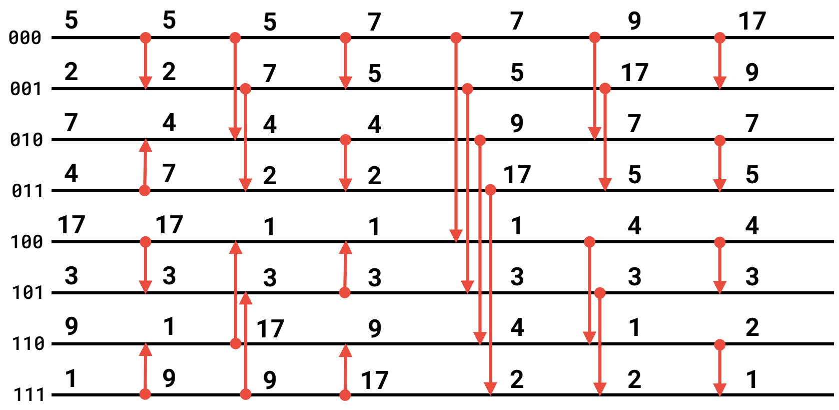
CSE 6220 Homework 3

Karl Hiner, @khiner6

1

Draw a 8-element bitonic sorting circuit to sort elements in non-increasing order. Use horizontal lines for the numbers and vertical lines to denote comparison operations. Label the comparison operations as ↑ or ↓ where the direction of the arrowhead indicates the destination of the smaller element. Illustrate how the input 15, 2, 7, 4, 17, 3, 9, 1 is sorted using the diagram.



The only modification needed to the bitonic sort algorithm to sort elements in *non-increasing* order, instead of *non-decreasing* order, is to flip the `Compare_Exchange` directions.

2

The *Bitonic Split* operation defined in class assumes that the bitonic sequence has an even length. Extend this operation to bitonic sequences of odd length. Now, show how the bitonic sequence 5, 3, 4, 7, 10, 14, 17, 13, 8 can be converted into a sorted sequence using repeated bitonic split operations.

Ordinary Bitonic Split works by decomposing a bitonic sequence $l = x_0, x_1, \cdots, x_{n-1}$ into

$$l_i^- \equiv \min\left(x_i, x_{i+\frac{n}{2}}\right), \quad l_i^+ \equiv \max\left(x_i, x_{i+\frac{n}{2}}\right), \quad 0 \le i \le \frac{n}{2} - 1.$$

One way to extend this operation to bitonic sequences of odd length is to use a similar strategy to the one we used to extend parallel prefix to non-power-of-two numbers of processors. In that case, we kept the exact same algorithm, "pretending" that there were $p = 2^m$ processors, and ignored them during communication. Similarly, we can "pretend" the given sequence l has length $2m$, by appending a "dummy" element if l has odd length. Then, after completion of the split, we can remove the dummy element.

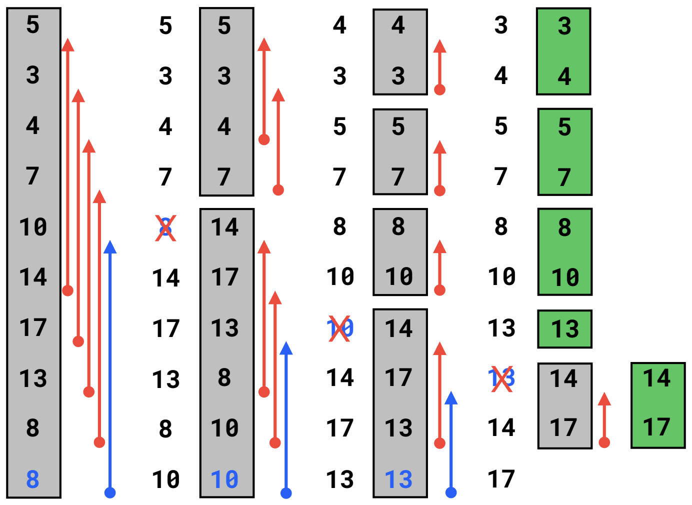
Let $\tilde{l} \equiv l + \{D\}$ refer to the sequence obtained by appending the element D to the sequence l. Then as long as we choose D such that $\tilde{l}$ is also bitonic, after we apply the usual split operation we will have bitonic split sequences $\tilde{l}^- \equiv \tilde{l}_{min}$ and $\tilde{l}^+ \equiv \tilde{l}_{max}$, and we will also have $\max(\tilde{l}^-) \leq \min(\tilde{l}^+)$.

When we later *remove* the appended element D, we must remove it from either $\tilde{l}^-$ or $\tilde{l}^+$. If D was in $\tilde{l}^-$, removing it will clearly maintain $\max\left(\tilde{l}^- - \{D\}\right) \leq \max\left(\tilde{l}^-\right)$. Likewise, removing D from $\tilde{l}^+$ will maintain $\min\left(\tilde{l}^+ - \{D\}\right) \geq \min\left(\tilde{l}^+\right)$. So we have $\max\left(\tilde{l}^- - \{D\}\right) \leq \min\left(\tilde{l}^+ - \{D\}\right)$. Finally, note that by the definition of bitonic sequences, removing *any* single element from a bitonic sequence results in another bitonic sequence.

Thus, if we append an element that produces another bitonic sequence, then perform the usual split operation on the resulting sequence, and finally remove the appended element, we maintain all properties required for the Bitonic Split operation. A simple choice for D which guarantees $\tilde{l}$ is bitonic is to *duplicate the last element*. (We could also *prepend* the first element.)

Here is our algorithm:

- If l has length 1 or 2, there is nothing to do.
- Otherwise, if l has even length, perform ordinary bitonic split.
- Otherwise, if $l = x_0, x_1, \cdots, x_{n-1}$, has odd length,
  - Append a new element $D = l_{n-1}$ to create $\tilde{l} = x_0, x_1, \cdots, x_{n-1}, x_{n-1}$.
  - Perform ordinary bitonic split on $\tilde{l}$, marking the new position of the element D during its compare-exchange.
  - Create split sequences $l^- \equiv \tilde{l}^- - \{D\}, \quad l^+ \equiv \tilde{l}^+ - \{D\}$. (To be clear, only one of $\tilde{l}^-$ or $\tilde{l}^+$ will contain D.)



3

Give an algorithm to merge two sorted sequences of lengths m and n, respectively. You may assume that the input is an array of length $m+n$ with one sequence followed by the other, distributed across processors such that each processor has a subarray of size $\frac{m+n}{p}$. What are the computation and communication times for your algorithm? (You may assume the use of any permutation-style communication, not necessarily hypercubic.)

Observe that for any two consecutive sorted sequences, we can reverse the second sequence to obtain a single bitonic sequence. Thus, we can merge two sorted sequences by reversing the second sequence, and then performing a single Bitonic Merge.

Reversing the second sequence can be done by swapping the last n elements, taking $O(\tau + \mu \frac{n}{p})$ communication time. Then, we can perform a Bitonic Merge in with $O(\frac{m+n}{p} \log p)$ computation time and $O(\tau \log p + \mu \frac{m+n}{p} \log p)$ communication time.

Total computation time: $O(\frac{m+n}{p} \log p)$
Total communication time: $O(\tau + \mu \frac{n}{p} + \tau \log p + \mu \frac{m+n}{p} \log p) = O(\tau \log p + \mu \frac{m+n}{p} \log p)$

4

Two algorithms are designed for the All-to-all communication primitive, and have the following runtimes:

$$\begin{aligned} \text{Algorithm 1} &: \Theta(\tau \log p + \mu m p \log p) \\ \text{Algorithm 2} &: \Theta(\tau p + \mu m p) \end{aligned}$$

Determine which algorithm runs faster asymptotically as a function of the message size.

As mentioned in class, we can simplify this problem by comparing the dominating terms. Since τp dominates $\tau \log p$, and $\mu m p \log p$ dominates $\mu m p$, we can simply compare these two terms.

Asymptotically with respect to m, Algorithm 1 will run faster roughly when

$$\Theta(\mu m p \log p) < \Theta(\tau p) \implies m < \frac{\tau}{\mu \log p}.$$

For messages sizes beyond this threshold, Algorithm 2 will run faster.

5

Consider a tree of n nodes and having a bounded degree (i.e., the number of children per node is bounded by a constant). The tree is stored in an array A of size n. Each node also contains the indices at which its parent and children are stored in the array. The array is distributed among processors using block decomposition, i.e., P_i contains $A[i\frac{n}{p}], A[i\frac{n}{p}+1], \ldots, A[(i+1)\frac{n}{p}-1]$. For each of the following operations, which communication primitive will you use and what is the runtime?

(a) Each node in the tree has $O(1)$ sized data that should be sent to its parent.

(b) Each node in the tree has $O(1)$ sized data that should be sent to all its children.

Since each node's parent and/or children may each be located in a different processor, we must either use the All-to-All or Many-to-Many communication primitive for both operations.

For (a), the size of each sent message will be constant, since each node apart from the root has exactly one parent. This would indicate using All-to-All. However, we can improve the runtime using Many-to-Many by observing that each *received* message size can be variable (since each parent may have a different number of children) and bounded (by the tree degree).

The runtime for Many-to-Many can be derived by replacing the message size m in the All-to-All runtime $\Theta(\tau p + \mu m p)$ with the sum of the maximum sent and received message sizes.

Let D denote the tree degree, and R and S denote the maximum received and sent message sizes for a single processor. Then we have

$$m \leq (R + S) = \left(D\frac{n}{p} + \frac{n}{p} \right) = \frac{n}{p}(D+1),$$

giving a runtime for (a) of

$$\Theta(\tau p + \mu m p) = \Theta(\tau p + \mu \frac{n}{p}(D+1)p) = \Theta(\tau p + \mu n(D+1)) = \Theta(\tau p + \mu n).$$

For (b), the upper bound for the received and sent message sizes are flipped, but this will not change the value of $R + S$. So the runtime for both (assuming we have a network allowing arbitrary permutations) will be $\Theta(\tau p + \mu n)$.