II. Matrix Multiplication

Thomas Sauerwald
Introduction

Serial Matrix Multiplication

Reminder: Multithreading

Multithreaded Matrix Multiplication
Matrix Multiplication

Remember: If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then the matrix product $C = A \cdot B$ is defined by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \quad \forall i, j = 1, 2, \ldots, n.$$
Matrix Multiplication

Remember: If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then the matrix product $C = A \cdot B$ is defined by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \quad \forall i, j = 1, 2, \ldots, n.$$ 

**Square-Matrix-Multiply** $(A, B)$

1. $n = A\.rows$
2. let $C$ be a new $n \times n$ matrix
3. for $i = 1$ to $n$
4. for $j = 1$ to $n$
5. \hspace{1em} $c_{ij} = 0$
6. for $k = 1$ to $n$
7. \hspace{1em} $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
8. return $C$
Matrix Multiplication

Remember: If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then the matrix product $C = A \cdot B$ is defined by

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \quad \forall i, j = 1, 2, \ldots, n.$$ 

**SQUARE-MATRIX-MULTIPLY** $(A, B)$

1. $n = A.rows$
2. let $C$ be a new $n \times n$ matrix
3. for $i = 1$ to $n$
   4. for $j = 1$ to $n$
      5. $c_{ij} = 0$
   6. for $k = 1$ to $n$
      7. $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
8. return $C$

**SQUARE-MATRIX-MULTIPLY** $(A, B)$ takes time $\Theta(n^3)$. 

II. Matrix Multiplication

Introduction
Matrix Multiplication

Remember: If \( A = (a_{ij}) \) and \( B = (b_{ij}) \) are square \( n \times n \) matrices, then the matrix product \( C = A \cdot B \) is defined by

\[
c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \quad \forall i, j = 1, 2, \ldots, n.
\]

This definition suggests that \( n \cdot n^2 = n^3 \) arithmetic operations are necessary.

\begin{algorithm}
1 \hspace{2em} \( n = A \text{.rows} \)
2 \hspace{2em} let \( C \) be a new \( n \times n \) matrix
3 \hspace{2em} for \( i = 1 \) to \( n \)
4 \hspace{4em} for \( j = 1 \) to \( n \)
5 \hspace{6em} \( c_{ij} = 0 \)
6 \hspace{4em} for \( k = 1 \) to \( n \)
7 \hspace{6em} \( c_{ij} = c_{ij} + a_{ik} \cdot b_{kj} \)
8 \hspace{2em} return \( C \)
\end{algorithm}

\( \text{SQUARE-MATRIX-MULTIPLY}(A, B) \) takes time \( \Theta(n^3) \).
Outline

Introduction

Serial Matrix Multiplication

Reminder: Multithreading

Multithreaded Matrix Multiplication
Divide & Conquer: First Approach

**Assumption:** \( n \) is always an exact power of 2.
Divide & Conquer: First Approach

**Assumption:** \( n \) is always an exact power of 2.

Divide & Conquer:
Partition \( A, B, \) and \( C \) into four \( n/2 \times n/2 \) matrices:

\[
A = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix},
B = \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix},
C = \begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix}.
\]

Hence the equation \( C = A \cdot B \) becomes:

\[
\begin{pmatrix}
C_{11} & C_{12} \\
C_{21} & C_{22}
\end{pmatrix} = \begin{pmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{pmatrix} \cdot \begin{pmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{pmatrix}.
\]

This corresponds to the four equations:

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

Each equation specifies two multiplications of \( n/2 \times n/2 \) matrices and the addition of their products.
Divide & Conquer: First Approach

**Assumption:** \( n \) is always an exact power of 2.

Divide & Conquer:
Partition \( A, B, \) and \( C \) into four \( n/2 \times n/2 \) matrices:

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.
\]
Assumption: \( n \) is always an exact power of 2.

Divide & Conquer:
Partition \( A, B, \) and \( C \) into four \( n/2 \times n/2 \) matrices:

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.
\]

Hence the equation \( C = A \cdot B \) becomes:
Divide & Conquer: First Approach

Assumption: \( n \) is always an exact power of 2.

Divide & Conquer:
Partition \( A \), \( B \), and \( C \) into four \( n/2 \times n/2 \) matrices:

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.
\]

Hence the equation \( C = A \cdot B \) becomes:

\[
\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}
\]
Divide & Conquer: First Approach

**Assumption:** $n$ is always an exact power of 2.

Divide & Conquer:
Partition $A$, $B$, and $C$ into four $n/2 \times n/2$ matrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}. $$

Hence the equation $C = A \cdot B$ becomes:

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

This corresponds to the four equations:

$$C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}$$
$$C_{12} = A_{11} \cdot B_{12} + A_{12} \cdot B_{22}$$
$$C_{21} = A_{21} \cdot B_{11} + A_{22} \cdot B_{21}$$
$$C_{22} = A_{21} \cdot B_{12} + A_{22} \cdot B_{22}$$
Divide & Conquer: First Approach

**Assumption:** \( n \) is always an exact power of 2.

Divide & Conquer:
Partition \( A, B, \) and \( C \) into four \( n/2 \times n/2 \) matrices:

\[
A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.
\]

Hence the equation \( C = A \cdot B \) becomes:

\[
\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}
\]

This corresponds to the four equations:

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

Each equation specifies two multiplications of \( n/2 \times n/2 \) matrices and the addition of their products.
4.2 Strassen's algorithm for matrix multiplication

SQUARE-MATRIX-MULTIPLY-RECURSIVE \(A; B\)

1. \(n \leq \text{D A: rows}\)

2. let \(C\) be a new \(n \times n\) matrix

3. if \(n = 1\)

4. \(C_{11} = A_{11} \cdot B_{11} + A_{12} \cdot B_{21}\)

5. else partition \(A\), \(B\), and \(C\) as in equations (4.9)

6. \(C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}.A_{11}; B_{11}/\)

7. \(C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}.A_{11}; B_{12}/\)

8. \(C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}.A_{12}; B_{21}/\)

9. \(C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}.A_{21}; B_{12}/\)

10. return \(C\)

This pseudocode glosses over one subtle but important implementation detail. How do we partition the matrices in line 5? If we were to create \(n=2\) new \(n/STX-n/STX\) matrices, we would spend \('.n^2/\) time copying entries. In fact, we can partition the matrices without copying entries. The trick is to use index calculations. We identify a submatrix by a range of row indices and a range of column indices of the original matrix. We end up representing a submatrix a little differently from how we represent the original matrix, which is the subtlety we are glossing over.

The advantage is that, since we can specify submatrices by index calculations, executing line 5 takes only \('.1/\) time (although we shall see that it makes no difference asymptotically to the overall running time whether we copy or partition in place).

Now, we derive a recurrence to characterize the running time of SQUARE-MATRIX-MULTIPLY-RECURSIVE. Let \(T(n)\) be the time to multiply two \(n \times n\) matrices using this procedure. In the base case, when \(n = 1\), we perform just the one scalar multiplication in line 4, and so \(T(1) = '.1/\) : (4.15)

The recursive case occurs when \(n > 1\). As discussed, partitioning the matrices in line 5 takes \('.1/\) time, using index calculations. In lines 6–9, we recursively call SQUARE-MATRIX-MULTIPLY-RECURSIVE at total \(8T(n/2)\) times. Because each recursive call multiplies two \(n/STX-n/STX\) matrices, thereby contributing \(T(n/2)\) to the overall running time, the time taken by all eight recursive calls is \(8T(n/2)\). We also must account for the four matrix additions in lines 6–9. Each of these matrices contains \(n^2/4\) entries, and so each of the four matrix additions takes \('.n^2/\) time.

Since the number of matrix additions is a constant, the total time spent adding matrices is: 

\[
\begin{align*}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{11} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}
\end{align*}
\]
Divide & Conquer: First Approach (Pseudocode)

**SQUARE-MATRIX-MULTIPLY-RECURSIVE** *(A, B)*

1. \( n = A\text{.rows} \)
2. let \( C \) be a new \( n \times n \) matrix
3. if \( n == 1 \)
4. \( c_{11} = a_{11} \cdot b_{11} \)
5. else partition \( A \), \( B \), and \( C \) as in equations (4.9)
6. \( C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21}) \)
7. \( C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22}) \)
8. \( C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21}) \)
9. \( C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22}) \)
10. return \( C \)

\[
\begin{align*}
C_{11} &= a_{11} \cdot b_{11} + a_{12} \cdot b_{21} \\
C_{12} &= a_{11} \cdot b_{12} + a_{12} \cdot b_{22} \\
C_{21} &= a_{21} \cdot b_{11} + a_{22} \cdot b_{21} \\
C_{11} &= a_{21} \cdot b_{12} + a_{22} \cdot b_{22}
\end{align*}
\]
Divide & Conquer: First Approach (Pseudocode)

\[ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A, B) \]
\begin{align*}
1 & \quad n = A.\text{rows} \\
2 & \quad \text{let } C \text{ be a new } n \times n \text{ matrix} \\
3 & \quad \text{if } n == 1 \\
4 & \quad \quad c_{11} = a_{11} \cdot b_{11} \\
5 & \quad \text{else partition } A, B, \text{ and } C \text{ as in equations (4.9)} \\
6 & \quad \quad C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11}) \\
7 & \quad \quad \quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21}) \\
8 & \quad \quad C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12}) \\
9 & \quad \quad \quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22}) \\
10 & \quad \quad C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11}) \\
11 & \quad \quad \quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21}) \\
12 & \quad \quad C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12}) \\
13 & \quad \quad \quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22}) \\
14 & \quad \text{return } C \\
\end{align*}

Line 5: Handle submatrices implicitly through index calculations instead of creating them.

\[
\begin{align*}
C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\
C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\
C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\
C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \\
\end{align*}
\]
**Square-Matrix-Multiply-Recursive** \((A, B)\)

1. \(n = A.\text{rows}\)
2. let \(C\) be a new \(n \times n\) matrix
3. if \(n == 1\)
   4. \(c_{11} = a_{11} \cdot b_{11}\)
5. else partition \(A, B,\) and \(C\) as in equations (4.9)
6. \(C_{11} = \text{Square-Matrix-Multiply-Recursive}(A_{11}, B_{11}) + \text{Square-Matrix-Multiply-Recursive}(A_{12}, B_{21})\)
7. \(C_{12} = \text{Square-Matrix-Multiply-Recursive}(A_{11}, B_{12}) + \text{Square-Matrix-Multiply-Recursive}(A_{12}, B_{22})\)
8. \(C_{21} = \text{Square-Matrix-Multiply-Recursive}(A_{21}, B_{11}) + \text{Square-Matrix-Multiply-Recursive}(A_{22}, B_{21})\)
9. \(C_{22} = \text{Square-Matrix-Multiply-Recursive}(A_{21}, B_{12}) + \text{Square-Matrix-Multiply-Recursive}(A_{22}, B_{22})\)
10. return \(C\)

Let \(T(n)\) be the runtime of this procedure.
Divide & Conquer: First Approach (Pseudocode)

**Square-Matrix-Multiply-Recursive** ($A$, $B$)

```
1   n = A.rows
2   let $C$ be a new $n \times n$ matrix
3   if $n == 1$
4       $c_{11} = a_{11} \cdot b_{11}$
5   else partition $A$, $B$, and $C$ as in equations (4.9)
6       $C_{11} = \text{Square-Matrix-Multiply-Recursive}(A_{11}, B_{11})$
7       $+ \text{Square-Matrix-Multiply-Recursive}(A_{12}, B_{21})$
8       $C_{12} = \text{Square-Matrix-Multiply-Recursive}(A_{11}, B_{12})$
9       $+ \text{Square-Matrix-Multiply-Recursive}(A_{12}, B_{22})$
10      $C_{21} = \text{Square-Matrix-Multiply-Recursive}(A_{21}, B_{11})$
11      $+ \text{Square-Matrix-Multiply-Recursive}(A_{22}, B_{21})$
12      $C_{22} = \text{Square-Matrix-Multiply-Recursive}(A_{21}, B_{12})$
13      $+ \text{Square-Matrix-Multiply-Recursive}(A_{22}, B_{22})$
14   return $C$
```

Let $T(n)$ be the runtime of this procedure. Then:

$$T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) + \Theta(n^2) & \text{if } n > 1.
\end{cases}$$
Divide & Conquer: First Approach (Pseudocode)

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  if n == 1
4     c_{11} = a_{11} \cdot b_{11}
5  else partition A, B, and C as in equations (4.9)
6     C_{11} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11})
7         + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
8     C_{12} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12})
9         + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
10    C_{21} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11})
11         + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
12    C_{22} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12})
13         + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
14  return C
```

Let $T(n)$ be the runtime of this procedure. Then:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T\left(\frac{n}{2}\right) + \Theta(n^2) & \text{if } n > 1.
\end{cases}
\]
**Divide & Conquer: First Approach (Pseudocode)**

**SQUARE-MATRIX-MULTIPLY-RECURSIVE** (*A, B*)

1. \( n = A.\text{rows} \)
2. let \( C \) be a new \( n \times n \) matrix
3. if \( n == 1 \)
   4. \( c_{11} = a_{11} \cdot b_{11} \)
5. else partition \( A, B, \) and \( C \) as in equations (4.9)
6. \( C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21}) \)
7. \( C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22}) \)
8. \( C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21}) \)
9. \( C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22}) \)
10. return \( C \)

Let \( T(n) \) be the runtime of this procedure. Then:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) & \text{if } n > 1.
\end{cases}
\]
Divide & Conquer: First Approach (Pseudocode)

\[ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A, B) \]

1. \( n = A.\text{rows} \)
2. let \( C \) be a new \( n \times n \) matrix
3. if \( n == 1 \)
4. \( c_{11} = a_{11} \cdot b_{11} \)
5. else partition \( A, B, \) and \( C \) as in equations (4.9)
6. \( C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11}) \)
7. \( \quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21}) \)
8. \( C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12}) \)
9. \( \quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22}) \)
10. \( C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11}) \)
11. \( \quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21}) \)
12. \( C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12}) \)
13. \( \quad + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22}) \)
14. return \( C \)

Let \( T(n) \) be the runtime of this procedure. Then:

\[ T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) & \text{if } n > 1.
\end{cases} \]
Divide & Conquer: First Approach (Pseudocode)

\textbf{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A, B)
1 \hspace{1em} n = A\.\text{rows}
2 \hspace{1em} let C be a new \(n \times n\) matrix
3 \hspace{1em} if \(n == 1\)
4 \hspace{1.2em} \(c_{11} = a_{11} \cdot b_{11}\)
5 \hspace{1em} else partition \(A, B,\) and \(C\) as in equations (4.9)
6 \hspace{1.2em} \(C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})\)
7 \hspace{1.4em} + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})
8 \hspace{1.2em} \(C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})\)
9 \hspace{1.4em} + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})
10 \hspace{1.2em} \(C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})\)
11 \hspace{1.4em} + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})
12 \hspace{1.2em} \(C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})\)
13 \hspace{1.4em} + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})
14 \hspace{1em} return \(C\)

Let \(T(n)\) be the runtime of this procedure. Then:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) + \Theta(n^2) & \text{if } n > 1.
\end{cases}
\]
Divide & Conquer: First Approach (Pseudocode)

**SQ** **UA** **RE** **-M** **ATR** **IX** **-M** **ULTI** **P** **LY-R** **ECUR** **SIVE** *(A, B)*

1. \( n = A.rows \)
2. let \( C \) be a new \( n \times n \) matrix
3. if \( n == 1 \)
   4. \( c_{11} = a_{11} \cdot b_{11} \)
4. else partition \( A, B, \) and \( C \) as in equations (4.9)
   5. \( C_{11} = \) **SQ** **UA** **RE** **-M** **ATR** **IX** **-M** **ULTI** **P** **LY-R** **ECUR** **SIVE** *(\( A_{11}, B_{11} \))*
      + **SQ** **UA** **RE** **-M** **ATR** **IX** **-M** **ULTI** **P** **LY-R** **ECUR** **SIVE** *(\( A_{12}, B_{21} \))*
   6. \( C_{12} = \) **SQ** **UA** **RE** **-M** **ATR** **IX** **-M** **ULTI** **P** **LY-R** **ECUR** **SIVE** *(\( A_{11}, B_{12} \))*
      + **SQ** **UA** **RE** **-M** **ATR** **IX** **-M** **ULTI** **P** **LY-R** **ECUR** **SIVE** *(\( A_{12}, B_{22} \))*
   7. \( C_{21} = \) **SQ** **UA** **RE** **-M** **ATR** **IX** **-M** **ULTI** **P** **LY-R** **ECUR** **SIVE** *(\( A_{21}, B_{11} \))*
      + **SQ** **UA** **RE** **-M** **ATR** **IX** **-M** **ULTI** **P** **LY-R** **ECUR** **SIVE** *(\( A_{22}, B_{21} \))*
   8. \( C_{22} = \) **SQ** **UA** **RE** **-M** **ATR** **IX** **-M** **ULTI** **P** **LY-R** **ECUR** **SIVE** *(\( A_{21}, B_{12} \))*
      + **SQ** **UA** **RE** **-M** **ATR** **IX** **-M** **ULTI** **P** **LY-R** **ECUR** **SIVE** *(\( A_{22}, B_{22} \))*
10. return \( C \)

Let \( T(n) \) be the runtime of this procedure. Then:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) + \Theta(n^2) & \text{if } n > 1.
\end{cases}
\]

Solution: \( T(n) = \)
Divide & Conquer: First Approach (Pseudocode)

\texttt{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A, B)
1 \hspace{1em} n = A.rows
2 \hspace{1em} let C be a new \( n \times n \) matrix
3 \hspace{1em} \textbf{if} n == 1
4 \hspace{2em} c_{11} = a_{11} \cdot b_{11}
5 \hspace{1em} \textbf{else} partition A, B, and C as in equations (4.9)
6 \hspace{1em} C_{11} = \texttt{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})
7 \hspace{2em} + \texttt{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})
8 \hspace{1em} C_{12} = \texttt{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})
9 \hspace{2em} + \texttt{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})
10 \hspace{1em} C_{21} = \texttt{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})
11 \hspace{2em} + \texttt{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})
12 \hspace{1em} C_{22} = \texttt{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})
13 \hspace{2em} + \texttt{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})
14 \hspace{1em} \textbf{return} C

Let \( T(n) \) be the runtime of this procedure. Then:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) + \Theta(n^2) & \text{if } n > 1.
\end{cases}
\]

Solution: \( T(n) = \Theta(8^{\log_2 n}) \)
**Divide & Conquer: First Approach (Pseudocode)**

**SQUARE-MATRIX-MULTIPLY-RECURSIVE** \((A, B)\)

1. \(n = A\).rows\)
2. let \(C\) be a new \(n \times n\) matrix
3. if \(n == 1\)
   4. \(c_{11} = a_{11} \cdot b_{11}\)
5. else partition \(A, B,\) and \(C\) as in equations (4.9)
6. \(C_{11} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11}) + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})\)
7. \(C_{12} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12}) + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})\)
8. \(C_{21} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11}) + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})\)
9. \(C_{22} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12}) + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})\)
10. return \(C\)

Let \(T(n)\) be the runtime of this procedure. Then:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) + \Theta(n^2) & \text{if } n > 1.
\end{cases}
\]

**Solution:** \(T(n) = \Theta(8^{\log_2 n}) = \Theta(n^3)\) **No improvement over the naive algorithm!**
Divide & Conquer: First Approach (Pseudocode)

SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1\ n = A.rows
2\ let C be a new n × n matrix
3\ if n == 1
4\ \ \ c_{11} = a_{11} \cdot b_{11}
5\ else partition A, B, and C as in equations (4.9)
6\ \ C_{11} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11})
7\ \ \ \ \ \ \ \ \ \ + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
8\ \ C_{12} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12})
9\ \ \ \ \ \ \ \ \ \ + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
10\ \ C_{21} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11})
11\ \ \ \ \ \ \ \ \ \ + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
12\ \ C_{22} = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12})
13\ \ \ \ \ \ \ \ \ \ + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
14\ return C

Let \( T(n) \) be the runtime of this procedure. Then:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) + \Theta(n^2) & \text{if } n > 1.
\end{cases}
\]

Solution: \( T(n) = \Theta(8^{\log_2 n}) = \Theta(n^3) \)
Divide & Conquer: First Approach (Pseudocode)

**SQ**AR**E**-*MA**TRIX*-**M**ULTIPLY-**R**ECURSIVE*(A, B)*

1. \( n = A\text{.rows} \)
2. let \( C \) be a new \( n \times n \) matrix
3. if \( n == 1 \)
   4. \( c_{11} = a_{11} \cdot b_{11} \)
5. else partition \( A, B, \) and \( C \) as in equations (4.9)
6. \( C_{11} = \text{SQ**A**RE-MAT**R**IX-MUL**T**IPLY-RECURSIVE}(A_{11}, B_{11}) + \text{SQ**A**RE-MAT**R**IX-MUL**T**IPLY-RECURSIVE}(A_{12}, B_{21}) \)
7. \( C_{12} = \text{SQ**A**RE-MAT**R**IX-MUL**T**IPLY-RECURSIVE}(A_{11}, B_{12}) + \text{SQ**A**RE-MAT**R**IX-MUL**T**IPLY-RECURSIVE}(A_{12}, B_{22}) \)
8. \( C_{21} = \text{SQ**A**RE-MAT**R**IX-MUL**T**IPLY-RECURSIVE}(A_{21}, B_{11}) + \text{SQ**A**RE-MAT**R**IX-MUL**T**IPLY-RECURSIVE}(A_{22}, B_{21}) \)
9. \( C_{22} = \text{SQ**A**RE-MAT**R**IX-MUL**T**IPLY-RECURSIVE}(A_{21}, B_{12}) + \text{SQ**A**RE-MAT**R**IX-MUL**T**IPLY-RECURSIVE}(A_{22}, B_{22}) \)
10. return \( C \)

Let \( T(n) \) be the runtime of this procedure. Then:

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \cdot T(n/2) + \Theta(n^2) & \text{if } n > 1.
\end{cases}
\]

Solution: \( T(n) = \Theta(8^\log_2 n) = \Theta(n^3) \) Goal: Reduce the number of multiplications
Divide & Conquer: Second Approach

**Idea:** Make the recursion tree less bushy by performing only 7 recursive multiplications of $n/2 \times n/2$ matrices.
Divide & Conquer: Second Approach

**Idea**: Make the recursion tree less bushy by performing only 7 recursive multiplications of $n/2 \times n/2$ matrices.

---

**Strassen’s Algorithm (1969)**

1. **Partition** each of the matrices into four $n/2 \times n/2$ submatrices.
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in the previous step.
3. Recursively compute 7 matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$
4. Compute $n/2 \times n/2$ submatrices of $C$ by adding and subtracting various combinations of the $P_i$.
Divide & Conquer: Second Approach

**Idea:** Make the recursion tree less bushy by performing only 7 recursive multiplications of $n/2 \times n/2$ matrices.

---

**Strassen’s Algorithm (1969)**

1. **Partition** each of the matrices into four $n/2 \times n/2$ submatrices
2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in the previous step.
3. **Recursively compute** 7 matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$
4. **Compute** $n/2 \times n/2$ submatrices of $C$ by adding and subtracting various combinations of the $P_i$.

Time for steps 1, 2, 4: $\Theta(n^2)$, hence $T(n) = 7 \cdot T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^{\log_7 7})$. 

II. Matrix Multiplication

Serial Matrix Multiplication
Solving the Recursion

\[ T(n) = 7 \cdot T(n/2) + c \cdot n^2 \]

\[ = 7 \cdot (7 \cdot T(n/4) + c \cdot (n/2)^2) + c \cdot n^2 \]

\[ = 7^2 \cdot T(n/4) + 7c \cdot (n/2)^2 + c \cdot n^2 \]

\[ = 7^2 \cdot (7 \cdot T(n/8) + c \cdot (n/4)^2) + 7c \cdot (n/2)^2 + c \cdot n^2 \]

\[ = 7^3 \cdot T(n/8) + 7^2 c \cdot (n/4)^2 + 7c \cdot (n/2)^2 + c \cdot n^2 \]

\[ = \ldots \]

\[ = 7^{\log_2 n} \cdot T(1) + \sum_{i=0}^{\log_2 n-1} 7^i \cdot c \cdot (n/2)^2 \]

\[ = 7^{\log_2 n} \cdot \Theta(1) + \sum_{i=0}^{\log_2 n-1} \left( \frac{7}{4} \right)^i \cdot c \cdot n^2 \]

\[ = 7^{\log_2 n} \cdot \Theta(1) + \Theta\left( \left( \frac{7}{4} \right)^{\log_2 n-1} \cdot n^2 \right) \]

\[ = 7^{\log_2 n} \cdot \Theta(1) + \Theta\left( 7^{(\log_2 n - 1)} \right) = \Theta\left( 2^{\log_2 n} \right) \]

\[ = \Theta(n \log n) \]
### The 10 Submatrices and 7 Products

<table>
<thead>
<tr>
<th>Product</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22})$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22}$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11}$</td>
</tr>
<tr>
<td>$P_4$</td>
<td>$A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11})$</td>
</tr>
<tr>
<td>$P_5$</td>
<td>$S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$</td>
</tr>
<tr>
<td>$P_6$</td>
<td>$S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$</td>
</tr>
<tr>
<td>$P_7$</td>
<td>$S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$</td>
</tr>
</tbody>
</table>
Details of Strassen’s Algorithm

The 10 Submatrices and 7 Products

\[
P_1 = A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22})
\]
\[
P_2 = S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22}
\]
\[
P_3 = S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11}
\]
\[
P_4 = A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11})
\]
\[
P_5 = S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})
\]
\[
P_6 = S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\]
\[
P_7 = S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})
\]

Claim

\[
\begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{21} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
= \begin{pmatrix}
P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\
- & - \\
P_3 + P_4 & P_5 + P_1 - P_3 - P_7
\end{pmatrix}
\]

Other three blocks can be verified similarly.
Details of Strassen’s Algorithm

The 10 Submatrices and 7 Products

\[ P_1 = A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22}) \]
\[ P_2 = S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22} \]
\[ P_3 = S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11} \]
\[ P_4 = A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11}) \]
\[ P_5 = S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \]
\[ P_6 = S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \]
\[ P_7 = S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \]

Claim

\[
\begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{21} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
= \begin{pmatrix}
P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\
P_3 + P_4 & P_5 + P_1 - P_3 - P_7
\end{pmatrix}
\]

Proof:
Details of Strassen’s Algorithm

The 10 Submatrices and 7 Products

\[ P_1 = A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22}) \]
\[ P_2 = S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22} \]
\[ P_3 = S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11} \]
\[ P_4 = A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11}) \]
\[ P_5 = S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \]
\[ P_6 = S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \]
\[ P_7 = S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \]

Claim

\[
\begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{21} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix} = \begin{pmatrix}
P_5 + P_4 - P_2 + P_6 \\
P_3 + P_4 \\
P_5 + P_1 - P_3 - P_7
\end{pmatrix}
\]

Proof:

\[ P_5 + P_4 - P_2 + P_6 = \]

II. Matrix Multiplication
Serial Matrix Multiplication
9
Details of Strassen’s Algorithm

The 10 Submatrices and 7 Products

\[
\begin{align*}
P_1 &= A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22}) \\
P_2 &= S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11}) \\
P_5 &= S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\
P_6 &= S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\
P_7 &= S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})
\end{align*}
\]

Claim

\[
\begin{pmatrix}
A_{11} B_{11} + A_{12} B_{21} & A_{11} B_{12} + A_{12} B_{21} \\
A_{21} B_{11} + A_{22} B_{21} & A_{21} B_{12} + A_{22} B_{22}
\end{pmatrix}
= \begin{pmatrix}
P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\
P_3 + P_4 & P_5 + P_1 - P_3 - P_7
\end{pmatrix}
\]

Proof:

\[
P_5 + P_4 - P_2 + P_6 = A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22} + A_{22} B_{21} - A_{22} B_{11} - A_{11} B_{22} - A_{12} B_{22} + A_{12} B_{21} + A_{12} B_{22} - A_{22} B_{21} - A_{22} B_{22}
\]
Details of Strassen’s Algorithm

The 10 Submatrices and 7 Products

\[ P_1 = A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22}) \]
\[ P_2 = S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22} \]
\[ P_3 = S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11} \]
\[ P_4 = A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11}) \]
\[ P_5 = S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \]
\[ P_6 = S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \]
\[ P_7 = S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \]

Claim

\[
\begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{21} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
= \begin{pmatrix}
P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\
P_3 + P_4 & P_5 + P_1 - P_3 - P_7
\end{pmatrix}
\]

Proof:

\[ P_5 + P_4 - P_2 + P_6 = A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} + A_{22}B_{21} - A_{22}B_{11} - A_{11}B_{22} - A_{12}B_{22} + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22} \]
Details of Strassen’s Algorithm

The 10 Submatrices and 7 Products

\[ P_1 = A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22}) \]
\[ P_2 = S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22} \]
\[ P_3 = S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11} \]
\[ P_4 = A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11}) \]
\[ P_5 = S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \]
\[ P_6 = S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \]
\[ P_7 = S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \]

Claim

\[
\begin{pmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{21} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{pmatrix}
= \begin{pmatrix}
P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\
P_3 + P_4 & P_5 + P_1 - P_3 - P_7
\end{pmatrix}
\]

Proof:

\[
P_5 + P_4 - P_2 + P_6 = A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} + A_{22}B_{21} - A_{22}B_{11}
- A_{11}B_{22} - A_{12}B_{22} + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}
= A_{11}B_{11} + A_{12}B_{21} \leq C_{11}
\]
Details of Strassen’s Algorithm

The 10 Submatrices and 7 Products

\[
P_1 = A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22})
\]
\[
P_2 = S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22}
\]
\[
P_3 = S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11}
\]
\[
P_4 = A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11})
\]
\[
P_5 = S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})
\]
\[
P_6 = S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\]
\[
P_7 = S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})
\]

Claim

\[
\begin{pmatrix}
A_{11} B_{11} + A_{12} B_{21} & A_{11} B_{12} + A_{12} B_{21} \\
A_{21} B_{11} + A_{22} B_{21} & A_{21} B_{12} + A_{22} B_{22}
\end{pmatrix} =
\begin{pmatrix}
P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\
P_3 + P_4 & P_5 + P_1 - P_3 - P_7
\end{pmatrix}
\]

Proof:

\[
P_5 + P_4 - P_2 + P_6 = A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22} + A_{22} B_{21} - A_{22} B_{11}
\]
\[
- A_{11} B_{22} - A_{12} B_{22} + A_{12} B_{21} + A_{12} B_{22} - A_{22} B_{21} - A_{22} B_{22}
\]
\[
= A_{11} B_{11} + A_{12} B_{21}
\]

Other three blocks can be verified similarly.
### Details of Strassen’s Algorithm

#### The 10 Submatrices and 7 Products

\[
P_1 = A_{11} \cdot S_1 = A_{11} \cdot (B_{12} - B_{22})
\]
\[
P_2 = S_2 \cdot B_{22} = (A_{11} + A_{12}) \cdot B_{22}
\]
\[
P_3 = S_3 \cdot B_{11} = (A_{21} + A_{22}) \cdot B_{11}
\]
\[
P_4 = A_{22} \cdot S_4 = A_{22} \cdot (B_{21} - B_{11})
\]
\[
P_5 = S_5 \cdot S_6 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})
\]
\[
P_6 = S_7 \cdot S_8 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})
\]
\[
P_7 = S_9 \cdot S_{10} = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})
\]

#### Claim

\[
P_i = \left( \alpha_{11} \cdot A_{11} + \alpha_{12} \cdot A_{12} + \alpha_{21} \cdot A_{21} + \alpha_{22} \cdot A_{22} \right) 
\cdot 
\left( \beta_{11} \cdot B_{11} + \beta_{12} \cdot B_{12} + \beta_{21} \cdot B_{21} + \beta_{22} \cdot B_{22} \right)
\]

#### Proof:

\[
P_5 + P_4 - P_2 + P_6 = A_{11} B_{11} + A_{11} B_{22} + A_{22} B_{11} + A_{22} B_{22} + A_{22} B_{21} - A_{22} B_{11} - A_{11} B_{22} - A_{12} B_{22} + A_{12} B_{21} - A_{22} B_{21} - A_{22} B_{22}
\]
\[
= A_{11} B_{11} + A_{12} B_{21}
\]

Other three blocks can be verified similarly.
Conjecture: Does a quadratic-time algorithm exist?
Current State-of-the-Art

Conjecture: Does a quadratic-time algorithm exist?

Asymptotic Complexities:
- $O(n^3)$, naive approach
Conjecture: Does a quadratic-time algorithm exist?

Asymptotic Complexities:

- $O(n^3)$, naive approach
- $O(n^{2.808})$, Strassen (1969)
Conjecture: Does a quadratic-time algorithm exist?

Asymptotic Complexities:

- $O(n^3)$, naive approach
- $O(n^{2.808})$, Strassen (1969)
- $O(n^{2.796})$, Pan (1978)
- $O(n^{2.522})$, Schönhage (1981)
- $O(n^{2.517})$, Romani (1982)
- $O(n^{2.496})$, Coppersmith and Winograd (1982)
- $O(n^{2.479})$, Strassen (1986)
- $O(n^{2.376})$, Coppersmith and Winograd (1989)
Current State-of-the-Art

Conjecture: Does a quadratic-time algorithm exist?

Asymptotic Complexities:

- $O(n^3)$, naive approach
- $O(n^{2.808})$, Strassen (1969)
- $O(n^{2.796})$, Pan (1978)
- $O(n^{2.522})$, Schönhage (1981)
- $O(n^{2.517})$, Romani (1982)
- $O(n^{2.496})$, Coppersmith and Winograd (1982)
- $O(n^{2.479})$, Strassen (1986)
- $O(n^{2.376})$, Coppersmith and Winograd (1989)
- $O(n^{2.374})$, Stothers (2010)
- $O(n^{2.3728642})$, V. Williams (2011)
- $O(n^{2.3728639})$, Le Gall (2014)
- ...
Outline

Introduction

Serial Matrix Multiplication

Reminder: Multithreading

Multithreaded Matrix Multiplication
Memory Models

### Distributed Memory

- Each processor has its private memory
- Access to memory of another processor via messages
Memory Models

Distributed Memory

- Each processor has its private memory
- Access to memory of another processor via messages

II. Matrix Multiplication
Reminder: Multithreading
Memory Models

---

**Distributed Memory**

- Each processor has its private memory
- Access to memory of another processor via messages

---

**Shared Memory**

- Central location of memory
- Each processor has direct access
Memory Models

- **Distributed Memory**
  - Each processor has its private memory
  - Access to memory of another processor via messages

- **Shared Memory**
  - Central location of memory
  - Each processor has direct access
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
- Use concurrency platform which coordinates all resources

Only logical parallelism, but not actual! Need a scheduler to map threads to processors.
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
- Use *concurrency platform* which coordinates all resources

Scheduling jobs, communication protocols, load balancing etc.
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
- Use concurrency platform which coordinates all resources

Functionalities:

- `spawn` (optional) prefix to a procedure call statement, procedure is executed in a separate thread
- `sync` wait until all spawned threads are done
- `parallel` (optional) prefix to the standard loop `for` each iteration is called in its own thread

Only logical parallelism, but not actual! Need a scheduler to map threads to processors.
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
- Use concurrency platform which coordinates all resources

Functionalities:

- spawn

II. Matrix Multiplication Reminder: Multithreading
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
- Use concurrency platform which coordinates all resources

Functionalities:
- `spawn`
  - (optional) prefix to a procedure call statement
  - procedure is executed in a separate thread
- `sync`
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
- Use concurrency platform which coordinates all resources

Functionalities:

- **spawn**
  - (optional) prefix to a procedure call statement
  - procedure is executed in a separate thread

- **sync**
  - wait until all spawned threads are done

- **parallel**
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
- Use concurrency platform which coordinates all resources

Functionalities:

- **spawn**
  - (optional) prefix to a procedure call statement
  - procedure is executed in a separate thread

- **sync**
  - wait until all spawned threads are done

- **parallel**
  - (optional) prefix to the standard loop for
  - each iteration is called in its own thread
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
- Use concurrency platform which coordinates all resources

Functionalities:

- **spawn**
  - (optional) prefix to a procedure call statement
  - procedure is executed in a separate thread

- **sync**
  - wait until all spawned threads are done

- **parallel**
  - (optional) prefix to the standard loop `for`
  - each iteration is called in its own thread

Only logical parallelism, but not actual! Need a scheduler to map threads to processors.
Computing Fibonacci Numbers Recursively (Fig. 27.1)

Figure 27.1 The tree of recursive procedure instances when computing F. Each instance of F with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

```plaintext
0: FIB(n)
1: if n<=1 return n
2: else x=FIB(n-1)
3: y=FIB(n-2)
4: return x+y
```

You would not really want to compute large Fibonacci numbers this way, because this computation does much repeated work. Figure 27.1 shows the tree of recursive procedure instances that are created when computing F6. For example, a call to FIB(6) recursively calls FIB(5) and then FIB(4). But, the call to FIB(5) also results in a call to FIB(4). Both instances of FIB(4) return the same result (F4=F3). Since the FIB procedure does not memoize, the second call to FIB(4) replicates the work that the first call performs.

Let T(n) denote the running time of FIB(n). Since FIB(n) contains two recursive calls plus a constant amount of extra work, we obtain the recurrence

\[ T(n) = T(n-1) + T(n-2) + c \]

This recurrence has solution

\[ T(n) = \phi^n, \text{where } \phi \text{ is the golden ratio} \]

where we show using the substitution method. For an inductive hypothesis, assume that T(n) is \(aF_n + b\), where \(a>1\) and \(b>0\) are constants. Substituting, we obtain

```
0: FIB(n)
1: if n<=1 return n
2: else x=FIB(n-1)
3: y=FIB(n-2)
4: return x+y
```
Computing Fibonacci Numbers Recursively (Fig. 27.1)

**Figure 27.1** The tree of recursive procedure instances when computing $F_{IB}$. Each instance of $F_{IB}$ with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

FIB(n)  
1: if $n \leq 1$ return $n$  
2: else $x = FIB(n-1)$  
3: $y = FIB(n-2)$  
4: return $x+y$
Computing Fibonacci Numbers Recursively (Fig. 27.1)

```java
0: FIB(n)
1: if n<=1 return n
2: else x=FIB(n-1)
3: y=FIB(n-2)
4: return x+y
```

Figure 27.1
The tree of recursive procedure instances when computing FIB(n). Each instance of FIB with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

You would not really want to compute large Fibonacci numbers this way, because this computation does much repeated work. Figure 27.1 shows the tree of recursive procedure instances that are created when computing FIB(n). For example, a call to FIB(6) recursively calls FIB(5) and then FIB(4). But, the call to FIB(5) also results in a call to FIB(4). Both instances of FIB(4) return the same result (F4=F3). Since the FIB procedure does not memoize, the second call to FIB(4) replicates the work that the first call performs.

Let T(n) denote the running time of FIB(n). Since FIB(n) contains two recursive calls plus a constant amount of extra work, we obtain the recurrence T(n) = T(n-1) + T(n-2) + c. This recurrence has solution T(n) = \phi^n - (-\phi)^{-n} - \frac{c}{2}, where \phi is the golden ratio. We can use the substitution method to verify this solution. For an inductive hypothesis, assume that T(n) = \phi^n - (-\phi)^{-n} - \frac{c}{2}, where \phi > 1 and c > 0 are constants. Substituting, we obtain:

```
T(n) = T(n-1) + T(n-2) + c
    = \phi^{n-1} - (-\phi)^{-(n-1)} - \frac{c}{2} + \phi^{n-2} - (-\phi)^{-(n-2)} - \frac{c}{2} + c
    = \phi^n - (-\phi)^{-n} - \frac{c}{2}
```

Very inefficient – exponential time!
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \( P\text{-FIB}(n) \)
1: \( \text{if } n \leq 1 \text{ return } n \)
2: \( \text{else } x = \text{spawn } P\text{-FIB}(n-1) \)
3: \( y = P\text{-FIB}(n-2) \)
4: \( \text{sync} \)
5: \( \text{return } x+y \)

\[ G = (V, E) \]

Total work \( \approx 17 \) nodes, longest path: 8 nodes

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

• Without `spawn` and `sync` same pseudocode as before
• `spawn` does not imply parallel execution (depends on scheduler)

0: `P-FIB(n)`
1: if `n<=1` return `n`
2: else `x=spawn P-FIB(n-1)`
3: `y=P-FIB(n-2)`
4: `sync`
5: return `x+y`
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

Computation Dag $G = (V, E)$

0: \texttt{P-FIB}(n)
1: \hspace{1em} \text{if } n \leq 1 \text{ return } n
2: \hspace{1em} \text{else } x = \texttt{spawn} \texttt{P-FIB}(n-1)
3: \hspace{1em} y = \texttt{P-FIB}(n-2)
4: \hspace{1em} \texttt{sync}
5: \hspace{1em} \texttt{return } x+y

• Without \texttt{spawn} and \texttt{sync} the same pseudocode as before
• \texttt{spawn} does not imply parallel execution (depends on scheduler)
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

Computation Dag $G = (V, E)$
- $V$ set of threads (instructions/strands without parallel control)

```plaintext
0: P-FIB(n)
1: if n<=1 return n
2: else x=spawn P-FIB(n-1)
3: y=P-FIB(n-2)
4: sync
5: return x+y
```
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

Computation Dag \( G = (V, E) \)
- \( V \) set of threads (instructions/strands \textit{without parallel control})
- \( E \) set of dependencies

0: \texttt{P-FIB}(n)  
1: \quad \textbf{if} \ n \leq 1 \ \textbf{return} \ n  
2: \quad \textbf{else} \ x = \texttt{spawn} \ \texttt{P-FIB}(n-1)  
3: \quad y = \texttt{P-FIB}(n-2)  
4: \quad \texttt{sync}  
5: \quad \textbf{return} \ x + y
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

Computation Dag $G = (V, E)$
- $V$ set of threads (instructions/strands without parallel control)
- $E$ set of dependencies

0: P–FIB(n)
1: if n<=1 return n
2: else x=spawn P–FIB(n–1)
3: y=P–FIB(n–2)
4: sync
5: return x+y
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: P-FIB(n)
1: if n<=1 return n
2: else x=spawn P-FIB(n-1)
3: y=P-FIB(n-2)
4: sync
5: return x+y

• Without spawn and sync same pseudocode as before
• spawn does not imply parallel execution (depends on scheduler)

Computation Dag $G = (V, E)$

• $V$ set of threads (instructions/strands without parallel control)
• $E$ set of dependencies

Total work $\approx 17$ nodes, longest path: 8 nodes
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: P-FIB(n)
1: if n<=1 return n
2: else x=spawn P-FIB(n-1)
3: y=P-FIB(n-2)
4: sync
5: return x+y
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB}(n)
1: if \(n \leq 1\) return \(n\)
2: else \(x = \text{spawn} \ P-FIB(n-1)\)
3: \(y = P-FIB(n-2)\)
4: sync
5: return \(x+y\)

\[\text{Computation Dag} \quad G = (V, E)\]
- \(V\): set of threads (instructions/strands without parallel control)
- \(E\): set of dependencies

Total work \(\approx 17\) nodes, longest path: 8 nodes

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

\[ \text{P-FIB}(n) \]

0: \text{P-FIB}(n)

1: \text{if } n \leq 1 \text{ return } n

2: \text{else } x = \text{spawn P-FIB}(n-1)

3: \quad y = \text{P-FIB}(n-2)

4: \quad \text{sync}

5: \quad \text{return } x + y

• Without \text{spawn} and \text{sync} same pseudocode as before
• \text{spawn} does not imply parallel execution (depends on scheduler)

Computation Dag

G = (V, E)

• V set of threads (instructions/strands without parallel control)
• E set of dependencies

Total work \approx 17 nodes, longest path: 8 nodes

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: P-FIB(n)
1: if n<=1 return n
2: else x=spawn P-FIB(n-1)
3: y=P-FIB(n-2)
4: sync
5: return x+y

• Without spawn and sync same pseudocode as before
• spawn does not imply parallel execution (depends on scheduler)

Computation Dag $G = (V, E)$
- $V$ set of threads (instructions/strands without parallel control)
- $E$ set of dependencies

Total work $\approx 17$ nodes, longest path: 8 nodes
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB}(n)
1: \texttt{if } n<=1 \texttt{ return } n
2: \texttt{else } x=\texttt{spawn } P-FIB(n-1)
3: \texttt{y=P-FIB(n-2)}
4: \texttt{sync}
5: \texttt{return } x+y

- \texttt{spawn} does not imply parallel execution (depends on scheduler)
- Without \texttt{spawn} and \texttt{sync} same pseudocode as before

**Computation Dag**

\[ G = (V, E) \]

- \( V \): set of threads (instructions/strands without parallel control)
- \( E \): set of dependencies

**Total work** \( \approx 17 \) nodes, longest path: 8 nodes
0: \texttt{P-FIB(n)}
1: \texttt{if n<=1 return n}
2: \texttt{else x=spawn P-FIB(n-1)}
3: \texttt{y=P-FIB(n-2)}
4: \texttt{sync}
5: \texttt{return x+y}
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \( \text{P-FIB}(n) \)
1: \( \text{if } n \leq 1 \text{ return } n \)
2: \( \text{else } x = \text{spawn P-FIB}(n-1) \)
3: \( y = \text{P-FIB}(n-2) \)
4: \( \text{sync} \)
5: \( \text{return } x + y \)
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB}(n)
1: \textbf{if} \ n <= 1 \ \textbf{return} \ n
2: \textbf{else} \ \texttt{x=spawn} \ \texttt{P-FIB}(n-1)
3: \ \ \ \texttt{y=}\texttt{P-FIB}(n-2)
4: \ \ \ \texttt{sync}
5: \ \ \ \texttt{return} \ x+y
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB}(n)
1: \textbf{if} \ n \leq 1 \ \textbf{return} \ n
2: \textbf{else} \ x = \texttt{spawn} \ \texttt{P-FIB}(n-1)
3: \quad y = \texttt{P-FIB}(n-2)
4: \quad \texttt{sync}
5: \quad \texttt{return} \ x + y

• Without \texttt{spawn} and \texttt{sync} same pseudocode as before
• \texttt{spawn} does not imply parallel execution (depends on scheduler)

Computation Dag $G = (V, E)$

- $V$ set of threads (instructions/strands without parallel control)
- $E$ set of dependencies

Total work $\approx 17$ nodes, longest path: 8 nodes
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB}(n)
1: \texttt{if} \ n \leq 1 \ \texttt{return} \ n
2: \texttt{else} \ \texttt{x=spawn} \ \texttt{P-FIB}(n-1)
3: \ \texttt{y=P-FIB}(n-2)
4: \ \texttt{sync}
5: \ \texttt{return} \ x+y

Computation Dag \( G = (V, E) \)

- \( V \): set of threads (instructions/strands without parallel control)
- \( E \): set of dependencies

Total work \( \approx 17 \) nodes, longest path: 8 nodes

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \textbf{P-FIB}(n)
1: \textbf{if} \ n\leq1 \ \textbf{return} \ n
2: \textbf{else} \ x=\textbf{spawn} \ \textbf{P-FIB}(n-1)
3: \hspace{1em} y=\textbf{P-FIB}(n-2)
4: \hspace{2em} \textbf{sync}
5: \hspace{1em} \textbf{return} \ x+y
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB(n)}
1: \textbf{if} \ n \leq 1 \ \textbf{return} \ n
2: \textbf{else} \ x=\texttt{spawn P-FIB(n-1)}
3: \quad y=\texttt{P-FIB(n-2)}
4: \quad \texttt{sync}
5: \quad \textbf{return} \ x+y
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB}(n)
1: \textbf{if } n \leq 1 \textbf{ return } n
2: \textbf{else } \texttt{x=spawn P-FIB}(n-1)
3: \hspace{1em} \texttt{y=P-FIB}(n-2)
4: \hspace{1em} \texttt{sync}
5: \hspace{1em} \texttt{return } x+y

• Without \texttt{spawn} and \texttt{sync} same pseudocode as before
• \texttt{spawn} does not imply parallel execution (depends on scheduler)

Computation Dag
\[ G = (V, E) \]

- \( V \) set of threads (instructions/strands without parallel control)
- \( E \) set of dependencies

Total work \( \approx 17 \) nodes, longest path: 8 nodes
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

\[
\begin{align*}
0: & \quad \text{P-FIB}(n) \\
1: & \quad \text{if } n \leq 1 \text{ return } n \\
2: & \quad \text{else } x = \text{spawn P-FIB}(n-1) \\
3: & \quad y = \text{P-FIB}(n-2) \\
4: & \quad \text{sync} \\
5: & \quad \text{return } x+y
\end{align*}
\]
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \text{P-FIB}(n)
1: \text{if } n \leq 1 \text{ return } n
2: \text{else } x = \text{spawn P-FIB}(n-1)
3: \quad y = \text{P-FIB}(n-2)
4: \quad \text{sync}
5: \quad \text{return } x+y
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB}(n)
1: \textbf{if} \ n \leq 1 \ \textbf{return} \ n
2: \textbf{else} \ x=\texttt{spawn} \ \texttt{P-FIB}(n-1)
3: \ y=\texttt{P-FIB}(n-2)
4: \ \texttt{sync}
5: \ \texttt{return} \ x+y

---

\*\*\* Matrix Multiplication Reminder: Multithreading \*\*\*
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

0: \texttt{P-FIB(n)}
1: \texttt{if n<=1 return n}
2: \texttt{else x=spawn P-FIB(n-1)}
3: \texttt{y=P-FIB(n-2)}
4: \texttt{sync}
5: \texttt{return x+y}

Total work \(\approx 17\) nodes, longest path: 8 nodes
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication
Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)
II. Matrix Multiplication

Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication

Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication
Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication

Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication

Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication

Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication Reminder: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)

II. Matrix Multiplication
Reminder: Multithreading
Performance Measures

**Work**

**Total time to execute everything on single processor.**
Performance Measures

Work

Total time to execute everything on single processor.
Performance Measures

Work

Total time to execute everything on single processor.

\[ \sum = 30 \]
Performance Measures

**Work**
- Total time to execute everything on single processor.

**Span**
- Longest time to execute the threads along any path.
Performance Measures

**Work**
Total time to execute everything on single processor.

**Span**
Longest time to execute the threads along any path.

II. Matrix Multiplication Reminder: Multithreading
Performance Measures

- **Work**: Total time to execute everything on a single processor.
- **Span**: Longest time to execute the threads along any path.

\[
\sum = 18
\]
Performance Measures

**Work**

*Total time to execute everything on single processor.*

**Span**

*Longest time to execute the threads along any path.*
Performance Measures

Work

Total time to execute everything on single processor.

Span

Longest time to execute the threads along any path.

If each thread takes unit time, span is the length of the critical path.

II. Matrix Multiplication
Reminder: Multithreading
Performance Measures

**Work**
Total time to execute everything on single processor.

**Span**
Longest time to execute the threads along any path.

If each thread takes unit time, span is the length of the critical path.
Performance Measures

**Work**

Total time to execute everything on single processor.

**Span**

Longest time to execute the threads along any path.

If each thread takes unit time, span is the length of the critical path.

#nodes = 5
Work Law and Span Law

\[ T_1 = \text{work}, \quad T_\infty = \text{span} \]

\[ P = \text{number of (identical) processors} \]

\[ T_P = \text{running time on } P \text{ processors} \]

Running time actually also depends on scheduler etc.!

\[ T_P \geq T_1 \]

\[ T_P \geq T_\infty \]

\[ \text{Speed-Up: } T_1 \quad T_P \]

\[ \text{Parallelism: } T_1 \quad T_\infty \]

Maximum Speed-Up bounded by \( P \)!

Maximum Speed-Up for \( \infty \) processors!

II. Matrix Multiplication Reminder: Multithreading
Work Law and Span Law

- $T_1 = \text{work}$, $T_\infty = \text{span}$
Work Law and Span Law

- $T_1 =$ work, $T_{\infty} =$ span
- $P =$ number of (identical) processors
- $T_P =$ running time on $P$ processors
Work Law and Span Law

- $T_1 = \text{work}$, $T_\infty = \text{span}$
- $P = \text{number of (identical) processors}$
- $T_P = \text{running time on } P \text{ processors}$

Running time actually also depends on scheduler etc.!
Work Law and Span Law

- $T_1 =$ work, $T_{\infty} =$ span
- $P =$ number of (identical) processors
- $T_P =$ running time on $P$ processors

**Work Law**

\[ T_P \geq \frac{T_1}{P} \]
Work Law and Span Law

- $T_1 =$ work, $T_\infty =$ span
- $P =$ number of (identical) processors
- $T_P =$ running time on $P$ processors

$T_P \geq \frac{T_1}{P}$

Time on $P$ processors can't be shorter than if all work all time

$T_1 = 8, P = 2$

II. Matrix Multiplication Reminder: Multithreading
Work Law and Span Law

- $T_1 = \text{work}$, $T_\infty = \text{span}$
- $P = \text{number of (identical) processors}$
- $T_P = \text{running time on } P \text{ processors}$

Time on $P$ processors can’t be shorter than if all work all time

$$T_P \geq \frac{T_1}{P}$$

$T_1 = 8$, $P = 2$

II. Matrix Multiplication
Reminder: Multithreading
Work Law and Span Law

- $T_1 = \text{work}, \ T_\infty = \text{span}$
- $P = \text{number of (identical) processors}$
- $T_P = \text{running time on } P \text{ processors}$

$T_P \geq \frac{T_1}{P}$

Time on $P$ processors can't be shorter than if all work all time

$T_1 = 8, \ P = 2$
Work Law and Span Law

- $T_1 =$ work, $T_\infty =$ span
- $P =$ number of (identical) processors
- $T_P =$ running time on $P$ processors

- $T_P \geq \frac{T_1}{P}$

Time on $P$ processors can’t be shorter than if all work all time
Work Law and Span Law

- $T_1 = \text{work}$, $T_\infty = \text{span}$
- $P = \text{number of (identical) processors}$
- $T_P = \text{running time on } P \text{ processors}$

**Work Law**

$$T_P \geq \frac{T_1}{P}$$

**Span Law**

$$T_P \geq T_\infty$$
Work Law and Span Law

- $T_1 = \text{work}, \ T_\infty = \text{span}$
- $P = \text{number of (identical) processors}$
- $T_P = \text{running time on } P \text{ processors}$

**Work Law**

$$T_P \geq \frac{T_1}{P}$$

**Span Law**

$$T_P \geq T_\infty$$

Time on $P$ processors can’t be shorter than time on $\infty$ processors
Work Law and Span Law

- $T_1 = \text{work, } T_\infty = \text{span}$
- $P = \text{number of (identical) processors}$
- $T_P = \text{running time on } P \text{ processors}$

**Work Law**

$$T_P \geq \frac{T_1}{P}$$

**Span Law**

$$T_P \geq T_\infty$$

- Speed-Up: $\frac{T_1}{T_P}$

$T_\infty = 5$
Work Law and Span Law

- \( T_1 = \text{work}, \ T_\infty = \text{span} \)
- \( P = \text{number of (identical) processors} \)
- \( T_P = \text{running time on} \ P \ \text{processors} \)

**Work Law**

\[ T_P \geq \frac{T_1}{P} \]

**Span Law**

\[ T_P \geq T_\infty \]

- Speed-Up: \( \frac{T_1}{T_P} \)

Maximum Speed-Up bounded by \( P \)!
Work Law and Span Law

- $T_1 = \text{work, } T_\infty = \text{span}$
- $P = \text{number of (identical) processors}$
- $T_P = \text{running time on } P \text{ processors}$

\[ T_P \geq \frac{T_1}{P} \]

\[ T_P \geq T_\infty \]

- Speed-Up: $\frac{T_1}{T_P}$
- Parallelism: $\frac{T_1}{T_\infty}$
Work Law and Span Law

- $T_1 = \text{work}, \ T_\infty = \text{span}
- P = \text{number of (identical) processors}
- T_P = \text{running time on } P \text{ processors}

**Work Law**

$T_P \geq \frac{T_1}{P}$

**Span Law**

$T_P \geq T_\infty$

- Speed-Up: $\frac{T_1}{T_P}$
- Parallelism: $\frac{T_1}{T_\infty}$

Maximum Speed-Up for $\infty$ processors!
Outline

Introduction

Serial Matrix Multiplication

Reminder: Multithreading

Multithreaded Matrix Multiplication
Warmup: Matrix Vector Multiplication

Remember: Multiplying an $n \times n$ matrix $A = (a_{ij})$ and $n$-vector $x = (x_j)$ yields an $n$-vector $y = (y_i)$ given by

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \quad \text{for } i = 1, 2, \ldots, n.$$
Warmup: Matrix Vector Multiplication

Remember: Multiplying an $n \times n$ matrix $A = (a_{ij})$ and $n$-vector $x = (x_j)$ yields an $n$-vector $y = (y_i)$ given by

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \quad \text{for } i = 1, 2, \ldots, n.$$
Warmup: Matrix Vector Multiplication

Remember: Multiplying an $n \times n$ matrix $A = (a_{ij})$ and $n$-vector $x = (x_j)$ yields an $n$-vector $y = (y_i)$ given by

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \quad \text{for } i = 1, 2, \ldots, n.$$
Warmup: Matrix Vector Multiplication

Remember: Multiplying an $n \times n$ matrix $A = (a_{ij})$ and $n$-vector $x = (x_j)$ yields an $n$-vector $y = (y_i)$ given by

$$y_i = \sum_{j=1}^{n} a_{ij}x_j \quad \text{for } i = 1, 2, \ldots, n.$$  

MAT-VEC($A, x$)

1. $n = A.rows$
2. let $y$ be a new vector of length $n$
3. **parallel for** $i = 1$ to $n$
   4. $y_i = 0$
5. **parallel for** $i = 1$ to $n$
   6. for $j = 1$ to $n$
      7. $y_i = y_i + a_{ij}x_j$
8. return $y$

The **parallel for**-loops can be used since different entries of $y$ can be computed concurrently.
Implementing parallel for based on Divide-and-Conquer

\[
\text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, i')
\]

1. if \(i == i'\)
2. for \(j = 1\) to \(n\)
3. \(y_i = y_i + a_{ij}x_j\)
4. else \(mid = \lfloor (i + i')/2 \rfloor\)
5. spawn \(\text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, mid)\)
6. \(\text{MAT-VEC-MAIN-LOOP}(A, x, y, n, mid + 1, i')\)
7. sync
Implementing parallel for based on Divide-and-Conquer

\[
\text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, i') \\
1 \quad \text{if } i == i' \\
2 \quad \text{for } j = 1 \text{ to } n \\
3 \quad \quad y_i = y_i + a_{ij} x_j \\
4 \quad \text{else } mid = \lfloor (i + i')/2 \rfloor \\
5 \quad \text{spawn } \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, mid) \\
6 \quad \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, mid + 1, i') \\
7 \quad \text{sync}
\]
Implementing parallel for based on Divide-and-Conquer

\[ \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, i') \]
1. if \( i = i' \)
2. for \( j = 1 \) to \( n \)
3. \( y_i = y_i + a_{ij}x_j \)
4. else mid = \( \lceil (i + i')/2 \rceil \)
5. spawn \( \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, \text{mid}) \)
6. \( \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, \text{mid} + 1, i') \)
7. sync

\[ \text{MAT-VEC}(A, x) \]
1. \( n = A.\text{rows} \)
2. let \( y \) be a new vector of length \( n \)
3. parallel for \( i = 1 \) to \( n \)
4. \( y_i = 0 \)
5. parallel for \( i = 1 \) to \( n \)
6. for \( j = 1 \) to \( n \)
7. \( y_i = y_i + a_{ij}x_j \)
8. return \( y \)
Implementing parallel for based on Divide-and-Conquer

\[ T_1(n) = \]

\[ \text{Mat-Vec-main-loop}(A, x, y, n, i, i') \]
1 \textbf{if} \ i == i'
2 \textbf{for} \ j = 1 \textbf{to} \ n
3 \quad y_i = y_i + a_{ij}x_j
4 \textbf{else} \ mid = \lfloor (i + i')/2 \rfloor
5 \textbf{spawn} \ Mat-Vec-main-loop(A, x, y, n, i, mid)
6 \textbf{Mat-Vec-main-loop}(A, x, y, n, mid + 1, i')
7 \textbf{sync}

\[ \text{Mat-Vec}(A, x) \]
1 \quad n = A.rows
2 \quad \textbf{let} \ y \ \textbf{be} \ \textbf{a new vector of length} \ n
3 \quad \textbf{parallel for} \ i = 1 \textbf{ to} \ n
4 \quad \quad y_i = 0
5 \quad \textbf{parallel for} \ i = 1 \textbf{ to} \ n
6 \quad \quad \textbf{for} \ j = 1 \textbf{ to} \ n
7 \quad \quad \quad y_i = y_i + a_{ij}x_j
8 \quad \textbf{return} \ y
Implementing parallel for based on Divide-and-Conquer

\[ T_1(n) = \]

Work is equal to running time of its serialization; overhead of recursive spawning does not change asymptotics.
Implementing parallel for based on Divide-and-Conquer

\[
\text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, i')
\]

1. if \( i == i' \)
2. for \( j = 1 \) to \( n \)
3. \( y_i = y_i + a_{ij}x_j \)
4. else \( \text{mid} = [(i + i')/2] \)
5. spawn \( \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, \text{mid}) \)
6. \( \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, \text{mid} + 1, i') \)
7. sync

\[
T_1(n) = \Theta(n^2)
\]

**Work** is equal to running time of its serialization; overhead of recursive spawning does not change asymptotics.
Implementing parallel for based on Divide-and-Conquer

\[
\text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, i')
\]

1. if \(i == i'\)
2. for \(j = 1\) to \(n\)
3. \(y_i = y_i + a_{ij}x_j\)
4. else \(\text{mid} = \lfloor (i + i')/2 \rfloor\)
5. \(\text{spawn} \text{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, \text{mid})\)
6. \(\text{MAT-VEC-MAIN-LOOP}(A, x, y, n, \text{mid} + 1, i')\)
7. \(\text{sync}\)

\[
T_1(n) = \Theta(n^2)
\]

\[
T_\infty(n) =
\]

**Work** is equal to running time of its serialization; overhead of recursive spawning does not change asymptotics.

\[
\text{MAT-VEC}(A, x)
\]

1. \(n = A\cdot\text{rows}\)
2. let \(y\) be a new vector of length \(n\)
3. \(\text{parallel for } i = 1\) to \(n\)
4. \(y_i = 0\)
5. \(\text{parallel for } i = 1\) to \(n\)
6. for \(j = 1\) to \(n\)
7. \(y_i = y_i + a_{ij}x_j\)
8. return \(y\)
Implementing parallel for based on Divide-and-Conquer

**Algorithm 27.1 The basics of dynamic multithreading**

```
MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')
1   if i == i'
2       for j = 1 to n
3           y_i = y_i + a_ij x_j
4   else mid = [(i + i')/2]
5       spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
6       MAT-VEC-MAIN-LOOP(A, x, y, n, mid + 1, i')
7       sync
```

MAT-VEC(A, x)
1   n = A.rows
2   let y be a new vector of length n
3   parallel for i = 1 to n
4       y_i = 0
5   parallel for i = 1 to n
6       for j = 1 to n
7       y_i = y_i + a_ij x_j
8   return y

\[ T_1(n) = \Theta(n^2) \]
\[ T_\infty(n) = \]
Implementing parallel for based on Divide-and-Conquer

\[
T_1(n) = \Theta(n^2)
\]

Work is equal to running time of its serialization; overhead of recursive spawning does not change asymptotics.

\[
T_\infty(n) = \Theta(\log n) + \max_{1 \leq i \leq n} \text{iter}(n)
\]

Span is the depth of recursive callings plus the maximum span of any of the \( n \) iterations.

---

**II. Matrix Multiplication**

**Multithreaded Matrix Multiplication**

21
Implementing parallel for based on Divide-and-Conquer

\[ T_1(n) = \Theta(n^2) \]

\[ T_\infty(n) = \Theta(\log n) + \max_{1 \leq i \leq n} \text{iter}(n) \]

\[ = \Theta(n). \]

**Work** is equal to running time of its serialization; overhead of recursive spawning does not change asymptotics.

Span is the depth of recursive callings plus the maximum span of any of the \( n \) iterations.
**Naive Algorithm in Parallel**

P-SQUARE-MATRIX-MULTIPLY \((A, B)\)

1. \(n = A\.\text{rows}\)
2. let \(C\) be a new \(n \times n\) matrix
3. parallel for \(i = 1 \) to \(n\)
4. parallel for \(j = 1 \) to \(n\)
5. \(c_{ij} = 0\)
6. for \(k = 1 \) to \(n\)
7. \(c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}\)
8. return \(C\)
Naive Algorithm in Parallel

P-SQUARE-MATRIX-MULTIPLY(A, B)
1 \( n = A.\text{rows} \)
2 let \( C \) be a new \( n \times n \) matrix
3 \textbf{parallel for} \( i = 1 \) to \( n \)
4 \hspace{1em} \textbf{parallel for} \( j = 1 \) to \( n \)
5 \hspace{2em} \( c_{ij} = 0 \)
6 \hspace{2em} \textbf{for} \( k = 1 \) to \( n \)
7 \hspace{3em} \( c_{ij} = c_{ij} + a_{ik} \cdot b_{kj} \)
8 \textbf{return} \( C \)

P-SQUARE-MATRIX-MULTIPLY(A, B) has work \( T_1(n) = \Theta(n^3) \) and span \( T_\infty(n) = \Theta(n) \).

The first two nested for-loops parallelise perfectly.
The Simple Divide & Conquer Approach in Parallel

**P-MATRIX-MULTIPLY-RECURSIVE** (*C, A, B*)

1. \( n = A.\text{rows} \)
2. \( \text{if } n == 1 \)
   3. \( c_{11} = a_{11}b_{11} \)
4. \( \text{else} \) let \( T \) be a new \( n \times n \) matrix
5. \( \text{partition } A, B, C, \text{and } T \) into \( n/2 \times n/2 \) submatrices
   6. \( \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE}(C_{11}, A_{11}, B_{11}) \)
   7. \( \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE}(C_{12}, A_{11}, B_{12}) \)
   8. \( \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE}(C_{21}, A_{21}, B_{11}) \)
   9. \( \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE}(C_{22}, A_{21}, B_{12}) \)
10. \( \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE}(T_{11}, A_{12}, B_{21}) \)
11. \( \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE}(T_{12}, A_{12}, B_{22}) \)
12. \( \text{spawn } \text{P-MATRIX-MULTIPLY-RECURSIVE}(T_{21}, A_{22}, B_{21}) \)
13. \( \text{P-MATRIX-MULTIPLY-RECURSIVE}(T_{22}, A_{22}, B_{22}) \)
14. \( \text{sync} \)
15. \( \text{parallel for } i = 1 \text{ to } n \)
   16. \( \text{parallel for } j = 1 \text{ to } n \)
      17. \( c_{ij} = c_{ij} + t_{ij} \)
The Simple Divide & Conquer Approach in Parallel

P-MATRIX-MULTIPLY-RECURSIVE \( (C, A, B) \)

1. \( n = A \text{.rows} \)
2. if \( n == 1 \)
3. \( c_{11} = a_{11} b_{11} \)
4. else let \( T \) be a new \( n \times n \) matrix
5. partition \( A, B, C, \) and \( T \) into \( n/2 \times n/2 \) submatrices
   \( A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22}; C_{11}, C_{12}, C_{21}, C_{22}; \)
   and \( T_{11}, T_{12}, T_{21}, T_{22}; \) respectively
6. \( \text{spawn P-MATRIX-MULTIPLY-RECURSIVE} (C_{11}, A_{11}, B_{11}) \)
7. \( \text{spawn P-MATRIX-MULTIPLY-RECURSIVE} (C_{12}, A_{11}, B_{12}) \)
8. \( \text{spawn P-MATRIX-MULTIPLY-RECURSIVE} (C_{21}, A_{21}, B_{11}) \)
9. \( \text{spawn P-MATRIX-MULTIPLY-RECURSIVE} (C_{22}, A_{21}, B_{12}) \)
10. \( \text{spawn P-MATRIX-MULTIPLY-RECURSIVE} (T_{11}, A_{12}, B_{21}) \)
11. \( \text{spawn P-MATRIX-MULTIPLY-RECURSIVE} (T_{12}, A_{12}, B_{22}) \)
12. \( \text{spawn P-MATRIX-MULTIPLY-RECURSIVE} (T_{21}, A_{22}, B_{21}) \)
13. \( \text{P-MATRIX-MULTIPLY-RECURSIVE} (T_{22}, A_{22}, B_{22}) \)
14. \( \text{sync} \)
15. parallel for \( i = 1 \) to \( n \)
   16. parallel for \( j = 1 \) to \( n \)
   17. \( c_{ij} = c_{ij} + t_{ij} \)

The same as before.

P-MATRIX-MULTIPLY-RECURSIVE has work \( T_1(n) = \Theta(n^3) \) and span \( T_\infty(n) = \)
The Simple Divide & Conquer Approach in Parallel

**P-MATRIX-MULTIPLY-RECURSIVE**($C, A, B$)

1. $n = A.rows$
2. if $n == 1$
   - $c_{11} = a_{11}b_{11}$
   - $T_\infty(n) = \Theta(1)$
3. else
   - let $T$ be a new $n \times n$ matrix
   - partition $A$, $B$, $C$, and $T$ into $n/2 \times n/2$ submatrices
     - $A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22}; C_{11}, C_{12}, C_{21}, C_{22};$
     - and $T_{11}, T_{12}, T_{21}, T_{22};$ respectively
4. spawn **P-MATRIX-MULTIPLY-RECURSIVE**($C_{11}, A_{11}, B_{11}$)
5. spawn **P-MATRIX-MULTIPLY-RECURSIVE**($C_{12}, A_{11}, B_{12}$)
6. spawn **P-MATRIX-MULTIPLY-RECURSIVE**($C_{21}, A_{21}, B_{11}$)
7. spawn **P-MATRIX-MULTIPLY-RECURSIVE**($C_{22}, A_{21}, B_{12}$)
8. spawn **P-MATRIX-MULTIPLY-RECURSIVE**($T_{11}, A_{12}, B_{21}$)
9. spawn **P-MATRIX-MULTIPLY-RECURSIVE**($T_{12}, A_{12}, B_{22}$)
10. spawn **P-MATRIX-MULTIPLY-RECURSIVE**($T_{21}, A_{22}, B_{21}$)
11. spawn **P-MATRIX-MULTIPLY-RECURSIVE**($T_{22}, A_{22}, B_{22}$)
12. sync
13. parallel for $i = 1$ to $n$
   - parallel for $j = 1$ to $n$
   - $c_{ij} = c_{ij} + t_{ij}$
14. $T_\infty(n) = \Theta(\log n)$

**P-MATRIX-MULTIPLY-RECURSIVE** has work $T_1(n) = \Theta(n^3)$ and span $T_\infty(n) =$

$$T_\infty(n) = T_\infty(n/2) + \Theta(\log n)$$
The Simple Divide & Conquer Approach in Parallel

\[ \text{P-MATRIX-MULTIPLY-RECURSIVE}(C, A, B) \]

1. \( n = A \text{.rows} \)
2. \textbf{if} \( n == 1 \)
3. \( c_{11} = a_{11}b_{11} \)
4. \textbf{else} let \( T \) be a new \( n \times n \) matrix
5. \quad partition \( A, B, C, \) and \( T \) into \( n/2 \times n/2 \) submatrices
   \quad \( A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22}; C_{11}, C_{12}, C_{21}, C_{22}; \)
   \quad and \( T_{11}, T_{12}, T_{21}, T_{22} \); respectively
6. \textbf{spawn} \text{P-MATRIX-MULTIPLY-RECURSIVE}(C_{11}, A_{11}, B_{11})
7. \textbf{spawn} \text{P-MATRIX-MULTIPLY-RECURSIVE}(C_{12}, A_{11}, B_{12})
8. \textbf{spawn} \text{P-MATRIX-MULTIPLY-RECURSIVE}(C_{21}, A_{21}, B_{11})
9. \textbf{spawn} \text{P-MATRIX-MULTIPLY-RECURSIVE}(C_{22}, A_{21}, B_{12})
10. \textbf{spawn} \text{P-MATRIX-MULTIPLY-RECURSIVE}(T_{11}, A_{12}, B_{21})
11. \textbf{spawn} \text{P-MATRIX-MULTIPLY-RECURSIVE}(T_{12}, A_{12}, B_{22})
12. \textbf{spawn} \text{P-MATRIX-MULTIPLY-RECURSIVE}(T_{21}, A_{22}, B_{21})
13. \textbf{spawn} \text{P-MATRIX-MULTIPLY-RECURSIVE}(T_{22}, A_{22}, B_{22})
14. \textbf{sync}
15. \textbf{parallel for} \( i = 1 \) to \( n \)
16. \quad \textbf{parallel for} \( j = 1 \) to \( n \)
17. \quad \( c_{ij} = c_{ij} + t_{ij} \)

The same as before.

\[ \text{P-MATRIX-MULTIPLY-RECURSIVE} \text{ has work } T_1(n) = \Theta(n^3) \text{ and span } T_\infty(n) = \Theta(\log^2 n). \]

\[ T_\infty(n) = T_\infty(n/2) + \Theta(\log n) \]
1. Partition each of the matrices into four $n/2 \times n/2$ submatrices
Strassen’s Algorithm (parallelised)

1. Partition each of the matrices into four $\frac{n}{2} \times \frac{n}{2}$ submatrices

This step takes $\Theta(1)$ work and span by index calculations.
Strassen's Algorithm in Parallel

**Strassen's Algorithm (parallelised)**

1. **Partition** each of the matrices into four \( n/2 \times n/2 \) submatrices

   This step takes \( \Theta(1) \) work and span by index calculations.

2. Create 10 matrices \( S_1, S_2, \ldots, S_{10} \). Each is \( n/2 \times n/2 \) and is the sum or difference of two matrices created in the previous step.
Strassen’s Algorithm in Parallel

---

### Strassen’s Algorithm (parallelised)

1. **Partition** each of the matrices into four $n/2 \times n/2$ submatrices

   This step takes $\Theta(1)$ work and span by index calculations.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in the previous step.

   Can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\log n)$ span using doubly nested *parallel for* loops.
Strassen’s Algorithm in Parallel

Strassen’s Algorithm (parallelised)

1. **Partition** each of the matrices into four \( \frac{n}{2} \times \frac{n}{2} \) submatrices

   This step takes \( \Theta(1) \) work and span by index calculations.

2. Create 10 matrices \( S_1, S_2, \ldots, S_{10} \). Each is \( \frac{n}{2} \times \frac{n}{2} \) and is the sum or difference of two matrices created in the previous step.

   Can create all 10 matrices with \( \Theta(n^2) \) work and \( \Theta(\log n) \) span using doubly nested parallel for loops.

3. Recursively compute 7 matrix products \( P_1, P_2, \ldots, P_7 \), each \( \frac{n}{2} \times \frac{n}{2} \)
Strassen’s Algorithm in Parallel

Strassen’s Algorithm (parallelised)

1. **Partition** each of the matrices into four $n/2 \times n/2$ submatrices

   This step takes $\Theta(1)$ work and span by **index calculations**.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the **sum** or **difference** of two matrices created in the previous step.

   Can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\log n)$ span using doubly nested **parallel for** loops.

3. Recursively compute **7 matrix products** $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$

   Recursively **spawn** the computation of the seven products.
Strassen’s Algorithm in Parallel

Strassen’s Algorithm (parallelised)

1. **Partition** each of the matrices into four $n/2 \times n/2$ submatrices

   This step takes $\Theta(1)$ work and span by index calculations.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in the previous step.

   Can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\log n)$ span using doubly nested **parallel for** loops.

3. Recursively compute **7 matrix products** $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$

   Recursively **spawn** the computation of the seven products.

4. Compute $n/2 \times n/2$ submatrices of $C$ by adding and subtracting various combinations of the $P_i$. 

II. Matrix Multiplication  
Multithreaded Matrix Multiplication  
24
Strassen’s Algorithm in Parallel

**Strassen’s Algorithm (parallelised)**

1. **Partition** each of the matrices into four $n/2 \times n/2$ submatrices

   This step takes $\Theta(1)$ work and span by **index calculations**.

2. **Create** 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the **sum or difference** of two matrices created in the previous step.

   Can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\log n)$ span using doubly nested **parallel for** loops.

3. **Recursively compute** 7 matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$

   **Recursively spawn** the computation of the seven products.

4. **Compute** $n/2 \times n/2$ submatrices of $C$ by adding and subtracting various combinations of the $P_i$.

   Using doubly nested **parallel for**
   this takes $\Theta(n^2)$ work and $\Theta(\log n)$ span.
Strassen’s Algorithm (parallelised)

1. **Partition** each of the matrices into four $n/2 \times n/2$ submatrices

   This step takes $\Theta(1)$ work and span by index calculations.

2. Create 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in the previous step.

   Can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\log n)$ span using doubly nested **parallel for** loops.

3. Recursively compute 7 matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$

   Recursively **spawn** the computation of the seven products.

4. Compute $n/2 \times n/2$ submatrices of $C$ by adding and subtracting various combinations of the $P_i$.

   Using doubly nested **parallel for** this takes $\Theta(n^2)$ work and $\Theta(\log n)$ span.

\[
T_1(n) = \Theta(n^{\log 7})
\]
Strassen’s Algorithm in Parallel

Strassen’s Algorithm (parallelised)

1. **Partition** each of the matrices into four $n/2 \times n/2$ submatrices

   This step takes $\Theta(1)$ work and span by index calculations.

2. **Create** 10 matrices $S_1, S_2, \ldots, S_{10}$. Each is $n/2 \times n/2$ and is the sum or difference of two matrices created in the previous step.

   Can create all 10 matrices with $\Theta(n^2)$ work and $\Theta(\log n)$ span using doubly nested **parallel for loops**.

3. **Recursively compute** 7 matrix products $P_1, P_2, \ldots, P_7$, each $n/2 \times n/2$

   Recursively **spawn** the computation of the seven products.

4. **Compute** $n/2 \times n/2$ submatrices of $C$ by adding and subtracting various combinations of the $P_i$.

   Using doubly nested **parallel for** this takes $\Theta(n^2)$ work and $\Theta(\log n)$ span.

$T_1(n) = \Theta(n^{\log 7})$

$T_\infty(n) = \Theta(\log^2 n)$
Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.
Matrix Multiplication and Matrix Inversion

**Theorem 28.1 (Multiplication is no harder than Inversion)**

If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$. 

Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.
Matrix Multiplication and Matrix Inversion

Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.

Theorem 28.1 (Multiplication is no harder than Inversion)

If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

Proof:
Matrix Multiplication and Matrix Inversion

Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.

Theorem 28.1 (Multiplication is no harder than Inversion)

If we can invert an \( n \times n \) matrix in time \( I(n) \), where \( I(n) = \Omega(n^2) \) and \( I(n) \) satisfies the regularity condition \( I(3n) = O(I(n)) \), then we can multiply two \( n \times n \) matrices in time \( O(I(n)) \).

Proof:

- Define a \( 3n \times 3n \) matrix \( D \) by:

\[
D = \begin{pmatrix}
I_n & A & 0 \\
0 & I_n & B \\
0 & 0 & I_n
\end{pmatrix}
\]
Matrix Multiplication and Matrix Inversion

Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.

Theorem 28.1 (Multiplication is no harder than Inversion)

If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

Proof:

- Define a $3n \times 3n$ matrix $D$ by:

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix} \quad \Rightarrow \quad D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}.$$
Matrix Multiplication and Matrix Inversion

Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.

Theorem 28.1 (Multiplication is no harder than Inversion)

If we can invert an \( n \times n \) matrix in time \( I(n) \), where \( I(n) = \Omega(n^2) \) and \( I(n) \) satisfies the regularity condition \( I(3n) = O(I(n)) \), then we can multiply two \( n \times n \) matrices in time \( O(I(n)) \).

Proof:

- Define a \( 3n \times 3n \) matrix \( D \) by:

\[
D = \begin{pmatrix}
I_n & A & 0 \\
0 & I_n & B \\
0 & 0 & I_n
\end{pmatrix}
\]

\[\Rightarrow\]

\[
D^{-1} = \begin{pmatrix}
I_n & -A & AB \\
0 & I_n & -B \\
0 & 0 & I_n
\end{pmatrix}.
\]
Matrix Multiplication and Matrix Inversion

Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.

Theorem 28.1 (Multiplication is no harder than Inversion)

If we can invert an \( n \times n \) matrix in time \( I(n) \), where \( I(n) = \Omega(n^2) \) and \( I(n) \) satisfies the regularity condition \( I(3n) = O(I(n)) \), then we can multiply two \( n \times n \) matrices in time \( O(I(n)) \).

Proof:

- Define a \( 3n \times 3n \) matrix \( D \) by:
  \[
  D = \begin{pmatrix}
  I_n & A & 0 \\
  0 & I_n & B \\
  0 & 0 & I_n
  \end{pmatrix}
  \Rightarrow
  D^{-1} = \begin{pmatrix}
  I_n & -A & AB \\
  0 & I_n & -B \\
  0 & 0 & I_n
  \end{pmatrix}.
  \]

- Matrix \( D \) can be constructed in \( \Theta(n^2) = O(I(n)) \) time,
Matrix Multiplication and Matrix Inversion

Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.

**Theorem 28.1 (Multiplication is no harder than Inversion)**

If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

**Proof:**

- Define a $3n \times 3n$ matrix $D$ by:

\[
D = \begin{pmatrix}
I_n & A & 0 \\
0 & I_n & B \\
0 & 0 & I_n
\end{pmatrix}
\Rightarrow D^{-1} = \begin{pmatrix}
I_n & -A & AB \\
0 & I_n & -B \\
0 & 0 & I_n
\end{pmatrix}.
\]

- Matrix $D$ can be constructed in $\Theta(n^2) = O(I(n))$ time,
- and we can invert $D$ in $O(I(3n)) = O(I(n))$ time.
Matrix Multiplication and Matrix Inversion

Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.

Theorem 28.1 (Multiplication is no harder than Inversion)

If we can invert an \( n \times n \) matrix in time \( I(n) \), where \( I(n) = \Omega(n^2) \) and \( I(n) \) satisfies the regularity condition \( I(3n) = O(I(n)) \), then we can multiply two \( n \times n \) matrices in time \( O(I(n)) \).

Proof:

- Define a \( 3n \times 3n \) matrix \( D \) by:

\[
D = \begin{pmatrix}
I_n & A & 0 \\
0 & I_n & B \\
0 & 0 & I_n
\end{pmatrix}
\Rightarrow
D^{-1} = \begin{pmatrix}
I_n & -A & AB \\
0 & I_n & -B \\
0 & 0 & I_n
\end{pmatrix}.
\]

- Matrix \( D \) can be constructed in \( \Theta(n^2) = O(I(n)) \) time,
- and we can invert \( D \) in \( O(I(3n)) = O(I(n)) \) time.

\[\Rightarrow\] We can compute \( AB \) in \( O(I(n)) \) time.
The Other Direction

Theorem 28.1 (Multiplication is no harder than Inversion)
If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

Theorem 28.2 (Inversion is no harder than Multiplication)
Suppose we can multiply two $n \times n$ real matrices in time $M(n)$ and $M(n)$ satisfies the two regularity conditions $M(n + k) = O(M(n))$ for any $0 \leq k \leq n$ and $M(n/2) \leq c \cdot M(n)$ for some constant $c < 1/2$. Then we can compute the inverse of any real nonsingular $n \times n$ matrix in time $O(M(n))$. 

II. Matrix Multiplication
Multithreaded Matrix Multiplication
26
The Other Direction

Theorem 28.1 (Multiplication is no harder than Inversion)
If we can invert an $n \times n$ matrix in time $I(n)$, where $I(n) = \Omega(n^2)$ and $I(n)$ satisfies the regularity condition $I(3n) = O(I(n))$, then we can multiply two $n \times n$ matrices in time $O(I(n))$.

Theorem 28.2 (Inversion is no harder than Multiplication)
Suppose we can multiply two $n \times n$ real matrices in time $M(n)$ and $M(n)$ satisfies the two regularity conditions $M(n + k) = O(M(n))$ for any $0 \leq k \leq n$ and $M(n/2) \leq c \cdot M(n)$ for some constant $c < 1/2$. Then we can compute the inverse of any real nonsingular $n \times n$ matrix in time $O(M(n))$.

Proof of this direction much harder (CLRS) – relies on properties of SPD matrices.
The Other Direction

Theorem 28.1 (Multiplication is no harder than Inversion)

If we can invert an \( n \times n \) matrix in time \( I(n) \), where \( I(n) = \Omega(n^2) \) and \( I(n) \) satisfies the regularity condition \( I(3n) = O(I(n)) \), then we can multiply two \( n \times n \) matrices in time \( O(I(n)) \).

Allows us to use Strassen’s Algorithm to invert a matrix!

Theorem 28.2 (Inversion is no harder than Multiplication)

Suppose we can multiply two \( n \times n \) real matrices in time \( M(n) \) and \( M(n) \) satisfies the two regularity conditions \( M(n + k) = O(M(n)) \) for any \( 0 \leq k \leq n \) and \( M(n/2) \leq c \cdot M(n) \) for some constant \( c < 1/2 \). Then we can compute the inverse of any real nonsingular \( n \times n \) matrix in time \( O(M(n)) \).

Proof of this direction much harder (CLRS) – relies on properties of SPD matrices.