、単純で仕事・時間量が最適な3つのPRAM上のkマージ問題を解くアルゴリズムを示す。まず、EREW-PRAM上で、$O(\log n)$時間で仕事量が$O(n \log k)$のkマージアルゴリズムと、CREW-PRAMとCRCW-PRAM上で$O(\log\log n + \log k)$時間で仕事量が$O(n \log k)$のkマージアルゴリズムを示す。また、これらのアルゴリズムが仕事量が$O(n \log k)$である限り、高速化はできないことを示す。

Work-Time Optimal k-merge Algorithms on the PRAM †

Tatsuya Hayashi, Koji Nakano
Dept. of Electrical and Computer Engineering
Nagoya Institute of Technology
Showa-ku, Nagoya 466, JAPAN
{hayashi,nakano}@elcom.nitech.ac.jp

Stephan Olariu
Department of Computer Science
Old Dominion University
Norfolk, Virginia 23529, USA
olariu@cs.odu.edu

The k-merge problem, given a collection of k, $(2 \leq k \leq n)$, sorted sequences of total length n, asks to merge them into a new sorted sequence. The main contribution of this work is to propose simple and intuitive work-time optimal algorithms for the k-merge problem on three PRAM models. Our k-merge algorithms runs in $O(\log n)$ time and performs $O(n \log k)$ work on the EREW-PRAM. and in $O(\log\log n + \log k)$ time and $O(n \log k)$ work both on the CREW-PRAM and on the CRCW-PRAM. We also prove that the computing time of these algorithms cannot be improved provided that the amount of work is bounded by $O(n \log k)$.

1 Introduction

The problem of merging k sorted sequences is, along with sorting and usual merging, one of the ubiquitous tasks in computer science. Just as merging, the k-merge problem finds applications to databases, information retrieval, and query processing [10, 19]. In addition, the k-merge problem has a lot of theoretical appeal since it provides a natural bridge between merging, corresponding to the case $k = 2$, and sorting where one merges $k = n$ single-item sequences. In the light of this, it is somewhat surprising that the k-merge problem has not received the attention it deserves. In particular, until very recently, only suboptimal parallel algorithms for the k-merge problems were available [11, 18].

Consider a parallel algorithm that solves an instance of size n of some problem in time $T_p(n)$, with p standing for the number of processors used. Traditionally, the main complexity measure for assessing the performance of the algorithm is the *work* $W(n)$ performed, defined as the product $p \times T_p(n)$. The algorithm is termed *work-optimal* if $W(n) \in \Theta(T^*(n))$, where $T^*(n)$ is the running time of the fastest *sequential* algorithm for the problem. An algorithm is termed *work-time optimal* [13] if it is work-optimal and, in addition, its running time $T_p(n)$ is best possible among the work-optimal algorithms in that model. Needless to say that one of the challenges of parallel algorithm design is to produce not only work-optimal but, indeed, whenever possible, *work-time* optimal algorithms. Occasionally, an even stronger complexity metric is being used – the so-called *time-optimality*. Specifically, an algorithm is termed time-optimal within a model if no other parallel algorithm solving the same problem and using a polynomial number of processors can run faster in that model. To anticipate, one of the algorithms that we design will turn out to be time-optimal.

Clearly, the naive approach of solving the k-merge problem by repeated pairwise merging may result, at best, in a work-optimal algorithm running in $O(\log n * \log k)$ time, which is not work-time optimal [12]. Quite recently, Wen [19] proposed an interesting and sophisticated work-optimal k-merge algorithm. Specifically, his algorithm merges a collection of k, $(2 \leq k \leq n)$, sequences of total length n in $O\left(\frac{n \log k}{p} + \log n\right)$ time, using p processors on the CREW-PRAM. In particular, for $p = \frac{n \log k}{\log n}$ his algorithm achieves a running time of $O(\log n)$. In spite of being work-optimal, Wen's algorithm is less than perfect, as we now point out. For one thing, his k-merge algorithm is not work-time optimal: it is well-known that the problem of merging two sorted sequences of combined length n can be solved in $O(\log\log n)$ time using $O(n)$ work on the CREW-PRAM [4]. For another, Wen's algorithm is complicated, relying on an intricate pipelining technique similar to that in Cole's merge-sort algorithm [7].

Our main contribution is to propose simple and in-

†Work supported, in part, by NSF grant CCR-9522093, by ONR grant N00014-95-1-0779, and by Grant-in-Aid for Encouragement of Young Scientists(08780265).

tuitive work-time optimal algorithms for the k-merge problem on three PRAM models, thus settling the status of the k-merge problem. Specifically, we begin by devising a k-merge algorithm running in $\Theta(\log n)$ time and performing $\Theta(n \log k)$ work on the EREW-PRAM. We then go on to show that this algorithm is work-time optimal and, in fact, even time-optimal. Next, we prove that $\Omega(\log\log n + \log k)$ time is required for solving the k-merge problem on the CREW-PRAM, provided that the amount of work is bounded by $O(n \log k)$. As it turns out, the same lower bound holds for the CRCW-PRAM. Finally, we design a work-time optimal CREW-PRAM k-merge algorithm that runs in $\Theta(\log\log n + \log k)$ time and performs $O(n \log k)$ work. This latter algorithm is also work-time optimal on the CRCW-PRAM model.

At the heart of our algorithms lies a novel deterministic sampling scheme reminiscent of the one developed recently by Olariu and Schwing [15]. The main feature of our sampling scheme is that, when used for bucket sorting, the resulting buckets are well balanced, making costly rebalancing unnecessary.

To put our contribution in perspective, we note that our k-merge algorithms improve on Wen's algorithm in several respects: first, we solve the problem in a weaker model of computation (EREW-PRAM); second, our algorithms are work-time optimal, while Wen's algorithm is not; third, our approach is simple and does not rely on a complicated pipelining strategy. As the results in this work were being written, we became aware of a similar and independent effort by Chen *et al.* [6]. As it turns out, our algorithms are simpler and more intuitive than the algorithm in [6].

The remainder of this paper is organized as follows. Section 2 states the problem formally and provides a review of basic results that will be used in subsequent sections. Section 3 offers a number of non-trivial lower bounds for the k-merge problem both sequentially and in parallel. Section 4 describes our sampling scheme that underlies the proposed optimal k-merge algorithms. Section 5 presents the details of our time- and work-optimal algorithm on the EREW-PRAM; Section 6 presents the work-time optimal algorithm for the CREW-PRAM and for the CRCW-PRAM.

2 Problem statement and background

Consider a collection A of n items consisting of k, $(2 \leq k \leq n)$, sorted sequences $A_1, A_2, \ldots, A_k$. The *k-merge problem* is to merge $A_1, A_2, \ldots, A_k$ into a new sorted sequence. The k-merge problem is fundamental, since it provides a common generalization of the well known *merging* problem, corresponding to $k = 2$, and of the problem of *sorting*, in case $k = n$.

For definiteness, we write for every i, $(1 \leq i \leq k)$, $A_i = \langle a_{i,1}, a_{i,2}, a_{i,2}, \ldots, a_{i,n_i} \rangle$. We note that, in general, each of the k sequences may have a different number of items. We assume, without loss of generality, that the items in A are distinct: should this not be the case, we convert each item $a_{i,j}$ in A_i to the triple $(a_{i,j}, i, j)$. Clearly, all the resulting triples are distinct.

The *prefix computation* problem turned out to be one of the basic techniques in parallel processing, being a key ingredient in many algorithms. The problem is stated as follows: given an associative binary operation $\circ$ and a sequence $x_1, x_2, \ldots, x_n$ of items, compute all the "sums" of the form x_1, $x_1 \circ x_2$, $x_1 \circ x_2 \circ x_3$, $\ldots$, $x_1 \circ x_2 \circ \ldots \circ x_n$. In many contexts one is interested in prefix sums (i.e. $\circ$ operation is addition), or in prefix maxima, etc. Cole and Vishkin [8] showed that the prefix computation problem can be solved optimally in parallel. More precisely, they proved the following result.

Proposition 2.1 *The task of computing the prefix sums of an n-item sequence can be performed in $O(\frac{n}{p})$ time using p, $p \leq \frac{n}{\log n}$, processors on the EREW-PRAM.* □

The task of merging two sorted sequences is, along with sorting, one of the fundamental operations in computer science [14]. The following result [12] shows that merging can be performed efficiently in parallel.

Proposition 2.2 *The task of merging two sorted sequences of size n can be performed in $O(\frac{n}{p})$ time using p, $p \leq \frac{n}{\log n}$, processors on the EREW-PRAM.* □

Borodin and Hopcroft [4] showed that merging can be performed faster on the CREW-PRAM, while preserving work-optimality. Specifically, they proved the following result.

Proposition 2.3 *The task of merging two sorted sequences of size n can be performed in $O(\log\log n)$ time and $O(n)$ work on the CREW-PRAM.* □

The first optimal parallel sorting algorithm (actually, a sorting network) was obtained by Ajtai *et al.* [2]. Their sorting network of I/O size n featured a depth of $O(\log n)$. However, the constant hidden in the Big-O was rather forbidding. This motivated Cole [7] to devise a sorting algorithm (not a sorting network) that sorts a sequence of n items in $O(\log n)$ time using $O(n \log n)$ work. For later reference, we now state an equivalent version of Cole's result.

Proposition 2.4 *The task of sorting of n items can be performed in $O(\frac{n}{p} \log n)$ time using p, $1 \leq p \leq n$, processors on the EREW-PRAM.* □

The following classic result, referred to as Brent's scheduling principle [5, 13] asserts that, under fairly general conditions, a parallel algorithm can be simulated by a p-processor algorithm.

Proposition 2.5 *A parallel algorithm that performs $T(n)$ computational steps and performs $W(n)$ work can be translated into a parallel algorithm to solve the same problem running in $\left\lfloor \frac{W(n)}{p} + T(n) \right\rfloor$ parallel steps, whenever the assignment of the p processors to their jobs can be done in constant time.* □

3 Work-time lower bounds for the k-merge problems

The purpose of this section is to establish non-trivial lower bounds, both sequential and parallel, for the k-merge problem.

We begin by establishing a sequential lower bound. Our arguments rely on the well-known $\Omega(n \log n)$ lower bound for sorting n items in the algebraic tree model [1].

Lemma 3.1 *The task of merging k, $(2 \leq k \leq n)$, sorted sequences of total length n requires $\Omega(n \log k)$ sequential time.*

Proof. We plan to derive a contradiction from the assumption that the k-merge problem can be solved in $o(n \log k)$ time.

Consider $\frac{n}{k}$ unsorted sequences $G_1, G_2, \ldots, G_{\frac{n}{k}}$ of k items each[‡]. Clearly, at least $\Omega(k \log k)$ time is required to sort each of the G_i's. Therefore, the task of sorting all the G_i's independently requires $\Omega(n \log k)$ time. However, as we are about to point out, this task can be performed in $o(n \log k)$ time by using the assumption that the k-merge problem can be solved in $o(n \log k)$ time.

For every j, $(1 \leq j \leq k)$, write $G_j = \{g_{j,1}, g_{j,2}, \ldots, g_{j,k}\}$ and construct a collection $A_1, A_2, \ldots, A_k$ of sequences of $\frac{n}{k}$ items each, such that for every i, $(1 \leq i \leq k)$, $A_i = \{(1, g_{1,i}), (2, g_{2,i}), \ldots, (\frac{n}{k}, g_{\frac{n}{k},i})\}$. Notice that each sequence A_i is sorted in lexicographic order of its items. By assumption, there exists a k-merge algorithm that sorts $A = A_1 \cup A_2 \cup \cdots \cup A_k$ in lexicographical order in $o(n \log k)$ time. In the resulting sorted sequence, the items in each G_j must occur consecutively and in sorted order. Therefore, all the G_j's can be sorted independently in $o(n \log k)$ overall time, a contradiction. □

Next, we show that the sequential lower bound of Lemma 3.1 is tight by exhibiting a optimal sequential algorithm for the k-merge problem running in $\Theta(n \log k)$ time.

Lemma 3.2 *The task of merging k, $(2 \leq k \leq n)$, sorted sequences of total length n can be performed in $\Theta(n \log k)$ sequential time.*

Proof. Consider k sorted sequences $A_1, A_2, \ldots, A_k$, and write $A_i = \langle a_{i,1}, a_{i,2}, \ldots, a_{i,n_i} \rangle$ with $a_{i,1} \leq a_{i,2} \leq \cdots \leq a_{i,n_i}$. Begin by constructing a bottom-heavy heap containing the items $a_{1,1}, a_{2,1}, \ldots a_{k,1}$. Clearly, this can be done in $O(k)$ time [1]. Let $a_{j,1}$ be the minimum. Remove $a_{j,1}$ and add $a_{j,2}$ to the heap. In other words, remove the minimum from the heap, pick a new item from the sequence to which the removed item belongs, and add it to the heap. It is easy to see that all the items are removed by iterating this procedure $O(n)$ times. Moreover, the order of removal corresponds to the sorted order. Since each iteration needs $O(\log k)$ time to maintain the heap, the k-merge can be performed in $O(k + n \log k) = O(n \log k)$ time which, by Lemma 3.1, is optimal. □

By Proposition 2.5, a $o(n \log k)$-work parallel algorithm on any PRAM model yields, by simulation, a $o(n \log k)$-time sequential algorithm. Thus, Lemma 3.1 implies the following result.

Corollary 3.3 *The task of merging k, $(2 \leq k \leq n)$, sorted sequences of total length n requires $\Omega(n \log k)$ work on the PRAM.* □

Next, we establish a time lower bound for the k-merge problem on the EREW-PRAM.

Lemma 3.4 *The task of merging k, $(2 \leq k \leq n)$, sorted sequences of total length n requires $\Omega(\log n)$ time on the EREW-PRAM.*

[‡]To simplify the notation, in the remainder of this work we shall omit the ceiling and floor operators.

Proof. A time lower bound of $\Omega(\log n)$ for merging two sorted sequences of total length n on the EREW-PRAM follows from a fundamental result of Snir [16]. Consequently, $\Omega(\log n)$ must be a time lower bound for the k-merge problem as well. □

Further, we show a non-trivial lower bound on the time required to solve the k-merge problem on the CREW-PRAM within work-optimality.

Lemma 3.5 *Any CREW-PRAM algorithm that performs $O(n \log k)$ work requires $\Omega(\log \log n + \log k)$ time to merge k, $(2 \leq k \leq n)$, sorted sequences of total length n.*

Proof. Since the task of merging two sequences of size n, within work-optimality, has a time lower bound of $\Omega(\log \log n)$ on the CREW-PRAM [4, 13, ?], the same time lower bound holds for the k-merge problem.

On the other hand, by using the k-merge algorithm, the OR of k bits can be computed by considering each bit as a one-element sequence and by merging the resulting sequences. It is well known [9] that the task of solving the OR problem has a time lower bound of $\Omega(\log k)$ on the CREW-PRAM, regardless of the number of processors and memory cells available. Thus, the k-merge problem inherits the same time lower bound. The conclusion follows. □

Finally, we exhibit a non-trivial lower bound on the running time of a CRCW-PRAM algorithm solving the k-merge problem within work-optimality. For this purpose, we rely on the lower bound on the amount of work needed for sorting n items on the parallel comparison model. Specifically, in [3] it is shown that for the task of sorting of n items, $\Omega(Tn^{1+1/T})$ work is required to achieve T, $(T \leq \log n)$, parallel time on the parallel comparison model. Since the parallel comparison model is more powerful than the PRAM for comparison problems, this lower bound holds for the CRCW-PRAM. Thus, we have the following result.

Lemma 3.6 *Any CRCW-PRAM algorithm that performs $O(n \log k)$ work requires $\Omega(\log \log n + \log k)$ time to merge k, $(2 \leq k \leq n)$, sorted sequences of total length n.* □

Proof. Since the task of merging two sequences of size n on the CRCW-PRAM, within work-optimality, has a time lower bound of $\Omega(\log \log n)$ [17], the same lower bound holds for the k-merge problem. As shown in the proof of Lemma 3.4, the task of sorting of k sequences of length $\frac{n}{k}$ items each can be reduced to the k-merge problem; on the other hand, $\Omega(Tk^{1+1/T})$ work is required to sort k items using T, $(T \leq \log k)$, time. Therefore, $\Omega(Tnk^{1/T})$ work is required to solve the k-merge problem in $O(T)$, $(T \leq \log k)$ time. Since $Tnk^{1/T} = \omega(n \log k)$ for every $T \in o(\log k)$, $\Omega(\log k)$ time is required to solve the k-merge problem if the amount of work is bounded by $O(n \log k)$. The conclusion follows. □

It is interesting to note that the lower bounds of Lemmas 3.5 and 3.6 only hold for algorithms performing within work-optimality. For example, it is straightforward to design a CREW-PRAM algorithm that solves the k-merge problem in $O(\log k)$ time if n^2 processors are available. Similarly, Cole argues [7] the

k-merge problem can be solved in $O(\log k/\log\log k)$ time using n^2 CRCW-PRAM processors.

4 Our sampling scheme

The main goal of this section is to present our sampling scheme that is key in designing our k-merge algorithms.

Given a collection A of k, $(2 \leq k \leq n)$, sorted sequences $A_1, A_2, \ldots, A_k$ of total length n, we write for every i, $(1 \leq i \leq k)$, $A_i = \langle a_{i,1}, a_{i,2}, a_{i,2}, \ldots, a_{i,n_i}\rangle$. Let s be a positive integer. We begin by extracting a sample of size $\lfloor \frac{n_i}{s} \rfloor$ by retaining every s-th item in each A_i. We let $sample_s(A_i)$ denote the corresponding sample and write $sample_s(A_i) = \langle a_{i,s}, a_{i,2s}, a_{i,3s}, \ldots, a_{i,s\lfloor n_i/s \rfloor}\rangle$.

Writing $e = \sum_{i=1}^{k} \lfloor \frac{n_i}{s} \rfloor$, we let $sample_s(A) = \langle s_1, s_2, \ldots, s_e\rangle$ be the sequence obtained by *sorting* the set $sample_s(A_1) \cup sample_s(A_2) \cup \cdots \cup sample_s(A_k)$ of samples extracted from all the A_i's.

For later reference, we now state the following technical result,

Lemma 4.1 *For every integer k, $k \neq 0$, we have $\lfloor \frac{e}{k} \rfloor = \lfloor \frac{n}{sk} \rfloor$.*

Consider, again, the collection A of k, $(2 \leq k \leq n)$, sorted sequences $A_1, A_2, \ldots, A_k$ of combined length n, and assume that for every i, $(1 \leq i \leq k)$, we have extracted $sample_s(A_i)$ and have sorted the set of resulting samples to obtain $sample_s(A)$ as discussed above. From the sorted sequence $sample_s(A)$ we extract a new sample, denoted $sample_k(sample_s(A))$, by retaining every k-th item in $sample_s(A)$. By construction, and by Lemma 4.1, $sample_k(sample_s(A))$ contains $\lfloor \frac{e}{k} \rfloor = \lfloor \frac{n}{sk} \rfloor$ items from A. Clearly, we have $sample_k(sample_s(A)) = \langle s_k, s_{2k}, \ldots, s_{\lfloor \frac{n}{sk} \rfloor k}\rangle$. To avoid handling boundary conditions we write $s_0 = -\infty$ and $s_{(\lfloor \frac{n}{sk} \rfloor + 1)k} = +\infty$.

Next, having obtained $sample_k(sample_s(A))$, we proceed to partition the set A into $\lfloor \frac{n}{sk} \rfloor + 1$ buckets $C_0, C_1, \ldots, C_{\lfloor \frac{n}{sk} \rfloor}$ such that for every j, $(0 \leq j \leq \lfloor \frac{n}{sk} \rfloor)$,

$$C_j = \{a \in A | s_{jk} < a \leq s_{(j+1)k}\}. \qquad (1)$$

In other words, we place into bucket C_j all items in A larger than s_{jk} and smaller than or equal to $s_{(j+1)k}$, i.e. $C_j = A \cap (s_{jk}, s_{(j+1)k}]$. It is easy to confirm that, by virtue of (1), all the buckets obtained are disjoint, and that every item of A belongs to exactly one such bucket. We refer the reader to Figure 1 for an illustration of our sampling scheme.

Our next result shows that the sampling scheme we just described results in buckets that are surprisingly well balanced.

Lemma 4.2 *No bucket C_j, $(0 \leq j \leq \lfloor \frac{n}{sk} \rfloor)$, contains more than $2sk$ items of A.*

Proof. If $sample_s(A_i) \cap (s_{jk}, s_{j(k+1)}]$ is empty, then each block $A_i \cap (s_{jk}, s_{j(k+1)}]$ contains at most $s-1$ items: this is because no group of s consecutive items from A_i contains more than one item in $sample_s(A_i)$. Similarly, if $sample_s(A_i) \cap (s_{jk}, s_{j(k+1)}]$ has exactly one item, then $A_i \cap (s_{jk}, s_{j(k+1)}]$ contains at most $2s-1$ items. More generally, if $sample_s(A_i) \cap (s_{jk}, s_{j(k+1)}]$ has m items, then $A_i \cap (s_{jk}, s_{j(k+1)}]$ contains at most $(m+1)s-1$ items. In other words, m sample items in $sample_s(A_i) \cap (s_{jk}, s_{j(k+1)}]$ correspond to at most $(m+1)s-1$ items in $A_i \cap (s_{jk}, s_{j(k+1)}]$. Since $sample_s(A) \cap (s_{jk}, s_{j(k+1)}]$ has exactly k sample items, it follows that bucket $C_j = A \cap (s_{jk}, s_{j(k+1)}]$ contains at most $2sk$ items from A. This completes the proof. □

In turn, the sampling scheme that we just presented suggests, in quite a natural way, the following generic algorithm for solving the k-merge problem. This algorithm will be instantiated in various ways in subsequent sections to yield work-time optimal algorithms on the PRAM.

Algorithm Basic-k-merge(A, s);

Step 1. For every i $(1 \leq i \leq k)$ compute $sample_s(A_i)$;

Step 2. Merge the k sorted sequences $sample_s(A_1)$, $sample_s(A_2)$, $\cdots$, $sample_s(A_k)$ to obtain $sample_s(A)$;

Step 3. Construct $sample_k(sample_s(A))$ and partition A into $\lfloor \frac{n}{ks} \rfloor + 1$ buckets, each containing no more than $2ks$ items from A;

Step 4. Sort each bucket and concatenate the resulting sorted sequences.

5 The EREW-PRAM k-merge algorithm

Consider a collection A of k, $(2 \leq k \leq n)$, sorted sequences $A_1, A_2, \ldots, A_k$ of total length n. We assume that $\frac{3n \log k}{\log n}$ processors are available. Our time- and work-optimal k-merge algorithm on the EREW-PRAM amounts to the call: Basic-k-merge$(A, \frac{\log n}{\log k})$.

We now present a detailed implementation of the four steps of the algorithm on the EREW-PRAM. Writing for every i, $(1 \leq i \leq k)$, $|A_i| = n_i$, Step 1 can be performed in $O(1)$ time by assigning $\frac{n_i \log k}{\log n}$ processors to each A_i. The total amount of work is clearly bounded by $O(\frac{n \log k}{\log n}) \subseteq O(n \log k)$.

Since $sample_{\frac{\log n}{\log k}}(A)$ contains $\frac{n \log k}{\log n}$ items, it can be sorted (Proposition 2.4) in $O(\log \frac{n \log k}{\log n}) \subseteq O(\log n)$ time and $O(\log n * \frac{n \log k}{\log n}) = O(n \log k)$ work, using the processors available to us. At the end of Step 2, we obtain the sorted sequence $sample_{\frac{\log n}{\log k}}(A) = \langle s_1, s_2, \ldots, s_{\frac{n \log k}{\log n}}\rangle$.

By Lemma 4.1, our sampling scheme yields the sequence

$$sample_k(sample_{\frac{\log n}{\log k}}(A)) = \langle s_k, s_{2k}, \ldots s_{k\frac{n \log k}{k \log n}}\rangle$$

containing $\frac{n \log k}{k \log n}$ items from A. In turn, this sequence is used to generate $\frac{n \log k}{k \log n} + 1$ buckets $C_0, C_1, \ldots, C_{\frac{n \log k}{k \log n}}$. By Lemma 4.2, no bucket contains more than $\frac{2k \log n}{\log k}$

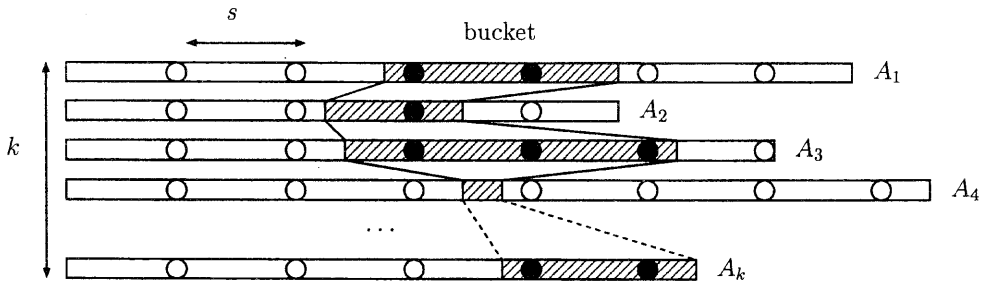

Figure 1: Our sampling scheme

items from A. It is easy to see that the task of constructing $sample_k(sample_{\frac{\log n}{\log k}}(A))$ can be performed in $O(1)$ time, using $\frac{n \log k}{k \log n}$ of the processors available.

To complete Step 3, two issues need to be addressed:

1. first, for each item in A we must identify the bucket to which it belongs;
2. second, we must place the items of A into the corresponding buckets.

The task of determining for each item in A the identity of the bucket to which it belongs is handled as follows. Each A_i is partitioned into blocks, $A_i \cap (-\infty, s_k]$, $A_i \cap (s_k, s_{2k}]$, ..., $A_i \cap (s_{\frac{n \log k}{k \log n} k}, +\infty)$. Recall that for each j, $(0 \leq j \leq \frac{n \log k}{k \log n})$, the items in the k blocks $A_1 \cap (s_{jk}, s_{(j+1)k}]$, $A_2 \cap (s_{jk}, s_{(j+1)k}]$, ..., $A_k \cap (s_{jk}, s_{(j+1)k}]$ belong to bucket C_j. To partition the A_is into blocks, we need to compute the rank of every item in $sample_k(sample_{\frac{\log n}{\log k}}(A))$ with respect to each A_i. For this purpose, we plan to merge $sample_k(sample_{\frac{\log n}{\log k}}(A))$ with each of the A_is. Notice, however, that in order to perform the merging in parallel, k copies of $sample_k(sample_{\frac{\log n}{\log k}}(A))$ must be made beforehand. This latter task can be performed by $\frac{n \log k}{\log n}$ of the processors available in $O(\log k)$ time. The total amount of work used is bounded by $O(\log k * \frac{n \log k}{\log n}) = O(\frac{n \log^2 k}{\log n}) \subseteq O(n \log k)$.

Once the k copies of $sample_k(sample_{\frac{\log n}{\log k}}(A))$ are available, each of them is merged with one of A_1, A_2, ..., A_k. More precisely, we assign to each pair consisting of A_i and of a copy of $sample_k(sample_{\frac{\log n}{\log k}}(A))$, $p_i = \frac{n_i}{\log n} + \frac{n \log k}{k \log^2 n}$ processors. In this arrangement, letting $N_i = |A_i \cup sample_k(sample_{\frac{\log n}{\log k}}(A))|$, we have $p_i \in O(\frac{N_i}{\log N_i})$ and, consequently, by Proposition 2.2, the task of merging A_i and the corresponding copy of $sample_k(sample_{\frac{\log n}{\log k}}(A))$ can be performed by the p_i processors in $O(\log n)$ time and at most $O(n_i \log k)$ work. Therefore, the task of determining the identity of the bucket to which each item of A belongs can be performed in $O(\log n)$ time and $O(n \log k)$ work.

What remains to be done is to move the items in A to their buckets. For this purpose, consider a generic bucket C_j consisting of the items in the k blocks $A_1 \cap (s_{jk}, s_{(j+1)k}]$, $A_2 \cap (s_{jk}, s_{(j+1)k}]$, ..., $A_k \cap (s_{jk}, s_{(j+1)k}]$. To determine the position of each item in bucket C_j we must compute the prefix sums of $|A_1 \cap (s_{jk}, s_{(j+1)k}]|$, $|A_2 \cap (s_{jk}, s_{(j+1)k}]|$, ..., $|A_k \cap (s_{jk}, s_{(j+1)k}]|$. Once this is done, we broadcast the prefix sum to the corresponding blocks, letting each item identify its position within the bucket.

To implement this plan, we shall assign to each bucket C_j containing c_j items $\frac{c_j}{\log n}$ "primary" and k "secondary" processors. To see that this processor assignment is possible, note first that $\sum_{j=1}^{\frac{n \log k}{k \log n}+1} c_j = n$ and so the specified number of primary processors can be assigned; moreover, since there are $\frac{n \log k}{k \log n} + 1$ buckets and at least $\frac{2n \log k}{\log n}$ unassigned processors, it is easy to verify that we can assign k secondary processors to each bucket.

By virtue of Proposition 2.1, the k secondary processors assigned to bucket C_j can compute the corresponding prefix sums in $O(\log k)$ time, with work optimality. The broadcasting task will be handled by the primary processors assigned to bucket C_j in $O(\log n)$ time with optimal work. In addition, the k secondary processors will be employed to move the items in bucket C_j to their corresponding position in $O(\frac{2k \log n}{k \log k}) = O(\frac{\log n}{\log k}) \subseteq O(\log n)$ time. The overall work performed is bounded by $O(n \log k)$.

The task of sorting the buckets in Step 4 is trickier to perform in $O(\log n)$ time. There are, essentially, two sorting strategies that come to mind:

1. sort each bucket from scratch using Cole's algorithm;
2. apply Algorithm Basic-k-merge a second time to sort each bucket.

As it will turn out, we will use both these strategies, depending on the values of k.

To begin, let us investigate the first strategy. Consider a generic bucket C_i, and assume that k (secondary) processors have been assigned to C_i. Recall that by Lemma 4.2, C_i contains no more than $\frac{2k \log n}{\log k}$ items. Now Proposition 2.4 guarantees that

with the available resources, bucket C_i can be sorted from scratch in

$$\begin{aligned} & O\left(\frac{2k\log n}{k\log k}\log\left(\frac{2k\log n}{\log k}\right)\right) \\ = \; & O\left(\frac{\log n}{\log k}\left(1+\log k+\log\frac{\log n}{\log k}\right)\right) \\ = \; & O\left(\log n+\frac{\log n}{\log k}\log\frac{\log n}{\log k}\right) \end{aligned}$$

time. Therefore, as long as

$$k\log k \geq \log n \tag{2}$$

we have

$$\log\frac{\log n}{\log k} \leq \log k$$

and, consequently, the task of sorting bucket C_i can be completed in $O(\log n)$ time and $O(k\log n)$ work. Therefore, if k satisfies (2), the total amount of work involved in sorting the buckets is bounded by $O(\frac{n\log k}{k\log n} * k\log n) = O(n\log k)$.

For values of k satisfying

$$k\log k < \log n \tag{3}$$

the strategy above cannot be used, if we are to restrict the running time to $O(\log n)$. Instead, we shall rely on the second strategy outlined above.

Consider again a generic bucket C_i. The reader should have no difficulty confirming that, by virtue of our sampling scheme, bucket C_i consists, in turn, of k sorted sequences. It is natural, therefore, to apply Algorithm Basic-k-merge to bucket C_i. Along this line of thought, we assume that C_i consists of the k sorted sequences $A_1^i, A_2^i, \ldots, A_k^i$ and that C_i contains a total of c_i ($\leq \frac{2k\log n}{\log k}$) items. This new application of Basic-k-merge corresponds to the call Basic-k-merge(C_i, $\frac{\log n}{k\log k}$). Accordingly, we begin by constructing for every j, $(1 \leq j \leq k)$, the sequences $sample_{\frac{\log n}{k\log k}}(A_j^i)$.

Notice that $sample_{\frac{\log n}{k\log k}}(C_i) = sample_{\frac{\log n}{k\log k}}(A_1^i) \cup sample_{\frac{\log n}{k\log k}}(A_2^i) \cup \cdots \cup sample_{\frac{\log n}{k\log k}}(A_k^i)$ involves a total of at most $c_i \cdot \frac{k\log k}{\log n} \leq \frac{2k\log n}{\log k} \cdot \frac{k\log k}{\log n} = 2k^2$ items. By Proposition 2.4, the k processors assigned to bucket C_i can sort $sample_{\frac{\log n}{k\log k}}(C_i)$ in $O(\frac{k^2}{k}\log k^2) = O(k\log k)$ time. By (3), the running time of Step 2 is bounded by $O(\log n)$, and the amount of work is optimal.

In Step 3, $sample_k(sample_{\frac{\log n}{k\log k}}(C_i))$ is constructed and C_i is partitioned into $k+1$ buckets each containing at most $\frac{4\log n}{\log k}$ items of A. At this moment, each such bucket is being assigned one of the k processors which proceeds to sort the corresponding bucket by performing the sequential k-merge. By Lemma 3.2, this takes $O\left(\frac{\log n}{\log k} * \log k\right) = O(\log n)$ time and optimal work. Thus, irrespective of which of the conditions (2) or (3) holds, the task of sorting the buckets in Step 4 of Algorithm Basic-k-merge can be completed in $O(\log n)$ time and $O(n\log k)$ work.

Once sorting is done, we need to concatenate the (sorted) buckets. We shall only demonstrate how this is done in the case of condition (2), the complementary range being similar. In order to do this, the rank of each item in $sample_k(sample_{\frac{\log n}{\log k}}(A))$ with respect to A must be computed. This can be done by computing the prefix sums of $|C_0|, |C_1|, \ldots, |C_{\frac{n\log k}{k\log n}}|$ and by broadcasting the prefix sum to the corresponding bucket. By using Proposition 2.1 the prefix sums can be computed in $O(\log\frac{n\log k}{k\log n}) \subseteq O(\log n)$ time and $O(\frac{n\log k}{k\log n})$ work, while the broadcasting can be done in $O(\frac{\log n}{\log k})$ time and $O(n)$ work. Since each item knows its rank, it can be moved to the correct position in $O(\frac{\log n}{\log k}) \subseteq O(\log n)$ time and $O(n)$ work. Hence, Step 4 can be completed in $O(\log n)$ time and $O(n\log k)$ work. Therefore, the entire algorithm can be performed in $O(\log n)$ time and $O(n\log k)$ work. By Lemma 3.4 and Corollary 3.3 this is time- and work-optimal.

We summarize our findings by stating the main result of this section.

Theorem 5.1 *The task of merging k, $(2 \leq k \leq n)$, sorted sequences of total length n can be performed in $O(\log n)$ time and $O(n\log k)$ work on the EREW-PRAM. Furthermore, this is both time- and work-optimal on this model.* □

6 The CREW-PRAM k-merge algorithm

The main goal of this section is to exhibit a work-time optimal parallel algorithm solving the k-merge problem in $O(\log\log n + \log k)$ time and $O(n\log k)$ work on the CREW-PRAM.

The input to the algorithm is a collection A of k sorted sequences $A_1, A_2, \ldots, A_k$ of total length n. We write for every i, $(1 \leq i \leq k)$, $|A_i| = n_i$. We assume that $\frac{2n\log k}{\log\log n+\log k}$ processors are available. At this point, we note that if

$$k^2\log n > n\log k, \tag{4}$$

applying the logarithm through, we obtain

$$O(\log n) \subseteq O(\log n + \log\log k) \subseteq O(\log\log n + \log k). \tag{5}$$

Consequently, if (4) holds, we apply Cole's sorting algorithm directly. From now on, we shall assume that

$$k^2\log n \leq n\log k. \tag{6}$$

The algorithm is very close in spirit to the EREW algorithm detailed in the previous section, and amounts to the call Basic-k-merge(A, $\frac{k\log n}{\log k}$).

The remainder of this section is devoted to a detailed implementation of the fours steps of the algorithm on the CREW-PRAM. To begin, Step 1 can be performed in $O(1)$ time by assigning $\frac{n_i\log k}{k\log n}$ processors to each A_i. By Proposition 2.5, the same task can be performed in $O(\log\log n + \log k)$ time by $\frac{n_i\log k}{k\log n(\log\log n+\log k)}$ processors associated with A_i.

To see that this latter processor assignment is feasible, observe that

$$\sum_{i=1}^{k} \frac{n_i \log k}{k \log n(\log\log n + \log k)} = \frac{n \log k}{k \log n(\log\log n + \log k)} < \frac{n \log k}{\log\log n + \log k}.$$

In both cases, the total amount of work is bounded by $O(\frac{n \log k}{k \log n}) \subseteq O(n \log k)$.

The implementation of Step 2 differs substantially from that of Step 2 in the EREW-PRAM implementation of Basic-k-merge. To wit, while the EREW-PRAM implementation uses Cole's optimal sorting, we shall rely, instead, on the doubly-logarithmic merging algorithm of Proposition 2.3. In outline, the idea is to sort $sample_{\frac{k \log n}{\log k}}(A_1) \cup sample_{\frac{k \log n}{\log k}}(A_2) \cup \cdots \cup sample_{\frac{k \log n}{\log k}}(A_k)$ by computing the rank of each item. To implement this idea, we plan to merge, pairwise, $sample_{\frac{k \log n}{\log k}}(A_i)$ and $sample_{\frac{k \log n}{\log k}}(A_j)$ for $1 \leq i < j \leq k$. This, of course, allows each item in a generic $sample_{\frac{k \log n}{\log k}}(A_i)$ to compute its rank with respect to each $sample_{\frac{k \log n}{\log k}}(A_j)$, $i \neq j$. What remains to be done is to add the corresponding ranks. To see how this is done efficiently, let $N_i = |sample_{\frac{k \log n}{\log k}}(A_i)|$ and let $N_j = |sample_{\frac{k \log n}{\log k}}(A_j)|$.

By Proposition 2.3, We can merge $sample_{\frac{k \log n}{\log k}}(A_i)$ and $sample_{\frac{k \log n}{\log k}}(A_j)$ in $O(\log\log(N_i + N_j))$ time and $O(N_i + N_j)$ work. By Proposition 2.5, the same task can be performed in $O(\log\log(N_i + N_j) + \log\log n) = O(\log\log n)$ time using $p_{ij} = \frac{N_i + N_j}{\log\log n}$ processors associated with the pair consisting of $sample_{\frac{k \log n}{\log k}}(A_i)$ and $sample_{\frac{k \log n}{\log k}}(A_j)$. Observe that the total number of processors thus assigned is bounded by:

$$\sum_{1 \leq i < j \leq k} p_{ij} = \frac{\log k}{k \log n \log\log n} \sum_{1 \leq i < j \leq k} (n_i + n_j). \quad (7)$$

Now a simple counting argument shows that

$$\sum_{1 \leq i < j \leq k} (n_i + n_j) = (k-1)n. \quad (8)$$

(To justify (8), note that each n_i occurs exactly $k - i$ times as the first term of a sum and exactly $i - 1$ as the second term of a sum. Thus, altogether, n_i occurs $k - 1$ times, as claimed.)
By virtue of (8), equation (7) becomes

$$\sum_{1 \leq i < j \leq k} p_{ij} = \frac{\log k}{k \log n \log\log n} \sum_{1 \leq i < j \leq k} (n_i + n_j) < \frac{nk \log k}{k \log n \log\log n} < \frac{n \log k}{\log\log n + \log k},$$

confirming that the processor assignment specified above is feasible. Moreover, the total amount of work involved in the merging phase of Step 2 does not exceed is $O(\frac{n \log k}{\log\log n + \log k} * \log\log n) \subseteq O(n \log k)$.

As a result, for each item in $sample_{\frac{k \log n}{\log k}}(A_i)$, the number of items in $sample_{\frac{k \log n}{\log k}}(A_j)$, $i \neq j$, smaller than it can be determined directly. By adding up these values, the rank of each item in $sample_{\frac{k \log n}{\log k}}(A)$ can be obtained. By Proposition 2.1 this task can be performed in $O(\log k)$ time and $O(k)$ work by assigning $\frac{k}{\log k}$ processors to each item in $sample_{\frac{k \log n}{\log k}}(A)$. By Lemma 2.5 the same task can be performed by $\frac{k}{\log\log n + \log k}$ processors in $O(\log\log n + \log k)$ time and optimal work. Therefore, the total amount of work involved in the second phase of Step 2 is bounded by $O(n \log k)$, confirming that Step 2 can be implemented to run in $O(\log\log n + \log k)$ time and $O(n \log k)$ work on the CREW-PRAM. At the end of Step 2, we obtain the sorted sequence $sample_{\frac{k \log n}{\log k}}(A) = \langle s_1, s_2, \ldots, s_{\frac{n \log k}{k \log n}} \rangle$.

By Lemma 4.1, our sampling scheme yields the sequence

$$sample_k(sample_{\frac{k \log n}{\log k}}(A)) = \langle s_k, s_{2k}, \ldots s_{\frac{n \log k}{k^2 \log n} k} \rangle$$

containing $\frac{n \log k}{k^2 \log n}$ items from A. In turn, this sequence is used to generate $\frac{n \log k}{k^2 \log n} + 1$ buckets C_0, C_1, ..., $C_{\frac{n \log k}{k^2 \log n}}$. By Lemma 4.2, no bucket contains more than $\frac{2k^2 \log n}{\log k}$ items from A. It is easy to see that the task of constructing $sample_k(sample_{\frac{k \log n}{\log k}}(A))$ can be performed in $O(1)$ time, using $\frac{n \log k}{k^2 \log n} < \frac{n \log k}{\log\log n + \log k}$ of the processors available.

Similarly to the implementation on the EREW-PRAM, to complete Step 3, we have to address the issues of identifying the bucket to which each item belongs and that of moving the items to their buckets.

First, the task of determining for each item in A the identity of the bucket to which it belongs can be performed by merging in exactly the same way as in the EREW-PRAM implementation: the only difference is that here, because of concurrent read capabilities, we do not have to make extra copies of $sample_k(sample_{\frac{k \log n}{\log k}}(A))$. Specifically, for each i, $(1 \leq i \leq k)$, we plan to merge $sample_k(sample_{\frac{k \log n}{\log k}}(A))$ and A_i. Notice that, by Proposition 2.3, the task at hand can be performed in $O(\log\log(n_i + \frac{n \log k}{k^2 \log n}))$ time and $O(n_i + \frac{n \log k}{k^2 \log n})$ work. By Proposition 2.5, the same task can be performed in $O(\log\log n + \log k)$ time using $\frac{n_i + \frac{n \log k}{k^2 \log n}}{\log\log n + \log k}$ processors and optimal work. To confirm that this latter processor assignment is feasible, we only need observe that for $k \geq 2$,

$$\sum_{i=1}^{k} \left(n_i + \frac{n \log k}{k^2 \log n} \right) = n + \frac{n \log k}{k \log n} \leq 2n \log k.$$

Next, we have to move the items in A to their buckets. In the same way as in the EREW-PRAM implementation, the prefix sums of $|A_1 \cap (s_{jk}, s_{(j+1)k}]|$, $|A_2 \cap (s_{jk}, s_{(j+1)k}]|$, ..., $|A_k \cap (s_{jk}, s_{(j+1)k}]|$ are computed in $O(\log k)$ time. Once this is done, we broadcast the prefix sum to the corresponding blocks, letting

each item identify its position within the bucket. This completes the $O(\log\log n + \log k)$ time and $O(n \log k)$ work implementation of the Step 3.

Finally, in Step 4, the task of sorting the resulting buckets can be performed by employing the optimal k-merge algorithm on the EREW-PRAM discussed in the previous section. More precisely, since each bucket has at most $\frac{2k^2 \log n}{\log k}$ items, Theorem 5.1 guarantees that it can be k-merged in $O(\log(\frac{2k^2 \log n}{\log k})) = O(\log\log n + \log k)$ time and $O(k^2 \log n)$ work. Thus, by Proposition 2.5, the same task can be performed by $\frac{k^2 \log n}{\log\log n + \log k}$ processors in $O(\frac{k^2 \log n}{\frac{k^2 \log n}{\log\log n + \log k}} + \log\log n + \log k) = O(\log\log n + \log k)$ time.

The total number of processors thus assigned is bounded by $(\frac{n \log k}{k^2 \log n} + 1) \cdot \frac{k^2 \log n}{\log\log n + \log k} = \frac{n \log k}{\log\log n + \log k} + \frac{k^2 \log n}{\log\log n + \log k}$. By (4), $\frac{k^2 \log n}{\log\log n + \log k} \leq \frac{n \log k}{\log\log n + \log k}$, confirming that the total number of processors assigned to all the buckets is bounded by $\frac{2n \log k}{\log\log n + \log k}$.

Therefore, the task of sorting the buckets in Step 4 can be performed in $O(\log\log n + \log k)$ time and $O(n \log k)$ work. What remains to be done is to concatenate the sorted buckets into the final sorted sequence. This is done, in the obvious way, in $O(\log\log n + \log k)$ time and $O(n \log k)$ work by computing prefix sums as discussed in the previous section.

Consequently, the algorithm runs in $O(\log\log n + \log k)$ time and performs at most $O(n \log k)$ work on the CREW-PRAM. By Lemma 3.5 and Corollary 3.3 the algorithm is work-time optimal.
To summarize our findings we state the main result of this section.

Theorem 6.1 *The task of merging k, $(2 \leq k \leq n)$, sorted sequences of total length n can be performed in $O(\log\log n + \log k)$ time and $O(n \log k)$ work on the CREW-PRAM.* □

Since every CREW-PRAM algorithm also runs without loss of time on the CRCW-PRAM, the algorithm developed in this section translates into a k-merge algorithm for the CRCW-PRAM. By Lemma 3.6 and Corollary 3.3 combined, this algorithm is work-time optimal. Thus, we have the following result.

Theorem 6.2 *The task of merging k, $(2 \leq k \leq n)$, sorted sequences of total length n can be performed in $O(\log\log n + \log k)$ time and $O(n \log k)$ work on the CRCW-PRAM.* □

References

[1] A. Aho, J. E. Hopcroft, J. Ullman, *Data Structures and Algorithms*, Addison-Wesley, Reading, Massachusetts, 1984.

[2] M. Ajtai, J. Komlos, and E. Szemeredi, Sorting in $c \log n$ parallel steps, *Combinatorica*, **3**, (1983), 1–19.

[3] Y. Azar and U. Vishkin, Tight comparison bounds on the complexity of parallel sorting, *SIAM Journal on Computing*, **16**, (1987), 458–464.

[4] A. Borodin and J. E. Hopcroft, Routing, merging and sorting on parallel models of computation, *Journal of Computer and System Sciences*, **30**, (1985), 130–145.

[5] R. P. Brent, The parallel evaluation of of general arithmetic expressions, *Journal of the ACM*, **21**, (1974), 201–208.

[6] D. Z. Chen, W. Chen, K. Wada, and K. Kawaguchi, Parallel algorithms for partitioning sorted sets and related problems, to appear in ESA96.

[7] R. Cole, Parallel merge sort, *SIAM Journal on Computing*, **17**, (1988), 770–785.

[8] R. Cole and U. Vishkin, Approximate parallel scheduling. Part 1: the basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM Journal on Computing*, **18**, (1988), 128–142.

[9] S. A. Cook, C. Dwork, and R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM Journal on Computing*, **15**, (1986), 87–97.

[10] E. Dekel and I. Ozsvath, Parallel external sorting, *Journal of Parallel and Distributed Computing*, **6**, (1989), 623–635.

[11] J. Y. Fu and F. C. Lin, Optimal parallel external merging under hardware constraints, *Proc. International Conference on Parallel Processing*, St-Charles, Illinois, August 1991, III, 70–74.

[12] T. Hagerup and C. Rub, Optimal merging and sorting on the EREW PRAM, *Information Processing Letters*, **33**, (1989), 181–185.

[13] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, Reading, Massachusetts, 1991.

[14] D. E. Knuth, *The Art of Computer Programming, Vol. 1, Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, Massachusetts, 1973.

[15] S. Olariu and J. Schwing, A new deterministic sampling scheme with applications to broadcast-efficient sorting on the reconfigurable mesh, *Journal of Parallel and Distributed Computing*, **32**, (1996), 215–222.

[16] M. Snir, On parallel searching, *SIAM Journal on Computing*, **14**, (1985), 688–708.

[17] Y. Shiloach and U. Vishkin, Finding the maximum, merging and sorting in parallel computation, *Journal of Algorithms*, **2**, (1981), 88–102.

[18] P. Valduriez and G. Gardarin, Join and semijoin algorithms for multiprocessors database machines, *ACM Transactions on Database Systems*, **9** (1984), 133–161.

[19] Z. Wen, Multi-way merging in parallel, *IEEE Transactions on Parallel and Distributed Systems*, **7**, (1996), 11–17.