

# CSE 6220/CX 4220

## Introduction to HPC

# Lecture 3: Analysis of Multithreaded Algorithms

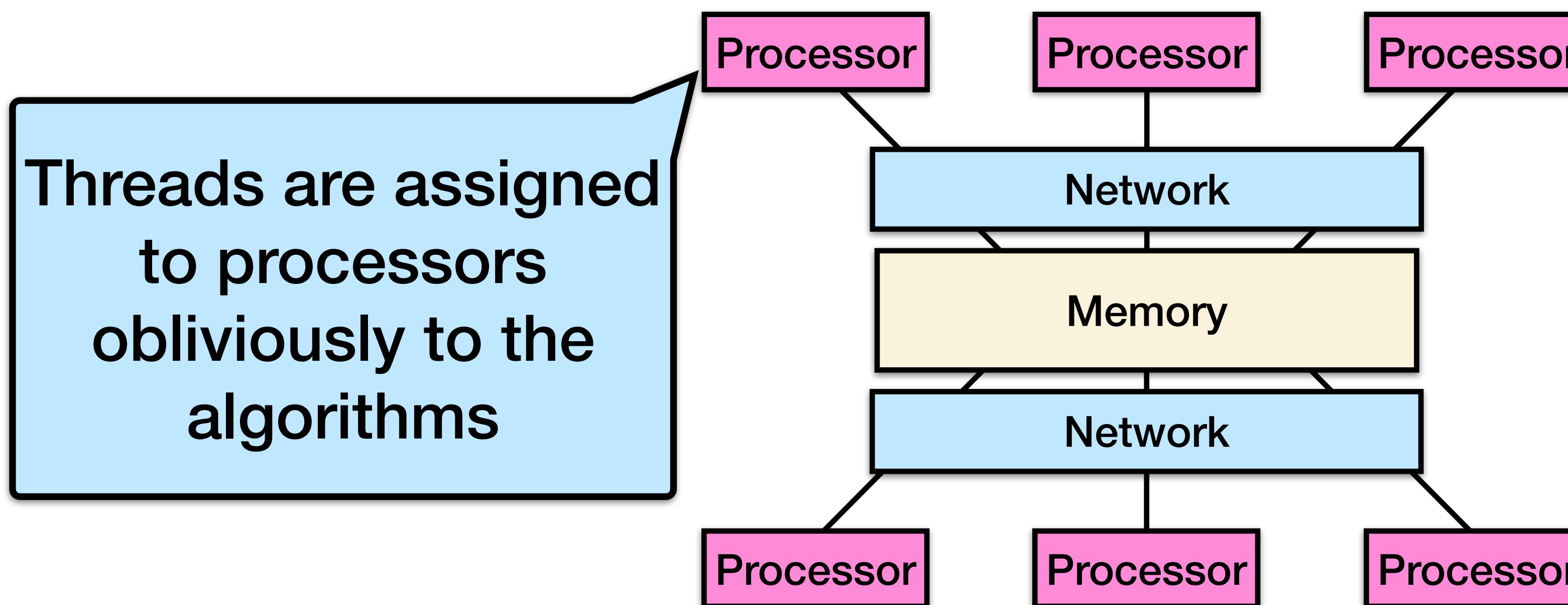
Helen Xu  
[hxu615@gatech.edu](mailto:hxu615@gatech.edu)



Georgia Tech College of Computing  
School of Computational  
Science and Engineering

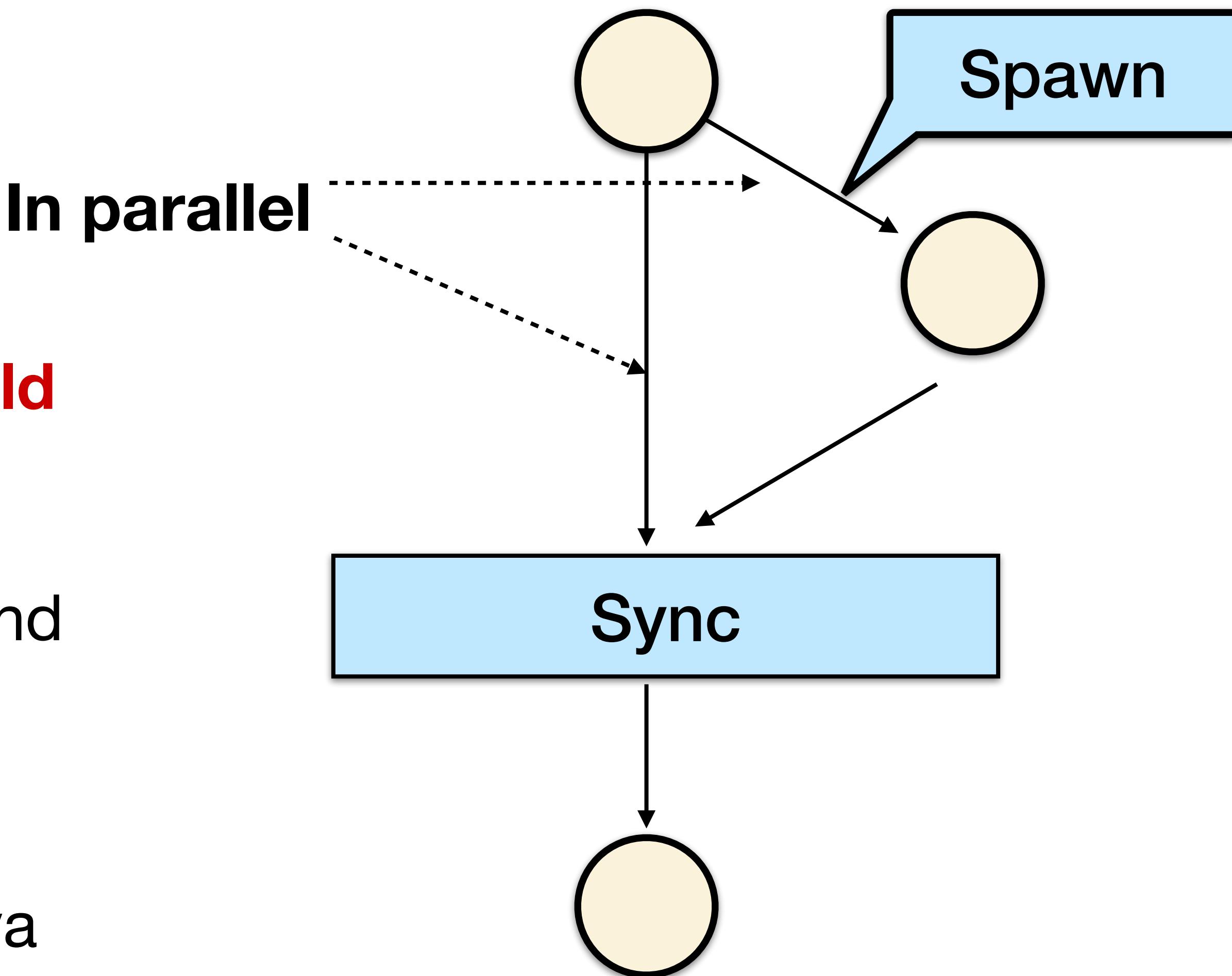
# Binary forking model

- In the binary forking model for multithreaded computations, threads can be created dynamically and can **run asynchronously in parallel**.
- Each thread acts like a standard **random-access machine** (RAM), with a constant number of registers, and a shared main memory.



# Binary forking primitives

- The model includes a fork (also called spawn) instruction that **forks an asynchronous child** thread.
- The model includes a sync instruction that signals to the algorithm to **wait until all child threads are finished**.
- A computation starts with a single thread and **finishes when all threads end**.
- The binary forking abstraction is supported by programming systems e.g., Cilk, the Java fork-join framework, Habanero, Intel Thread Building Blocks (TBB), etc.



# Example: Fib via binary forking

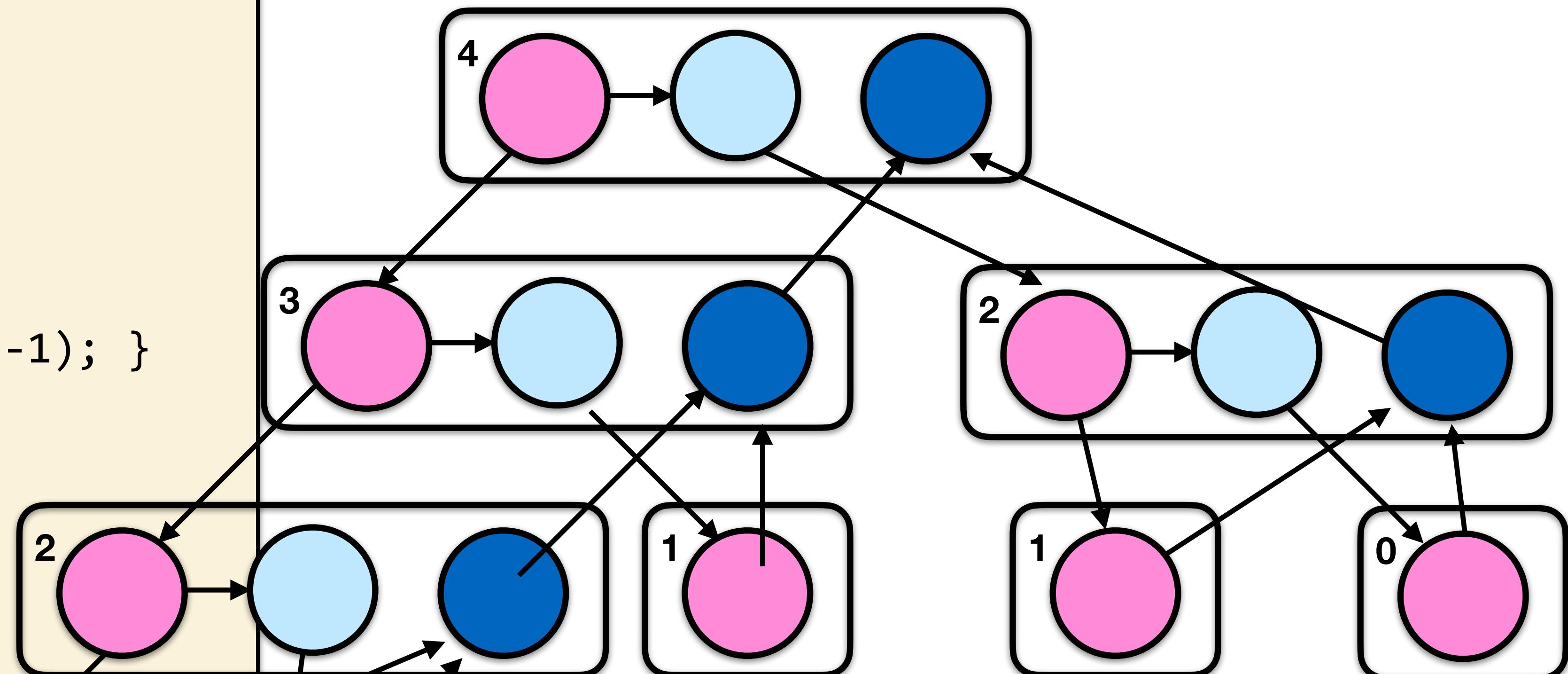
Example: fib(4)

```
int64_t fib(int64_t n) {
    if (n < 2) {
        return n;
    } else {
        int64_t x, y;
        parallel_spawn { fib(n-1); }
        fib(n-2);
        parallel_sync;
        return (x + y);
    }
}
```

# Example: Fib via binary forking

```
int64_t fib(int64_t n) {  
    if (n < 2) {  
        return n;  
    } else {  
        int64_t x, y;  
        parallel_spawn { fib(n-1); }  
        fib(n-2);  
        parallel_sync;  
        return (x + y);  
    }  
}
```

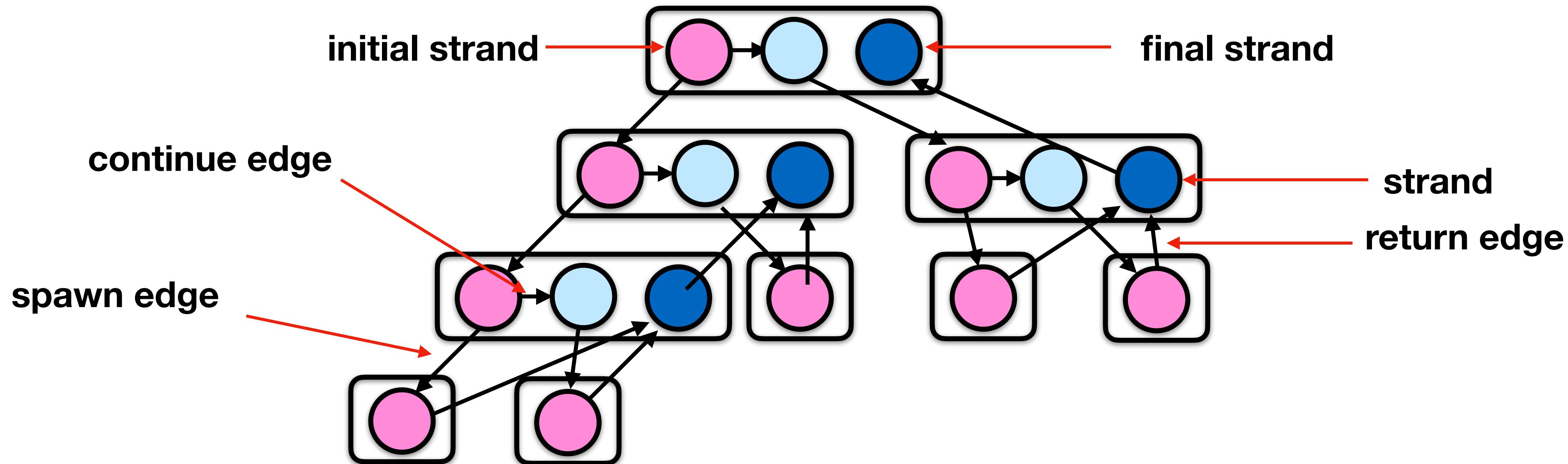
Example: fib(4)



**“Processor oblivious”**

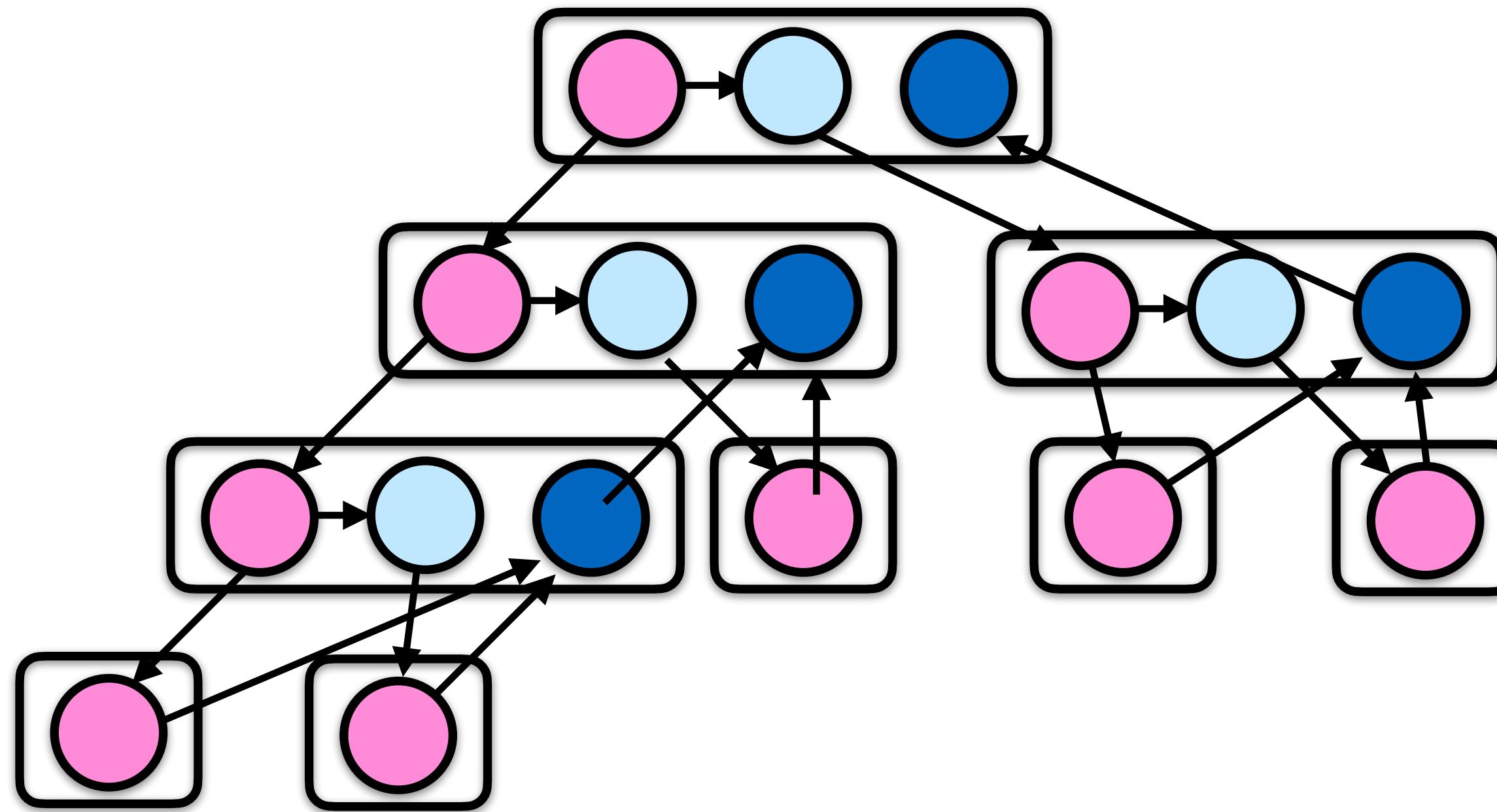
The **computation DAG** unfolds dynamically.

# Computation DAG



- A parallel instruction stream is a dag  $G = (V, E)$ .
- Each vertex  $v \in V$  is a **strand**: a sequence of instructions not containing a spawn, sync, or return from a spawn.
- An edge  $e \in E$  is a spawn, call (function), return, or continue edge.
- Loop parallelism (parallel for) can be converted (for analysis purposes) to task spawn and wait using recursive divide-and-conquer.

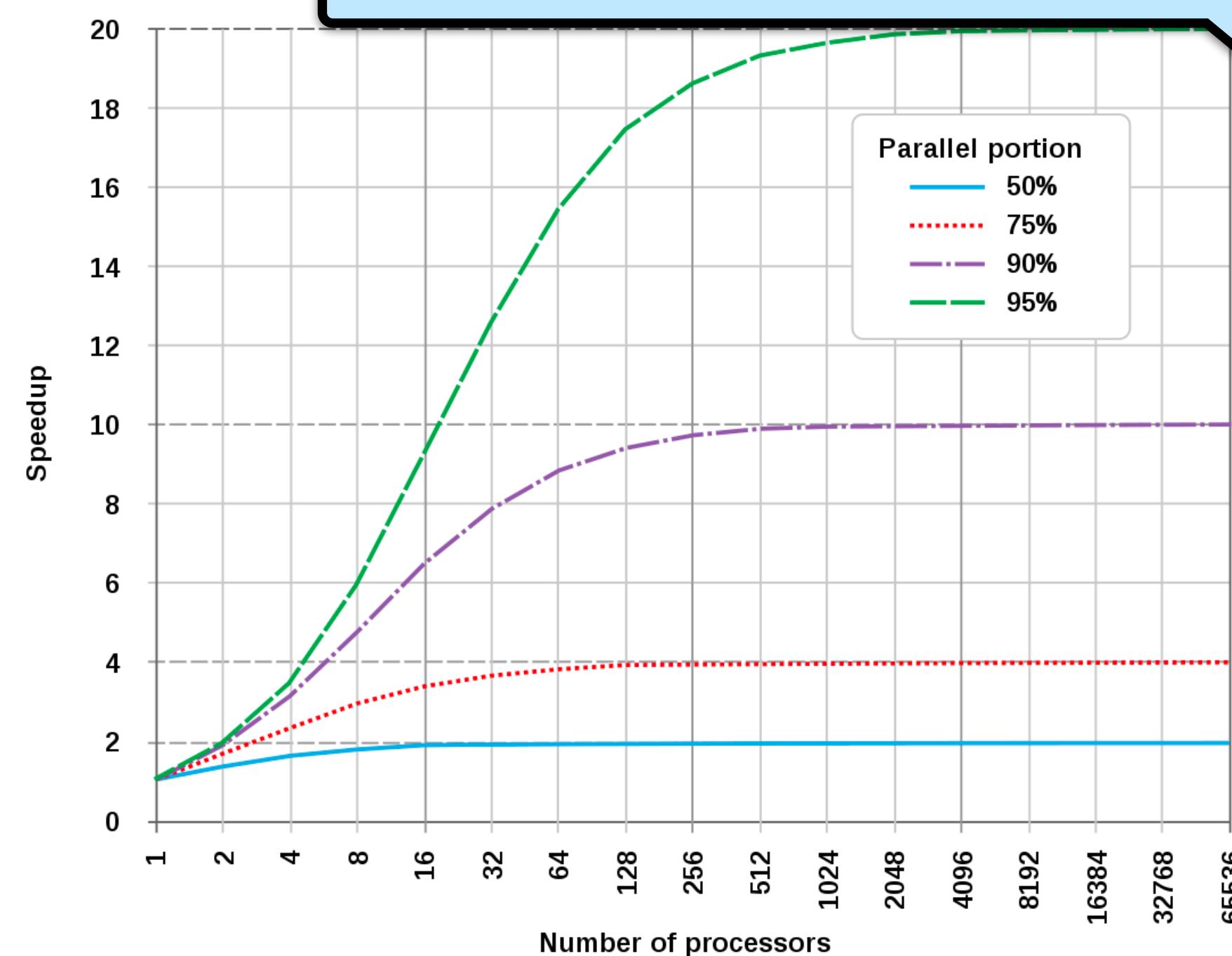
# How Much Parallelism?



Assuming that each strand executes in unit time, what is the **parallelism** of this computation?

# Recall: Amdahl's Law

Work done by the  
**best sequential  
algorithm**



Amdahl's law bounds **strong scaling**, or the speedup of a fixed problem given more parallel resources.

$s$  = fraction of work done sequentially (Amdahl fraction)

$1-s$  = parallelizable fraction

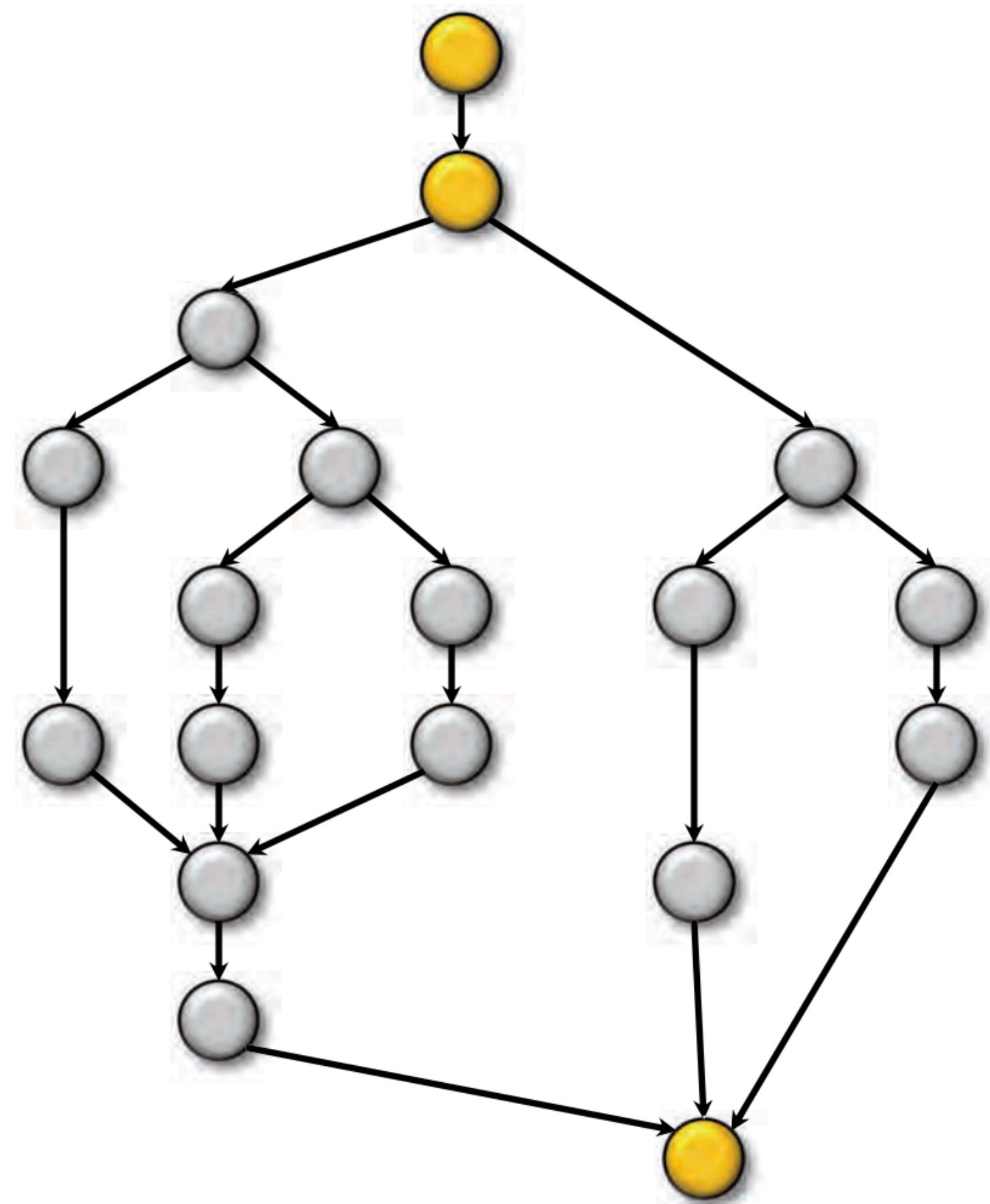
$P$  = number of processors

$$\begin{aligned} \text{Speedup (P)} &= \text{Time}(1) / \text{Time}(P) \\ &\leq 1/(s + (1-s)/P) \\ &\leq 1/s \end{aligned}$$

For a fixed problem, the upper limit of speedup is **determined by the serial fraction**.

# Quantifying Parallelism

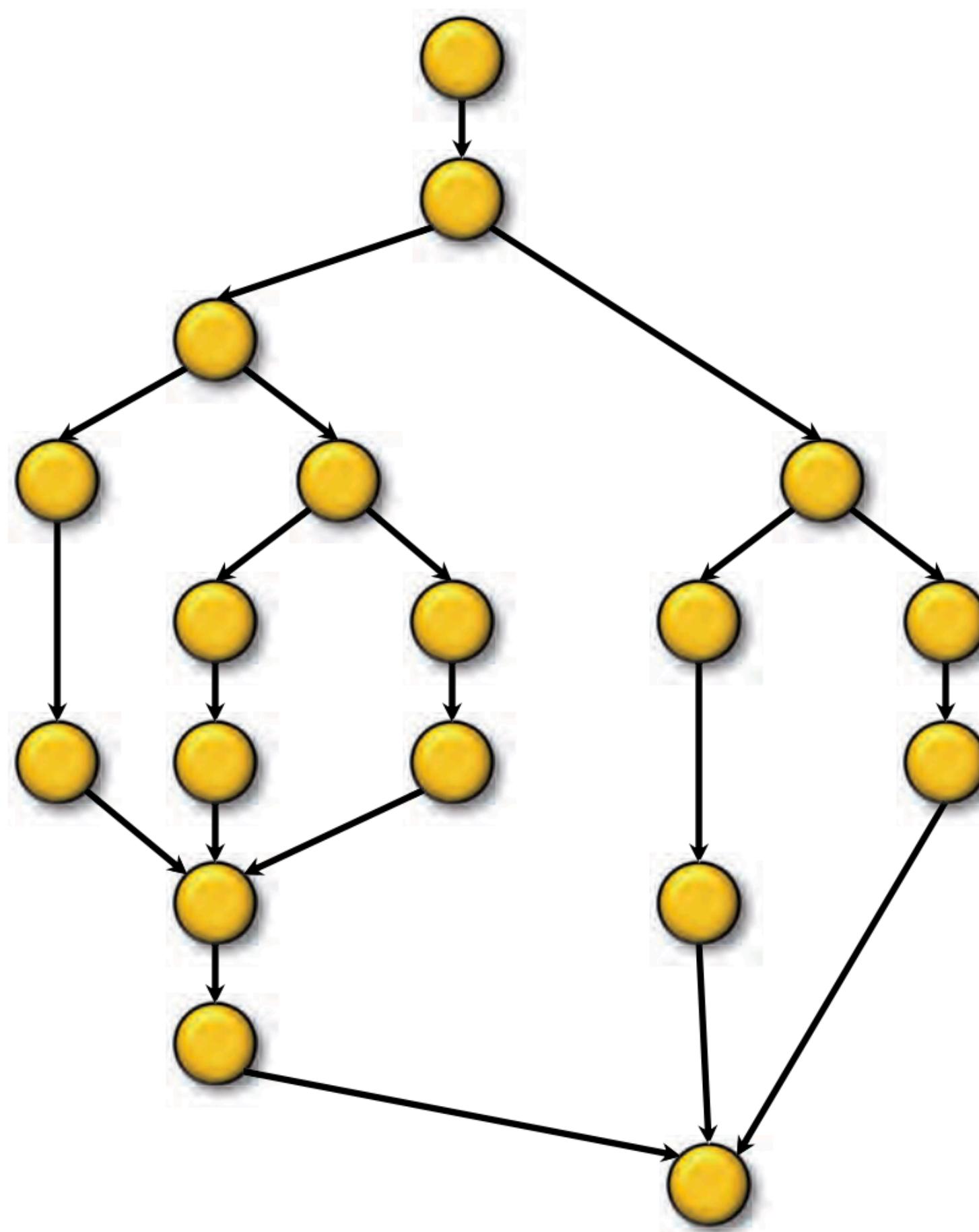
What is the **parallelism** of this computation?



Amdahl's Law says that since the serial fraction is  $3/18 = 1/6$ , the speedup is upper bounded by 6.

# Performance Measures

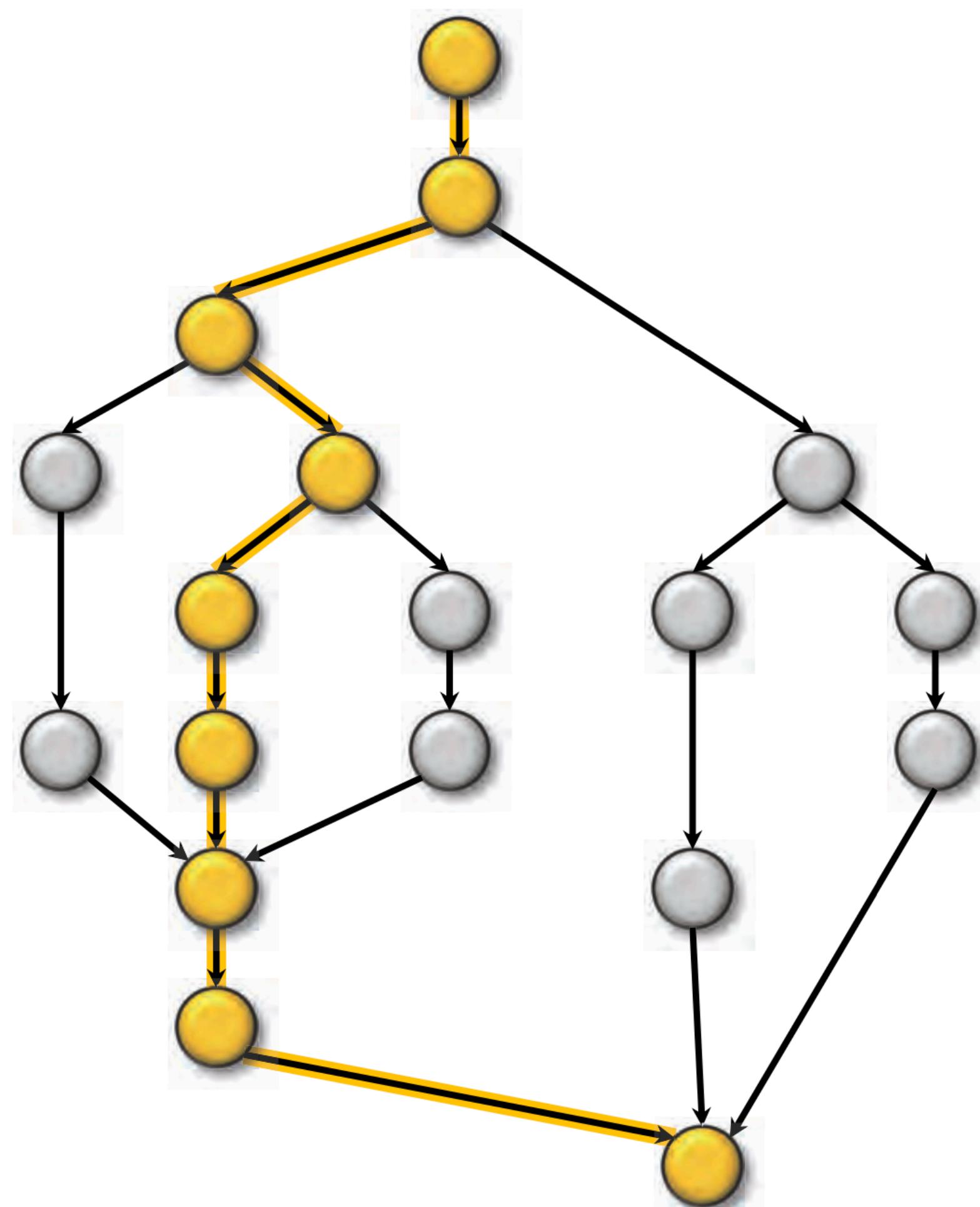
$T_p$  = execution time on P processors



$$T_1 = \text{work} \\ = 18$$

# Performance Measures

$T_p$  = execution time on P processors



$$T_1 = \text{work} \\ = 18$$

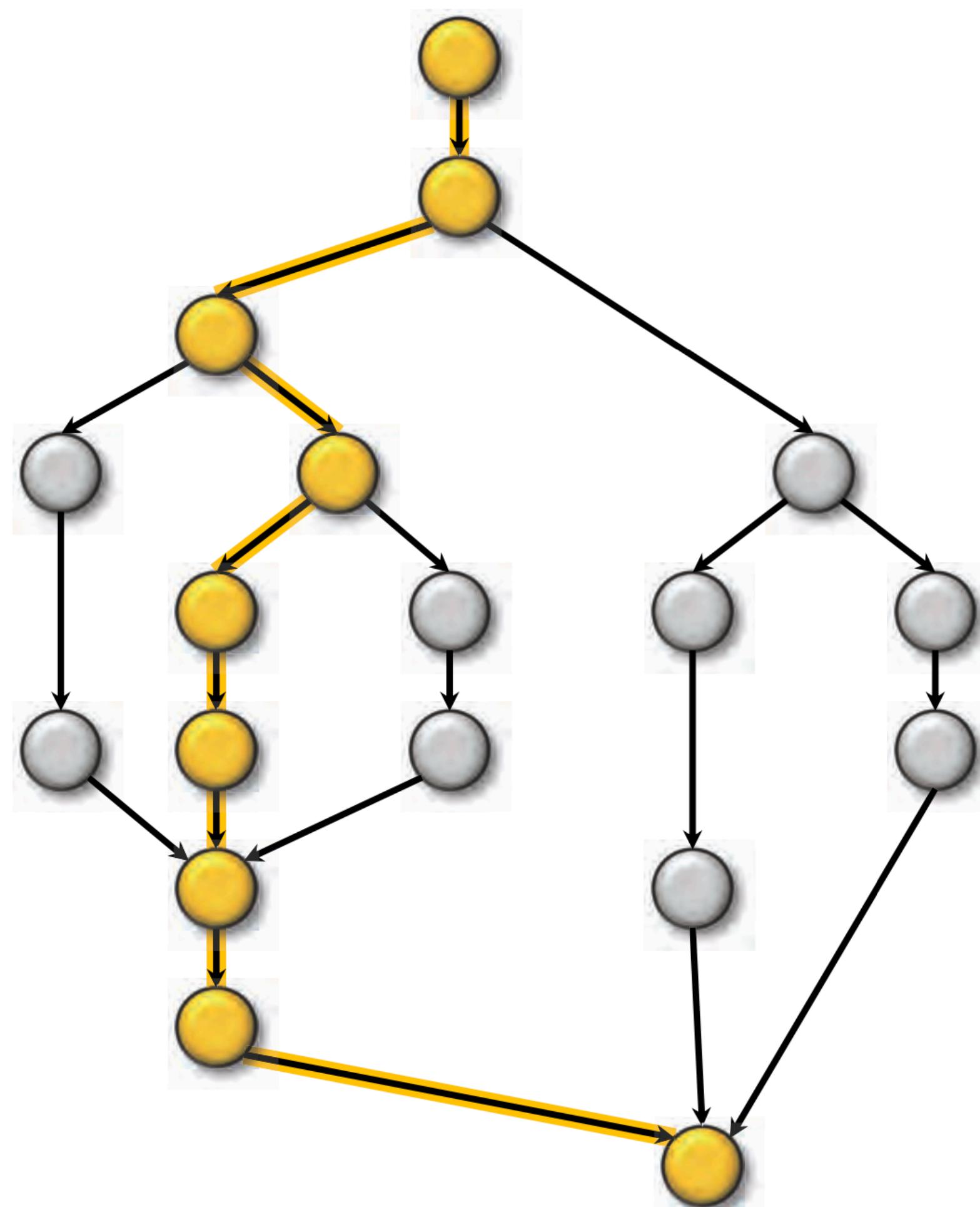
Time with an infinite number of processors

$$T_\infty = \text{span}^* \\ = 9$$

\* Also called **critical-path length** or **computational depth**

# Performance Measures

$T_p$  = execution time on P processors



$$T_1 = \text{work} = 18$$

$$T_\infty = \text{span}^* = 9$$

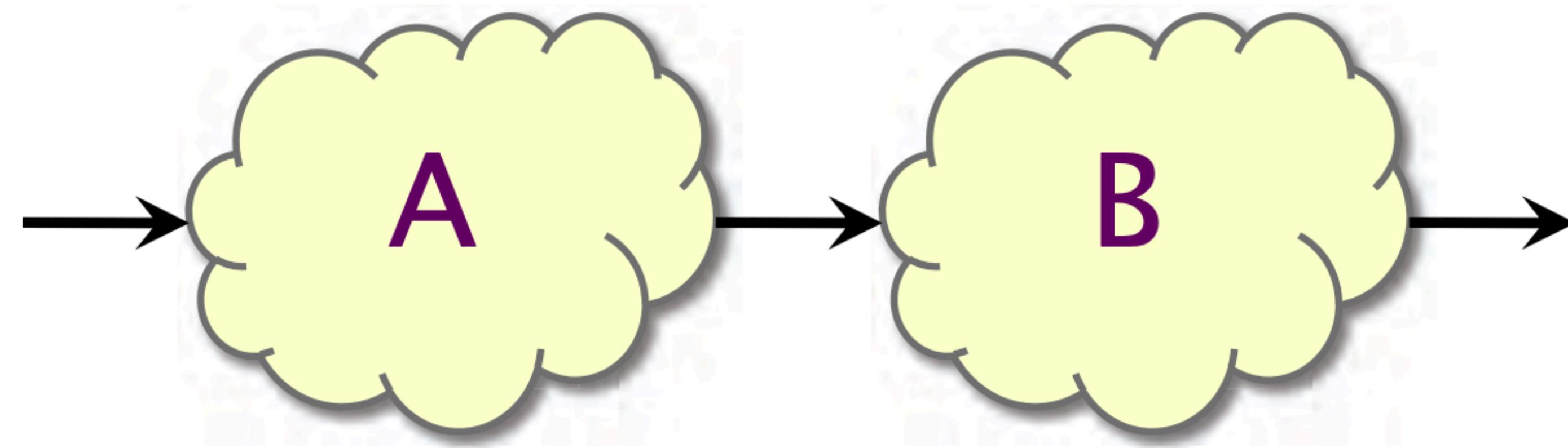
**Work Law**  
 $T_p \geq T_1/P$

**Span Law**  
 $T_p \geq T_\infty$

\* Also called **critical-path length** or **computational depth**

Time with an infinite number of processors

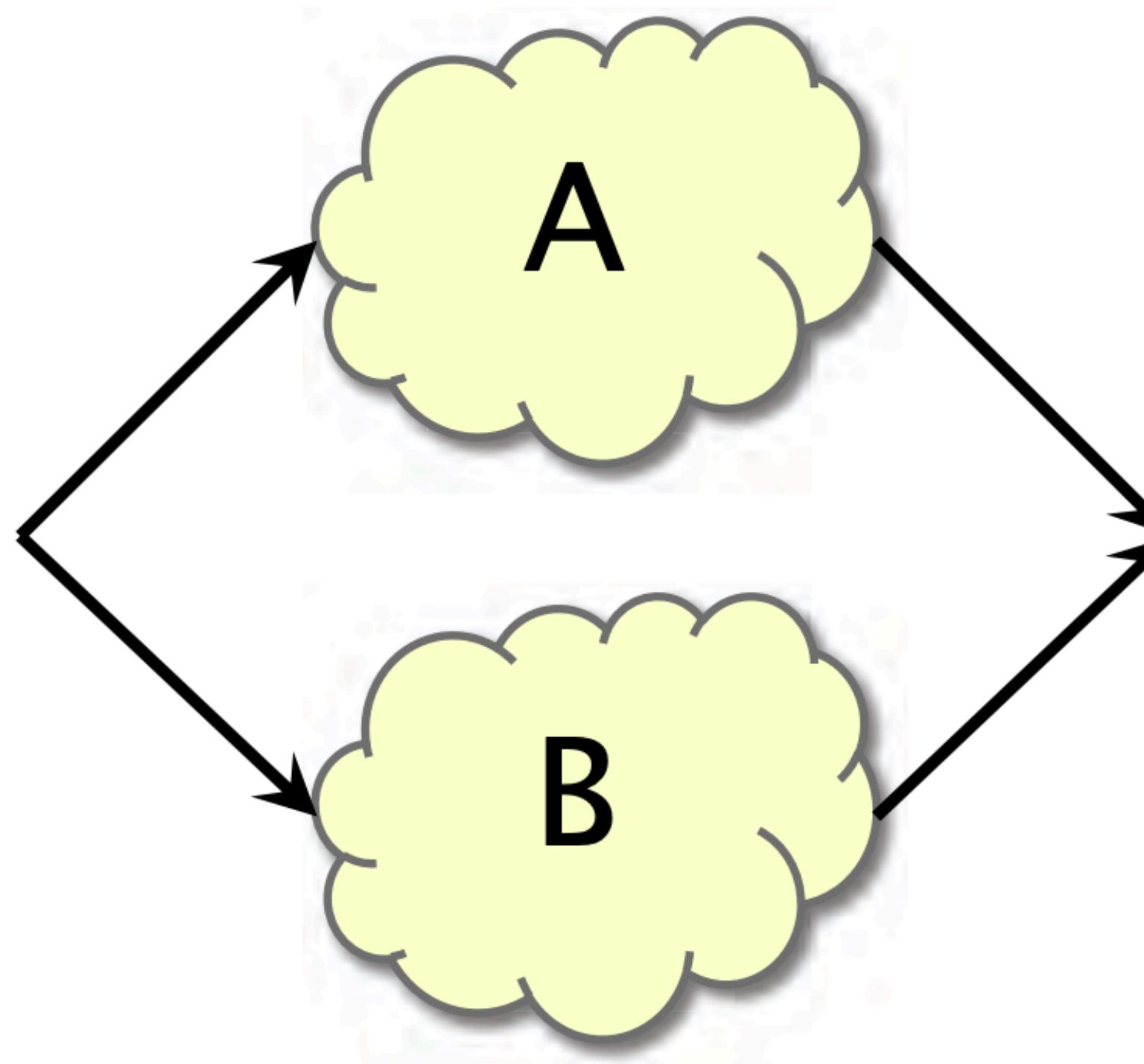
# Series Composition



*Work:*  $T_1(A \cup B) = T_1(A) + T_1(B)$

*Span:*  $T_\infty(A \cup B) = T_\infty(A) + T_\infty(B)$

# Parallel Composition



*Work:*  $T_1(A \cup B) = T_1(A) + T_1(B)$

*Span:*  $T_\infty(A \cup B) = \max\{T_\infty(A), T_\infty(B)\}$

# Speedup

Definition.  $T_1/T_P$  = speedup on  $P$  processors.

---

- If  $T_1/T_P < P$ , we have **sublinear speedup**.
  - If  $T_1/T_P = P$ , we have **(perfect) linear speedup**.
  - If  $T_1/T_P > P$ , we have **superlinear speedup**,  
which is not possible in this simple  
performance model, because of the **WORK LAW**  
 $T_P \geq T_1/P$ .
-

# Brent's Law

Analysis of parallel algorithms usually assumes that there is an **unbounded number of processors**.

Although the assumption is unrealistic, it is not a problem in a theoretical since, since any computation that can run in parallel on N processors can be **simulated on p<N** processors by letting each processor execute multiple units of work.

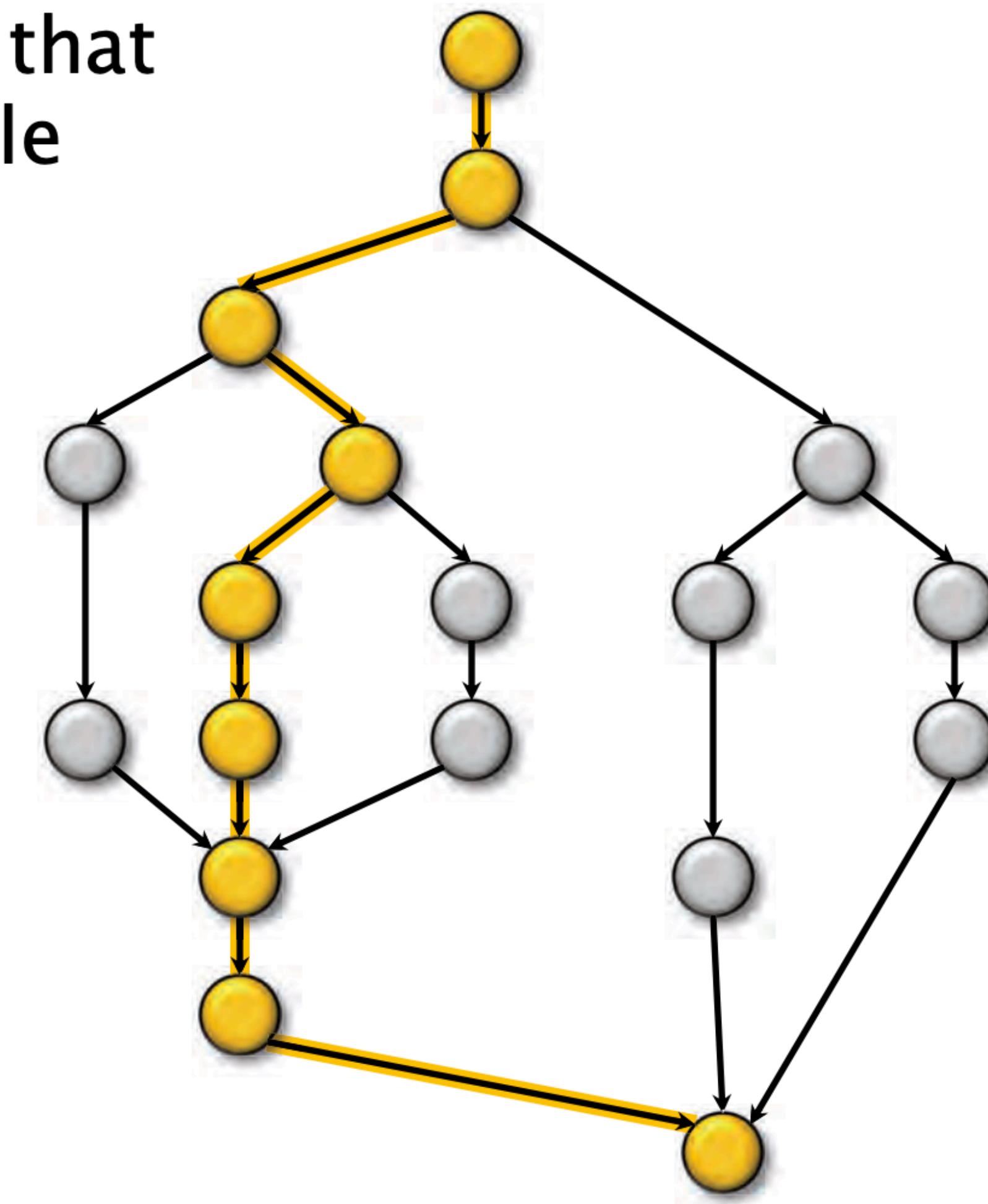
## Brent's Law

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty$$

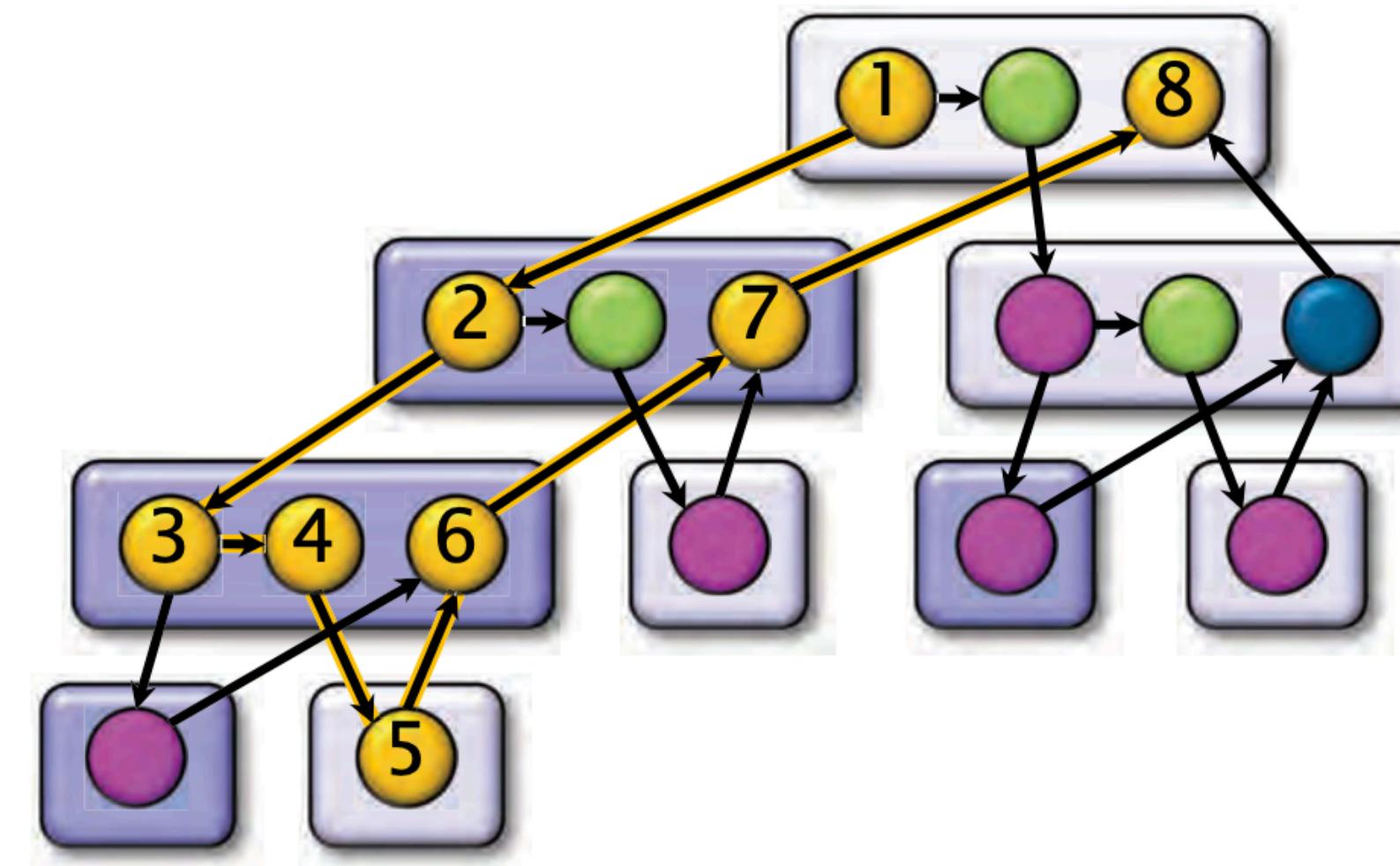
# Parallelism

Because the **SPAN LAW** dictates that  $T_p \geq T_\infty$ , the maximum possible speedup given  $T_1$  and  $T_\infty$  is

$$\begin{aligned} T_1/T_\infty &= \text{parallelism} \\ &= \text{the average amount of work per step along the span} \\ &= 18/9 \\ &= 2. \end{aligned}$$



# Example: fib(4)



Assume for simplicity  
that each strand in  
**fib(4)** takes unit  
time to execute.

*Work:*  $T_1 = 17$

*Span:*  $T_\infty = 8$

*Parallelism:*  $T_1/T_\infty = 2.125$

Using many more than 2 processors can  
yield only marginal performance gains.

# Example: Parallel Quicksort

```
static void quicksort(size_t *left, size_t* right) {  
    if (left == right) return;  
    size_t *p = partition(left, right); // run serially  
    parallel_spawn quicksort(left, p);  
    quicksort(p+1, right);  
    parallel_sync;  
}
```

Assuming partition has  
 $\Theta(n)$  work and span

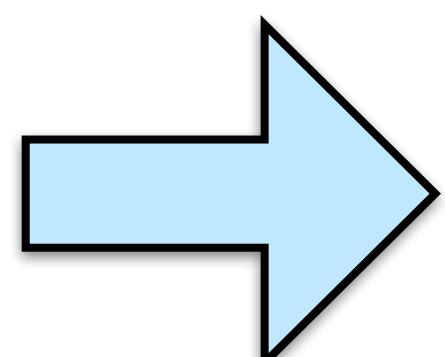
What is the expected work and span  
in the binary forking model?

# Example: Parallel Quicksort

```
static void quicksort(size_t *left, size_t* right) {  
    if (left == right) return;  
    size_t *p = partition(left, right); // run serially  
    parallel_spawn quicksort(left, p);  
    quicksort(p+1, right);  
    parallel_sync;  
}
```

Assuming partition has  
 $\Theta(n)$  work and span

Expected work =  $\Theta(n \lg n)$



Parallelism =  $\Theta(\lg n)$

Expected span =  $\Theta(n)$

# Interesting Practical Algorithms

Algorithm	Work	Span	Parallelism
Merge sort	$\Theta(n \lg n)$	$\Theta(\lg^3 n)$	$\Theta(n/\lg^2 n)$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\lg n)$	$\Theta(n^3/\lg n)$
Strassen	$\Theta(n^{\lg 7})$	$\Theta(\lg^2 n)$	$\Theta(n^{\lg 7}/\lg^2 n)$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \lg n)$	$\Theta(n^2/\lg n)$
Tableau construction	$\Theta(n^2)$	$\Theta(n^{\lg 3})$	$\Theta(n^{2-\lg 3})$
FFT	$\Theta(n \lg n)$	$\Theta(\lg^2 n)$	$\Theta(n/\lg n)$
Breadth-first search	$\Theta(E)$	$\Theta(\Delta \lg V)$	$\Theta(E/\Delta \lg V)$

# Divide-and-Conquer Recurrences

# The Master Method

The **Master Method** for solving divide-and-conquer recurrences applies to recurrences of the form\*

$$T(n) = aT(n/b) + f(n),$$

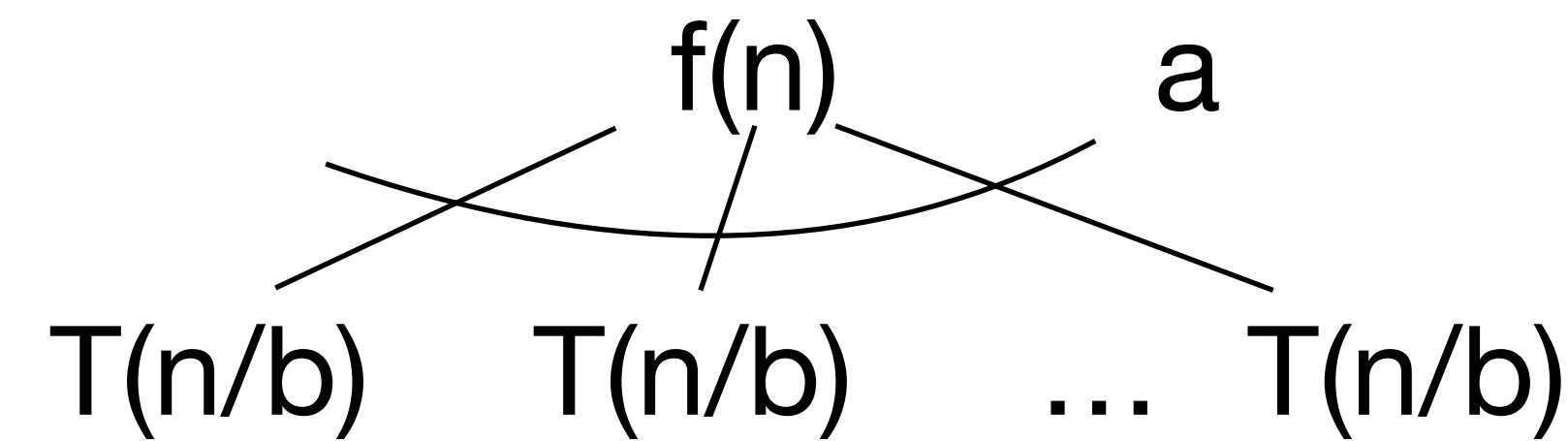
where  $a \geq 1, b > 1$ , and  $f$  is asymptotically positive.

\*The unstated base case is  $T(n) = \Theta(1)$  for sufficiently small  $n$ .

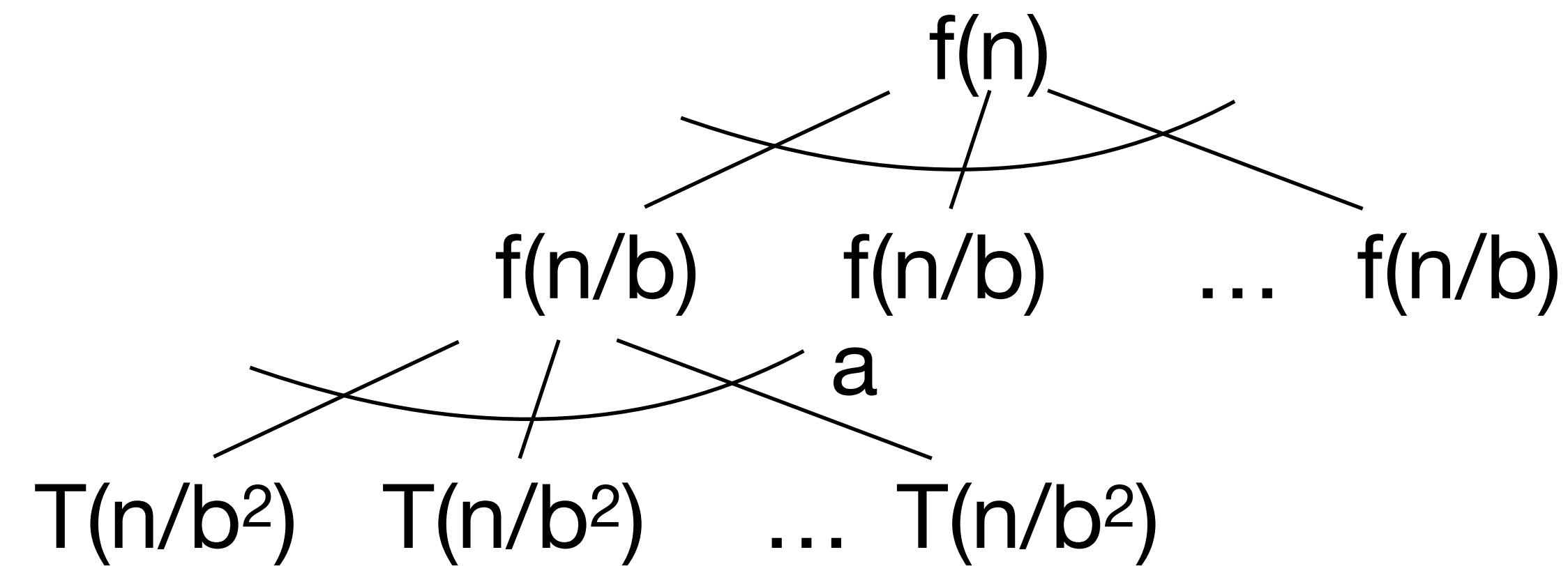
# Recursion Tree: $T(n) = aT(n/b) + f(n)$

$T(n)$

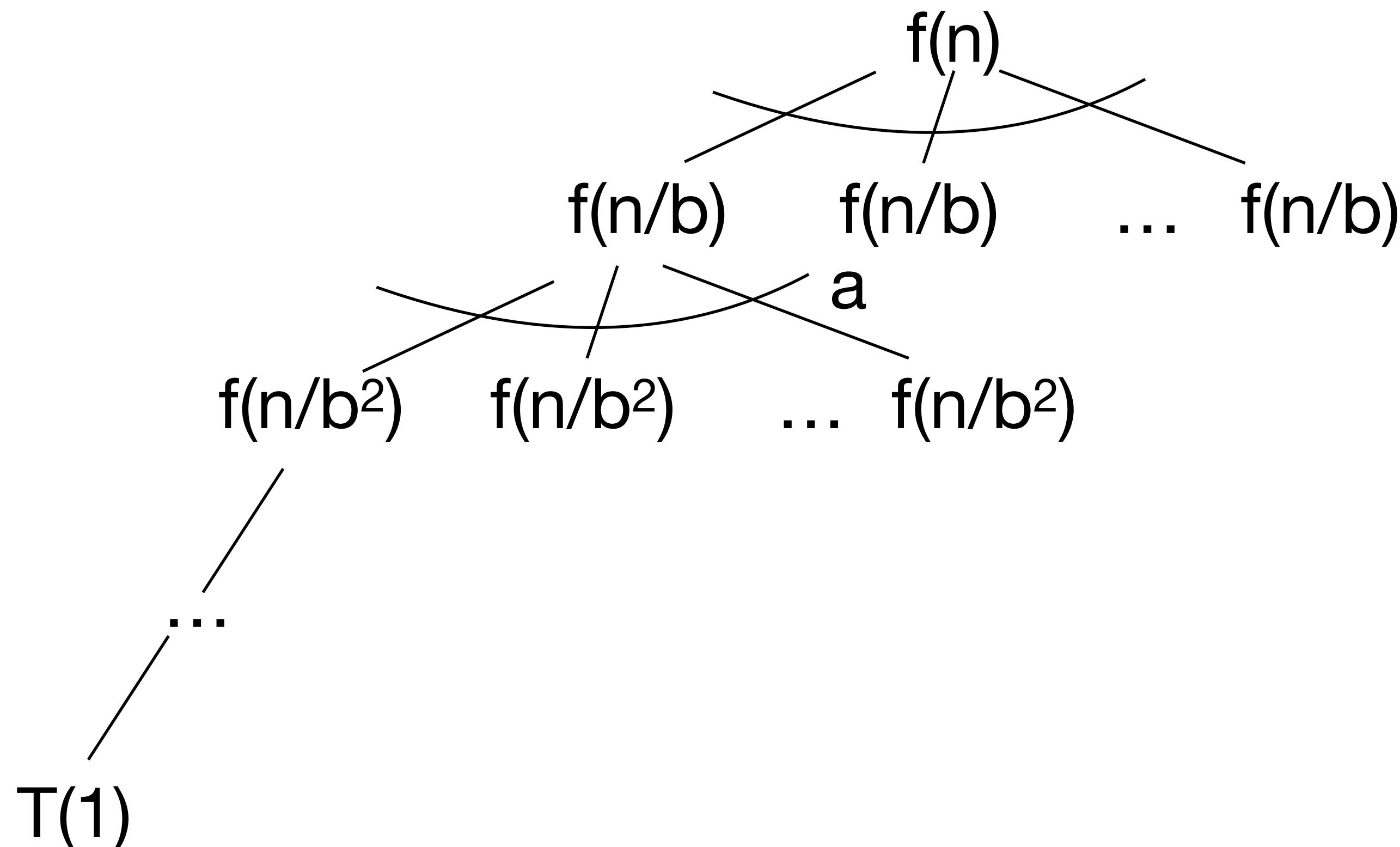
# Recursion Tree: $T(n) = aT(n/b) + f(n)$



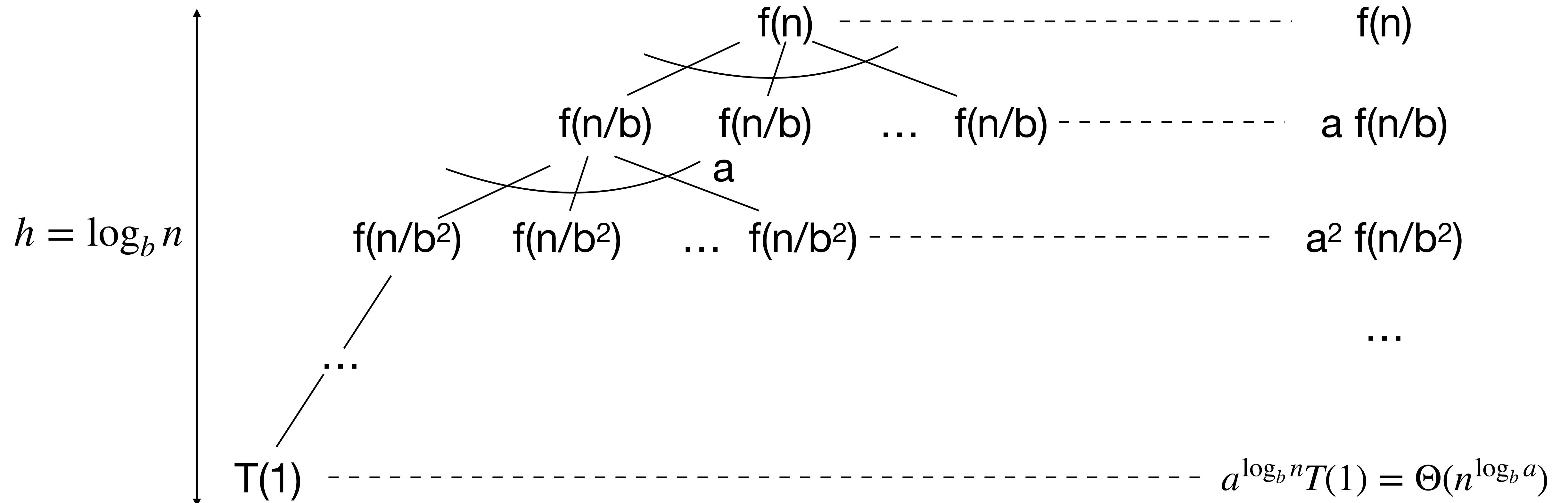
# Recursion Tree: $T(n) = aT(n/b) + f(n)$



# Recursion Tree: $T(n) = aT(n/b) + f(n)$

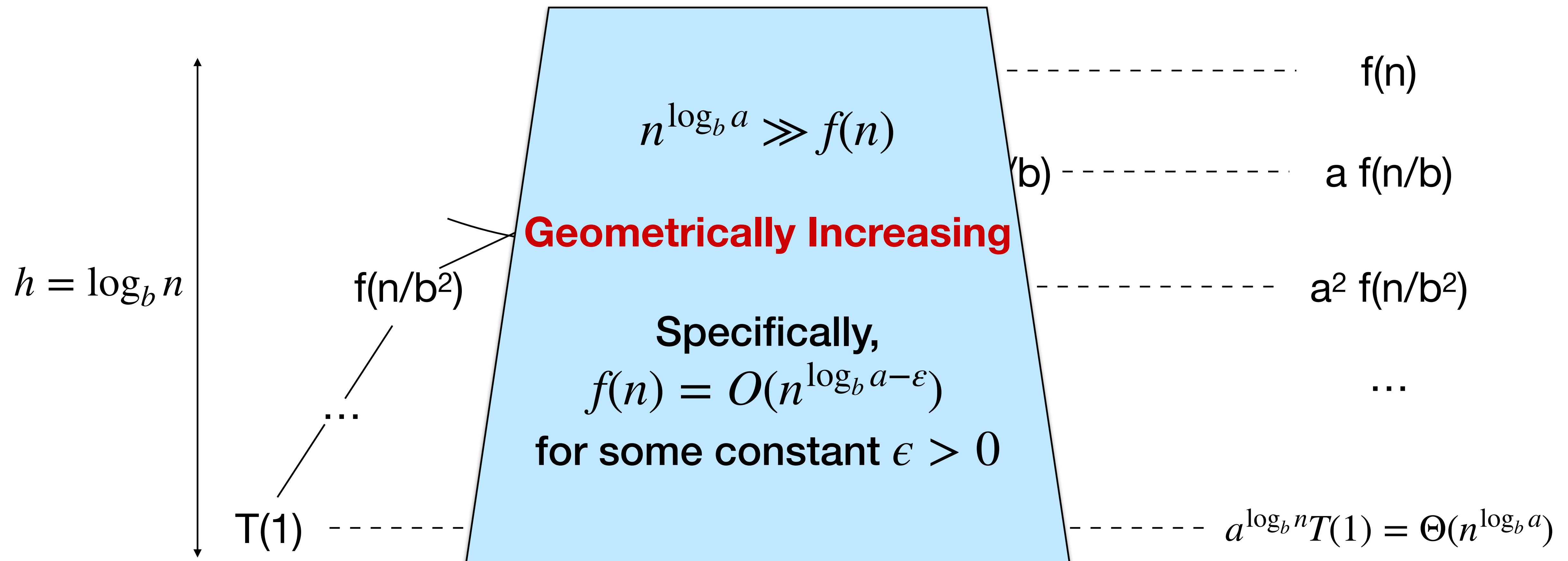


# Recursion Tree: $T(n) = aT(n/b) + f(n)$



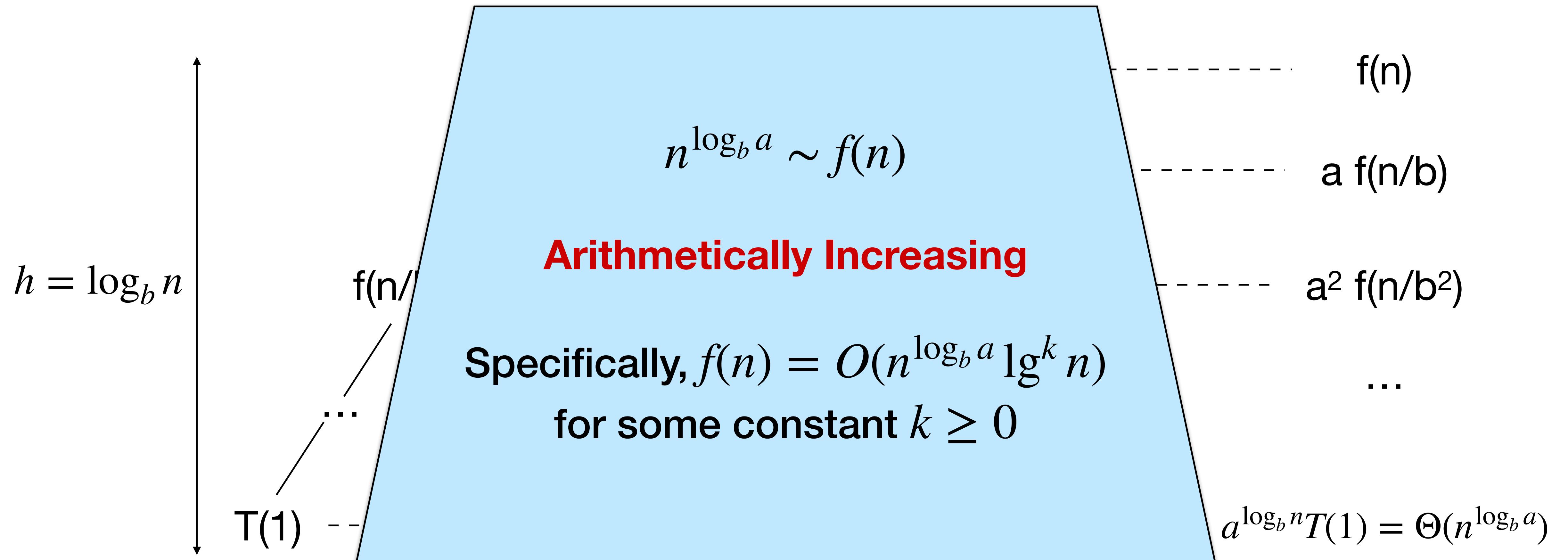
Idea: Compare  $n^{\log_b a}$  with  $f(n)$ .

# Master Method - Case 1



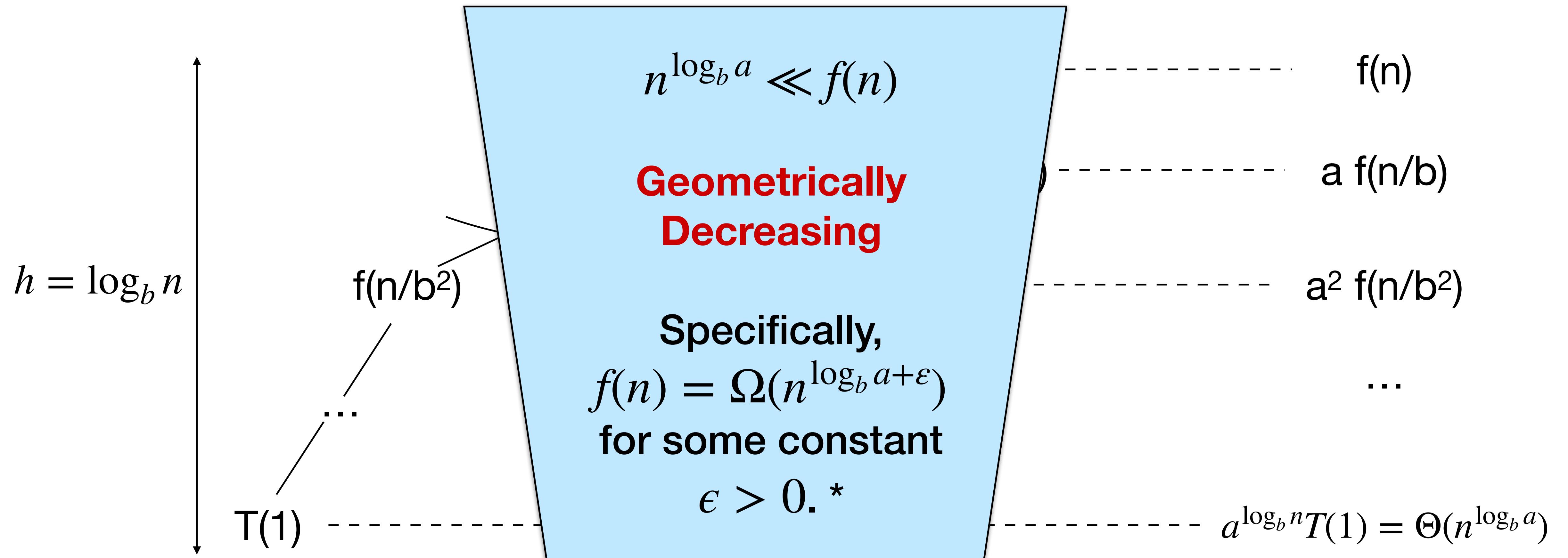
$$T(n) = \Theta(n^{\log_b a})$$

# Master Method - Case 2



$$T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$$

# Master Method - Case 3



$$T(n) = \Theta(f(n))$$

\*and  $f(n)$  satisfies the **regularity condition** that  $a f(n/b) \leq c f(n)$  for some constant  $c < 1.$

# Master-Method Cheat Sheet

Solve

$$T(n) = a T(n/b) + f(n) ,$$

where  $a \geq 1$  and  $b > 1$ .

CASE 1:  $f(n) = O(n^{\log_b a - \epsilon})$ , constant  $\epsilon > 0$   
 $\Rightarrow T(n) = \Theta(n^{\log_b a})$  .

CASE 2:  $f(n) = \Theta(n^{\log_b a} \lg^k n)$ , constant  $k \geq 0$   
 $\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$  .

CASE 3:  $f(n) = \Omega(n^{\log_b a + \epsilon})$ , constant  $\epsilon > 0$   
(and regularity condition)  
 $\Rightarrow T(n) = \Theta(f(n))$  .

# Master Method Quiz

- $T(n) = 4T(n/2) + n$   
 $n^{\log_b a} = n^2 \gg n \Rightarrow \text{CASE 1: } T(n) = \Theta(n^2).$
- $T(n) = 4T(n/2) + n^2$   
 $n^{\log_b a} = n^2 = n^2 \lg^0 n \Rightarrow \text{CASE 2: } T(n) = \Theta(n^2 \lg n).$
- $T(n) = 4T(n/2) + n^3$   
 $n^{\log_b a} = n^2 \ll n^3 \Rightarrow \text{CASE 3: } T(n) = \Theta(n^3).$
- $T(n) = 4T(n/2) + n^2/\lg n$   
Master method does not apply!  
Answer is  $T(n) = \Theta(n^2 \lg \lg n)$ . (Prove by substitution.)

More general (but more complicated)  
solution: Akra–Bazzi method.

# Parallel Loops

# Analysis of Parallel Loops

Assuming each iteration takes  $O(1)$  work

A parallel for loop with  $n$  iterations has **span  $O(\lg n)$**  in binary forking model - implemented with divide-and-conquer.

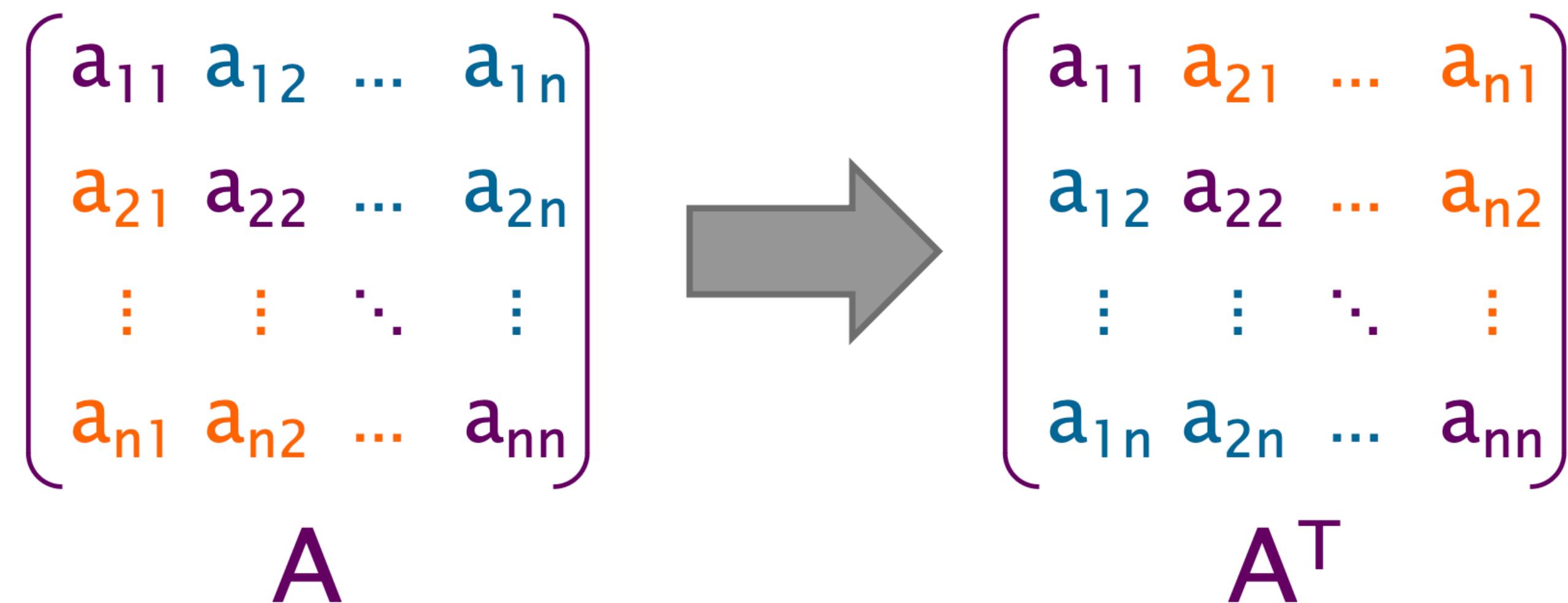
Span bounds in the binary forking model are at most **polylog factors** off of those in the PRAM model.

Classical  
parallelism  
model

```
parallel_for(int i = 0; i < n; ++i) {  
    foo();  
}
```

# Example: Matrix Transpose

Example: In-place matrix transpose



```
for(int i = 1; i < n; ++i) {
    for(int j = 0; j < i; ++j) {
        double temp = A[i][j];
        A[i][j] = A[j][i];
        A[j][i] = temp;
    }
}
```

# Example: Matrix Transpose

Example: In-place matrix transpose

```
parallel_for(int i = 1; i < n; ++i) {  
    for(int j = 0; j < i; ++j) {  
        double temp = A[i][j];  
        A[i][j] = A[j][i];  
        A[j][i] = temp;  
    }  
}
```

Assuming loop iterations execute in parallel

What is the work and span in the binary forking model?

# Example: Matrix Transpose

Example: In-place matrix transpose

```
parallel_for(int i = 1; i < n; ++i) {  
    for(int j = 0; j < i; ++j) {  
        double temp = A[i][j];  
        A[i][j] = A[j][i];  
        A[j][i] = temp;  
    }  
}
```

Assuming loop iterations execute in parallel

**Work:**  $T_1(n) = \Theta(n^2)$

**Span:**  $T_\infty(n) = \Theta(n + \lg n) = \Theta(n)$

**Parallelism:**  $T_1(n)/T_\infty(n) = \Theta(n)$

# Example: Matrix Transpose

Example: In-place matrix transpose

```
parallel_for(int i = 1; i < n; ++i) {  
    parallel_for(int j = 0; j < i; ++j) {  
        double temp = A[i][j];  
        A[i][j] = A[j][i];  
        A[j][i] = temp;  
    }  
}
```

Assuming loop iterations execute in parallel

What is the work and span in the binary forking model?

# Example: Matrix Transpose

Example: In-place matrix transpose

```
parallel_for(int i = 1; i < n; ++i) {  
    parallel_for(int j = 0; j < i; ++j) {  
        double temp = A[i][j];  
        A[i][j] = A[j][i];  
        A[j][i] = temp;  
    }  
}
```

Assuming loop iterations execute in parallel

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

Span:  $T_\infty(n) = \Theta(\lg n)$

Parallelism:  $T_1(n)/T_\infty(n) = \Theta(n^2/\lg n)$

# Quiz on Parallel Loops

```
parallel_for(int i = 0; i < n; i += 32) {  
    for(int j = i; j < min(i+32, n); ++j) {  
        A[j] += B[j];  
    }  
}
```

Work:  $T_1 = \Theta(n)$

Span:  $T_\infty = \Theta(\lg n)$

Parallelism:  $T_1/T_\infty = \Theta(n/\lg n)$

```
parallel_for(int i = 0; i < n; i += n/P) {  
    for(int j = i; j < min(i+n/P, n); ++j) {  
        A[j] += B[j];  
    }  
}
```

Work:  $T_1 = \Theta(n)$

Span:  $T_\infty = \Theta(n/P)$

Parallelism:  $T_1/T_\infty = \Theta(P)$

# Quiz on Parallel Loops

```
parallel_for(int i = 0; i < n; i += 32) {  
    for(int j = i; j < min(i+32, n); ++j) {  
        A[j] += B[j];  
    }  
}
```

Work:  $T_1 = \Theta(n)$

Span:  $T_\infty = \Theta(\lg n)$

Parallelism:  $T_1/T_\infty = \Theta(n/\lg n)$

```
parallel_for(int i = 0; i < n; i += n/P) {  
    for(int j = i; j < min(i+n/P, n); ++j) {  
        A[j] += B[j];  
    }  
}
```

Work:  $T_1 = \Theta(n)$

Span:  $T_\infty = \Theta(n/P)$

Parallelism:  $T_1/T_\infty = \Theta(P)$

Too low! We  
want

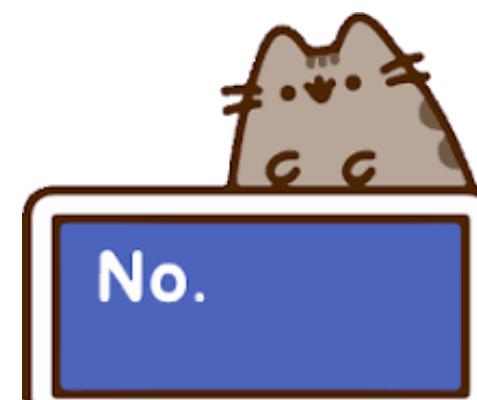
$T_1/T_\infty \gg P$

# Three Performance Tips

1. Minimize the span to maximize parallelism. Try to generate 10 times more parallelism than processors for near-perfect linear speedup.
2. If you have plenty of parallelism, try to trade some of it off to reduce work overhead.
3. Use divide-and-conquer recursion or parallel loops rather than spawning one small thing after another.



```
parallel_for(int i = 0; i < n; ++i) {  
    foo(i);  
}
```



```
for(int i = 0; i < n; ++i) {  
    spawn foo(i);  
}  
sync;
```

# And Three More

4. Ensure that `work/#spawns` is sufficiently large.
  - Coarsen by using function calls and **inlining** near the leaves of recursion, rather than spawning.
5. Parallelize **outer loops**, as opposed to inner loops, if you're forced to make a choice.
6. Watch out for **scheduling overheads**.



```
parallel_for(int i = 0; i < 2; ++i) {  
    for (int j = 0; j < n; ++j) {  
        f(i, j);  
    }  
}
```



```
for (int j = 0; j < n; ++j) {  
    parallel_for (int i = 0; i < 2; ++i) {  
        f(i, j);  
    }  
}
```

# **Matrix Multiplication**

# Square-Matrix Multiplication

$$\begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

**C**                    **A**                    **B**

$$c_{ij} = \sum_{k=1}^n a_{ik} b_{kj}$$

Assume for simplicity that  $n = 2^k$ .

# Parallelizing Matrix Multiply

```
parallel_for(int i = 0; i < n; ++i) {  
    parallel_for(int j = 0; j < n; ++j) {  
        for(int k = 0; k < n; ++k) {  
            C[i][j] += A[i][k] * B[k][j];  
        }  
    }  
}
```

Work:  $T_1(n) = \Theta(n^3)$

Span:  $T_\infty(n) = \Theta(n)$

Parallelism:  $T_1(n)/T_\infty(n) = \Theta(n^2)$

For  $1000 \times 1000$  matrices, parallelism  $\sim 10^6$

# Recursive Matrix Multiplication

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \cdot \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

$$= \begin{bmatrix} A_{00}B_{00} & A_{00}B_{01} \\ A_{10}B_{00} & A_{10}B_{01} \end{bmatrix} + \begin{bmatrix} A_{01}B_{10} & A_{01}B_{11} \\ A_{11}B_{10} & A_{11}B_{11} \end{bmatrix}$$

8 multiplications of  $n/2 \times n/2$  matrices.

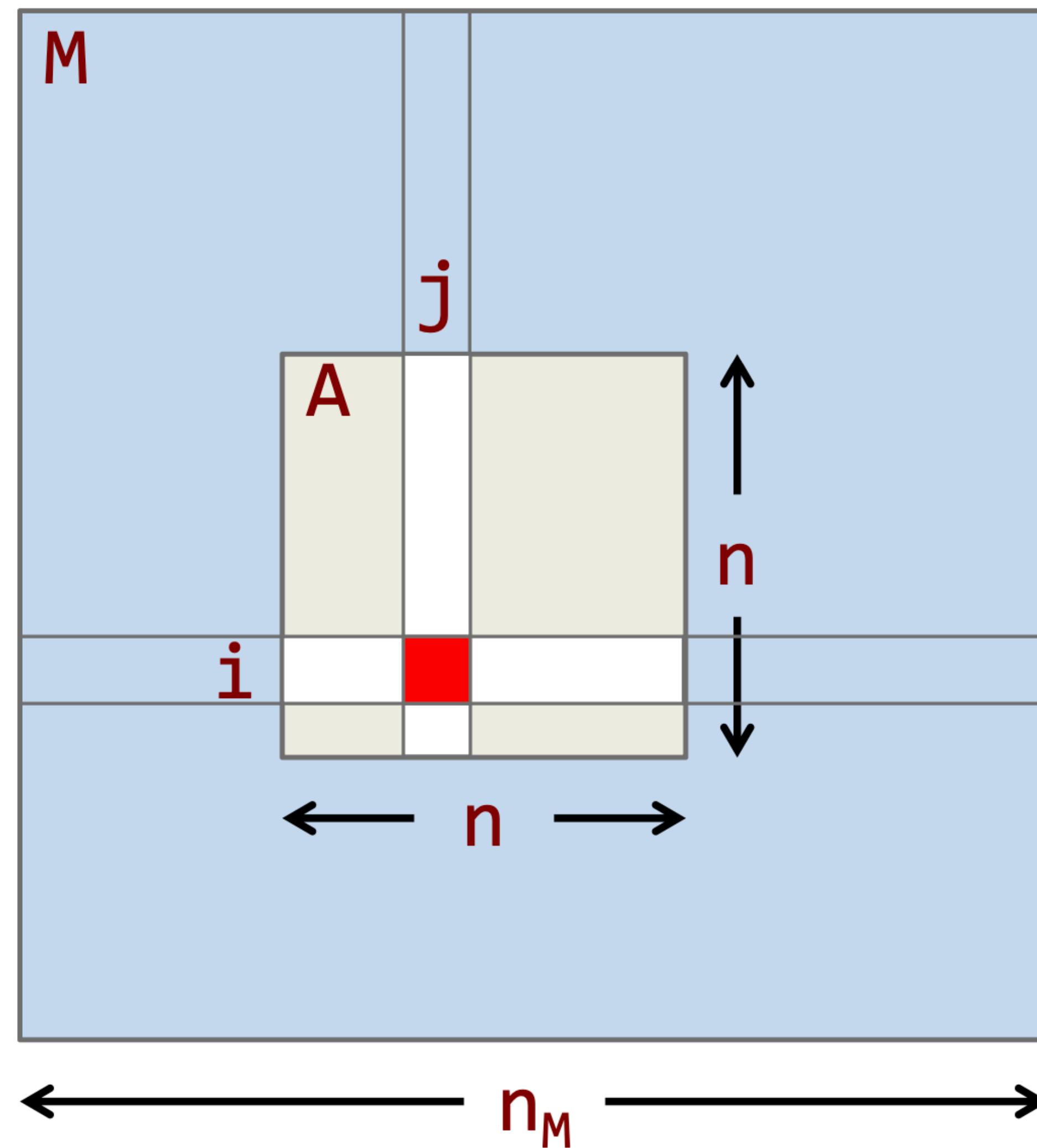
1 addition of  $n \times n$  matrices.

# Recall: Row-Major Layout

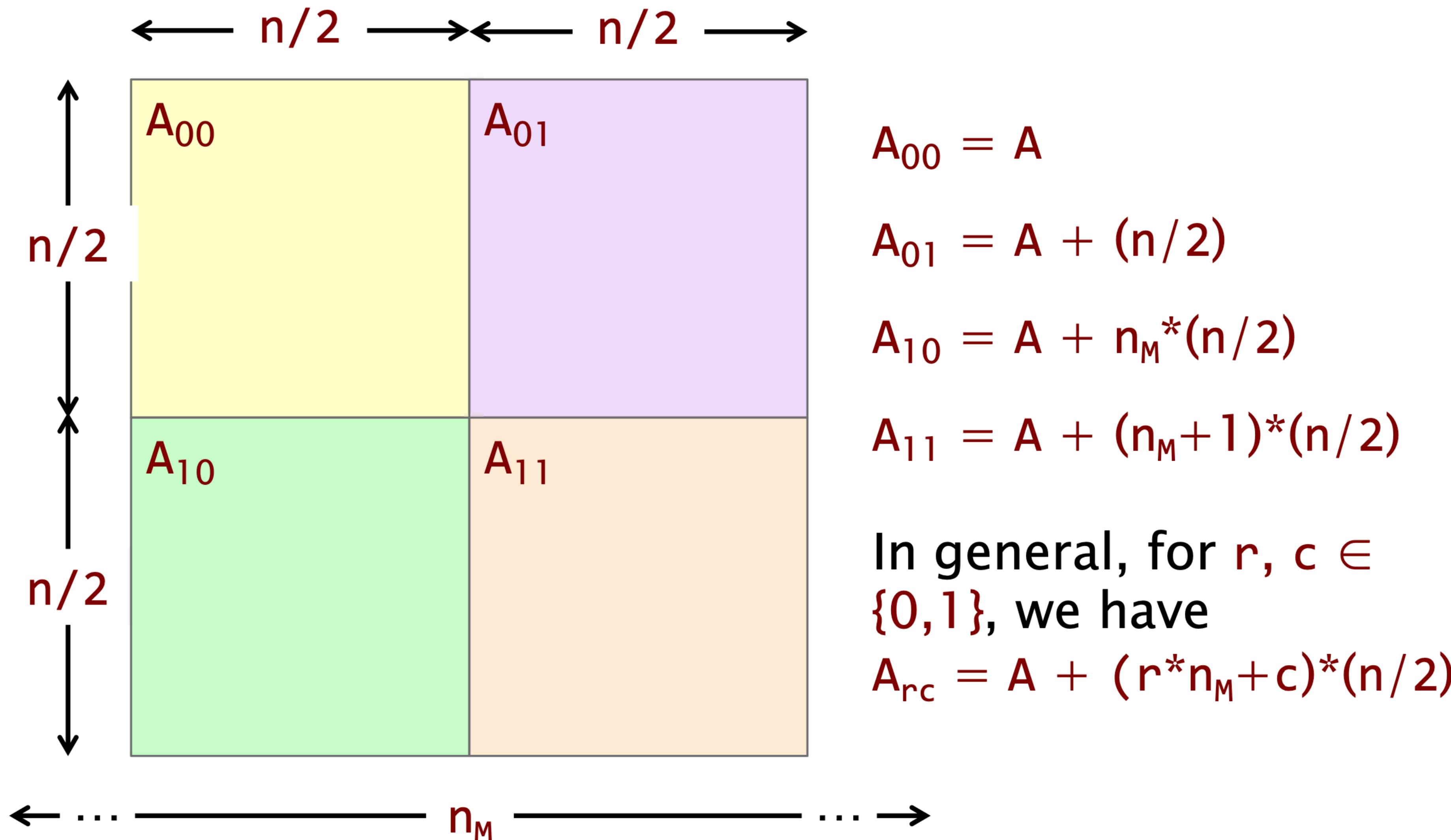
## Row-major layout

If  $A$  is an  $n \times n$  submatrix of an underlying matrix  $M$  with row size  $n_M$ , then the  $(i, j)$  element of  $A$  is  $A[n_M * i + j]$ .

**Note:** The dimension  $n$  does not enter into the calculation, although it does matter for bounds checking of  $i$  and  $j$ .



# Recall: Divide-and-Conquer Matrices



# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}

```

# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}

```

The compiler can assume that the input matrices are not aliased

# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}

```

Row size of the underlying matrices

# D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B);
    } else {
        double *D = malloc(n * n * sizeof(double));
        assert(D != NULL);

#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2)
        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}
```

The three input  
matrices are  $n \times n$

# D&C Matrix Multiplication

```

void mm_dac(double *res,
            double *C, double *D, double *A, double *B, int n)
{
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D;
        assert(D != NULL);
#define n_D n
#define X(M, i, j) ((M) + (i) * (n/2) + (j))
        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}

```

Coarsen the leaves of the recursion to lower the overhead

The function adds the matrix product  $A^*B$  to  $C$

# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int THRESHOLD) {
    if (n_C <= THRESHOLD)
        mm_base(C, n_C, A, n_A, B, n_B);
    } else {
        double *D = malloc(n_C * n_C * sizeof(*D));
        assert(D != NULL);

#define n_D n_C
#define X(M, r, c) X((M), (r), (c))
        parallel_spawn mm_dac(X(D, 0, 0), n_D, X(A, 0, 0), n_A, X(B, 0, 0), n_B, n/2);
        parallel_spawn mm_dac(X(D, 0, 1), n_D, X(A, 0, 0), n_A, X(B, 0, 1), n_B, n/2);
        parallel_spawn mm_dac(X(D, 1, 0), n_D, X(A, 1, 0), n_A, X(B, 0, 0), n_B, n/2);
        parallel_spawn mm_dac(X(D, 1, 1), n_D, X(A, 1, 0), n_A, X(B, 0, 1), n_B, n/2);
        parallel_spawn mm_dac(X(D, 0, 0), n_D, X(A, 0, 1), n_A, X(B, 1, 0), n_B, n/2);
        parallel_spawn mm_dac(X(D, 0, 1), n_D, X(A, 0, 1), n_A, X(B, 1, 1), n_B, n/2);
        parallel_spawn mm_dac(X(D, 1, 0), n_D, X(A, 1, 1), n_A, X(B, 1, 0), n_B, n/2);
        parallel_spawn mm_dac(X(D, 1, 1), n_D, X(A, 1, 1), n_A, X(B, 1, 1), n_B, n/2);
        mm_dac(X(D, 0, 0), n_D, X(A, 0, 0), n_A, X(B, 0, 0), n_B, n/2);
        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}

```

**Allocate a temporary  $n \times n$  array D**

**The temporary array D has underlying row size n**

# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D;
        assert(D != NULL);
        parallel_spawn mm_dac(X(C, 0, 0), n_D, X(A, 0, 0), n_A, X(B, 0, 0), n_B, n/2);
        parallel_spawn mm_dac(X(C, 0, 1), n_D, X(A, 0, 1), n_A, X(B, 0, 1), n_B, n/2);
        parallel_spawn mm_dac(X(C, 1, 0), n_D, X(A, 1, 0), n_A, X(B, 0, 0), n_B, n/2);
        parallel_spawn mm_dac(X(C, 1, 1), n_D, X(A, 1, 1), n_A, X(B, 0, 1), n_B, n/2);
        parallel_spawn mm_dac(X(D, 0, 0), n_D, X(A, 0, 0), n_A, X(B, 0, 0), n_B, n/2);
        parallel_spawn mm_dac(X(D, 0, 1), n_D, X(A, 0, 1), n_A, X(B, 0, 1), n_B, n/2);
        parallel_spawn mm_dac(X(D, 1, 0), n_D, X(A, 1, 0), n_A, X(B, 1, 0), n_B, n/2);
        mm_dac(X(D, 1, 1), n_D, X(A, 1, 1), n_A, X(B, 1, 1), n_B, n/2);
        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}

```

**Macro to compute submatrix indices**

**The C preprocessor's token-pasting operator**

# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2)
        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        parallel_sync;
        m_add(C, n_C, D, n_
              free(D);
    }
}

```

Perform the 8 multiplications of  $(n/2) \times (n/2)$  submatrices recursively in parallel

Wait for all spawned subcomputations to finish

# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);

#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2)

        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);

        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}

```

Add the temporary matrix D into output matrix C

# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);

#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2)

        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);

        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}

```

Add the temporary matrix D into output matrix C

```

void m_add (double *restrict C, int n_C,
            double *restrict D, int n_D,
            int n) {
    parallel_for(int i = 0; i < n; ++i) {
        parallel_for(int j = 0; j < n; ++j) {
            C[i*n_C + j] += D[i*n_D + j];
        }
    }
}

```

# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
        parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
        parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
        mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
        parallel_sync;
        m_add(C, n_C, D, n_D, n);
        free(D);
    }
}

```

Clean up and return



# Analysis of Matrix Addition

```
void m_add (double *restrict C, int n_C,
            double *restrict D, int n_D,
            int n) {
    parallel_for(int i = 0; i < n; ++i) {
        parallel_for(int j = 0; j < n; ++j) {
            C[i*n_C + j] += D[i*n_D + j];
        }
    }
}
```

Work:  $A_1(n) = \Theta(n^2)$

Span:  $A_\infty(n) = \Theta(\lg n)$

BF model

# Work of Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    // base case up here
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,0,0), n_B, n/2);
    mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
}
```

Case 1 of Master Method

$$n^{\log_b a} = n^{\log_2 8} = n^3$$

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

$$\begin{aligned} \text{Work: } M_1(n) &= 8M_1(n/2) + A_1(n) + \Theta(1) \\ &= 8M_1(n/2) + \Theta(n^2) \\ &= \Theta(n^3) \end{aligned}$$

# Span of Matrix Multiplication

Maximum over spawns

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    base case up here
    parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    parallel_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    parallel_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
    mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    parallel_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
}
```

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_2 1} = 1$$

$$f(n) = \Theta(n^{\log_b a} \lg^1 n)$$

$$\begin{aligned}\text{Span: } M_\infty(n) &= M_\infty(n/2) + A_\infty(n) + \Theta(1) \\ &= M_\infty(n/2) + \Theta(\lg n) \\ &= \Theta(\lg^2 n)\end{aligned}$$

# Parallelism of Matrix Multiply

Work:  $M_1(n) = \Theta(n^3)$

Span:  $M_\infty(n) = \Theta(\lg^2 n)$

BF model

Parallelism:  $M_1(n)/M_\infty(n) = \Theta(n^3/\lg^2 n)$

For  $1000 \times 1000$  matrices, parallelism  $\sim (10^3)^3/10^2 = 10^7$

# D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
        double *D = malloc(n * n * sizeof(*D));
        assert(D != NULL);
#define n_D n
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2)
        parallel_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_
        parallel_
        parallel_
        mm_dac(X(
parallel_
parallel_
parallel_
m_add(C,
free(D);
}
}

```

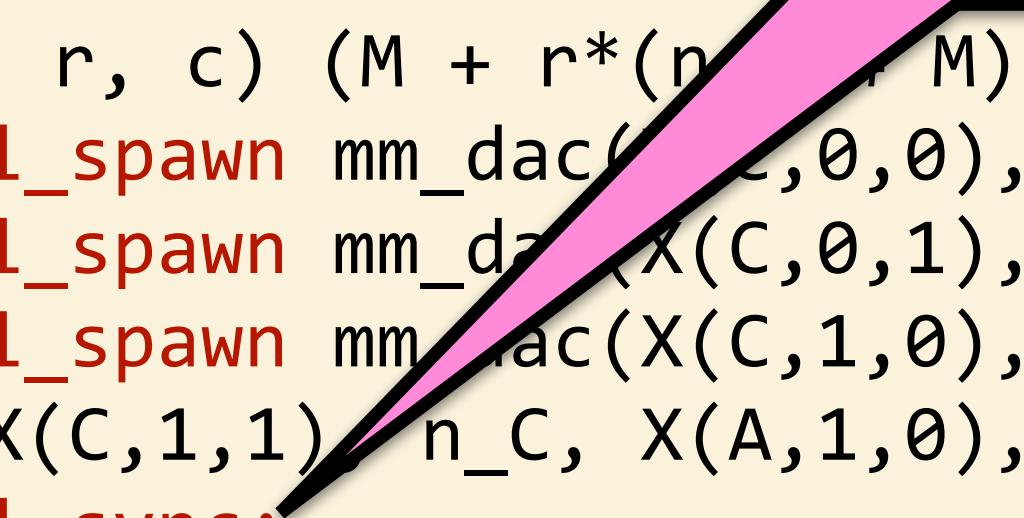
Idea: Since minimizing storage tends to yield higher performance, **trade some of the ample parallelism for less storage.**

# No-Temp D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B, int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n);
    } else {
#define X(M, r, c) ((M + r*(n - M) + c) * (n/2))
        parallel_spawn mm_dac(C, 0, 0), n_C, X(A, 0, 0), n_A, X(B, 0, 0), n_B, n/2;
        parallel_spawn mm_dac(C, 0, 1), n_C, X(A, 0, 0), n_A, X(B, 0, 1), n_B, n/2;
        parallel_spawn mm_dac(C, 1, 0), n_C, X(A, 1, 0), n_A, X(B, 0, 0), n_B, n/2;
        mm_dac(C, 1, 1), n_C, X(A, 1, 0), n_A, X(B, 0, 1), n_B, n/2;
        parallel_sync;

        parallel_spawn mm_dac(X(C, 0, 0), n_D, X(A, 0, 1), n_A, X(B, 1, 0), n_B, n/2);
        parallel_spawn mm_dac(X(C, 0, 1), n_D, X(A, 0, 1), n_A, X(B, 1, 1), n_B, n/2);
        parallel_spawn mm_dac(X(C, 1, 0), n_D, X(A, 1, 1), n_A, X(B, 1, 0), n_B, n/2);
        mm_dac(X(C, 1, 1), n_D, X(A, 1, 1), n_A, X(B, 1, 1), n_B, n/2);
        parallel_sync;
    }
}
```

Do 4 subproblems, sync,  
then do 4 more  
subproblems



# No-Temp D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
        parallel_spawn mm_dac(X(C,0,0), n_D, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_D, X(A,0,1), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_D, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_sync;
    }
}

```

**Reuse C  
without racing**

Saves space, but at what expense?

# No-Temp D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
        parallel_spawn mm_dac(X(C,0,0), n_D, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_D, X(A,0,1), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_D, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_sync;
    }
}

```

Reuse C without racing

A, X(B,0,0), n\_B, n/2);  
A, X(B,0,1), n\_B, n/2);  
A, X(B,0,0), n\_B, n/2);  
B, n/2);

Case 1 of Master Method  
 $n^{\log_b a} = n^{\log_2 8} = n^3$   
 $f(n) = \Theta(1)$

Work:  $M_1(n) = 8M_1(n/2) + \Theta(1) = \Theta(n^3)$

# No-Temp D&C Matrix Multiplication

```

void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n) {
    if (n <= THRESHOLD) {
        mm_base(C, n_C, A, n_A, B, n_B, n);
    } else {
#define X(M, r, c) (M + r*(n_ ## M) + c) * (n/2))
        parallel_spawn mm_dac(X(C,0,0), n_D, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
        parallel_spawn mm_dac(X(C,0,1), n_D, X(A,0,1), n_A, X(B,0,1), n_B, n/2);
        parallel_spawn mm_dac(X(C,1,0), n_D, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
        mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
        parallel_sync;
    }
}

```

Reuse C  
without racing

Case 1 of Master Method  
 $n^{\log_b a} = n^{\log_2 2} = n$   
 $f(n) = \Theta(1)$

$$\text{Span: } M_\infty(n) = 2M_\infty(n/2) + \Theta(1) = \Theta(n)$$

# Parallelism of No-Temp Multiply

Work:  $M_1(n) = \Theta(n^3)$

Span:  $M_\infty(n) = \Theta(n)$

Higher than  
with temp

Parallelism:  $M_1(n)/M_\infty(n) = \Theta(n^2)$

For  $1000 \times 1000$  matrices, parallelism  $\sim (10^3)^2 = 10^6$

No-temp version is faster in practice

# Merge Sort

# Merging Two Sorted Arrays

```
void merge(int *C, int *A, int na, int *B, int nb) {  
    while (na > 0 && nb > 0) {  
        if (*A <= *B) {  
            *C++ = *A++, na--;  
        } else {  
            *C++ = *B++, nb--;  
        }  
    }  
    while (na > 0) {  
        *C++ = *A++, na--;  
    }  
    while (nb > 0) {  
        *C++ = *B++, nb--;  
    }  
}
```

Work to merge n elements =  $\Theta(n)$

# Merge Sort

```
void merge_sort(int *B, int *A, int n) {
    if (n == 1) {
        B[0] = A[0];
    } else {
        int C[n];
        parallel_spawn merge_sort(C, A, n/2);
        merge_sort(C+n/2, A+n/2, n-n/2);
        parallel_sync;
        merge(B, C, n/2, C+n/2, n-n/2);
    }
}
```

$$\begin{aligned}\text{Work: } T_1(n) &= 2T_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n)\end{aligned}$$

# Work of Merge Sort

```
void merge_sort(int *B, int *A, int n) {
    if (n == 1) {
        B[0] = A[0];
    } else {
        int C[n];
        parallel_spawn merge_sort(C, A, n/2);
        merge_sort(C+n/2, A+n/2, n-n/2);
        parallel_sync;
        merge(B, C, n/2, C+n/2, n-n/2);
    }
}
```

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_2 2} = n$$

$$f(n) = \Theta(n^{\log_b a} \lg^0 n)$$

$$\begin{aligned} \text{Work: } T_1(n) &= 2T_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

# Span of Merge Sort

```
void merge_sort(int *B, int *A, int n) {
    if (n == 1) {
        B[0] = A[0];
    } else {
        int C[n];
        parallel_spawn merge_sort(C, A, n/2);
        merge_sort(C+n/2, A+n/2, n-n/2);
        parallel_sync;
        merge(B, C, n/2, C+n/2, n-n/2);
    }
}
```

Case 3 of Master Method

$$n^{\log_b a} = n^{\log_2 1} = 1$$

$$f(n) = \Theta(n)$$

$$\begin{aligned}\text{Span: } T_\infty(n) &= T_\infty(n/2) + \Theta(n) \\ &= \Theta(n)\end{aligned}$$

# Parallelism of Merge Sort

Work:  $T_1(n) = \Theta(n \lg n)$

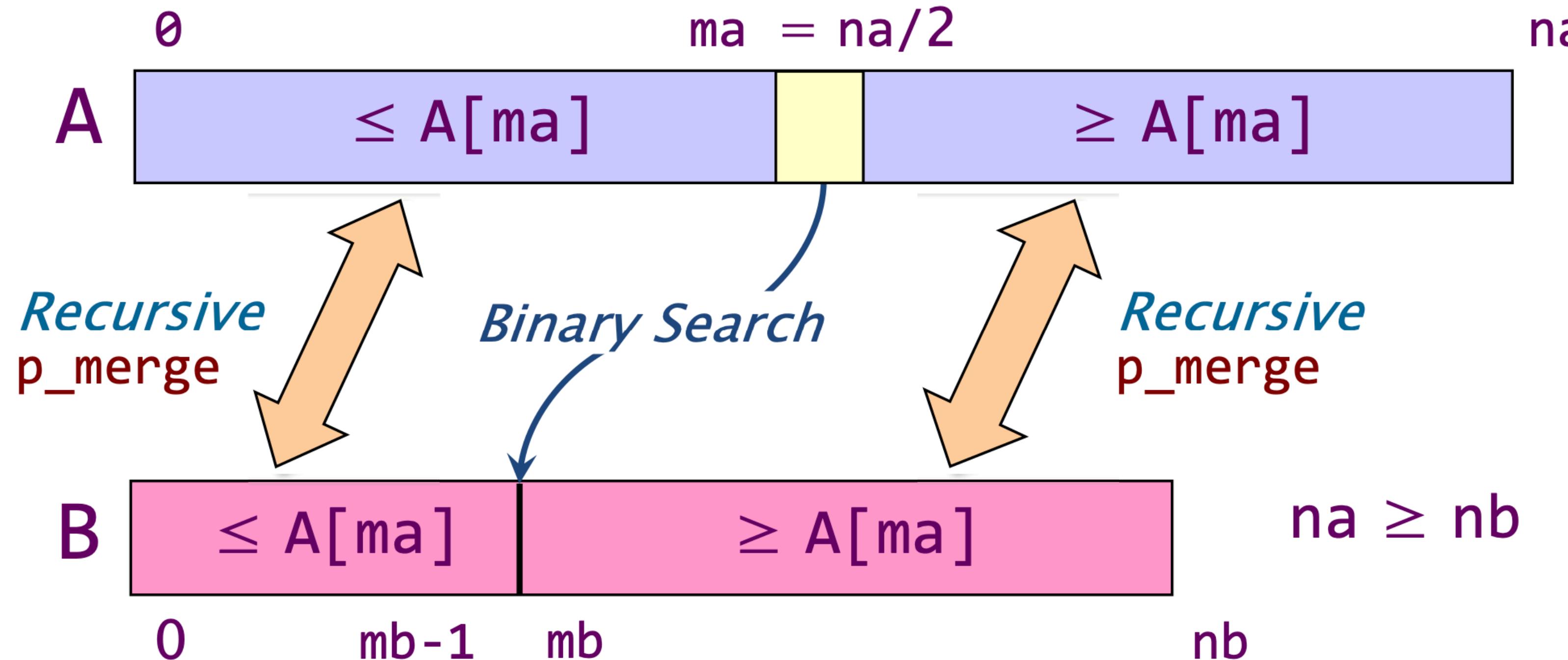
Span:  $T_\infty(n) = \Theta(n)$

Parallelism:  $T_1(n)/T_\infty(n) = \Theta(\lg n)$



We need to  
parallelize the  
merge!

# Parallel Merge



**KEY IDEA:** If the total number of elements to be merged in the two arrays is  $n = na + nb$ , the total number of elements in the larger of the two recursive merges is at most  $(3/4)n$ .

# Parallel Merge

```
void p_merge(int *C, int *A, int na, int *B, int nb) {
    if (na < nb) {
        p_merge(C, B, nb, A, na); // flip A and B
    } else if (na == 0) {
        return;
    } else {
        int ma = na/2;
        int mb = binary_search(A[ma], B, nb);
        C[ma+mb] = A[ma];
        parallel_spawn p_merge(C, A, ma, B, mb);
        p_merge(C+ma+mb+1, A+ma+1, na-ma-1, B+mb, nb-mb);
        parallel_sync;
    }
}
```

Coarsen base cases for efficiency.

# Span of Parallel Merge

```
void p_merge(int *C, int *A, int na, int *B, int nb) {
    if (na < nb) {
        p_merge(C, B, nb, A, na); // flip A and B
    } else if (na == 0) {
        return;
    } else {
        int ma = na/2;
        int mb = binary_search(A[ma], B, nb);
        C[ma+mb] = A[ma];
        parallel_spawn p_merge(C, A, ma, B, mb);
        p_merge(C+ma+mb+1, A+ma+1, na-ma-1, B+mb, nb-mb);
        parallel_sync;
    }
}
```

Case 2 of Master Method

$$n^{\log_b a} = n^{\log_{4/3} 1} = 1$$

$$f(n) = \Theta(n^{\log_b a} \lg^1 n)$$

$$\begin{aligned} \text{Span: } T_\infty(n) &= T_\infty(3n/4) + \Theta(\lg n) \\ &= \Theta(\lg^2 n) \end{aligned}$$

# Work of Parallel Merge

```
void p_merge(int *C, int *A, int na, int *B, int nb) {
    if (na < nb) {
        p_merge(C, B, nb, A, na); // flip A and B
    } else if (na == 0) {
        return;
    } else {
        int ma = na/2;
        int mb = binary_search(A[ma], B, nb);
        C[ma+mb] = A[ma];
        parallel_spawn p_merge(C, A, ma, B, mb);
        p_merge(C+ma+mb+1, A+ma+1, na-ma-1, B+mb, nb-mb);
        parallel_sync;
    }
}
```

Complicated / not in  
Master Method cases

$$\text{Work: } T_1(n) = T_1(\alpha n) + T_1((1 - \alpha)n) + \Theta(\lg n),$$

$$\text{where } 1/4 \leq \alpha \leq 3/4$$

$$\text{Claim: } T_1(n) = \Theta(n)$$

# Analysis of Work Recurrence

$$T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n),$$

where  $1/4 \leq \alpha \leq 3/4$ .

**Substitution method:** Inductive hypothesis is  $T_1(k) \leq c_1 k - c_2 \lg k$ , where  $c_1, c_2 > 0$ . Prove that the relation holds, and solve for  $c_1$  and  $c_2$ .

$$\begin{aligned} T_1(n) &\leq c_1(\alpha n) - c_2 \lg(\alpha n) + c_1(1-\alpha)n - c_2 \lg((1-\alpha)n) + \Theta(\lg n) \\ &\leq c_1 n - c_2 \lg(\alpha n) - c_2 \lg((1-\alpha)n) + \Theta(\lg n) \\ &\leq c_1 n - c_2 (\lg(\alpha(1-\alpha)) + 2 \lg n) + \Theta(\lg n) \\ &\leq c_1 n - c_2 \lg n - (c_2(\lg n + \lg(\alpha(1-\alpha)))) - \Theta(\lg n) \\ &\leq c_1 n - c_2 \lg n, \end{aligned}$$

by choosing  $c_2$  large enough. Choose  $c_1$  large enough to handle the base case.

# Parallelism of Parallel Merge

Work:  $T_1(n) = \Theta(n)$

Span:  $T_\infty(n) = \Theta(\lg^2 n)$

Parallelism:  $T_1(n)/T_\infty(n) = \Theta(n/\lg^2 n)$

# Parallel Merge Sort

```
void p_merge_sort(int *B, int *A, int n) {
    if (n == 1) {
        B[0] = A[0];
    } else {
        int C[n];
        parallel_spawn merge_sort(C, A, n/2);
        merge_sort(C+n/2, A+n/2, n-n/2);
        parallel_sync;
        p_merge(B, C, n/2, C+n/2, n-n/2);
    }
}
```

Same as before except  
with parallel merge

Case 2 of Master Method  
 $n^{\log_b a} = n^{\log_2 2} = n$   
 $f(n) = \Theta(n^{\log_b a} \lg^0 n)$

$$\begin{aligned}\text{Work: } T_1(n) &= 2T_1(n/2) + \Theta(n) \\ &= \Theta(n \lg n)\end{aligned}$$

# Parallel Merge Sort

```
void p_merge_sort(int *B, int *A, int n) {
    if (n == 1) {
        B[0] = A[0];
    } else {
        int C[n];
        parallel_spawn merge_sort(C, A, n/2);
        merge_sort(C+n/2, A+n/2, n-n/2);
        parallel_sync;
        p_merge(B, C, n/2, C+n/2, n-n/2);
    }
}
```

Same as before except  
with parallel merge

Case 2 of Master Method  
 $n^{\log_b a} = n^{\log_2 1} = 1$   
 $f(n) = \Theta(n^{\log_b a} \lg^2 n)$

$$\begin{aligned}\text{Span: } T_\infty(n) &= T_\infty(n/2) + \Theta(\lg^2 n) \\ &= \Theta(\lg^3 n)\end{aligned}$$

# Parallelism of Parallel Merge Sort

Work:  $T_1(n) = \Theta(n \lg n)$

Span:  $T_\infty(n) = \Theta(\lg^3 n)$

Parallelism:  $T_1(n)/T_\infty(n) = \Theta(n/\lg^2 n)$

# Tableau Construction

# Constructing a Tableau

Problem: Fill in an  $n \times n$  tableau  $A$ , where  $A[i][j] = f( A[i][j-1], A[i-1][j], A[i-1][j-1] )$ .

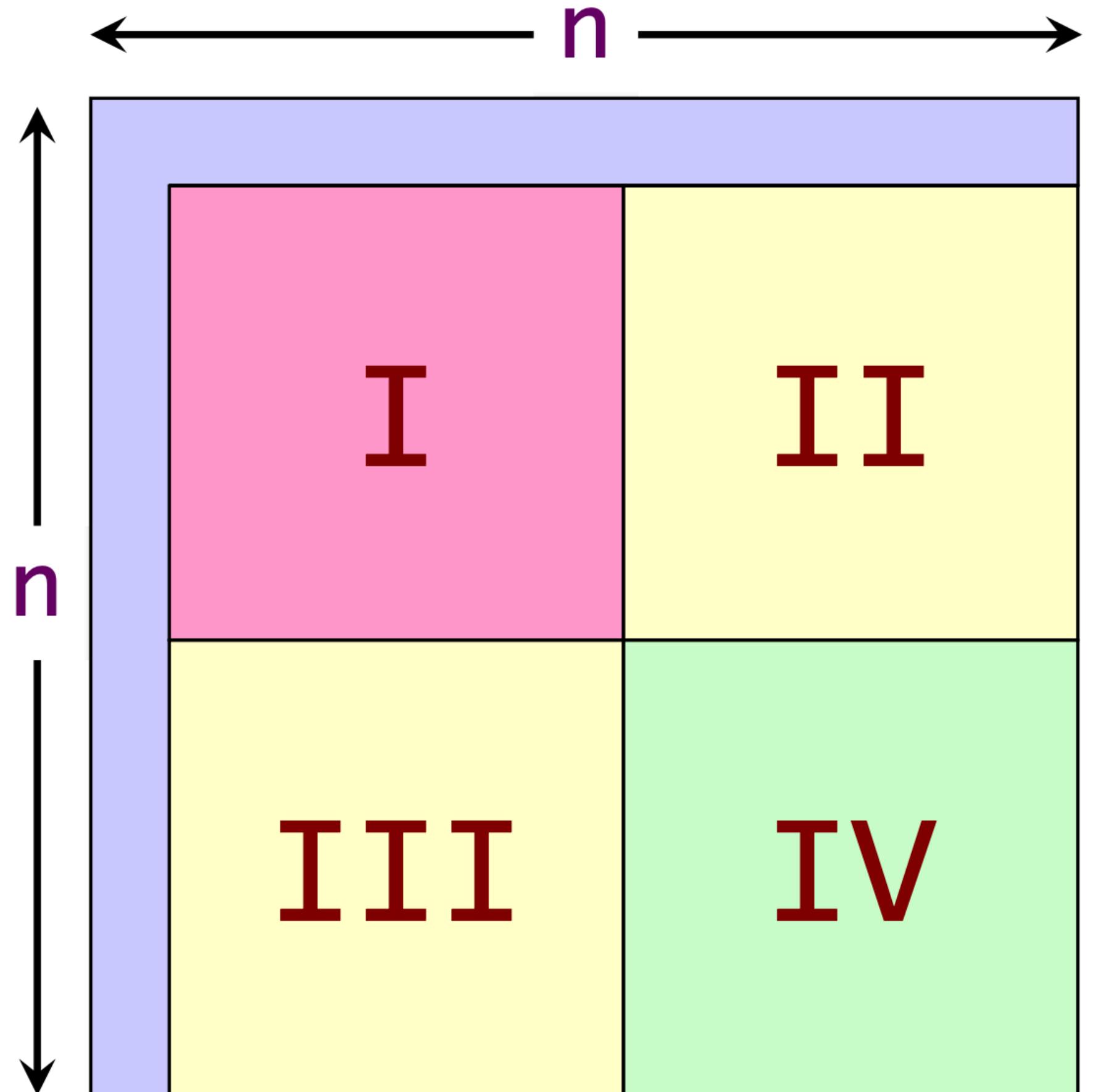
00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Dynamic programming

- Longest common subsequence
- Edit distance
- Time warping

Work =  $\Theta(n^2)$

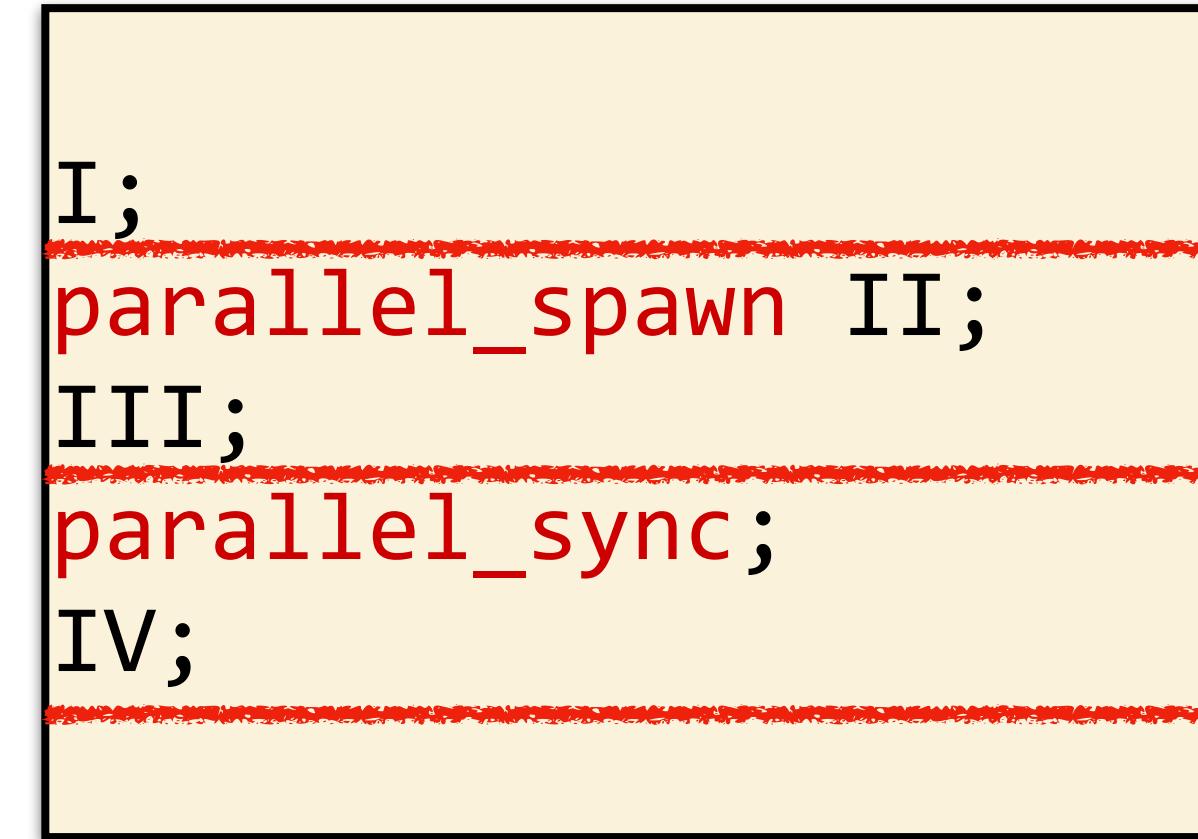
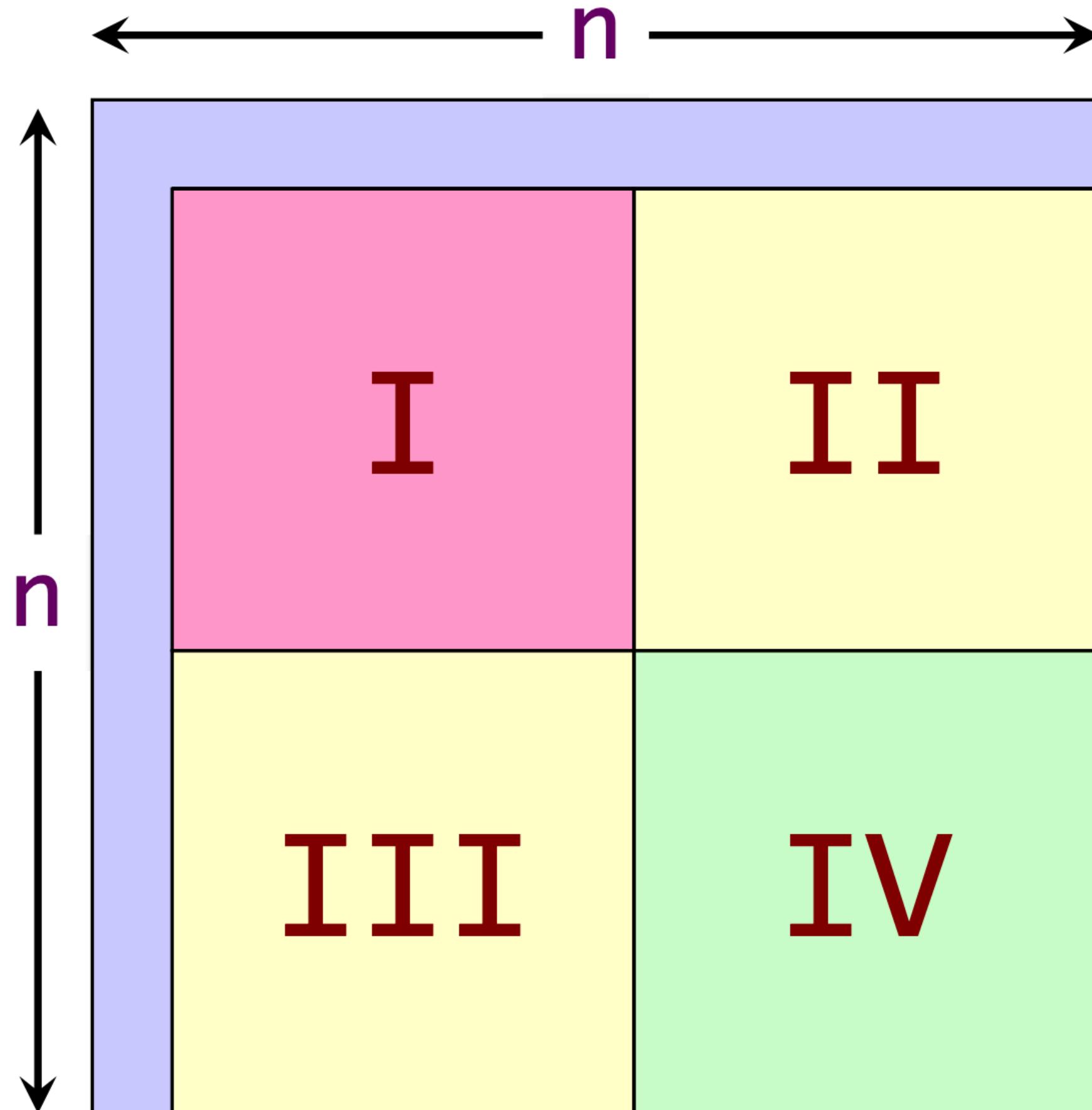
# Recursive Construction



```
I;  
parallel_spawn II;  
III;  
parallel_sync;  
IV;
```

What is the recurrence for  
the work?

# Recursive Construction



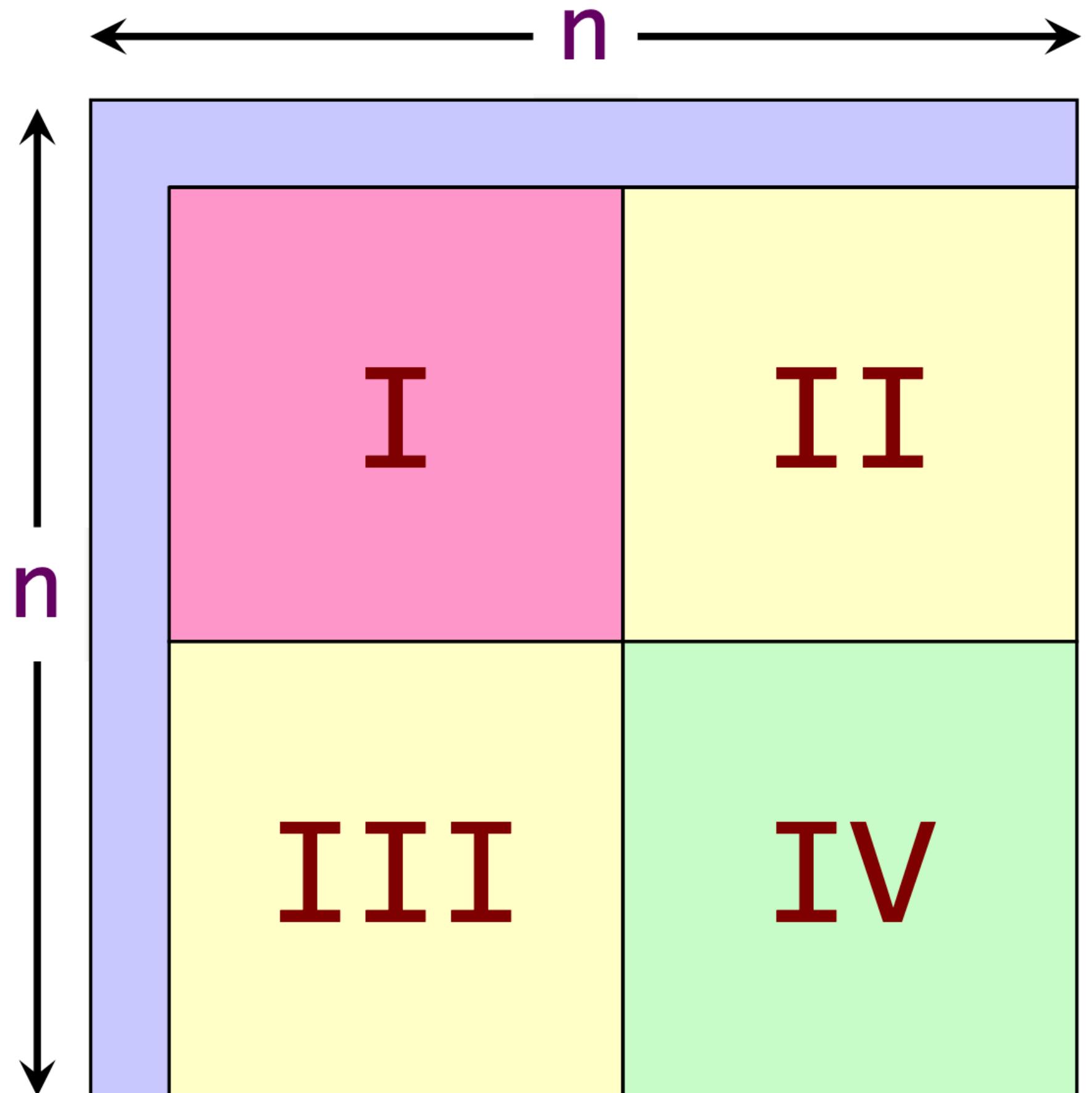
Work:  $T_1(n) = 4T_1(n/2) + \Theta(1) = \Theta(n^2)$

Case 1 of Master Method

$$n^{\log_b a} = n^{\log_2 4} = n^2$$

$$f(n) = \Theta(1)$$

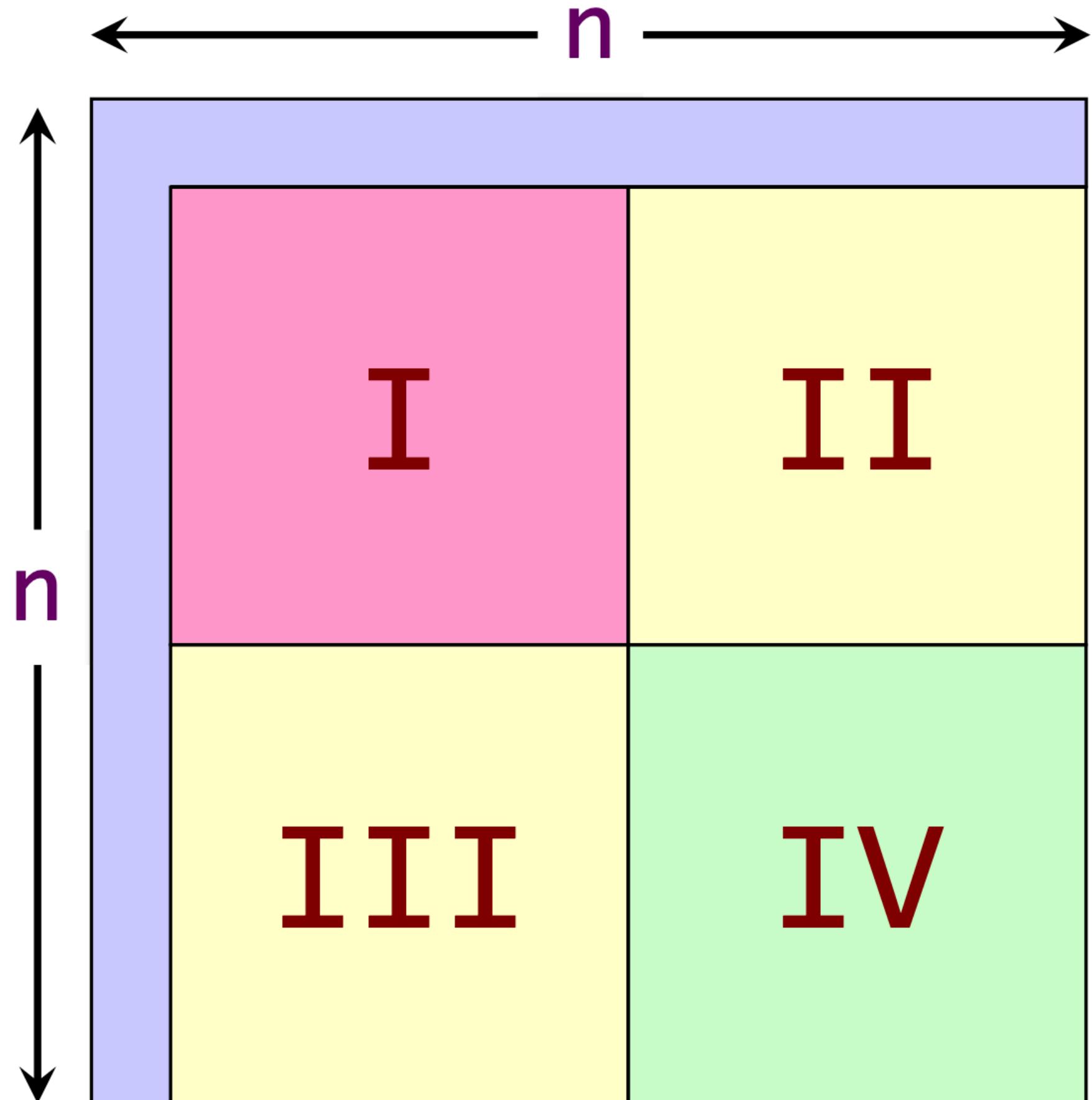
# Recursive Construction



```
I;  
parallel_spawn II;  
III;  
parallel_sync;  
IV;
```

What is the recurrence for  
the span?

# Recursive Construction



```
I;  
parallel_spawn II;  
III;  
parallel_sync;  
IV;
```

Span:  $T_\infty(n) = 3T_\infty(n/2) + \Theta(1) = \Theta(n^{\lg 3})$

Case 1 of Master Method

$$n^{\log_b a} = n^{\log_2 3} = n^{\lg 3}$$

$$f(n) = \Theta(1)$$

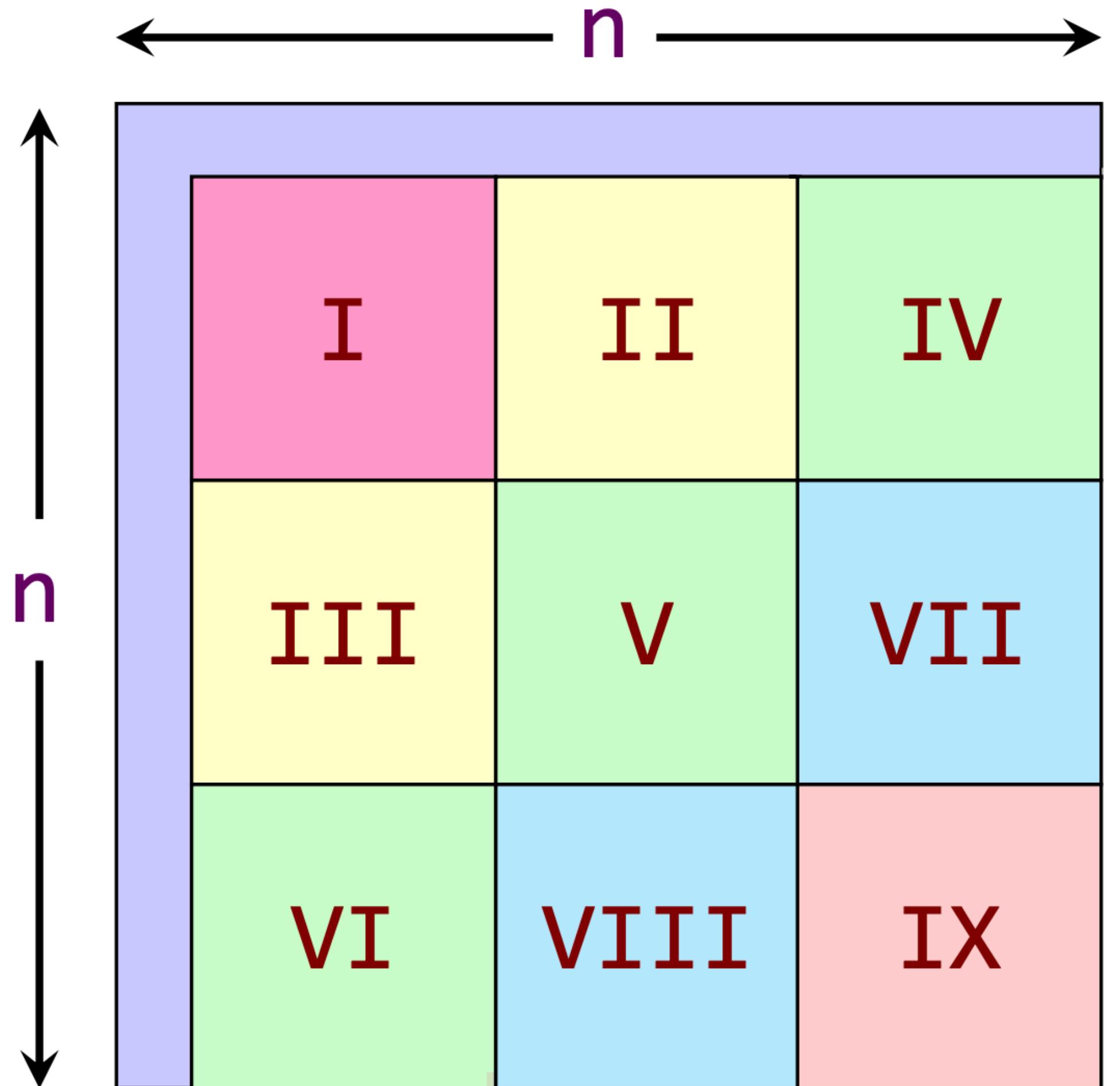
# Analysis of Tableau Construction

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

Span:  $T_\infty(n) = \Theta(\lg^3 n) = O(n^{1.59})$

Parallelism:  $T_1(n)/T_\infty(n) = \Theta(n^{2-\lg 3}) = \Omega(n^{0.41})$

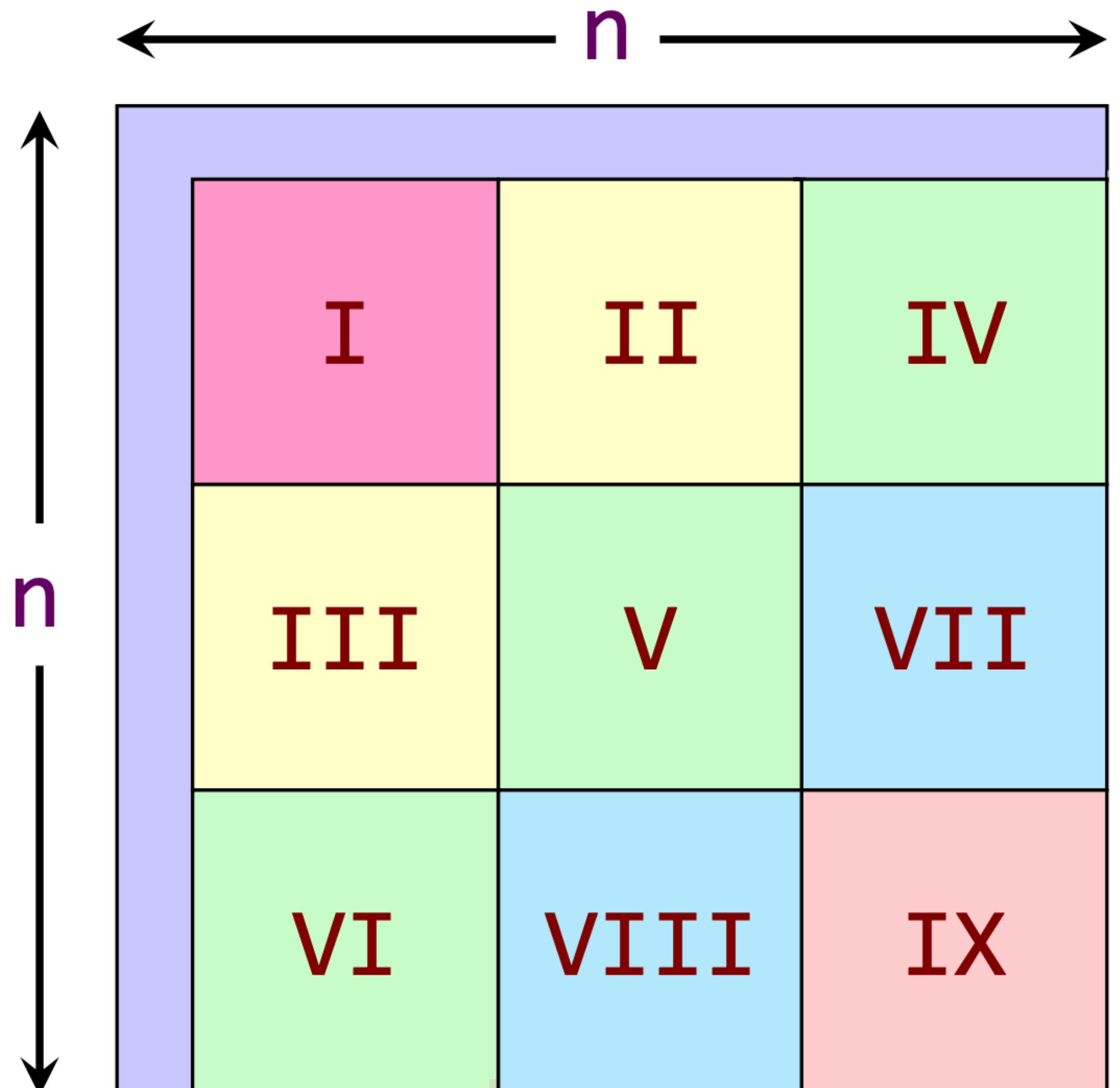
# A More Parallel Construction



```
I;  
parallel_spawn II;  
III;  
parallel_sync;  
parallel_spawn IV;  
parallel_spawn V;  
VI;  
parallel_sync;  
parallel_spawn VII;  
VIII;  
parallel_sync;  
IX;
```

What is the recurrence for  
the work?

# A More Parallel Construction



```
I;  
parallel_spawn II;  
III;  
parallel_sync;  
parallel_spawn IV;  
parallel_spawn V;  
VI;  
parallel_sync;  
parallel_spawn VII;  
VIII;  
parallel_sync;  
IX;
```

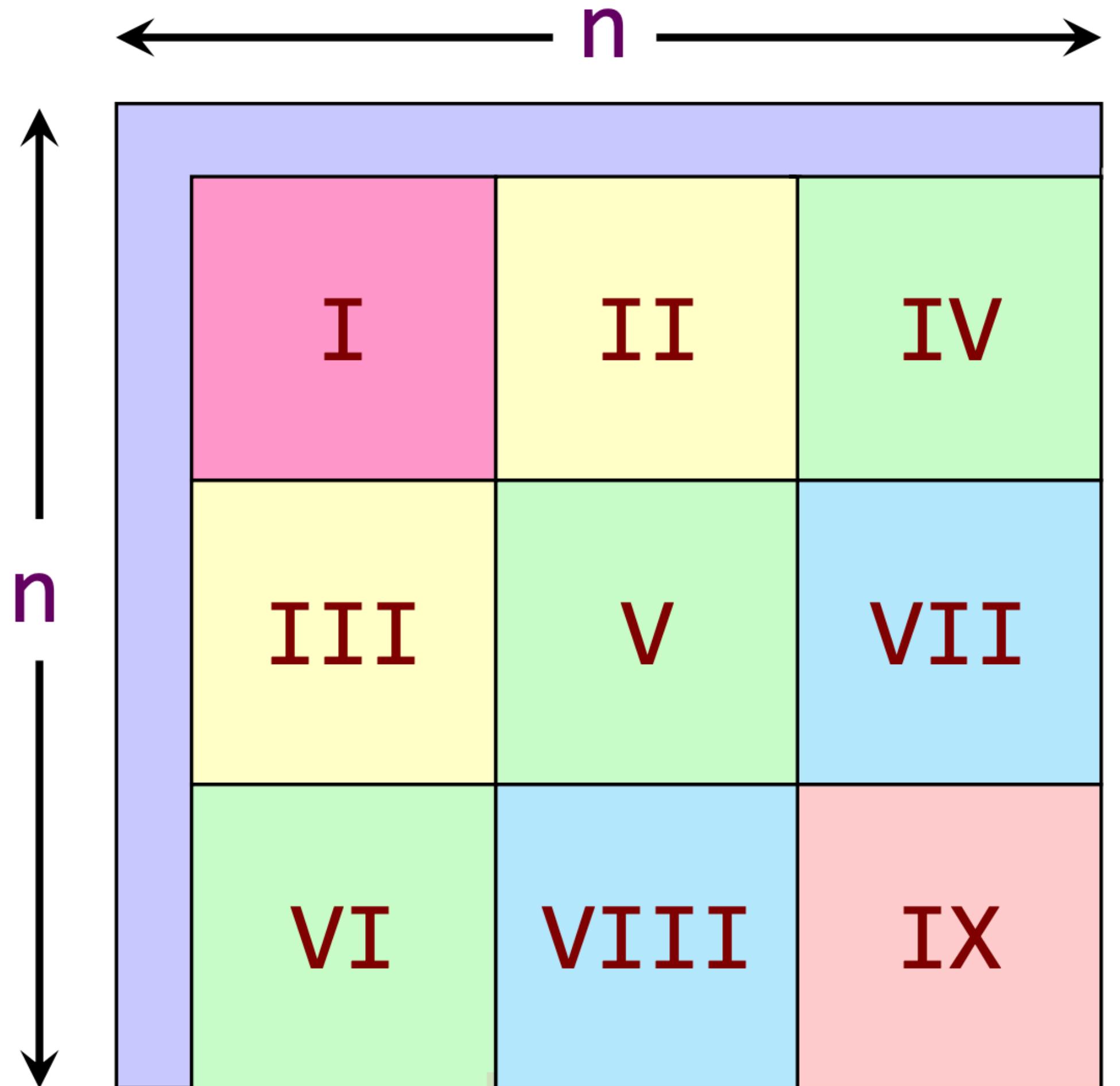
$$\text{Work: } T_1(n) = 9T_1(n/3) + \Theta(1) = \Theta(n^2)$$

Case 1 of Master Method

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

$$f(n) = \Theta(1)$$

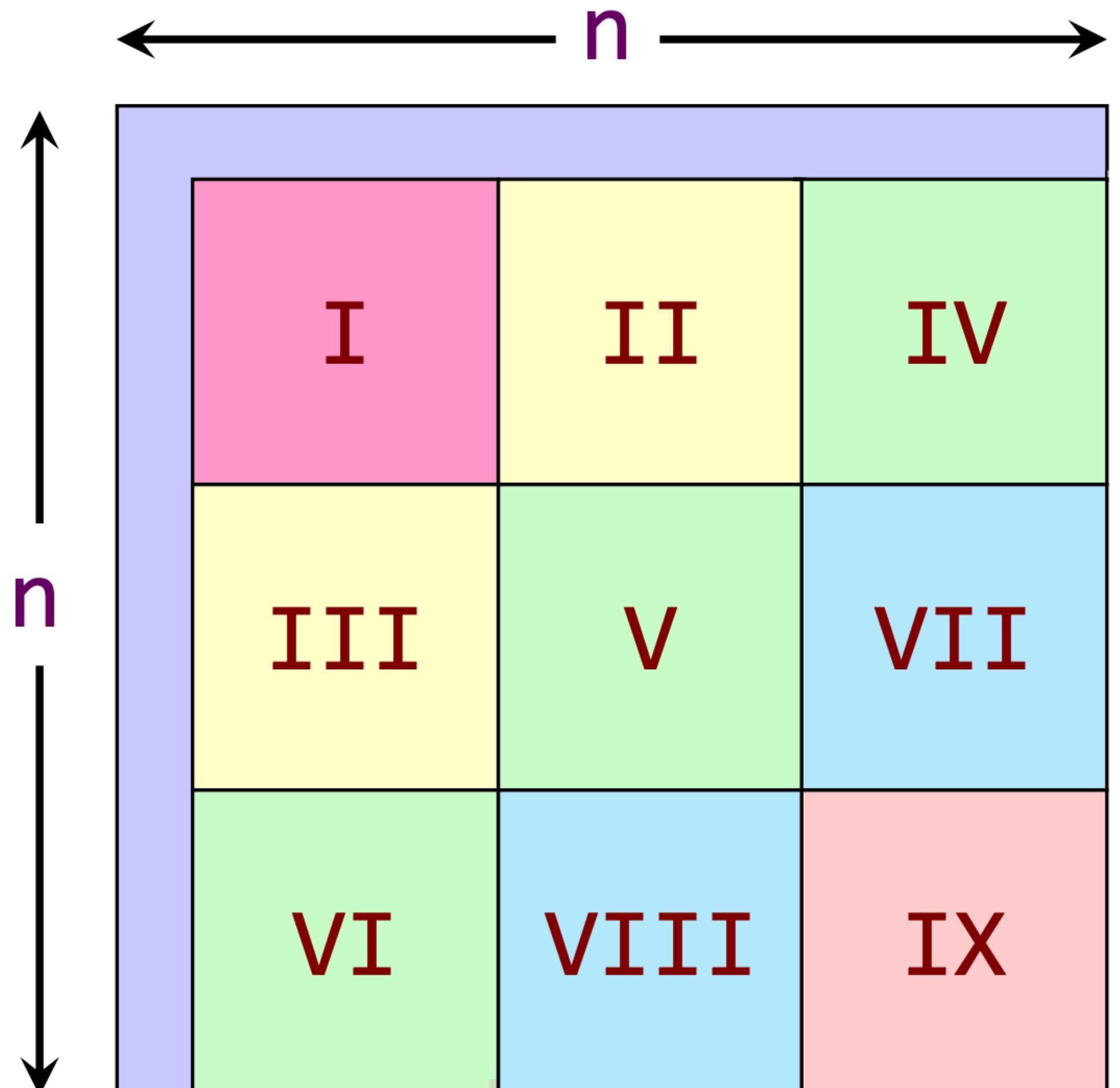
# A More Parallel Construction



```
I;  
parallel_spawn II;  
III;  
parallel_sync;  
parallel_spawn IV;  
parallel_spawn V;  
VI;  
parallel_sync;  
parallel_spawn VII;  
VIII;  
parallel_sync;  
IX;
```

What is the recurrence for  
the span?

# A More Parallel Construction



```
I;  
parallel_spawn II;  
III;  
parallel_sync;  
parallel_spawn IV;  
parallel_spawn V;  
VI;  
parallel_sync;  
parallel_spawn VII;  
VIII;  
parallel_sync;  
IX;
```

$$\text{Work: } T_\infty(n) = 5T_\infty(n/3) + \Theta(1) = \Theta(n^{\log_3 5})$$

Case 1 of Master Method

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

$$f(n) = \Theta(1)$$

# Analysis of Revised Method

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

Span:  $T_\infty(n) = \Theta(\log_3 5) = O(n^{1.47})$

Parallelism:  $T_1(n)/T_\infty(n) = \Theta(n^{2-\log_3 5}) = \Omega(n^{0.53})$

Nine-way divide-and-conquer has  
about  $\Omega(n^{0.12})$  **more parallelism**  
than four-way divide-and-conquer,  
but it exhibits **less cache locality**

# Puzzle

What is the largest parallelism that can be obtained for the tableau-construction problem in the binary forking model?

- You may only use basic fork-join control constructs (e.g., `parallel_spawn`, `parallel_sync`, `parallel_for`) for synchronization.
- No using locks, atomic instructions, synchronizing through memory, etc.