CSE341T/CSE549T **09/08/2014**

Matrix Multiplication I

Lecture 4

In the next two lectures, we will look at various algorithms for computing matrix multiplication and analyze the work, span, and parallelism of these algorithms.

Before we look at the problem of matrix multiplication, let's first look at how one can analyze the work and span of a simple parallel for loop, since we will use **parallel_for** in one of the algorithms for matrix multiplication.

1 Analyzing parallel_for loops

Say we have a simple **parallel_for** for copying elements of an array to another array:

```
1  parallel_for i ← 1 to n
2       do B[i] ← A[i]
```

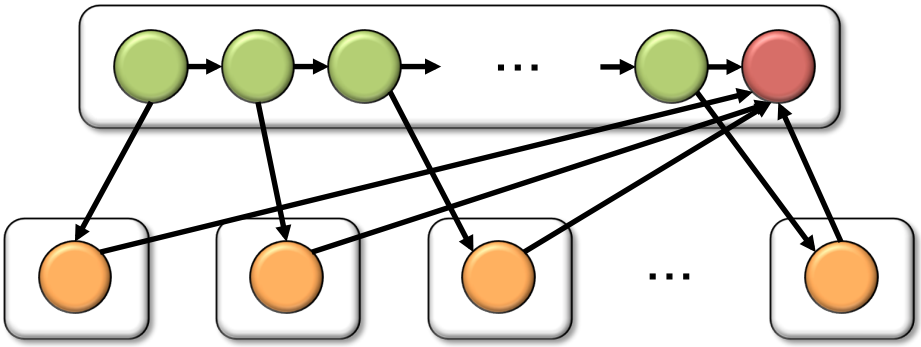
The **parallel_for** is just syntactic sugar, and is actually implemented using **spawn** and **sync**. So let's think about how we can do that. A **parallel_for** simply means that all iterations can potentially execute in parallel, so one simple way of implementing it is to spawn the iterations in a **for** loop:

```
COPYELEMT(A, B, i)
1  B[i] ← A[i]
2  return

3  for i ← 1 to n
4       do spawn COPYELEMT(A, B, i)
```

By spawning off each iteration, the code indicates that this iteration can execute in parallel with the rest of the **for** loop (the later iterations).

What is the work and span of this simple piece of code? It's easy to see the why that's the work and span by looking at the computation dag generated by this computation:



The work is $\theta(n)$, which is asymptotically the same as if we had removed the **spawn** keyword (albeit the spawning does increase the work by a constant amount compared to its serial counterpart, i.e., code without **spawn**). What about the span? The span is also $\theta(n)$ — the longest path follows the entire chain of nodes that spawn off each iteration plus a node that executes an iteration. Thus, this code sadly has PUNY amount of parallelism — $\theta(1)$! The fundamental problem here is that, this way of implmenting **parallel_for** causes the spawning of each iteration to depend on one another, and we are throwing away the parallelism among iteratsions. This may be O.K. if each iteration contains substantial amount of work, but that's not the case here.

Exercise 1 *What are the work and span in the code above, if we had replace* COPYELEMT *with some other function that has work and span of* $\theta(n)$*?*

Decreasing the span via coarsening

Well, there isn't much point to run a parallel program that has a parallelism of $\theta(1)$ (you might as well execute it serially!), so let's see if we can increase the parallelism by decreasing span.

Let's coarsen the base case so that now each time we spawn, we execute G iterations (G for ***grainsize***):

COPYMULTIPLEELEMTS$(A, B, start, end)$

1 **for** $j \leftarrow start$ **to** end
2 **do** $B[j] \leftarrow A[j]$
3 **return**

4 **for** $i \leftarrow 1$ **to** n **by** G
5 **do spawn** COPYMULTIPLEELEMTS$(A, B, i, MIN(i + G, n))$

With this strategy, work is $nT_{iter} + (n/G)T_{spawn}$, which is still $\theta(n)$. Even though we didn't asymptotically decrease the work, it is less overhead compared to no coarsening (which is $nT_{iter} + nT_{spawn}$).

Span has also decreased: $T_\infty = n/G + G$, and if we use $G = \sqrt{n}$, we get $T_\infty = \theta(\sqrt{n})$, which is asymptotically less than the span without coarsening.

In this case, since the work in each iteration is small, coarsening helps with parallelism, because we are removing the serial dependencies between each iteration. In general, coarsening tends to decrease the work but increase the span, and therefore decrease the parallelism. Depending on the computation, this can still help. In particular, if the computation already has plenty parallelism, it is a good idea to trade-off parallelism to decrease the work overhead due to **spawn**.

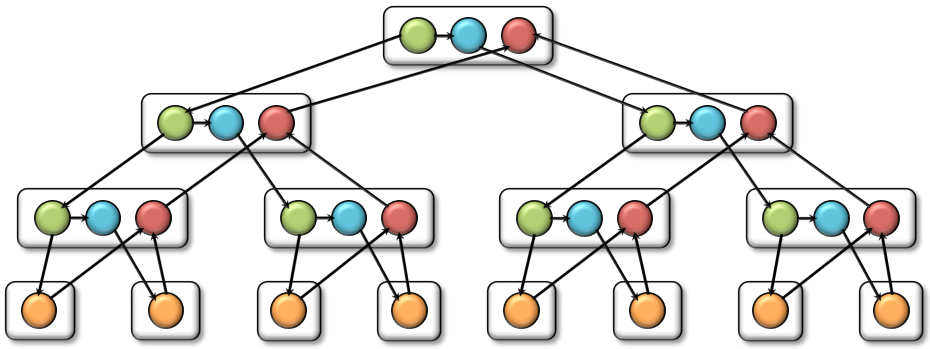
Binary splitting the iteration space

Besides coarsening, what else can we do to decrease the span? One strategy is to spawn-off parallel iterations in a binary tree:

```
COPYARRAY(A, B, start, end)
1  if start = end
2     then B[start] ← A[start]
3          return

4  mid ← ⌊(start + end)/2⌋
5  spawn COPYARRAY(A, B, start, mid)
6  COPYARRAY(A, B, mid + 1, end)
7  return
```

Assuming $n = 8$, this computation generates the following dag:



The work of this computation is again $\theta(n)$, which you can analyze by replacing the **parallel_for** with just a regular **for** . It may seem that we are not accounting for the spawning overhead when we analyze the work this way. In fact as we have seen earlier, the spawning does increase the work, but not asymptotically, because in a binary tree, the number of internal nodes equals to the number of leaf nodes plus one, and assuming we are doing constant amount of work per internal

node, we are not increasing the work asymptotically. Of course, if the work in each iteration is small, this is still substantial work overhead just to spawn off iterations.

Spawning off iterations in a binary tree indeed helps with the span. The span of this computation is just $\theta(\lg n) + \theta(1)$ (which is just $\theta(\lg n)$), because the part where the iterations are being spawned off has a recurrence of $T_\infty(n) = T_\infty(n/2) + \theta(1)$, which is $\theta(\lg n)$, and the span of a leaf node is just $\theta(1)$.

Exercise 2 *How would you coarsen* COPYARRAY*? What would be the work, span, and parallelism of coarsened* COPYARRAY*? Does coarsening help?*

General rule of thumb when writing parallel code:

- Try to make sure that the amount of work per spawn is large enough.
- If you have plenty of parallelism, try to trade some of it off to reduce work overhead from spawning.
- Use `cilk_for` (implemented using binary splitting) instead of a regular for loop with `cilk_spawn`.[1]
- If you have nested loops that you can parallelize, parallelize the outer loops as opposed to the inner loops, if you are forced to make a choice.

2 Matrix Multiplication (MM)

Now we are ready to look at matrix multiplication. Say we have two square $(n \times n)$ matrices A and B. To compute their product and store it into another $n \times n$ matrix C, simply do:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

which can be easily down by writing a triple-nested for loops. That means, it's also easily parallelizable using **parallel_for** loops:

```
1  let C be a new n × n matrix
2  parallel_for i ← 1 to n
3      do parallel_for j ← 1 to n
4          do C_ij ← 0
5             for k ← 1 to n
6                 do C_ij ← C_ij + A_ik · B_kj
```

[1] In Cilk Plus, there is a compiler pragma: `#pragma cilk grainsize = G` that goes with `cilk_for`, which tells the compiler to generate code that uses base case of G iterations.

The work of this computation is $\theta(n^3)$, same as the running time if we had a triple-nested serial loops. The span of this computation is $\theta(\lg n) + \theta(\lg n) + \theta(n)$, because it follows the path down the binary tree of the first outer **parallel_for** ($\theta(\lg n)$), then the binary tree of the second inner **parallel_for** ($\theta(\lg n)$), and finally the last inner-most serial **for** ($\theta(n)$), so the overall span is $\theta(n)$.

Reduce: computing sum of n elements in parallel

Note that we cannot use **parallel_for** for the inner-most **for** loop, because otherwise we will be racing on the memory location C_{ij}. If we could somehow compute the sum of these elements in parallel, then we can potentially reduce the span. A *reduce* operation is designed for exactly that purpose, so let's see how that works.

A ***reduce*** operation applies an *associative* function (such as sum, max, min, logical and, logical or ... etc) to a list or a sequence. We can compute a reduction in small work and span. For example, let's use reduction to computes the sum of all the elements of an array:

```
SUM(A, start, end)
1  if start = end
2     then return A[start]
3  mid ← ⌊(start+end)/2⌋
4  s1 ← spawn SUM(A, start, mid)
5  s2 ← SUM(A, mid + 1, end)
6  sync
7  return s1 + s2
```

The work of this procedure is $O(n)$ and the span is $O(\lg n)$. Note that we can perform any associative operation in the same way.

Using this SUM that employs a reduce operation, we can rewrite the matrix multiplication to achieve smaller span:

```
1  let C be a new n × n matrix
2  parallel_for i ← 1 to n
3       do parallel_for j ← 1 to n
4            do parallel_for k ← 1 to n
5                 do T[i] ← A_ik · B_kj
6               C_ij ← SUM(T, 1, n)
```

This code has the same work asymptotically, $\theta(n^3)$, and a smaller span $\theta(\lg n)$. Each **parallel_for** contributes $\theta(\lg n)$ to the span; the inner-most loop iteration is just $\theta(1)$; after the inner-most **parallel_for** is done, we call SUM, which has span $\theta(\lg n)$. Thus, this program has parallelism of $\theta(n^3/\lg n)$, which is pretty good.

Exercise 3 *Is this the only way to parallelize matrix multiplication? What else can you do?*