CSE 539 **01/22/2015**

Parallel Algorithms

Lecture 3 *Scribe: Angelina Lee*

Outline of this lecture:

1. Implementation of **cilk_for**
2. Parallel Matrix Multiplication

1 Implementation of cilk_for

We mentioned in last lecture that Cilk also support **cilk_for**, which allows iterations to execute in parallel:

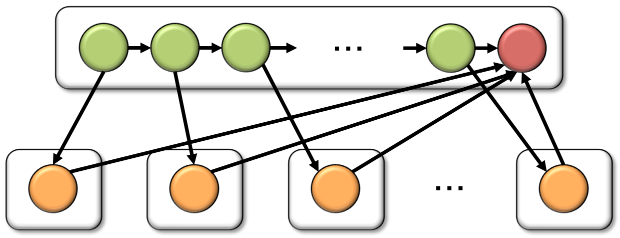
```
1  cilk_for $i \leftarrow 1$ to $n$
2      do FOO($i$)
```

The **cilk_for** is just syntactic sugar, and is actually implemented using **cilk_spawn** and **cilk_sync**. So let's think about how we can do that. A **cilk_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:

```
1  for $i \leftarrow 1$ to $n$
2      do cilk_spawn FOO($i$)
3  cilk_sync
```

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? For now let's assume FOO has constant amount of work and span (so it's a piece of serial code). This piece of code generates the following computation dag:



The work is $\Theta(n)$, which is asymptotically the same as if we had removed the **cilk_spawn** keyword (albeit the spawning does increase the work by a constant amount compared to its serial elision, i.e., code without **cilk_spawn** and **cilk_sync**).

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 implementing **cilk_for** causes the spawning of each iteration to depend on one another, and we are throwing away the parallelism among iterations. This may be O.K. if each iteration contains substantial amount of work, but the spawning overhead dominates if each iteration has little work.

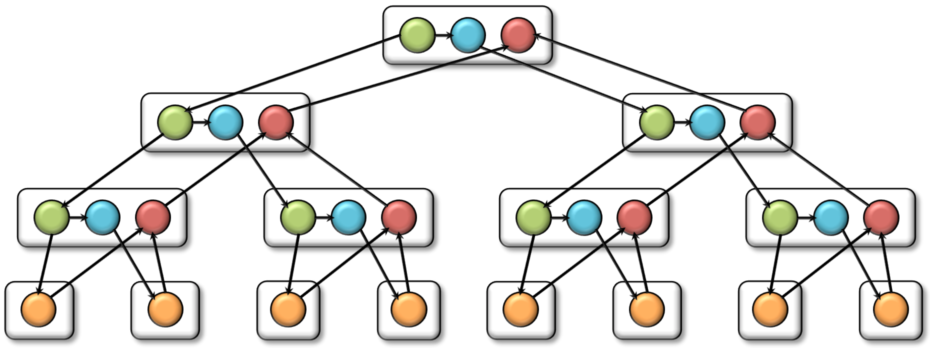
Binary splitting the iteration space

What can we do to decrease the span? One strategy is to spawn-off parallel iterations in a binary tree:

```
DoCilkFor($start, end$)
1 if $(end - start) = 1$
2     then foo($star$)
3 return

4 $mid \leftarrow \lfloor (start + end)/2 \rfloor$
5 cilk_spawn DoCilkFor($start, mid$)
6 DoCilkFor($mid, end$)
7 cilk_sync
8 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 **cilk_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 $\Theta(1)$.

Coarsening to decrease spawn overhead

If each iteration has little work, in practice, it may be a good idea to coarsen your base case in order to decrease spawn (and recursion) overhead. That is, instead of recursing down to base case of one iteration, we coarsen the base case to do, say, G iteration:

```
DOCILKFOR($start, end$)
1  if $(end - start) \le G$
2     then for $i \leftarrow 1$ to $G$
3              do FOO($i + start$)
4           return

5  $mid \leftarrow \lfloor (start + end)/2 \rfloor$
6  cilk_spawn DOCILKFOR($start, mid$)
7  DOCILKFOR($mid, end$)
8  cilk_sync
9  return
```

What's the work and span in this case? The work should be the same — we are still doing the same amount of work regardless of whether we are doing them in parallel or not (although we did decrease the spawn overhead slightly, which is now $\Theta(n/G)$, but that doesn't change the overall work asymptotically). Thus, the work is still $\Theta(n)$, assuming work of FOO is constant. The span, on the other hand, is $\Theta(\lg(n/G) + G)$, because we are only spawning down to a leaf with size G, with n/G leaves, so the height of the tree is now $\lg(n/G)$, but each leaf now has a span of $\Theta(G)$, since we are doing G iterations serially. Note that this decreases the overall parallelism, but in practice, it may pay off, because we are saving some spawn overhead and recursive call overhead. General rule of thumb when writing parallel code in practice:

- 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 spawn overhead.
- 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.

Avoiding races

We haven't formally define different kinds of races, but for now, think of a ***race*** as two logically parallel subcomputations that access the same memory location, with at least one of the access being a write. For instance, the following code has a race:

[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.

```
**cilk_for** $i \leftarrow 1$ **to** $n$
    **do** $x \leftarrow x + array[i]$
```

In particular, each iteration is doing the following in parallel:

```
r ← read x
r ← r + 1
write x ← r
```

If the instructions interleave in certain way, we may get ***lost-update***, where we are missing some increments. (For instance, both iteration reads x at the same time and thus getting the same value for x.) As an homework exercise, think about how one can avoid race by rewriting this code without using any lock or reducer.[2]

2 Parallel Matrix Multiplication

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 done by writing a triple-nested for loops. That means, it's also easily parallelizable using **cilk_for** loops:

```
let $C$ be a new $n \times n$ matrix
**cilk_for** $i \leftarrow 1$ **to** $n$
    **do cilk_for** $j \leftarrow 1$ **to** $n$
            **do** $C_{ij} \leftarrow 0$
               **for** $k \leftarrow 1$ **to** $n$
                   **do** $C_{ij} \leftarrow C_{ij} + A_{ik} \cdot B_{kj}$
```

Note that we cannot parallelize the inner-most loop, since each iteration writes to the same C_{ij}, and we would cause a race if we parallelize the inner-most loop.

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 **cilk_for** ($\Theta(\lg n)$), then the binary tree of the second inner **cilk_for** ($\Theta(\lg n)$), and finally the last inner-most serial **for** ($\Theta(n)$), so the overall span is $\Theta(n)$.

[2] Reducer is a linguistic mechanism supported in various dialect of Cilk that allow parallel subcomputations to update the same non-local variable; we will cover its implementation in more depth later in the class.

Solving matrix multiply using divide-and-conquer technique

Another to parallelize matrix multiplication is to use a recursive divide-and-conquer algorithm (as a side note, a lot of parallel algorithms employ divide-and-conquer technique). This algorithm uses the following formulation, where matrix A multiplies matrix B to produce a matrix C:

$$\begin{aligned}\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} &= \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \\ &= \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.\end{aligned}$$

This suggests a straightforward divide-and-conquer algorithm. You can compute all 8 parts in parallel and then add them:

```
MULT($C$, $A$, $B$, $n$)
 1  if $n = 1$
 2      then $c_{11} \leftarrow a_{11}b_{11}$ return

 3  partition $A$, $B$, and $C$, into 4 sub-matrices
 4  create $T$, a temporary $n \times n$ matrix

 5  cilk_spawn MULT($C_{11}$, $A_{11}$, $B_{11}$, $n/2$)
 6  cilk_spawn MULT($C_{12}$, $A_{11}$, $B_{12}$, $n/2$)
 7  cilk_spawn MULT($C_{21}$, $A_{21}$, $B_{11}$, $n/2$)
 8  cilk_spawn MULT($C_{22}$, $A_{21}$, $B_{12}$, $n/2$)
 9  cilk_spawn MULT($T_{11}$, $A_{12}$, $B_{21}$, $n/2$)
10  cilk_spawn MULT($T_{12}$, $A_{12}$, $B_{22}$, $n/2$)
11  cilk_spawn MULT($T_{21}$, $A_{22}$, $B_{21}$, $n/2$)
12  MULT($T_{22}$, $A_{22}$, $B_{22}$, $n/2$)
13  cilk_sync
```

Note that you have to do the last four MULT calls with a temporary matrix T, because otherwise you would have a race on C, since both parts need to sum the result back to C. Then you must add the pairwise matrices (the combine step).

The analysis of the algorithms in this section requires the use of the Master Theorem. We state the Master Theorem here for convenience.

Theorem 1 (Master Theorem) *Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence*

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. *If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.*

2. *If $f(n) = \Theta(n^{log_b a} lg^k n)$ for some constant $k \geq 0$, then $T(n) = \Theta(n^{log_b a} lg^{k+1} n)$.*

3. *If $f(n) = \Omega(n^{log_b a+\epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.*

We begin by analyzing the work for MULT. The work is the running time of the algorithm on one processor, which we compute by solving the recurrence relation for the serial equivalent of the algorithm. We note that the matrix partitioning in MULT takes $O(1)$ time, as it requires only a constant number of indexing operations.

The combine step, adding pair-wise matrices can be done in parallel, either using two **cilk_for** loops or by dividing into 4 parts and doing it recursively. Either way, the work of adding 2 $n \times n$ matrices is $\theta(n^2)$ and the span is $\theta(\lg n)$.

Thus, the recurrence for the work of MULT (denoted $M_1(n)$) is:

$$\begin{aligned} M_1(n) &= 8M_1(n/2) + \Theta(n^2) && (1)\\ &= \Theta(n^3). && (2)\end{aligned}$$

We solve this recurrence with case 1 of the Master Theorem. The work is the same as for the traditional triply-nested-loop serial algorithm.

The span is the maximum path length through a computation. Since the span of the combine step is $\Theta(\lg n)$, the recurrence for the span of MULT (denoted $M_\infty(n)$) is:

$$\begin{aligned} M_\infty &= M_\infty(n/2) + \Theta(\lg n) && (3)\\ &= \Theta(\lg^2 n), && (4)\end{aligned}$$

by case 2 of the Master Theorem. From the work and span, we compute the parallelism:

$$M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n). \qquad (5)$$

As an example, if n = 1000, the parallelism $\approx 10^7$. In practice, multiprocessor systems don't have more that $\approx 64,000$ processors, so the algorithm has more than adequate parallelism.

In fact, it is possible to trade parallelism for an algorithm that runs faster in practice. MULT may run slower than an in-place algorithm because of the memory usage (notice how it needs a temporary matrix at every level of recursion). We introduce a new algorithm, MULTADD, that trades parallelism in exchange for eliminating the need for the temporary matrix T. The algorithm do the first 4 recursive calls first, sync, and then do the remaining 4 recursive calls:

```
MultAdd($C$, $A$, $B$, $n$)
if $n = 1$
    then $c_{11} \leftarrow c_{11} + a_{11}b_{11}$ return

partition $A$, $B$, and $C$, into 4 sub-matrices
cilk_spawn MultAdd($C_{11}$, $A_{11}$, $B_{11}$, $n/2$)
cilk_spawn MultAdd($C_{12}$, $A_{11}$, $B_{12}$, $n/2$)
cilk_spawn MultAdd($C_{21}$, $A_{21}$, $B_{11}$, $n/2$)
MultAdd($C_{22}$, $A_{21}$, $B_{12}$, $n/2$)
cilk_sync
cilk_spawn MultAdd($C_{11}$, $A_{12}$, $B_{21}$, $n/2$)
cilk_spawn MultAdd($C_{12}$, $A_{12}$, $B_{22}$, $n/2$)
cilk_spawn MultAdd($C_{21}$, $A_{22}$, $B_{21}$, $n/2$)
MultAdd($C_{22}$, $A_{22}$, $B_{22}$, $n/2$)
cilk_sync
```

The work for MULTADD (denoted $M'_1(n)$) is the same as the work for MULT, $M'_1(n) = \Theta(n^3)$. Since the algorithm now executes four recursive calls in parallel followed in series by another four recursive calls in parallel, the span (denoted $M'_\infty(n)$) is

$$\begin{aligned} M'_\infty(n) &= 2M'_\infty(n/2) + \Theta(1) & (6) \\ &= \Theta(n) & (7) \end{aligned}$$

by case 1 of the Master Theorem. The parallelism is now

$$M'_1(n)/M'_\infty = \Theta(n^2). \tag{8}$$

When n = 1000, the parallelism $\approx 10^6$, which is still quite high.

In practice, even though MULTADD has less parallelism, it has more than adequate parallelism and uses less space. Thus, it likely performs better in practice. Moreover, when you write a matrix multiplication code in practice, you probably don't want to recurs down to the base case of size 1. Instead, it's a good idea to coarsen your base case. As an exercise homework, think about what the work and span will be if you coarsen the base case.