CS4402 With elements of correction

Problem Set 1 (Tracks A & B)

CS9535 *Submission instructions on last page*

Problem 1. [Track A or B] The objective of this problem is to prove that, with respect to the Theorem of Graham & Brent, a greedy scheduler achieves the stronger bound:

$$T_P \leq (T_1 - T_\infty)/p + T_\infty.$$

Let $G = (V, E)$ be the DAG representing the instruction stream for a multithreaded program in the fork-join parallelism model. The sets V and E denote the vertices and edges of G respectively. Let T_1 and T_∞ be the work and span of the corresponding multithreaded program. We assume that G is connected. We also assume that G admits a single source (vertex with no predecessors) denoted by s and a single target (vertex with no successors) denoted by t. Recall that T_1 is the total number of elements of V and T_∞ is the maximum number of nodes on a path from s to t (counting s and t).

Let $S_0 = \{s\}$. For $i \geq 0$, we denote by S_{i+1} the set of the vertices w satisfying the following two properties:

(i) all immediate predecessors of w belong to $S_i \cup S_{i-1} \cup \cdots \cup S_o$,

(ii) at least one immediate predecessor of w belongs to S_i.

Therefore, the set S_i represents all the units of work which can be done during the i−-th parallel step (and not before that point) on infinitely many processors.

Let $p > 1$ be an integer. For all $i \geq 0$, we denote by w_i the number of elements in S_i. Let ℓ be the largest integer i such that $w_i \neq 0$. Observe that $S_0, S_1, \ldots, S_\ell$ form a partition of V. Finally, we define the following sequence of integers:

$$c_i = \begin{cases} 0 & \text{if } w_i \leq p \\ \lceil w_i/p \rceil - 1 & \text{if } w_i > p \end{cases}$$

Question 1. [4 points] For the computation of the 4-th Fibonacci number (as studied in class) what are $S_0, S_1, S_2, \ldots$?

Question 2. [4 points] Show that $\ell + 1 = T_\infty$ and $w_0 + \cdots + w_\ell = T_1$ both hold.

Answer. *For each $i = 0 \cdots \ell - 1$, the set S_{i+1} consists of strands which cannot be exexuted before those in $S_i \cup S_{i-1} \cup \cdots \cup S_o$ are exexuted. Therefore the span T_∞ is at least $\ell + 1$. On the other hand, all strands in S_{i+1} can be executed (concurrently) atfer those in $S_i \cup S_{i-1} \cup \cdots \cup S_o$ are exexuted. Therefore the T_∞ is at most $\ell + 1$. These two observations imply $\ell + 1 = T_\infty$.*

Since $S_0, S_1, \ldots, S_\ell$ form a partition of V, we clearly have $w_0 + \cdots + w_\ell = T_1$.

Question 3. [4 points] Show that we have:

$$c_0 + \cdots + c_\ell \leq (T_1 - T_\infty)/p.$$

Answer. *We havve*

$$\begin{array}{rcl} c_0 + \cdots + c_\ell & \leq & \sum_{i=0}^{i=\ell} \left(\lceil w_i/p \rceil - 1\right) \\ & \leq & \sum_{i=0}^{i=\ell} \left(w_i/p - 1/p\right) \\ & \leq & \frac{1}{p} \sum_{i=0}^{i=\ell} \left(w_i - 1\right) \\ & \leq & \frac{1}{p} \left(T_1 - T_\infty\right). \end{array} \tag{1}$$

Indeed, for every positive integer a, b, one can easily verify the following inequality

$$\lceil \frac{a}{b} \rceil - 1 \ \leq \ \frac{a-1}{b}. \tag{2}$$

Question 4. [4 points] Prove the desired inequality:

$$T_P \leq (T_1 - T_\infty)/p + T_\infty.$$

Answer. *We start by an interpretation of the quantity c_i:*

- *if $w_i \geq p$, that is, if one could perform at least one complete with the strands in S_i, then c_i counts the number of other steps (incomplete or complete) that can be done after that first complete step,*
- *if $w_i < p$, that is, if one can only perform one step (in fact, an incomplete one) with the strands in S_i, then $c_i = 0$*

Therefore, in all cases, c_i counts the number steps the number of other steps that can be done in S_i after that first one whether it is complete or incomplete. Hence $c_0 + \cdots + c_\ell = T_P - (\ell + 1)$. Recall that we have $\ell + 1 = T_\infty$. With the result of the previous question, we deduce the desired inequality

$$T_P - T_\infty \ \leq \ \frac{1}{p} \left(T_1 - T_\infty\right). \tag{3}$$

Question 5. [4 points] Application: Professor Brown takes some measurements of his (deterministic) multithreaded program, which is scheduled using a greedy scheduler, and finds that $T_4 = 80$ seconds and $T_{64} = 10$ seconds. Give lower bound and an upper bound for Professor Brown's computation running time on p processors, for $1 \leq p \leq 100$? Using a plot is recommended.

Problem 2. [Track A or B]

A **$\otimes$-reduction** of an array $x[1 \ldots n]$, where $\otimes$ is an associative operator, is the value

$$y = x[1] \otimes x[2] \otimes \ldots \otimes x[n].$$

The following procedure computes the $\otimes$-reduction of a subarray $x[i \ldots j]$ serially.

Algorithm 1: REDUCE(x, i, j)

```
1: y = x[i]
2: for k = i + 1 to j
3:     y = y ⊗ x[k]
4: return y
```

Question 1. [6 points] Use fork-joint parallelism to design a multithreaded algorithm P-REDUCE, which performs the same function with $\Theta(n)$ work and $\Theta(\lg n)$ span. Write your algorithm in pseudo-code and analyze your algorithm.

Answer. *The algorithm below answers the question. Clearly it has $\Theta(n)$ work and $\Theta(\lg n)$ span.*

Algorithm 2: P-REDUCE(x, i, j)

```
1: if i == j then return x[i]
2: k = (i + j)/2
3: a = span P-REDUCE(x, i, k)
4: b = P-REDUCE(x, k + 1, j)
5: sync
6: return a ⊗ b
```

A related problem is that of computing a **$\otimes$-prefix computation**, sometimes called a **$\otimes$-scan**, on an array $x[1 \ldots n]$, where $\otimes$ is once again an associative operator. The $\otimes$-scan produces the array $y[1 \ldots n]$ given by

$$
\begin{aligned}
y[1] &= x[1], \\
y[2] &= x[1] \otimes x[2], \\
y[3] &= x[1] \otimes x[2] \otimes x[3], \\
&\vdots \\
y[n] &= x[1] \otimes x[2] \otimes x[3] \otimes \cdots \otimes x[n],
\end{aligned}
$$

that is, all prefixes of the array x "summed" using the $\otimes$ operator. The following serial procedure SCAN performs a $\otimes$-prefix computation:

Algorithm 3: SCAN(x)

```
1: n = x.length
2: let y[1...n] be a new array
3: y[1] = x[1]
4: for i = 2 to n
5:     y[i] = y[i − 1] ⊗ x[i]
6: return y
```

Unfortunately, multithreading SCAN is not straightforward. For example, changing the **for** loop to a **parallel for** loop would create races, since each iteration of the loop body depends on the previous iteration. The following procedure P-SCAN-1 performs the $\otimes$-prefix computation in parallel, albeit inefficiently:

Algorithm 4: P-SCAN-1(x)

```
1: n = x.length
2: let y[1...n] be a new array
3: P-SCAN-1-AUX(x, y, 1, n)
4: return y
```

Algorithm 5: P-SCAN-1-AUX(x, y, i, j)

```
1: parallel for ℓ = i to j
2: y[ℓ] = P-REDUCE(x, 1, ℓ)
```

Question 2. [6 points] Analyze the work, span, and parallelism of P-SCAN-1.

Answer. *The P-SCAN-1-AUX algorithm has $\Theta(n^2)$ work and $\Theta(\lg n)$ span. (This follows immediately from the work and span of the algorithm P-REDUCE.) The same work and span estimates hold for P-SCAN-1.*

Algorithm 6: P-SCAN-2(x)

```
1: n = x.length
2: let y[1...n] be a new array
3: P-SCAN-2-AUX(x, y, 1, n)
4: return y
```

Algorithm 7: P-SCAN-2-AUX(x, y, i, j)

```
1: if i == j
2:     y[i] = x[i]
3: else k = ⌊(i + j)/2⌋
4:     spawn P-SCAN-2-AUX(x, y, i, k)
5:     P-SCAN-2-AUX(x, y, k + 1, j)
6:     sync
7:     parallel for ℓ = k + 1 to j
8:         y[ℓ] = y[k] ⊗ y[ℓ]
```

By using nested parallelism, we can obtain a more efficient $\otimes$-prefix computation:

Question 3. [6 points] Argue that P-SCAN-2 is correct, and analyze its work, span, and parallelism.

Answer. *First, we observe that the recursive calls operate on disjoint segments, preventing race conditions. Nex, one can verify by induction that the serialization of the routine is correct, because*

- *the left recursive call calculates the prefix sum up to the pivot, and*
- *the right recursive call calculates a prefix sum from the pivot to the upper bound, and,*
- *the parent call correctly combines the result of the left recursive call with each element from the right recursive call.*

Therefore, the parallel routine is correct as well, since its semanctics is identical to that of its serialization, provided that no race conditions are present.

The work satisifies $T_1(n) = 2T_1(n/2) + \Theta(n)$ *hence, we have* $T_1(n) \in \Theta(n \lg(n))$. *The span satisifies* $T_\infty(n) = T_\infty(n/2) + \Theta(\lg(n))$ *hence, we have* $T_\infty(n) \in \Theta(\lg^2(n))$. *Therefore, the parallelism is in* $\Theta(n/\lg(n))$, *which seems attractive. Unfortunately, this algorithm is not work-efficient since the best serial algorithm runs in* $\Theta(n)$.

We can improve on both P-SCAN-1 and P-SCAN-2 by performing the $\otimes$-prefix computation in two distinct passes over the data. On the first pass, we gather the terms for various

contiguous subarrays of x into a temporary array t, and on the second pass we use the terms in t to compute the final result y. The following pseudocode implements this strategy, but certain expressions have been omitted:

Algorithm 8: P-SCAN-3(x)

```
1: n = x.length
2: let y[1...n] and t[1...n] be new arrays
3:     y[1] = x[1]
4: if n > 1
5:     P-SCAN-UP(x, t, 2, n)
6:     P-SCAN-DOWN(x[1], x, t, y, 2, n)
7: return y
```

Algorithm 9: P-SCAN-UP(x, t, i, j)

```
1: if i == j
2:     return x[i]
3: else
4:     k = ⌊(i + j)/2⌋
5:     t[k] = spawn P-SCAN-UP(x, t, i, k)
6:     right = P-SCAN-UP(x, t, k + 1, j)
7:     sync
8:     return _____                    //fill in the blank
```

Algorithm 10: P-SCAN-DOWN(v, x, t, y, i, j)

```
1: if i == j
2:     y[i] = v ⊗ x[i]
3: else
4:     k = ⌊(i + j)/2⌋
5:     spawn P-SCAN-DOWN(_____, x, t, y, i, k)      //fill in the blank
6:     P-SCAN-DOWN(_____, x, t, y, k + 1, j)        //fill in the blank
7:     sync
```

Question 4. [6 points] Fill in the three missing expressions in line 8 of P-SCAN-UP and lines 5 and 6 of P-SCAN-DOWN. Argue that with expressions you supplied, P-SCAN-3 is correct. (Hint: Prove that the value v passed to P-SCAN-DOWN(v, x, t, y, i, j) satisfies $v = x[1] \otimes x[2] \otimes \cdots \otimes x[i-1]$.)

Answer. *The completed versions of P-SCAN-UP and P-SCAN-DOWN are shown below. For a nice expalantion, with illustration, look at the section* parallel algorithm *of*

http://en.wikipedia.org/wiki/Prefix_sum

as well as the book chapter

https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf

written by Guy E. Blelloch, the inventor of the P-SCAN-3 algorithm.

Algorithm 11: P-SCAN-3(x)

```
1: n = x.length
2: let y[1...n] and t[1...n] be new arrays
3:     y[1] = x[1]
4: if n > 1
5:     P-SCAN-UP(x, t, 2, n)
6:     P-SCAN-DOWN(x[1], x, t, y, 2, n)
7: return y
```

Algorithm 12: P-SCAN-UP(x, t, i, j)

```
1: if i == j
2:     return x[i]
3: else
4:     k = ⌊(i + j)/2⌋
5:     t[k] = spawn P-SCAN-UP(x, t, i, k)
6:     right = P-SCAN-UP(x, t, k + 1, j)
7:     sync
8:     return t[k] ⊗ right
```

Algorithm 13: P-SCAN-DOWN(v, x, t, y, i, j)

```
1: if i == j
2:     y[i] = v ⊗ x[i]
3: else
4:     k = ⌊(i + j)/2⌋
5:     spawn P-SCAN-DOWN(v, x, t, y, i, k)
6:     P-SCAN-DOWN(v ⊗ t[k], x, t, y, k + 1, j)
7:     sync
```

Question 5. [6 points] Analyze the work, span, and parallelism of P-SCAN-3.

Answer. *Clearly, each of P-SCAN-UP and P-SCAN-DOWN has* $\Theta(n)$ *work and* $\Theta(\lg n)$ *span. Then, so does P-SCAN-3.*

Problem 3. [Track A] This is an experimental follow-up on Problem 2.

Question 1. [5 points] Implement in `Cilk++` the four algorithms SCAN, P-SCAN-1, P-SCAN-2, P-SCAN3 as stated in Problem 2.

Question 2. [5 points] For P-SCAN-1, P-SCAN-2, P-SCAN3, collect experimental data using `Cilkview`. To this end use `int` arrays as input and multiplication as the operation $\otimes$.

Question 3. [5 points] Propose interpretation for the collected data and propose techniques (such as the use of a *serial base case*) to improve the performances of P-SCAN-2, P-SCAN3. (Indeed, P-SCAN-1 is not work-efficient so we discard it.)

Question 4. [5 points] Implement your improvements and collect data using `Cilkview`.

Problem 4. [Track B] This is a follow-up on Problem 2, regarding cache complexity.

Using the ideal cache model, with a cache of Z words and a cache line of L words, given an input array of length n, state and justify a cache complexity upper bound $Q(Z, L, n)$ for:

Question 1. [5 points] SCAN as stated in Problem 2.

Question 2. [5 points] P-SCAN-1 (as as stated in Problem 2 run on 1 processor.

Question 3. [5 points] P-SCAN-2 (as as stated in Problem 2 run on 1 processor.

Question 4. [5 points] P-SCAN-3 (as as stated in Problem 2 run on 1 processor.

Problem 5. [Track A or B]

The longest common subsequence (LCS) problem [1] is to find the longest subsequence common to two character strings. For example, the LCS of "BANANA" and "CANADA" is "ANAA". The problem can be recursively solved by breaking the sequence down into

[1] `http://en.wikipedia.org/wiki/Longest_common_subsequence_problem`

shorter subsequences. Denote by LCS(i, j) the LCS of sequences $X = (x_1, x_2, \ldots, x_i)$ and $Y = (y_1, y_2, \ldots, y_j)$. Then LCS$(i, j)$ can be computed by the following LCS function:

$$\text{LCS}(i,j) = \begin{cases} \emptyset & \text{if } i = 0 \text{ or } j = 0 \\ (\text{LCS}(i-1, j-1), x_i) & \text{if } x_i = y_j \\ \text{longest}(\text{LCS}(i, j-1), \text{LCS}(i-1, j)) & \text{if } x_i \neq y_j \end{cases}$$

where in the second case, $(\text{LCS}(i-1, j-1), x_i)$ means appending x_i to the end of the LCS of sequences $(x_1, x_2, \ldots, x_{i-1})$ and $(y_1, y_2, \ldots, y_{j-1})$.

Question 1. [5 points] Design a parallel algorithm to compute the **length** of LCS of two input sequences. You can assume that the input sequences have the same length. Describe your algorithm in pseudo-code, using of the algorithms in Problem . Finding the most efficient parallel algorithm is **not** required. **Hint:** you may consider a 2-way divide-and-conquer similar to the tableau construction (as studied in class).

Answer. *The parallel algorithm below computes the LCS of two input sequences x and y, with respective lengths i and j, and encoded as arrays with indexes in the range $0 \cdots (i-1)$ and $0 \cdots (j-1)$, respectively.*

Algorithm 14: LCS(x, y, i, j)

```
1: if i == 0 or j == 0
2:     return 0
3: else if x[i − 1] == y[j − 1]
4:     return 1+LCS(x, y, i − 1, j − 1)
5: else
6:     k = spawn LCS(x, y, i, j − 1)
7:     ℓ = LCS(x, y, i − 1, j)
8:     sync
9:     return max(k, ℓ)
```

Question 2. [5 points] Analyze the work, span and parallelism of your algorithm.

Answer. *This algorithm has a which is not higher than that of a tableau construction based on pattern of the Pascal Triangle. Therefore, the work is in $O(n^2)$. In the best scenario, that is $x = y$, the work is clearly linear in n, that is, $\Theta(n)$.*

Each recursive call reduces the sum of the sizes of x and y at least by 1 and at most by 2. Therefore, the span is in $\Theta(n)$.

It follows that the parallelism depends on the input sequences. It may be in $\Theta(n)$, if the sequences do not have large common subseqences. It may be in $\Theta(1)$, otherwise.

Problem 6. [Track A] This is an experimental follow-up on Problem 5.

Question 1. [20 points] Write a reasonably efficient Cilk++ program for your algorithm in Problem 5. By "reasonably efficient", we mean that your parallel program should run faster on $p = 2, 4, 6, 8$ processors than its serial counterpart (its C++ elision) for sufficiently large input data. Compare your Cilk++ program with its C++ elision with different sizes of inputs. Using a plot is highly recommenced, for each of $p = 2, 4, 6, 8$.

Question 2. [5 points, bonus] Is there another algorithm which is asymptotically as efficient in terms of work, but more parallel? If so, briefly describe the idea of the algorithm. If not, explain why. Make whatever interesting observations you can. You can consult the literature.

Problem 7. [Track B] This is an experimental follow-up on Problem 5.

Question 1. [20 points] Using the ideal cache model, with a cache of Z words and a cache line of L words, given an input array of length n, state (with justification) a cache complexity upper bound $Q(Z, L, n)$ for **a** serial algorithm computing the LCS function of Problem 5. Finding the most efficient algorithm in terms of cache complexity is **not** required. **Hint:** You may consider a 2-way divide-and-conquer similar to the tableau construction (as studied in class).

Question 2. [5 points, bonus] Is there another algorithm which is asymptotically as efficient in terms of work, but with a better cache complexity? If so, briefly describe the idea of the algorithm. If not, explain why. Make whatever interesting observations you can. You can consult the literature.

Submission instructions.

Format: The answers to the problem questions should be typed.

- If these are programs, input test files and a `Makefile` (for compiling and running) are required.
- If these are algorithms or complexity analyzes, LaTeX is highly recommended; in any case a PDF file should gather all these answers.

All the files should be archived using the UNIX `tar` command.

Submission: The assignment should be returned to the instructor by email.

Collaboration. You are expected to do this assignment *on your own* without assistance from anyone else in the class. However, you can use literatures and if you do so, briefly list your references in the assignment. Be careful! You might find on the web solutions to our problems which are not appropriate. For instance, because the parallelism model is different. So please, avoid those traps and work out the solutions by yourself. You should not hesitate to contact me if you have any questions regarding this assignment. I will be more than happy to help.

Marking. This assignment will be marked out of 100. A 10 % bonus will be given if your paper is clearly organized, the answers are precise and concise, the typography and the language are in good order. Messy assignments (unclear statements, lack of correctness in the reasoning, many typographical and language mistakes) may give rise to a 10 % malus.