# Chapter 2

# Multiway Merging in Parallel

Multiway merging is the problem of merging $k$ sorted arrays into a single sorted array. When the value of $k$ is understood, we shall also call this problem $k$-way merging. This problem can be seen as a generalisation of sorting and merging: the special case of $k = 2$ is ordinary merging, and that of $k = n$ is sorting.

Parallel solutions for the multiway merging problem form the topic of this chapter. An algorithm schema for multiway merging is described in Section 2.1. The implementation of the schema on various PRAM models for the comparison based version of the problem is discussed in Section 2.3; matching lower bounds are also presented. These results improve the existing CREW PRAM upper bound of $O(\log n)$ [97]. Multiway merging of integers is considered in Section 2.4.

## 2.1 The Algorithm Schema

The following definitions are due to Cole [19]. For three keys $x, y$ and $z$, we say that $y$ is *between* $x$ and $z$, or that $x$ and $z$ *straddle* $y$, if $x \leq y < z$. For a key $x$ and a list $L$, the *rank* of $x$ in $L$ is the number of elements in $L$ that are not larger than $x$. A list $L_1$ is said to be *ranked* in another list $L_2$ if every element of $L_1$ knows its rank in $L_2$; $L_1$ and $L_2$ are *cross ranked* if they are ranked into each other.
The algorithm schema is now described:

---

**Input:** Arrays $A_i$ of size $n_i$ for $1 \leq i \leq k$, each sorted in increasing order, where $\sum_{i=1}^{k} n_i = n$; $p$ processors are available. Without loss of generality, we assume that all elements are distinct, because otherwise, for each $i$, the $j$-th element $a_j^i$ of $A_i$, can be replaced by a triplet $\langle a_j^i, i, j \rangle$, with the linear ordering extended lexicographically.
**Output:** $A_1 \bigcup \ldots \bigcup A_k$ sorted in increasing order in an array $B[1 \ldots n]$.

**Step 1:** For a parameter $\lambda$ to be appropriately chosen, if $n < k\lambda$, sort the elements using the fastest available sorting algorithm with $p$ processors; **exit**.

**Step 2:** Partition $A_i$, for $1 \leq i \leq k$, into segments of size $\lambda$ each. Here, by a segment we mean a subarray of consecutive locations. (If $n_i$ is not divisible by $\lambda$, the last segment may be of a smaller size.) There are $t_i = \lceil \frac{n_i}{\lambda} \rceil$ segments in $A_i$. So the total number of segments $t = \sum_{i=1}^{k} t_i \leq k + \frac{n}{\lambda} \leq \frac{2n}{\lambda}$. For each segment, let its first element be called its *leader*. Let an array $A'_i$, for $1 \leq i \leq k$, contain the set of leaders from $A_i$ arranged in increasing order.

**Step 3:** For $1 \leq i, j \leq k$, $i \neq j$, cross-rank the leader arrays $A'_i$ and $A'_j$. Each leader $x$ now has $k$ ranks, $r_i(x)$ for $1 \leq i \leq k$, assigned to it, $r_i(x)$ being the number of elements less than or equal to $x$ in $A'_i$.

**Step 4:** For each leader $x$, in parallel, add together all $r_i(x)$'s. Consequently, each leader knows the total number of smaller leaders over all arrays; that is, now, all leaders can be placed in an array $L$ of size $t$, sorted in increasing order.

**Step 5:** Each leader $x$ knows the two consecutive leaders of $A_i$, for $1 \leq i \leq k$, that straddle $x$; note that when $x \in A_i$, the smaller of the two leaders is $x$ itself. In other words, $x$ knows the segment $s_i(x)$ of $A_i$ to which it belongs. For each $x \in L$ and for each array $A_i$, search $s_i(x)$ in parallel to find the two consecutive elements in it that straddle $x$. Let $R_i(x)$ be the index in $A_i$ of the smaller one of these two. Thus, $R_i(x)$ is the number of elements in $A_i$ smaller than or equal to $x$.

**Step 6:** For each leader $x$, compute $I(x) = \sum_{i=1}^{k} R_i(x)$, which clearly is the index of $x$ in $B$, the output array.

REMARK: For any two consecutive elements $x$ and $y$ of $L$, there can be at most $\lambda$ elements in each $A_i$, $1 \leq i \leq k$, that $x$ and $y$ straddle. That is, $R_i(y) - R_i(x) \leq \lambda$, for all $i$. Hence, $I(y) - I(x) \leq k\lambda$.

**Step 7:** For every pair of consecutive elements $x$ and $y$ of $L$, let

$$\delta_i(x) = \begin{cases} R_i(y) - R_i(x) & \text{when } y \notin A_i \\ R_i(y) - R_i(x) - 1 & \text{when } y \in A_i \end{cases}$$

Note that $\delta_i(x)$ is the number of elements that are larger than $x$ but smaller than $y$ in $A_i$.

Let $\langle \Delta_1(x), \ldots, \Delta_k(x) \rangle$ be the prefix sums computed over $\langle \delta_1(x), \ldots, \delta_k(x) \rangle$; $\Delta_0(x)$ is set to zero.

**Step 8:** Copy $A_i[R_i(x)+1 \ldots R_i(x)+\delta_i(x)]$ to $S_x[\Delta_{i-1}(x)+1 \ldots \Delta_i(x)]$. That is, we isolate all elements that are larger than $x$ but smaller than $y$ into an array $S_x$.

**Step 9:** Sort the elements of $S_x$ for each $x \in L$, in parallel, using $p$ processors overall, and place the output in consecutive locations of the subarray of $B$ that begins at index $I(x)+1$.

REMARK: Note that $N(x) =\mid S(x) \mid = I(y) - I(x) - 1 < k\lambda$.

## 2.2 Preliminaries

The time needed to solve instances of size $n$, respectively of, prefix summation of $b$ bit numbers, merging, searching, sorting and $k$-way merging, using $p$ processors in each case, will be denoted by $\text{PREFIX}(n,p,b)$, $\text{MERGE}(n,p)$, $\text{SEARCH}(n,p)$, $\text{SORT}(n,p)$ and $k\_\text{WAY\_MERGE}(n,p)$.

We use in the sequel, the following upper bounds that are well known in literature.

**Upper Bound 2.1** *[55] When comparison is the only operation allowed on the keys, on a CREW PRAM,* $\text{MERGE}(n,p) = O(\log\log n + \frac{n}{p})$.

**Upper Bound 2.2** *[55] When comparison is the only operation allowed on the keys, with $p > n$, on a CREW PRAM,* $\text{MERGE}(n,p) = O(\log \frac{\log n}{\log(p/n)})$.

**Upper Bound 2.3** *[78] When the keys are integers drawn from the range $[0 \ldots m-1]$, on a CREW PRAM,* $\text{MERGE}(n,p) = O(\log\log\log m + \frac{n}{p})$.

**Upper Bound 2.4** *[78] When the keys are integers drawn from the range $[0 \ldots n-1]$, on a CRCW PRAM,* $\text{MERGE}(n,p) = O(\alpha(n) + \frac{n}{p})$.

**Upper Bound 2.5** *[46] On an EREW PRAM,* $\text{MERGE}(n,p) = O(\log n + \frac{n}{p})$.

**Upper Bound 2.6** *[22, 91, 41] On a CRCW PRAM,*

$$\text{PREFIX}(n,p,b) = O\left(\min\left\{\log\frac{b}{\log p}, \log n\right\} + \frac{\log n}{\log\log p} + \frac{n}{p}\right).$$

**Upper Bound 2.7** *[55] On an EREW PRAM,* $\text{PREFIX}(n,p,b) = O(\log n + \frac{n}{p})$.

**Upper Bound 2.8** *[55] On a CREW PRAM,* $\text{SEARCH}(n,p) = O(\frac{\log n}{\log p})$.

**Upper Bound 2.9** *[3, 81, 19] When comparison is the only operation allowed on the keys, on an EREW PRAM,*

$$\text{SORT}(n,p) = O\left(\log n + \frac{n\log n}{p}\right).$$

**Upper Bound 2.10** *[45] When comparison is the only operation allowed on the keys, on a CRCW PRAM,*

$$\text{SORT}(n,p) = O\left(\frac{\log n}{\log(p/n)}(\log\log(p/n))^5 2^{O(\log^* n - \log^*(p/n)+1)} + \frac{\log n}{\log\log p}\right).$$

**Upper Bound 2.11** *[7] When the keys can be treated as integers, on an ARBITRARY CRCW PRAM,* $\text{SORT}(n,p) = O(\log n + \frac{n\log\log n}{p})$.

**Upper Bound 2.12** *[12] When the keys are integers drawn from the range $[0 \ldots m-1]$, on an ARBITRARY CRCW PRAM,* $\text{SORT}(n,p) = O(\frac{\log n}{\log\log n} + \log\log m + \frac{n\log\log m}{p})$.

**Upper Bound 2.13** *[55] The minimum of $n$ numbers can be found in $O(1)$ time, with $O(n^{1+\epsilon})$ processors, on a COMMON CRCW PRAM, for any fixed $\epsilon > 0$.*

Following [40], we say, a set of $n$ objects drawn from a linearly ordered set are padded-sorted if they are arranged in an array of size $O(n)$ in non-decreasing order.

**Upper Bound 2.14** *[45] A set of $n$ integers drawn from the range $[0 \ldots n-1]$ can be stably padded-sorted with a constant padding factor in $O((\log n)^{1/2}(\log\log n)^{3/2})$ time and $O(n \log\log n)$ operations on an ARBITRARY CRCW PRAM.*

We shall encounter several times in the course of this chapter, the following allocation problem $\mathcal{R}$. Given an array $A = \langle n_1, \ldots, n_k \rangle$, where $\sum_{i=1}^{k} n_i = n$, fill an array $P[1 \ldots n]$ so that, for $1 \leq i \leq n$, $P[i] = j$ iff $\sum_{x=0}^{j-1} n_x < i \leq \sum_{x=0}^{j} n_x$, where $n_0 = 0$.

When $p$ processors are available, the problem $\mathcal{R}$ can be solved as follows. Find the prefix sums $\langle \eta_1, \ldots, \eta_k \rangle$ of $A$. Let $B[i] = \langle i, 1 \rangle$, and $C[j] = \langle \eta_j, 2 \rangle$ for $1 \leq i \leq n$ and $1 \leq j \leq k$. Cross rank $B$ and $C$. Now, $P$ can be filled as required in the problem: if the rank of $B[i]$ in $C$ is $j$ then $P[i] = j$. Hence we have the lemma below:

**Lemma 2.1** *The problem $\mathcal{R}$ can be solved in $O(\textsc{Prefix}(k, p, \log n) + \textsc{Merge}(n, p))$ time.*

We assume that on all computational models under consideration here, the upper bounds for merging, prefix summing and sorting satisfy the the "regularity" condition that larger problem instances with the same processor-advantage $(p/n)$ will take more time. That is, $\textsc{Merge}(n_1, n_1\beta) \leq c\textsc{Merge}(n_2, n_2\beta)$, for $n_1 < n_2$ and a constant $c > 0$; similarly for sorting and prefix summing.

## 2.3 Comparison Based Multiway Merging

### 2.3.1 The schema on PRAMs with Concurrent Reads

First, we consider some aspects common to CREW and CRCW PRAM implementations of the algorithm schema presented in Section 2.1. Let us set $\lambda = k^2$.

In Step 1, where we are sorting all the elements together, $n < k^3$. Since, solving a larger problem instance with the same processor advantage cannot be easier, the time taken by this step is $O(\textsc{Sort}(n, p)) = O(\textsc{Sort}(k^3, \frac{pk^3}{n}))$.

We can both partition the input arrays and form the leader arrays in $O(1)$ time, using $n$ processors, once they are assigned one per element. But here we have $p$ processors. Assign one processor for every $\Theta(\frac{n}{p})$ elements of the input, using Lemma 2.1. Each processor can now sequentially select leaders from the elements assigned to it. So, Step 2 takes $O(\frac{n}{p} + \textsc{Merge}(n, p) + \textsc{Prefix}(k, p, \log n))$ time.

In Step 3, $\frac{k(k-1)}{2}$ merges, each of size at most $t$, are to be performed in parallel. Assign $\Theta(\frac{p}{k^2})$ processors per merge instance. Solve each instance independently. The time taken is clearly $O(\textsc{Merge}(t, \frac{p}{k^2}))$.

Steps 4 and 6 each requires every leader to perform an addition of size $k$, of $\log n$ bit numbers. Use Lemma 2.1 to assign $\Theta(\frac{p}{t})$ processors per leader. Find the sums using a prefix summation algorithm in $O(\textsc{Prefix}(k, \frac{p}{t}, \log n))$ time. The total time taken is $O(\textsc{Merge}(n, p) + \textsc{Prefix}(k, \frac{p}{t}, \log n))$.

Each leader has to perform $k$ searches in Step 5, each over a segment of size $k^2$; that is, a total of $tk$ searches have to be performed in parallel. At most $\lfloor \frac{p}{tk} \rfloor$ processors are available per search. If $k$ processors were to be used per search, the time taken would be $\textsc{Search}(k^2, k)$, which is $O(\frac{\log k^2}{\log k}) = O(1)$, by Upper Bound 2.8. With $\lfloor \frac{p}{tk} \rfloor$ processors per search, hence, we can finish the step in $O(\frac{k}{p/tk} + \textsc{Search}(k^2, k))$ time. As $tk^2 < 2n$ (see Step 2 of the schema), the time needed for Step 5 is $O(\frac{n}{p} + \textsc{Search}(k^2, k)) = O(\frac{n}{p})$.

Step 7 requires every leader to perform an addition of size $k$, of $\log k$ bit numbers. With a processor allocation similar to that in Step 4, the sums can be found in $O(\textsc{Prefix}(k, \frac{p}{t}, \log k))$ time. Since the leaders are now merged into the array $L$, the allocation incurs only $O(1)$ time.

In Step 8, we prepare the sorting instances. Each leader $x$, for $1 \leq i \leq k$, if $\delta_i(x) \neq 0$, then tags element $A_i[R_i(x) + 1]$ with the address of the location $S_x[\Delta_{i-1}(x) + 1]$; note that $A_i[R_i(x) + 1]$ is to be copied to $S_x[\Delta_{i-1}(x) + 1]$. Assign $\Theta(\frac{p}{t})$ processors to each segment as in Step 7; and for each element find the largest tagged element smaller than it in its array. As the second element in the segment is guaranteed to be tagged, the first being a leader, this does not require any inter-segment information access; a prefix minima over each segment, done in parallel, will do. Thus, each element can compute the address of the cell it must be copied to. Finally, the copying itself can be done in $O(\frac{n}{p})$ time. So, the total time taken is $O(\frac{n}{p} + \textsc{Prefix}(k^2, \frac{p}{t}, 2\log k))$.

In Step 9, we again assign $\Theta(\frac{p}{t})$ processors per segment, as in Step 7. For each leader $x$, dedicate to $x$, a fraction $\Theta(\frac{\delta_i(x)}{k^2})$ of the processors assigned to $s_i(x)$, the segment of $A_i$ to which $x$ belongs, by carrying out another prefix summation over each segment of $A_i$, for $1 \leq i \leq k$. That is each leader $x$ is now assigned $\Theta(\frac{N(x)p}{n})$ processors. Since the prefix summation can be clubbed with that of Step 8, this step can be finished in $O(\textsc{Sort}(k^3, \frac{pk^3}{n}))$ time.

On a PRAM that allows concurrent reads, $\textsc{Sort}(m,q)$, $\textsc{Prefix}(m,q,b)$ and $\textsc{Merge}(m,q)$ are all at most $O(\frac{m}{q}+\log m)$. Hence, for a constant $c$,

$$\textsc{Sort}(m^{c+1},qm^c)=\Theta(\textsc{Sort}(m,q))$$

$$\textsc{Prefix}(m^{c+1},qm^c,b)=\Theta(\textsc{Prefix}(m,q,b))$$

and

$$\textsc{Merge}(m^{c+1},qm^c)=\Theta(\textsc{Merge}(m,q)).$$

Thus, we have the following lemma:

**Lemma 2.2** *When concurrent reads are allowed, $k$ sorted arrays of a total size $n$, can be merged in*

$$O\left(\textsc{Sort}(k,\tfrac{pk}{n})+\textsc{Prefix}(k,\tfrac{p}{kt},\log n)+\textsc{Merge}(n,p)+\frac{n}{p}\right)$$

*time using $p$ processors.*

On a CREW PRAM, we can sort $n$ elements in $\textsc{Sort}(n,p)=O(\frac{n\log n}{p}+\log n)$ time (Upper Bound 2.9), find the prefix sum of $n$ $b$-bit numbers in $\textsc{Prefix}(n,p,b)=O(\frac{n}{p}+\log n)$ time (Upper Bound 2.7), and merge two arrays of a total size $n$ in $\textsc{Merge}(n,p)=O(\frac{n}{p}+\log\log n)$ time (Upper Bound 2.1).
Thus, we have, when $p\le n$,

$$\textsc{Sort}(k,\tfrac{pk}{n})=O\left(\frac{k\log k}{pk/n}+\log k\right)=O\left(\frac{n\log k}{p}+\log k\right) \tag{2.1}$$

$$\textsc{Prefix}(k,\tfrac{p}{kt},\log n)=O\left(\frac{k^2t}{p}+\log k\right)=O\left(\frac{n}{p}+\log k\right) \tag{2.2}$$

Applying Lemma 2.2, we have the following theorem:

**Theorem 2.1** *On a PRAM that allows concurrent reads, using $p$ processors, $k$ sorted arrays of a total size $n$ can be merged in time*

$$k\text{-}\textsc{Way\_Merge}(n,p)=O\left(\log k+\log\log n+\frac{n\log k}{p}\right)$$

**Theorem 2.2** *On a CREW PRAM, using $p>n$ processors, $k$ sorted arrays of a total size $n$ can be merged in time*

$$k\text{-}\textsc{Way\_Merge}(n,p)=O\left(\log\left(\frac{\log n}{\log(p/n)}\right)+\log k\right)$$

**Proof:** Use Upper Bound 2.9 for sorting, Upper Bound 2.2 for merging, and Upper Bound 2.7 for prefix summation, in Lemma 2.2. □

**Corollary 2.1** *On a CREW PRAM,*

$$k\text{-}\textsc{Way\_Merge}(n,p)=O(\textsc{Sort}(k,\tfrac{pk}{n})+\textsc{Merge}(n,p)).$$

**Theorem 2.3** *On a CRCW PRAM, $k$ sorted arrays of a total size $n$ can be merged in*

$$O\left(\log\left(\frac{\log n}{\log(p/n)}\right)+\textsc{Sort}(k,\tfrac{pk}{n})\right)$$

*time, using $p>n$ processors.*

**Proof:** On a CRCW PRAM, with $p>n$ processors, we can find the prefix sum of $n$ $b$-bit numbers in

$$\textsc{Prefix}(n,p,b)=O\left(\min\left\{\log\frac{b}{\log p},\log n\right\}+\frac{\log n}{\log\log p}+\frac{n}{p}\right)$$

time (Upper Bound 2.6) and merge two arrays of a total size $n$ in $\textsc{Merge}(n,p)=O(\log\frac{\log n}{\log(p/n)})$ time (Upper Bound 2.2). If, in addition, we use the best known CRCW PRAM sorting algorithm, by Lemma 2.2, the algorithm schema can be implemented in time

$$O\left(\log\frac{\log n}{\log(p/n)}+\min\left\{\log\frac{\log n}{\log(p/tk)},\log k\right\}+\frac{\log k}{\log\log(p/tk)}+\textsc{Sort}(k,\tfrac{pk}{n})\right)$$

$$=O\left(\log\frac{\log n}{\log(p/n)}+\frac{\log k}{\log\log(pk/n)}+\textsc{Sort}(k,\tfrac{pk}{n})\right).$$

Since $\textsc{Sort}(k,\frac{pk}{n})=\Omega(\frac{\log k}{\log\log(pk/n)})$, the theorem follows. □