The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 16

July 14, 2009

Exercise 185. Consider the following code for an in-place merge-sort:

```
void mergeSort(int[] A, int lo, int hi) {
  if (hi > lo) {
    int mid = (hi − lo)/2;
    executor.submit(new mergeSort(A, lo, mid));
    executor.submit(new mergeSort(A, mid+1, hi));
    awaitTermination();
    merge(A, lo, mid, hi);
}
```

Assuming that the merge method has no internal parallelism, give the work, critical path length, and parallelism of this algorithm. Give your answers both as recurrences and as $\Theta(f(n))$, for some function f.

solution Assuming that merge is $\Theta(n)$, we have $T_1(n) = 2T_1(n/2) + \Theta(n)$ hence $T_1(n) = \Theta(n \log n)$. $T_\infty(n) = T_1(n/2) + \Theta(n) = \Theta(n)$. The parallelism is $\Theta(\log n)$.

Exercise 186. You may assume that the actual running time of a parallel program on a dedicated P-processor machine is

$$T_P = T_1/P + T_\infty.$$

Your research group has produced two chess programs, a simple one and an optimized one. The simple one has $T_1 = 2048$ seconds and $T_\infty = 1$ second. When you run it on your 32-processor machine, sure enough, the running time is 65 steps. Your students then produce an "optimized" version with $T_1' = 1024$ seconds and $T_\infty = 8$ seconds. Why is it optimized? When you run it on your 32-processor machine, the running time is 40 steps, as predicted by our formula.

Which program will scale better to a 512-processor machine?

Solution. Plug in the different formulas:

$$T_{512} = \frac{T_1}{P} + T_\infty = \frac{2048}{512} + 1 = 5s$$
$$T'_{512} = \frac{T'_1}{P} + T'_\infty = \frac{1028}{512} + 8 = 10s$$

So, the "unoptimized" version runs significantly faster on a machine with more processors. The larger critical path length of the second implementation greatly impairs its performance.

Exercise 187. Write a class, ArraySum that provides a method

```
static public int sum(int[] a)
```

that uses divide-and-conquer to sum the elements of the array argument in parallel.

Solution See code folder.

Exercise 188. Professor Jones 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. What is the fastest that the professor's computation could possibly run on 10 processors? Use the following inequalities and the bounds implied by them to derive your answer. Note that P is the number of processors.

$$T_P \geq \frac{T_1}{P} \tag{0.0.1}$$

$$T_P \geq T_\infty \tag{0.0.2}$$

$$T_P \leq \frac{T_1 - T_\infty)}{P} + T_\infty \tag{0.0.3}$$

(The last inequality holds on a greedy scheduler.)

Solution. First, let's get an upper bound for T_1. From Equation 0.0.1:

$$T_4 \geq \frac{T_1}{4} \tag{0.0.4}$$

$$80 \geq \frac{T_1}{4} \tag{0.0.5}$$

$$320 \geq T_1. \tag{0.0.6}$$

Using T_{64} yields a worse bound:

$$T_{64} \geq \frac{T_1}{64} \tag{0.0.7}$$

$$10 \geq \frac{T_1}{64} \tag{0.0.8}$$

$$640 \geq T_1. \tag{0.0.9}$$

From Equation 0.0.2:

$$T_{64} \geq T_\infty$$

$$10 \geq T_\infty.$$

From Equation 0.0.3:

$$\begin{aligned} T_{10} &\leq \frac{(T_1 - T_\infty)}{10} + T_\infty \\ &\leq \frac{T_1}{10} + \frac{9 \cdot T_\infty)}{10} \\ &\leq \frac{320}{10} + \frac{9 \cdot T_\infty)}{10} \\ &\leq \frac{320}{10} + \frac{9 \cdot 10)}{10} \\ &\leq 32 + 9 \\ &\leq 41 \end{aligned}$$

Exercise 189. Give an implementation of the Matrix class used in this chapter. Make sure your split () method takes constant time.

Solution. See code folder for this chapter.

Exercise 190. Let $P(x) = \sum_{i=0}^{d} p_i x^i$ and $Q(x) = \sum_{i=0}^{d} q_i x^i$ be polynomials of degree d, where d is a power of 2. We can write

$$P(x) = P_0(x) + (P_1(x) \cdot x^{d/2})$$
$$Q(x) = Q_0(x) + (Q_1(x) \cdot x^{d/2})$$

where $P_0(x), P_1(x), Q_0(x)$, and $Q_1(x)$ are polynomials of degree $d/2$.

The Polynomial class provides put() and get() methods to access coefficients and it provides a constant-time split () method that splits a d-degree polynomial $P(x)$ into the two $(d/2)$-degree polynomials $P_0(x)$ and $P_1(x)$ defined above, where changes to the split polynomials are reflected in the original, and vice versa.

Your task is to devise parallel addition and multiplication algorithms for this polynomial class.

1. The *sum* of $P(x)$ and $Q(x)$ can be decomposed as follows:

$$P(x) + Q(x) = (P_0(x) + Q_0(x)) + (P_1(x) + Q_1(x)) \cdot x^{d/2}.$$

   (a) Use this decomposition to construct a task-based concurrent polynomial addition algorithm in the manner of Fig. **??**.

   (b) Compute the work and critical path length of this algorithm.

2. The *product* of $P(x)$ and $Q(x)$ can be decomposed as follows:

$$P(x) \cdot Q(x) = (P_0(x) \cdot Q_0(x)) + (P_0(x) \cdot Q_1(x) + P_1(x) \cdot Q_0(x)) \cdot x^{d/2} + (P_1(x) \cdot Q_1(x))$$

   (a) Use this decomposition to construct a task-based concurrent polynomial multiplication algorithm in the manner of Fig. **??**

   (b) Compute the work and critical path length of this algorithm.

Solution.

1. Polynomial addition:

   (a) For a task-based concurrent polynomial addition algorithm see this chapter's code folder.

   (b) The work for polynomial addition is given by the following recurrence:

$$\begin{aligned} A_1(n) &= 2A_1(n/2) + \Theta(1) \\ &= \Theta(n). \end{aligned}$$

   Solution.

   i. Polynomial addition:

      A. For a task-based concurrent polynomial addition algorithm see this chapter's code folder.

      B. The work for polynomial addition is given by the following recurrence:

$$\begin{aligned} A_1(n) &= 2A_1(n/2) + \Theta(1) \\ &= \Theta(n). \end{aligned}$$

      Adding two n-degree polynomials requres adding two half-size polynomials plus als requres adding two half-size polynomials plus a constant overhead.
      The critical path length for polynomial addition is given by the following recurrence:

$$\begin{aligned} A_\infty(n) &= A_\infty(n/2) + \Theta(1) \\ &= \Theta(\log n). \end{aligned}$$

      Adding the two half-size polynomials can be done in parallel.

   ii. Polynomial multiplication:

      A. For a task-based concurrent polynomial multiplication algorithm see this chapter's code folder.

      B. The work for polynomial multiplication is given by the following recurrence:

$$\begin{aligned} M_1(n) &= 4M_1(n/2) + 3A_1(n/2) + \Theta(1) \\ &= \Theta(n^2). \end{aligned}$$

      Multiplying two n-degree polynomials requres multiplying four half-size polynomials, adding three half-size polynomials, plus a constant overhead. (Multiplication by $x^{d/2}$ takes constant time.)

The critical path length for polynomial multiplication is given by the following recurrence:

$$\begin{aligned} M_\infty(n) &= M_\infty(n/2) + 2A_\infty(n/2) + \Theta(1) \\ &= \Theta(\log n). \end{aligned}$$

Multiplying the half-size polynomials can be done in parallel, followed by two parallel additions, followed by a single addition.

Exercise 191. Give an efficient and highly parallel multithreaded algorithm for multiplying an $n \times n$ matrix A by a length-n vector x that achieves work $\Theta(n^2)$ and critical path $\Theta(\log n)$. Analyze the

work and critical-path length of your implementation, and give the parallelism.

Exercise 192. Fig. **??** shows an alternate way of rebalancing two work queues: first, lock the larger queue, then lock the smaller queue, and rebalance if their difference exceeds a threshold. What is wrong with this code?

Solution. In the code in Figure **??**, the queues can change size between reading the first and second queue sizes. As a result two threads could lock the same two queues in different orders, resulting in deadlock.

A second way this code can deadlock if `q0` and `q1` have equal size, and different threads break the tie differently.

Exercise 193.

i. In the popBottom() method of Fig. **??**, the bottom field is volatile to assure that in popBottom() the decrement at Line **??** is immediately visible. Describe a scenario that explains what could go wrong if bottom were not declared as volatile.

ii. Why should we attempt to reset the bottom field to zero as early as possible in the popBottom() method? Which line is the earliest in which this reset can be done safely? Can our BoundedDEQueueoverflow anyway? Describe how.

Solution.

i. Since each thread observes its own updates to variables, the only way a problem can arise is between an owner A and a thief B. Suppose bottom is 3 and top is 4. A calls popBottom() and B calls popTop().
   - A sets bottom to 4 (Line 15), but the update remains invisible to other processes. Now the queue is empty.
   - A sets bottom and top to 0 (Lines 23 and 24), but the update to bottom remains invisible to other processes.
   - B sees top is 0 (Line 3) and bottom is 3. It incorrectly sets top to 1 (Line 8) and returns the meaningless item at position 0.

ii. When taking the last item, this program is correct as long as bottom is decremented before updating top (Line 24). Making this decrement visible as early as possible reduces the likelihood that a thief will execute an unsuccessful compareAndSet() (Line 8).

Exercise 194.

- In popTop(), if the compareAndSet() in Line **??** succeeds, it returns the element it read right before the successful compareAndSet() operation. Why is it important to read the element from the array before we do the compareAndSet()?
- Can we use isEmpty() in Line **??** of popTop()?

Solution.

- It is important to read the element from the array before doing the compareAndSet() because otherwise that array slot might be overwritten between the compareAndSet() and the read.
- We could use use isEmpty() in Line **??** of popTop(): both read top then bottom, and compare the two values.

Exercise 195. What are the linearization points of the `UnboundedDEQueue` methods? Justify your answers.

Solution.

- For pushBottom() the pushed item becomes visible to a thief when it modifies the circular buffer at Line 26.
- For a popTop() that returns a task, the linearization point occurs at Line 9, because that is where the task is actually removed from the queue.
- For a popTop() that returns *null*, the linearization point occurs at Line 4, when the bottom field is read. If that value is less than or equal to the previously-read top field, the method returns *null*.
- For a popBottom() that returns a task, the linearization point occurs at Line 27, that is where the item is removed.
- For a popBottom() that returns a task, the linearization point occurs either at Line 19 (if the queue is observed to be empty), or at Line 27 (if the queue is modified concurrently).

Exercise 196. Modify the popTop() method of the linearizable BoundedDEQueue implementation so it will return *null* only if there are no tasks in the queue. Notice that you may need to make its implementation blocking.

Solution. The popTop() method returns *null* if the queue is observed to be empty (Line 5), or if a concurrent update modifies the top field after it was read (Line 24). To prevent the second case, we must ensure the top field cannot be modified after being read. The simplest way to do this is to make the popTop() and popBottom() methods **synchronized** (or equivalently, have them acquire a lock).

Exercise 197. Do you expect that the isEmpty() method call of a BoundedDEQueue in the executer pool code will actually improve its performance?

Solution. Probably not, since the popTop() code does essentially the same thing.