6.172 Performance Engineering of Software Systems

LECTURE 8
ANALYSIS OF MULTITHREADED ALGORITHMS

Charles E. Leiserson

Divide-and-Conquer Recurrences

The Master Method

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

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

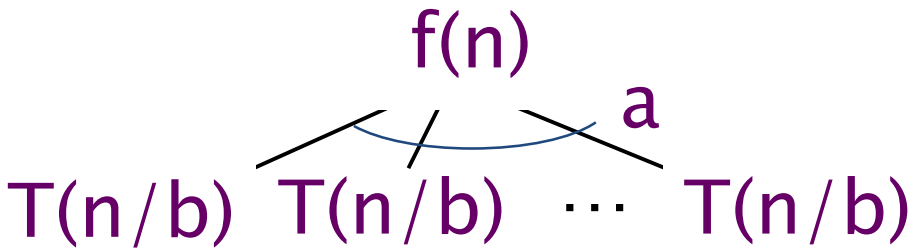
where $a \geq 1$, $b > 1$, and f is asymptotically positive.

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

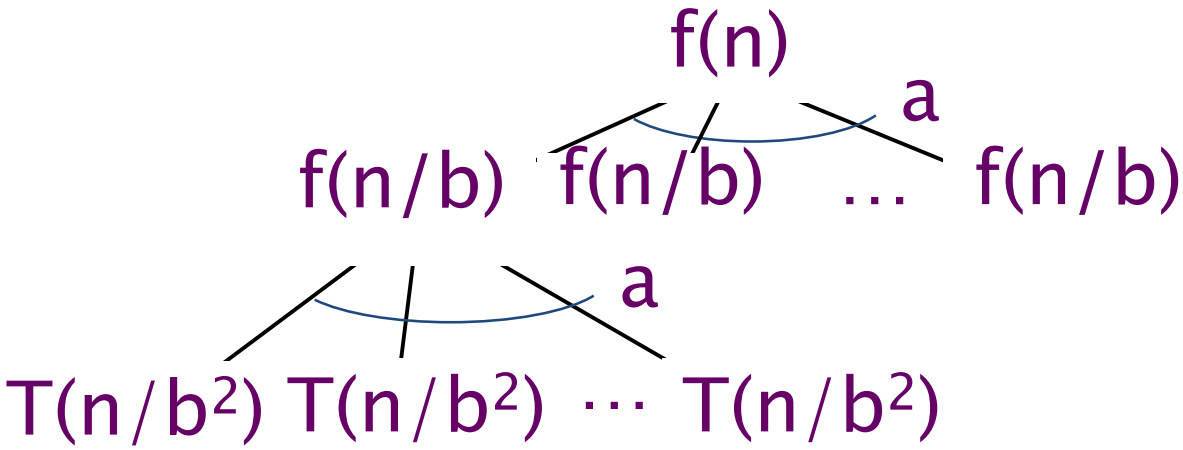
Recursion Tree: $T(n) = aT(n/b) + f(n)$

$T(n)$

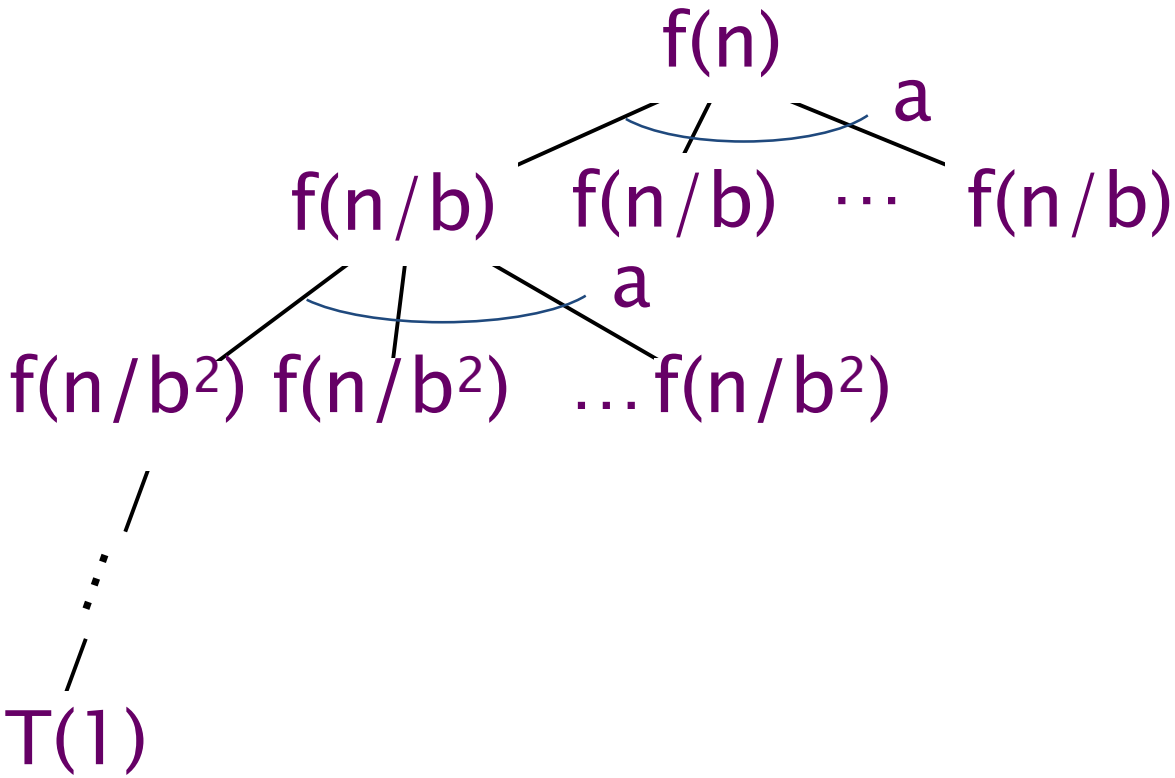
Recursion Tree: $T(n) = aT(n/b) + f(n)$



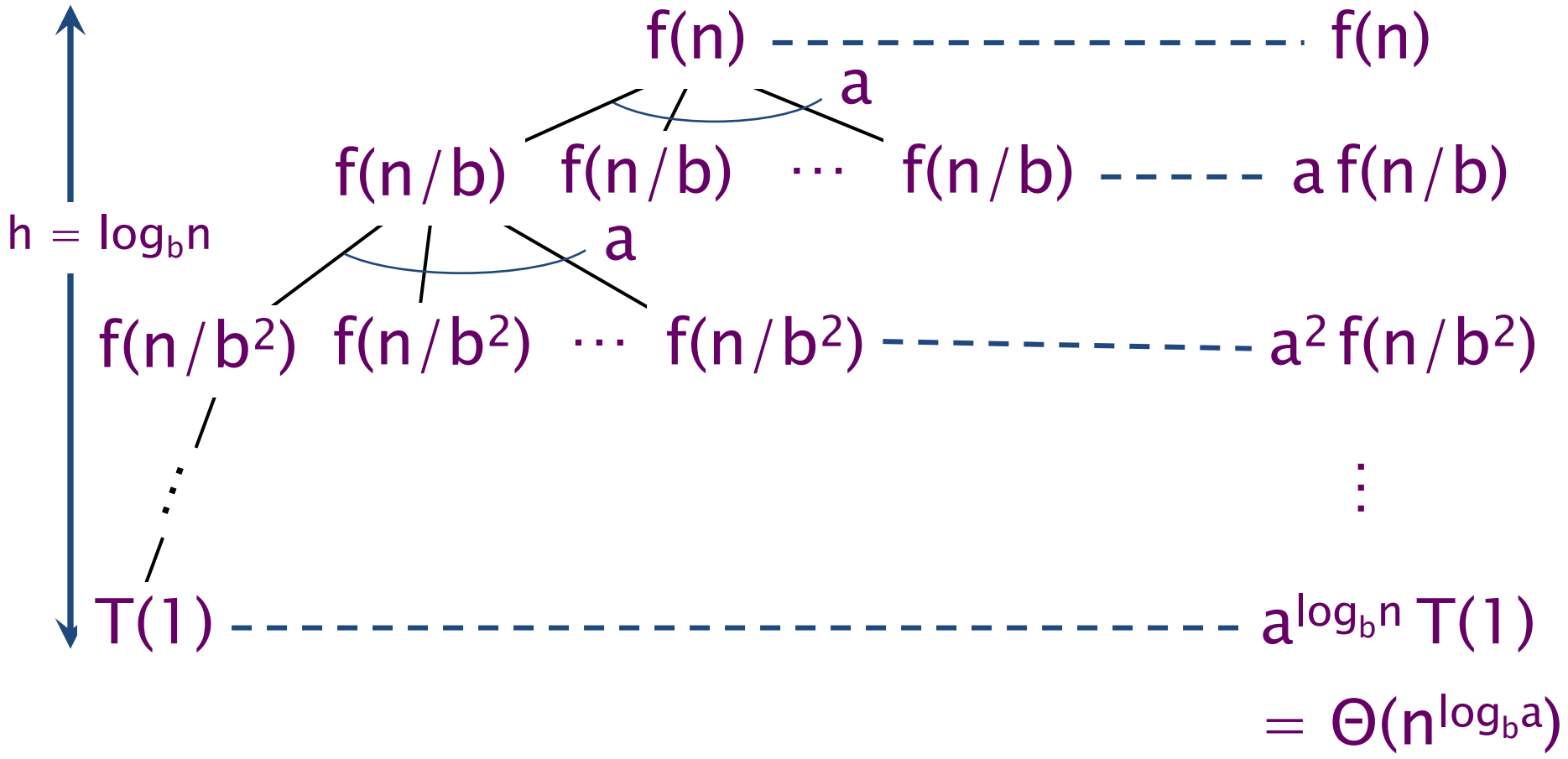
Recursion Tree: $T(n) = aT(n/b) + f(n)$



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

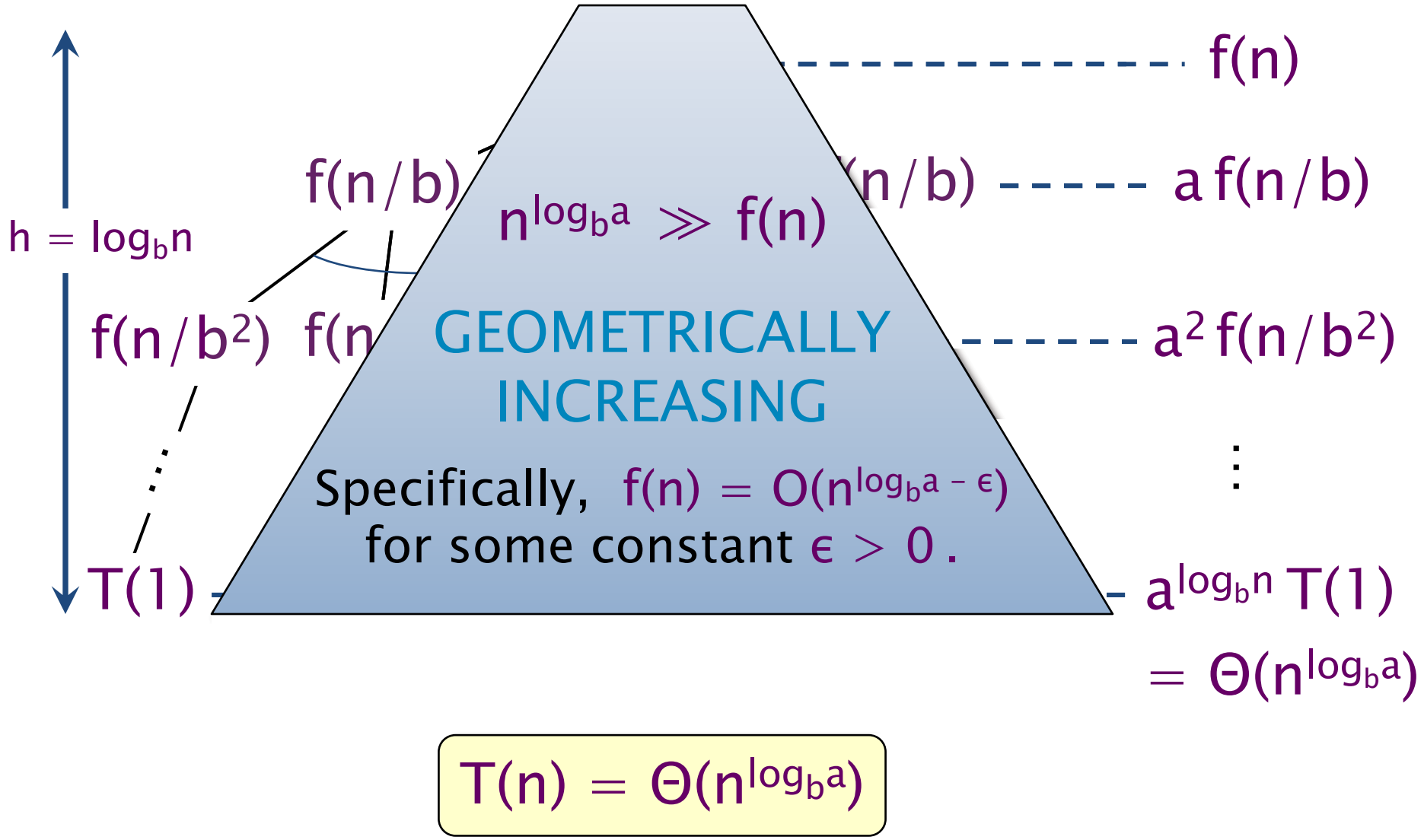


Recursion Tree: $T(n) = aT(n/b) + f(n)$



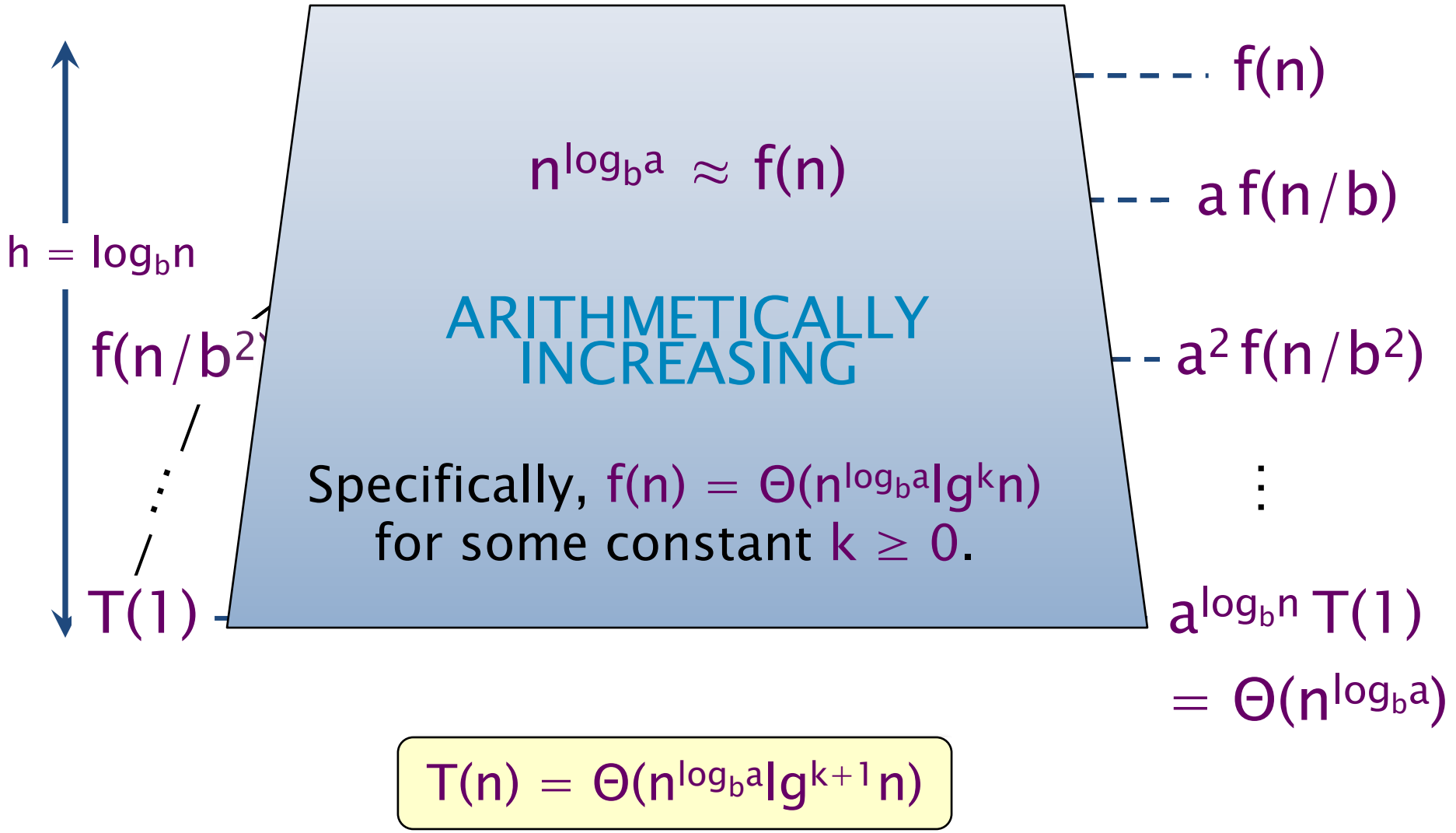
IDEA: Compare $n^{\log_b a}$ with $f(n)$.

Master Method — Case I



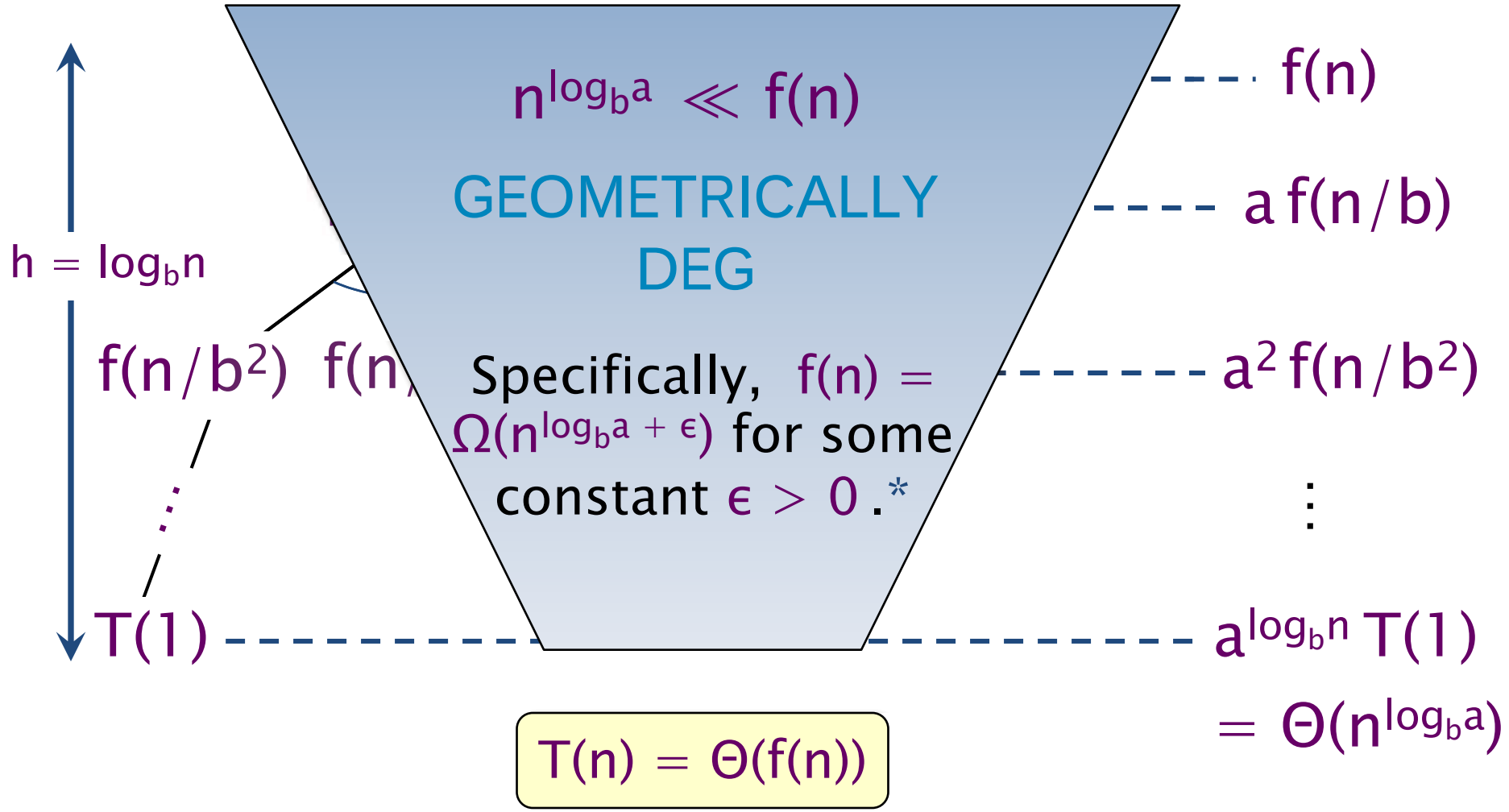
$$T(n) = \Theta(n^{\log_b a})$$

Master Method — CASE 2



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

Master Method — Case 3



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

Master-Method Cheat Sheet

Solve

$$\boxed{T(n) = a\,T(n/b) + f(n)\ ,}$$

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

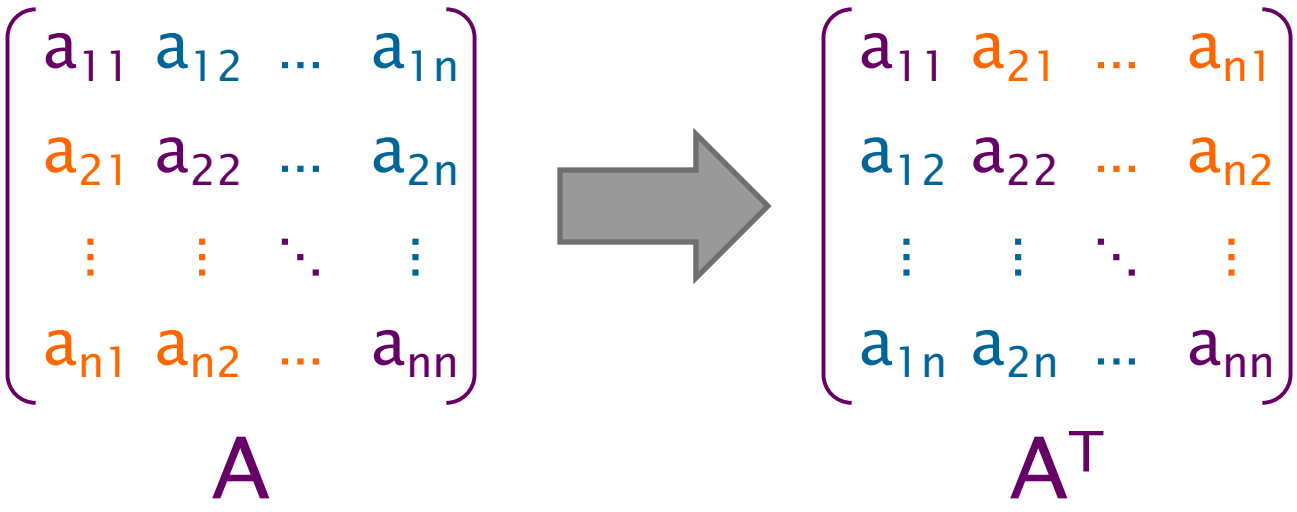
- $T(n) = 4T(n/2) + n$
  $n^{\log_b a} = n^2 \gg n \Rightarrow$ CASE 1: $T(n) = \Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$
  $n^{\log_b a} = n^2 = n^2 \lg^0 n \Rightarrow$ CASE 2: $T(n) = \Theta(n^2 \lg n)$.
- $T(n) = 4T(n/2) + n^3$
  $n^{\log_b a} = n^2 \ll n^3 \Rightarrow$ CASE 3: $T(n) = \Theta(n^3)$.
- $T(n) = 4T(n/2) + n^2/\lg n$
  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.

Cilk Loops

Loop Parallelism in Cilk

Example: In-place matrix transpose



The iterations of a `cilk_for` loop execute in parallel.

```
// indices run from 0, not 1
cilk_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;
  }
}
```

Implementation of Parallel Loops

```
// indices run from 0, not 1
cilk_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;
  }
}
```

Divide-and-conquer implementation

```
void recur(int lo, int hi)  //half open
{
  if (hi > lo + 1) {
    int mid = lo + (hi - lo)/2;
    cilk_spawn recur(lo, mid);
               recur(mid, hi);
    cilk_sync;
    return;
  }
  int i = lo;
  for (int j=0; j<i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
⋮
recur(1, n);
```

The Tapir/LLVM compiler implements `cilk_for` loops this way at optimization level `-O1` or higher.

Implementation of Parallel Loops

```
// indices run from 0, not 1
cilk_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;
  }
}
```

cilk_for
loop control

```
void recur(int lo, int hi) //half open
{
  if (hi > lo + 1) {
    int mid = lo + (hi - lo)/2;
    cilk_spawn recur(lo, mid);
               recur(mid, hi);
    cilk_sync;
    return;
  }
  int i = lo;
  for (int j=0; j<i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
⋮
recur(1, n);
```

Divide-and-conquer implementation

lifted
loop body

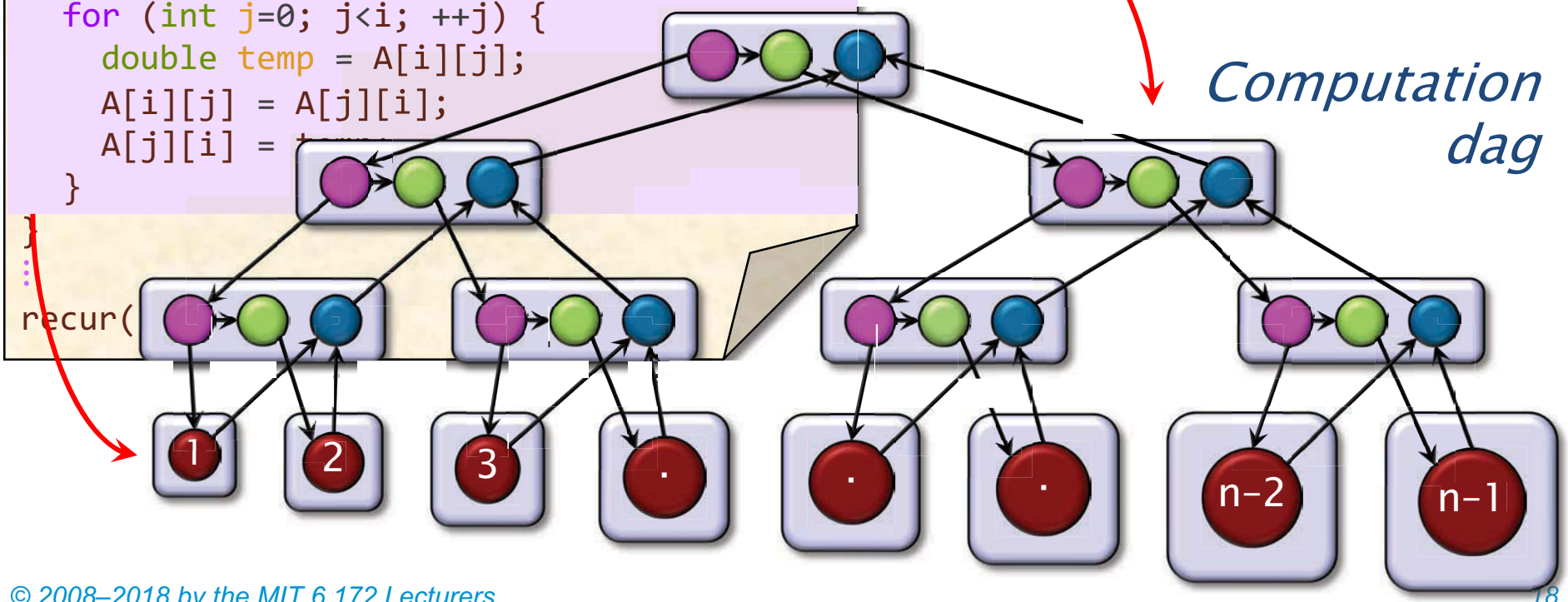
Execution of Parallel Loops

```
void recur(int lo, int hi)  //half open
{
  if (hi > lo + 1) {
    int mid = lo + (hi - lo)/2;
    cilk_spawn recur(lo, mid);
               recur(mid, hi);
    cilk_sync;
    return;
  }
  int i = lo;
  for (int j=0; j<i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
...
recur(
```

Divide-and-conquer implementation

`cilk_for` loop control

Computation dag

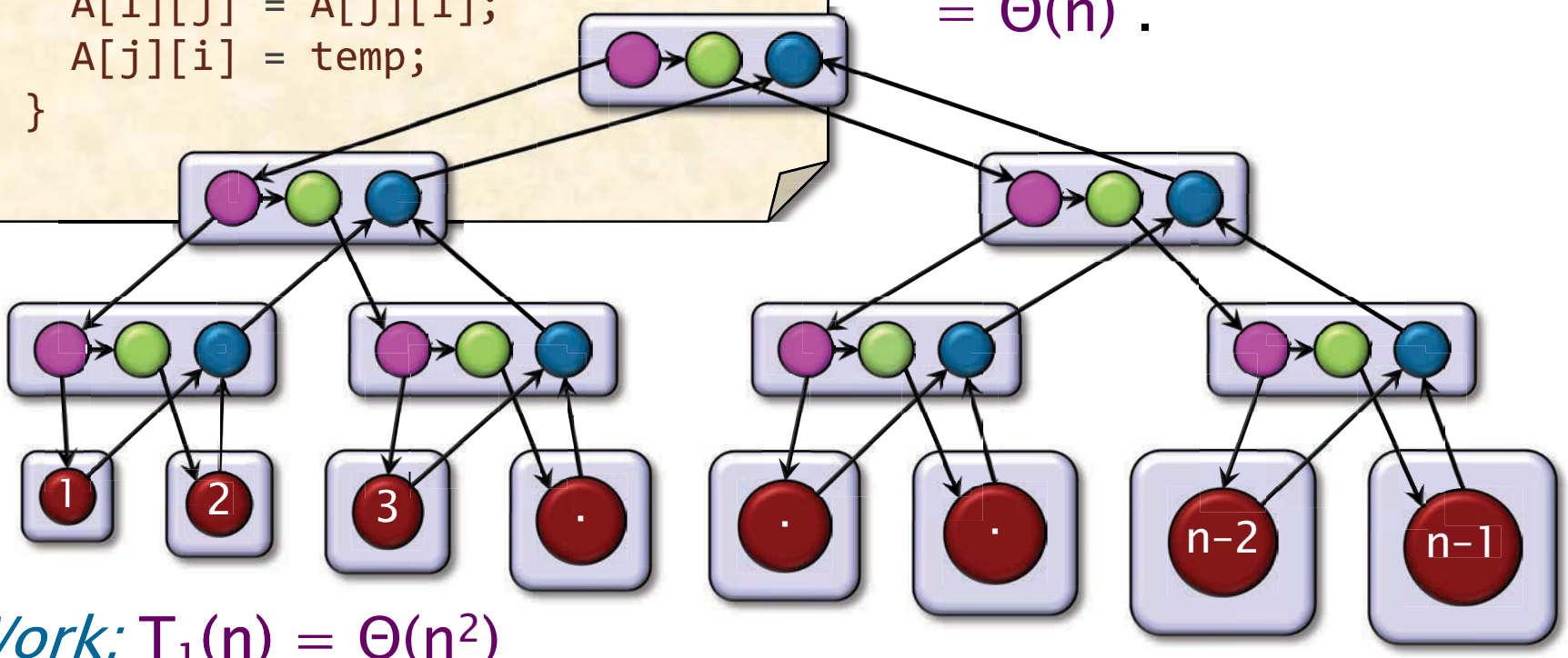


Analysis of Parallel Loops

```
// indices run from 0, not 1
cilk_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;
  }
}
```

Span of loop control $= \Theta(\lg n)$.

Max span of body $= \Theta(n)$.



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

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

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

Analysis of Nested Parallel Loops

```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
  cilk_for (int j=0; j<i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
```

Span of outer loop control $= \Theta(\lg n)$.

Max span of inner loop control $= \Theta(\lg n)$.

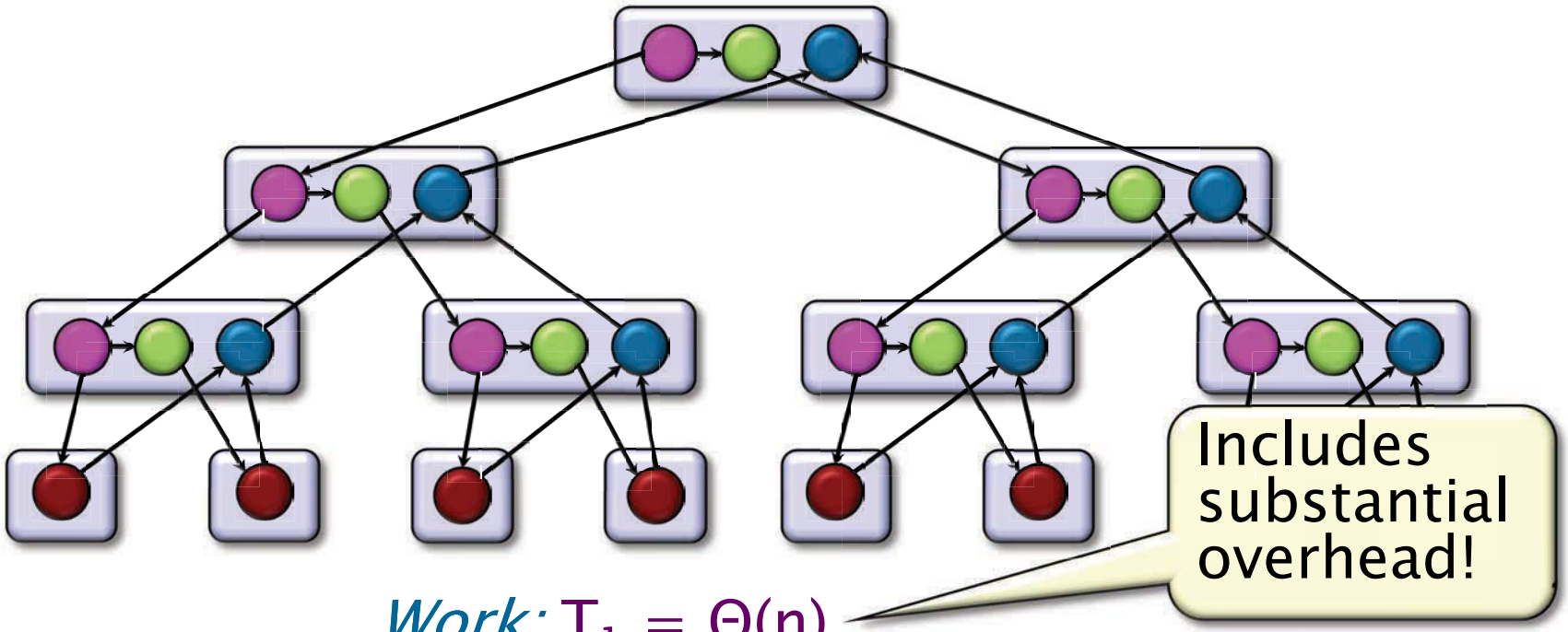
Span of body $= \Theta(1)$.

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

A Closer Look at Parallel Loops

Vector addition

```
cilk_for (int i=0; i<n; ++i) {
  A[i] += B[i];
}
```



Work: $T_1 = \Theta(n)$

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

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

Coarsening Parallel Loops

```
#pragma cilk grainsize G
cilk_for (int i=0; i<n; ++i) {
  A[i] += B[i];
}
```

Implementation with coarsening

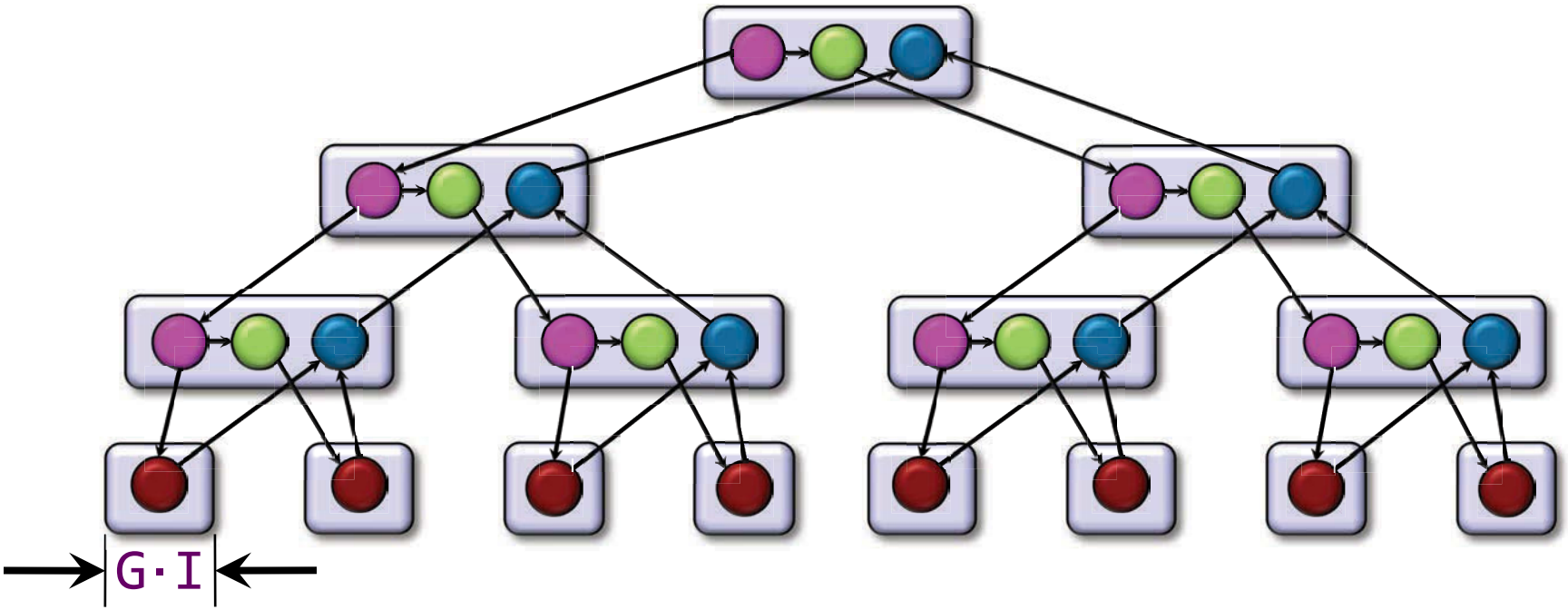
If a grainsize pragma is not specified, the Cilk runtime system makes its best guess to minimize overhead.

```
void recur(int lo, int hi) { //half open
  if (hi > lo + G) {
    int mid = lo + (hi - lo)/2;
    cilk_spawn recur(lo, mid);
               recur(mid, hi);
    cilk_sync;
    return;
  }
  for (int i=lo; i<hi; ++i) {
    A[i] += B[i];
  }
}
⋮
recur(0, n);
```

Loop Grain Size

Vector addition

```
#pragma cilk grainsize G
cilk_for (int i=0; i<n; ++i) {
  A[i] += B[i];
}
```

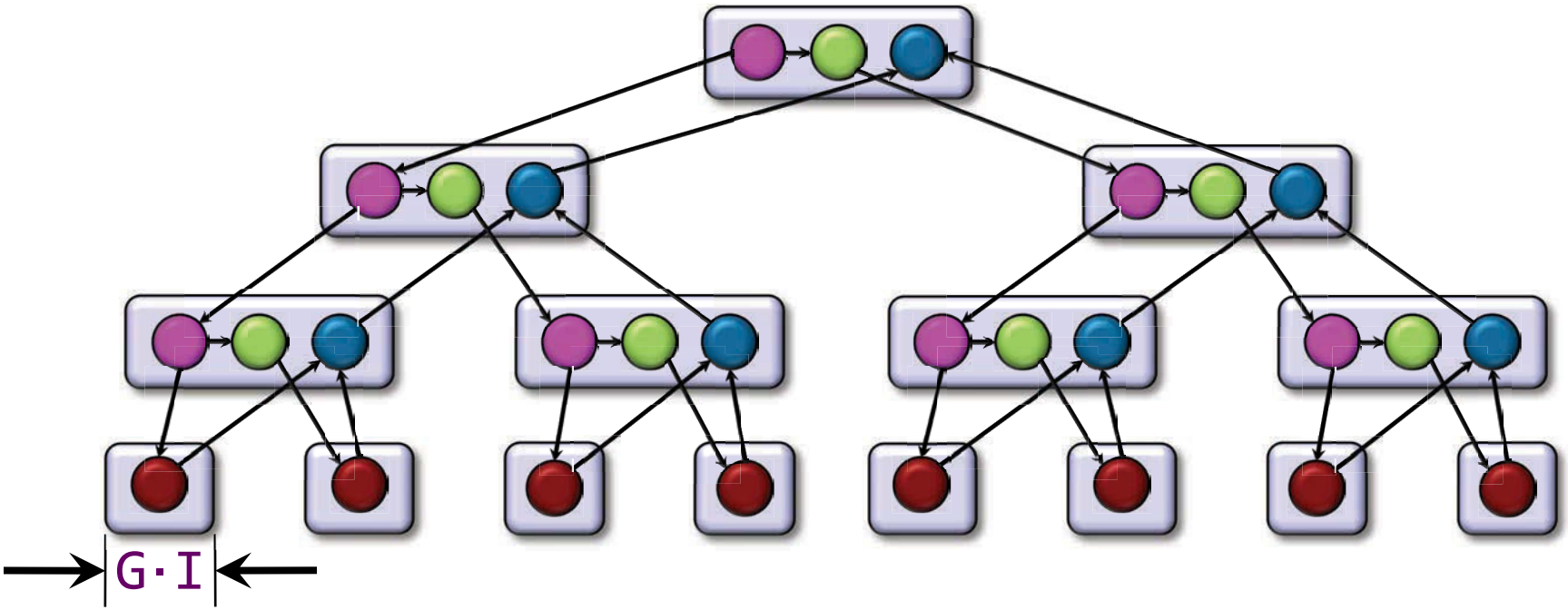


Let I be the time for one iteration of the loop body.
Let S be the time to perform a spawn and return.

Loop Grain Size

Vector addition

```
#pragma cilk grainsize G
cilk_for (int i=0; i<n; ++i) {
  A[i] += B[i];
}
```



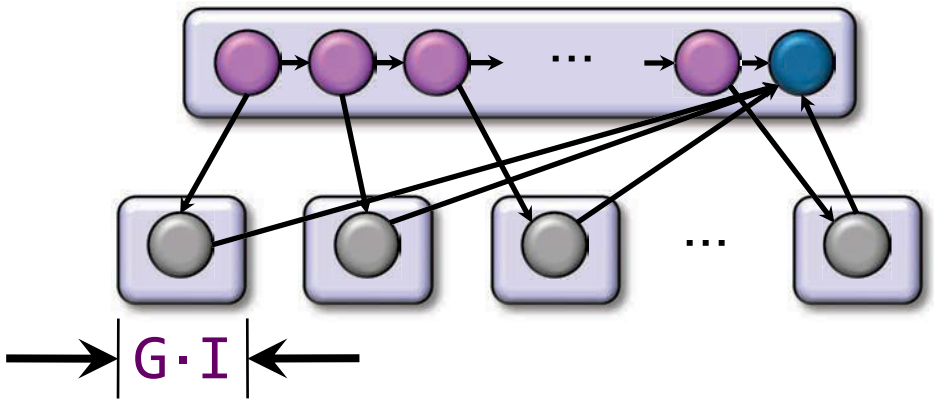
Work: $T_1 = n \cdot I + (n/G - 1) \cdot S$

Span: $T_\infty = G \cdot I + \lg(n/G) \cdot S$

Want $G \gg S/I$ and G small.

Another Implementation

```
void vadd (double *A, double *B, int n){
  for (int i=0; i<n; i++) A[i] += B[i];
}
⋮
for (int j=0; j<n; j+=G) {
  cilk_spawn vadd(A+j, B+j, MIN(G,n-j));
}
cilk_sync;
```



Assume that $G = 1$.

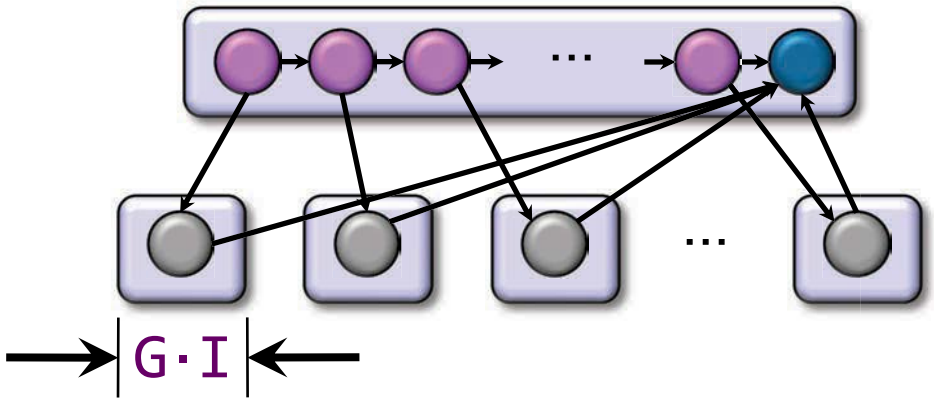
Work: $T_1 = \Theta(n)$

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

Parallelism: $T_1/T_\infty = \Theta(1)$

Another Implementation

```
void vadd (double *A, double *B, int n){
  for (int i=0; i<n; i++) A[i] += B[i];
}
⋮
for (int j=0; j<n; j+=G) {
  cilk_spawn vadd(A+j, B+j, MIN(G,n-j));
}
cilk_sync;
```



Choose $G = \sqrt{n}$ to minimize.

Analyze in terms of G:

Work: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(G + n/G) = \Theta(\sqrt{n})$

Parallelism: $T_1/T_\infty = \Theta(\sqrt{n})$

Quiz on Parallel Loops

Question: Let $P \ll n$ be the number of workers on the system. How does the parallelism of Code A compare to the parallelism of Code B? (Differences highlighted.)

Code A

```
#pragma cilk grainsize 1
cilk_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: $T_\infty = \Theta(\lg(n/32) + 32)$
$= \Theta(\lg n)$

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

Code B

```
#pragma cilk grainsize 1
cilk_for (int i=0; i<n; i+=n/P) {
  for (int j=i; j<MIN(i+   , n); ++j)
    A[j] += B[j];
}
```

Work: $T_1 = \Theta(n)$

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

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

Quiz on Parallel Loops

Question: Let $P \ll n$ be the number of workers on the system. How does the parallelism of Code A compare to the parallelism of Code B? (Differences highlighted.)

Code A

```
#pragma cilk grainsize 1
cilk_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: $T_\infty = \Theta(\lg(n/32) + 32)$
$= \Theta(\lg n)$

Parallelism:
$T_1/$

RELATIVELY PUNY!
Want $T_1/T_\infty \gg P$.

Code B

```
#pragma cilk grainsize 1
cilk_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:

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

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

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.

Do this:

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

Not this:

```
for (int i=0; i<n; ++i) {
  cilk_spawn foo(i);
}
cilk_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*.

Do this:

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

Not this:

```
for (int j=0; j<n; ++j) {
  cilk_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

```
cilk_for (int i=0; i<n; ++i) {
  cilk_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 1000×1000 matrices, parallelism $\approx (10^3)^2 = 10^6$.

Recursive Matrix Multiplication

Divide and conquer — uses cache more efficiently, as we'll see later in the term.

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

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

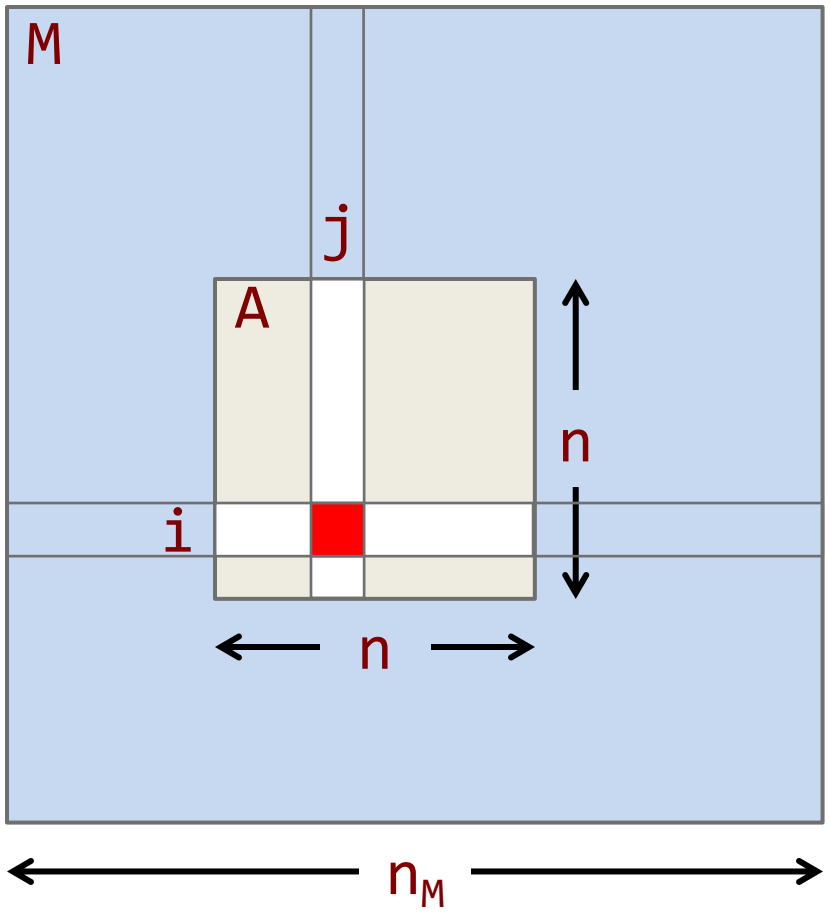
8 multiplications of $n/2 \times n/2$ matrices.
1 addition of $n \times n$ matrices.

Representation of Submatrices

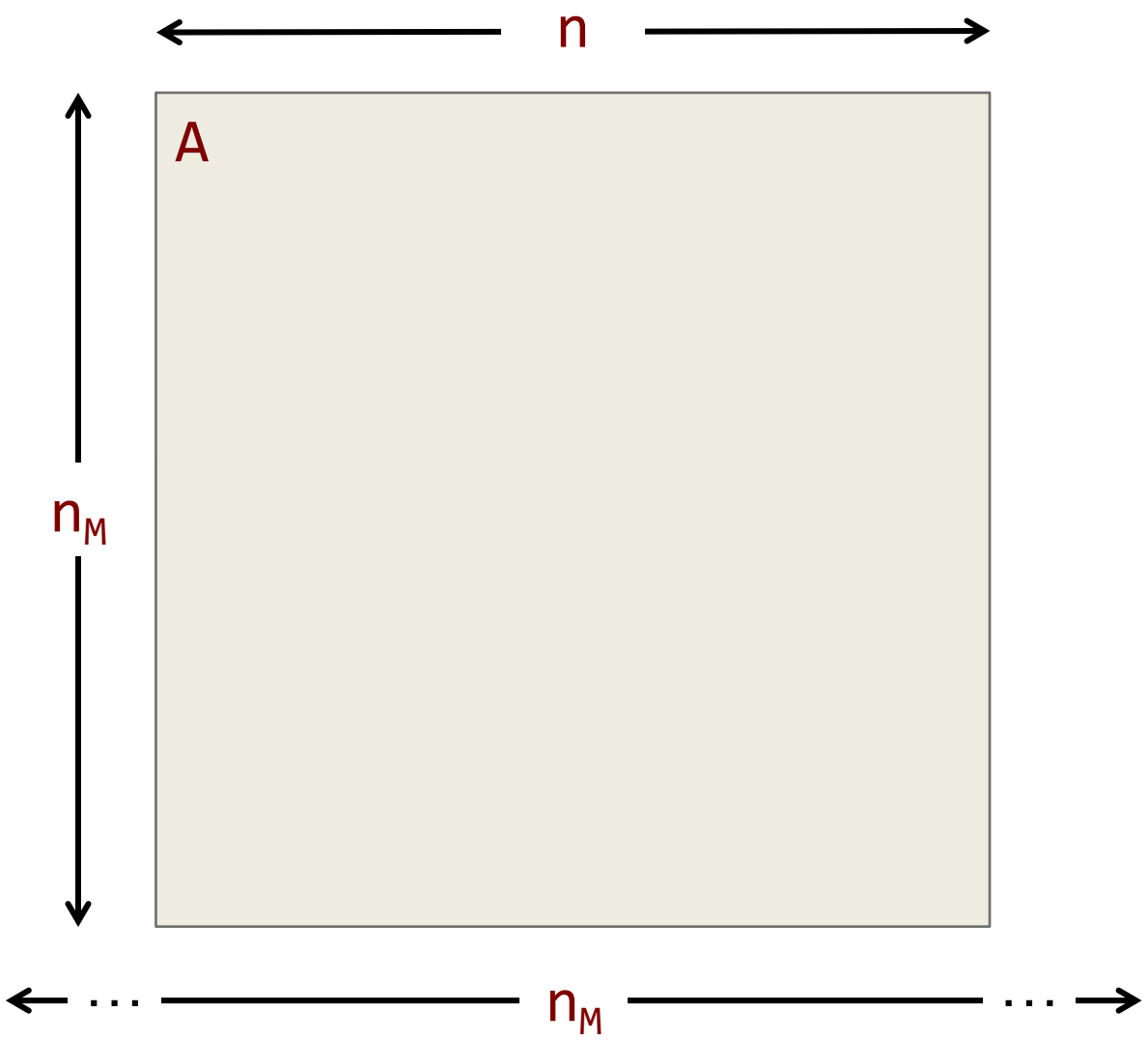
Row-major layout

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]`.

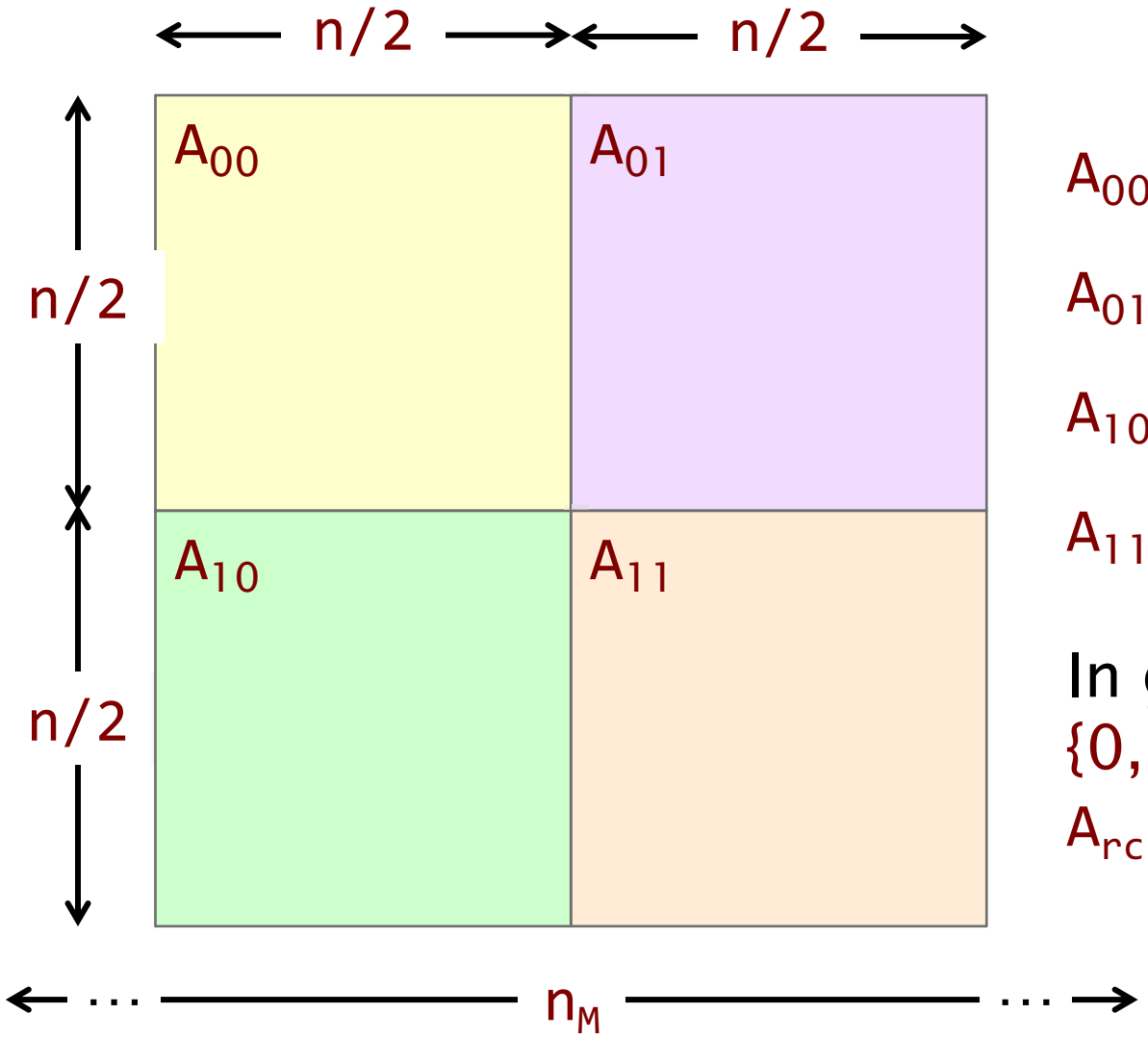
Note: The dimension `n` does not enter into the calculation, although it does matter for bounds checking of `i` and `j`.



Divide-and-Conquer Matrices



Divide-and-Conquer Matrices



$A_{00} = A$

$A_{01} = A + (n/2)$

$A_{10} = A + n_M*(n/2)$

$A_{11} = A + (n_M+1)*(n/2)$

In general, for $r, c \in \{0,1\}$, we have

$A_{rc} = A + (r*n_M+c)*(n/2)$

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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

The row sizes 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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_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 *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

The function adds the matrix product A * B to 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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Assert that n is a power of 2.

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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Coarsen the leaves of the recursion to lower the overhead.

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)
{ // C += A * B
  assert((n & (-n)) == 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
    cilk_spawn mm_dac(X
    cilk_spawn mm_dac(X
    cilk_spawn mm_dac(X
    cilk_spawn mm_dac(X
    cilk_spawn mm_dac(X
    cilk_spawn mm_dac(X
    cilk_spawn mm_dac(X
    cilk_spawn mm_dac(X
               mm_dac(X
    cilk_sync;
    m_add(C, n_C, D, n_
    free(D);
  }
}
```

Coarsen the leaves of the recursion to lower the overhead.

```
void mm_base(double *restrict C, int n_C,
             double *restrict A, int n_A,
             double *restrict B, int n_B,
             int n)
{ // C += A * B
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      for (int k = 0; k < n; ++k) {
        C[i*n_C + j] += A[i*n_A + k] * B[k*n_B + 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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Allocate a temporary $n \times n$ array 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)
{ // C += A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * si
    assert(D != NULL);
#define n_D n
#define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

A clever macro to compute indices of submatrices.

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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

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

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)
{ // C += A * B
  assert((n & (-n)) == 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)*(
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Wait for all spawned subcomputations to complete.

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)
{ // C += A * B
  assert((n & (-n)) == 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
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Add the temporary matrix D into the 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)
{ // C += A * B
  assert((n & (-n)) == 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
    cilk_spawn mm_dac(X(C,0,0),
    cilk_spawn mm_dac(X(C,0,1),
    cilk_spawn mm_dac(X(C,1,0),
    cilk_spawn mm_dac(X(C,1,1),
    cilk_spawn mm_dac(X(D,0,0),
    cilk_spawn mm_dac(X(D,0,1),
    cilk_spawn mm_dac(X(C,1,0),
               mm_dac(X(D,1,1),
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Add the temporary matrix D into the output matrix C.

```
void m_add (double *restrict C, int n_C,
            double *restrict D, int n_D,
            int n)
{ // C += D
  cilk_for (int i = 0; i < n; ++i) {
    cilk_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)
{ // C += A * B
  assert((n & (-n)) == 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)*(
    cilk_spawn mm_dac(X(C,0,0), n_C, X(
    cilk_spawn mm_dac(X(C,0,1), n_C,
    cilk_spawn mm_dac(X(C,1,0), n_
    cilk_spawn mm_dac(X(C,1,1),
    cilk_spawn mm_dac(X(D,0,
    cilk_spawn mm_dac(X(D,
    cilk_spawn mm_dac(
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Clean up, and then return.

Analysis of Matrix Addition

```
void m_add (double *restrict C, int n_C,
            double *restrict D, int n_D,
            int n)
{ // C += D
  cilk_for (int i = 0; i < n; ++i) {
    cilk_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: $A_\infty(n) = \Theta(\lg n)$

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)
{ // ...
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

CASE 1

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

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

Work: $M_1(n) = 8M_1(n/2) + A_1(n) + \Theta(1)$

$= 8M_1(n/2) + \Theta(n^2)$

$= \Theta(n^3)$

Span of Matrix Multiplication

maximum

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // ...
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

CASE 2

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

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

Span: $M_\infty(n) = M_\infty(n/2) + A_\infty(n) + \Theta(1)$

$= M_\infty(n/2) + \Theta(\lg n)$

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

Parallelism of Matrix Multiply

$$\textit{Work:} \quad M_1(n) = \Theta(n^3)$$

$$\textit{Span:} \quad M_\infty(n) = \Theta(\lg^2 n)$$

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

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

Temporaries

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cil
    ci
    ci
    ci
    ci
    m_
    fre
  }
}
```

IDEA

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

How to Avoid the Temporary?

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_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);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

No-Temp Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C
            double *restrict A, int n_
            double *restrict B, int n_
            int n)
{ // C += A * B
  assert((n & (-n)) == 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
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, 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);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
  }
}
```

Do 4 subproblems, sync, and then do 4 more subproblems.

No-Temp 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)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, 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);
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(C,1,1), n_C, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
  }
}
```

Reuse C without racing.

Saves space, but at what expense?

Work of No-Temp Multiply

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, 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
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1 , n_
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1), n_
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1 , n_
               mm_dac(X(C,1,1), n_C, X(A,1,1 , n_
    cilk_sync;
  }
}
```

CASE 1

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

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

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

$= \Theta(n^3)$

Span of No-Temp Multiply

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C += A * B
  assert((n & (-n)) == 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))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, 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),
    cilk_sync;
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,1),
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,1),
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,1),
               mm_dac(X(C,1,1), n_C, X(A,1,1),
    cilk_sync;
  }
}
```

max

max

CASE 1

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

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

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

$= \Theta(n)$

Parallelism of No-Temp Multiply

$$\textit{Work:} \quad M_1(n) = \Theta(n^3)$$

$$\textit{Span:} \quad M_\infty(n) = \Theta(n)$$

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

For 1000×1000 matrices,
parallelism $\approx (10^3)^2 = 10^6$.

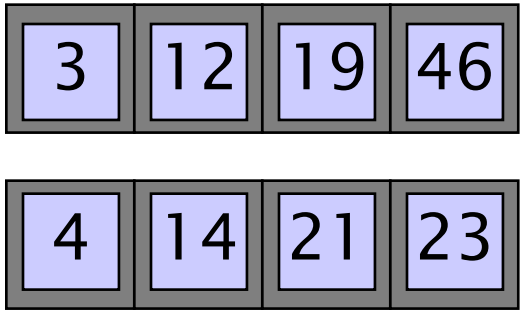
Faster in practice!

Merge Sort

Merging Two Sorted Arrays

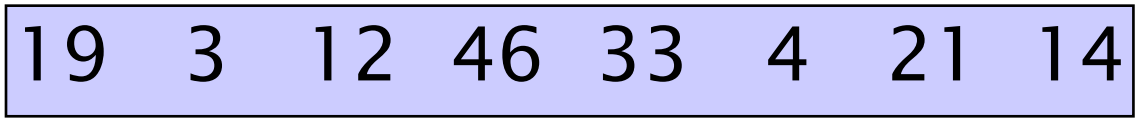
```
void merge(int *C, int *A, int na, int *B, int nb) {
  while (na>0 && nb>0) {
    if (*A <= *B) {
      *C++ = *A++; na--;
    } else {
      *C++ = *B++; nb--;
    }
  }
  while (na>0) {
    *C++ = *A++; na--;
  }
  while (nb>0) {
    *C++ = *B++; nb--;
  }
}
```

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



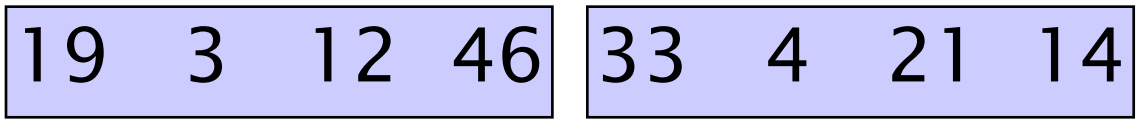
Merge Sort

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



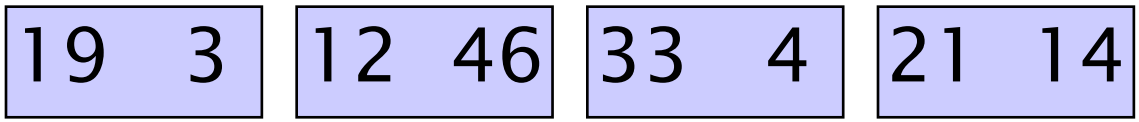
Merge Sort

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



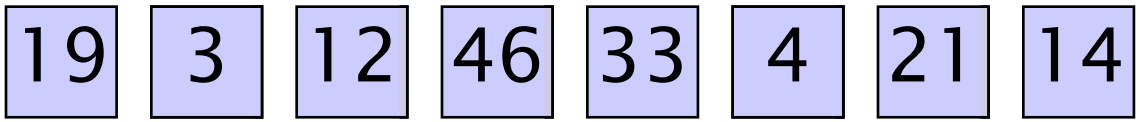
Merge Sort

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



Merge Sort

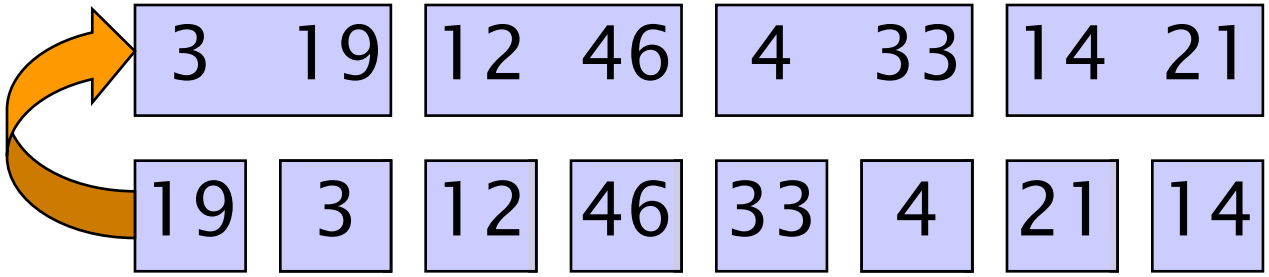
```
void merge_sort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int C[n];
    cilk_spawn merge_sort(C, A, n/2);
               merge_sort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```



Merge Sort

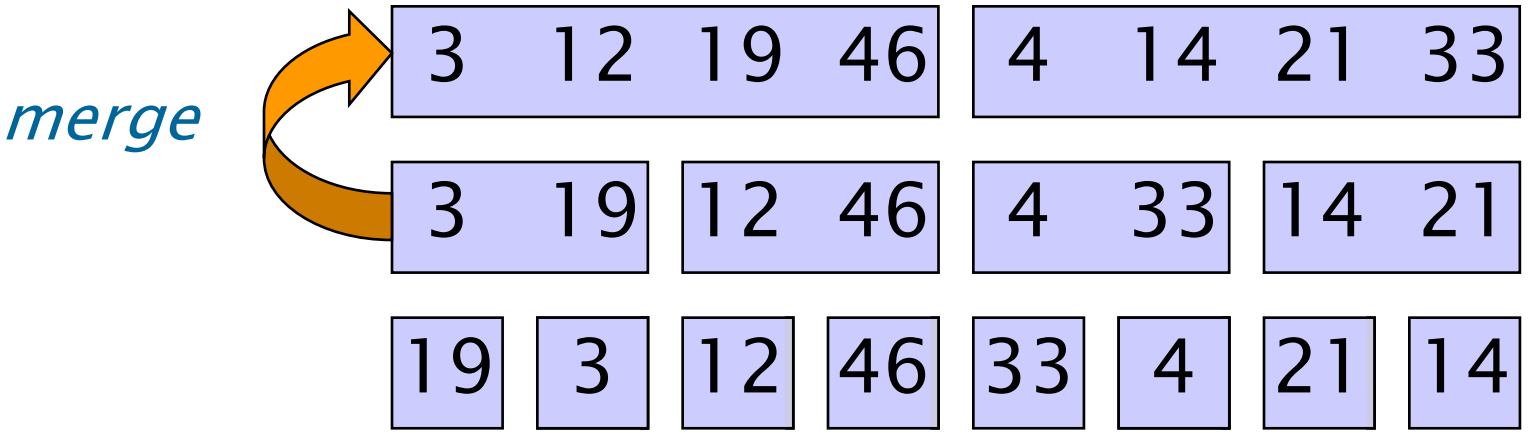
```
void merge_sort(int *B, int *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    int C[n];
    cilk_spawn merge_sort(C, A, n/2);
               merge_sort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    merge(B, C, n/2, C+n/2, n-n/2);
  }
}
```

merge



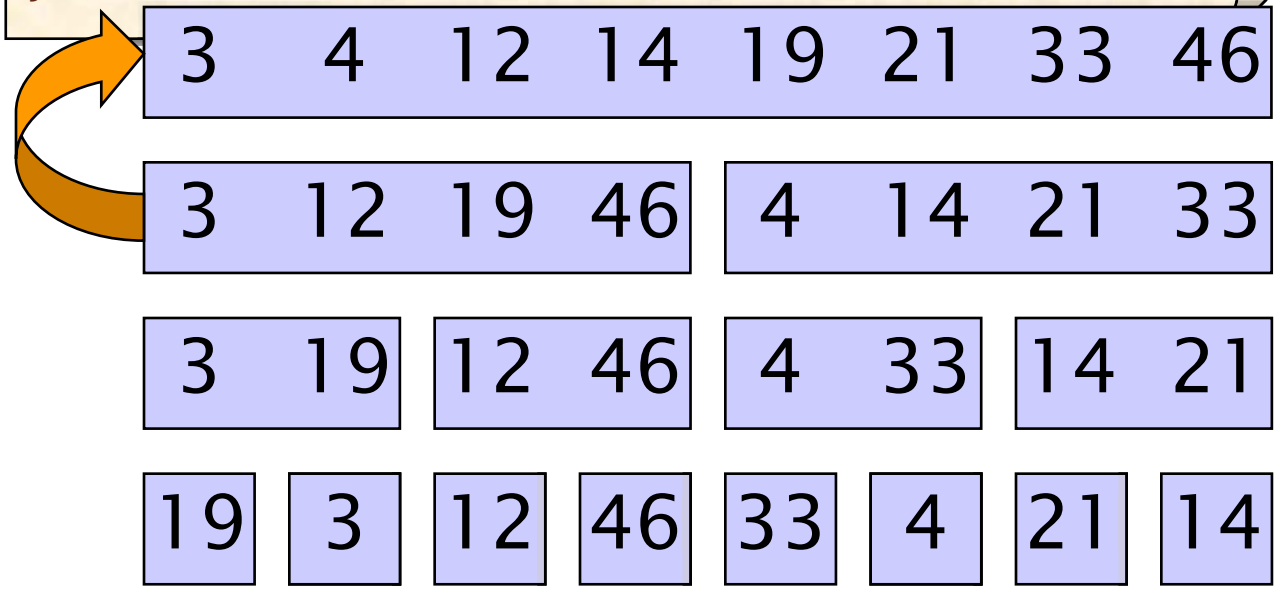
Merge Sort

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



Merge Sort

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



merge

Work of Merge Sort

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

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

$= \Theta(n \lg n)$

CASE 2

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

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

Span of Merge Sort

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

Span: $T_\infty(n) = T_\infty(n/2) + \Theta(n)$

$= \Theta(n)$

CASE 3

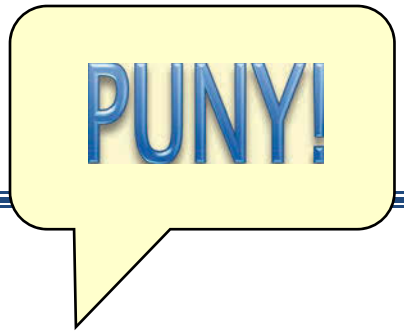
$n^{\log_b a} = n^{\log_2 1} = 1$

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

Parallelism of Merge Sort

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

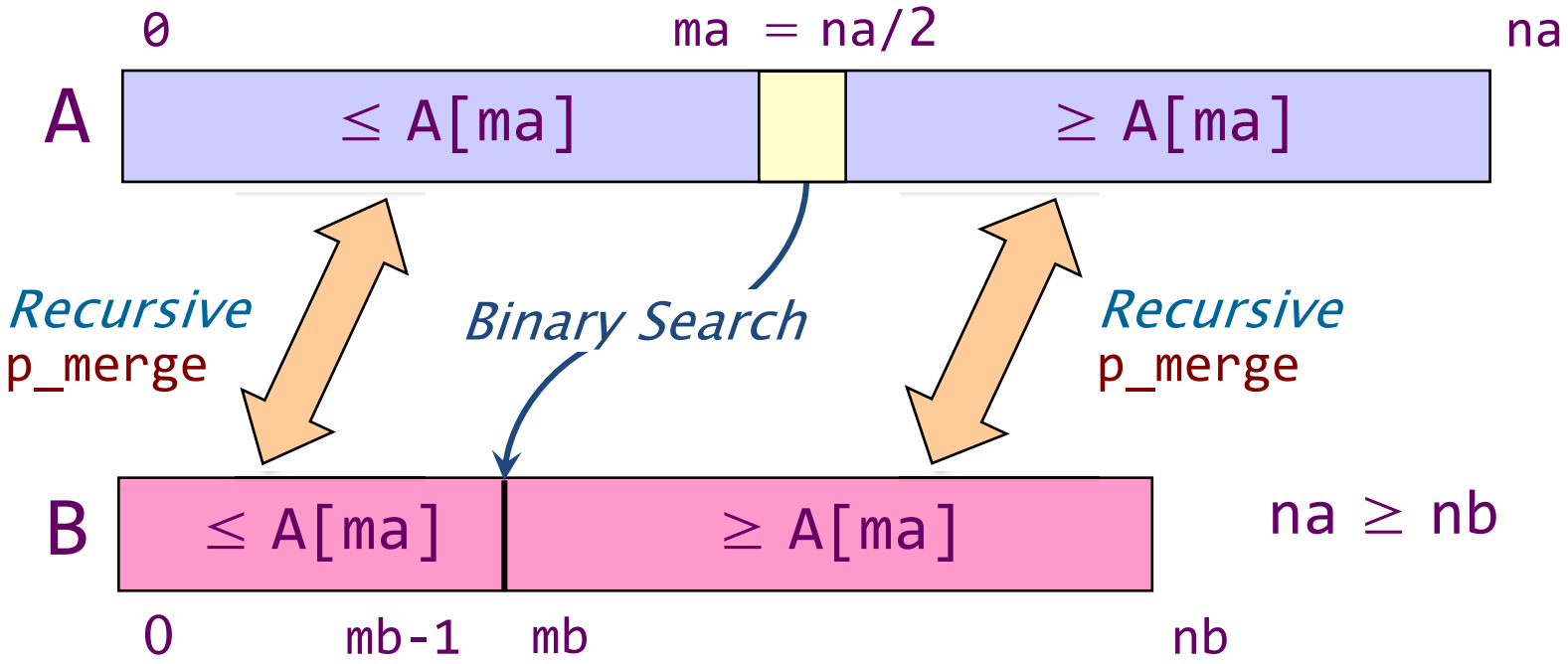
Span: $T_\infty(n) = \Theta(n)$



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

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

CASE 2

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

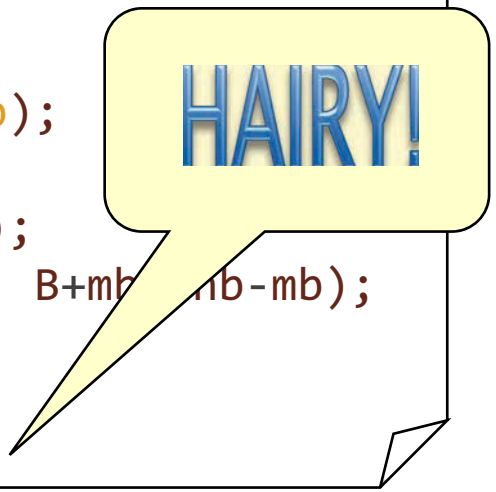
$f(n) = \Theta(n^{\log_b a} \lg^1 n)$

Span: $T_\infty(n) \le T_\infty(3n/4) + \Theta(\lg n)$

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

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);
  } else if (na==0) {
      return;
  } else {
   int ma = na/2;
   int mb = binary_search(A[ma], B, nb);
   C[ma+mb] = A[ma];
   cilk_spawn p_merge(C, A, ma, B, mb);
   p_merge(C+ma+mb+1, A+ma+1, na-ma-1, B+mb, nb-mb);
   cilk_sync;
  }
}
```



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

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 $1/4 \le \alpha \le 3/4$.

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

$$\begin{aligned}
T_1(n) &\le c_1(\alpha n) - c_2\lg(\alpha n) + c_1(1-\alpha)n - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1 n - c_2\lg(\alpha n) - c_2\lg((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1 n - c_2\,(\,\lg(\alpha(1-\alpha)) + 2\,\lg n\,) + \Theta(\lg n) \\
&\le c_1 n - c_2\,\lg n - (c_2(\lg n + \lg(\alpha(1-\alpha))) - \Theta(\lg n)) \\
&\le c_1 n - c_2\,\lg n,
\end{aligned}$$
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: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$

Parallel Merge Sort

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

CASE 2

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

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

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

$= \Theta(n \lg n)$

Parallel Merge Sort

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

CASE 2

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

$f(n) = \Theta(n^{\log_b a} \lg^2 n)$

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

$= \Theta(\lg^3 n)$

Parallelism of P_MergeSort

$$\textit{Work:} \quad T_1(n) = \Theta(n \lg n)$$

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

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

Tableau Construction

Constructing a Tableau

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

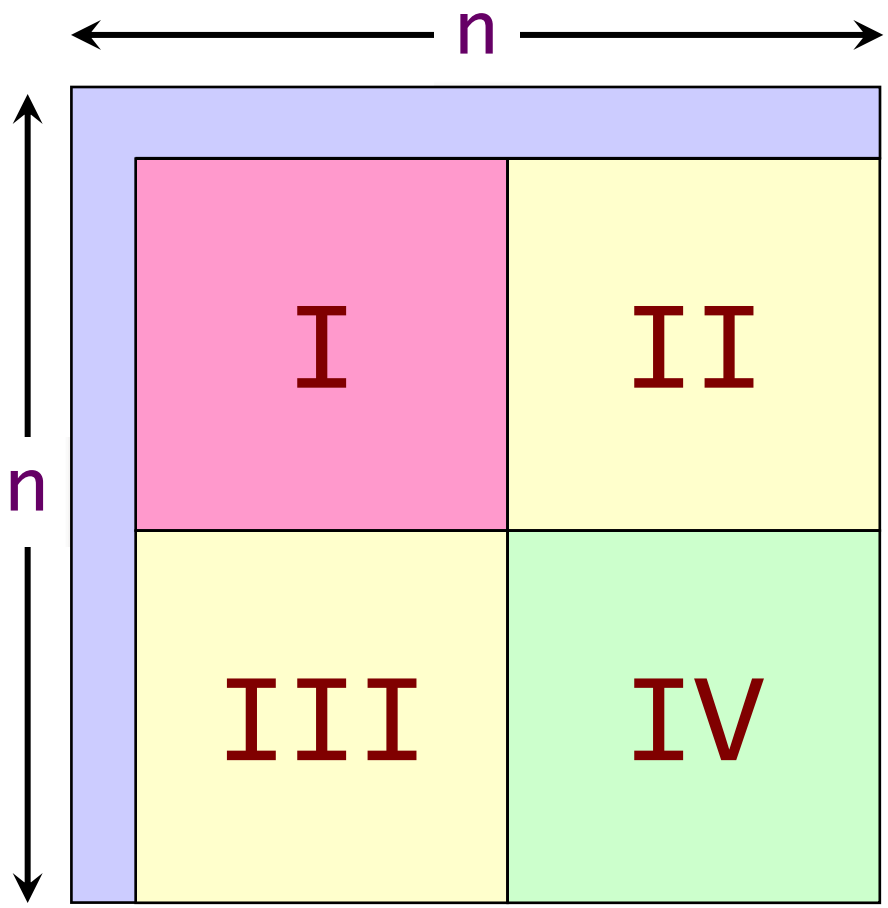
00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Dynamic programming

- Longest common subsequence
- Edit distance
- Time warping

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

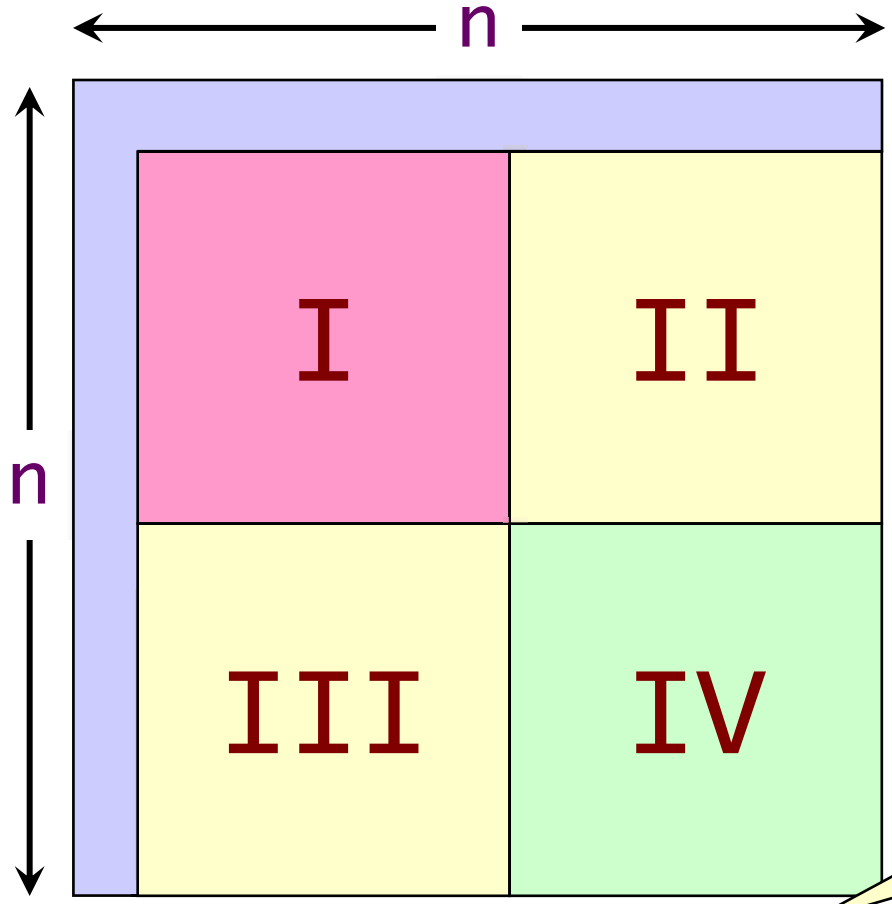
Recursive Construction



Parallel code

```
I;
cilk_spawn II;
III;
cilk_sync;
IV;
```

Recursive Construction



Parallel code

```
I;
cilk_spawn II;
III;
cilk_sync;
IV;
```

CASE 1

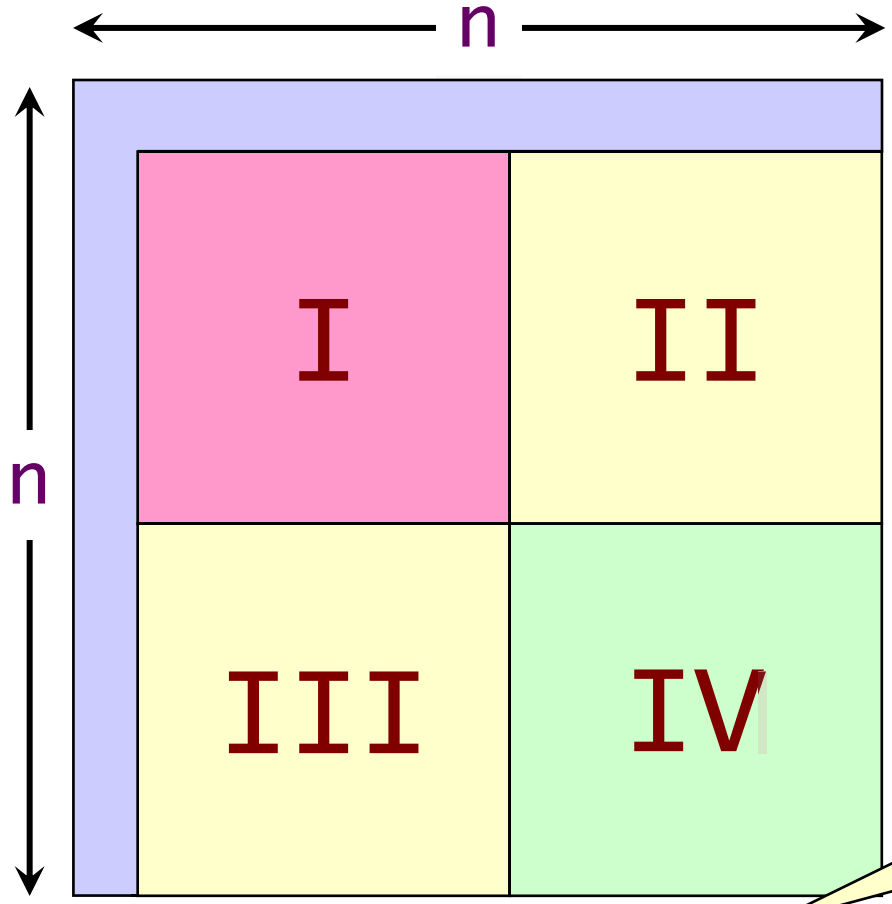
$n^{\log_b a} = n^{\log_2 4} = n^2$

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

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

$= \Theta(n^2)$

Recursive Construction



Parallel code

```
I;
cilk_spawn II;
III;
cilk_sync;
IV;
```

CASE 1

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

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

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

$= \Theta(n^{\lg 3})$

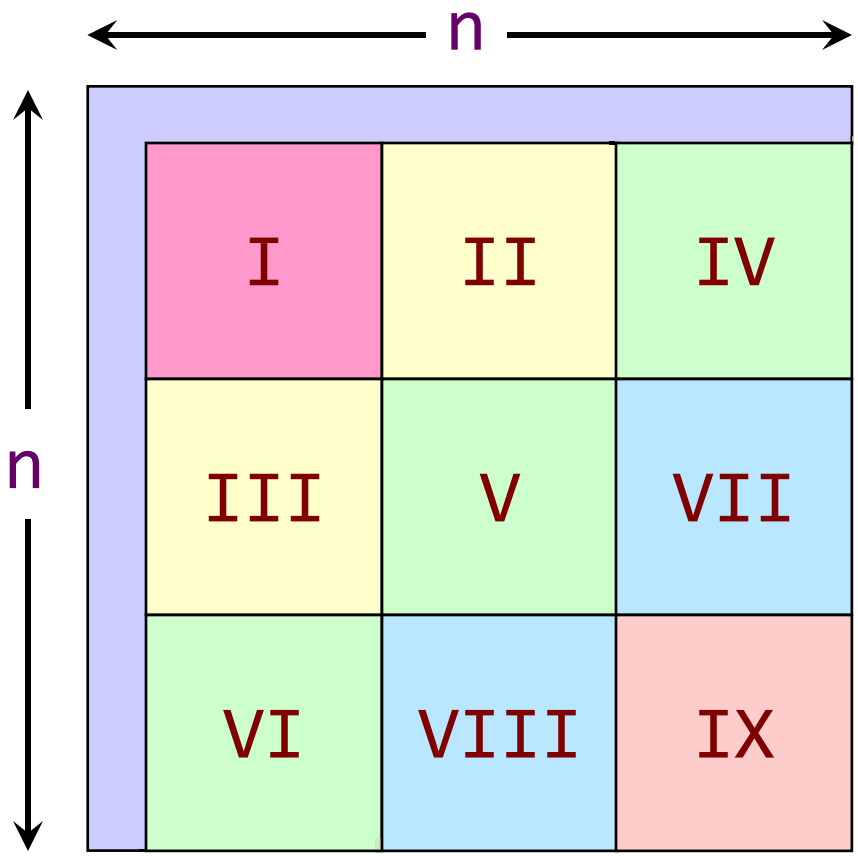
Analysis of Tableau Constr.

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

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

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

A More-Parallel Construction




```
I;
cilk_spawn II;
III;
cilk_sync;
cilk_spawn IV;
cilk_spawn V;
VI;
cilk_sync;
cilk_spawn VII;
VIII;
cilk_sync;
IX;
```

Work: $T_1(n) = 9T_1(n/3) + \Theta(1)$

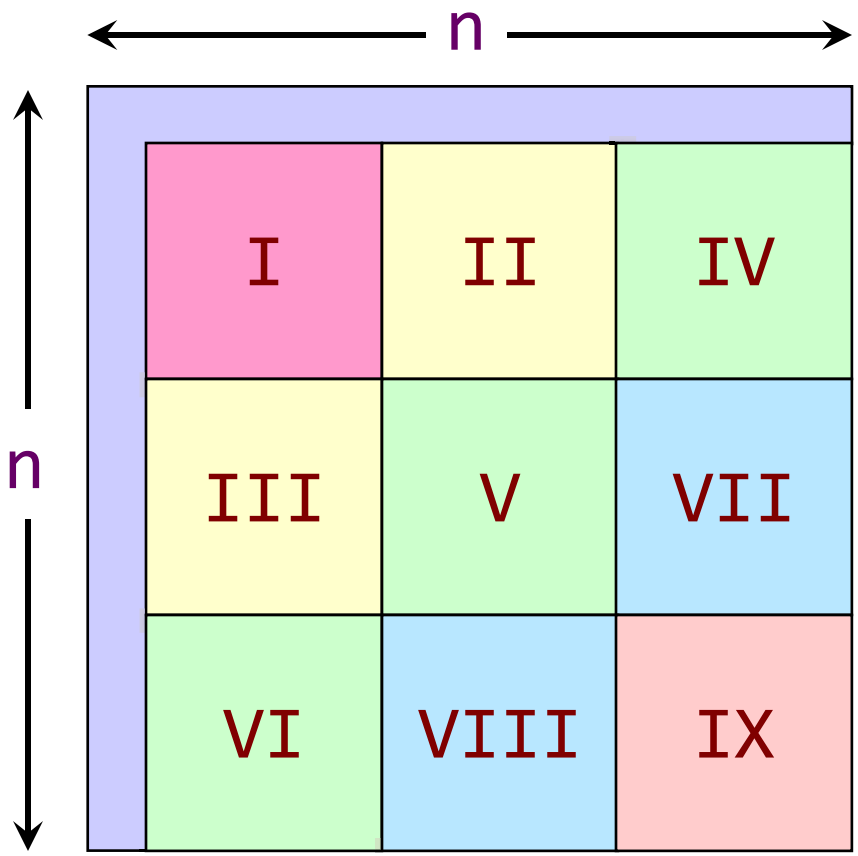
$= \Theta(n^2)$

CASE 1

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

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

A More-Parallel Construction




```
I;
cilk_spawn II;
III;
cilk_sync;
cilk_spawn IV;
cilk_spawn V;
VI;
cilk_sync;
cilk_spawn VII;
VIII;
cilk_sync;
IX;
```

Span: $T_\infty(n) = 5T_\infty(n/3) + \Theta(1)$

$= \Theta(n^{\log_3 5})$

CASE 1

$n^{\log_b a} = n^{\log_3 5}$

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

Analysis of Revised Method

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

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

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

Nine-way divide-and-conquer has about $\Theta(n^{0.12})$ 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 using *pure* Cilk?

- You may only use basic fork-join control constructs (`cilk_spawn`, `cilk_sync`, `cilk_for`) for synchronization.
- No using locks, atomic instructions, synchronizing through memory, etc.