Before we consider matrix multiplication, let's see what happens when for-loops have a larger size iterations. Recall the dag for parallel for loop. The span of this dag is $O(\lg n)$ since the number of levels is $O(\lg n)$ and you walk down the tree once, you take a horizontal link once, and then you walk up the tree again.

What if each of the leaves were $O(m)$ work instead of $O(1)$ work? What is the work and span of the computation? By the same logic, we would spend $O(\lg n)$ span in walking up and down the tree and $O(m)$ in the leaves, leading to the total span of $O(m + \lg n)$.

Now we can analyze the matrix multiplication code we saw in the first lecture — the triply nested for loop.

1. let $C$ be a new $n \times n$ matrix
2. parallel_for $i \leftarrow 1$ to $n$
3. \hspace{1em} do parallel_for $j \leftarrow 1$ to $n$
4. \hspace{2em} do $C_{ij} \leftarrow 0$
5. \hspace{1em} for $k \leftarrow 1$ to $n$
6. \hspace{2em} do $C_{ij} \leftarrow C_{ij} + A_{ik} \cdot B_{kj}$

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 $\theta(n)$. We can check this by drawing the appropriate dag.

We can parallelize the inner-for loop using a reduce operations. Using this Sum that employs a reduce operation, we can rewrite the matrix multiplication to achieve smaller span:
let $C$ be a new $n \times n$ matrix

```plaintext
parallel_for i ← 1 to n
do parallel_for j ← 1 to n
do parallel_for k ← 1 to n
do $T[k] ← A_{ik} \cdot B_{kj}$
$C_{ij} ← \text{SUM}(T, 1, n)$
```

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 the best we can hope for.

## 1 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:

\[
\begin{align*}
C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\
C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\
C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\
C_{22} &= A_{21}B_{12} + A_{22}B_{22}
\end{align*}
\]

This suggests a straightforward divide and conquer algorithm. You can compute all 8 parts in parallel and then add them:
MM($C, A, B, n$)
1  if $n = 1$
2      then $c_{11} \leftarrow a_{11}b_{11}$ return
3  partition $A$, $B$, and $C$, into 4 submatrices
4  create $T$, a temporary $n \times n$ 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 $i \leftarrow 1$ to $n$
15     do parallel_for $j \leftarrow 1$ to $n$
16        do $c_{ij} \leftarrow c_{ij} + t_{ij}$

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 $2n \times n$ matrices is $\theta(n^2)$ and the span is $\theta(\lg n)$.

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.

Let’s try to reduce the work of matrix multiplication.

**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:

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