



The lecture is partly based on  
Charles Leiserson's Slides on Cilk

# The Master Method

The *Master Method* for solving recurrences applies to recurrences of the form

$$T(n) = a T(n/b) + f(n) \text{ ,}^*$$

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

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

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

# Master Method — CASE 1

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

$$n^{\log_b a} \gg f(n)$$

Specifically,  $f(n) = O(n^{\log_b a - \varepsilon})$  for some constant  $\varepsilon > 0$ .

**Solution:**  $T(n) = \Theta(n^{\log_b a})$  .

# Master Method — CASE 2

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

$$n^{\log_b a} \approx f(n)$$

Specifically,  $f(n) = \Theta(n^{\log_b a} \lg^k n)$  for some constant  $k \geq 0$ .

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

# Master Method — CASE 3

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

$$n^{\log_b a} \ll f(n)$$

Specifically,  $f(n) = \Omega(n^{\log_b a + \varepsilon})$  for some constant  $\varepsilon > 0$  and  $f(n)$  satisfies the *regularity condition* that  $af(n/b) \leq cf(n)$  for some constant  $c < 1$ .

**Solution:**  $T(n) = \Theta(f(n))$  .

# Master Method Summary

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

**CASE 1:**  $f(n) = O(n^{\log_b a - \varepsilon})$ , constant  $\varepsilon > 0$

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

**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:**  $f(n) = \Omega(n^{\log_b a + \varepsilon})$ , constant  $\varepsilon > 0$ ,  
and regularity condition

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

# Master Method Quiz

- $T(n) = 4 T(n/2) + n$

$n^{\log_b a} = n^2 \gg n \Rightarrow \text{CASE 1: } T(n) = \Theta(n^2).$

- $T(n) = 4 T(n/2) + n^2$

$n^{\log_b a} = n^2 = n^2 \lg^0 n \Rightarrow \text{CASE 2: } T(n) = \Theta(n^2 \lg n).$

- $T(n) = 4 T(n/2) + n^3$

$n^{\log_b a} = n^2 \ll n^3 \Rightarrow \text{CASE 3: } T(n) = \Theta(n^3).$

- $T(n) = 4 T(n/2) + n^2/\lg n$

*Master method does not apply!*

# Outline

- Recurrences (Review)
- Matrix Multiplication
- Merge Sort
- Tableau Construction
- Conclusion

# Square-Matrix Multiplication

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

$C$        $A$        $B$

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

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

# Recursive Matrix Multiplication

*Divide and conquer —*

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

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

1 addition of  $n \times n$  matrices.

# Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
    float *T = Cilk_alloc(n*n*sizeof(float));
    <base case & partition matrices>
    spawn Mult(C11,A11,B11,n/2);
    spawn Mult(C12,A11,B12,n/2);
    spawn Mult(C22,A21,B12,n/2);
    spawn Mult(C21,A21,B11,n/2);
    spawn Mult(T11,A12,B21,n/2);
    spawn Mult(T12,A12,B22,n/2);
    spawn Mult(T22,A22,B22,n/2);
    spawn Mult(T21,A22,B21,n/2);
    sync;
    spawn Add(C,T,n);
    sync;
    return;
}
```

$$C = A \cdot B$$

*Absence of type declarations.*

# Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
    float *T = Cilk_alloca(n*n*sizeof(float));
    <base case & partition matrices>
    spawn Mult(C11,A11,B11,n/2);
    spawn Mult(C12,A11,B12,n/2);
    spawn Mult(C22,A21,B12,n/2);
    spawn Mult(T11,A21,B11,n/2);
    spawn Mult(T12,A12,B21,n/2);
    spawn Mult(T12,A12,B22,n/2);
    spawn Mult(T22,A22,B22,n/2);
    spawn Mult(T21,A21,B11,n/2);
    sync;
    spawn Add(C,T,n);
    sync;
    return;
}
```

$$C = A \cdot B$$

*Coarsen base cases  
for efficiency.*

# Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
    float *T = Cilk_alloca(n*n*sizeof(float));
    <base case & partition matrices>
    spawn Mult(C11,A11,B11,n/2);
    spawn Mult(C12,A11,B12,n/2);
    spawn Mult(C22,A21,B12,n/2);
    spawn Mult(C21,A21,B12,n/2);
    spawn Mult(T11,A12,B21,n/2);
    spawn Mult(T12,A12,B22,n/2);
    spawn Mult(T22,A22,B22,n/2);
    spawn Mult(T21,A22,B21,n/2);
    sync;
    spawn Add(C,T,n);
    sync;
    return;
}
```

$$C = A \cdot B$$

*Also need a row-size argument for array indexing.*

*Submatrices are produced by pointer calculation, not copying of elements.*

# Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
    float *T = Cilk_alloca(n*n*sizeof(float));
    <base case & partition matrices>
    spawn Mult(C11,A11,B11,n/2);
    spawn Mult(C12,A11,B12,n/2);
    spawn Mult(C22,A21,B12,n/2);
    spawn Mult(C21,A21,B11,n/2);
    spawn Mult(T11,A12,B21,n/2);
    spawn Mult(T12,A12,B22,n/2);
    spawn Mult(T22,A22,B22,n/2);
    spawn Mult(T21,A22,B21,n/2);
    sync;
    spawn Add(C,T,n);
    sync;
    return;
}
```

$$C = A \cdot B$$

$$C = C + T$$

```
cilk void Add(*C, *T, n) {
    <base case & partition matrices>
    spawn Add(C11,T11,n/2);
    spawn Add(C12,T12,n/2);
    spawn Add(C21,T21,n/2);
    spawn Add(C22,T22,n/2);
    sync;
    return;
}
```

# Work of Matrix Addition

```
cilk void Add(*C, *T, n) {  
    <base case & partition matrices>  
    spawn Add(C11, T11, n/2);  
    spawn Add(C12, T12, n/2);  
    spawn Add(C21, T21, n/2);  
    spawn Add(C22, T22, n/2);  
    sync;  
    return;  
}
```

*Work:*     $A_1(n) = 4A_1(n/2) + \Theta(1)$   
                       $= \Theta(n^2)$  — CASE 1

$$n^{\log_b a} = n^{\log_2 4} = n^2 \gg \Theta(1).$$

# Span of Matrix Addition

maximum

```
cilk void Add(*C, *T, n) {  
    <base case & partition matrices>  
    {  
        spawn Add(C11, T11, n/2);  
        spawn Add(C12, T12, n/2);  
        spawn Add(C21, T21, n/2);  
        spawn Add(C22, T22, n/2);  
        sync;  
        return;  
    }  
}
```

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

$$n^{\log_b a} = n^{\log_2 1} = 1 \Rightarrow f(n) = \Theta(n^{\log_b a} \lg^0 n) .$$

# Work of Matrix Multiplication

8

```
cilk void Mult(*C, *A, *B, n) {
    float *T = Cilk_alloca(n*n*sizeof(float));
    <base case & partition matrices>
    spawn Mult(C11,A11,B11,n/2);
    spawn Mult(C12,A11,B12,n/2);
    :
    spawn Mult(T21,A22,B21,n/2);
    sync;
    spawn Add(C,T,n);
    sync;
    return;
}
```

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

$$n^{\log_b a} = n^{\log_2 8} = n^3 \gg \Theta(n^2).$$

# Span of Matrix Multiplication

8

```
cilk void Mult(*C, *A, *B, n) {
    float *T = Cilk_malloc(n*n*sizeof(float));
    <base case & partition matrices>
    spawn Mult(C11,A11,B11,n/2);
    spawn Mult(C12,A11,B12,n/2);
    :
    spawn Mult(T21,A22,B21,n/2);
    sync;
    spawn Add(C,T,n);
    sync;
    return;
}
```

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

$$n^{\log_b a} = n^{\log_2 1} = 1 \Rightarrow f(n) = \Theta(n^{\log_b a} \lg^1 n) .$$

# Parallelism of Matrix Multiply

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

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

---

---

---

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

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

# Stack Temporaries

```
cilk void Mult(*C, *A, *B, n) {
    float *T = Cilk_alloca(n*n*sizeof(float));
    <base case & partition matrices>
    spawn Mult(C11,A11,B11,n/2);
    spawn Mult(C12,A11,B12,n/2);
    :
    spawn Mult(T21,A22,B21,n/2);
    sync;
    spawn Add(C,T,n);
    sync;
    return;
}
```

*In hierarchical-memory machines (especially chip multiprocessors), memory accesses are so expensive that minimizing storage often yields higher performance.*

**IDEA:** Trade off parallelism for less storage.

# No-Temp Matrix Multiplication

```
cilk void MultA(*C, *A, *B, n) {
    // C = C + A * B
    <base case & partition matrices>
    spawn MultA(C11,A11,B11,n/2);
    spawn MultA(C12,A11,B12,n/2);
    spawn MultA(C22,A21,B12,n/2);
    spawn MultA(C21,A21,B11,n/2);
    sync;
    spawn MultA(C21,A22,B21,n/2);
    spawn MultA(C22,A22,B22,n/2);
    spawn MultA(C12,A12,B22,n/2);
    spawn MultA(C11,A12,B21,n/2);
    sync;
    return;
}
```

*Saves space, but at what expense?*

# Work of No-Temp Multiply

```
cilk void MultA(*C, *A, *B, n) {
    // C = C + A * B
    ⟨base case & partition matrices⟩
    spawn MultA(C11,A11,B11,n/2);
    spawn MultA(C12,A11,B12,n/2);
    spawn MultA(C22,A21,B12,n/2);
    spawn MultA(C21,A21,B11,n/2);
    sync;
    spawn MultA(C21,A22,B21,n/2);
    spawn MultA(C22,A22,B22,n/2);
    spawn MultA(C12,A12,B22,n/2);
    spawn MultA(C11,A12,B21,n/2);
    sync;
    return;
}
```

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

# Span of No-Temp Multiply

maximum

maximum

```
cilk void MultA(*C, *A, *B, n) {
    // C = C + A * B
    { base case & partition matrices }
    spawn MultA(C11,A11,B11,n/2);
    spawn MultA(C12,A11,B12,n/2);
    spawn MultA(C22,A21,B12,n/2);
    spawn MultA(C21,A21,B11,n/2);
    sync;
    { base case & partition matrices }
    spawn MultA(C21,A22,B21,n/2);
    spawn MultA(C22,A22,B22,n/2);
    spawn MultA(C12,A12,B22,n/2);
    spawn MultA(C11,A12,B21,n/2);
    sync;
    return;
}
```

$$\begin{aligned} \textit{Span: } M_\infty(n) &= 2 M_\infty(n/2) + \Theta(1) \\ &= \Theta(n) \quad \text{— CASE 1} \end{aligned}$$

# Parallelism of No-Temp Multiply

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

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

---

---

---

*Parallelism:*  $\frac{M_1(n)}{M_\infty(n)} = \Theta(n^2)$

For  $1000 \times 1000$  matrices,  
parallelism  $\approx (10^3)^3/10^3 = 10^6$ .

*Faster in practice!*

# Testing Synchronization

*Cilk language feature:* A programmer can check whether a Cilk procedure is “synched” (without actually performing a **sync**) by testing the pseudovariable **SYNCED**:

- **SYNCED = 0**  $\Rightarrow$  some spawned children might not have returned.
- **SYNCED = 1**  $\Rightarrow$  all spawned children have definitely returned.

# Best of Both Worlds

```
cilk void Mult1(*C, *A, *B, n) { // multiply & store
    // base case & partition matrices
    spawn Mult1(C11,A11,B11,n/2); // multiply & store
    spawn Mult1(C12,A11,B12,n/2);
    spawn Mult1(C22,A21,B12,n/2);
    spawn Mult1(C21,A21,B11,n/2);
    if (SYNCHED) {
        spawn MultA1(C11,A12,B21,n/2); // multiply & add
        spawn MultA1(C12,A12,B22,n/2);
        spawn MultA1(C22,A22,B22,n/2);
        spawn MultA1(C21,A22,B21,n/2);
    } else {
        float *T = Cilk_alloca(n*n*sizeof(float));
        spawn Mult1(T11,A11,B11,n/2);
        spawn Mult1(T12,A11,B12,n/2);
        spawn Mult1(T21,A21,B11,n/2);
        spawn Mult1(T22,A21,B12,n/2);
        sync;
        spawn Add(C,T11,T12,T21,T22);
    }
    sync;
    return;
}
```

This code is just as parallel as the original, but it only uses more space if runtime parallelism actually exists.

# Ordinary Matrix Multiplication

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

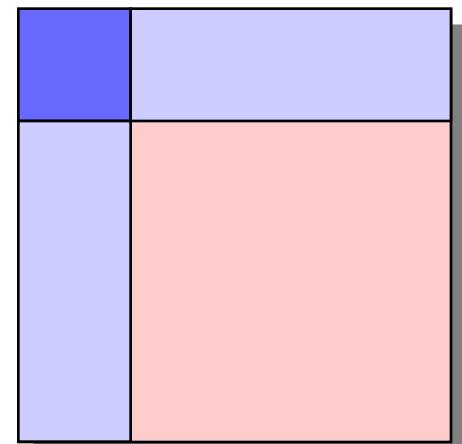
---

**IDEA:** Spawn  $n^2$  inner products in parallel.  
Compute each inner product in parallel.

**Work:**  $\Theta(n^3)$   
**Span:**  $\Theta(\lg n)$   
**Parallelism:**  $\Theta(n^3/\lg n)$

---

**BUT,** this algorithm exhibits poor locality and does not exploit the cache hierarchy of modern microprocessors, especially CMP's.



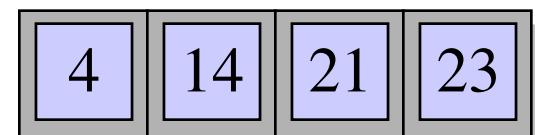
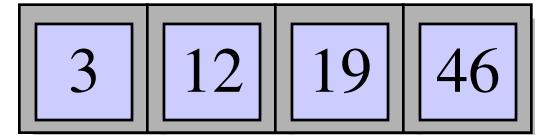
# Outline

- Recurrences (Review)
- Matrix Multiplication
- Merge Sort
- Tableau Construction
- Conclusion

# Merging Two Sorted Arrays

```
void Merge(int *C, int *A, int *B, int na, 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--;  
    }  
}
```

Time to merge  $n$  elements =  $\Theta(n)$ .



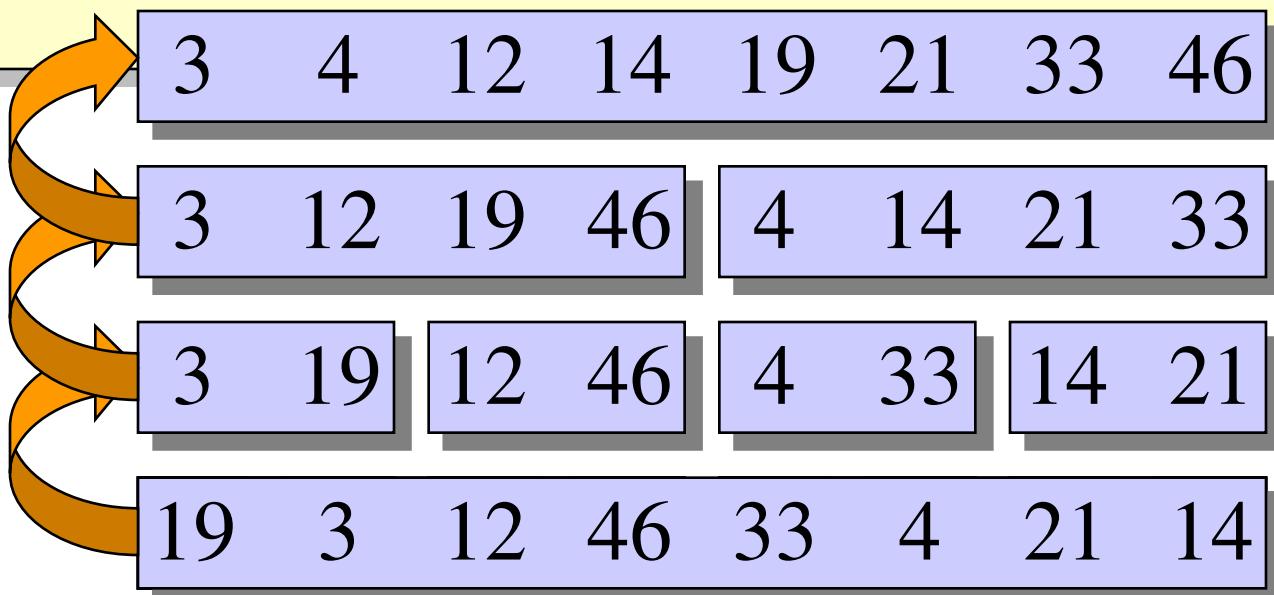
# Merge Sort

```
cilk void MergeSort(int *B, int *A, int n) {  
    if (n==1) {  
        B[0] = A[0];  
    } else {  
        int *C;  
        C = (int*) Cilk_alloca(n*sizeof(int));  
        spawn MergeSort(C, A, n/2);  
        spawn MergeSort(C+n/2, A+n/2, n-n/2);  
        sync;  
        Merge(B, C, C+n/2, n/2, n-n/2);  
    }  
}
```

*merge*

*merge*

*merge*



# Work of Merge Sort

```
cilk void MergeSort(int *B, int *A, int n) {  
    if (n==1) {  
        B[0] = A[0];  
    } else {  
        int *C;  
        C = (int*) Cilk_alloca(n*sizeof(int));  
        spawn MergeSort(C, A, n/2);  
        spawn MergeSort(C+n/2, A+n/2, n-n/2);  
        sync;  
        Merge(B, C, C+n/2, n/2, n-n/2);  
    }  
}
```

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

$$n^{\log_b a} = n^{\log_2 2} = n \Rightarrow f(n) = \Theta(n^{\log_b a} \lg^0 n) .$$

# Span of Merge Sort

```
cilk void MergeSort(int *B, int *A, int n) {  
    if (n==1) {  
        B[0] = A[0];  
    } else {  
        int *C;  
        C = (int*) Cilk_alloca(n*sizeof(int));  
        spawn MergeSort(C, A, n/2);  
        spawn MergeSort(C+n/2, A+n/2, n-n/2);  
        sync;  
        Merge(B, C, C+n/2, n/2, n-n/2);  
    }  
}
```

$$\begin{aligned} \textit{Span: } T_\infty(n) &= T_\infty(n/2) + \Theta(n) \\ &= \Theta(n) \quad \text{— CASE 3} \end{aligned}$$

$$n^{\log_b a} = n^{\log_2 1} = 1 \ll \Theta(n).$$

# Parallelism of Merge Sort

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

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

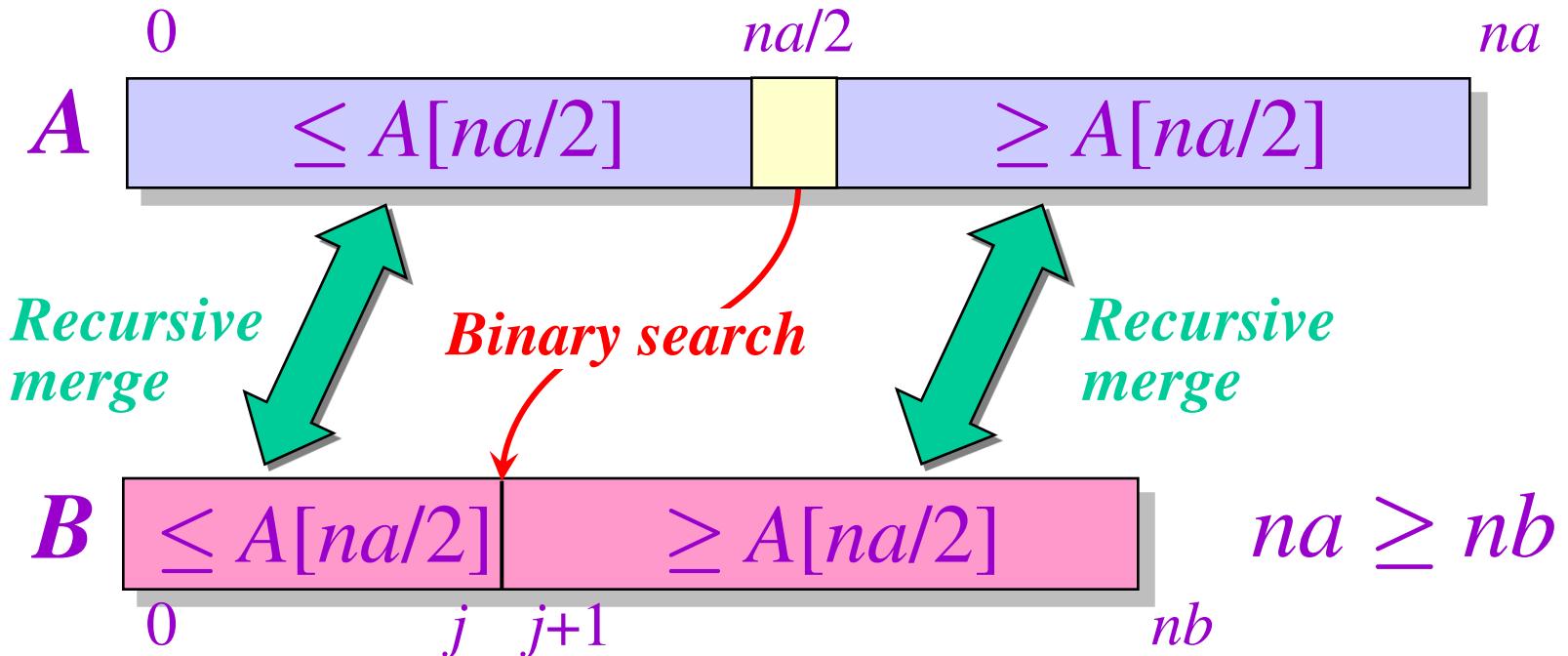
PUNY!

---

*Parallelism:*  $\frac{T_1(n)}{T_\infty(n)} = \Theta(\lg n)$

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

```
cilk void P_Merge(int *C, int *A, int *B,
                  int na, int nb) {
    if (na < nb) {
        spawn P_Merge(C, B, A, nb, na);
    } else if (na==1) {
        if (nb == 0) {
            C[0] = A[0];
        } else {
            C[0] = (A[0]<B[0]) ? A[0] : B[0]; /* minimum */
            C[1] = (A[0]<B[0]) ? B[0] : A[0]; /* maximum */
        }
    } else {
        int ma = na/2;
        int mb = BinarySearch(A[ma], B, nb);
        spawn P_Merge(C, A, B, ma, mb);
        spawn P_Merge(C+ma+mb, A+ma, B+mb, na-ma, nb-mb);
        sync;
    }
}
```

*Coarsen base cases for efficiency.*

# Span of P\_Merge

```
cilk void P_Merge(int *C, int *A, int *B,
                  int na, int nb) {
    if (na < nb) {
        .
        .
    } else {
        int ma = na/2;
        int mb = BinarySearch(A[ma], B, nb);
        spawn P_Merge(C, A, B, ma, mb);
        spawn P_Merge(C+ma+mb, A+ma, B+mb, na-ma, nb-mb);
        sync;
    }
}
```

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

$$n^{\log_b a} = n^{\log_{4/3} 1} = 1 \Rightarrow f(n) = \Theta(n^{\log_b a} \lg^1 n) .$$

# Work of P\_Merge

```
cilk void P_Merge(int *C, int *A, int *B,
                  int na, int nb) {
    if (na < nb) {
        .
        .
    } else {
        int ma = na/2;
        int mb = BinarySearch(A[ma], B, nb);
        spawn P_Merge(C, A, B, ma, mb);
        spawn P_Merge(C+ma+mb, A+ma, B+mb,
                      nb-mb);
        sync;
    }
}
```

HAIRY!

**Work:**  $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$ ,  
where  $1/4 \leq \alpha \leq 3/4$ .

**CLAIM:**  $T_1(n) = \Theta(n)$  .

# Parallelism of P\_Merge

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

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

---

---

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

# Parallel Merge Sort

```
cilk void P_MergeSort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        int *C;
        C = (int*) Cilk_alloca(n*sizeof(int));
        spawn P_MergeSort(C, A, n/2);
        spawn P_MergeSort(C+n/2, A+n/2, n-n/2);
        sync;
        spawn P_Merge(B, C, C+n/2, n/2, n-n/2);
    }
}
```

# Work of Parallel Merge Sort

```
cilk void P_MergeSort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        int *C;
        C = (int*) Cilk_alloca(n*sizeof(int));
        spawn P_MergeSort(C, A, n/2);
        spawn P_MergeSort(C+n/2, A+n/2, n-n/2);
        sync;
        spawn P_Merge(B, C, C+n/2, n/2, n-n/2);
    }
}
```

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

# Span of Parallel Merge Sort

```
cilk void P_MergeSort(int *B, int *A, int n) {
    if (n==1) {
        B[0] = A[0];
    } else {
        int *C;
        C = (int*) Cilk_alloca(n*sizeof(int));
        spawn P_MergeSort(C, A, n/2);
        spawn P_MergeSort(C+n/2, A+n/2, n-n/2);
        sync;
        spawn P_Merge(B, C, C+n/2, n/2, n-n/2);
    }
}
```

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

$$n^{\log_b a} = n^{\log_2 1} = 1 \Rightarrow f(n) = \Theta(n^{\log_b a} \lg^2 n) .$$

# Parallelism of Merge Sort

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

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

---

---

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

# Outline

- Recurrences (Review)
- Matrix Multiplication
- Merge Sort
- **Tableau Construction**
- Conclusion

# Tableau Construction

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

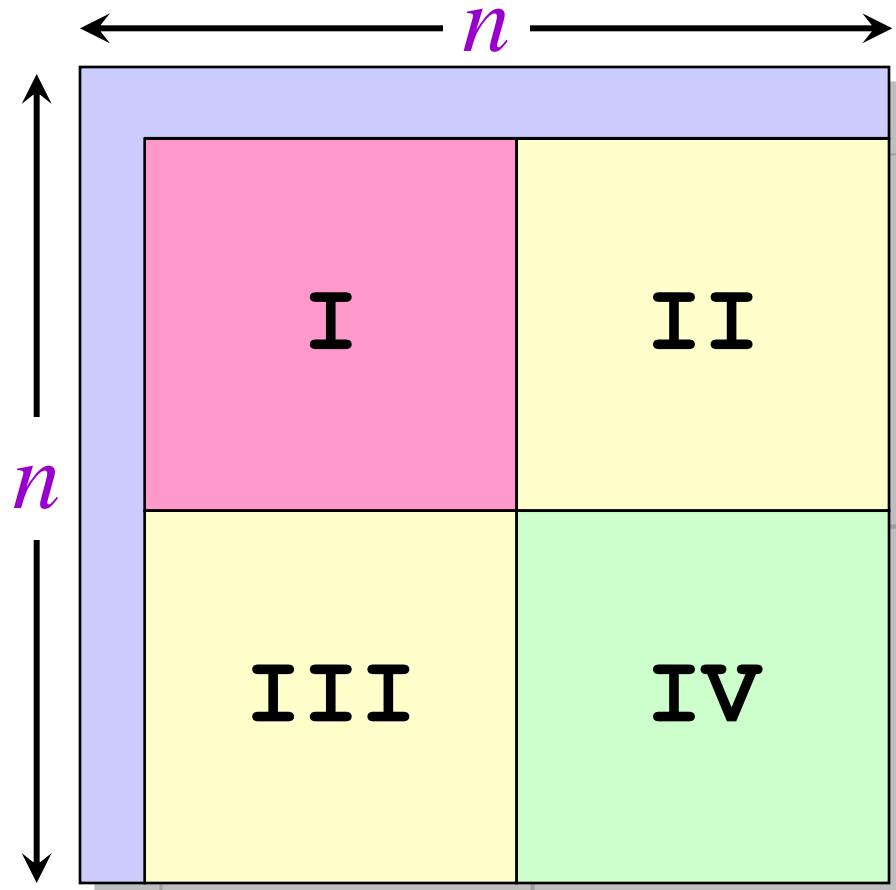
00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

*Dynamic  
programming*

- Longest common subsequence
- Edit distance
- Time warping

*Work:  $\Theta(n^2)$ .*

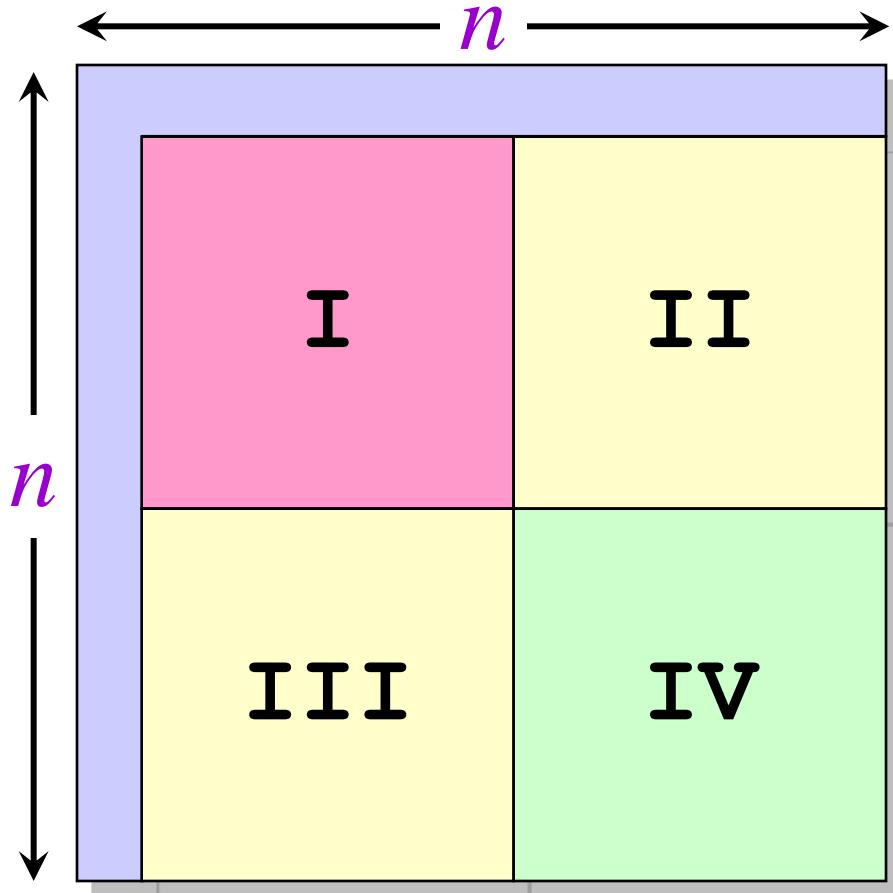
# Recursive Construction



*Cilk code*

```
spawn I;  
sync;  
spawn II;  
spawn III;  
sync;  
spawn IV;  
sync;
```

# Recursive Construction

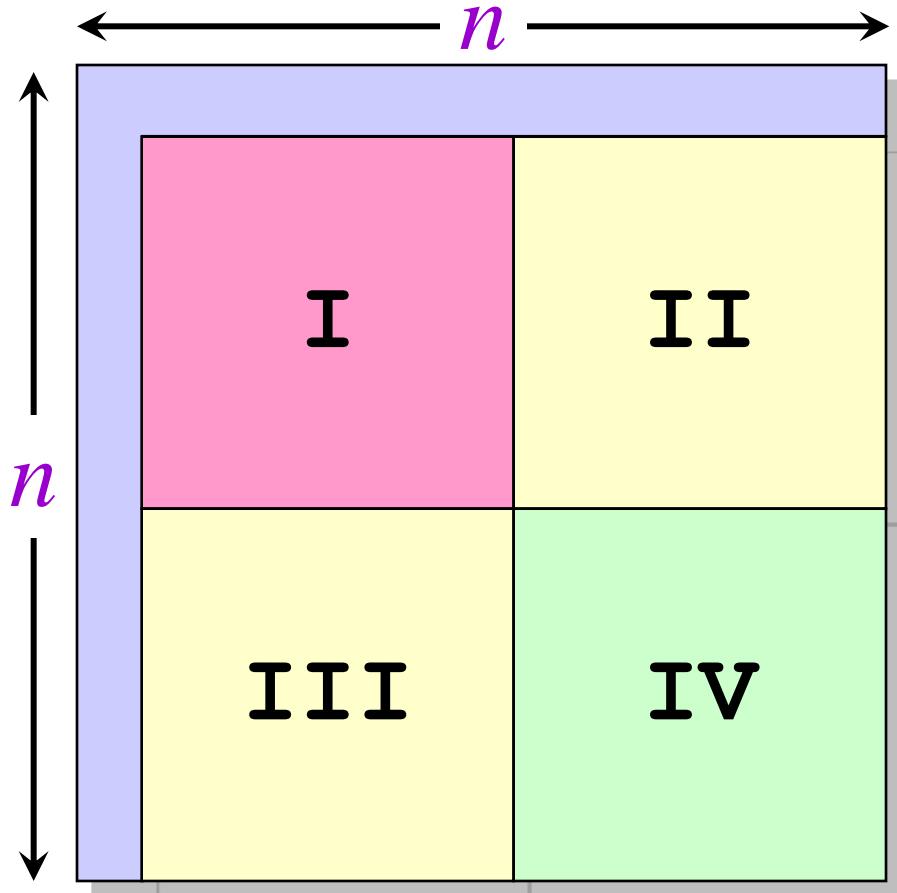


*Cilk code*

```
spawn I;  
sync;  
spawn II;  
spawn III;  
sync;  
spawn IV;  
sync;
```

$$\begin{aligned} \text{Work: } T_1(n) &= 4T_1(n/2) + \Theta(1) \\ &= \Theta(n^2) \quad \text{— CASE 1} \end{aligned}$$

# Recursive Construction



*Cilk code*

```
spawn I;  
sync;  
spawn II;  
spawn III;  
sync;  
spawn IV;  
sync;
```

$$\begin{aligned} \textit{Span: } T_\infty(n) &= 3T_\infty(n/2) + \Theta(1) \\ &= \Theta(n^{\lg 3}) \quad \text{— CASE 1} \end{aligned}$$

# Analysis of Tableau Construction

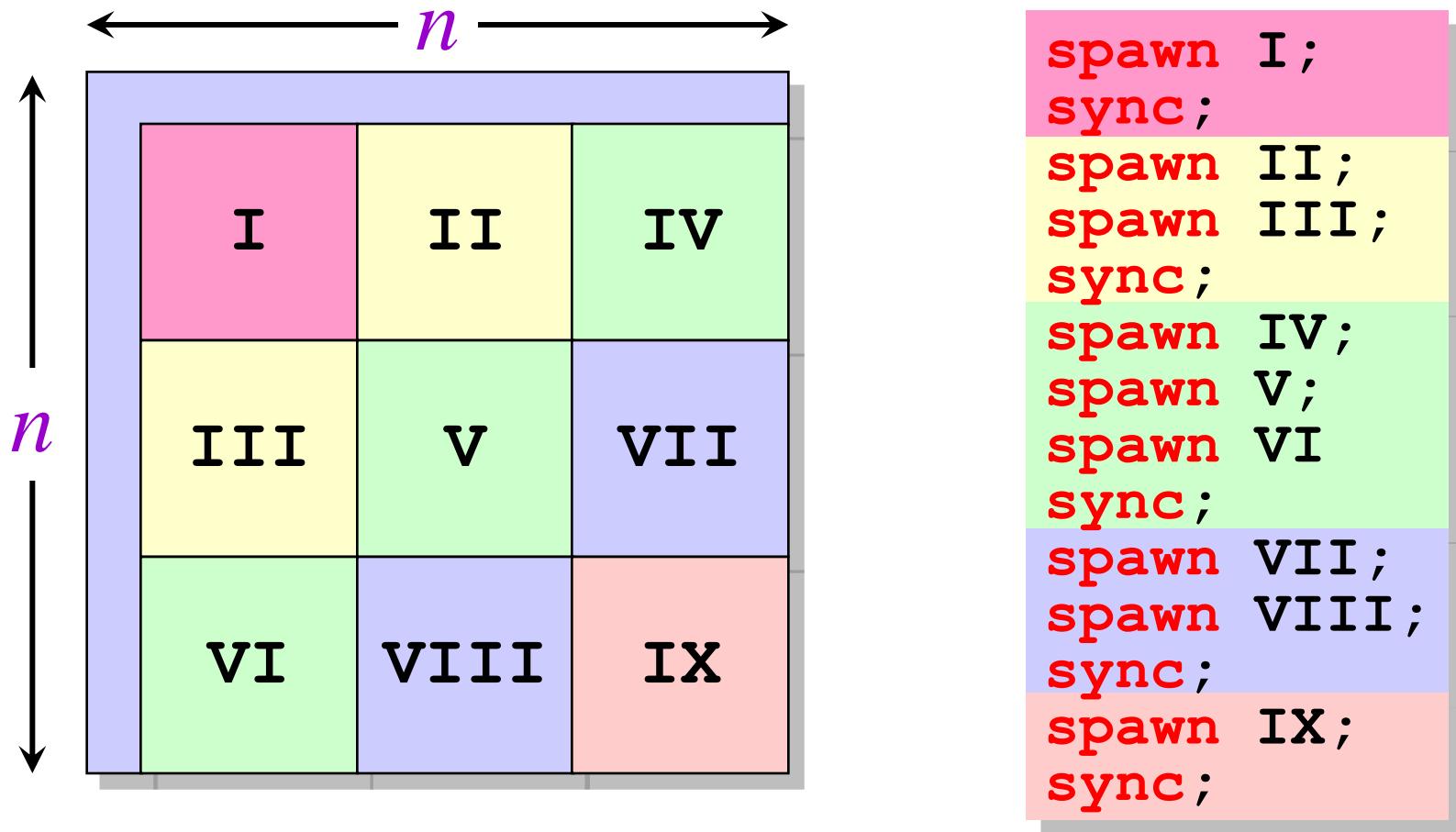
*Work:*  $T_1(n) = \Theta(n^2)$

*Span:*  $T_\infty(n) = \Theta(n^{\lg 3})$   
 $\approx \Theta(n^{1.58})$

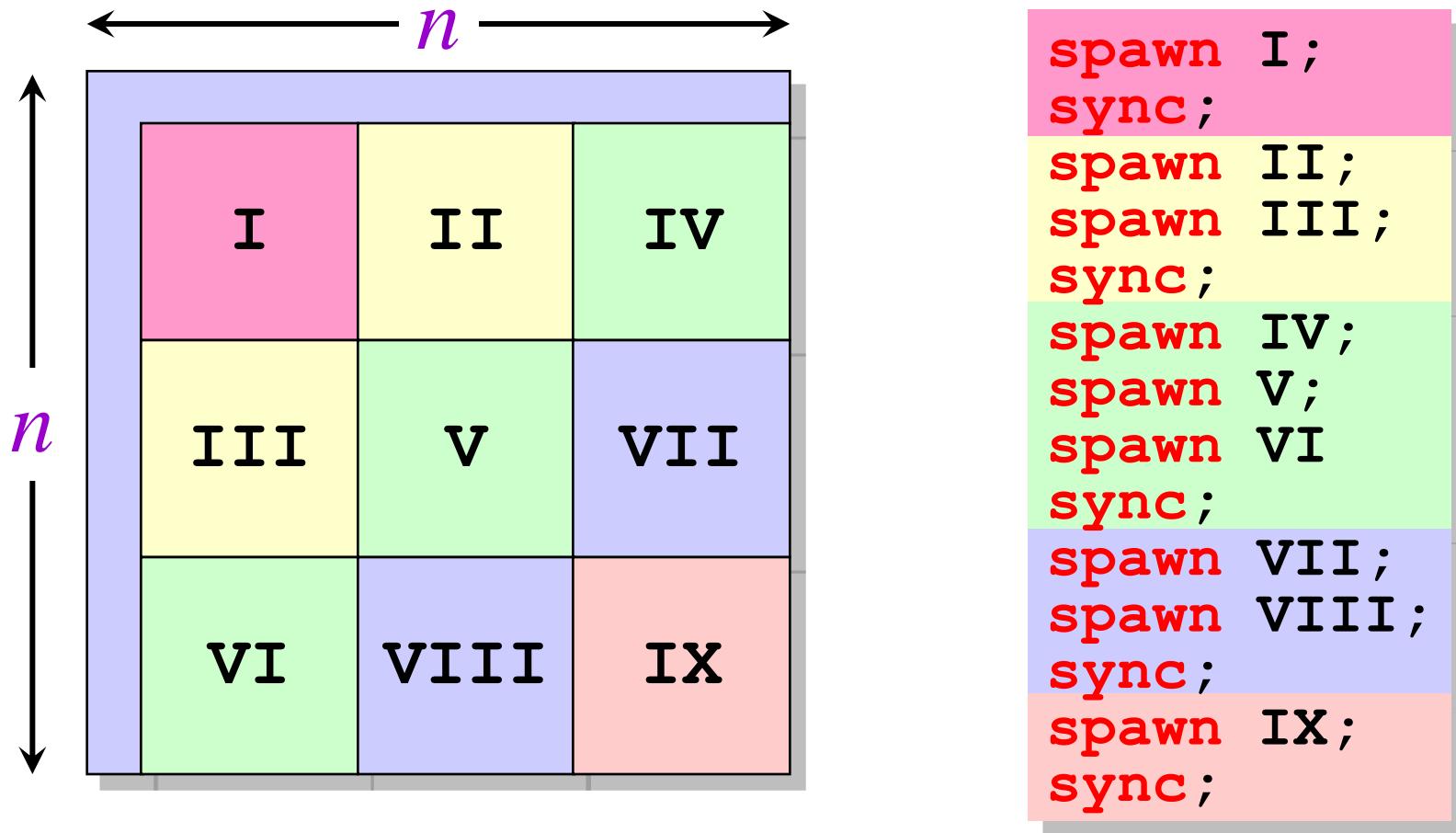
---

*Parallelism:*  $\frac{T_1(n)}{T_\infty(n)} \approx \Theta(n^{0.42})$

# A More-Parallel Construction

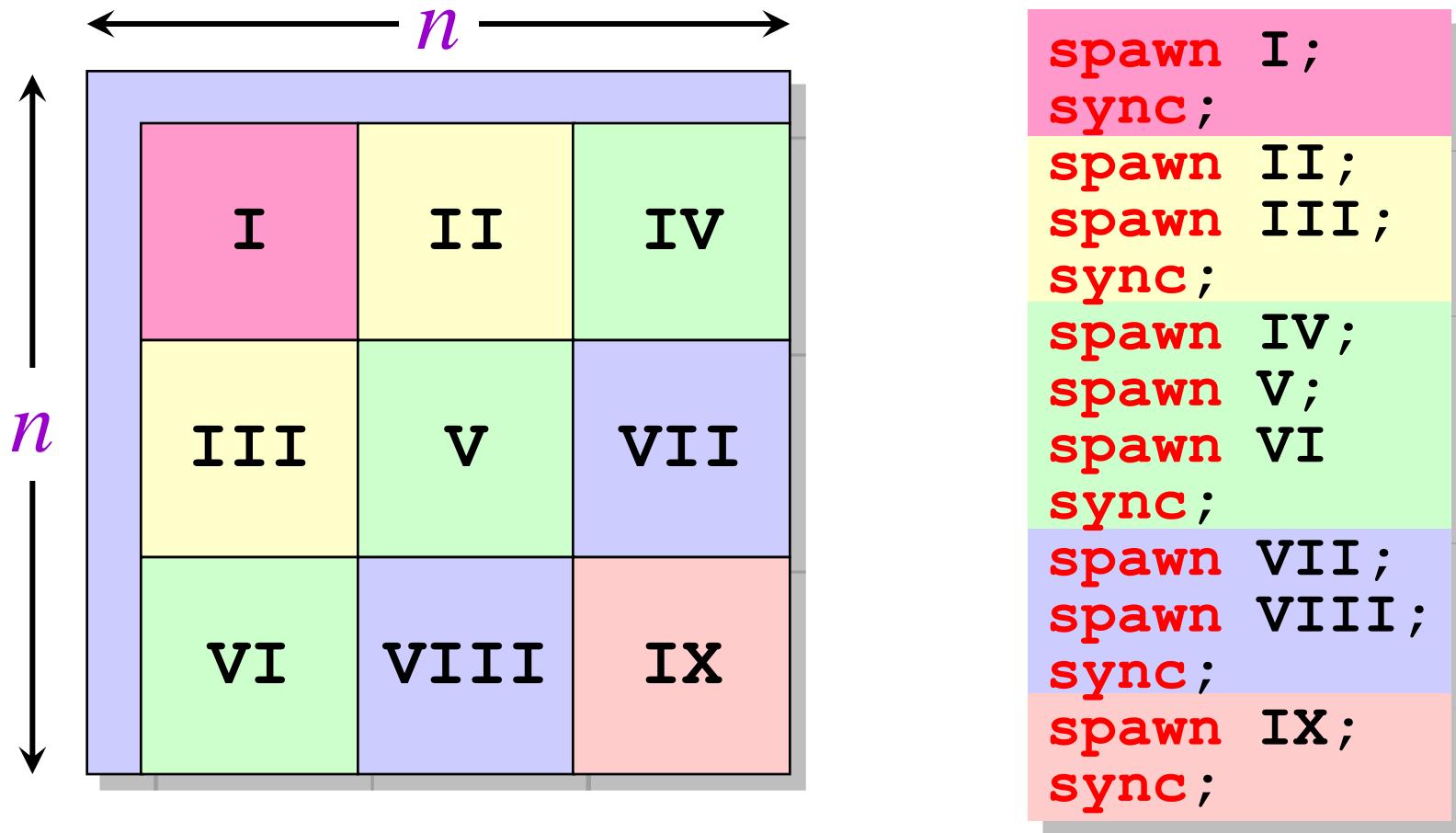


# A More-Parallel Construction



$$\begin{aligned} \text{Work: } T_1(n) &= 9T_1(n/3) + \Theta(1) \\ &= \Theta(n^2) \quad \text{— CASE 1} \end{aligned}$$

# A More-Parallel Construction



$$\begin{aligned} \textit{Span: } T_\infty(n) &= 5T_\infty(n/3) + \Theta(1) \\ &= \Theta(n^{\log_3 5}) \quad \text{— CASE 1} \end{aligned}$$

# Analysis of Revised Construction

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

*Span:*  $T_\infty(n) = \Theta(n^{\log_3 5})$   
 $\approx \Theta(n^{1.46})$

---

*Parallelism:*  $\frac{T_1(n)}{T_\infty(n)} \approx \Theta(n^{0.54})$

More parallel by a factor of

$$\Theta(n^{0.54})/\Theta(n^{0.42}) = \Theta(n^{0.12}) .$$

# Puzzle

*What is the largest parallelism that can be obtained for the tableau-construction problem using Cilk?*

- You may only use basic Cilk control constructs (**spawn**, **sync**) for synchronization.
- No locks, synchronizing through memory, etc.