II. Matrix Multiplication

Thomas Sauerwald
Introduction

Serial Matrix Multiplication

Digression: Multithreading

Multithreaded 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. \hspace{1em} **for** $j = 1$ **to** $n$
5. \hspace{2em} $c_{ij} = 0$
6. \hspace{2em} **for** $k = 1$ **to** $n$
7. \hspace{3em} $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
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

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \quad \forall i, j = 1, 2, \ldots, n.$$
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 \( n^2 \cdot n = n^3 \) arithmetic operations are necessary.

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

Introduction

Serial Matrix Multiplication

Digression: 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:

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

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:

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

II. Matrix Multiplication
Serial Matrix Multiplication 5
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:

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

4.2 Strassen's algorithm for matrix multiplication

\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)

\begin{verbatim}
QUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
1  n = A.rows
2  let C be a new n x 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
\end{verbatim}

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

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

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

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

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

1. \( n = A\.rows \)
2. let \( C \) be a new \( n \times n \) matrix
3. if \( n \equiv 1 \)
   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. \( C_{12} = **SQUARE-MATRIX-MULTIPLY-RECURSIVE**(A_{11}, B_{12}) \)
   8. \( C_{21} = **SQUARE-MATRIX-MULTIPLY-RECURSIVE**(A_{21}, B_{11}) \)
   9. \( C_{22} = **SQUARE-MATRIX-MULTIPLY-RECURSIVE**(A_{21}, B_{12}) \)
10. return \( C \)

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

\[
T(n) = \begin{cases} 
\Theta(1) & \text{if } n = 1, \\
8 \text{ Multiplications} & \text{if } n > 1.
\end{cases}
\]

II. Matrix Multiplication
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} · 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:

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

8 Multiplications

II. Matrix Multiplication

Serial Matrix Multiplication
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. \( C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11}) \)
   7. \( C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12}) \)
   8. \( C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11}) \)
   9. \( C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12}) \)
10. return \( C \)

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

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

8 Multiplications 4 Additions and Partitioning
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. \( C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11}) \)
   \(+ \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21}) \)
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:

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

\textbf{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A, B)

1 \hspace{1em} n = A.\text{rows}
2 \hspace{1em} \text{let } C \text{ be a new } n \times n \text{ matrix}
3 \hspace{1em} \text{if } n == 1
4 \hspace{2em} c_{11} = a_{11} \cdot b_{11}
5 \hspace{1em} \text{else partition } A, B, \text{ and } C \text{ as in equations (4.9)}
6 \hspace{1em} C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11})
7 \hspace{1.5em} + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})
8 \hspace{1em} C_{12} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{12})
9 \hspace{1.5em} + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{22})
10 \hspace{1em} C_{21} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{11})
11 \hspace{1.5em} + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{21})
12 \hspace{1em} C_{22} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{21}, B_{12})
13 \hspace{1.5em} + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{22}, B_{22})
14 \hspace{1em} \text{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. \(C_{11} = \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{11}, B_{11}) + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21})\)
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:

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

Solution: \(T(n) = \Theta(8^{\log_2 n}) = \Theta(n^3)\) No improvement over the naive algorithm!
4 Additions and Partitioning

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

Goal: Reduce the number of multiplications

No improvement over the naive algorithm!

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)

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

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. \( + \text{SQUARE-MATRIX-MULTIPLY-RECURSIVE}(A_{12}, B_{21}) \)
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. \( \text{return } C \)

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 \( \frac{n}{2} \times \frac{n}{2} \) 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 $P_i$. 

**II. Matrix Multiplication**

**Serial Matrix Multiplication**
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 $T(n) = 7 \cdot T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^{\log_2 7})$. 
Solving the Recursion

\[ T(n) = 7 \cdot T(n/2) + c \cdot n^2 \]
Details of Strassen’s Algorithm

The 10 Submatrices and 7 Products

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

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

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

Proof:

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

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

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

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

Proof:

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

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

Proof:

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

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

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

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

Proof:

\[
P_5 + P_4 - P_2 + P_6 = A_{11}B_{11} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + \cancel{A_{22}B_{22}} + A_{22}B_{21} - \cancel{A_{22}B_{11}} - A_{11}B_{22} - \cancel{A_{12}B_{22}} + A_{12}B_{21} + A_{12}B_{22} - \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{22}}
\]
\[
= A_{11}B_{11} + A_{12}B_{21}
\]
Open Problem: Is there an algorithm with quadratic complexity?
Open Problem: Is there an algorithm with quadratic complexity?

Asymptotic Complexities:
- $O(n^3)$, naive approach
Current State-of-the-Art

Open Problem: Is there an algorithm with quadratic complexity?

Asymptotic Complexities:

- $O(n^3)$, naive approach
- $O(n^{2.808})$, Strassen (1969)
Open Problem: Is there an algorithm with quadratic complexity?

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

Open Problem: Is there an algorithm with quadratic complexity?

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)
- ...
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 Digression: 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

II. Matrix Multiplication Digression: Multithreading
Dynamic Multithreading

- Programming shared-memory parallel computer difficult
Dynamic Multithreading

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

Functionalities:
- `spawn` (optional) prefix to a procedure call statement
- `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

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
  
  The procedure is executed in a separate thread.

- **sync**
  
  Wait until all spawned threads are done.

- **parallel** (optional) prefix to the standard loop
  
  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`
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 $F_n$. Each instance of $F_n$ 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 $x + y$

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 $F_6$. For example, a call to $F_6$ recursively calls $F_5$ and then $F_4$. But, the call to $F_5$ also results in a call to $F_4$. Both instances of $F_4$ return the same result ($F_4 = 3$). Since the $FIB$ procedure does not memoize, the second call to $F_4$ 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 $T(n) = T(n-1) + T(n-2) + \Theta(1)$.

This recurrence has solution $T(n) = \Theta(F_n)$, which shows us using the substitution method. For an inductive hypothesis, assume that $T(n) = \Theta(aF_n + b)$, where $a > 1$ and $b > 0$ are constants. Substituting, we obtain

Very inefficient – exponential time!
Computing Fibonacci Numbers Recursively (Fig. 27.1)

Figure 27.1

The tree of recursive procedure instances when computing $FIB(n)$. Each in

stance of $FIB(n)$ with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

$FIB(n) = \begin{cases} 
1, & \text{if } n \leq 1 \\
FIB(n-1) + FIB(n-2), & \text{otherwise}
\end{cases}$

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 $FIB(6)$. 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 ($FIB(4) = 3$). 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 $T(n) = \phi^n - \gamma^n$, where $\phi$ and $\gamma$ are the roots of the characteristic equation. For an inductive hypothesis, assume that $T(n) = a\phi^n + b\gamma^n$, where $a > 1$ and $b > 0$ are constants. Substituting, we obtain

$T(n) = a\phi^{n+1} - a\phi^n + b\gamma^{n+1} - b\gamma^n$.

Very inefficient – exponential time!

0: $FIB(n)$
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)

Figure 27.1 The tree of recursive procedure instances when computing \( F_{n} \).

Each instance of \( F_{n} \) with the same argument does the same work to produce the same result, providing an inefficient but interesting way to compute Fibonacci numbers.

\[
\begin{align*}
F(n) & \text{ if } n \leq 1 \\
& \text{ else } x = FIB(n-1) \\
y = FIB(n-2) \\
& \text{ return } x + y
\end{align*}
\]

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 \( F_{n} \).

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 \((F_4 = 3)\). 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

\[
T(n) = T(n-1) + T(n-2) + \text{constant}.
\]

This recurrence has solution \( T(n) = \Theta(n) \), which shows us using the substitution method. For an inductive hypothesis, assume that \( T(n) = \Theta(n) \), where \( a > 1 \) and \( b > 0 \) are constants. Substituting, we obtain

Very inefficient – exponential time!

0: \text{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 in Parallel (Fig. 27.2)

0: $\text{P-FIB}(n)$
1: if $n \leq 1$ return $n$
2: else $x = \text{spawn} \ P-\text{FIB}(n-1)$
3: $y = \text{P-FIB}(n-2)$
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 $G = (V, E)$

Total work $\approx 17$ nodes, longest path: 8 nodes
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: P-FIB(n)
1: if $n \leq 1$ 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 without parallel control)

0: P-FIB(n)
1:  if $n \leq 1$ 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 *without parallel control*)
- $E$ set of dependencies

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: \text{ y = P-FIB}(n-2)
4: \text{ sync}
5: \text{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

```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)

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

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

- Without \texttt{spawn} and \texttt{sync}, the 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

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

0: \texttt{P-FIB}(n)
1: \textbf{if} \ n\leqslant 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
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 ≈ 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

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

II. Matrix Multiplication

Digression: Multithreading
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

• Without \textit{spawn} and \textit{sync} same pseudocode as before
• \textit{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

• 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

II. Matrix Multiplication Digression: 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: 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
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 ≈ 17 nodes, longest path: 8 nodes
Computing Fibonacci Numbers in Parallel (Fig. 27.2)

\[ P-FIB(n) \]

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

\[ G = (V, E) \]
\begin{itemize}
\item \( V \) set of threads (instructions/strands without parallel control)
\item \( E \) set of dependencies
\end{itemize}

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

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

II. Matrix Multiplication

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

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

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

II. Matrix Multiplication

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

II. Matrix Multiplication

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

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

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

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

II. Matrix Multiplication

Digression: Multithreading
Computing Fibonacci Numbers in Parallel (DAG Perspective)
Performance Measures

Work

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

Work

Total time to execute everything on a single processor.

II. Matrix Multiplication
Digression: Multithreading
Performance Measures

Work

Total time to execute everything on a single processor.

\[ \sum = 30 \]
Performance Measures

**Work**

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

**Span**

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

**Work**

Total time to execute everything on a single processor.

**Span**

Longest time to execute the threads along any path.

\[
\sum = 30 \\
\sum = 18 \\
\text{# nodes} = 5
\]
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 a single processor.

Span

Longest time to execute the threads along any path.

\[ \sum = 30 \]

\[ \sum = 18 \]

\# nodes = 5
Performance Measures

Work

Total time to execute everything on a 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 a 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 a 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.!

\[T_P \geq T_1\]

\[\text{Work Law}\]

Time on \(P\) processors can't be shorter than if all work all time

\[T_P \geq T_\infty\]

\[\text{Span Law}\]

Time on \(P\) processors can't be shorter than time on \(\infty\) processors

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

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

Maximum Speed-Up bounded by \(P!

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

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

- $T_1 = \text{work}$, $T_\infty = \text{span}$
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.!

$T_P \geq T_1$

Work Law

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

$T_P \geq T_\infty$

Span Law

Time on $P$ processors can't be shorter than time on $\infty$ processors

Speed-Up:

$\frac{T_1}{T_P}$

Parallelism:

$\frac{T_\infty}{T_1}$

Maximum Speed-Up bounded by $P$!

Maximum Speed-Up for $\infty$ processors!
Work Law and Span Law

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

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

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

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

$T_1 = 8, \ P = 2$

II. Matrix Multiplication

Digression: Multithreading
Work Law and Span Law

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

**Work Law**

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

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

**Span Law**

Running time actually also depends on scheduler etc.!

$$T_P \geq T_\infty$$

Maximum Speed-Up bounded by $P$

$T_1 = 8$, $P = 2$

$P = 2$ processors!
Work Law and Span Law

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

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

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

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

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

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

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

---

II. Matrix Multiplication

Digression: Multithreading
Work Law and Span Law

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

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.

$T_\infty = 5$
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: $\frac{T_1}{T_P}$

$T_\infty = 5$
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 $P$!

$T_\infty = 5$

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

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

**Work Law**

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

**Span Law**

$$T_P \geq T_\infty$$

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

$T_\infty = 5$
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:

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

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

Digression: 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

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

**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** \( y \)
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. \hspace{1em} $y_i = 0$
5. **parallel for** $i = 1$ to $n$
6. \hspace{1em} **for** $j = 1$ to $n$
7. \hspace{2em} $y_i = y_i + a_{ij}x_j$
8. **return** $y$

The **parallel for**-loops can be used since different entries of $y$ can be computed concurrently.
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. \textbf{parallel for } \( i = 1 \) to \( n \)
4. \hspace{1em} \( y_i = 0 \)
5. \textbf{parallel for } \( i = 1 \) to \( n \)
6. \hspace{1em} \textbf{for } \( j = 1 \) to \( n \)
7. \hspace{2em} \( y_i = y_i + a_{ij} x_j \)
8. \textbf{return } y

How can a compiler implement the \textbf{parallel for}-loop?
Implementing parallel for based on Divide-and-Conquer

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

\textbf{Mat-Vec}(A, x)
1 \quad n = A.rows
2 \quad \textbf{let} y \textbf{be 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

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

```
MAT-VEC(A, x)
    n = A.rows
    let y be a new vector of length n
    parallel for i = 1 to n
        y_i = 0
    parallel for i = 1 to n
        for j = 1 to n
            y_i = y_i + a_ij x_j
    return y
```
Implementing parallel for based on Divide-and-Conquer

\[ T_1(n) = \]

```
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
```
Implementing parallel for based on Divide-and-Conquer

\[ T_1(n) = \begin{align*} &\text{Work is equal to running time of its serialization; overhead of recursive spawning does not change asymptotics.} \\
\end{align*} \]

\textbf{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, i')
\begin{enumerate}
\item if \( i == i' \)
\item \hspace{1em} for \( j = 1 \) to \( n \)
\item \hspace{2em} \( y_i = y_i + a_{ij}x_j \)
\item else \( mid = \lfloor (i + i')/2 \rfloor \)
\item \hspace{1em} spawn \textbf{MAT-VEC-MAIN-LOOP}(A, x, y, n, i, mid)
\item \hspace{1em} \textbf{MAT-VEC-MAIN-LOOP}(A, x, y, n, mid + 1, i')
\item sync
\end{enumerate}

\textbf{MAT-VEC}(A, x)
\begin{enumerate}
\item \( n = A.\text{rows} \)
\item \hspace{1em} let \( y \) be a new vector of length \( n \)
\item \hspace{1em} \textbf{parallel for} \( i = 1 \) to \( n \)
\item \hspace{2em} \( y_i = 0 \)
\item \hspace{1em} \textbf{parallel for} \( i = 1 \) to \( n \)
\item \hspace{2em} \hspace{1em} \textbf{for} \( j = 1 \) to \( n \)
\item \hspace{2em} \hspace{2em} \hspace{1em} \( y_i = y_i + a_{ij}x_j \)
\item return \( y \)
Implementing parallel for based on Divide-and-Conquer

```
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
```

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

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

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

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

\[
T_\infty(n) =
\]

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

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

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

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

\[ T_\infty(n) = \text{Work is equal to running time of its serialization; overhead of recursive spawning does not change asymptotics.} \]

\[ \text{Span is the depth of recursive callings plus the maximum span of any of the } n \text{ iterations.} \]
Implementing parallel for based on Divide-and-Conquer

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

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

**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.
Implementing parallel for based on Divide-and-Conquer

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

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

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

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

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

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

1. \(n = A\).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 \(C\)

With a more careful implementation, \(T_\infty(n) = \Theta(\log n)\) (CLRS, Exercise 27.2-3)

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

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

1. \(n = A.rows\)
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. 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. \textbf{spawn} P-MATRIX-MULTIPLY-RECURSIVE\((C_{11}, A_{11}, B_{11})\)
7. \textbf{spawn} P-MATRIX-MULTIPLY-RECURSIVE\((C_{12}, A_{11}, B_{12})\)
8. \textbf{spawn} P-MATRIX-MULTIPLY-RECURSIVE\((C_{21}, A_{21}, B_{11})\)
9. \textbf{spawn} P-MATRIX-MULTIPLY-RECURSIVE\((C_{22}, A_{21}, B_{12})\)
10. \textbf{spawn} P-MATRIX-MULTIPLY-RECURSIVE\((T_{11}, A_{12}, B_{21})\)
11. \textbf{spawn} P-MATRIX-MULTIPLY-RECURSIVE\((T_{12}, A_{12}, B_{22})\)
12. \textbf{spawn} P-MATRIX-MULTIPLY-RECURSIVE\((T_{21}, A_{22}, B_{21})\)
13. \textbf{spawn} P-MATRIX-MULTIPLY-RECURSIVE\((T_{22}, A_{22}, B_{22})\)
14. \textbf{sync}
15. parallel for \(i = 1\) to \(n\)
16. \hspace{1em} parallel for \(j = 1\) to \(n\)
17. \hspace{2em} \(c_{ij} = c_{ij} + t_{ij}\)
The Simple Divide & Conquer Approach in Parallel

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

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

The same as before.

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

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

1. \(n = A\. rows\)
2. if \(n == 1\)
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
6. \(A_{11}, A_{12}, A_{21}, A_{22}; B_{11}, B_{12}, B_{21}, B_{22}; C_{11}, C_{12}, C_{21}, C_{22};\)
   and \(T_{11}, T_{12}, T_{21}, T_{22}\); respectively
7. spawn P-MATRIX-MULTIPLY-RECURSIVE\((C_{11}, A_{11}, B_{11})\)
8. spawn P-MATRIX-MULTIPLY-RECURSIVE\((C_{12}, A_{11}, B_{12})\)
9. spawn P-MATRIX-MULTIPLY-RECURSIVE\((C_{21}, A_{21}, B_{11})\)
10. spawn P-MATRIX-MULTIPLY-RECURSIVE\((C_{22}, A_{21}, B_{12})\)
11. spawn P-MATRIX-MULTIPLY-RECURSIVE\((T_{11}, A_{12}, B_{21})\)
12. spawn P-MATRIX-MULTIPLY-RECURSIVE\((T_{12}, A_{12}, B_{22})\)
13. spawn P-MATRIX-MULTIPLY-RECURSIVE\((T_{21}, A_{22}, B_{21})\)
14. spawn P-MATRIX-MULTIPLY-RECURSIVE\((T_{22}, A_{22}, B_{22})\)
15. sync
16. parallel for \(i = 1\) to \(n\)
17. parallel for \(j = 1\) to \(n\)
18. \(c_{ij} = c_{ij} + t_{ij}\)

The same as before.

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

\(T_\infty(n) = T_\infty(n/2) + \Theta(\log n)\)
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. spawn P-MATRIX-MULTIPLY-RECURSIVE\((C_{11}, A_{11}, B_{11})\)
   7. spawn P-MATRIX-MULTIPLY-RECURSIVE\((C_{12}, A_{11}, B_{12})\)
   8. spawn P-MATRIX-MULTIPLY-RECURSIVE\((C_{21}, A_{21}, B_{11})\)
   9. spawn P-MATRIX-MULTIPLY-RECURSIVE\((C_{22}, A_{21}, B_{12})\)
10. spawn P-MATRIX-MULTIPLY-RECURSIVE\((T_{11}, A_{12}, B_{21})\)
11. spawn P-MATRIX-MULTIPLY-RECURSIVE\((T_{12}, A_{12}, B_{22})\)
12. spawn P-MATRIX-MULTIPLY-RECURSIVE\((T_{21}, A_{22}, B_{21})\)
13. P-MATRIX-MULTIPLY-RECURSIVE\((T_{22}, A_{22}, B_{22})\)
14. sync
15. parallel for \(i = 1\) to \(n\)
   16. parallel for \(j = 1\) to \(n\)
      \(c_{ij} = c_{ij} + t_{ij}\)

The same as before.

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

\[T_\infty(n) = T_\infty(n/2) + \Theta(\log n)\]
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
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.
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 (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 $n/2 \times n/2$
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 \( \frac{n}{2} \times \frac{n}{2} \)

   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 \( P_i \).
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 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.

   \[ T_1(n) = \Theta(n^{\log 7}) \]
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.

\[ T_1(n) = \Theta(n^{\log 7}) \]
\[ T_\infty(n) = \Theta(\log^2 n) \]
Matrix Multiplication and Matrix Inversion

Speedups for Matrix Inversion by an equivalence with Matrix Multiplication.

Theorem 28.1 (Multiplication is no harder than Inversion)

Proof:

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

$$D = \begin{bmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{bmatrix}$$

Then $D^{-1}$ is:

$$D^{-1} = \begin{bmatrix} I_n & -A & 0 \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{bmatrix}$$

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.
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 \( O(I(n)) \).
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:

$$D = \begin{pmatrix} I_n & A & 0 \\ 0 & I_n & B \\ 0 & 0 & I_n \end{pmatrix} \ \Rightarrow \ \ D^{-1} = \begin{pmatrix} I_n & -A & AB \\ 0 & I_n & -B \\ 0 & 0 & I_n \end{pmatrix}.$$
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 \( D \) 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:

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

- 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:

$$
D = \begin{pmatrix}
l_n & A & 0 \\
0 & l_n & B \\
0 & 0 & l_n
\end{pmatrix}
\Rightarrow
D^{-1} = \begin{pmatrix}
l_n & -A & AB \\
0 & l_n & -B \\
0 & 0 & l_n
\end{pmatrix}.
$$

- 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.

⇒ 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 $O(M(n))$. 

Proof of this direction much harder (CLRS) – relies on properties of SPD matrices. Allows us to use Strassen's Algorithm to invert a matrix!
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 directon much harder (CLRS) – relies on properties of SPD matrices.