CSE 6220 Homework 2

Karl Hiner, @khiner6

1

Determine if the parallel prefix algorithm can be used to compute prefix sums of a sequence of n numbers based on the binary operation $\bigoplus$ defined as:

The parallel prefix algorithm can be used to compute the prefix sums of a sequence of n numbers if and only if $\oplus$ is a binary associative operator.

(a) $a \oplus b = 2a + b$

Checking associativity,

$$\begin{aligned} (a \oplus b) \oplus c &\stackrel{?}{=} a \oplus (b \oplus c) \\ 2(2a + b) + c &\stackrel{?}{=} 2a + (2b + c) \\ 4a + 2b + c &\neq 2a + 2b + c \end{aligned}$$

This shows the binary operation defined as $a \oplus b = 2a + b$ *is not* associative, and thus *cannot* be used by the parallel prefix algorithm to compute prefix sums.

(b) $a \oplus b = \sqrt{a^2 + b^2}$

$$\begin{aligned} (a \oplus b) \oplus c &\stackrel{?}{=} a \oplus (b \oplus c) \\ \sqrt{\left(\sqrt{a^2 + b^2}\right)^2 + c^2} &\stackrel{?}{=} \sqrt{a^2 + \left(\sqrt{b^2 + c^2}\right)^2} \\ \left(\sqrt{a^2 + b^2}\right)^2 + c^2 &\stackrel{?}{=} a^2 + \left(\sqrt{b^2 + c^2}\right)^2 \\ a^2 + b^2 + c^2 &= a^2 + b^2 + c^2 \end{aligned}$$

This shows the binary operation defined as $a \oplus b = \sqrt{a^2 + b^2}$ *is* associative, and thus *can* be used by the parallel prefix algorithm to compute prefix sums.

2

In the game of Photosynthesis, points are given for trees that receive sunlight. Consider n trees $T_0, T_1, ..., T_{n-1}$ planted along a single row of spaces, and sunlight is colinear with this row of trees. The tree placement is modeled by an array A of size n, where $A[i]$ denotes the height of the tree T_i. A tree tall enough to get sunlight exposure scores photosynthesis points according to its height, i.e., T_i is given $A[i]$ points. However, a tree can also be blocked from sunlight by an earlier tree *of equal height or taller*, in which case the blocked tree receives no points.

Design a parallel algorithm to compute the total number of points for a configuration given by A, and compute its runtime.

Assume the array A is block-distributed across all nodes. Let p_k refer to the node holding the n/p elements $A_k \equiv \Big[A[\frac{kn}{p}], ..., A[\frac{(k+1)n}{p} - 1]\Big]$.

1. Perform parallel prefix with $\oplus = \max$, storing the result in array M_i. (Note that the max operation is binary associative, and thus can be used for parallel prefix.) After parallel prefix, $M_i[j] \geq A[0, 1, ..., \frac{ni}{p} + j]$. Parallel prefix takes $\Theta(\frac{n}{p} + \log p)$ time, with communication time $\Theta((\tau + \mu)\log p)$.
2. Use a right-shift to send the last element of M_{i-1} to its right neighbor processor i, storing this value in $\hat{m}_i$. This is necessary since each element of A_i will need to be compared with the *previous* max element. Communication cost: $\Theta(\tau + \mu)$
3. Compute the local total tree score for each node, as follows:
   - Initialize the running total local score $s_k \leftarrow 0$.
   - For each tree height in the local list A_i, if $A_i[j] > M_i[j-1]$, add the tree height to the total local score, $s_k \mathrel{+}= A_j[j]$. *For* $j - 1 = -1$*, use* $\hat{m}_i$ instead of (nonexistent) $M_i[-1]$.

   This step takes $\Theta(\frac{n}{p})$ time.
4. Use the parallel sum algorithm to sum all local tree scores to find the total tree score, $S = \sum_k s_k$. This final step takes $\Theta(\log p)$ time.

Computation time: $\Theta(2\Big(\frac{n}{p} + \log p\Big))$.

Communication time: $\Theta((\tau + \mu)(\log p + 1))$.

3

A sequence of nested parenthesis is said to be well-formed if

1. there are an equal number of left and right parenthesis, and
2. each right parenthesis is matched by a left parenthesis that occurs to its left in the sequence.

For example, ((( ) ( )) ( )) is well-formed, but ( )) (is not.

There is a nested parenthesis sequence of length n distributed across p processors using block decomposition. Design a parallel algorithm to determine if it is well-formed and specify its run-time.

1. Assign -1 to a closed parenthesis, and 1 to an open parenthesis. Then each processor i has a list of values $A_i \in (-1, 1)^{\frac{n}{p}}$.
2. Use parallel prefix with $a \oplus b = a + b$ (parallel prefix sum).
3. If any (global) prefix sum element is negative, return "not well-formed".

4. If all prefix sums are non-negative, and the final prefix sum (the total sum) is equal to 0, the algorithm returns "well-formed". Otherwise, return "not well-formed".

The runtime of this algorithm is the same as parallel prefix sum, $\Theta(\frac{n}{p} + \log p)$. Unlike prefix sum, however, the best-case runtime is $\Theta(1)$, since the first processor could return "not well-formed" if its first element is a closed parenthesis (-1).

4

Let A be an array of n elements and L be a boolean array of the same size. We want to assign a unique rank in the range $1, 2, ..., n$ to each element of A such that for any $i < j$:

- If $L[i] = L[j]$, $A[i]$ has lower rank than $A[j]$.
- If $L[i] = 0$ and $L[j] = 1$, $A[i]$ has lower rank than $A[j]$.
- If $L[i] = 1$ and $L[j] = 0$, $A[j]$ has lower rank than $A[i]$.

Design a parallel algorithm to compute the ranks and specify its run-time. (Hint: Think of L as specifying labels. Then, all elements with 0 label receive lower ranks than any element with label 1. Within the same label, ranks are given in left to right order as per array A.)

Assume arrays A and L are block-distributed across p processors, such that each processor holds $\frac{n}{p}$ contiguous elements from each array in local arrays A_i and L_i.

1. Compute the number of 0s and 1s in each node's L_i array, and store in $\boldsymbol{n_i} = \begin{bmatrix} \text{num 0s} \\ \text{num 1s} \end{bmatrix}$. This step takes $\Theta(\frac{n}{p})$ time.
2. Perform a parallel prefix operation across $\boldsymbol{n_i}$, using vector sum as the operator, and store the results in $\boldsymbol{N_i} = \sum_{j \le i} \boldsymbol{n_j}$. (Note that after this step, $\boldsymbol{N_i} - \boldsymbol{n_i}$ gives the total number of zeros/ones *before* node i.) This step takes $\Theta(2 \log p)$ computation time and $\Theta((\tau + 2\mu) \log p)$ communication time.
3. Broadcast the total number of 0s in L (stored in $\boldsymbol{N_p}[0]$, where p is the index of the last processor), to every other node. Store the result as Z in each node. This takes $\Theta((\tau + \mu) \log p)$ communication time.
4. For each node, traverse its L_i and assign ranks to elements in A as follows, storing the result in local rank array R_i:

$$R_i[j] = \begin{cases} c_0 + \boldsymbol{N_i}[0] - \boldsymbol{n_i}[0], & L_i[j] = 0 \quad (c_0\text{+num 0s before me}) \\ c_1 + Z + \boldsymbol{N_i}[1] - \boldsymbol{n_i}[1], & L_i[j] = 1 \quad (c_1\text{+total 0s + num 1s before me}) \end{cases},$$

where c_0/c_1 are the running count of 0s/1s encountered so far during the traversal. This step takes $\Theta(\frac{n}{p})$ time.

Computation time: $\Theta(2\left(\frac{n}{p} + \log p\right))$.

Communication time: $\Theta((2\tau + 3\mu) \log p)$.

Note: Instead of using Z, the total number of 0s in L, as a rank offset when $L_i[j] = 1$, we could use any $N \ge Z$. For example, we could use n. We would still need to broadcast n, as we did for Z, but it could be done in the first step. Alternatively, if we were allowed to provide n to each processor along with its L_i and A_i (as in PA1), we could skip the broadcast step entirely, resulting in a total communication time of $\Theta((\tau + 2\mu) \log p)$.

5

Invent Segmented Parallel Prefix: Segmented parallel prefix is a generalization of the parallel prefix problem where the prefix sums need to be restarted at specified positions. Consider array X containing n numbers and a boolean array B of the same size. We wish to compute prefix sums on X but the sum resets at every position i where $B[i] = 1$. Formally, we wish to compute array S of size n such that

$$
\begin{aligned}
S[0] &= X[0] \\
S[i] &= \begin{cases} S[i-1] + X[i], & \text{if B[i] = 0} \\ X[i], & \text{if B[i] = 1} \end{cases}
\end{aligned}
$$

Design parallel segmented prefix algorithm and specify its run-time.

(Hint: The problem can be transformed into a standard prefix sums problem.)

We are told this can be transformed into a standard prefix sums problem. So we must find a binary associative operator that satisfies the given definition for S.

Simplifying the $S[i]$ case:

$$
\begin{aligned}
S[i] &= \begin{cases} S[i-1] + X[i], & \text{B[i] = 0} \\ X[i], & \text{B[i] = 1} \end{cases} \\
&= X[i] + \begin{cases} S[i-1], & \text{B[i] = 0} \\ 0, & \text{B[i] = 1} \end{cases} \\
&= X[i] + S[i-1](1 - B[i])
\end{aligned}
$$

For the case of $i = 1$, we can solve this using the base case of $S[0] = X[0]$ to get rid of the reference to the previous S element on the right:

$$
\begin{aligned}
S[1] &= X[1] + S[0](1 - B[1]) \\
&= X[1] + X[0](1 - B[1])
\end{aligned}
$$

Our binary operator must act on element types containing all the information needed in this expression. Let this element type be defined as $\boldsymbol{A_i} \equiv \begin{bmatrix} X_i \\ B_i \end{bmatrix} \equiv \begin{bmatrix} X[i] \\ B[i] \end{bmatrix}$. That is, $\boldsymbol{A_i}$ is the two-element vector formed by combining the ith element of X with the ith element of B. Then we can consider the inputs X and B as being provided in a single $2 \times n$ matrix $\boldsymbol{A}$, whose columns $\boldsymbol{A_i}$ we block-distribute across p processes. (We do not actually need to create these elements - this is only a conceptual reformulation with no incurred cost.)

From our solution to $S[1]$ above, we know exactly how to compute the first item (the numeric value) in our binary associative operator $\oplus$, given two consecutive columns of $\boldsymbol{A}$ (since $S[1] \equiv (\boldsymbol{A_0} \oplus \boldsymbol{A_1})_0$). We then only need to consider how to compute the secont item (the boolean value). I must admit I did this step by trial and error, so I don't provide a derivation here, but found that the logical-or (max) works.

Thus, our boolean operator is:

$$\boldsymbol{A_i} \oplus \boldsymbol{A_j} \implies \begin{bmatrix} X_i \\ B_i \end{bmatrix} \oplus \begin{bmatrix} X_j \\ B_j \end{bmatrix} = \begin{bmatrix} X_j + X_i(1 - B_j) \\ \max(B_i, B_j) \end{bmatrix}$$

Checking associativity:

$$\begin{gathered}
(\boldsymbol{A_a} \oplus \boldsymbol{A_b}) \oplus \boldsymbol{A_c} \stackrel{?}{=} \boldsymbol{A_a} \oplus (\boldsymbol{A_b} \oplus \boldsymbol{A_c}) \\
\left(\begin{bmatrix} X_a \\ B_a \end{bmatrix} \oplus \begin{bmatrix} X_b \\ B_b \end{bmatrix} \right) \oplus \begin{bmatrix} X_c \\ B_c \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} X_a \\ B_a \end{bmatrix} \oplus \left(\begin{bmatrix} X_b \\ B_b \end{bmatrix} \oplus \begin{bmatrix} X_c \\ B_c \end{bmatrix} \right) \\
\left(\begin{bmatrix} X_b + X_a(1 - B_b) \\ \max(B_a, B_b) \end{bmatrix} \right) \oplus \begin{bmatrix} X_c \\ B_c \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} X_a \\ B_a \end{bmatrix} \oplus \left(\begin{bmatrix} X_c + X_b(1 - B_c) \\ \max(B_b, B_c) \end{bmatrix} \right) \\
\begin{bmatrix} X_c + (X_b + X_a(1 - B_b))(1 - B_c) \\ \max(\max(B_a, B_b), B_c) \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} X_c + X_b(1 - B_c) + X_a(1 - \max(B_b, B_c)) \\ \max(B_a, \max(B_b, B_c)) \end{bmatrix} \\
\begin{bmatrix} X_c + X_b(1 - B_c) + X_a(1 - B_b)(1 - B_c) \\ \max(B_a, B_b, B_c) \end{bmatrix} \stackrel{?}{=} \begin{bmatrix} X_c + X_b(1 - B_c) + X_a(1 - \max(B_b, B_c)) \\ \max(B_a, B_b, B_c) \end{bmatrix}
\end{gathered}$$

Cancelling out all like terms, we are left with

$$\begin{aligned}
(1 - B_b)(1 - B_c) &\stackrel{?}{=} 1 - \max(B_b, B_c). \\
B_b + B_c - B_b B_c &= \max(B_b, B_c).
\end{aligned}$$

This equality can be readily verified with a truth table (not provided here).

Finally, here is the algorithm, assuming X and B are block distributed.

1. Perform parallel prefix using $\begin{bmatrix} X_i \\ B_i \end{bmatrix} \oplus \begin{bmatrix} X_j \\ B_j \end{bmatrix} = \begin{bmatrix} X_j + X_i(1 - B_j) \\ \max(B_i, B_j) \end{bmatrix}$, storing the resulting two-element vectors as columns in the $\left(2 \times \frac{n}{p}\right)$ matrix $\boldsymbol{A_i}$.
2. The array S can be read directly from the first row of each $\boldsymbol{A_i}$.

Computation time: $\Theta\left(4\left(\frac{n}{p} + \log p\right)\right)$ (parallel prefix with ≈ 4 flops per op).

Communication time: $\Theta((\tau + 2\mu) \log p)$ (parallel prefix with 2D elements).