CS 6230: Midterm

Christopher Mertin

Due Date: March 31, 2016

Question 1: Parallel Maximum Subarray . . . .

You are given a one dimensional array that may contain both positive and negative numbers, develop a parallel algorithm to find the sum of contiguous subarray of numbers which has the largest sum. For example, if the given array is $[-2, -5, \underline{6}, \underline{-2}, \underline{-3}, \underline{1}, \underline{5}, -6]$, then the maximum subarray sum is 7 (underlined numbers).

Solution: The sequential version of this is known as Kadane's Algorithm for the maximum subarray problem and, and the pseudocode is given as

Algorithm 1 Kadane's Algorithm

Input: Array $A \in \mathbb{R}^{n\times 1}$
Output: Maximum sum of contiguous subarray

$max_1 \leftarrow 0$
$max_2 \leftarrow 0$
for $i = 0$ **to** $n-1$ **do**
    $max_2 \leftarrow max_1 + A_i$
    if $max_2 < 0$ **then**
        $max_2 \leftarrow 0$
    end if
    if $max_1 < max_2$ **then**
        $max_1 \leftarrow max_2$
    end if
end for
return max_1

which is $\mathcal{O}(n)$ in complexity. This can be done in parallel in $\mathcal{O}\left(\log^2(n/p)\right)$ in the following way, where a Reduction would be performed at *each level* of the Prefix Sum, hence it's denoted as "PrefixSumReduction."

Algorithm 2 Parallel Max Subarray Problem

Input: Array $A \in \mathbb{R}^{n\times 1}$
Output: Maximum sum of contiguous subarray

$b_1 \leftarrow$ PrefixSumReduction(A,Max) {PrefixSum is inclusive}
$b_2 \leftarrow$ PrefixSumReduction(A,Max) {PrefixSum is exclusive}
return Max(b_1,b_2)

Question 2: Parallel Array Reordering . . . .

Suppose we are given a set of n elements stored in an array A together with an array L such that $L_i \in \{1, 2, \dots, k\}$ represents the label of element A_i, where k is a constant. Develop an optimal $\mathcal{O}(\log(n))$ time Er–Ew Pram algorithm that stores all the elements of A with label 1 into the upper part of A while preserving their initial ordering, followed by the elements labeled 2, with the same initial ordering, and so on. For Example:

$$A = [6, 5, 3, 9, 11, 12, 8, 17, 21, 2]$$
$$L = [1, 1, 2, 3, \ \ 2, \ \ 1, 1, \ \ 2, \ \ 3, 3]$$

produces

$$A = [6, 5, 12, 8, 3, 11, 17, 9, 21, 2]$$

Solution: A sequential version of this code would be as follows

Algorithm 3 Sequential Array Reordering

Input: $A \in \mathbb{R}^{n \times 1}$, $L \in \mathbb{R}^{n \times 1}$
Output: $C \in \mathbb{R}^{n \times 1}$ as a sorted array according to L

Create $M \in \mathbb{R}^{k \times n}$
for $i = 0$ **to** $n - 1$ **do**
  $j \leftarrow L_i$
  $M_{j,i} \leftarrow A_i$
end for
$C \leftarrow$ In-Order Traversal of elements in M
return C

This would be done in $\mathcal{O}(n)$. In parallel, for a Er–Ew Pram model, to do this in $\mathcal{O}(\log(n))$ can be done with the following algorithm, provided that $p > \frac{n}{\log(n)}$

Algorithm 4 Parallel Array Reordering

Input: $A \in \mathbb{R}^{n \times 1}$, $L \in \mathbb{R}^{n \times 1}$
Output: $C \in \mathbb{R}^{n \times 1}$ as a sorted array according to L

$\ell_{max} \leftarrow$ Reduction(L,Max)
for $i = 0$ **to** ℓ_{max} **do**
  $S \leftarrow \{\}$
  $\ell \leftarrow$Reduction(L,Min+i)
  for $j = \frac{n}{p}(thread)$ **to** $\frac{n}{p}(thread + 1)$ **do**
    if $L_j = \ell$ **then**
      $S \leftarrow$ Append(A_j)
    end if
  end for
  Gather(C, S) {Gathers all arrays of S to C}
end for
return C

Question 3: Parallel Fibonacci Numbers . . . .

Develop a work and depth optimal parallel algorithm for computing the first n Fibonacci Numbers

Solution: There are two sequential implementations for calculating the Fibonacci Sequence. Algorithm 5 is the brute force approach, which is just based on the definition of calculating the Fibonacci Sequence. The time complexity of this algorithm is $T(n) = T(n-1) + T(n-2)$, which is exponential in n.

On the other hand, Algorithm 6 is an approach that is faster but trades off with utilizing storage. Instead of recalculating *every* Fibonacci number, Algorithm 6 stores the ones that have been calculated already. It is therefore a much more efficient algorithm, though it does take up $\mathcal{O}(n)$ storage. The time complexity for this algorithm is only $\mathcal{O}(n)$ which is much better than Algorithm 5, so it would be more ideal to parallelize this version to get a larger benefit.

Algorithm 5 *Fibonacci*(n)

Input: n^{th} Fibonacci Number you're calculating
Output: Value of n^{th} Fibonacci Number

```
if n = 0 then
    return 0
else if n = 0 or n = 2 then
    return 1
else
    return Fibonacci(n − 1) + Fibonacci(n − 2)
end if
```

Algorithm 6 *Fibonacci*(A, n)

Input: Array $A \in \mathbb{R}^{n \times 1}$, n^{th} Fibonacci Number you're calculating
Output: Value of n^{th} Fibonacci Number

```
if n = 0 then
    return 0
else if n = 1 or n = 2 then
    A_n ← 1
    return 1
else if A_n ≠ 0 then
    return A_n
else
    A_n ← Fibonacci(A, n − 1) + Fibonacci(A, n − 2)
    return A_n
end if
```

Algorithm 7 (Parallel) $Fibonacci(A, n)$

Input: Array $A \in \mathbb{R}^{n \times 1}$, n^{th} Fibonacci Number you're calculating
Output: Value of n^{th} Fibonacci Number

```
if n = 0 then
    return 0
else if n = 1 or n = 2 then
    A_n ← 1
    return 1
else if A_n ≠ 0 then
    return A_n
else
    A_n^(1) ← Fibonacci(A, n − 1) {Done by original thread}
    A_n^(2) ← Fibonacci(A, n − 2) {Given to a new thread to be done in parallel}
    (Sync the threads)
    return A_n^(1) + A_n^(2) {From the original thread}
end if
```

Since this is being done in parallel, it the complexity would reduce down to $\mathcal{O}\left(n/p\right)$.

Question 4: Distributed Matrix Transpose . . . .

Suppose an $n \times n$ matrix is embedded in a hypercube (we assume that n is a power of 2). Find an algorithm for transposing this matrix in $\mathcal{O}(\log(n))$ time.

Hint: $2^k = n^2 = 2^{2q}$

Solution: The sequential case of Matrix Transpose takes $\mathcal{O}(n^2)$ time via the following algorithm

Algorithm 8 Sequential Matrix Transpose

Input: Matrix $A \in \mathbb{R}^{n \times n}$
Output: $A^T \in \mathbb{R}^{n \times n}$

for $i = 0$ **to** $n - 1$ **do**
  for $j = 0$ **to** $n - 1$ **do**
    $swap(A_{i,j}, A_{j,i})$
  end for
end for

For the distributed case, we first have to define the set of diagonals. This problem *assumes* that each node contains one index of each matrix. Therefore, the node numbers is in the set $\{1, 2, 3, \ldots, n \times n\}$. Furthermore, we can define the diagonals of the matrix as being in the set $\{n \times n, n \times (n-1) - 1, n \times (n-2) - 2, \ldots, n \times 1 - (n-1)\}$. For the algorithm, these will be denoted as "$Diag$."

Algorithm 9 Parallel Matrix Transpose

Input: $nodeID$ (rank of each node), n
Output: Transposed Matrix

if $nodeID \notin Diag$ **then**
  $column \leftarrow \left\lceil \frac{nodeID}{n} \right\rceil$
  $row \leftarrow nodeID$ **mod** n
  if $row = 0$ **then**
    $row \leftarrow n$
  end if
  $newColumn \leftarrow row$
  $newRow \leftarrow row$
  $otherNode \leftarrow n \times newColumn - (n - newRow)$
  Send($nodeID, otherNode$)
  Receive($otherNode, nodeID$)
end if

As the *maximum* communication time from one end of a hypercube to the other is $\mathcal{O}(log(n))$, then the time complexity of this algorithm is $\mathcal{O}(log(n))$. This above algorithm assumes that the node numbering is sequential along the row up through n. For example, the first column for 3×3 is $\{1, 2, 3\}$, the second column is $\{4, 5, 6\}$, and the third is $\{7, 8, 9\}$.

Question 5: Parallel Horner's Algorithm . . . .

Let $p(x) = a_0x^n + a_1x^{n-1} + \cdots + a_{n-1}x + a_n$ be a given polynomial. *Horner's Algorithm* can be used to compute $p(x)$ at a point x_0 is based on rewriting the expression for $p(x_0)$ as follows:

$$p(x_0) = (\cdots((a_0x_0 + a_1)\,x_0 + a_2)\,x_0 + \cdots + a_{n-1})\,x_0 + a_n$$

An obvious sequential algorithm for this problem has $\mathcal{O}(n)$ complexity. Is it possible to develop a work optimal parallel algorithm whose complexity is $\mathcal{O}(n/p + \log(n))$ for p processors? Give pseudocode for the best possible parallel algorithm.

Solution: The sequential code that was discussed in the question can be seen in the algorithm below, which is $\mathcal{O}(n)$.

Algorithm 10 Horner's Algorithm (Sequential)

Input: $a \in \mathbb{R}^{n\times 1}$, x
Output: $p(x)$

```
p ← 0
for i = n to 0 do
    p ← a_i + x · p
end for
return p
```

The operation can be done in $\mathcal{O}(n/p + \log(n))$, but it is not work optimal as it requires more work. In order to do so, take a look at the following example where $n = 40$ with 4 nodes. This will put 10 on each node.

$$\begin{gathered}
\left(a_{39}x^9 + a_{38}x^8 + \cdots + a_{30}\right)x^{30} \\
\left(a_{29}x^9 + a_{28}x^8 + \cdots + a_{20}\right)x^{20} \\
\left(a_{19}x^9 + a_{18}x^{18} + \cdots + a_{10}\right)x^{10} \\
\left(a_9x^9 + a_8x^8 + \cdots + a_1x + a_0\right)
\end{gathered}$$

From here, a reduction operation can be performed such to get the $\mathcal{O}(\log(n))$ complexity, as the above gives n/p. This is not work optimal as you have to do more work, but the depth is reduced.

Algorithm 11 Horner's Algorithm (Parallel)

Input: $a \in \mathbb{R}^{n\times 1}$, x
Output: $p(x)$

$p \leftarrow 0$
if $thread \neq 1$ **then**
  $high \leftarrow \frac{n}{p}(thread)$
  $low \leftarrow \frac{n}{p}(thread - 1)$
  $p \leftarrow \left(\sum_{i=low}^{high} a_i x^{i-low}\right) x^{low}$
else
  $p \leftarrow \left(\sum_{i=n/p}^{1} a_i x^{(n/p)-i}\right) x + a_0$
end if
return Reduce(p,Sum)