Topic 22 Multithreading -- Solutions

Designing Parallel/Multithreaded Algorithms

In this problem we will design and implement a parallel algorithm for adding *n* numbers in an array. Consider the following code, for adding *n* numbers.

```
Iterative-Sum(A, n)
1 s = 0
2 for i = 1 to n
3     s = s + A[i]
4 return s
```

1. Parallelizing an iterative algorithm. First we will try to parallelize this algorithm directly.

```
Parallel-Iterative-Sum(A, n)
1 s = 0
2 parallel for i = 1 to n
3     s = s + A[i]
4 return s
```

(a) What is the work $T_1(n)$?

Θ(n) because in the sequential algorithm there is a loop that grows with n and has constant work inside the loop (the rest of the code is constant time).

(b) What is the span $T_\infty(n)$?

A naive analysis might say

Θ(1) because in the computation DAG the vertex for line 1 branches to n vertices for the parallel invocations of line 3, and these converge on the vertex for line 4. Thus the computation DAG has depth 3, or O(1).

However, simultaneous parallel branching to arbitrary n is not possible in general, and the concurrency platform compiler will use a divide & conquer strategy to implement the **parallel for** by dividing the problem in half, spawning one half the computation into a separate thread (see web notes or CLRS). So the more accurate answer is:

Θ(lg n) because line 2 is implemented with binary divide & conquer, resulting in a computation DAG of height lg n.

(c) What is the potential parallelism T_1/T_∞ ?

Θ(n) if we assume simultaneous parallel for execution.

Θ(n/lg n) if we assume divide & conquer implementation of parallel for.

(d) What could `Parallel-Iterative-Sum([3, 5],2)` return, and why? (It may be helpful to draw the computation DAG on paper or the whiteboard.)

3, 5 or 8.

- The only way it would return 8 if the actual execution were sequential: no other process accessed s between the retrieval of s from memory and writing it back out to memory.
- It could return 3 if the thread for i=1 executed "`s + A[i]` of line 3 first (before the other thread updated s) but then did the assignment "`s =`" and returned s after the thread for i=2 finished.
- It could return 5 if the thread for i=2 executed "`s + A[i]` of line 3 first (before the other thread updated s) but then did the assignment "`s =`" and returned s after the thread for i=1 finished.

What looked like an easy parallelization failed!

2. Parallelizing a recursive Divide & Conquer algorithm. Recursive algorithms using the Divide & Conquer strategy are among the easiest to multithread.

(a) Design a single threaded recursive Divide & Conquer algorithm to add *n* numbers (it has parameters for the start and end position of the array to be added, so we do not need n):

Notes: Any variation that works is OK.

```
Recursive-Sum(A, start, end)
1 if start == end
2     return A[start]
3 mid = floor((start + end)/2)
4 s1 = Recursive-Sum(A, start, mid)
5 s2 = Recursive-Sum(A, mid+1, end)
6 return s1 + s2
```

(b) Use **spawn** and **sync** keywords to parallelize your algorithm.

```
Parallel-Recursive-Sum(A, start, end)
if start == end
    return A[start]
mid = floor((start + end)/2)
spawn s1 = Recursive-Sum(A, start, mid)
s2 = Recursive-Sum(A, mid+1, end)
sync // what can go wrong if this is not here?
return s1 + s2
```

(c) Do you have to worry about race conditions? Why or why not? (If you do have race conditions, fix your algorithm until you can say why you don't have them!)

No, for two reasons: Most importantly, each parallel thread is operating on a disjoint partition of the input data. Also, unlike `s` in the iterative version, there are no race conditions on reading and writing `s1` and `s2` because in recursion each call has its own copy of these local variables.

(d) What is the work $T_1(n)$ of `Parallel-Recursive-Sum`?

$\Theta(n)$ because the recursion ends in $\Theta(n)$ leaves for the base cases, and since the recursion tree (computation DAG) is a full binary tree there are n-1 or $\Theta(n)$ internal nodes, each of constant time work to split the data in half and make the recursive calls. $\Theta(n) + \Theta(n) = \Theta(n)$.

(e) What is the span $T_\infty(n)$ of `Parallel-Recursive-Sum`?

$\Theta(\lg n)$ because this is the height of the recursion tree needed to divide the input array into half repeatedly until reaching single array elements. Of course, we do $\Theta(\lg n)$ work on the way down the recursion and $\Theta(\lg n)$ on the return, but this is $\Theta(2 \lg n) = \Theta(\lg n)$

(f) What is the potential parallelism T_1/T_∞ ?

$\Theta(n/\lg n)$

3. Comparing potential parallelism. Compare the parallelism of `Parallel-Iterative-Sum` in 1(c) to that of `Parallel-Recursive-Sum` in 2(f).

(a) Which has more parallelism, if any?

- If you assumed that parallel for could branch n computations simultaneously in constant time, then the iterative algorithm had more parallelism.
- If you assumed that parallel for required a divide & conquer implementation using spawn operators, then the two algorithms have equal parallelism.

(b) Which can be parallelized correctly, if any?

Only the recursive version could be parallelized correctly. The iterative version had serious problems with race conditions (any parallelism resulted in race condition errors).

(c) Why does this make sense that the answers (parallelism vs correctness) are different?

The numbers describing the structure of the computation DAG don't tell us whether the computation is *correct*, just how long it takes under parallelism. So it might be the case that an algorithm that has more or equal parallelism according to the theory is actually not viable as a parallel algorithm due to race conditions.