II. Matrix Multiplication

Thomas Sauerwald

Easter 2017

UNIVERSITY OF CAMBRIDGE

Outline

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

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

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

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

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
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

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

```
SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

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

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

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

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

SQUARE-MATRIX-MULTIPLY(A, B)

```
n = A.rows
let C be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

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:

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:

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:

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

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:

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

Each equation specifies two multiplications of $n/2 \times n/2$ matrices and the addition of their products.

Divide & Conquer: First Approach (Pseudocode)

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

Divide & Conquer: First Approach (Pseudocode)

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

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

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       c11 = a11 · b11
5   else partition A, B, and C as in equations (4.9)
6       C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
7       C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
8       C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
9       C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
              + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
10  return C
```

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

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

Divide & Conquer: First Approach (Pseudocode)

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

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

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

Divide & Conquer: First Approach (Pseudocode)

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

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

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

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

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ & \text{if } n > 1. \end{cases}$$

8 Multiplications

Divide & Conquer: First Approach (Pseudocode)

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

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

Divide & Conquer: First Approach (Pseudocode)

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
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)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
return C
```

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

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

8 Multiplications

4 Additions and Partitioning

Divide & Conquer: First Approach (Pseudocode)

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

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

```
SQUARE-MATRIX-MULTIPLY-RECURSIVE(A, B)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
        + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
        + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
        + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
        + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
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)
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
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)

```
n = A.rows
let C be a new n × n matrix
if n == 1
    c11 = a11 · b11
else partition A, B, and C as in equations (4.9)
    C11 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B21)
    C12 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A11, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A12, B22)
    C21 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B11)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B21)
    C22 = SQUARE-MATRIX-MULTIPLY-RECURSIVE(A21, B12)
          + SQUARE-MATRIX-MULTIPLY-RECURSIVE(A22, B22)
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)

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

$n = A.rows$
let C be a new $n \times n$ matrix
if $n == 1$
  $c_{11} = a_{11} \cdot b_{11}$
else partition A, B, and C as in equations (4.9)
  $C_{11} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{11})
    $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{21})
  $C_{12} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{11}, B_{12})
    $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{12}, B_{22})
  $C_{21} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{11})
    $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{21})
  $C_{22} =$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{21}, B_{12})
    $+$ SQUARE-MATRIX-MULTIPLY-RECURSIVE(A_{22}, B_{22})
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 $T(n) = 7 \cdot T(n/2) + \Theta(n^2) \Rightarrow T(n) = \Theta(n^{\log 7})$.

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

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

Details of Strassen's Algorithm

The 10 Submatrices and 7 Products

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

Claim

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

Details of Strassen's Algorithm

The 10 Submatrices and 7 Products

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

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

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

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

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

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:

$$
\begin{aligned}
P_5 + P_4 - P_2 + P_6 = {} & A_{11}B_{11} + A_{11}B_{22} + A_{22}B_{11} + A_{22}B_{22} + A_{22}B_{21} - A_{22}B_{11} \\
& - A_{11}B_{22} - A_{12}B_{22} + A_{12}B_{21} + A_{12}B_{22} - A_{22}B_{21} - A_{22}B_{22}
\end{aligned}
$$

Details of Strassen's Algorithm

The 10 Submatrices and 7 Products

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

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:

$$\begin{aligned}
P_5 + P_4 - P_2 + P_6 = A_{11}B_{11} &+ \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + \cancel{A_{22}B_{22}} + \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{11}} \\
&- \cancel{A_{11}B_{22}} - \cancel{A_{12}B_{22}} + A_{12}B_{21} + \cancel{A_{12}B_{22}} - \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{22}}
\end{aligned}$$

Details of Strassen's Algorithm

The 10 Submatrices and 7 Products

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

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:

$$\begin{aligned}
P_5 + P_4 - P_2 + P_6 &= A_{11}B_{11} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + \cancel{A_{22}B_{22}} + \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{11}} \\
&\quad - \cancel{A_{11}B_{22}} - \cancel{A_{12}B_{22}} + A_{12}B_{21} + \cancel{A_{12}B_{22}} - \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{22}} \\
&= A_{11}B_{11} + A_{12}B_{21}
\end{aligned}$$

Details of Strassen's Algorithm

The 10 Submatrices and 7 Products

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

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.

Proof:

$$
\begin{aligned}
P_5 + P_4 - P_2 + P_6 &= A_{11}B_{11} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + \cancel{A_{22}B_{22}} + \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{11}} \\
&\quad - \cancel{A_{11}B_{22}} - \cancel{A_{12}B_{22}} + A_{12}B_{21} + \cancel{A_{12}B_{22}} - \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{22}} \\
&= A_{11}B_{11} + A_{12}B_{21}
\end{aligned}
$$

Details of Strassen's Algorithm

The 10 Submatrices and 7 Products

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

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.

Proof:

$$\begin{aligned}
P_5 + P_4 - P_2 + P_6 &= A_{11}B_{11} + \cancel{A_{11}B_{22}} + \cancel{A_{22}B_{11}} + \cancel{A_{22}B_{22}} + \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{11}} \\
&\quad - \cancel{A_{11}B_{22}} - \cancel{A_{12}B_{22}} + A_{12}B_{21} + \cancel{A_{12}B_{22}} - \cancel{A_{22}B_{21}} - \cancel{A_{22}B_{22}} \\
&= A_{11}B_{11} + A_{12}B_{21} \qquad \square
\end{aligned}$$

Current State-of-the-Art

Open Problem: Is there an algorithm with quadratic complexity?

Current State-of-the-Art

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)

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)

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)
- ...

Outline

Memory Models

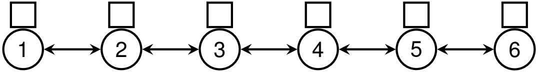
Distributed Memory

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

Memory Models

Distributed Memory

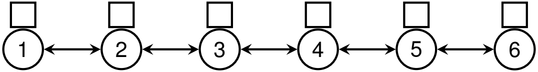
- Each processor has its private memory
- Access to memory of another processor via messages



Memory Models

Distributed Memory

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



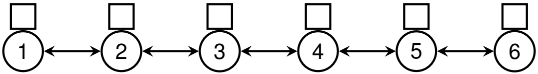
Shared Memory

- Central location of memory
- Each processor has direct access

Memory Models

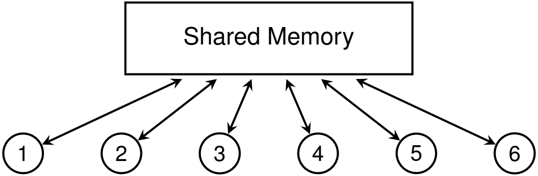
Distributed Memory

- 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

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:

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`**
  - (optinal) 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:

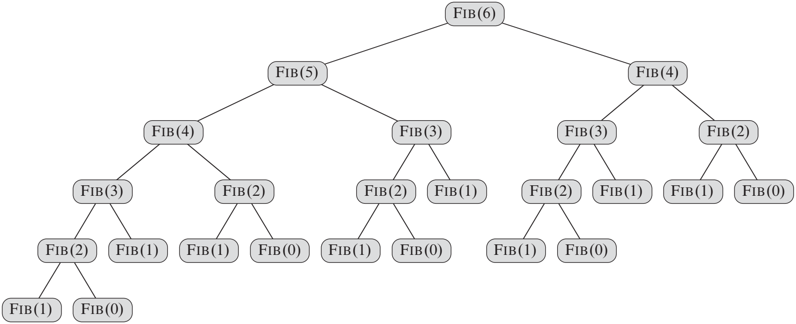
- **`spawn`**
  - (optional) prefix to a procedure call statement
  - procedure is executed in a separate thread
- **`sync`**
  - wait until all spawned threads are done
- **`parallel`**
  - (optinal) prefix to the standard loop **`for`**
  - each iteration is called in its own thread

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

Computing Fibonacci Numbers Recursively (Fig. 27.1)

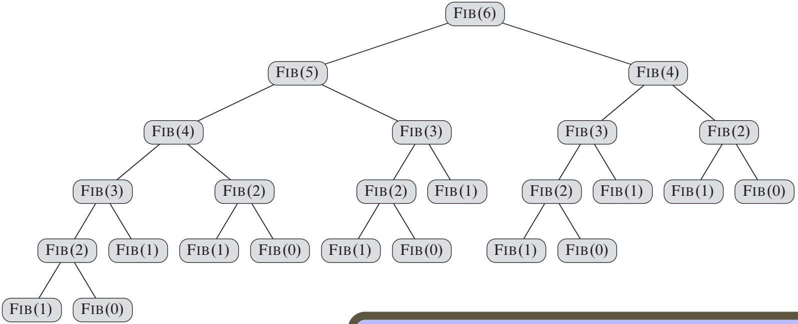
```
FIB(n)
  if n<=1 return n
  else x=FIB(n-1)
       y=FIB(n-2)
       return x+y
```

Computing Fibonacci Numbers Recursively (Fig. 27.1)




```
FIB(n)
  if n<=1 return n
  else x=FIB(n-1)
       y=FIB(n-2)
       return x+y
```

Computing Fibonacci Numbers Recursively (Fig. 27.1)



Very inefficient – exponential time!

```
FIB(n)
  if n<=1 return n
  else x=FIB(n-1)
       y=FIB(n-2)
       return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)

```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       return x+y
```

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)

```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)

Computation Dag $G = (V, E)$

```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       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)

```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       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

```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)

Computation Dag $G = (V, E)$

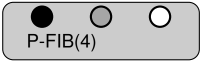
- V set of threads (instructions/strands without parallel control)
- E set of dependencies

```
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)

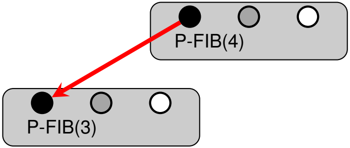
```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
        y=P-FIB(n-2)
        sync
        return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



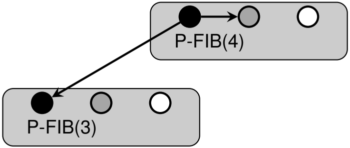

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
          y=P-FIB(n-2)
          sync
          return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



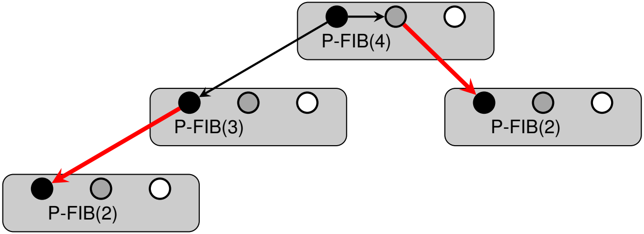

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
        y=P-FIB(n-2)
        sync
        return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



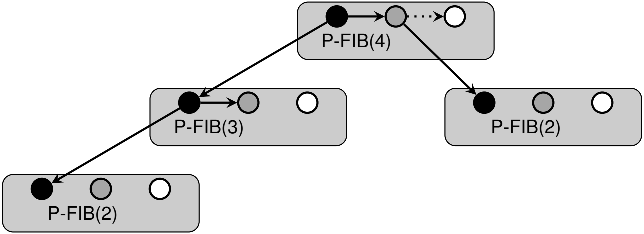

```
0: P-FIB(n)
1:    if n<=1 return n
2:    else x=spawn P-FIB(n-1)
3:         y=P-FIB(n-2)
4:         sync
5:         return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



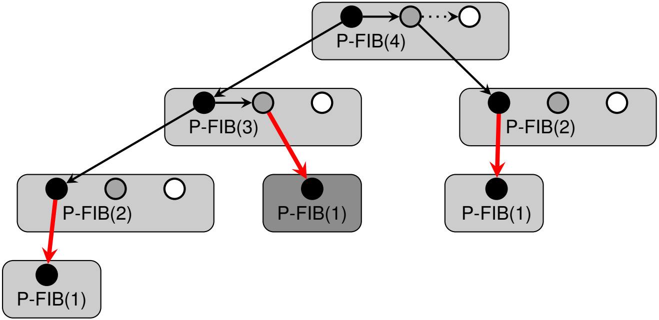

```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



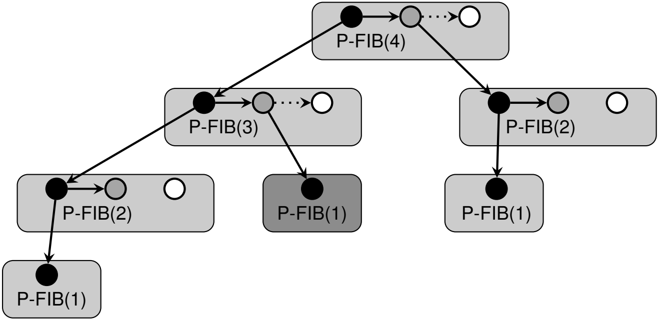

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
        y=P-FIB(n-2)
        sync
        return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



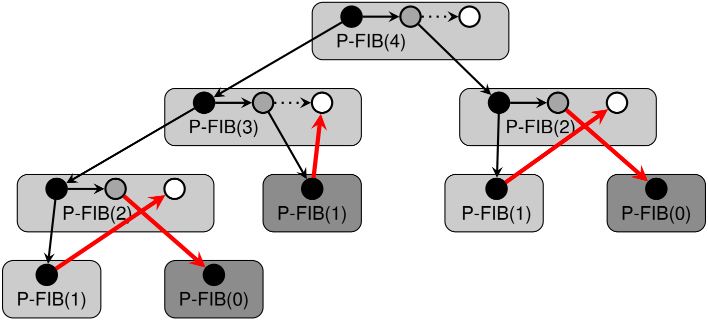

```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



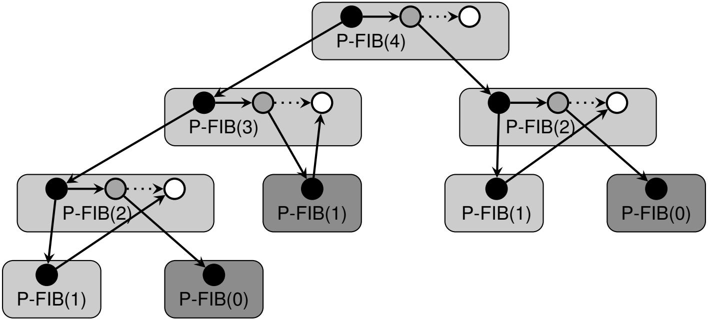

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
          y=P-FIB(n-2)
          sync
          return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



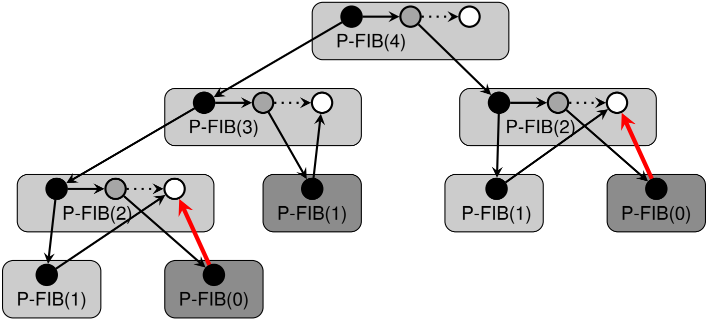

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
        y=P-FIB(n-2)
        sync
        return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



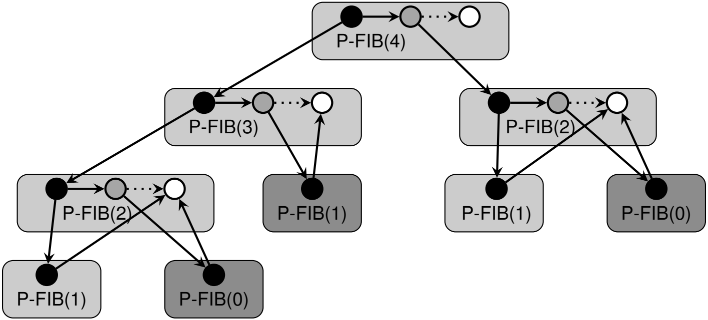

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
        y=P-FIB(n-2)
        sync
        return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



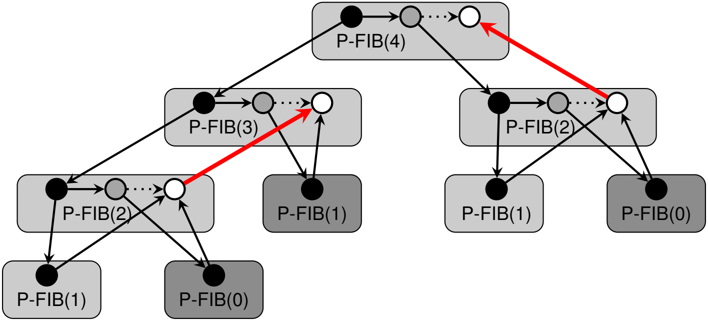

```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



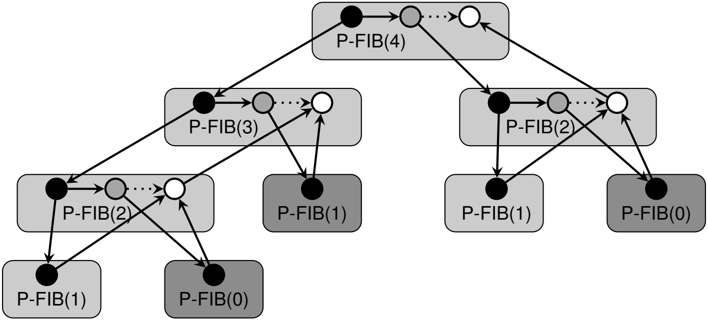

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
        y=P-FIB(n-2)
        sync
        return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



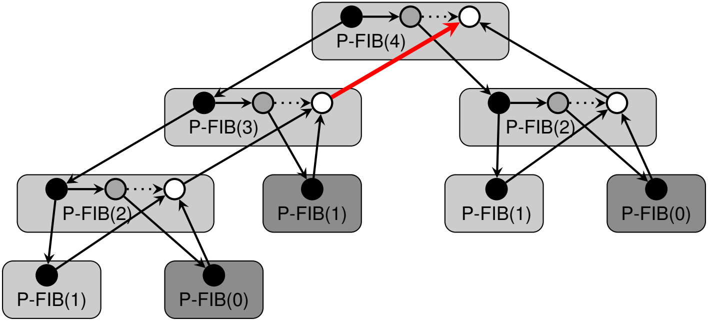

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
        y=P-FIB(n-2)
        sync
        return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



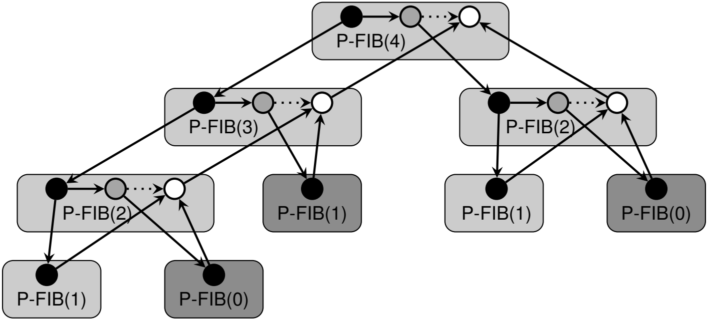

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
        y=P-FIB(n-2)
        sync
        return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



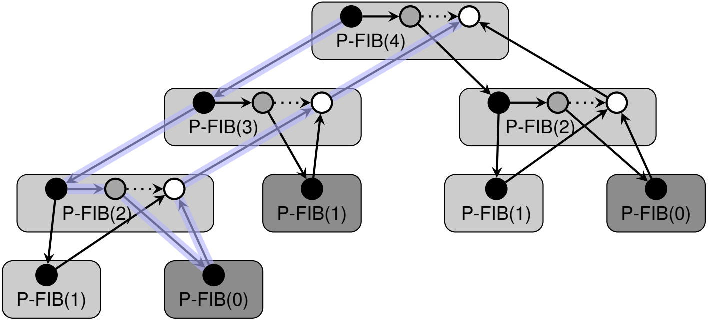

```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)



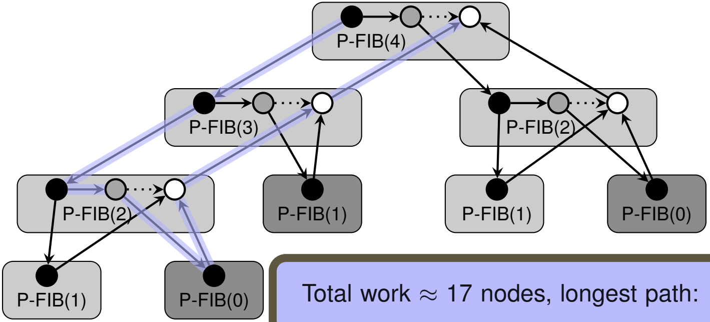

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
        y=P-FIB(n-2)
        sync
        return x+y
```

Computing Fibonacci Numbers in Parallel (Fig. 27.2)




```
P-FIB(n)
  if n<=1 return n
  else x=spawn P-FIB(n-1)
       y=P-FIB(n-2)
       sync
       return x+y
```

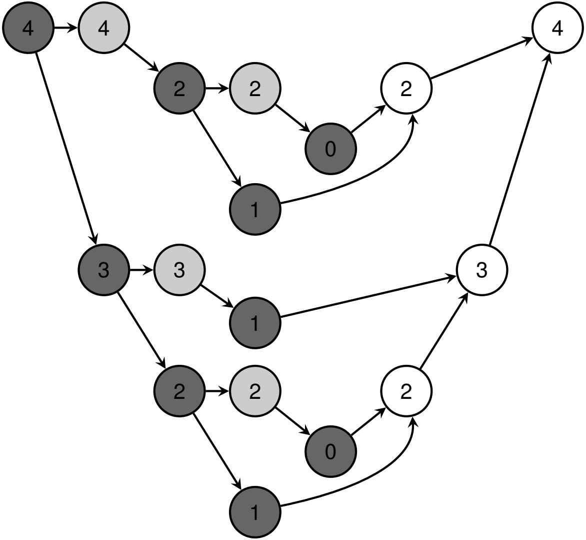
Computing Fibonacci Numbers in Parallel (Fig. 27.2)



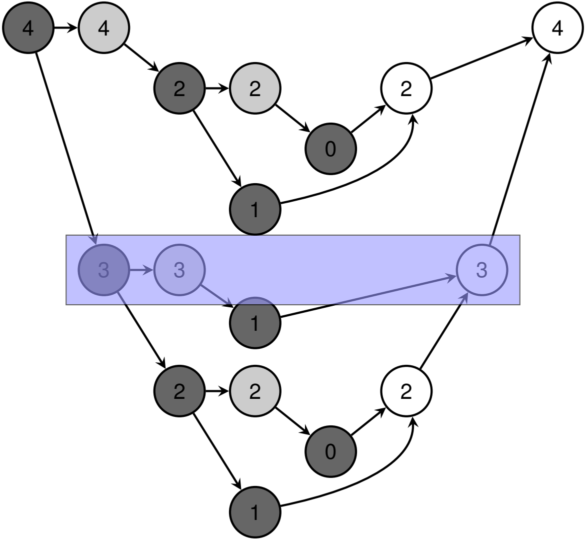
Total work $\approx$ 17 nodes, longest path: 8 nodes

```
P-FIB(n)
   if n<=1 return n
   else x=spawn P-FIB(n-1)
          y=P-FIB(n-2)
          sync
          return x+y
```

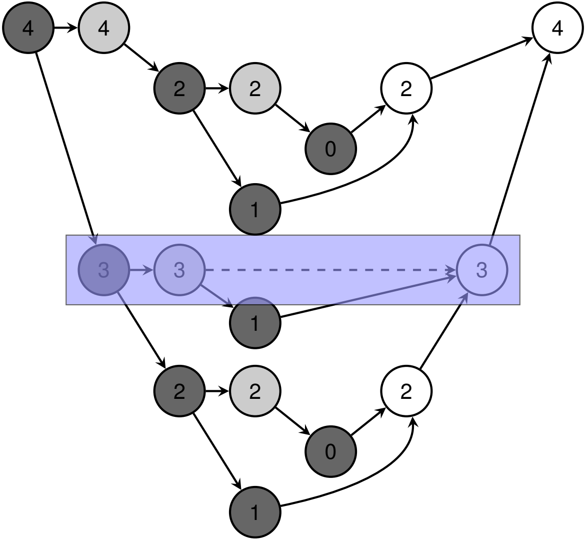
Computing Fibonacci Numbers in Parallel (DAG Perspective)



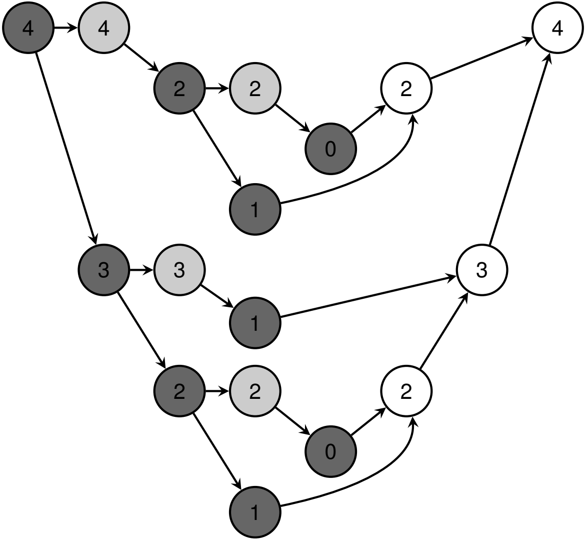
Computing Fibonacci Numbers in Parallel (DAG Perspective)



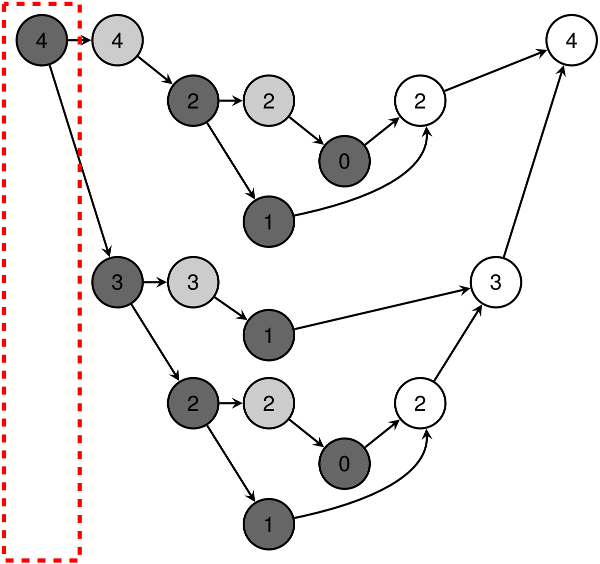
Computing Fibonacci Numbers in Parallel (DAG Perspective)



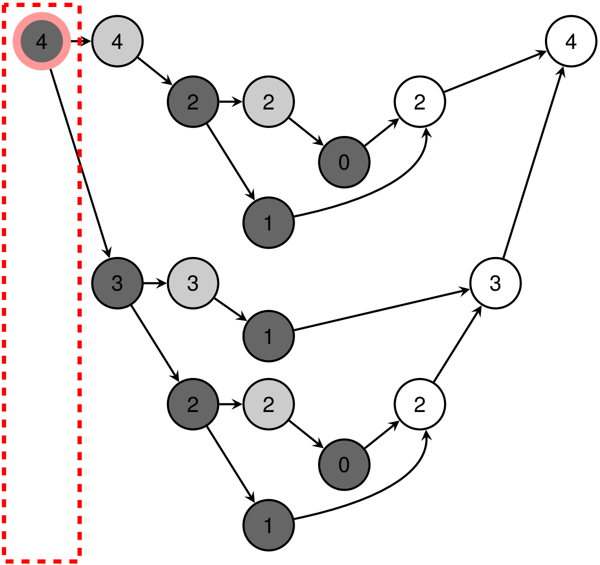
Computing Fibonacci Numbers in Parallel (DAG Perspective)



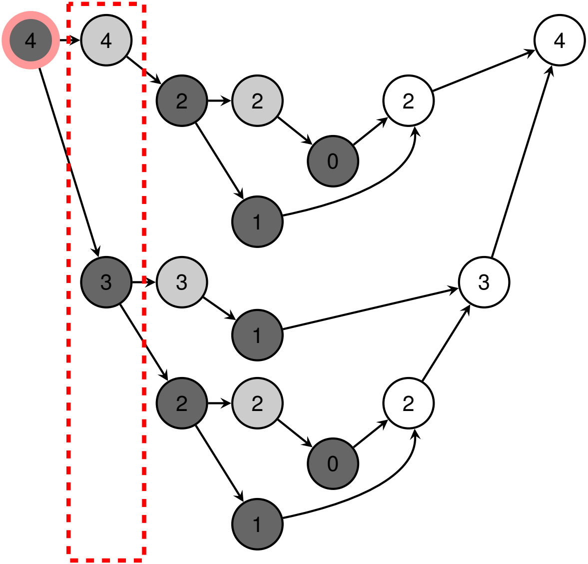
Computing Fibonacci Numbers in Parallel (DAG Perspective)



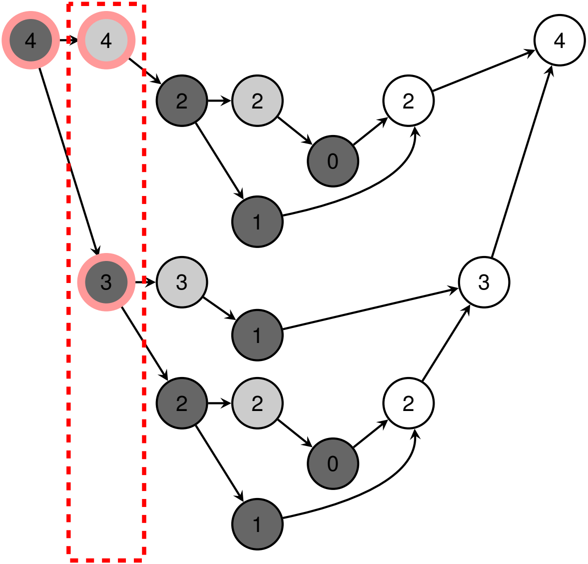
Computing Fibonacci Numbers in Parallel (DAG Perspective)



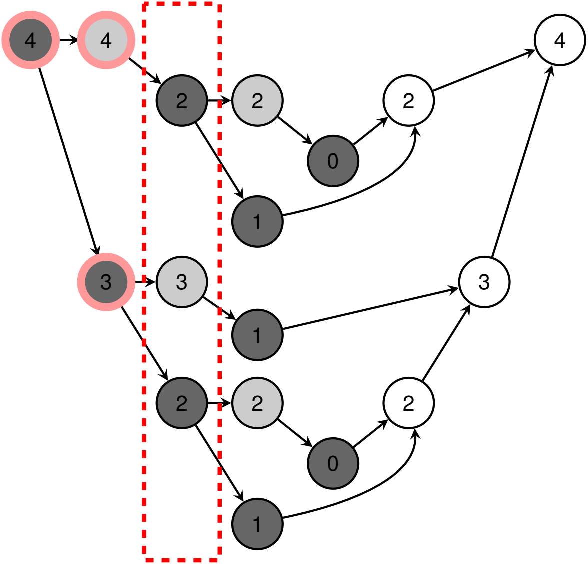
Computing Fibonacci Numbers in Parallel (DAG Perspective)



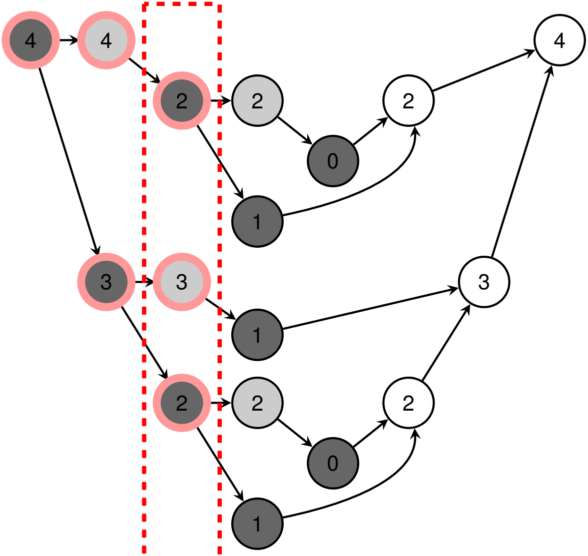
Computing Fibonacci Numbers in Parallel (DAG Perspective)



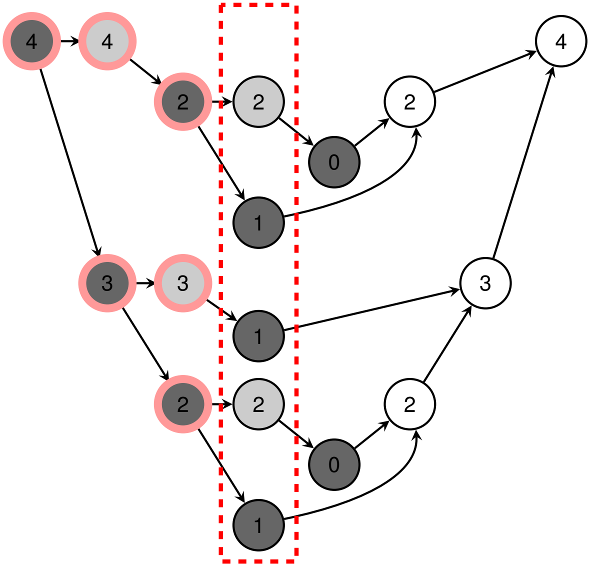
Computing Fibonacci Numbers in Parallel (DAG Perspective)



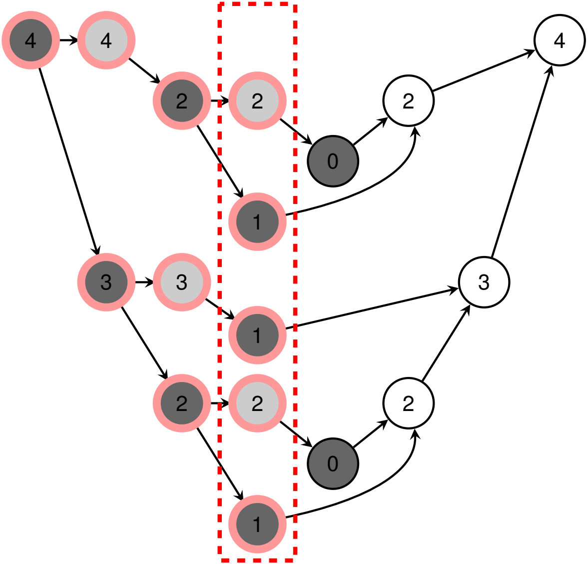
Computing Fibonacci Numbers in Parallel (DAG Perspective)



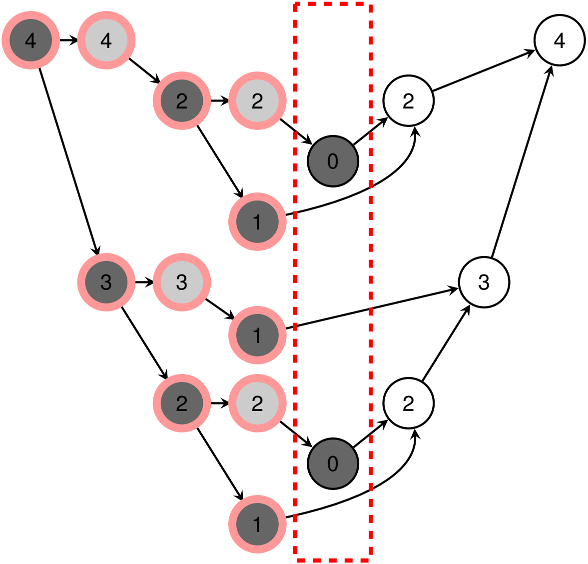
Computing Fibonacci Numbers in Parallel (DAG Perspective)



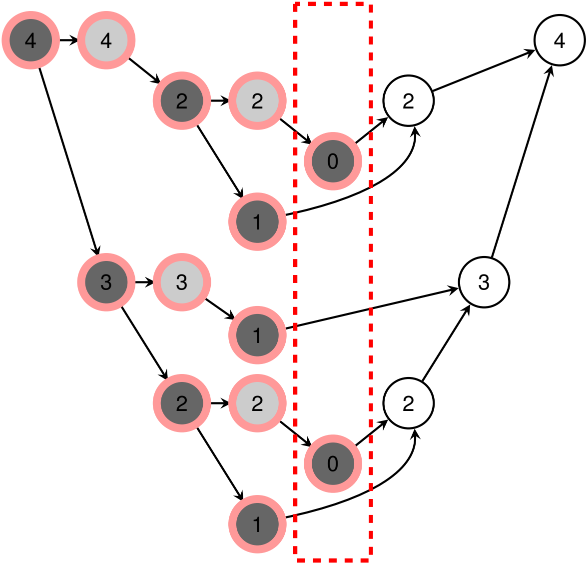
Computing Fibonacci Numbers in Parallel (DAG Perspective)



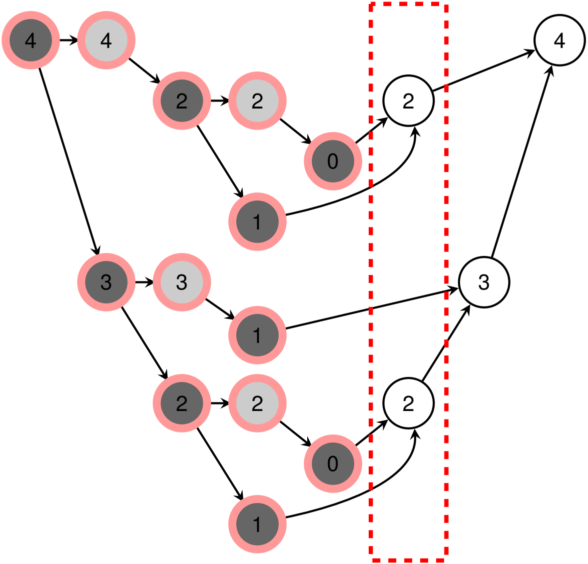
Computing Fibonacci Numbers in Parallel (DAG Perspective)



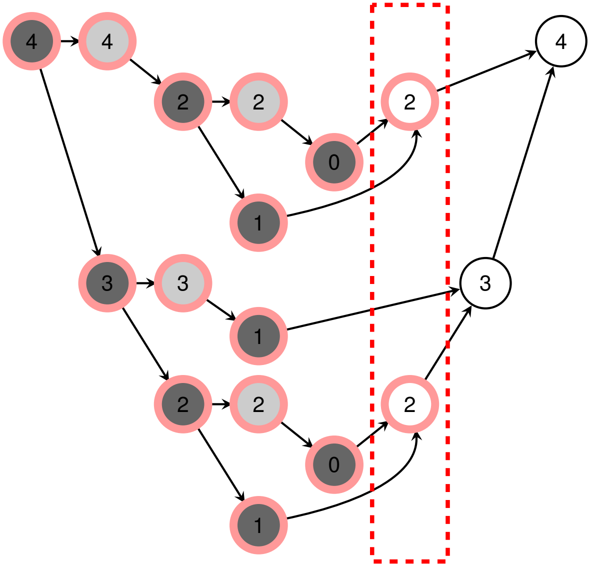
Computing Fibonacci Numbers in Parallel (DAG Perspective)



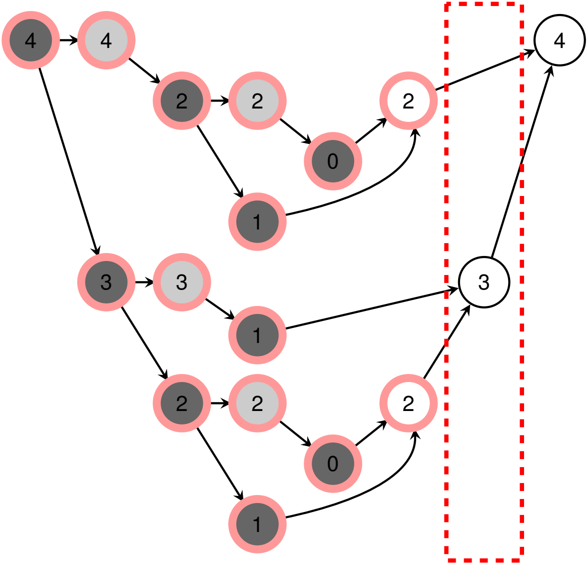
Computing Fibonacci Numbers in Parallel (DAG Perspective)



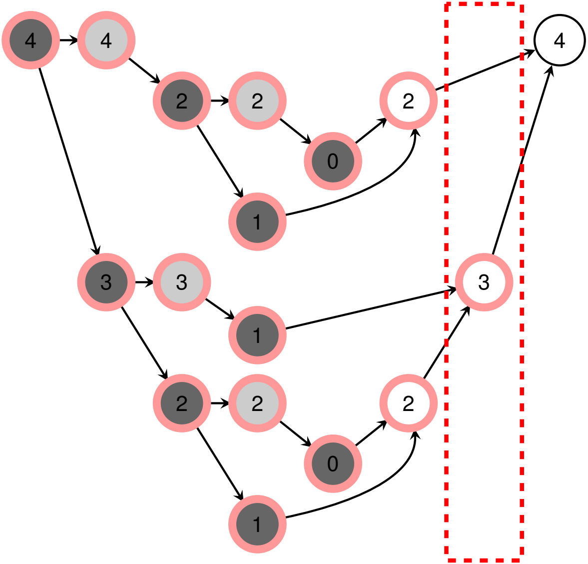
Computing Fibonacci Numbers in Parallel (DAG Perspective)



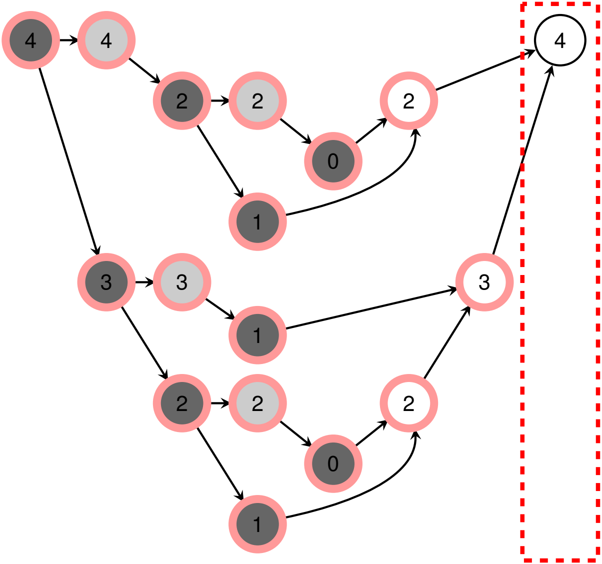
Computing Fibonacci Numbers in Parallel (DAG Perspective)



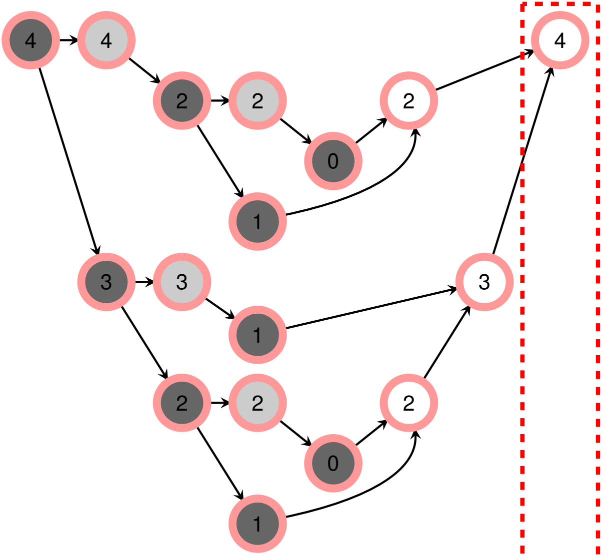
Computing Fibonacci Numbers in Parallel (DAG Perspective)



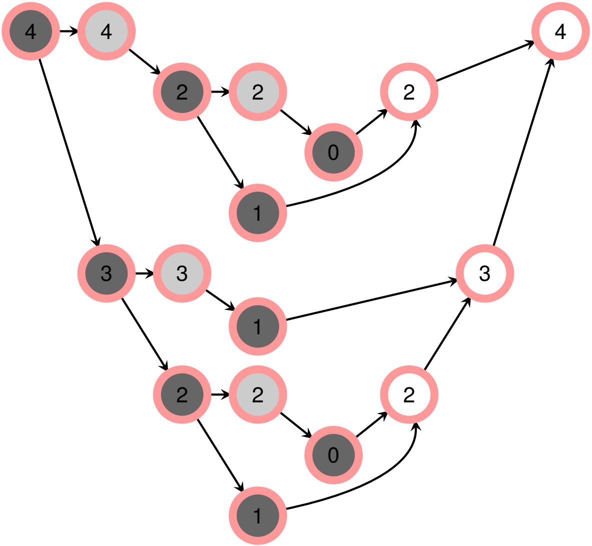
Computing Fibonacci Numbers in Parallel (DAG Perspective)



Computing Fibonacci Numbers in Parallel (DAG Perspective)



Computing Fibonacci Numbers in Parallel (DAG Perspective)



Performance Measures

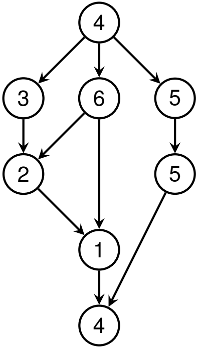
Work

Total time to execute everything on a single processor.

Performance Measures

Work

Total time to execute everything on a single processor.

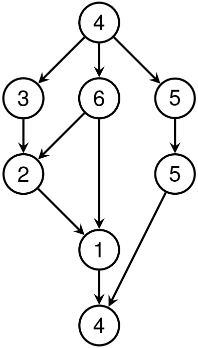


Performance Measures

$\sum = 30$

Work

Total time to execute everything on a single processor.



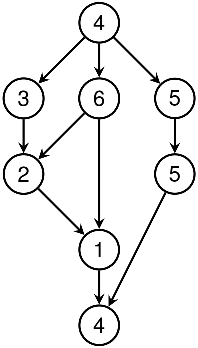
Performance Measures

Work

Total time to execute everything on a single processor.

Span

Longest time to execute the threads along any path.



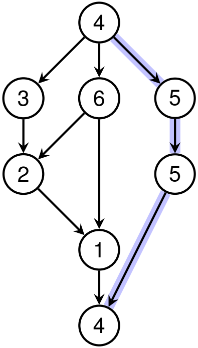
Performance Measures

Work

Total time to execute everything on a single processor.

Span

Longest time to execute the threads along any path.



Performance Measures

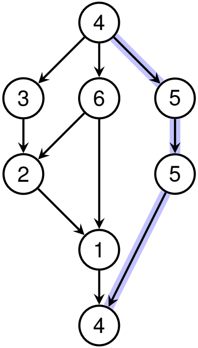
$\sum = 18$

Work

Total time to execute everything on a single processor.

Span

Longest time to execute the threads along any path.



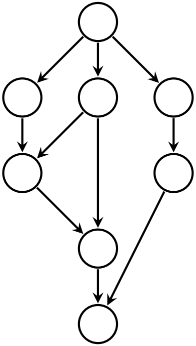
Performance Measures

Work

Total time to execute everything on a single processor.

Span

Longest time to execute the threads along any path.



Performance Measures

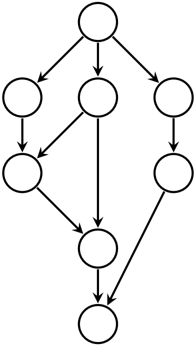
Work

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

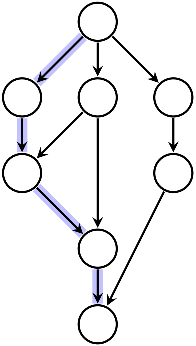
Work

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

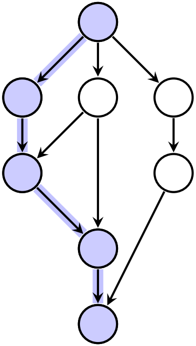
Total time to execute everything on a single processor.

Span

Longest time to execute the threads along any path.

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

$\#\text{nodes} = 5$



Work Law and Span Law

Work Law and Span Law

- $T_1 =$ work, $T_\infty =$ span

Work Law and Span Law

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

Work Law and Span Law

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

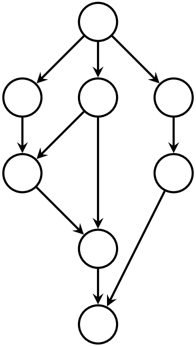
Running time actually also depends on scheduler etc.!

Work Law and Span Law

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

Work Law

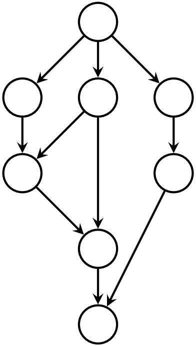
$$T_P \geq \frac{T_1}{P}$$



Work Law and Span Law

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

$T_1 = 8,\ P = 2$



Work Law

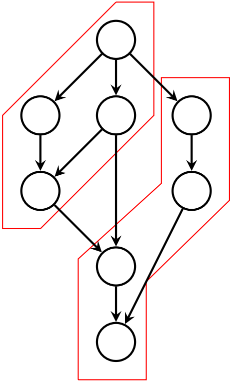
$$T_P \geq \frac{T_1}{P}$$

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

Work Law and Span Law

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

$T_1 = 8$, $P = 2$



Work Law

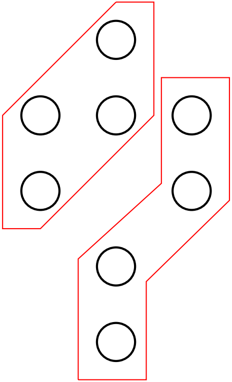
$$T_P \geq \frac{T_1}{P}$$

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

Work Law and Span Law

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

$T_1 = 8,\ P = 2$



Work Law

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

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

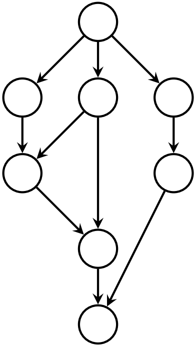
Work Law and Span Law

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

Work Law

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

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



Work Law and Span Law

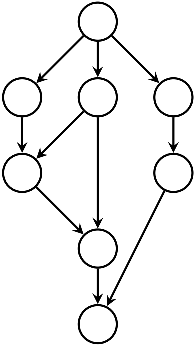
- T_1 = work, T_∞ = span
- P = number of (identical) processors
- T_P = running time on P processors

Work Law

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

Span Law

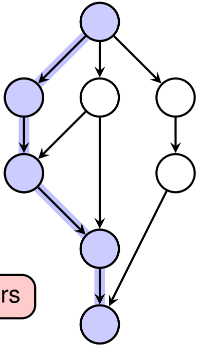
$$T_P \geq T_\infty$$



Work Law and Span Law

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

$T_\infty = 5$



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 ∞ processors

Work Law and Span Law

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

Work Law

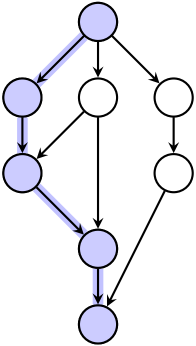
$$T_P \geq \frac{T_1}{P}$$

Span Law

$$T_P \geq T_\infty$$

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

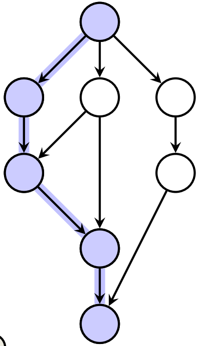
$T_\infty = 5$



Work Law and Span Law

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

$T_\infty = 5$



Work Law

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

Span Law

$$T_P \geq T_\infty$$

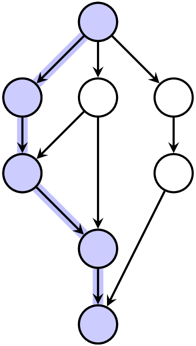
- Speed-Up: $\frac{T_1}{T_P}$

Maximum Speed-Up bounded by P!

Work Law and Span Law

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

$T_\infty = 5$



Work Law

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

Span Law

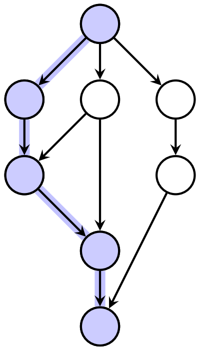
$$T_P \geq T_\infty$$

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

Work Law and Span Law

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

$T_\infty = 5$



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 ∞ 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

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

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

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

```
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
```

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

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

```
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
```

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

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

```
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
```

The **parallel for**-loops can be used since different entries of y can be computed concurrently.

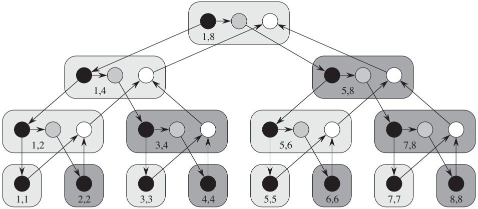
How can a compiler implement the **parallel for**-loop?

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



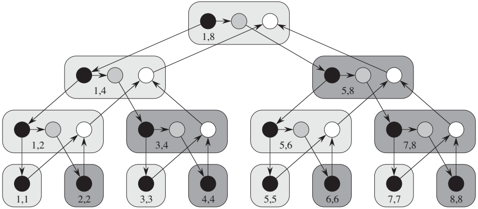
MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')

if $i == i'$
    for $j = 1$ **to** n
        $y_i = y_i + a_{ij} x_j$
else $mid = \lfloor (i + i')/2 \rfloor$
    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



MAT-VEC-MAIN-LOOP(A, x, y, n, i, i')

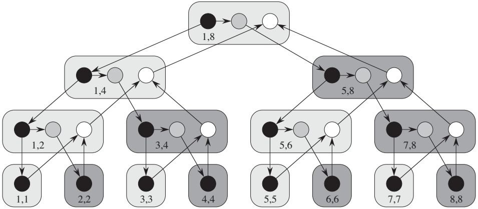
if $i == i'$
    for $j = 1$ **to** n
        $y_i = y_i + a_{ij} x_j$
else $mid = \lfloor (i + i')/2 \rfloor$
    spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
    MAT-VEC-MAIN-LOOP$(A, x, y, n, mid + 1, i')$
    sync

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

$$T_1(n) =$$

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 $mid = \lfloor (i + i')/2 \rfloor$
  spawn MAT-VEC-MAIN-LOOP(A, x, y, n, i, mid)
  MAT-VEC-MAIN-LOOP$(A, x, y, n, mid + 1, i')$
  sync

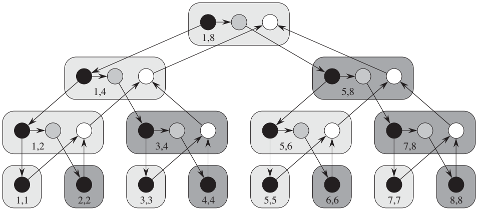
MAT-VEC(A, x)

$n = A.rows$
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

$T_1(n) =$

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



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
```

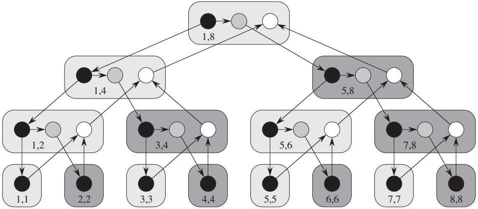
MAT-VEC(A, x)

```
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
```

$$T_1(n) = \Theta(n^2)$$

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



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 $mid = \lfloor (i + i')/2 \rfloor$
    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

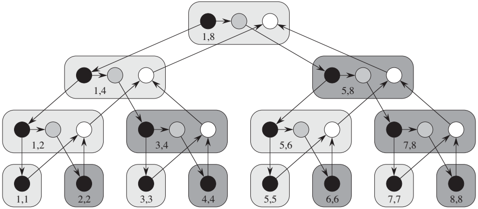
$$T_1(n) = \Theta(n^2)$$

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.

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
```

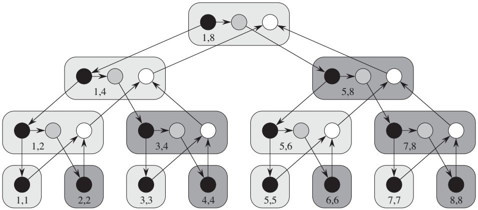
$$T_1(n) = \Theta(n^2)$$

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.

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
```

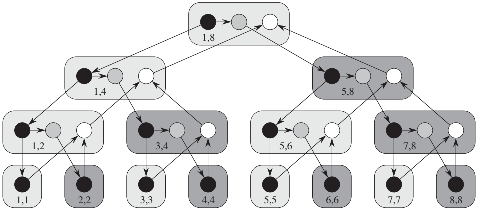
$$T_1(n) = \Theta(n^2)$$

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.

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 $mid = \lfloor (i + i')/2 \rfloor$
    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

$$T_1(n) = \Theta(n^2)$$

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

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

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)
n = A.rows
let C be a new n × n matrix
parallel for i = 1 to n
    parallel for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

Naive Algorithm in Parallel

```
P-SQUARE-MATRIX-MULTIPLY(A, B)
n = A.rows
let C be a new n × n matrix
parallel for i = 1 to n
    parallel for j = 1 to n
        c_ij = 0
        for k = 1 to n
            c_ij = c_ij + a_ik · b_kj
return C
```

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

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)
n = A.rows
if n == 1
    c11 = a11 b11
else let T be a new n × n matrix
    partition A, B, C, and T into n/2 × n/2 submatrices
        A11, A12, A21, A22; B11, B12, B21, B22; C11, C12, C21, C22;
        and T11, T12, T21, T22; respectively
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C11, A11, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C12, A11, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C21, A21, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C22, A21, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T11, A12, B21)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T12, A12, B22)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T21, A22, B21)
    P-MATRIX-MULTIPLY-RECURSIVE(T22, A22, B22)
    sync
    parallel for i = 1 to n
        parallel for j = 1 to n
            cij = cij + tij
```

The Simple Divide&Conquer Approach in Parallel

```
P-MATRIX-MULTIPLY-RECURSIVE(C, A, B)
n = A.rows
if n == 1
    c11 = a11 b11
else let T be a new n × n matrix
    partition A, B, C, and T into n/2 × n/2 submatrices
        A11, A12, A21, A22; B11, B12, B21, B22; C11, C12, C21, C22;
        and T11, T12, T21, T22; respectively
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C11, A11, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C12, A11, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C21, A21, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C22, A21, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T11, A12, B21)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T12, A12, B22)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T21, A22, B21)
    P-MATRIX-MULTIPLY-RECURSIVE(T22, A22, B22)
    sync
    parallel for i = 1 to n
        parallel for j = 1 to n
            cij = cij + tij
```

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)
n = A.rows
if n == 1
    c11 = a11 b11
else let T be a new n × n matrix
    partition A, B, C, and T into n/2 × n/2 submatrices
        A11, A12, A21, A22; B11, B12, B21, B22; C11, C12, C21, C22;
        and T11, T12, T21, T22; respectively
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C11, A11, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C12, A11, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C21, A21, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C22, A21, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T11, A12, B21)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T12, A12, B22)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T21, A22, B21)
    P-MATRIX-MULTIPLY-RECURSIVE(T22, A22, B22)
    sync
    parallel for i = 1 to n
        parallel for j = 1 to n
            cij = cij + tij
```

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)
n = A.rows
if n == 1
    c11 = a11 b11
else let T be a new n × n matrix
    partition A, B, C, and T into n/2 × n/2 submatrices
        A11, A12, A21, A22; B11, B12, B21, B22; C11, C12, C21, C22;
        and T11, T12, T21, T22; respectively
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C11, A11, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C12, A11, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C21, A21, B11)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(C22, A21, B12)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T11, A12, B21)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T12, A12, B22)
    spawn P-MATRIX-MULTIPLY-RECURSIVE(T21, A22, B21)
    P-MATRIX-MULTIPLY-RECURSIVE(T22, A22, B22)
    sync
    parallel for i = 1 to n
        parallel for j = 1 to n
            cij = cij + tij
```

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 $n/2 \times n/2$ submatrices

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.

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 $n/2 \times n/2$ 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 $n/2 \times n/2$ 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.

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.

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 $O(I(n))$.

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:

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

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 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:

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

- 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. $\square$

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 $O(M(n))$.

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 directon 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.