COMP 633: Parallel Computing
Fall 2021
Written Assignment 1: Sample Solutions

September 18, 2021

I. The Work-Time (W-T) presentation of EREW sequence reduction (Algorithm 2 in the PRAM handout) has work complexity $W(n) = O(n)$ and step complexity $S(n) = O(\lg n)$. Following the strategy of Brent's theorem, the translation of this algorithm will yield a p processor EREW PRAM program with running time $T_C(n, p) = O(n/p + \lg n)$.

(a) Construct an alternate sequence reduction algorithm directly for the bare bones EREW PRAM model with running time $T_C(n, p) = O(n/p + \lg p)$.

(b) Explain why your solution to (a) cannot be constructed by translation from a program in the W-T model.

Sample solution (a) Using a p processor PRAM we can compute the sum of an n element sequence in parallel by performing p sequential summations, each of size $\lceil n/p \rceil$, and combining the p results using parallel summation. This is a form of cascading.

The PRAM program on the left of the next page implements this strategy assuming that $n = 2^k$ and $p = 2^r$, for some $0 \leq r \leq k$. The PRAM program to its right assumes only that n and p are positive, hence is a complete implementation of sequence reduction. The input is sequence A with n elements of type T, and a binary associative operator $\oplus : T \times T \to T$ with left identity $0_\oplus$. The result is $s = \bigoplus_{j=1}^{n} A_j$. The processor id is i.

For the algorithm on the left, p evenly divides n. The order in which the elements from A and B are added in lines 7 and 12 corresponds to a left-to-right reduction of A whenever $\oplus$ is associative. The concurrent running time of this algorithm is

line	$T_C(n, p)$
4	$\Theta(1)$
5–8	$\Theta(n/p)$
10–14	$\Theta(\lg p)$
16–18	$\Theta(1)$
4–18	$\Theta(\frac{n}{p} + \lg p)$

```
1  local T B[1..p]
2  local integer h, j, ℓ
3
4  B[i] ← 0⊕
5  for ℓ = 1 to n/p do
6     j ← (i − 1)(n/p) + ℓ
7     B[i] ← B[i] ⊕ A[j]
8  enddo
9
10 for h = 1 to lg p do
11    if i ⩽ p/2^h then
12       B[i] ← B[2i − 1] ⊕ B[2i]
13    endif
14 enddo
15
16 if i = 1 then
17    s ← B[1]
18 endif
```

```
local T B[1..p]
local integer h, j, ℓ

B[i] ← 0⊕
for ℓ = 1 to ⌈n/p⌉ do
   j ← (i − 1)⌈n/p⌉ + ℓ
   if j ⩽ n then
      B[i] ← B[i] ⊕ A[j]
   endif
enddo
for h = 1 to ⌈lg p⌉ do
   if i ⩽ ⌈p/2^h⌉ then
      if 2i ⩽ ⌈p/2^(h−1)⌉ then
         B[i] ← B[2i − 1] ⊕ B[2i]
      else
         B[i] ← B[2i − 1]
      endif
   endif
enddo
if i = 1 then
   s ← B[1]
endif
```

The algorithm performs only exclusive reads, since all references to shared array A in line 7 and shared array B in line 12 are to unique elements within each iteration, and the reference to $B[1]$ in line 17 is performed by a single processor. The algorithm also performs only exclusive writes, since all assignments to B are to unique elements according to the processor index, and the assignment to s in line 17 is only performed by a single processor. Thus the PRAM model is EREW, as desired.

For the algorithm on the right, the same remarks apply concerning the EREW PRAM model. The time complexity is given by $T_C(n,p) = O(\lceil n/p \rceil + \lceil \lg p \rceil)$ which is still $O(n/p + \lg p)$. Note that when $p > n$, the $\lg p$ term dominates.

(b) The PRAM algorithms above cannot be expressed directly in the W-T model because the improved division of work involves the number of processors p, a quantity that is not part of the design, expression, and analysis of a W-T program. When $p = o(n)$, the algorithm given here is asymptotically faster than the PRAM algorithm obtained by the application of Brent's theorem to the W-T formulation of sequence reduction. Thus Brent's theorem applied to an optimal W-T algorithm does not always yield an optimal PRAM running time for the problem.

II. Given a sequence $s[1..n]$, the maximum contiguous subsequence sum (*mcss*) of s is the largest sum of any contiguous subsequence of s (including the empty subsequence, with sum zero), i.e.

$$\text{mcss} = \max\left(0, \max_{1 \le i \le j \le n} \sum_{k=i}^{j} s_k\right)$$

When all elements of s are positive the *mcss* is the sum of all elements in s. When all elements are negative, the *mcss* is zero, corresponding to the sum of an empty subsequence. The assignment includes an optimal sequential algorithm for this problem with $O(n)$ time complexity. Design a work efficient EREW algorithm in the Work-Time framework for this problem with step complexity $O(\lg n)$.

Divide and conquer solution. The sequential algorithm given in the assignment incrementally updates the *mcss* of successively larger prefixes of the input sequence by maintaining enough information to determine how inclusion of the next element alters the *mcss*. For the sequential algorithm it suffices to maintain the *mcss* of the prefix and the maximum segment sum that ends in the last position of the prefix.

We can consider a similar approach for a parallel algorithm, using a divide-and-conquer strategy on a balanced binary tree. Working upwards from the lowest level in the tree, solutions for consecutive subsections are combined at each level.

To simplify the presentation define $i{:}j$, where $i \le j$, to be the sequence of values $i, i+1, ..., j$, i.e. the range includes value j. Note that $i : i$ is the sequence with i as its only element, and there is no way to make an empty sequence. We can characterize the *mcss* of an arbitrary subsequence $s[i : j]$ of s using the tuple $R_{i:j} = (t_{i:j}, l_{i:j}, r_{i:j}, m_{i:j})$ where $t_{i:j} = \sum_{k \in i:j} s_k$ is the sum of all elements in the section, $l_{i:j}$ is the maximum prefix sum in the section, $r_{i:j}$ is the maximum suffix sum in the section, and $m_{i:j}$ is the maximum contiguous subsequence sum in the section.

Given $R_{i:j}$ and $R_{j+1:k}$ for two consecutive subsequences of s, we can compute $R_{i:k} = R_{i:j} \oplus R_{j+1:k}$ for the combined subsequence as follows

$$\begin{aligned} t_{i:k} &= t_{i:j} + t_{j+1:k} \\ l_{i:k} &= \max(l_{i:j}, t_{i:j} + l_{j+1:k}) \\ r_{i:k} &= \max(r_{j+1:k}, r_{i:j} + t_{j+1:k}) \\ m_{i:k} &= \max(m_{i:j}, m_{j+1,k}, r_{i:j} + l_{j+1:k}) \end{aligned}$$

In the last line, the *mcss* of two contiguous sections is the larger of the individual *mcss* of each section and the maximum sum subsequence that straddles the two sections. It is simple to verify that the $\oplus$ operation on consecutive subsequences of s is associative, i.e.

$$\forall\, 1 \le i \le j \le k \le \ell \le n : (R_{i:j} \oplus R_{j+1:k}) \oplus R_{k+1:\ell} = R_{i:j} \oplus (R_{j+1:k} \oplus R_{k+1:\ell})$$

Hence we can compute $R_{1:n} = \bigoplus_{i \in 1:n} R_{i:i}$ directly using a parallel reduction algorithm. For all $1 \leqslant i \leqslant n$, we initialize $R_{i:i} = (s_i, \max(0, s_i), \max(0, s_i), \max(0, s_i))$. The reduction will yield $R_{1:n}$, and within this tuple $mcss = m_{1:n}$.

The complexity of combining two tuples using $\oplus$ is $O(1)$ work and $O(1)$ steps under the EREW model, hence parallel reduction of n values has work complexity $W(n) = O(n)$ and step complexity $S(n) = O(\lg n)$ in the EREW model.

Data-parallel solution. It is possible to solve this problem using simple data parallel operations such as reversals, reductions, and parallel prefix sums on the input sequence. Possible, but not easy to derive!

For this approach, we need a couple of definitions. If w is a sequence of n elements with index domain $1..n$, let $\overline{w}$ denote the *reverse* of w, meaning $(\overline{w})_i = w_{n+1-i}$. For v and w integer sequences of equal length and $\oplus$ some binary operation on integers, let $v \hat{\oplus} w$ be the elementwise extension of $\oplus$ to corresponding elements of v and w, i.e. $\left(v \hat{\oplus} w\right)_i = v_i \oplus w_i$.

Now $mcss = \max(0, m)$ where

$$
\begin{aligned}
m &= \max_{1 \le i \le j \le n} \left(\sum_{k \in i:j} s_k \right) \\
&= \max_{i \in 1:n} \left(\max_{j \in i:n} \left(\sum_{k \in i:j} s_k \right) \right) \\
&= \max_{i \in 1:n} \left(\max_{j \in i:n} \left(s_i + \sum_{k \in 1:j} s_k - \sum_{k \in 1:i} s_k \right) \right) \\
&= \max_{i \in 1:n} \left(s_i - \sum_{k \in 1:i} s_k + \max_{j \in i:n} \left(\sum_{k \in 1:j} s_k \right) \right) \\
&= \max_{i \in 1:n} \left(s_i - \mathrm{prefix\_sum}(s)_i + \max_{j \in i:n} \left(\mathrm{prefix\_sum}(s)_j \right) \right) \\
&= \max_{i \in 1:n} \left(s_i - \mathrm{prefix\_sum}(s)_i + \overline{\mathrm{prefix\_max}(\overline{\mathrm{prefix\_sum}(s)})}_i \right) \\
&= \max_{i \in 1:n} \left(s \mathbin{\hat{-}} \mathrm{prefix\_sum}(s) \mathbin{\hat{+}} \overline{\mathrm{prefix\_max}(\overline{\mathrm{prefix\_sum}(s)})} \right)_i \\
&= \mathrm{reduce\_max} \left(s \mathbin{\hat{-}} \mathrm{prefix\_sum}(s) \mathbin{\hat{+}} \overline{\mathrm{prefix\_max}(\overline{\mathrm{prefix\_sum}(s)})} \right)
\end{aligned}
$$

In the final expression, *pref_sum*, *prefix_max*, and *reduce_max* are familiar parallel primitives with work complexity $O(n)$ and step complexity $O(\lg n)$ under the EREW model. Elementwise operations $\hat{-}$ and $\hat{+}$, as well as the reversal operation have work complexity $O(n)$ and step complexity $O(1)$ in the same model. Finally we construct *mcss* from m using constant work and steps.

In summary, the following program meets the desired complexity bounds.

integer MCSS(**sequence**⟨**integer** ⟩ s)

```
1 integer m, ss[1 : n], ssm[1 : n]
2 ss = prefix_sum(s)
3 ssm = reverse(prefix_max(reverse(ss)))
4 m = reduce_max(s ^+ ssm ^- ss )
5 return max(0,m)
```

III. Let A and B be sets of integers with $|A| = m \leq n = |B|$. The elements of the sets are stored in increasing order in arrays $A[1..m]$ and $B[1..n]$, respectively (since A and B are sets, there are no duplicate elements in either of these arrays). Using this representation, construct a CREW W-T algorithm to determine whether $A \subseteq B$ with $O(\lg n)$ step complexity and $O(n)$ work complexity.

Sample Solution. Since the sets are represented as sorted sequences, it is tempting to use a fast sequential binary search to determine in parallel whether every element of A occurs in B:

boolean $R[1:n]$
forall $i \in 1:m$ **do**
  R[i] $\leftarrow$ binarySearch(A[i],B[1:n])
enddo
return and_reduce($R[1:n]$)

However, the $O(m \lg n)$ work complexity of this algorithm is too high. We may improve the work complexity by cascading the algorithm above with an efficient sequential algorithm *seq_subset* that can determine whether $A' \subseteq B'$ in work and steps linear in $|A'| + |B'|$. The *seq_subset* algorithm sequentially advances through through B' (expected to be the more numerous set) while its current element is less than the current element in A'. It advances in both sequences when elements are equal, The algorithm reports failure if ever the current element in A' exceeds the current element in B', and reports success when A' is exhausted. As long as we can create independent subproblems of size no larger than $O(\lg n)$, we can use *seq_subset* on each subproblem in parallel without exceeding our target step complexity.

To partition A and B into such subproblems, we start by dividing B (the larger set) into chunks of size $\lg n$. For simplicity assume n is a power of 2 so $\lg n$ is integral, and assume $k = n/\lg n$ is also integral. Each chunk of B induces a matching chunk in A, whose extents can be found by binary search in A of the endpoint values of the B chunk, as shown in lines 3- 5 of the algorithm below. The matching chunks in A may be as short as zero elements and as long as n elements. If $A \subseteq B$, the length of each induced chunk of A should no greater than the length of the corresponding chunk of B (else A could not be a subset of B). We can check that this is true for all chunks in $O(\lg k)$ steps and $O(k)$ work, as shown in lines 6- 9.

Finally, since all our subproblems are now known to be of appropriate size, we may apply the *seq_subset* procedure to each of them in parallel (in lines 12- 14) and compute the overall result (in line 15). The algorithm follows. The input consists of arrays $A[1:m]$ and $B[1:n]$. The algorithm returns true iff $A \subseteq B$.

```
1  boolean bounds[0 : k], R[1 : k]
2  bounds[0] ← 1
3  forall i ∈ 1 : k do
4     bounds[i] ← binary_search_posn(B[i(lg n)], A[1 : m])
5  enddo
6  forall i ∈ 1 : k do
7     R[i] ← (bounds[i] − bounds[i − 1]) ⩽ lg n
8  enddo
9  if ¬and_reduce(R[1 : k]) then
10    return false
11 endif
12 forall i ∈ 1 : k do
13    R[i] ← seq_subset(A[bounds[i − 1] : bounds[i]], B[(i − 1)(lg n) + 1 : i(lg n)])
14 enddo
15 return and_reduce(R[1:k])
```

Complexity analysis:

line	$S(n)$	$W(n)$
3–5	$O(\lg m)$	$O(k \lg m)$
6–8	$O(1)$	$O(k)$
9	$O(\lg k)$	$O(k)$
12–14	$O(\lg n)$	$O(k \lg n)$
15	$O(\lg k)$	$O(k)$
3–15	$O(\lg n)$	$O(k \lg n) = O(n)$

The execution model is CREW because of potential concurrent reads of elements in A by parallel invocations of *binary_search_posn*. There are no concurrent writes.