SIAM Annual Meeting

July 8, 2009

The Design and Analysis of Multithreaded Algorithms

Charles E. Leiserson

Cilk++

- Small set of linguistic extensions to C++ to support fork-join parallelism.
- Open-source product developed by Cilk Arts, Inc., an MIT spin-off.
- Based on the award-winning Cilk multithreaded language developed at MIT.
- Features a provably efficient work-stealing scheduler.
- See Session MS38 — 10:30 A.M. today in Four Seasons I.

Nested Parallelism in Cilk++

```
int fib(int n)
{
   if (n < 2) return n
   int x, y;
   x = cilk_spawn fib(n-1);
   y = fib(n-2);
   cilk_sync;
   return x+y;
}
```

The named *child* function may execute in parallel with the *parent* caller.

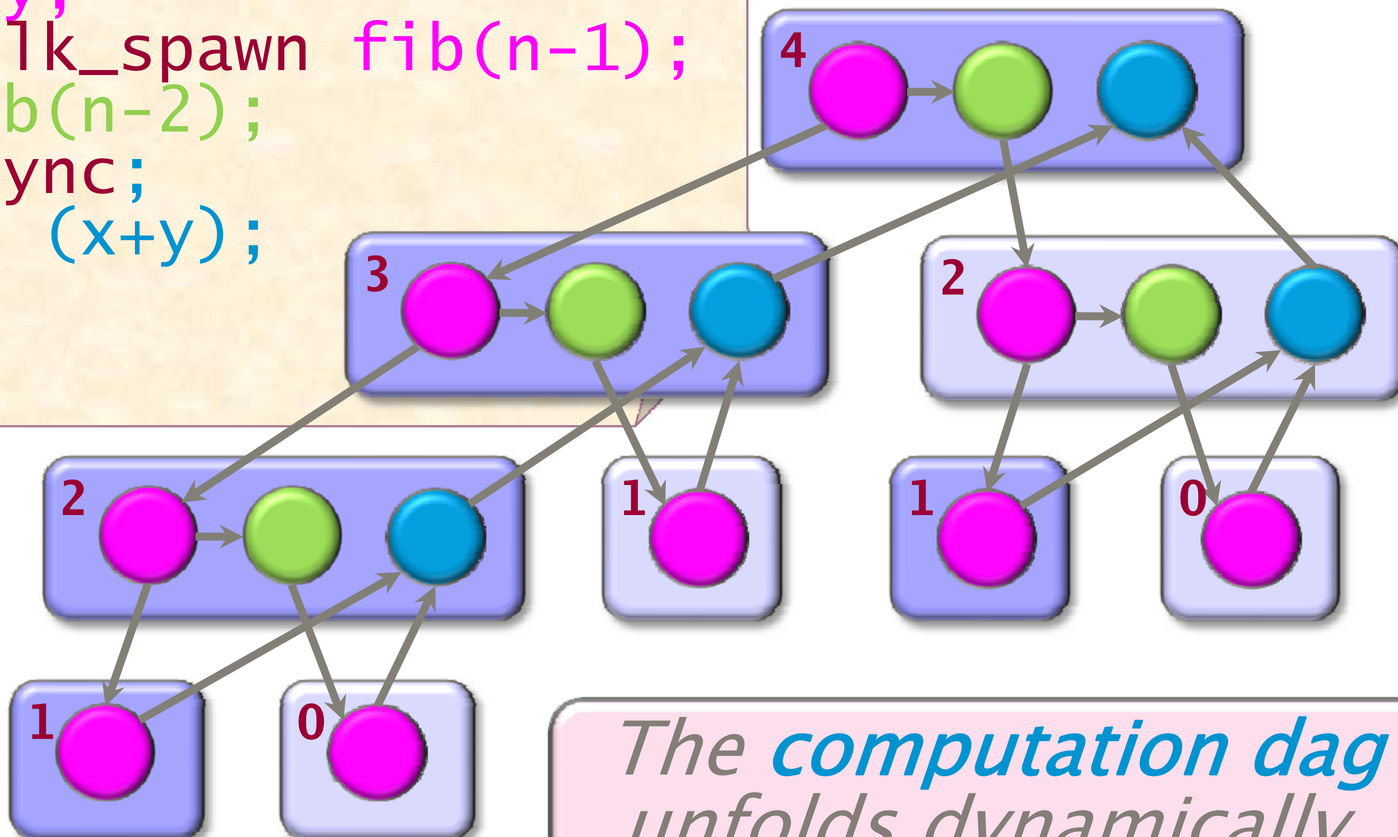
Control cannot pass this point until all spawned children have returned.

Cilk++ keywords *grant permission* for parallel execution. They do not *command* parallel execution.

Execution Model

```
int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x+y);
  }
}
```

Example:
`fib(4)`



"Processor oblivious"

*The **computation dag** unfolds dynamically.*

OUTLINE

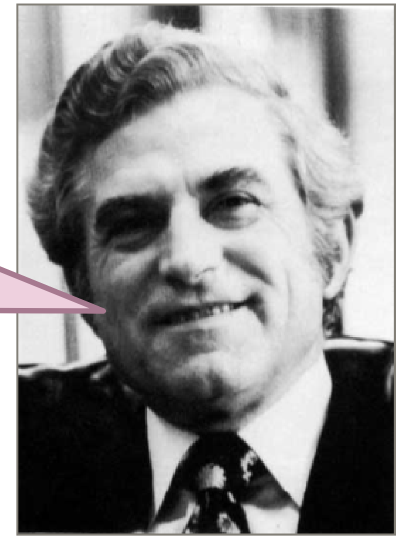
- What the $#@! Is Parallelism, Anyhow?
- Ins and Outs of Parallel Loops
- A Refresher on Recurrences
- A New Look at Matrix Multiplication
- All's Well That Ends Well

OUTLINE

- What the $#@! Is Parallelism, Anyhow?
- Ins and Outs of Parallel Loops
- A Refresher on Recurrences
- A New Look at Matrix Multiplication
- All's Well That Ends Well

Amdahl's Law

If 50% of your application is parallel and 50% is serial, you can't get more than a factor of 2 speedup, no matter how many processors it runs on.*



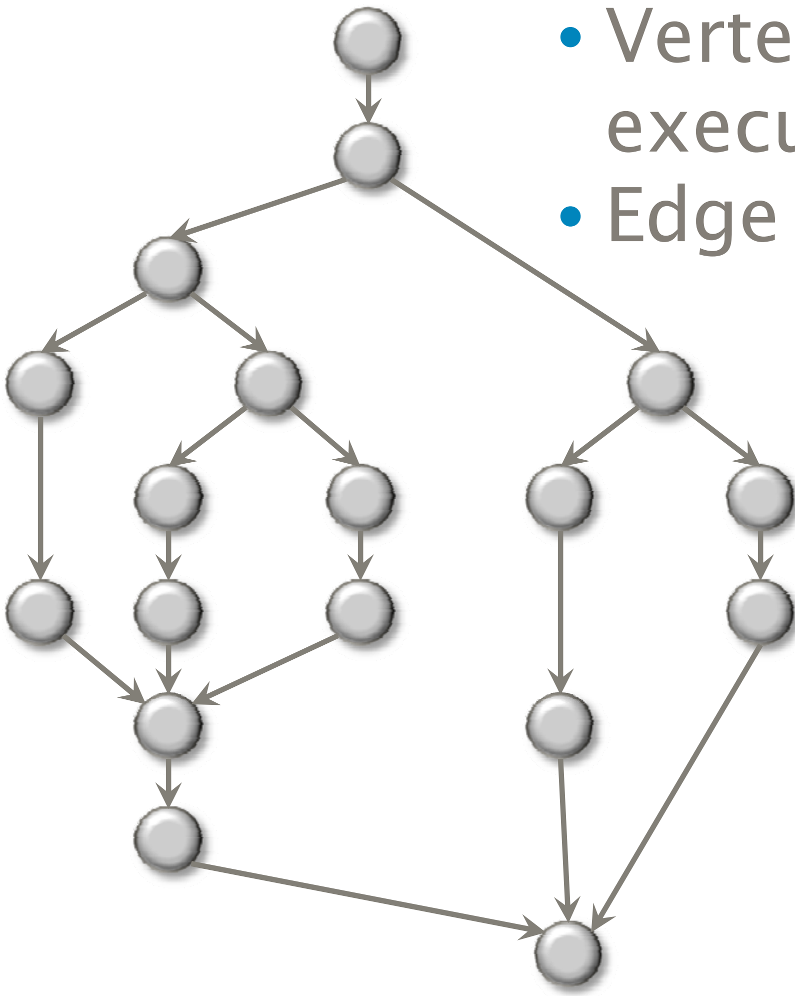
Gene M. Amdahl

*In general, if a fraction p of an application can be run in parallel and the rest must run serially, the speedup is at most $1/(1-p)$.

But, whose application can be decomposed into just a serial part and a parallel part? For *my* application, what speedup should I expect?

Simple Theoretical Model

Computation dag for an application
- Vertex = *strand* (serial chain of executed instructions)
- Edge = control *dependency*

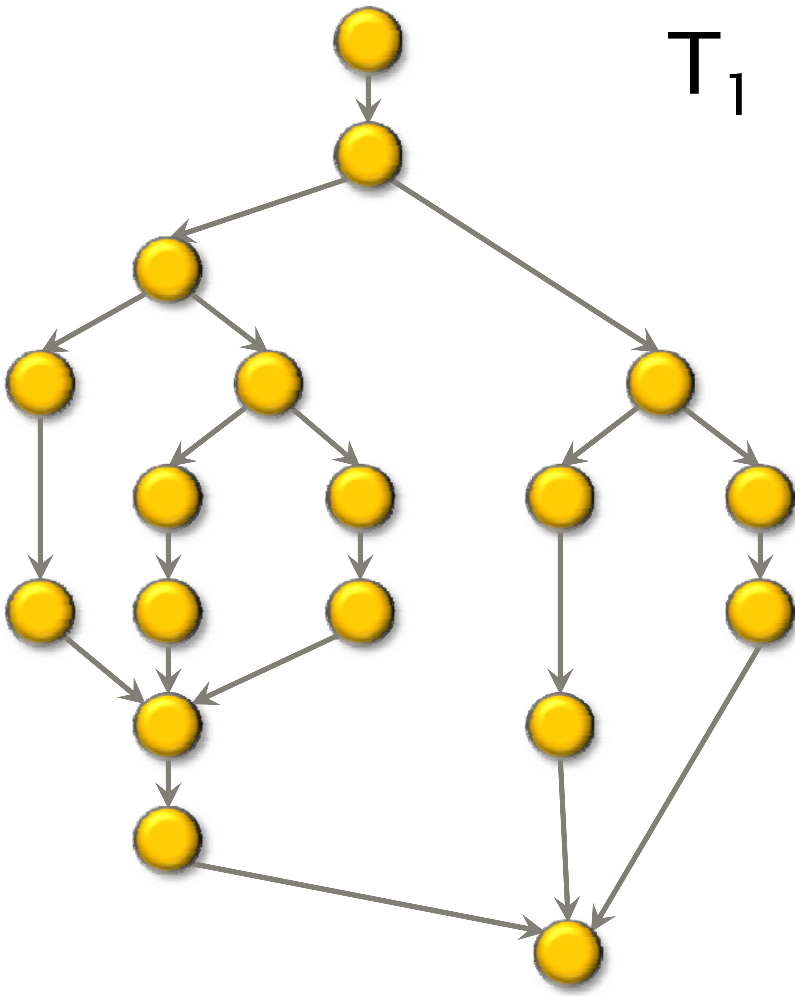


A typical multicore *concurrency platform* (e.g., Cilk, Cilk++, Fortress, OpenMP, TBB, X10) contains a runtime *scheduler* that maps the computation onto the available processors.

Complexity Measures

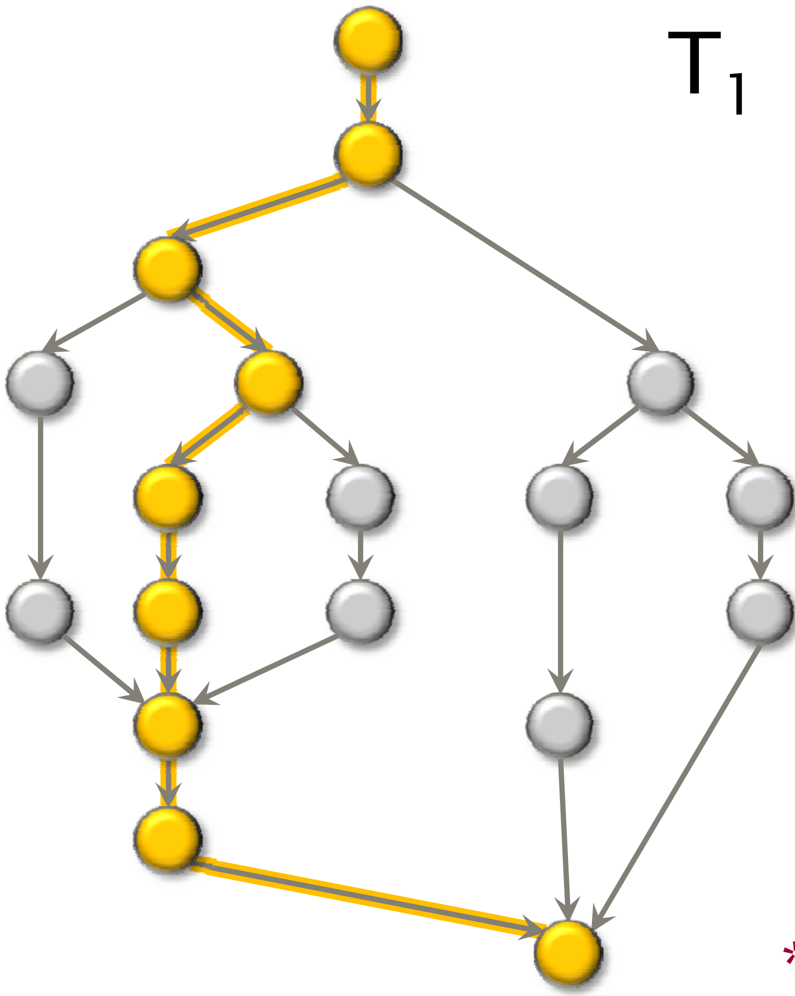
T_P = execution time on P processors

T_1 = *work*



Complexity Measures

T_P = execution time on P processors

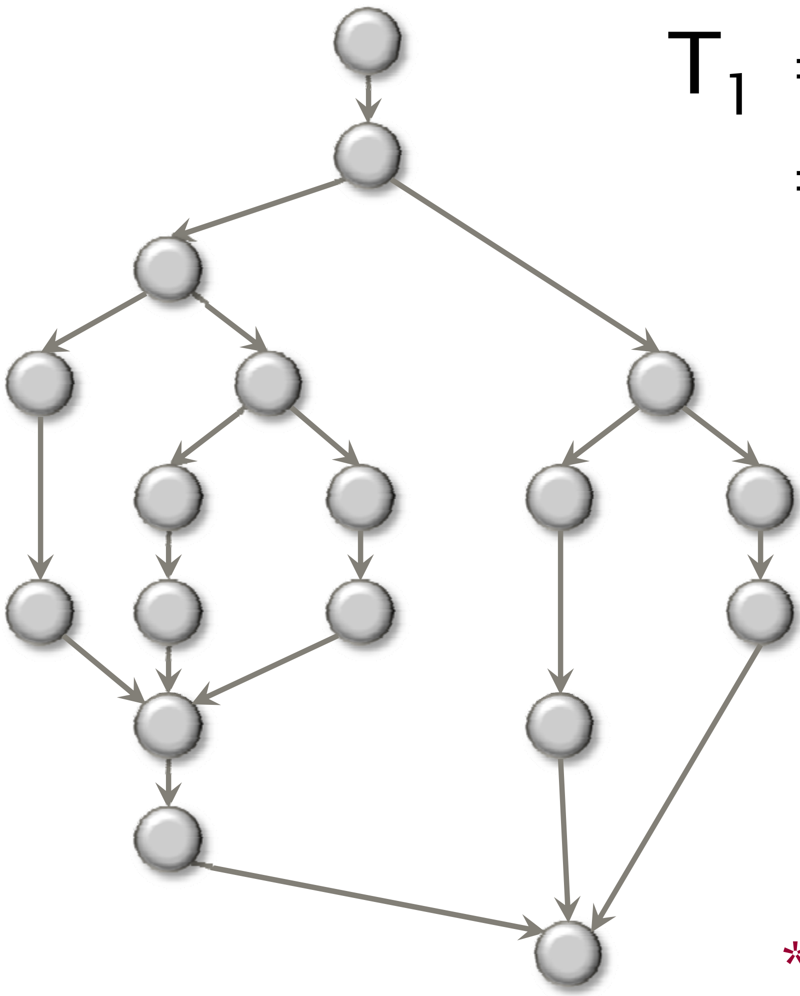


T_1 = ***work*** T_∞ = ***span****

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

Complexity Measures

T_P = execution time on P processors



T_1 = *work* $= 18$

T_∞ = *span** $= 9$

WORK LAW

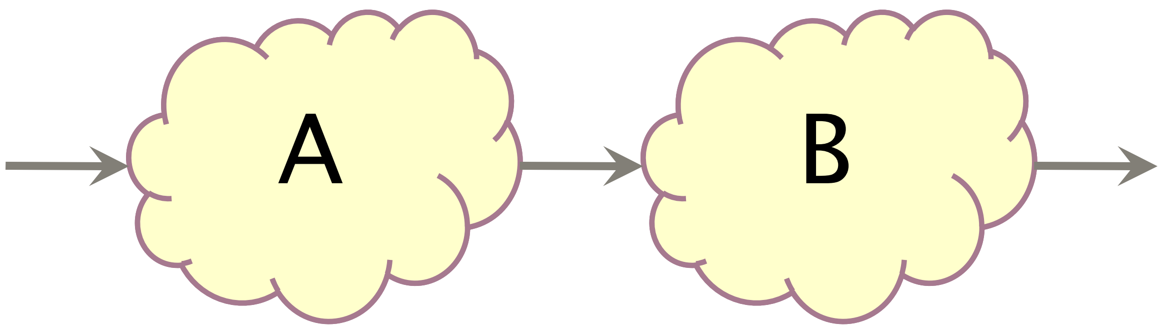
- $T_P \geq T_1/P$

SPAN LAW

- $T_P \geq T_\infty$

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

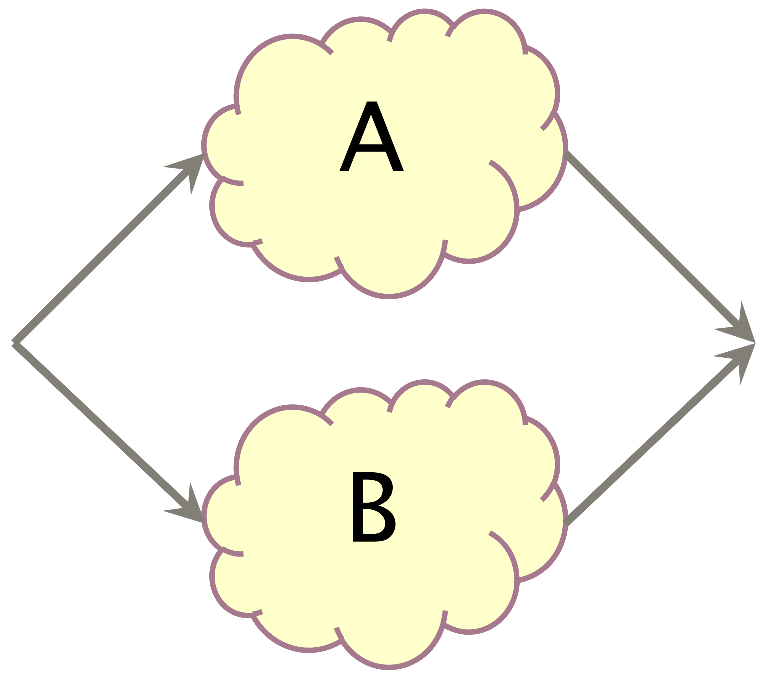
Series Composition



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

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

Parallel Composition



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

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

Speedup

Def. T_1/T_P = ***speedup*** on P processors.

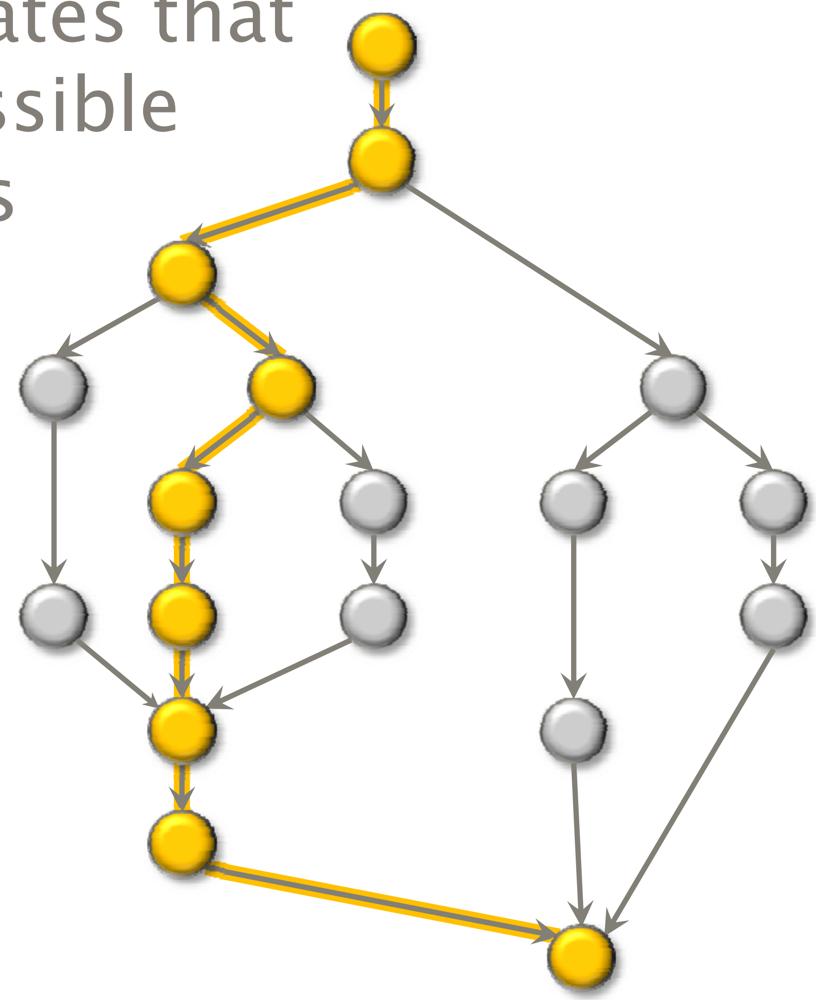
If $T_1/T_P = \Theta(P)$, we have *linear speedup*,
$= P$, we have *perfect linear speedup*,
$> P$, we have *superlinear speedup*,
which is not possible in this performance model, because of the Work Law $T_P \geq T_1/P$.

Parallelism

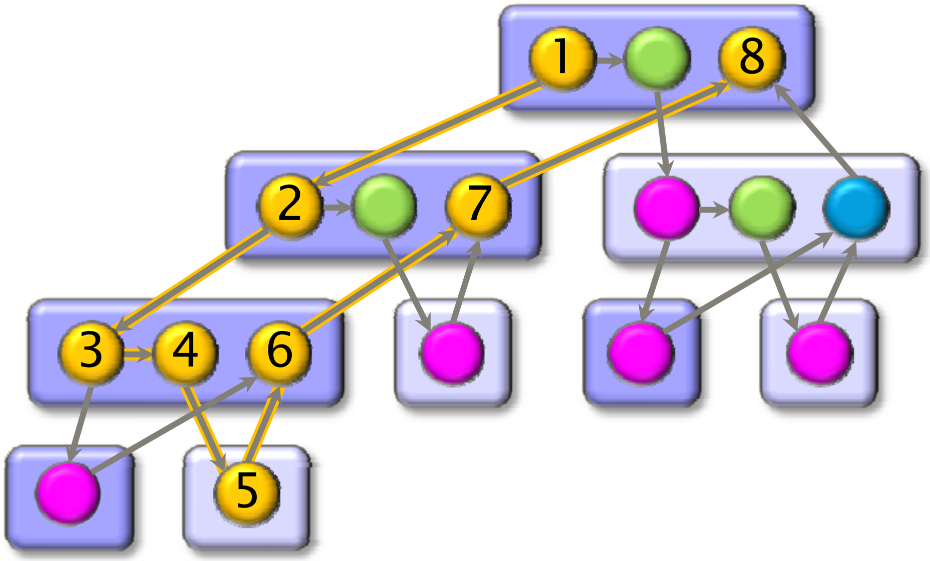
Because the Span Law dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = *parallelism*

= the average amount of work per step along the span



Example: `fib(4)`



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

Work: $T_1 = 17$

Span: $T_\infty = 8$

Parallelism: $T_1/T_\infty = 2.125$

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

Provably Good Scheduling

Theorem. Cilk++'s randomized work-stealing scheduler achieves expected time

$$T_P \le T_1/P + O(T_\infty). \quad \blacksquare$$

Corollary. Near-perfect linear speedup when

$$T_1/T_\infty \gg P,$$

i.e., ample ***parallel slackness***.

Proof. Since $T_1/T_\infty \gg P$ is equivalent to $T_\infty \ll T_1/P$, we have

$$\begin{aligned} T_P &\le T_1/P + O(T_\infty) \\ &\approx T_1/P . \end{aligned}$$

Thus, the speedup is $T_1/T_P \approx P$. $\blacksquare$

OUTLINE

- What the $#@! Is Parallelism, Anyhow?
- Ins and Outs of Parallel Loops
- A Refresher on Recurrences
- A New Look at Matrix Multiplication
- All's Well That Ends Well

Loop Parallelism in Cilk++

Example: In-place matrix transpose

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

$$A \qquad\qquad\qquad A^T$$

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) {
        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);
```

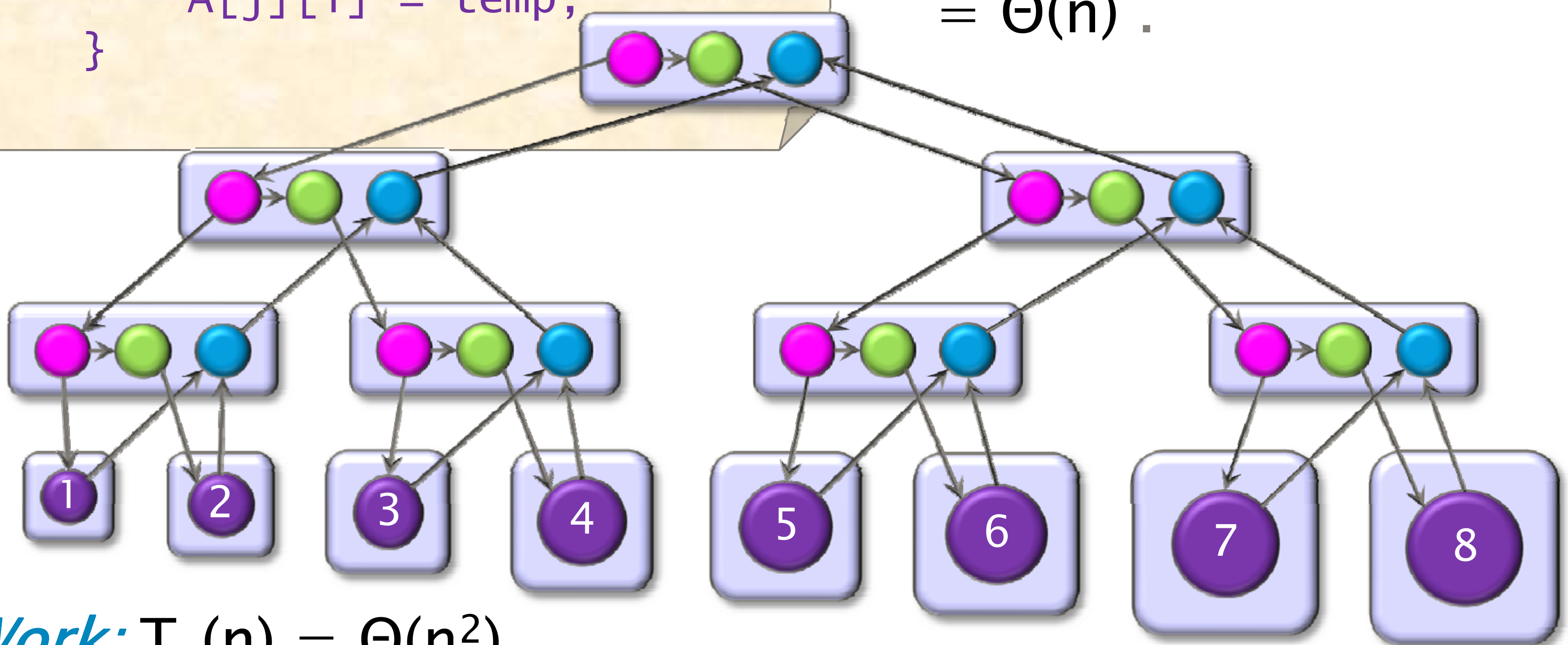
In practice, the recursion is ***coarsened*** to minimize overheads.

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

Analysis of Nested 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: $T_1(n)/T_\infty(n) = \Theta(n^2/\lg n)$

OUTLINE

- What the $#@! Is Parallelism, Anyhow?
- Ins and Outs of Parallel Loops
- A Refresher on Recurrences
- A New Look at Matrix Multiplication
- All's Well That Ends Well

The Master Method

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

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

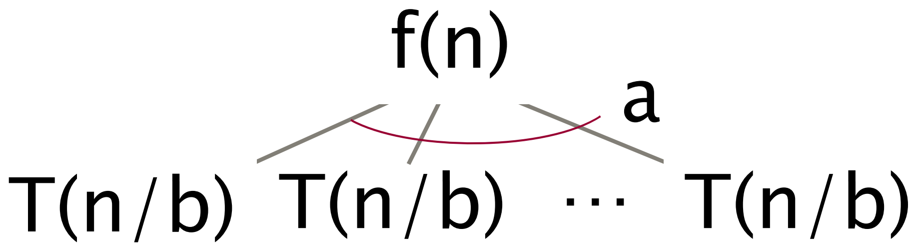
where $a \ge 1$, $b > 1$, and f is asymptotically positive.

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

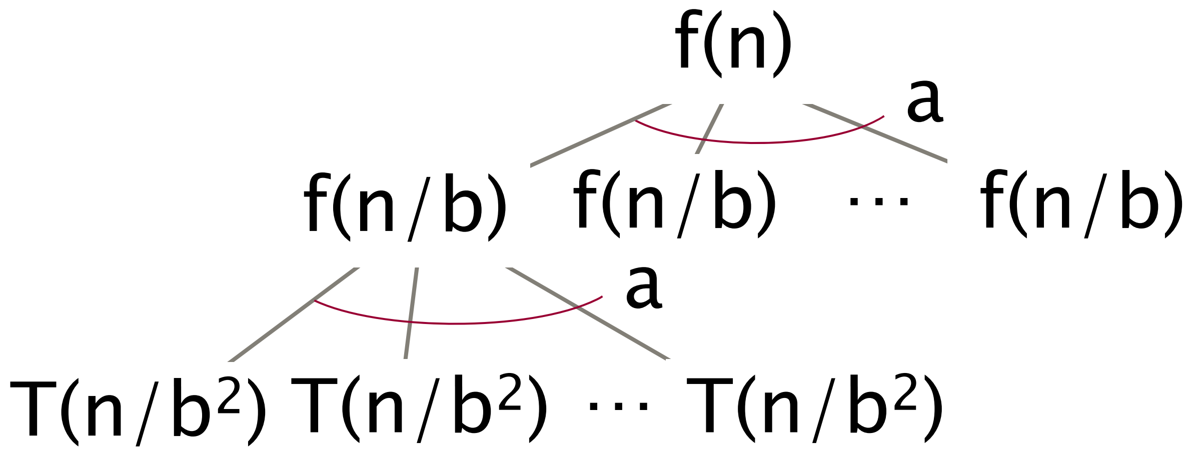
Recursion Tree: $T(n) = a\,T(n/b) + f(n)$

$$T(n)$$

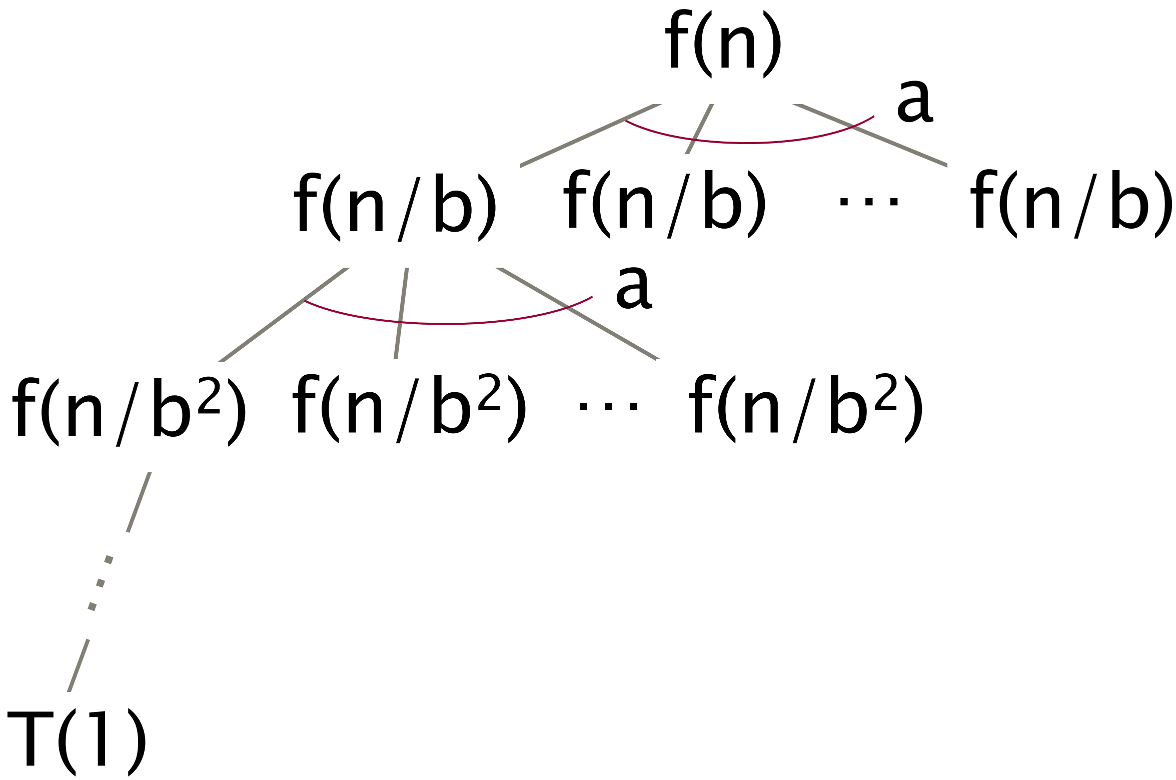
Recursion Tree: $T(n) = a\,T(n/b) + f(n)$



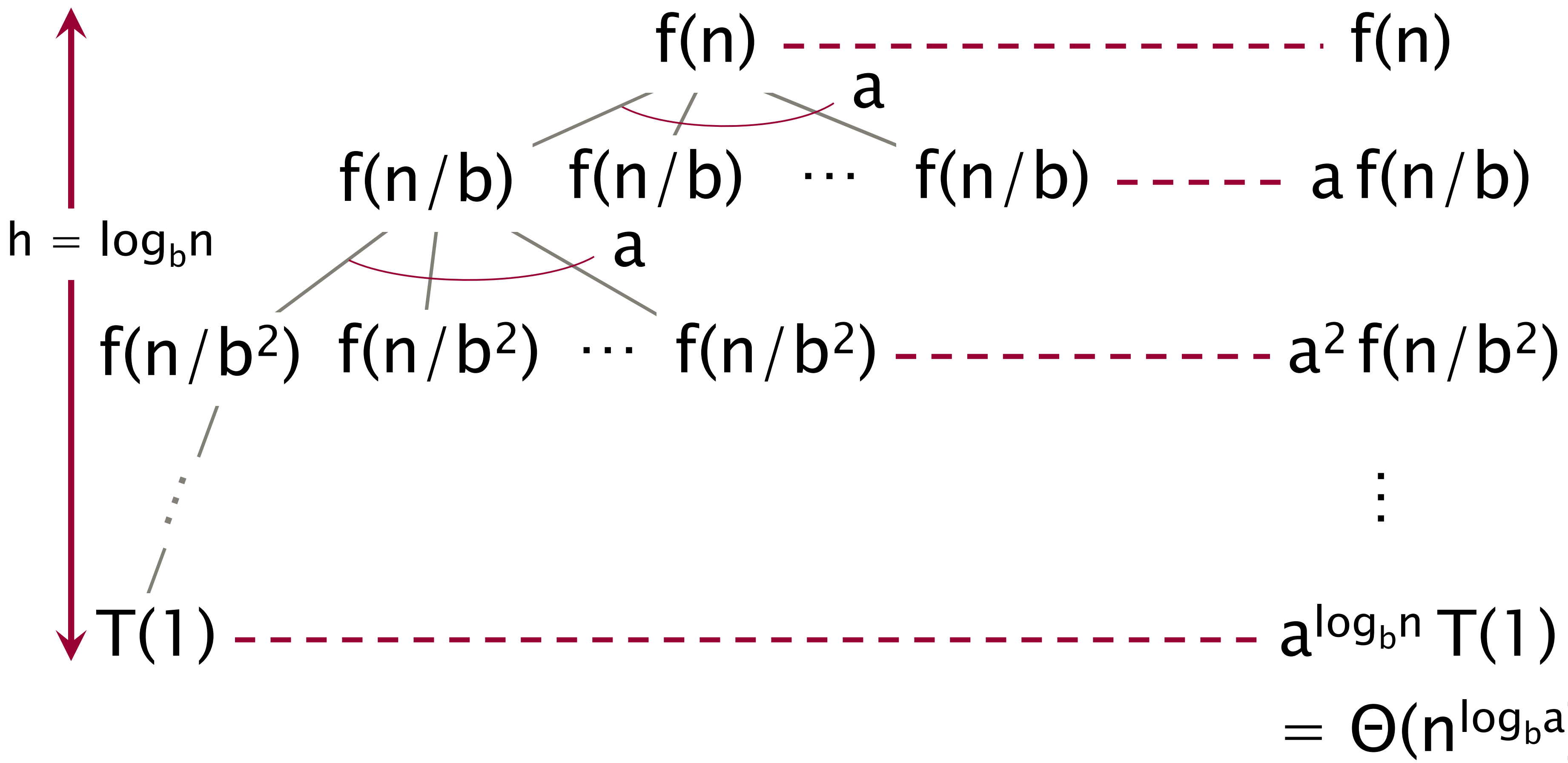
Recursion Tree: $T(n) = a\,T(n/b) + f(n)$



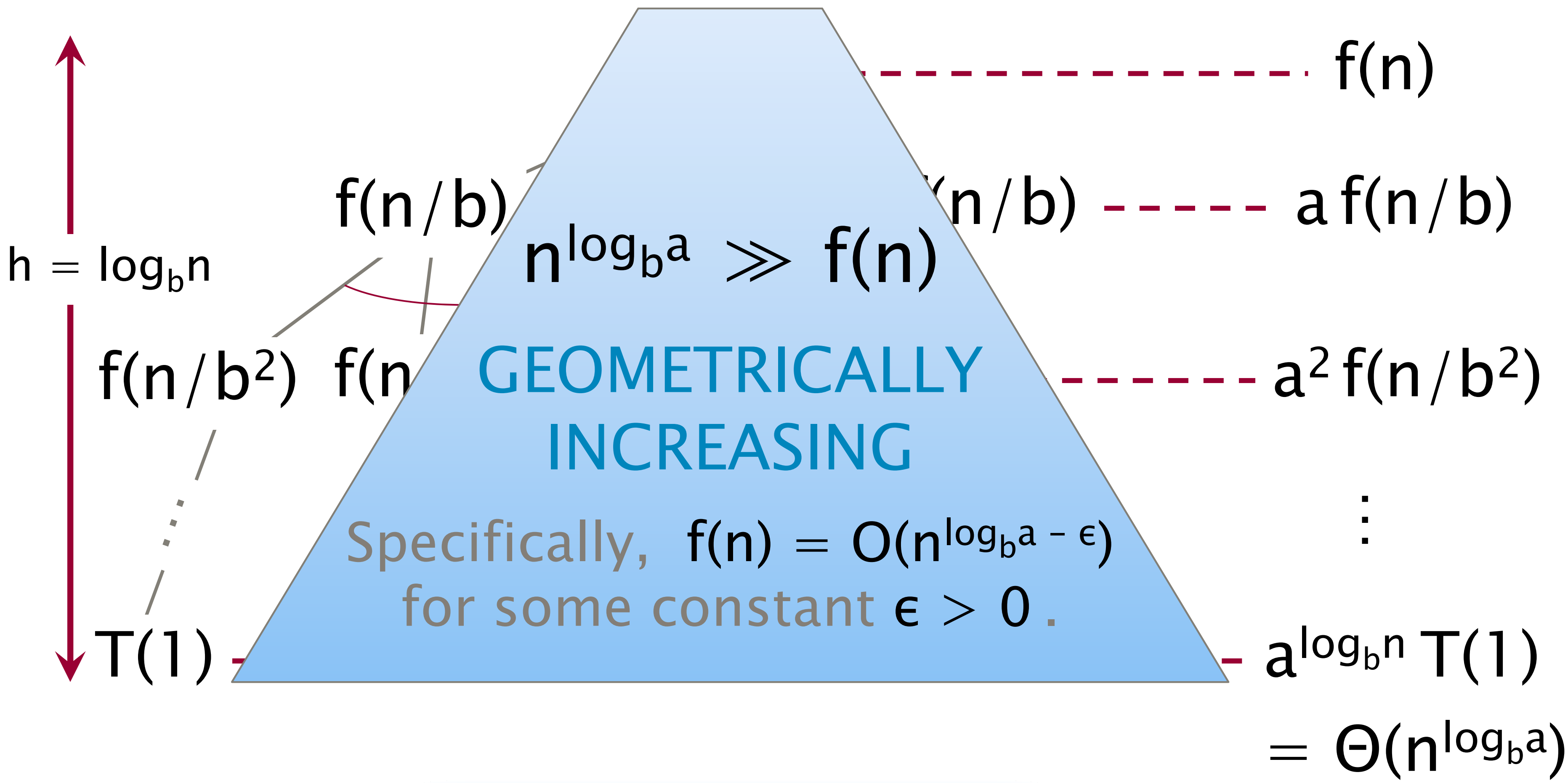
Recursion Tree: $T(n) = a\,T(n/b) + f(n)$



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



Master Method — Case I



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

Master Method — Case 2

$h = \log_b n$

$f(n/b^2)$

$T(1)$

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

ARITHMETICALLY INCREASING

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

$f(n)$

$a\,f(n/b)$

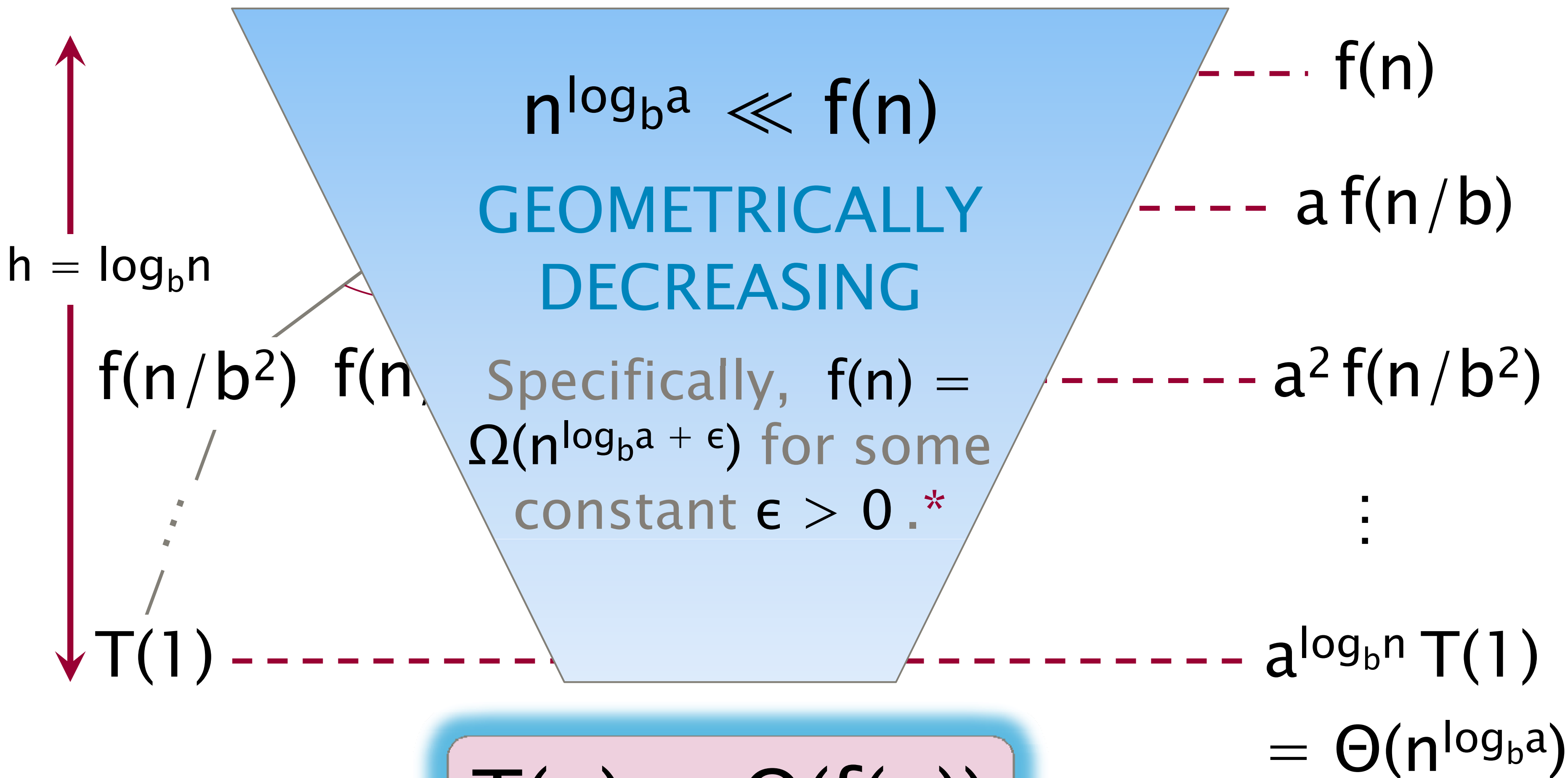
$a^2 f(n/b^2)$

$\vdots$

$a^{\log_b n}\, T(1) = \Theta(n^{\log_b a})$

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

Master Method — Case 3



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

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

Master-Method Cheat Sheet

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

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)

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

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!

OUTLINE

- What the $#@! Is Parallelism, Anyhow?
- Ins and Outs of Parallel Loops
- A Refresher on Recurrences
- A New Look at Matrix Multiplication
- All's Well That Ends Well

Square-Matrix Multiplication

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

$$C \qquad\qquad A \qquad\qquad B$$

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

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

Parallelizing Matrix Multiply

```
// index from 0, not 1
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: $T_1 = \Theta(n^3)$

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,   ze);
  cil  spawn MMult(C1 , A11, B12, n/2   ze);
  cil   pawn MMult(C2 , A21,            );
             MMult(C2   A21,            );
              (D1   A12,                );
              (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;
}
```

Coarsen for efficiency

Determine submatrices by index calculation

Row length of matrices

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];
    }
  }
}
```

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

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
  ⋮
  cilk_spawn MMult(D22
             MMult(D21
  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

maximum

```
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;
}
```

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 $\approx (10^3)^3/10^2 = 10^7$.

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?

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          11, n/2, size);
  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(C                         );
  cilk_sync;
  cilk_spawn MMult2(C1                        );
  cilk_spawn MMult2(C1                        );
  cilk_spawn MMult2(C2                        );
             MMult2(C21,             21, n/2, size);
  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!

OUTLINE

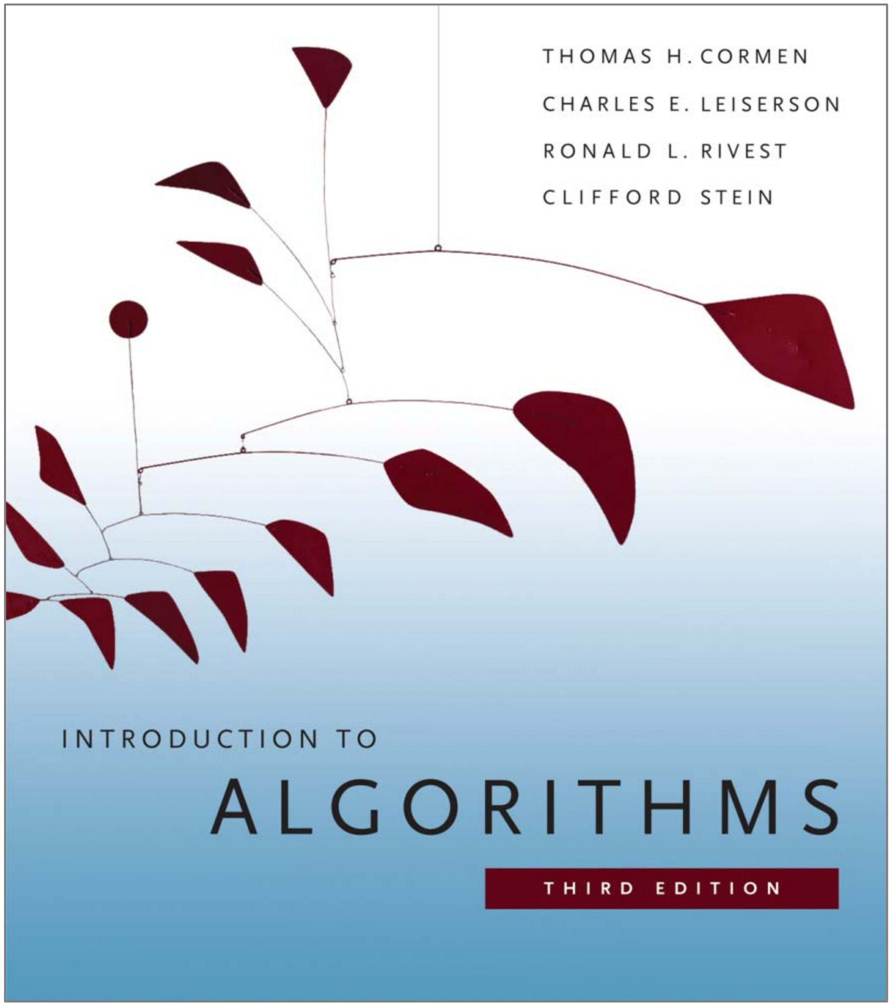
- What the $#@! Is Parallelism, Anyhow?
- Ins and Outs of Parallel Loops
- A Refresher on Recurrences
- A New Look at Matrix Multiplication
- All's Well That Ends Well

Interesting Practical* Algorithms

Algorithm	Work	Span
Strassen	$\Theta(n^{\lg 7})$	$\Theta(\lg^2 n)$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \lg n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(\lg^3 n)$
Tableau construction	$\Theta(n^2)$	$\Omega(n^{\lg 3})$
FFT	$\Theta(n \lg n)$	$\Theta(n / \lg n)$
SpMV (CSR)	$\Theta(nnz)$	$\Theta(\lg n)$
SpMV & SpMV_T (CSB)	$\Theta(nnz)$	$\Theta(\sqrt{nnz} \lg n)$
Breadth-first search	$\Theta(E)$	$\Theta(E/d \lg V)$

*Cilk++ work ≈ work of competive C++ algorithm.

Introduction to Algorithms



- The 3rd edition of *Introduction to Algorithms* will be available in August.
- It features a new chapter on the design and analysis of multithreaded algorithms.

Repeat in Unison

OH–WA

TAH–GOO

SIAM

Repeat until you attain enlightenment!