Parallel Algorithms and Programming
Week 3

Kishore Kothapalli

September 11, 2009
Chapter 4

Sorting, Searching, and Merging

In this chapter we study algorithms for fundamental information processing algorithms such as searching, merging, and sorting. The problem of sorting on PRAM has attracted a lot of work because of its importance. Let us start with the problem of searching.

4.1 Searching

The problem of searching can be described as follows. Given a set $A$ of $n$ sorted elements from totally ordered set and an element $x$, find an index $i$ such that $a_i \leq x < a_{i+1}$. For simplicity, we may add two elements to $A$, namely $a_0$ and $a_{n+1}$ so that $a_0 = -\infty$ and $a_{n+1} = \infty$.

This problem has a very elegant and simple solution in the form of binary search which runs in $O(\log n)$ time. To design a parallel algorithm for this problem, let us recall the approach of binary search. The element $x$ is compared with the middle element and depending on the outcome of the comparison, we make a further comparison. A natural parallel extension would be to compare $x$ with multiple elements of $A$ and identify a small segment where to search for $x$ based on the outcomes of the comparisons.

Suppose we have $p$ processors that we can use to perform a search in parallel. Then we make $p$ comparisons of $x$ and elements of $A$. For this we make the comparisons to every $|A|/p$th element of $A$ so that $A$ can be treated as being composed of $p$ blocks. Let us denote by 0 the outcome that $x$ is greater than the element compared to and by 1 otherwise. So each processor is generating an outcome. In the list of outcomes, there will be a phase change from 1 to 0 at one block. It then suffices to continue the search in that block. An example is provided below.

**Example.** Let $A = (12, 18, 32, 46, 48, 54, 68, 73, 92)$ and $x = 55$ and suppose we have $p = 3$ processors. Then we compare $x$ with 32, 54, and 92. The outcomes are 1, 1, and 0 and the phase change happens at block 2.

The search continues in this fashion until the block size is more than $p$. Once the block size becomes $p$, then each processor can simply perform one additional comparison and find the required index $i$.

It will be a homework assignment to implement this strategy in the lab. A pseudocode can be found in the book. We shall now try to prove the time and work bounds.

**Theorem 4.1.1** $n$ element sorted array, search can be done in $O\left(\frac{\log n}{\log p}\right)$ parallel time, CREW model, $p$ processors.

**Proof.** Notice that the size of the array required to search drops by a factor of $p$ during every iteration. Hence in $O\left(\frac{\log n}{\log p}\right)$ iterations, the size drops to less than $p$. Each iteration takes $O(1)$ time. Hence the stated time bounds holds.
It shows that search is work optimal only for $p$ being a constant. But even this algorithm has some applications, for example in the faster merging algorithm we see next.

### 4.2 Merging

Recall that we have presented an $O(\log n)$ time parallel optimal merging algorithm in the last chapter. In this section, we try to improve the runtime to $O(\log \log n)$ while still retaining optimality. The solution is interesting because on the CREW model, merging is one problem with a very fast algorithm compared to most other simple problems such as finding the Boolean OR or maximum. Let us recall that the problem of merging can be reduced to finding two sequences: Rank($A, B$) and Rank($B, A$).

#### 4.2.1 Step 1: Ranking when $|A| = \omega(|B|)$

In this section, we illustrate the use of the parallel searching algorithm developed in the previous section to speed up ranking of the elements of $B$ in the array $A$ when the size of $A$ exceeds the size of $B$ by a polynomial amount. Formally, if $|A| = n$, we let the size of $B$ to be $n^\epsilon$ where $0 < \epsilon < 1$. We show the following theorem.

**Theorem 4.2.1** Let $A$ be a sorted sequence and $B$ be an arbitrary sequence so that $|A| = n$ and $|B| = n^\epsilon$ for $0 < \epsilon < 1$. Then, all the elements of $B$ can be ranked in $X$ in $O(1)$ time using $O(n)$ operations on the CREW model.

**Proof.** Use the parallel search algorithm for each element of $B$ in parallel with $n^{1-\epsilon}$ processors. Runtime $= O(1/1 - \epsilon)$ and total operations $= O(n^\epsilon \cdot n^{1-\epsilon}) = O(n)$.  

#### 4.2.2 Step 2: A fast non-optimal algorithm

Let us use the result of Step 1 to design a fast parallel algorithm which turns out to be slightly non-optimal. This is again common in the course of designing parallel algorithms. We first can develop a slightly non-optimal algorithm and use it to get to optimality without increasing the runtime.

The basic idea of the algorithm is to rank $\sqrt{n}$ elements of $B$ in $A$ using the previous algorithm. These ranks induce partitions in $A$ and $B$. Moreover, the problem reduces to ranking the previously unranked elements in the appropriate partition.

The strategy is elaborated as pseudocode in the following with $|A| = n$ and $|B| = m$ being two sorted sequences.

**Algorithm FastMerge($A, B$)**

1. **Step 1:** If $m < 4$ then apply parallel search for each element of $B$ in $A$ and return
2. **Step 2:** Rank the elements $B(\sqrt{m}), B(2\sqrt{m}), \ldots, B(m)$ in $A$ in parallel using $\sqrt{n}$ processors for each search.
   - Let $r(i) = \text{rank}(B(i\sqrt{m}), A)$.
3. **Step 3:** Let $B_i = (B(i\sqrt{m} + 1, \ldots, (i + 1)\sqrt{m} - 1))$ be the partitions of unranked element of $B$.
4. **Step 4:** Recursively compute rank($B_i, A_i$)
5. **Step 5:** To compute the final rank of $B(k)$ in $A$ where $k$ is not a multiple of $\sqrt{m}$, set $\text{rank}(B(k)) = j(i) + \text{rank}(B(k), A_i)$ where $i = \lfloor k/\sqrt{m} \rfloor$. 


Let us look at an example.

**Example.** Take an example here. □

We are now ready to show the following theorem.

**Theorem 4.2.2** The algorithm is correct, runs in parallel time of $O(\log \log m)$, total operations $= O((m + n) \log \log m)$, model = CREW

**Proof.** Proof of correctness by induction on size of $B$. For the time complexity, let $T(n, m)$ be parallel run time when $|A| = n$ and $|B| = m$. Then, Step 2 takes a parallel time of $O(1)$ using a total of $O(\sqrt{n} \cdot \sqrt{m}) = O(n + m)$ operations. Steps 4 and 5 also take $O(1)$ time. We need to worry about Step 4, the time spent in the recursive call. Here, notice that if $n_i$ denotes the size of the array $A_i$, then the time spent in the recursive call is bounded by $\max_i T(n_i, \sqrt{m})$ with $T(n, 3) = O(1)$. Hence,

$$T(n, m) = \max_i T(n_i, \sqrt{m}) + O(1)$$

The solution to this recurrence relation is $T(n, m) = O(\log \log m)$.

For the number of operations performed, notice that in each iteration a total of $O(n + m)$ operations are performed for a total of $O((n + m) \log \log m)$ operations. □

### 4.2.3 Step 3: Towards an Optimal Algorithm

Our goal now is to use the algorithm designed in the previous section to get an optimal algorithm. The approach we use is standard in the parallel algorithm community. If the algorithm we have is sub-optimal, try to use the sub-optimal algorithm on a reduced version of the input.

Since the previous algorithm is sub-optimal by a $O(\log \log m)$ factor, we intend to reduce the problem size by exactly that much factor and use the algorithm to get an optimal algorithm for the reduced version. Formally, we partition $A$ and $B$ into blocks $A_1, A_2, \ldots$ and $B_1, B_2, \ldots$ where each block has size at most $\lceil \log \log n \rceil$. Also, let $A' = A_1(1), A_2(1), \ldots$, and $B' = B_1(1), B_2(1), \ldots$, denote the arrays obtained by taking the first element in each block. Notice that $|A'| = |B'| = n/\log \log n$ enabling us to use the algorithm designed in the previous section to merge $A'$ and $B'$ in $O(\log \log n)$ time and $O(n)$ total work. The entire details of the algorithm are given below.

**Algorithm FastOptimalMerge($A, B$)**

- Step 1: Compute $A'$ and $B'$ as above and merge $A'$ and $B'$ the sub-optimal algorithm thus obtaining rank($A'$, $B'$) and rank($B'$, $A'$)
- Step 2: Rank $A'$ in $B$ and rank $B'$ in $A$
- Step 3: Now compute the ranks of the remaining elements of $A$ in $B$

and the remaining elements of $B$ in $A$.

Some explanation of the previous steps are in order. The utility of step 1 will be apparent later in step 3. But notice that step 1 uses a non-optimal algorithm on a problem of reduced size. In step 2, we proceed as follows. Let $r(i) = \text{rank}(A_i(1), B')$. Then the place of $A_i(1)$ in $B$ is in the $r(i)$th block of $B$. Why? Since the size of each block is at most $\lceil \log \log n \rceil$, for each element of $A'$ we can perform a search for its right location in the corresponding block of $B$. This search can simply be a sequential search using a single processor per element of $A'$. The parallel time is no more than $O(\log \log n)$ using $O(n/\log \log n)$ processors for a total of $O(n)$ operations. Reversing the roles of $A$ and $B$ lets us also rank $B'$ in $A$.

Let us now look at step 3. Consider any element of $A$ that is not ranked in $B$ yet. Let it belong to the $i$th block of $A$. Let $R(i) = \text{rank}(A_i(1), B)$ be the rank of the first element of the $i$th block of $A$ in $B$. Then the elements in the $i$th block of $A$ have a rank between $B_{R(i)}$ and $B_{R(i+1)}$. We differentiate between two cases.
• Case 1: \( R(i) = R(i + 1) \): In this case, all the elements of the \( i \)th block of \( A \) have a rank of \( R(i) \) in \( B \).

• Case 2: \( R(i + 1) > R(i) \): In this case, observe that all elements of \( B \) with indices between \( R(i) \) and \( R(i + 1) \) fit between \( A_i(1) \) and \( A_{i+1}(1) \). If the number of such elements is less than \( \log \log n \), then we can merge \( A_i \) with the corresponding subarray in \( B \). This merge can be performed in \( O(\log \log n) \) time using a single processor and the problem is independent of other subproblems. Otherwise, the size of the problem may exceed \( O(\log \log n) \). In this case, we can use the ranks of \( B_i(1) \) in \( A \) as follows. Let \( \text{rank}(B_i(1), A) = T(i) \). There exist elements of \( B \), say \( B_k(1), B_{k+1}(1), \ldots, B_{k+s}(1) \), that have a rank between \( A_i(1) \) and \( A_{i+1}(1) \) as shown in Figure ???. Since the ranks of these elements in \( A \), \( T(k), T(k + 1), \ldots, T(k + s) \), are known, these ranks can be used to create \( s \) independent subproblems each of which has a size of \( O(\log \log n) \). These subproblems can be solved sequentially.

Putting together everything, we have the following theorem:

**Theorem 4.2.3** optimal merging in \( O(\log \log n) \) time on CREW model.

### 4.3 Sorting Networks: Hardware for Sorting

Let us look at special purpose hardware for sorting. A **comparator** is a hardware unit with two inputs \( x \) and \( y \) and two outputs which are ordered: \( \min\{x, y\} \) and \( \max\{x, y\} \), as shown in Figure 4.2. The direction of the arrow is significant in ordering the output.

A comparator network consists of several comparators so that the final output is the ordered list of input elements. The size of a comparator network is defined to be the number of comparators and the depth of a comparator network is the maximum number of comparators along any input to output path. There is an easy way to realize a parallel algorithm given a comparator network. The running time of such a parallel algorithm would be proportional to the depth of the network and the total number of operations would be proportional to the size of the network. An example of a comparator network is shown in Figure ???.

In this section, we first develop a sorting network for a special class of inputs called **bitonic sequences**.

**Definition 4.3.1 (Bitonic Sequence)** A sequence \( X = \{x_0, x_1, \ldots, x_{n-1}\} \) with elements from a linearly ordered set is called bitonic if there exist indices \( k, \ell \) such that \( x_k \leq x_{k+1} \leq \cdots x_\ell \leq x_{\ell+2} \geq \cdots x_{k-1} \) where the indices are computed modulo \( n \).
Contrast the definition of bitonic sequences with that of monotonic sequences. The elements of a bitonic sequence can be arranged so that it has two monotonic sequences. Bitonic sequences have several interesting properties such as:

- A subsequence of a bitonic sequence is also bitonic.
- Every bitonic sequence has the unique cross-over property.

**Unique Crossover Property**

The unique crossover property of bitonic sequences can be explained as follows. For simplicity, assume that in the above definition of bitonic sequences \( k = 0 \) and \( \ell = n/2 \) so that we have \( x_0 \leq x_1 \leq \cdots \leq x_{n/2-1} \) and \( x_{n/2} \geq x_{n/2+1} \geq \cdots \geq x_{n-1} \). Arrange these two sets of as shown in Figure ??.

Then the crossover property states that there exists an index \( i \) such that:

1. for any \( a \in \{x_0, x_1, \cdots, x_{i-1}\} \) and for any \( b \in \{x_{n/2+i-1}, x_{n/2+i}, \cdots, x_{n/2}\} \), it holds that \( a \leq b \).
2. for any \( a \in \{x_i, x_{i+1}, \cdots, x_{n/2-1}\} \) and for any \( b \in \{x_{n/2+i}, x_{n/2+i+1}, \cdots, x_{n-1}\} \), it holds that \( a > b \).
3. for any \( a \in \{x_0, x_1, \cdots, x_{i-1}\} \) and for any \( b \in \{x_i, x_{i+1}, \cdots, x_{n/2-1}\} \), it holds that \( a \leq b \), and
4. for any \( a \in \{x_{n/2}, x_{n/2+1}, \cdots, x_{n/2+i-1}\} \) and for any \( b \in \{x_{n/2+i}, x_{n/2+i+1}, \cdots, x_{n-1}\} \), it holds that \( a \geq b \).

To check these properties, we just have to exhibit such an index \( i \) which we do as follows. Let \( i \) be the smallest index such that \( x_i > x_{n/2+i} \) with \( 0 \leq i \leq n/2 - 1 \). It then holds that for any \( 0 \leq j \leq i - 1 \), \( x_0 \leq x_1 \leq \cdots \leq x_{i-1} \leq x_{n/2+i-1} \leq x_{n/2+i-2} \leq \cdots \leq x_{n/2} \). Why? So property (1) holds. Property (3) follows easily. A similar verification of properties (2) and (4) can be done. But what if no such index exits? In that case we can take \( i \) to be \( n/2 \). It can also happen that \( i = 0 \).

Consider an example here. It is illustrative to write the elements in a circular order.

But what we wanted to show is that every bitonic sequence has the unique crossover property. This is easy to establish too as shown in [?], Lemma4.8. So from now on we assume so.

Let us now explore the connections of bitonic sequences to sorting. Suppose that a sequence \( X \) is bitonic. Let \( L(X) = \{\min\{x_i, x_{n/2+i}\}, 0 \leq i \leq n/2 - 1\} \) and \( R(X) = \{\max\{x_i, x_{n/2+i}\}, 0 \leq i \leq n/2 - 1\} \). Then we show the following lemma.

**Lemma 4.3.2** If \( X \) is bitonic then both \( L(X) \) and \( R(X) \) are bitonic and every element of \( L(X) \) has a value at most the value of any element of \( R(X) \).

**Proof.** Since \( L \) and \( R \) are sub-sequences of \( X \), they are bitonic as \( X \) is bitonic. Further, notice that the sequence \( X \) has the unique crossover property and hence there exists an index \( i \) that satisfies that property. We can then identify that \( L(X) = \{x_0, x_1, \cdots, x_{i-1}\} \cup \{x_{n/2+i}, x_{n/2+i+1}, \cdots, x_{n-1}\} \) and \( R(X) = \{x_{n/2}, x_{n/2+1}, \cdots, x_{n/2+i}\} \cup \{x_i, x_{i+1}, \cdots, x_{n/2-1}\} \). By definition of the index \( i \), the statement that any element of \( L(X) \) has a value that is at most the value of any element of \( R(X) \) holds. \( \square \)

The above lemma can be used to develop the following partitioning based algorithm for sorting a bitonic sequence \( X \).
Algorithm BitonicSort($X$)
Step 1: for $1 \leq in/2 - 1$ do in parallel
$L_i = \min\{x_i, x_{i+n/2}\}$
$R_i = \max\{x_i, x_{i+n/2}\}$
Step 2. Recursively sort $L$ and $R$
Step 3. Output $L$ followed by $R$

The above algorithm also leads us to a sorting network for a bitonic sequence $X$. We develop a recursive construction for the network. If $n = 2$, then the network consists of just one comparator. Otherwise, the network is built using two networks for size $n/2$. Let $B(n)$ denote the network for $n$ elements. Then, following the above algorithm, $B(n)$ is composed of two $B(n/2)$ corresponding to step 2. For step 1, we add a further $n/2$ comparators as shown in Figure ??.

Lemma 4.3.3 Size $= O(n \log n)$, depth $= O(\log n)$

Proof. Size of the network, denoted $S(n)$, can be expressed as the following recurrence.

$$S(n) = S(n/2) + n/2$$

For the above recurrence, $S(n) = O(n \log n)$ is a solution.

Similarly, for the depth of the network, denote $D(n)$, the recurrence $D(n) = D(n/2) + 1$ holds for which the solution is $D(n) = O(\log n)$. \qed

But our job is only half-done as the above network, or the algorithm, works only for bitonic sequences. Fortunately, however, the above can be used to develop a network for sorting arbitrary sequences. Notice that every sequence of length 1 or 2 is bitonic. So we can use these to build bitonic sequences of length 4, 8, and so on. Each stage we can use the above network for sorting bitonic sequences. The entire network for an arbitrary $n$ element sequence is shown in Figure ??.

The following theorem follows.

Theorem 4.3.4 Size $= O(n \log^2 n)$, depth $= O(\log^2 n)$.

Remark 4.3.5 We only touched upon the topic of design of networks, or circuits, which has a rich theory of its own. Even for the sorting problem, in a remarkable development Ajtai, Komlos, and Szemeredi have shown that sorting network of size $O(n \log n)$ and depth $O(\log n)$ can be constructed. However, the constants involved are quite huge which make them impractical.

4.4 Sorting

Sorting is a fundamental problem in computer science and a lot of research has been carried out for sorting even in the sequential model of computation. The newest sorting algorithm is in 2005. The problem has attracted similar attention also in the parallel model. In this section, we study two algorithms for sorting in parallel.

4.4.1 A Simple Algorithm

We can use the optimal algorithm for merge sort developed in the previous section to design a parallel sorting algorithm. This design is similar to that of the sequential merge-sort algorithm.
4.4. SORTING

Just like the sequential merge sort, the computation here too can be modeled as a complete binary tree. Each node $v$ can be associated with a list $L_v$ which contains the sorted list of elements in the sub-tree rooted at $v$. By definition, the list $L_v$ for the root node $v$ is the sorted list and the list at a leaf node consists of a single element.

The algorithm is presented below:

**Algorithm ParallelSort**

Step 1: for $i = 1$ to $n$ do

$L(0, i) = A(i)$

Step 2: for level $= 1$ to $\log n$ do

for $1 \leq j \leq n/2^{\text{level}}$ do

$L(\text{level}, j) = \text{Merge}(L(\text{level} - 1, 2j - 1), L(\text{level} - 1, 2j))$

end.

The following theorem can be shown:

**Theorem 4.4.1** \textit{Time} = $O(\log n \log \log n)$, \textit{work} = $O(n \log n)$, \textit{model} = CREW, optimal algorithm