6.172
Performance Engineering of Software Systems

LECTURE 14
Analysis of Multithreaded Algorithms

Charles E. Leiserson

October 28, 2010

© 2010 Charles E. Leiserson

OUTLINE

- Divide-&-Conquer Recurrences
- Cilk Loops
- Matrix Multiplication
- Merge Sort
- Tableau Construction

© 2010 Charles E. Leiserson

OUTLINE

- Divide-&-Conquer Recurrences
- Cilk Loops
- Matrix Multiplication
- Merge Sort
- Tableau Construction

The Master Method

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

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

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

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

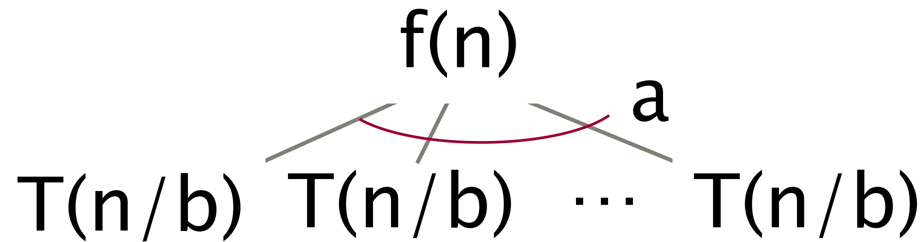
© 2010 Charles E. Leiserson

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

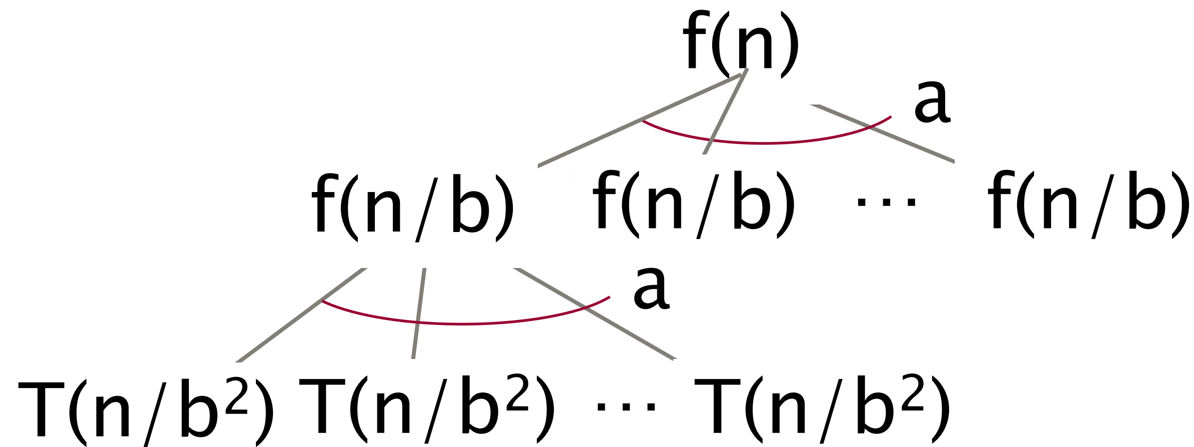
$T(n)$

© 2010 Charles E. Leiserson

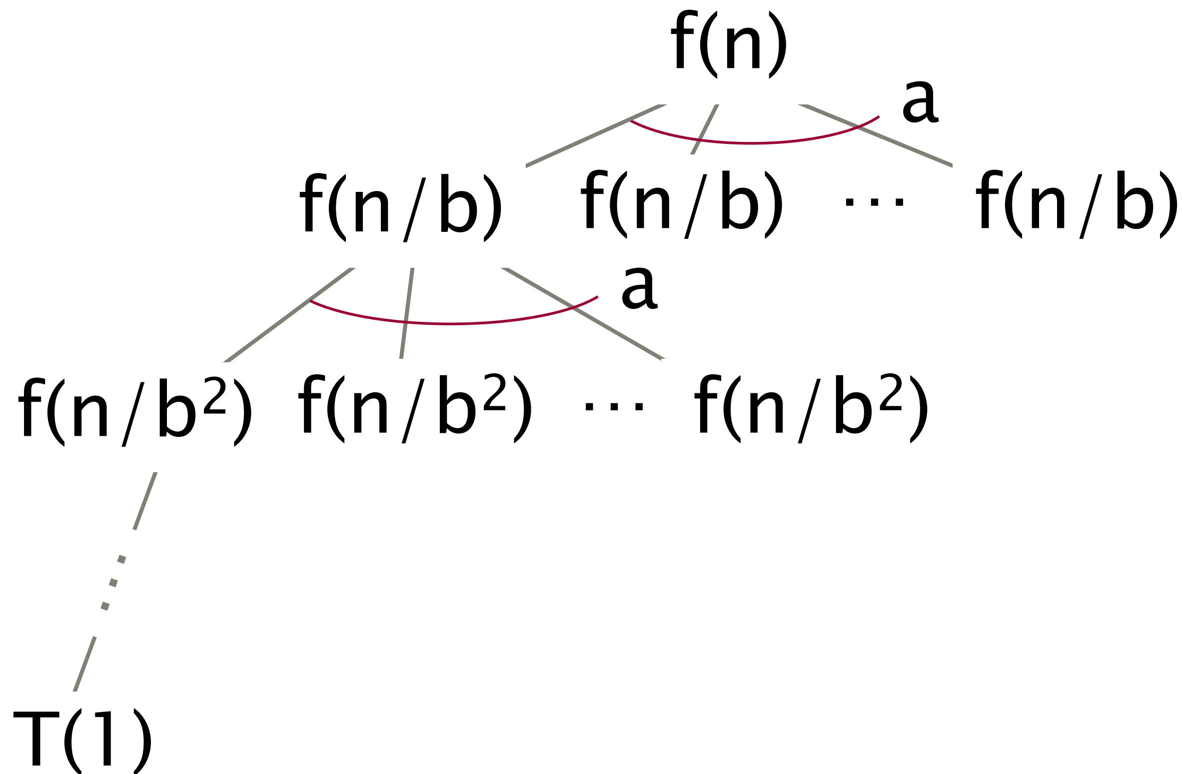
Recursion Tree: $T(n) = aT(n/b) + f(n)$



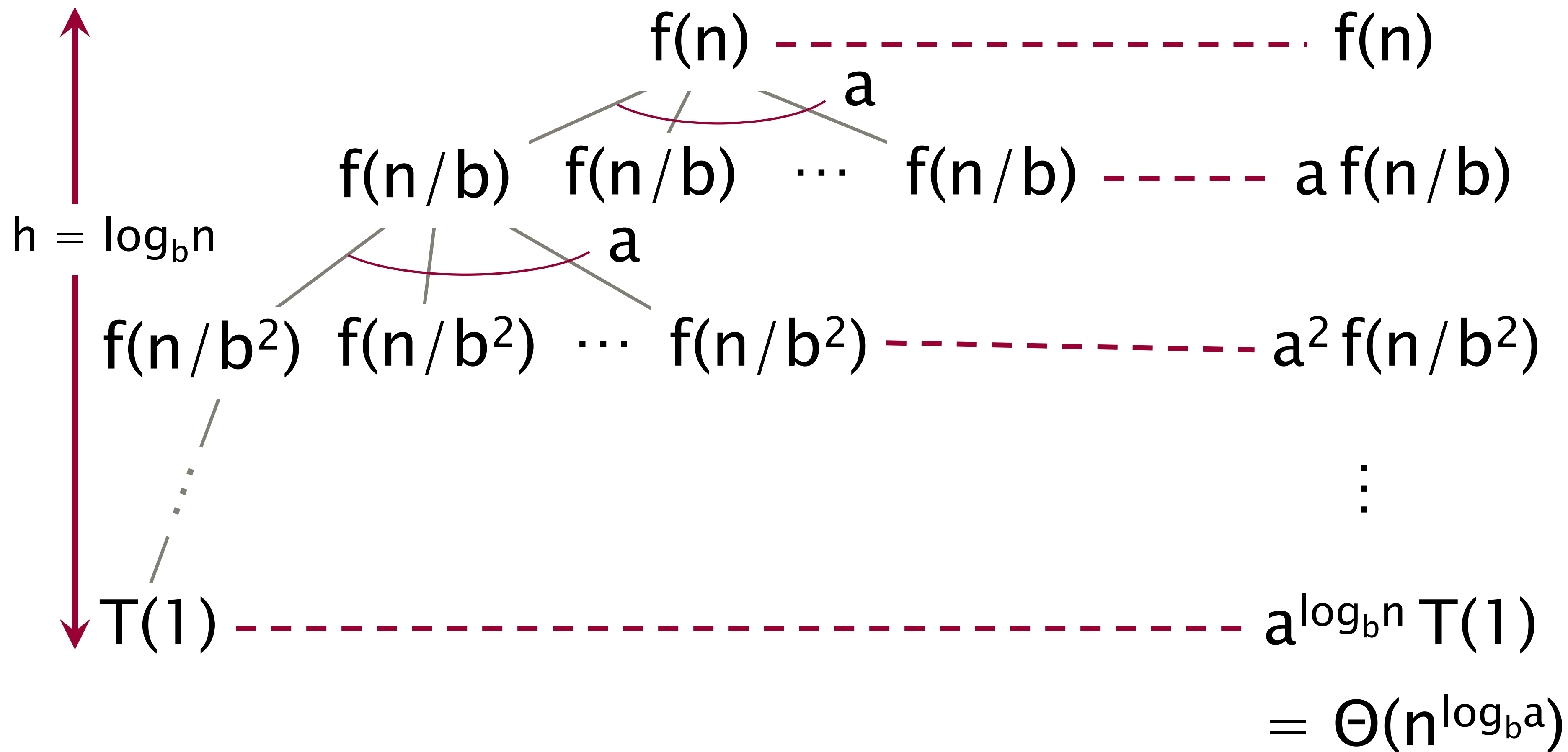
Recursion Tree: $T(n) = aT(n/b) + f(n)$



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



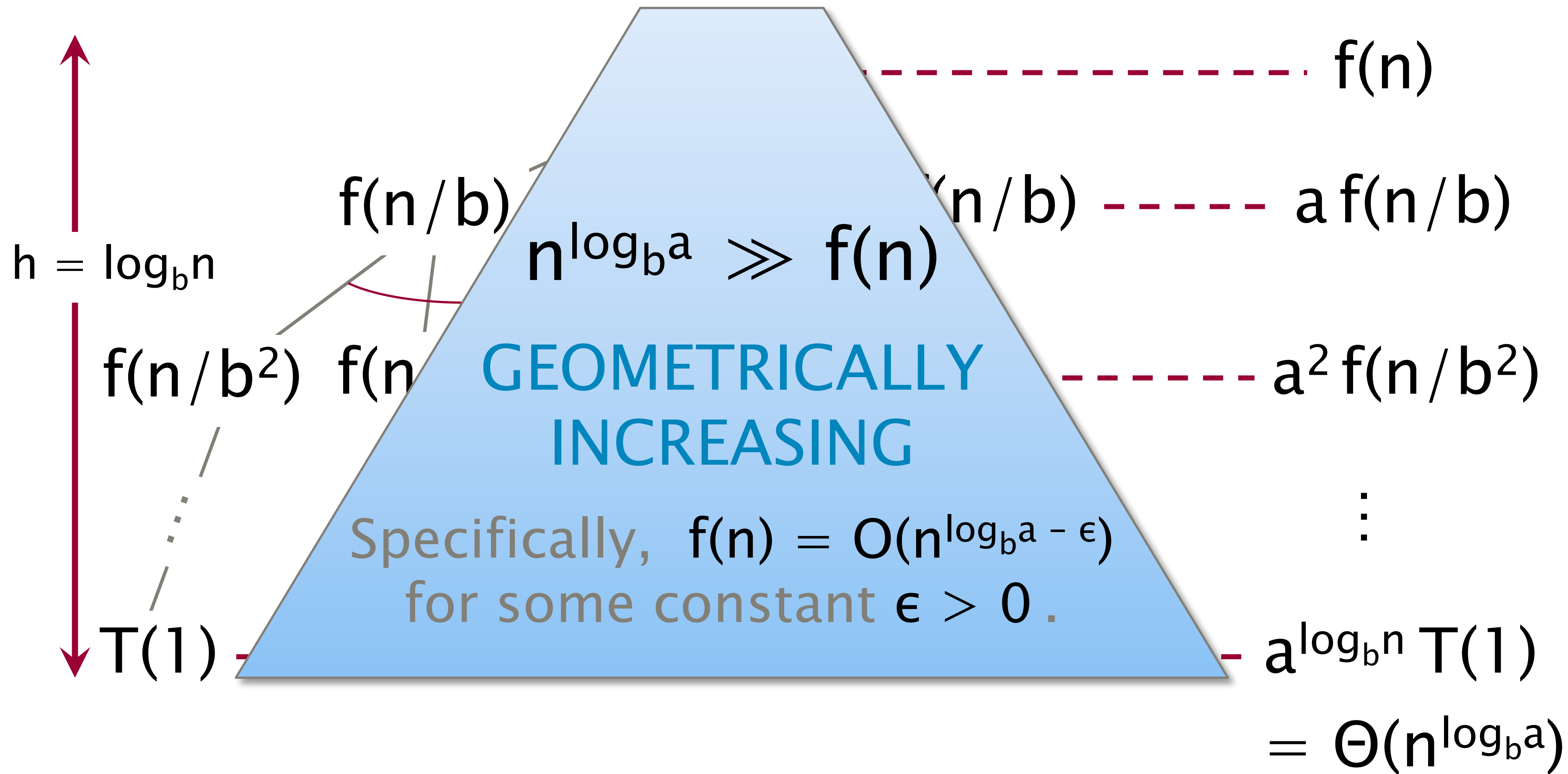
Recursion Tree: $T(n) = a\,T(n/b) + f(n)$



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

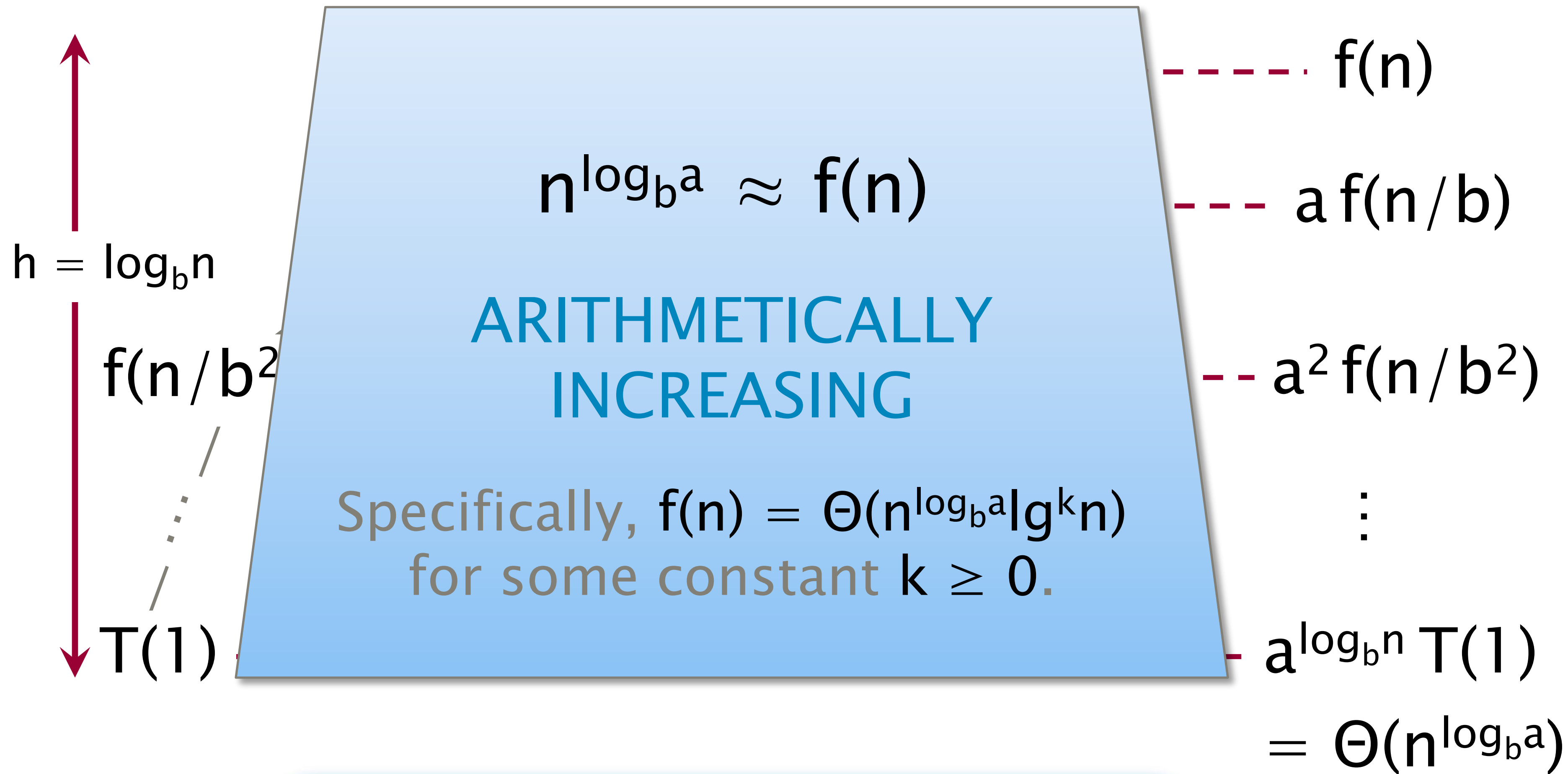
© 2010 Charles E. Leiserson

Master Method — Case I



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

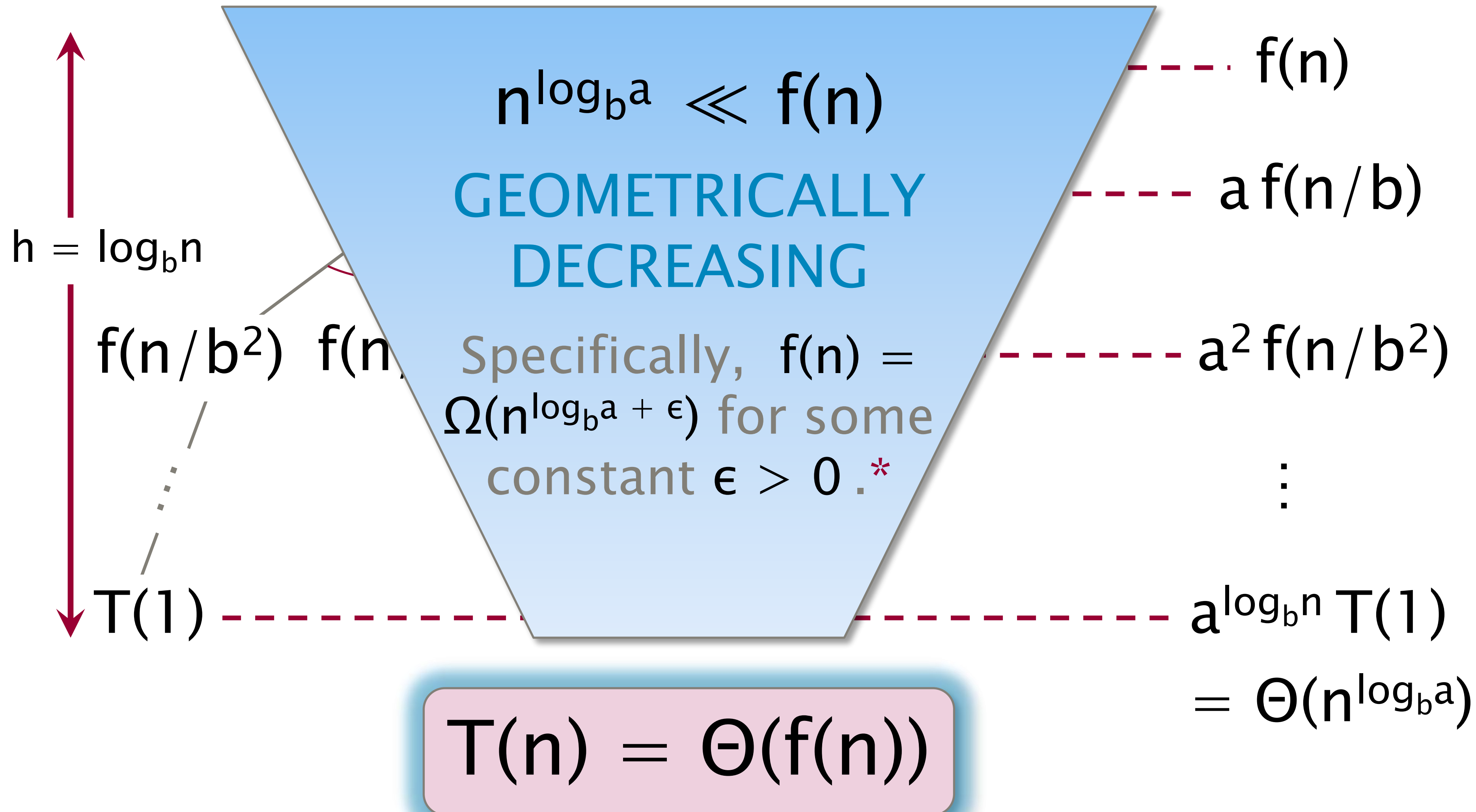
Master Method — Case 2



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

© 2010 Charles E. Leiserson

Master Method — Case 3



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

© 2010 Charles E. Leiserson

Master–Method Cheat Sheet

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

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

© 2010 Charles E. Leiserson

Master Method Quiz

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

© 2010 Charles E. Leiserson

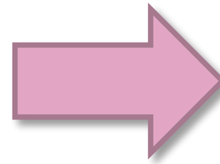
OUTLINE

- Divide-&-Conquer Recurrences
- Cilk Loops
- Matrix Multiplication
- Merge Sort
- Tableau Construction

Loop Parallelism in Cilk++

Example: In-place matrix transpose

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



$$A^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}$$

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;
    }
}
```

Implemention of Parallel Loops

```
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) {
    if (hi > lo) { // coarsen
        int mid = lo + (hi - lo)/2;
        cilk_spawn recur(lo, mid);
        recur(mid, hi);
        cilk_sync;
    } else
        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-1);
```

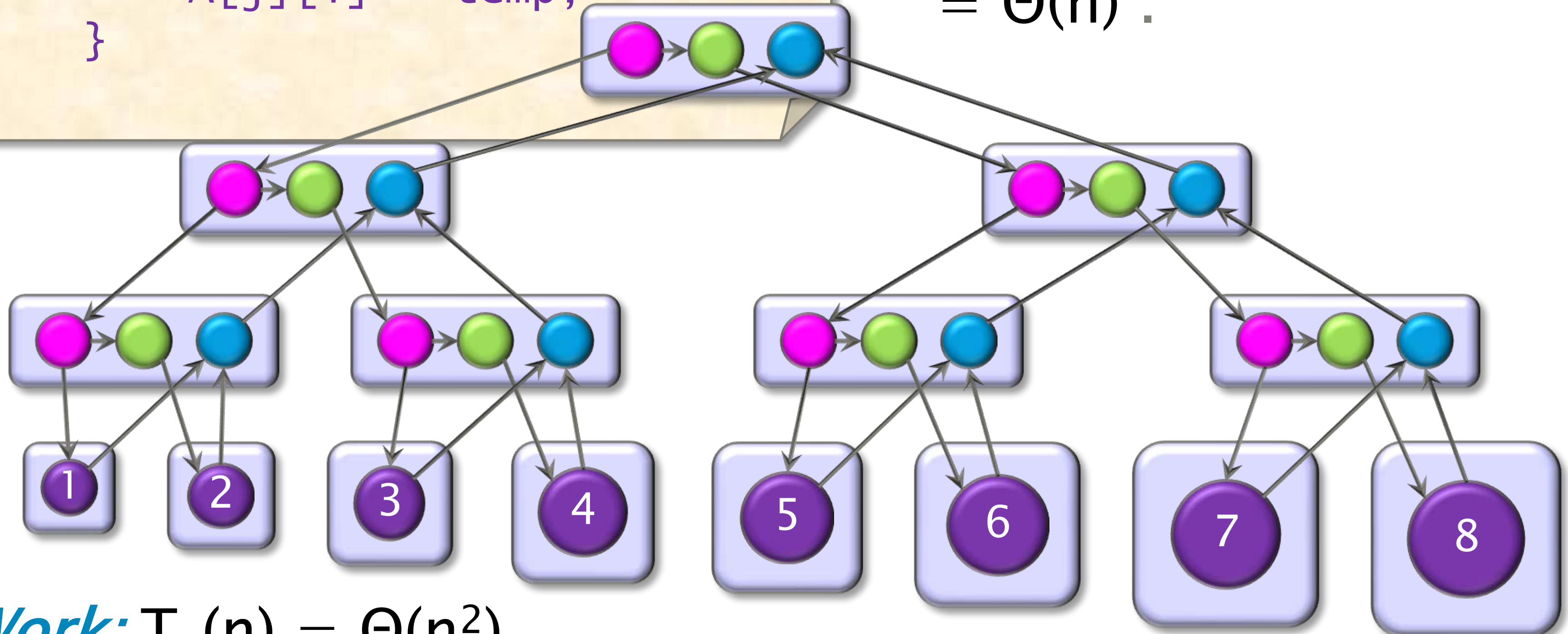
In practice, the recursion is ***coarsened*** to minimize overheads.

Analysis of Parallel Loops

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

© 2010 Charles E. Leiserson

Analysis of Nested Parallel Loops

```
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 $= \Theta(\lg n)$.

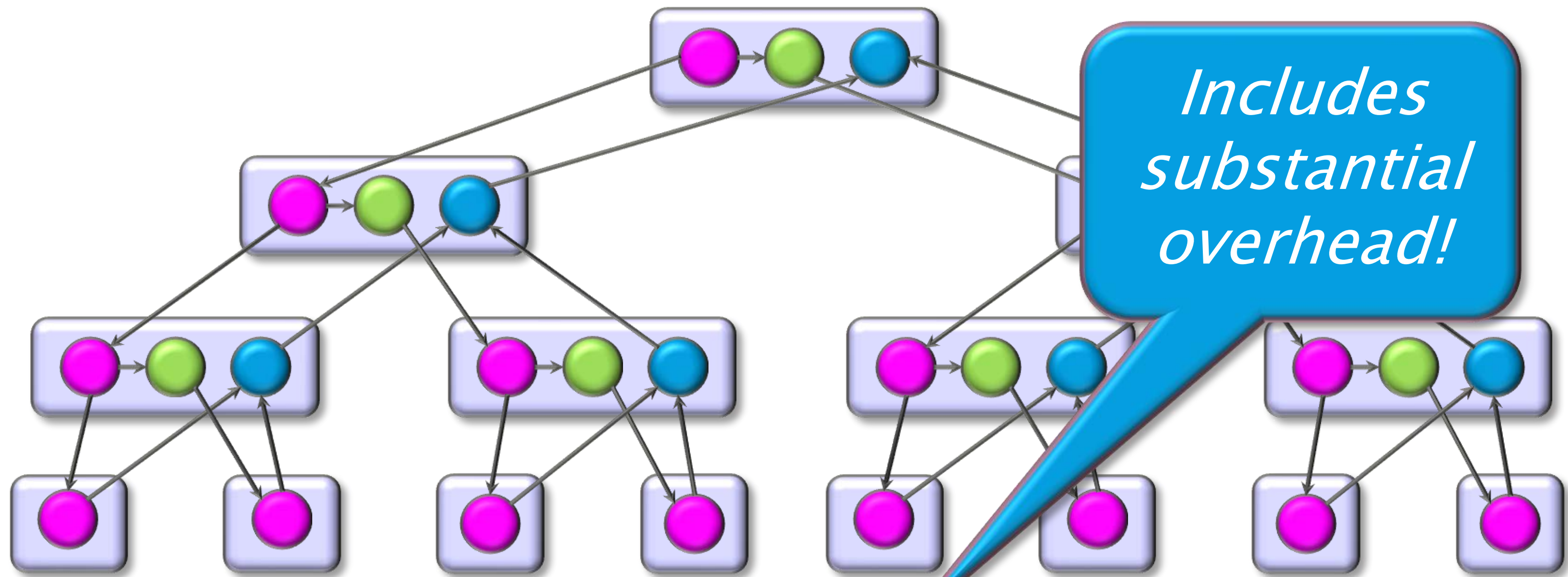
Span of body $= \Theta(1)$.

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];
}
```



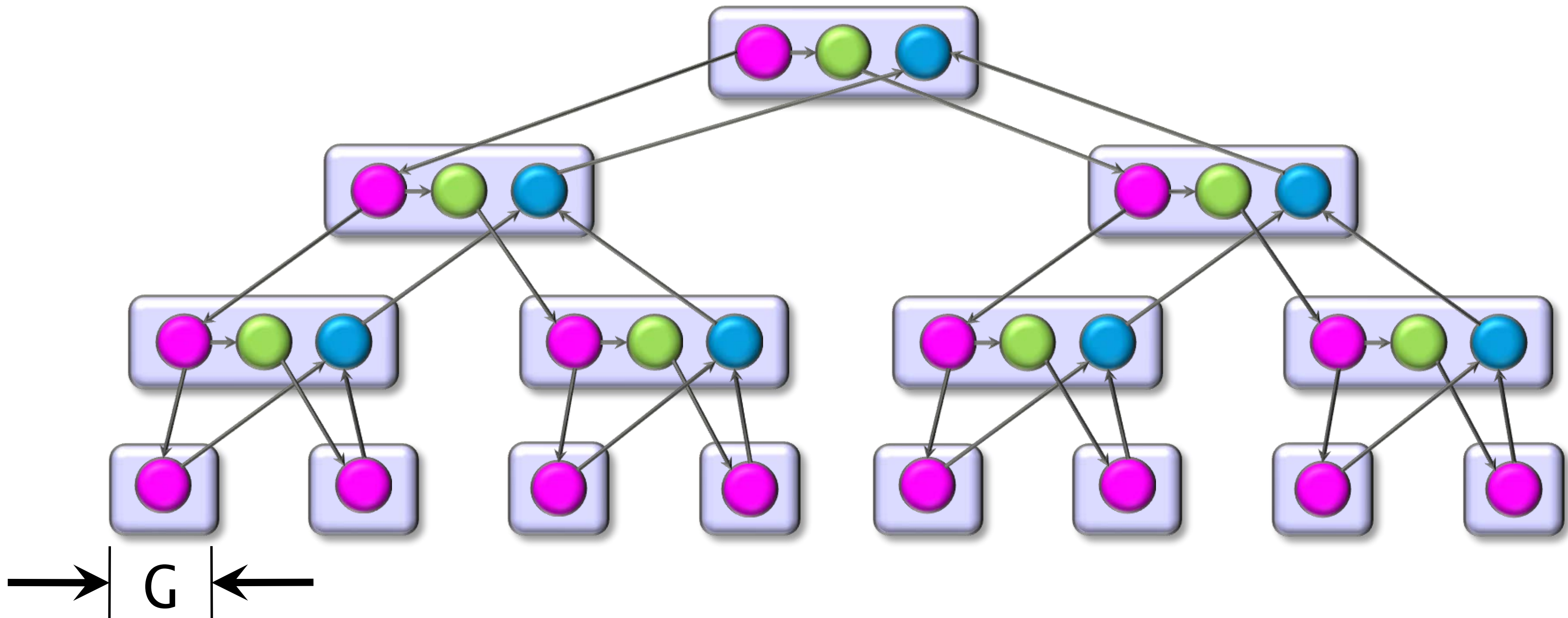
Work: $T_1 = \Theta(n)$
Span: $T_\infty = \Theta(\lg n)$
Parallelism: $T_1/T_\infty = \Theta(n/\lg n)$

© 2010 Charles E. Leiserson

Loop Grain Size

Vector addition

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



Work: $T_1 = n \cdot t_{iter} + (n/G) \cdot t_{spawn}$

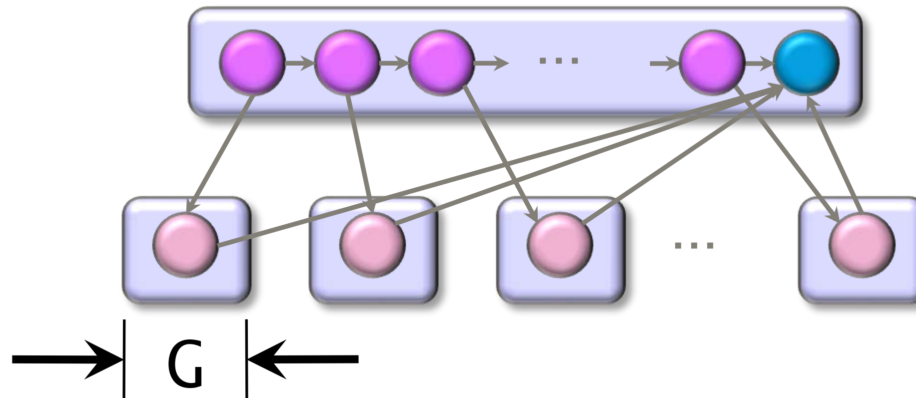
Span: $T_\infty = G \cdot t_{iter} + \lg(n/G) \cdot t_{spawn}$

Want $G \gg t_{spawn}/t_{iter}$ 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: $T_\infty = \Theta(n)$

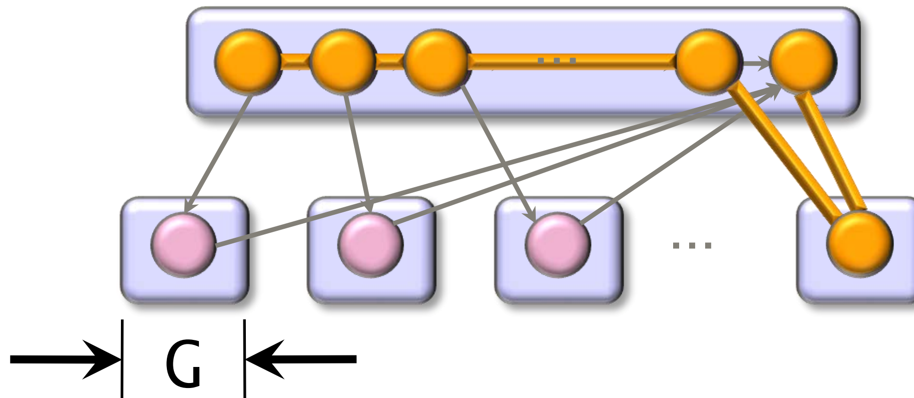
Parallelism: $T_1/T_\infty = \Theta(1)$

© 2010 Charles E. Leiserson

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

© 2010 Charles E. Leiserson

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;
```

© 2010 Charles E. Leiserson

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);
}
```

(Cilkview will show a high *burdened parallelism*.)

© 2010 Charles E. Leiserson

OUTLINE

- Divide-&-Conquer Recurrences
- Cilk Loops
- **Matrix Multiplication**
- Merge Sort
- Tableau Construction

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=1; 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: $T_1 = \Theta(n^3)$

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

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

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

Recursive Matrix Multiplication

Divide and conquer —

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

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

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

D&C Matrix Multiplication

```
template <typename T>
void MMult(T *C, T *A, T *B, int n, int size) {
  T *D = new T[n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2, size);
  cilk_spawn MMult(C12, A11, B12, n/2, size);
  cilk_spawn MMult(C22, A21, B12, n/2, size);
  cilk_spawn MMult(C21, A21, B11, n/2, size);
  cilk_spawn MMult(D11, A12, B21, n/2, size);
  cilk_spawn MMult(D12, A12, B22, n/2, size);
  cilk_spawn MMult(D22, A22, B22, n/2, size);
             MMult(D21, A22, B21, n/2, size);
  cilk_sync;
  MAdd(C, D, n, size); // C += D;
  delete[] D;
}
```

© 2010 Charles E. Leiserson

D&C Matrix Multiplication

```
template <typename T>
void MMult(T *C, T *A, T *B, int n, int size) {
  T *D = new T[n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2,   ze);
  cilk_spawn MMult(C12, A11, B12, n/2    ze);
  cilk_spawn MMult(C22, A21,                 );
  cilk_spawn MMult(C21, A21,                 );
  cilk_spawn MMult(D11, A12,                 );
  cilk_spawn MMult(D12, A12, B22, n/2, size);
  cilk_spawn MMult(D22, A22, B22, n/2, size);
             MMult(D21, A22, B21, n/2, size);
  cilk_sync;
  MAdd(C, D, n, size); // C += D;
  delete[] D;
}
```

Row length of matrices

© 2010 Charles E. Leiserson

D&C Matrix Multiplication

```
template <typename T>
void MMult(T *C, T *A, T *B, int n, int size) {
  T *D = new T[n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2, size);
  cilk_spawn MMult(C12, A11, B12, n/2, size);
  cilk_spawn MMult(C22, A21, B12, n/2, size);
  cilk_spawn MMult(C21, A21, B11, n/2, size);
  cilk_spawn MMult(D11, A12, B21, n/2, size);
  cilk_spawn MMult(D12, A12, B22, n/2, size);
  cilk_spawn MMult(D22, A22, B22, n/2, size);
             MMult(D21, A22, B21, n/2, size);
  cilk_sync;
  MAdd(C, D, n, size); // C += D;
  delete[] D;
}
```

Coarsen for efficiency

D&C Matrix Multiplication

```
template <typename T>
void MMult(T *C, T *A, T *B, int n, int size) {
  T *D = new T[n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2, size);
  cilk_spawn MMult(C1   A11, B12, n/2, size);
  cilk_spawn MMult(C2   A21, B12, n/2, size);
  cilk_spawn MMult(C2   A21, B11, n/2, size);
  cilk_spawn MMult(D1   A12, B21, n/2, size);
  cilk_spawn MMult(D1   A12, B22, n/2, size);
  cilk_spawn MMult(D2   22, B22, n/2, size);
             MMult(                n/2, size);
  cilk_sync;
  MAdd(C, D, n, siz
  delete[] D;
}
```

Determine submatrices by index calculation

Matrix Addition

```
template <typename T>
void MMult(T *C, T *A, T *B, int n, int size) {
  T *D = new T[n*n]);
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2, size);
  cilk_spawn MMult(C12, A11, B12, n/2, size);
  cilk_spawn MMult(C22, A21, B12, n/2, size);
  cilk_spawn MMult(C21, A21, B11, n/2, size);
  cilk_sp
  cilk_sp
  cilk_sp

  cilk_sy
  MAdd(C,
  delete[
}
```

```
template <typename T>
void MAdd(T *C, T *D, int n, int size) {
  cilk_for (int i=0; i<n; ++i) {
    cilk_for (int j=0; j<n; ++j) {
      C[size*i+j] += D[size*i+j];
    }
  }
}
```

© 2010 Charles E. Leiserson

Analysis of Matrix Addition

```
template <typename T>
void MAdd(T *C, T *D, int n, int size) {
  cilk_for (int i=0; i<n; ++i) {
    cilk_for (int j=0; j<n; ++j) {
       C[size*i+j] += D[size*i+j];
    }
  }
}
```

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

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

© 2010 Charles E. Leiserson

Work of Matrix Multiplication

```
template <typename T>
void MMult(T *C, T *A, T *B, int n, int size) {
  T *D = new T[n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2, size);
  cilk_spawn MMult(C12, A11, B12, n/2, size);
   ⋮
  cilk_spawn MMult(D22, A22, B22, n/2, size);
             MMult(D21, A22, B21, n/2, size);
  cilk_sync;
  MAdd(C, D, n, size); // C += D;
  delete[] D;
}
```

Work:

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

Work of Matrix Multiplication

```
template <typename T>
void MMult(T *C, T *A, T *B, int n, int size) {
  T *D = new T[n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2, size);
  cilk_spawn MMult(C12                    ze);
  ⋮
  cilk_spawn MMult(D22                    ze);
             MMult(D21                    ze);
  cilk_sync;
  MAdd(C, D, n, size); //      = D;
  delete[] 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

```
template <typename T>
void MMult(T *C, T *A, T *B, int n, int size) {
  T *D = new T[n*n];
  //base case & partition matrices
  cilk_spawn MMult(C11, A11, B11, n/2, size);
  cilk_spawn MMult(C12                        );
  ⋮
  cilk_spawn MMult(D22                        );
             MMult(D21                        );
  cilk_sync;
  MAdd(C, D, n, size); //      D;
  delete[] D;
}
```

maximum

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

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

© 2010 Charles E. Leiserson

Temporaries

```
template <typename T>
void MMult(T *C, T *A, T *B, int n, int size) {
  T *D = new T[n*n];
   //base case & partition matrices
   cilk_spawn MMult(C11, A11, B11, n/2, size);
   cilk_spawn MMult(C12, A11, B12, n/2, size);
   cilk_spawn MMult(C22, A21, B12, n/2, size);
   cilk_spawn MMult(C21, A21, B11, n/2, size);
   cilk_spawn MMult(D11, A12, B21, n/2, size);
   cilk_spawn MMult(D12, A12, B22, n/2, size);
   cilk_spawn MMult(D22, A22, B22, n/2, size);
              MMult(D21, A22, B21, n/2, size);
   cilk_sync;
   MAdd(C, D, n, size); // C += D;
   delete[] D;
}
```

IDEA: Since minimizing storage tends to yield higher performance, trade off parallelism for less storage.

No-Temp Matrix Multiplication

```
// C += A*B;
template <typename T>
void MMult2(T *C, T *A, T *B, int n, int size)
{
  //base case & partition matrices
  cilk_spawn MMult2(C11, A11, B11, n/2, size);
  cilk_spawn MMult2(C12, A11, B12, n/2, size);
  cilk_spawn MMult2(C22, A21, B12, n/2, size);
             MMult2(C21, A21, B11, n/2, size);
  cilk_sync;
  cilk_spawn MMult2(C11, A12, B21, n/2, size);
  cilk_spawn MMult2(C12, A12, B22, n/2, size);
  cilk_spawn MMult2(C22, A22, B22, n/2, size);
             MMult2(C21, A22, B21, n/2, size);
  cilk_sync;
}
```

Saves space, but at what expense?

© 2010 Charles E. Leiserson

Work of No-Temp Multiply

```
// C += A*B;
template <typename T>
void MMult2(T *C, T *A, T *B, int n, int size) {
  //base case & partition matrices
  cilk_spawn MMult2(C11, A11, B11, n/2, size);
  cilk_spawn MMult2(C12, A11, B12, n/2, size);
  cilk_spawn MMult2(C2
             MMult2(C2
  cilk_sync;
  cilk_spawn MMult2(C1
  cilk_spawn MMult2(C1
  cilk_spawn MMult2(C2
             MMult2(C21, A
  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

```
// C += A*B;
template <typename T>
void MMult2(T *C, T *A, T *B, int n, int size) {
  //base case & partition matrices
  cilk_spawn MMult2(C11, A11, B11, n/2, size);
  cilk_spawn MMult2(C12, A11, B12, n/2, size);
  cilk_spawn MMult2(C2
             MMult2(C2
  cilk_sync;
  cilk_spawn MMult2(C1
  cilk_spawn MMult2(C1
  cilk_spawn MMult2(C2
             MMult2(C21,
  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

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

Faster in practice!

© 2010 Charles E. Leiserson

OUTLINE

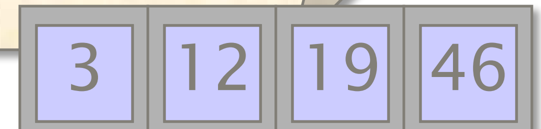
- Divide-&-Conquer Recurrences
- Cilk Loops
- Matrix Multiplication
- Merge Sort
- Tableau Construction

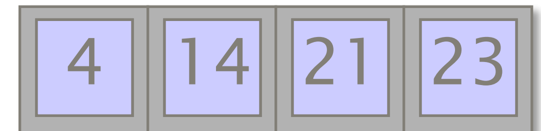
© 2010 Charles E. Leiserson

Merging Two Sorted Arrays

```
template <typename T>
void Merge(T *C, T *A, T *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--;
  }
}
```

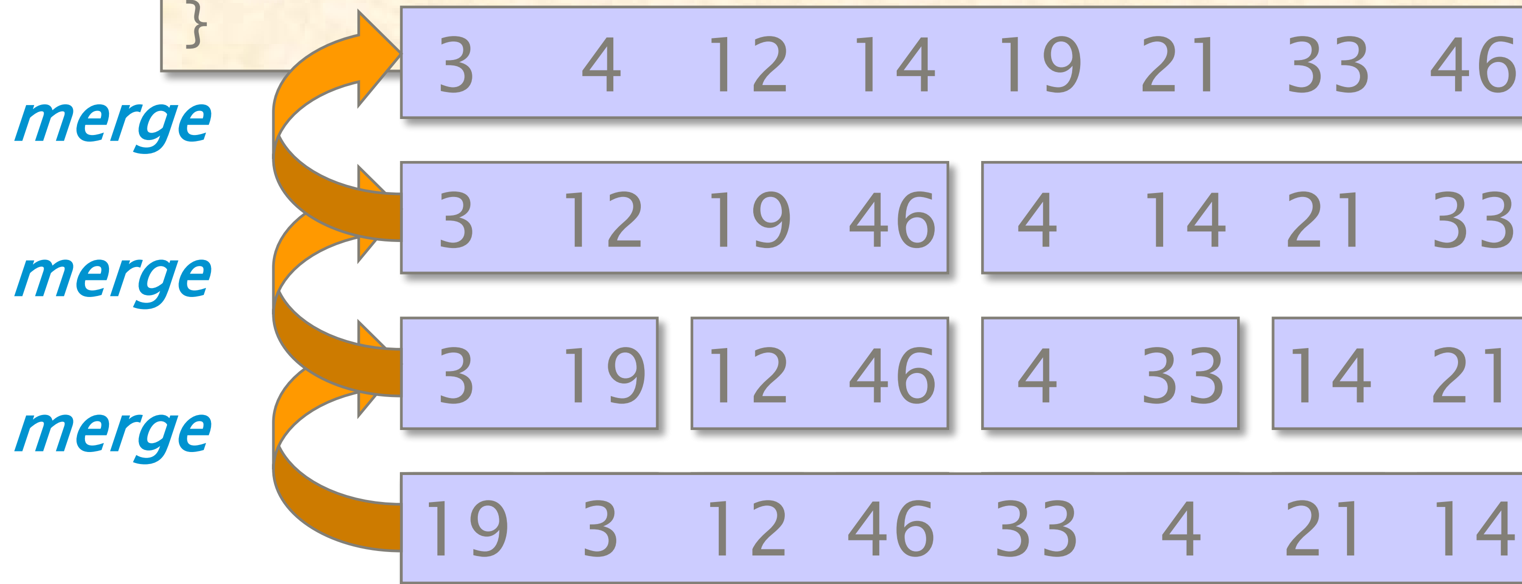
Time to merge n elements = $\Theta(n)$.





Merge Sort

```
template <typename T>
void MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T* C[n];
    cilk_spawn MergeSort(C, A, n/2);
               MergeSort(C+n/2, A+n/2, n-n/2);
    cilk_sync;
    Merge(B, C, C+n/2, n/2, n-n/2);
  }
}
```



Work of Merge Sort

```
template <typename T>
void MergeSort(T *B, T *A, i
  if (n==1) {
    B[0] = A[0];
  } else {
    T* C[n];
    cilk_spawn MergeSort(C,
               MergeSort(C+n/2,        2, n-n/2);
    cilk_sync;
    Merge(B, C, C+n/2, 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: $T_1(n) = 2T_1(n/2) + \Theta(n)$

$= \Theta(n \lg n)$

Span of Merge Sort

```
template <typename T>
void MergeSort(T *B, T *A, i
  if (n==1) {
    B[0] = A[0];
  } else {
    T* C[n];
    cilk_spawn MergeSort(C,
               MergeSort(C+n/2,             , n-n/2);
    cilk_sync;
    Merge(B, C, C+n/2, n/2, n-          );
  }
}
```

CASE 3:

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

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

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

$= \Theta(n)$

Parallelism of Merge Sort

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

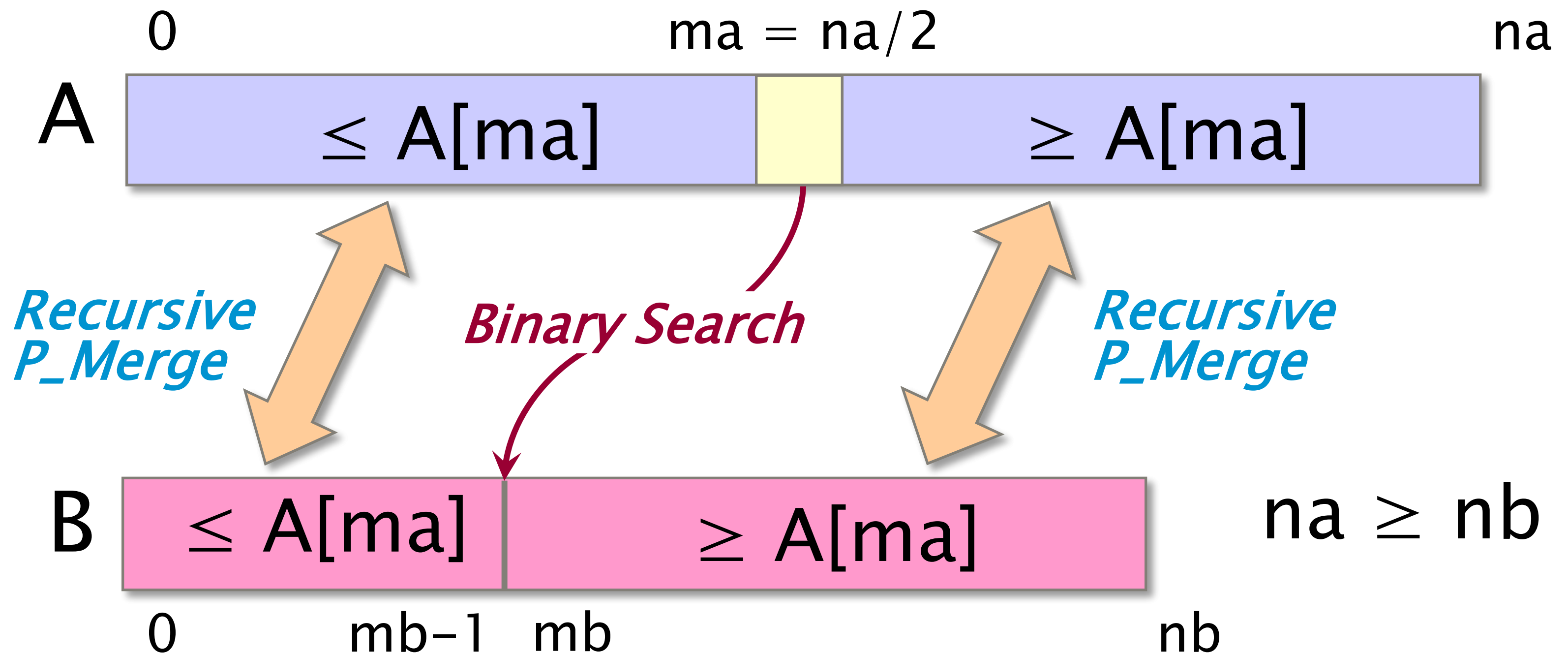
Span: $T_\infty(n) = \Theta(n)$

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

We need to parallelize the merge!

© 2010 Charles E. Leiserson

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 $(3/4)n$.

© 2010 Charles E. Leiserson

Parallel Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    P_Merge(C, B, A, nb, na);
  } else if (na==0) {
      return;
  } else {
    int ma = na/2;
    int mb = BinarySearch(A[ma], B, nb);
    C[ma+mb] = A[ma];
    cilk_spawn P_Merge(C, A, B, ma, mb);
    P_Merge(C+ma+mb+1, A+ma+1, B+mb, na-ma-1, nb-mb);
    cilk_sync;
  }
}
```

Coarsen base cases for efficiency.

© 2010 Charles E. Leiserson

Span of Parallel Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    ⋮
    int mb = BinarySearch(A[ma]
    C[ma+mb] = A[ma];
    cilk_spawn P_Merge(C, A, B,
    P_Merge(C+ma+mb+1, A+ma+1, B
    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: $T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n)$

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

© 2010 Charles E. Leiserson

Work of Parallel Merge

```
template <typename T>
void P_Merge(T *C, T *A, T *B, int na, int nb) {
  if (na < nb) {
    ⋮
    int mb = BinarySearch(A[ma], B, nb);
    C[ma+mb] = A[ma];
    cilk_spawn P_Merge(C, A, B, ma, mb)
    P_Merge(C+ma+mb+1, A+ma+1, B+mb, na-                 );
    cilk_sync;
  }
}
```

HAIRY!

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

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

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

© 2010 Charles E. Leiserson

Analysis of Work Recurrence

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

$$
\begin{aligned}
T_1(n) &= T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n) \\
&\le c_1(\alpha n) - c_2\lg(\alpha n) \\
&\quad + c_1(1-\alpha)n - c_2\lg((1-\alpha)n) + \Theta(\lg n)
\end{aligned}
$$

© 2010 Charles E. Leiserson

Analysis of Work Recurrence

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

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

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

© 2010 Charles E. Leiserson

Parallelism of P_Merge

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

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

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

© 2010 Charles E. Leiserson

Parallel Merge Sort

```
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T C[n];
    cilk_spawn P_MergeSort
    P_MergeSort(C+n/2, A+n/2,            );
    cilk_sync;
    P_Merge(B, C, C+n/2, n/2      n/2);
  }
}
```

CASE 2:

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

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

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

$= \Theta(n \lg n)$

© 2010 Charles E. Leiserson

Parallel Merge Sort

```
template <typename T>
void P_MergeSort(T *B, T *A, int n) {
  if (n==1) {
    B[0] = A[0];
  } else {
    T C[n];
    cilk_spawn P_MergeSort
    P_MergeSort(C+n/2, A+n/2,          );
    cilk_sync;
    P_Merge(B, C, C+n/2, n/2     n/2);
  }
}
```

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

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

- Divide-&-Conquer Recurrences
- Cilk Loops
- Matrix Multiplication
- Merge Sort
- **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

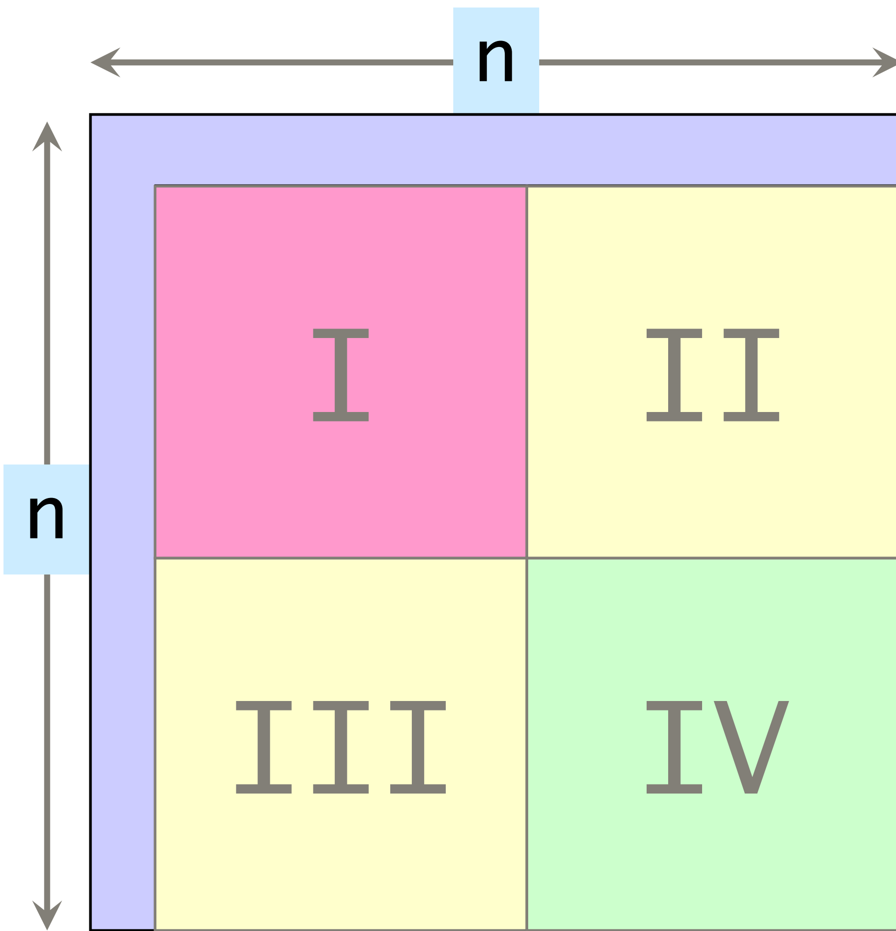
Dynamic programming

- Longest common subsequence
- Edit distance
- Time warping

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

© 2010 Charles E. Leiserson

Recursive Construction

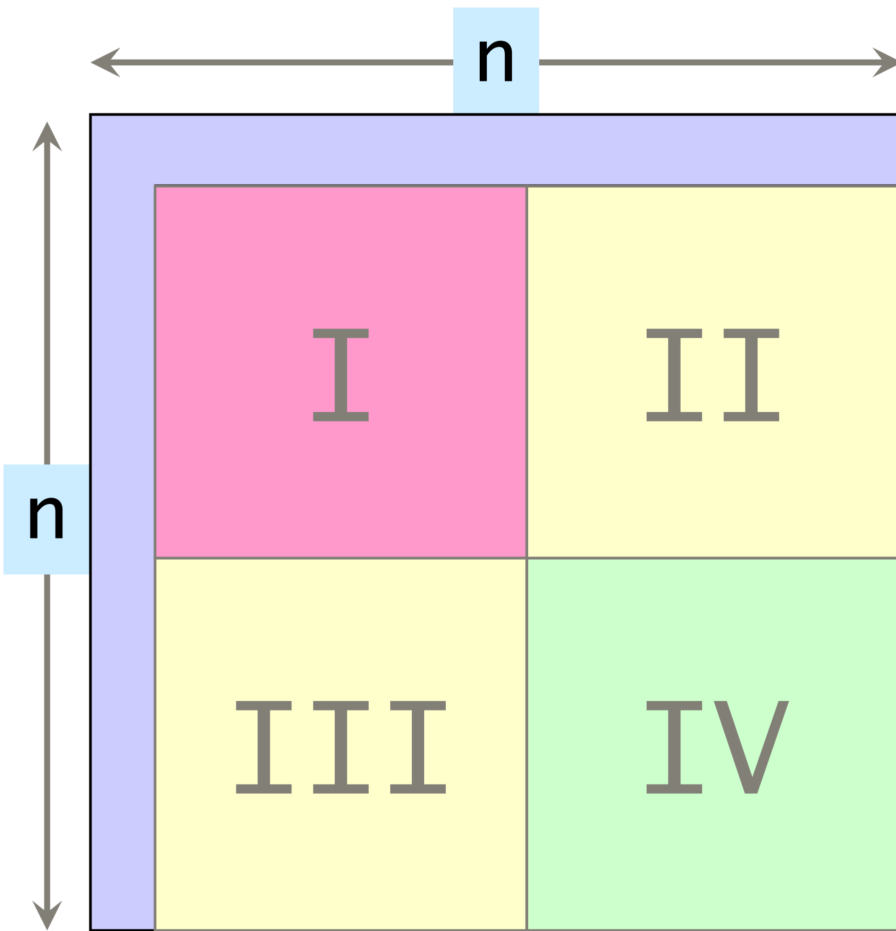


Parallel code

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

© 2010 Charles E. Leiserson

Recursive Construction



Parallel code

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

CASE 1:

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

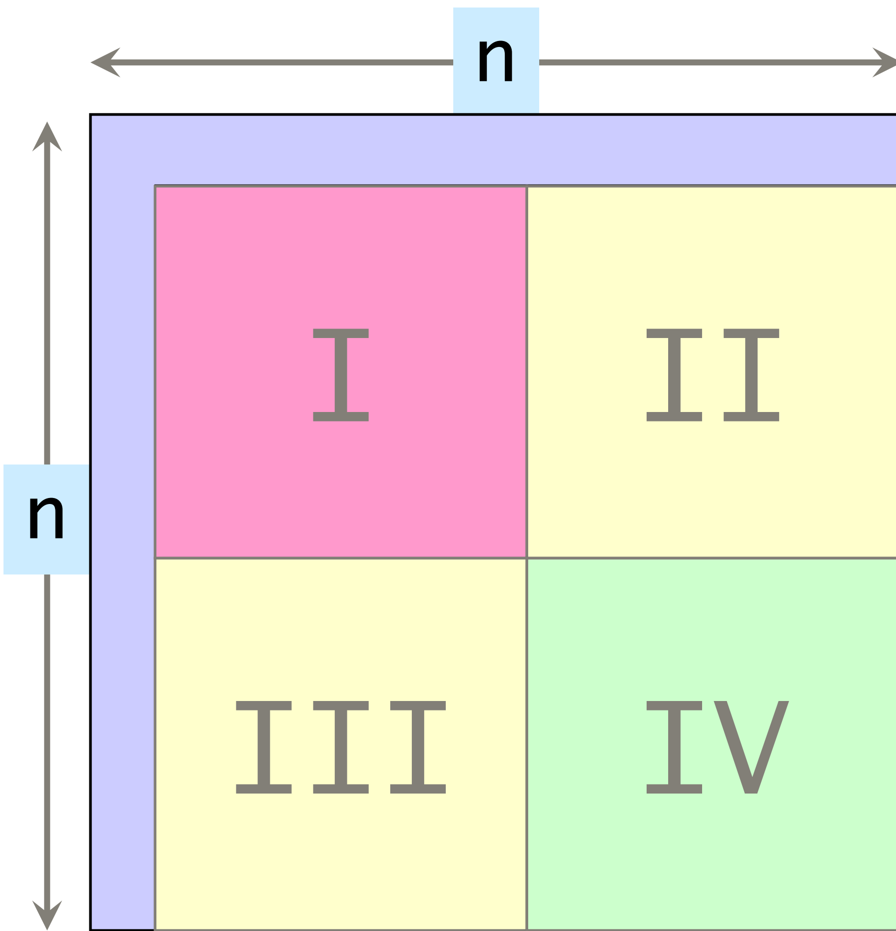
$f(n) = \Theta(1)$

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

$= \Theta(n^2)$

© 2010 Charles E. Leiserson

Recursive Construction



Parallel code

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

CASE 1:

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

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

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

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

© 2010 Charles E. Leiserson

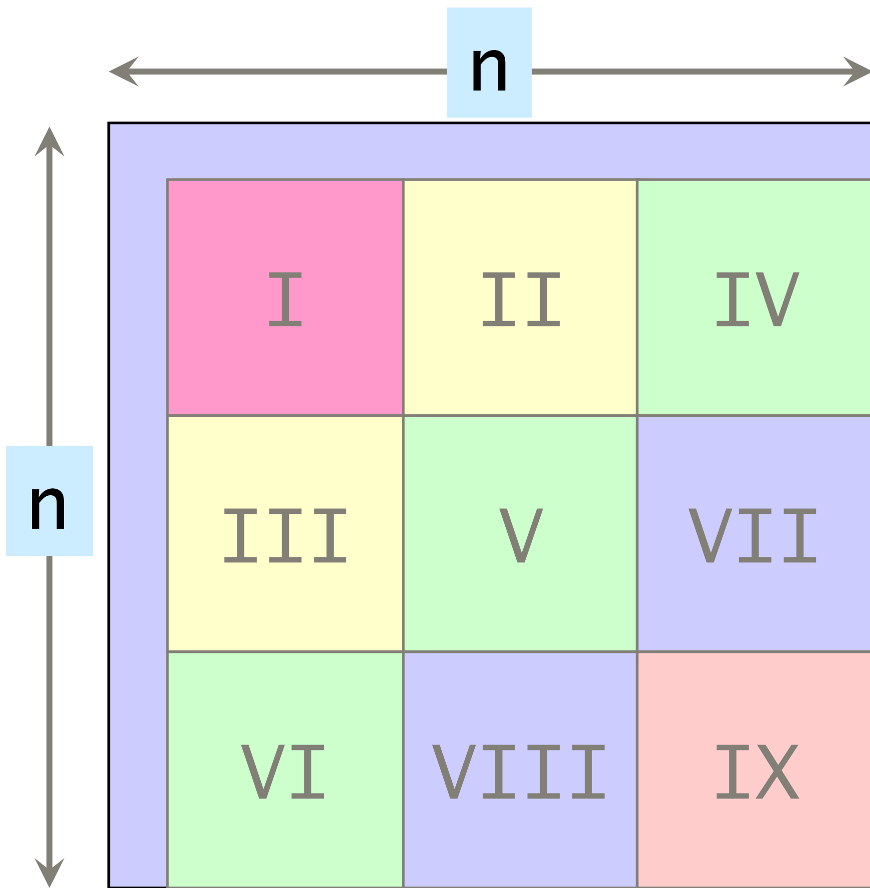
Analysis of Tableau Constr.

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;
```

CASE 1:

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

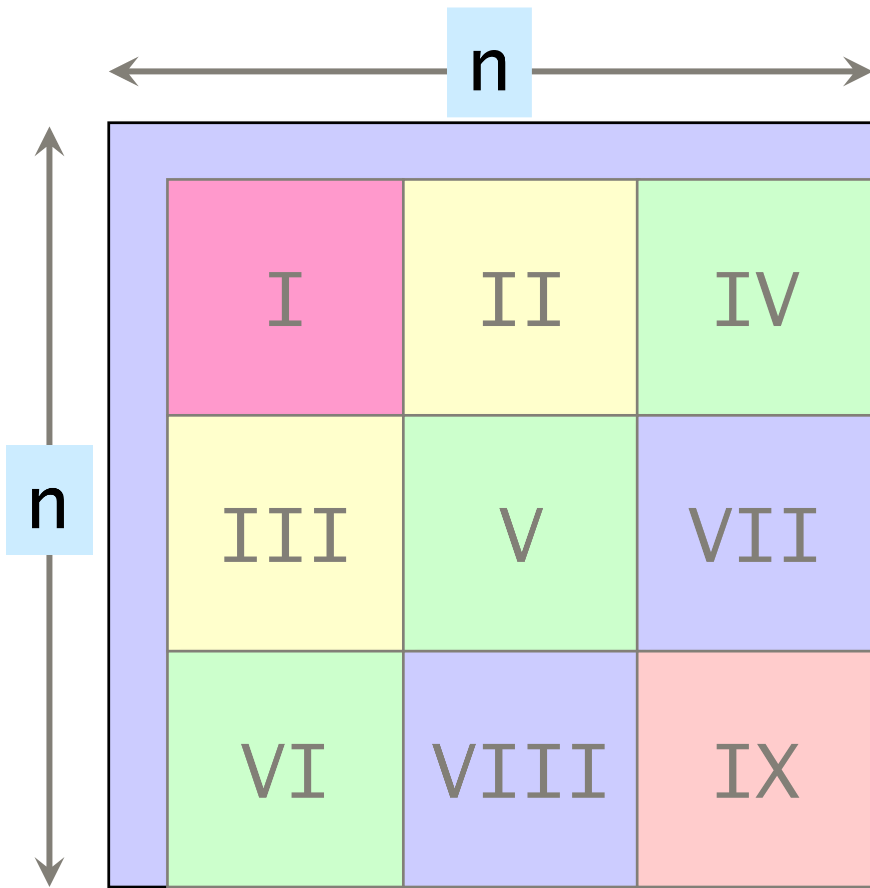
$f(n) = \Theta(1)$

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

$= \Theta(n^2)$

© 2010 Charles E. Leiserson

A More-Parallel Method




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

CASE 1:

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

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

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

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

© 2010 Charles E. Leiserson

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 Cilk++?

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

© 2010 Charles E. Leiserson

MIT OpenCourseWare
http://ocw.mit.edu

6.172 Performance Engineering of Software Systems
Fall 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.