Today, 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:

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

3 for i ← 1 to n
4 do spawn COPYELEM(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 implementing `parallel_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 that’s not the case here.

**Exercise 1** What are the work and span in the code above, if we had replace `COPYELEM` 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 ← start to end
3  return

4  for i ← 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 \texttt{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:

\begin{verbatim}
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
\end{verbatim}

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

![Binary Tree Diagram]

The work of this computation is again $\theta(n)$, which you can analyze by replacing the \texttt{parallel_for} with just a regular \texttt{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`.
- 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 done by writing a triple-nested for loops. That means, it’s also easily parallelizable using `parallel_for` loops:

```c
1  let C be a new n x 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} \cdot B_{kj}
```

1In 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 $\theta(\log n) + \theta(\log n) + \theta(n)$, because it follows the path down the binary tree of the first outer `parallel_for` ($\theta(\log n)$), then the binary tree of the second inner `parallel_for` ($\theta(\log n)$), 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:

```plaintext
SUM(A, start, end)
1 if start = end
2 then return A[start]
3 mid ← \left\lceil \frac{start+end}{2} \right\rceil
4 s_1 ← spawn SUM(A, start, mid)
5 s_2 ← SUM(A, mid + 1, end)
6 sync
7 return s_1 + s_2
```

The work of this procedure is $O(n)$ and the span is $O(\log n)$. 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:

```plaintext
1 let $C$ be a new $n \times n$ 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 $T[i] ← A_{ik} \cdot B_{kj}$
6 $C_{ij} ← SUM(T, 1, n)$
```

This code has the same work asymptotically, $\theta(n^3)$, and a smaller span $\theta(\log n)$. Each `parallel_for` contributes $\theta(\log n)$ 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 $\theta(\log n)$. Thus, this program has parallelism of $\theta(n^3 / \log n)$, which is pretty good.
3 Divide and Conquer Algorithms

We have already seen a couple of divide and conquer algorithms in this lecture. The reduce algorithm and the algorithm to copy elements of the array are both D&C algorithms. We will use this technique a lot in this class, so let's formalize it a bit.

Divide and conquer algorithms generally have 3 steps: divide the problem into subproblems, recursively solve the subproblems and combine the solutions of subproblems to create the solution to the original problem.

The structure of a divide-and-conquer algorithm follows the structure of a proof by (strong) induction. This makes it easy to show correctness and also to figure out cost bounds. The general structure looks as follows:

— **Base Case:** When the problem is sufficiently small, return the trivial answer directly or resort to a different, usually simpler, algorithm, which works great on small instances.

— **Inductive Step:** First, the algorithm divides the current instance \( I \) into parts, commonly referred to as *subproblems*, each smaller than the original problem. Then, it recurses on each of the parts to obtain answers for the parts. In proofs, this is where we assume inductively that the answers for these parts are correct, and based on this assumption, it combines the answers to produce an answer for the original instance \( I \).

This technique is even more useful for parallel algorithms. Generally, you can solve the subproblems in parallel. If the divide and combine step is inexpensive, then you are done. If either the divide and combine step (or both) is expensive, however, you may want to parallelize it (them), which can be difficult, depending on the algorithms.

Let us assume that the subproblems can be solved independently. Say the problem of size \( n \) is broken into \( k \) subproblems of size \( n_1, \ldots, n_k \). How would you write the Cilk Plus program? What if \( k = 2 \)?

```cilk
F(n)
1 if n \leq n_0
2 then Base-Case
3 return
4 Divide into 2 parts of size \( n_1 \) and \( n_2 \)
5 spawn F(n_1)
6 F(n_2)
7 sync
8 Combine.
```
With the strategy above, the work is

\[ W(n) = W_{\text{divide}}(n) + W(n_1) + W(n_2) + W_{\text{combine}}(n) \]

And the span is

\[ S(n) = S_{\text{divide}}(n) + \max\{S(n_1), S(n_2)\} + S_{\text{combine}}(n) \]

Note that the work recurrence is simply adding up the work across all components. More interesting is the span recurrence: First, note that a divide and conquer algorithm has to split a problem instance into subproblems before these subproblems are recursively solved. Furthermore, it cannot combine the results from these subproblems to generate the ultimate answer until the recursive calls on the subproblems are complete. This forms a chain of sequential dependencies, explaining why we add their span together. The parallel execution takes place among the recursive calls since we assume that the subproblems can be solved independently — this is why we take the \( \max \) over the subproblems’ span.

Now consider arbitrary \( k \). What is the pseudocode?

```plaintext
F(n)
1  if n ≤ n₀
2      then Base-Case
3      return
4  Divide into \( k \) parts of size \( n_1, n_2, \ldots, n_k \)
5  parallel_for i ← 1 to k
6      do F(nᵢ)
7  Combine.

W(n) = W_{\text{divide}}(n) + \sum_{i=1}^{k} W(n_i) + W_{\text{combine}}(n)
```

And the span is

\[ S(n) = S_{\text{divide}}(n) + +\log k + \max_{i=1}^{k} \{S(n_i)\} + S_{\text{combine}}(n) \]

Applying this formula results in familiar recurrences such as \( W(n) = 2W(n/2) + O(n) \). In the rest of this lecture, we’ll get to see other recurrences—and learn how to derive a closed-form for them.
4 Solving matrix multiplication using divide and conquer

It turns out that, one can solve matrix multiplication using divide and conquer. You can divide your matrix into 4 quarters and get the following:

\[
C_{11} = A_{11}B_{11} + A_{12}B_{21} \\
C_{12} = A_{11}B_{12} + A_{12}B_{22} \\
C_{21} = A_{21}B_{11} + A_{22}B_{21} \\
C_{22} = A_{21}B_{12} + A_{22}B_{22}
\]

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

```plaintext
MM(C, A, B, n)
1  if n = 1
2      then c_{11} ← a_{11}b_{11} return
3  partition A, B, and C, into 4 submatrices
4  create T, a temporary n × n matrix
5  spawn MM(C_{11}, A_{11}, B_{11}, n/2)
6  spawn MM(C_{12}, A_{11}, B_{12}, n/2)
7  spawn MM(C_{21}, A_{21}, B_{11}, n/2)
8  spawn MM(C_{22}, A_{21}, B_{12}, n/2)
9  spawn MM(T_{11}, A_{12}, B_{21}, n/2)
10 spawn MM(T_{12}, A_{12}, B_{22}, n/2)
11 spawn MM(T_{21}, A_{22}, B_{21}, n/2)
12 spawn MM(T_{22}, A_{22}, B_{22}, n/2)
13 sync
14 parallel_for i ← 1 to n
15      do parallel_for j ← 1 to n
16          do c_{ij} ← c_{ij} + t_{ij}
```

Then you must add the pairwise matrices (the combine step). Adding pair-wise matrices can also be done in parallel, either using 2 for loops or by dividing into 4 parts and doing it recursively. Either way, the work of adding 2 n × n matrices is θ(n²) and the span is θ(lg n).

Therefore, the work of the overall algorithm is \( T_1(n) = 8T_1(n/2) + \theta(n^2) = \theta(n^3) \), and the span is \( T_\infty(n) = T_\infty(n/2) + \theta(lg n) = \theta(lg^2 n) \).

How about if you didn’t want to create temporary matrices? You can do 4 recursive calls, sync, and the do the remaining 4 recursive calls. Doing so rid of the combine step. You get work \( T_1(n) = 8T_1(n/2) + \theta(1) \), which is still \( \theta(n^3) \) and span \( T_\infty(n) = 2T_\infty(n/2) + \theta(1) = \theta(n) \),
which is less but generally adequate parallelism. Even though this version of matrix multiply has
less parallelism, but in practice, $\theta(n^2)$ is plenty of parallelism — even on a small input (e.g., 1000
by 1000 matrices), the amount of parallelism is already $10^6$. With even larger input, you would
have more parallelism than you know what to do with. Thus, in practice, it’s actually better to
use the version without the temporary, because it actually has smaller work overhead, albeit by
constant amount due to less memory usage, that constant indeed makes a difference in running
time in actual implementation.

**Strassen’s method**

Another interesting matrix multiplication algorithm is Strassen’s method. First, you divide input
matrices into 4 submatrices respectively. Then you create 10 temporary matrices by adding or
subtracting the input submatrices (think of this as part of divide step):

\[
S_1 = B_{12} - B_{22} \\
S_2 = A_{11} + A_{12} \\
S_3 = A_{21} + A_{22} \\
S_4 = B_{21} - B_{11} \\
S_5 = A_{11} + A_{22} \\
S_6 = B_{11} + B_{22} \\
S_7 = A_{12} - A_{22} \\
S_8 = B_{21} + B_{22} \\
S_9 = A_{11} - A_{21} \\
S_{10} = B_{11} + B_{12}
\]

Then you can recursively spawn the following subcomputations:

\[
P_1 = A_{11} \cdot S_1 \\
P_2 = S_2 \cdot B_{22} \\
P_3 = S_3 \cdot B_{11} \\
P_4 = A_{22} \cdot S_4 \\
P_5 = S_5 \cdot S_6 \\
P_6 = S_7 \cdot S_8 \\
P_7 = S_9 \cdot S_{10}
\]
Finally, you combine the results from the subcomputations:

\[
C_{11} = P_5 + P_4 - P_2 + P_6 \\
C_{12} = P_1 + P_2 \\
C_{21} = P_3 + P_4 \\
C_{22} = P_5 + P_1 - P_3 - P_7
\]

**Exercise 3**  *Parallelize Strassen’s algorithm and compute its work and span.*