CS528
Cilk

Slides are adopted from
http://supertech.csail.mit.edu/cilk/
Charles E. Leiserson

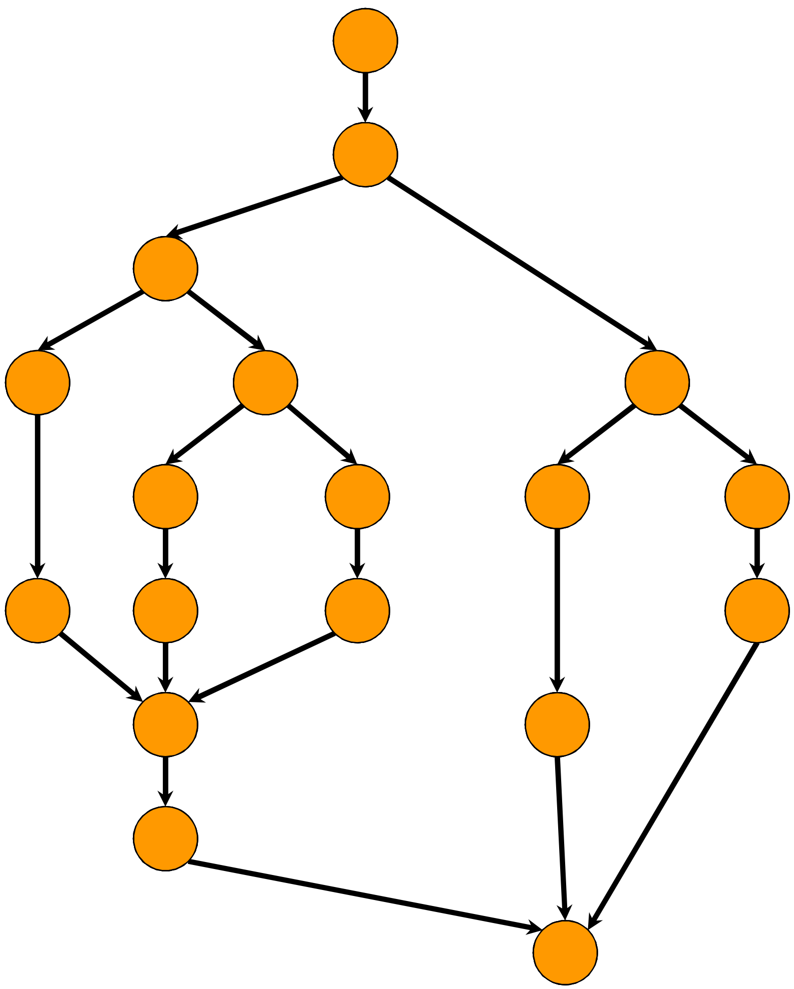
A Sahu
Dept of CSE, IIT Guwahati

Cilk

- Developed by **Leiserson at CSAIL, MIT**
  - **Chapter 27, Multithreaded Algorithm, Introduction to Algorithm, Coreman, Leiserson and Rivest**
- Initiated a startup: Cilk Plus
  - Added Cilk_for Keyword, Cilk Reduction features
  - Acquired by Intel, Intel uses Cilk Scheduler
- Addition of 6 keywords to standard C
  - Easy to install in linux system
  - With gcc and pthread

Algorithmic Complexity Measures

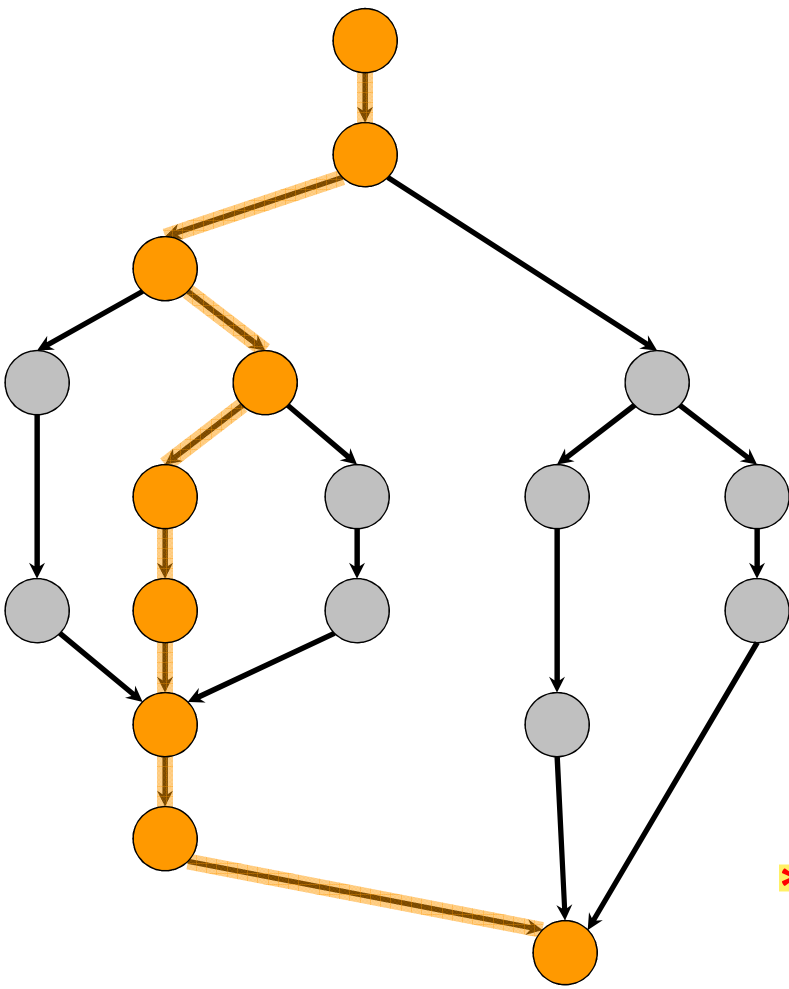
T_P = execution time on P processors



T_1 = ***work***

Algorithmic Complexity Measures

T_P = execution time on P processors



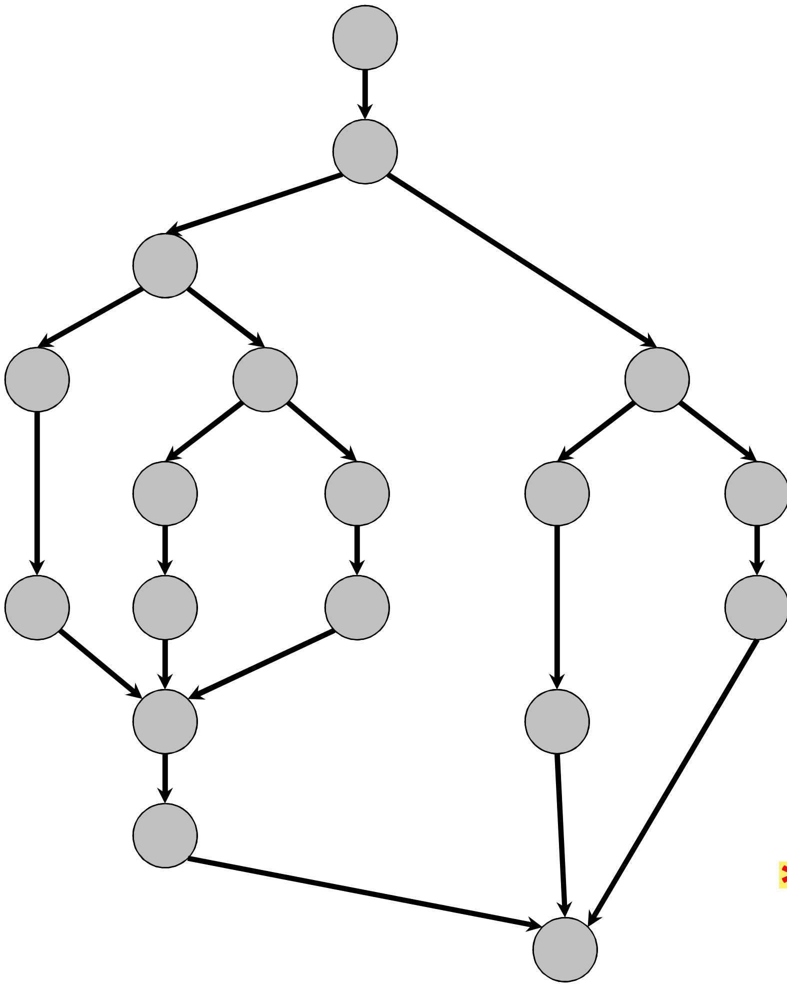
T_1 = ***work***

T_∞ = ***span*** *

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

Algorithmic Complexity Measures

T_P = execution time on P processors



T_1 = ***work***

T_∞ = ***span****

LOWER BOUNDS

- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

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

Speedup

Definition: T_1/T_P = ***speedup*** on P processors.

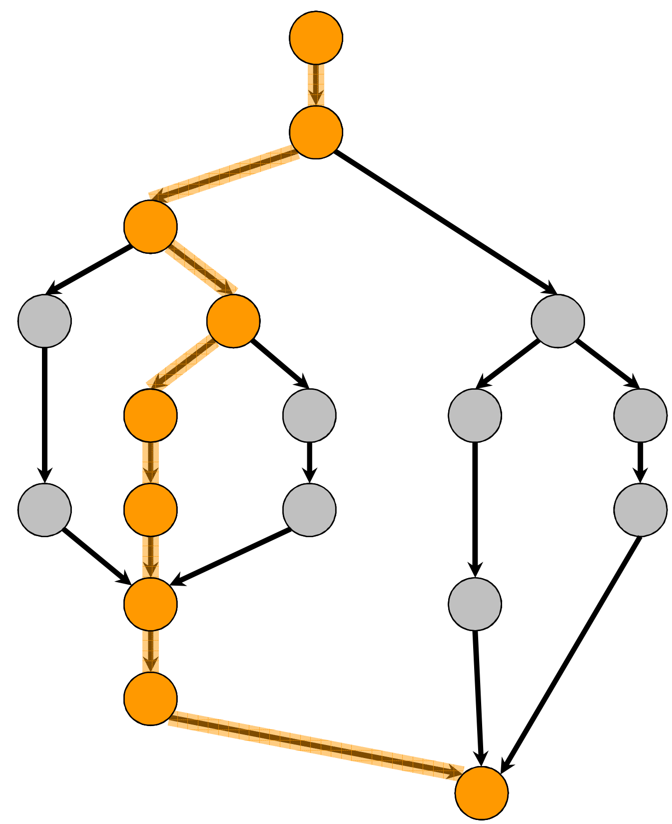
If $T_1/T_P = \Theta(P) < P$, we have ***linear speedup***;
$= P$, we have ***perfect linear speedup***;
$> P$, we have ***superlinear speedup***,
which is not possible in our model, because of the lower bound $T_P \geq T_1/P$.

Parallelism

Because we have the lower bound $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

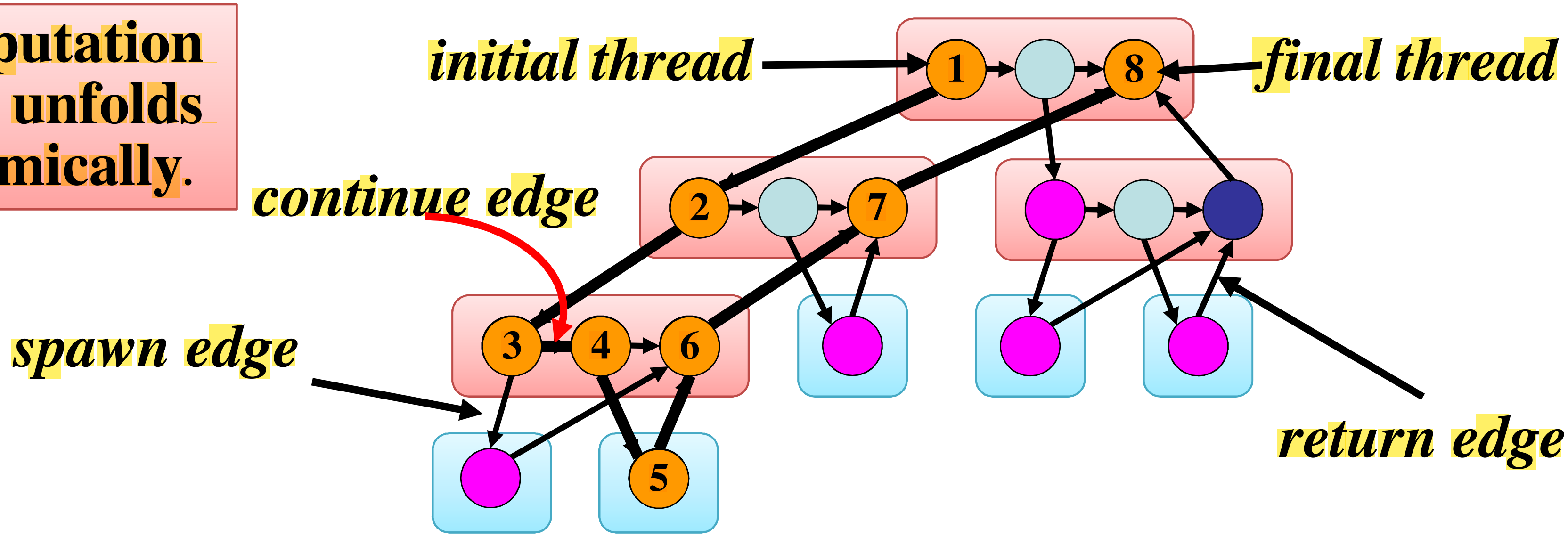
T_1 / T_∞ = *parallelism*

= the average amount of work per step along the span.



CILK Example: Fib(4)

Computation DAG unfolds dynamically.



Assume for simplicity that each Cilk thread in `fib()` takes unit time to execute.

Work: $T_1 = 17$

Span: $T_\infty = 8$

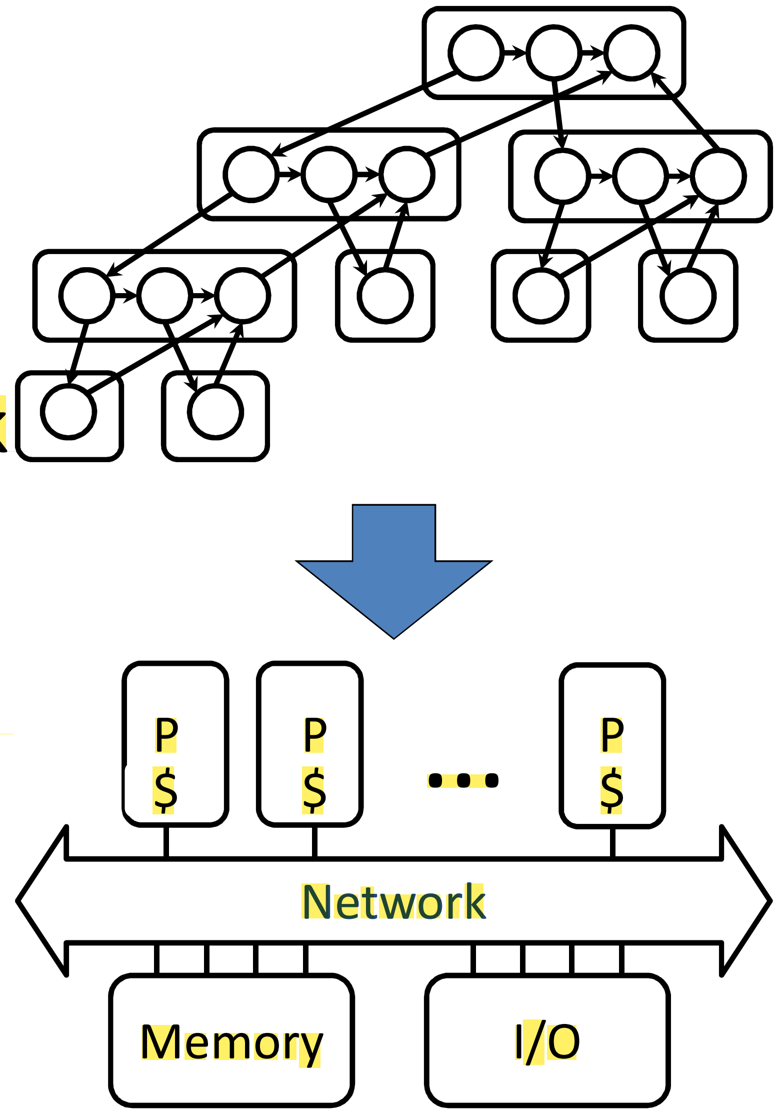
Parallelism: $T_1/T_\infty = 2.125$

Using many more than **2** processors makes little sense.

Ref1: The Cilk System for Parallel Multithreaded Computing, MIT PhD Thesis
Ref2: The Implementation of the Cilk-5 Multithreaded Language, 1998 ACM SIGPLAN

Scheduling

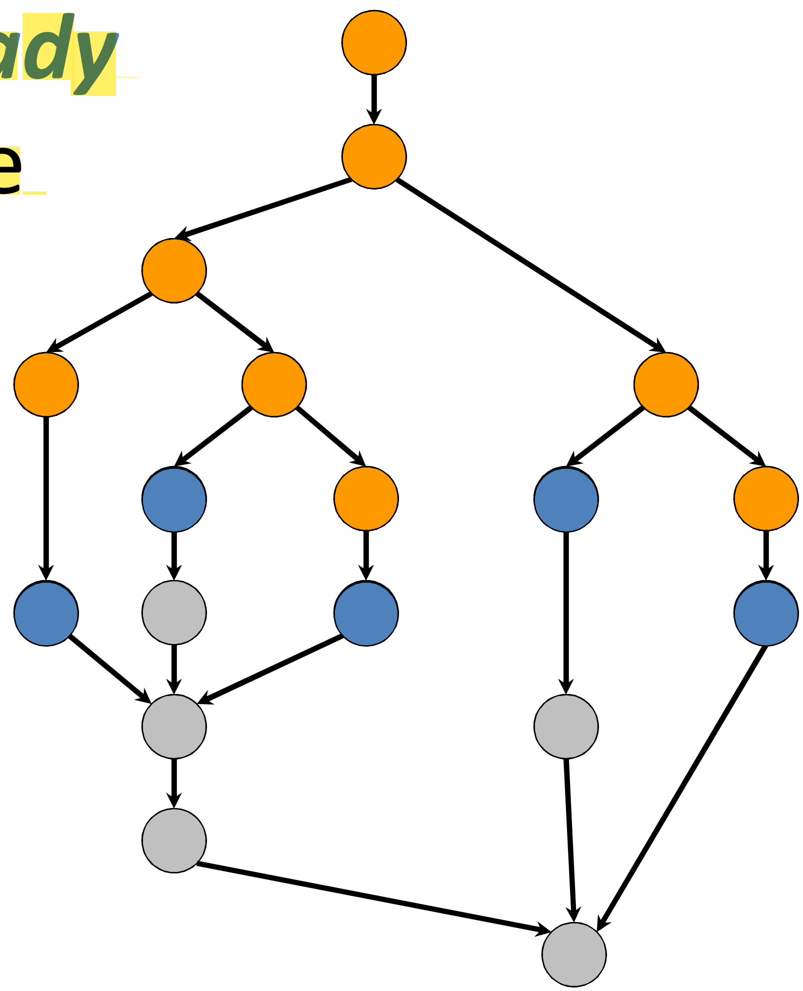
- Cilk allows the programmer to express *potential* parallelism in an application.
- The Cilk ***scheduler*** maps Cilk threads onto processors dynamically at runtime.
- Since ***on-line*** schedulers are complicated, we'll illustrate the ideas with an ***off-line*** scheduler.



Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A thread is ***ready*** if all its predecessors have ***executed***.



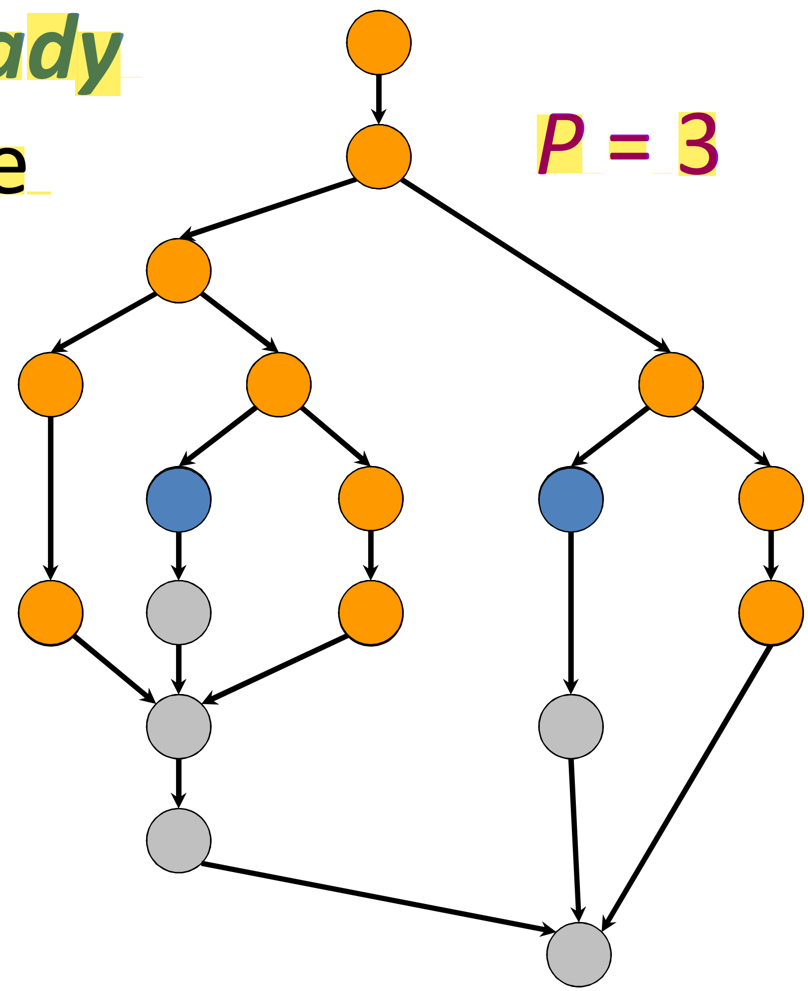
Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A thread is ***ready*** if all its predecessors have ***executed***.

Complete step

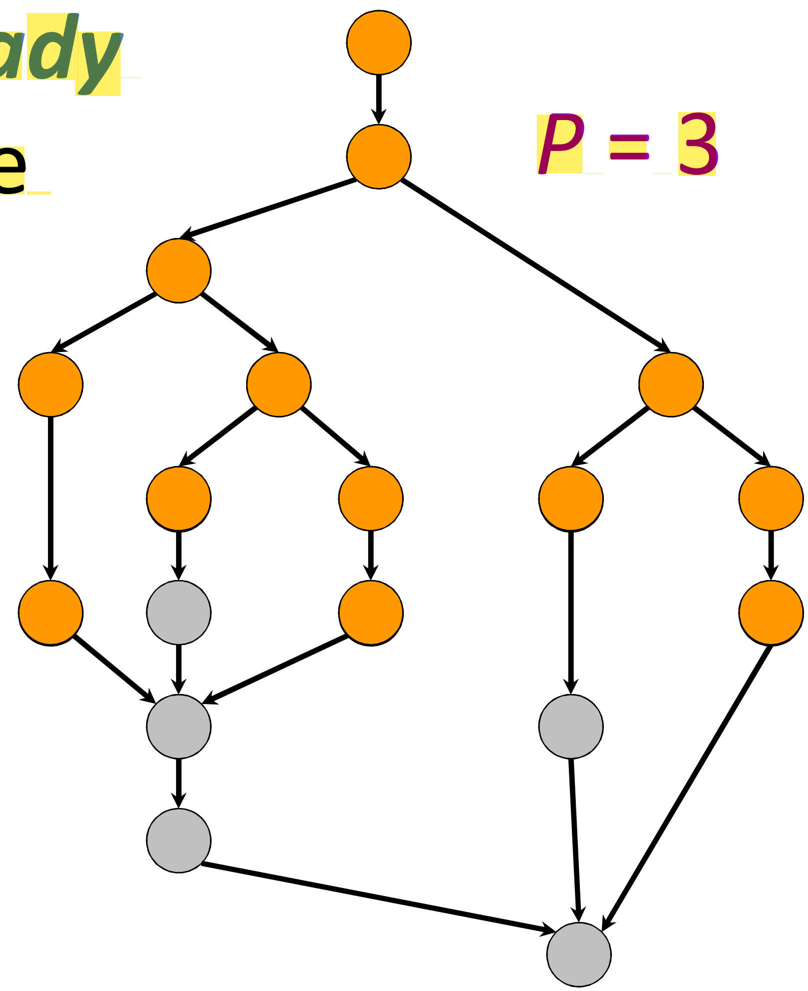
- ≥ P threads ready.
- Run any P.



Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition: A thread is ***ready*** if all its predecessors have ***executed***.



Complete step

- $\geq P$ threads ready.
- Run any P.

Incomplete step

- $< P$ threads ready.
- Run all of them.

Greedy-Scheduling Theorem

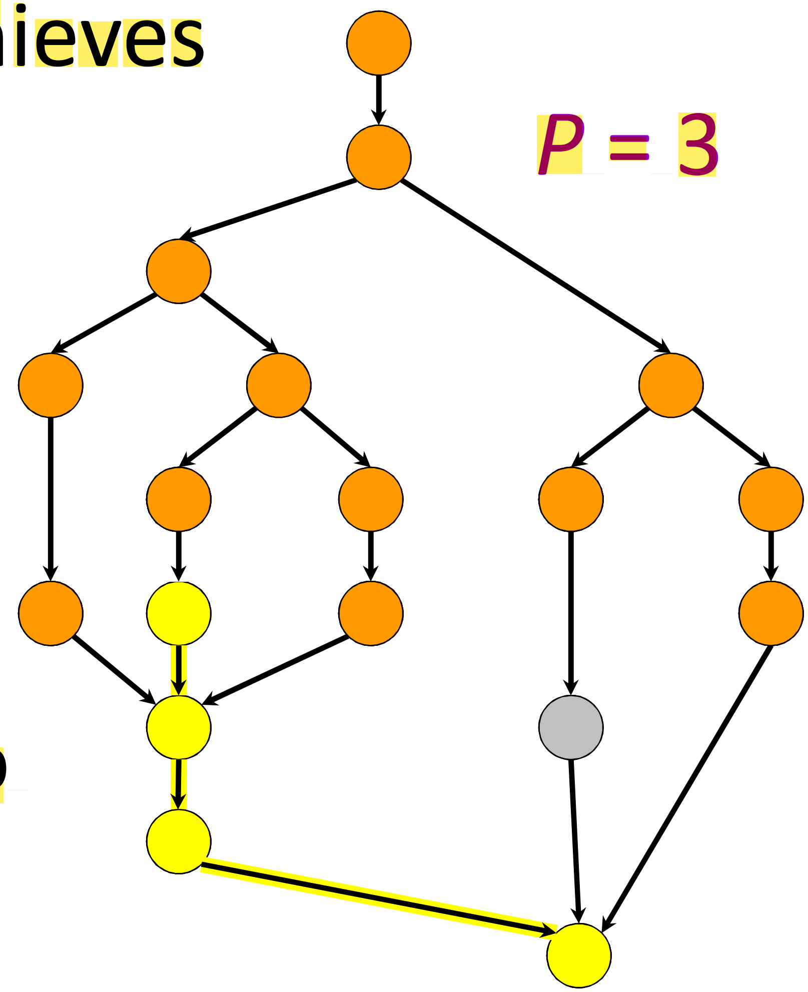
Theorem [Graham '68 & Brent '75].

Any greedy scheduler achieves

$$T_P \leq T_1/P + T_\infty.$$

Proof.

- # complete steps $\leq T_1/P$, since each complete step performs P work.
- # incomplete steps $\leq T_\infty$, since each incomplete step reduces the span of the unexecuted dag by 1. ■



Optimality of Greedy

Corollary. Any greedy scheduler achieves within a factor of 2 of optimal.

Proof. Let T_P* be the execution time produced by the optimal scheduler. Since $T_P^* \geq \max\{T_1/P, T_\infty\}$ (lower bounds), we have

$$\begin{aligned} T_P &\leq T_1/P + T_\infty \\ &\leq 2\max\{T_1/P, T_\infty\} \\ &\leq 2T_P^* . \quad \blacksquare \end{aligned}$$

Linear Speedup

Corollary. Any greedy scheduler achieves near-perfect linear speedup whenever $T_1/T_\infty \gg P$

Proof. Since $T_1/T_\infty \gg P$ ➔ $T_\infty \ll T_1/P$, the Greedy Scheduling Theorem gives us

$$T_P \le T_1/P + T_\infty$$
$$\approx T_1/P .$$

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

Definition. The quantity $(T_1/ T_\infty)/P$ is called the ***parallel slackness***.

Cilk Performance

- Cilk's "work-stealing" scheduler achieves
  - $T_P = T_1/P + O(T_\infty)$ expected time (provably);
  - $T_P \approx T_1/P + T_\infty$ time (empirically).
- Near-perfect linear speedup if $P \ll T_1/T_\infty$.
- Instrumentation in Cilk allows the user to determine accurate measures of T_1 and T_∞ .
- The average cost of a `spawn` in Cilk-5 is only 2–6 times the cost of an ordinary C function call, depending on the platform.

Parallelizing Vector Addition

C

```
void vadd(float *A, float *B, int N)
     int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Parallelization strategy:

1. Convert loops to recursion.

Parallelizing Vector Addition

C

```
void vadd(float *A, float *B, int N)
     int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

C

```
      void vadd(float *A, float *B, int N){

   if (n<=BASE) {
     int i; for (i=0; i<n; i++) A[i]+=B[i];
   } else {
                 vadd (A, B, n/2);
                 vadd (A+n/2, B+n/2, n/2);

}
```

Parallelization strategy:

1. Convert loops to recursion.
2. Insert Cilk keywords.

Parallelizing Vector Addition

C

```
void vadd(float *A, float *B, int N)
     int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Cilk

```
cilk void vadd(float *A, float *B, int N){

   if (n<=BASE) {
     int i; for (i=0; i<n; i++) A[i]+=B[i];
   } else {
         spawn  vadd (A, B, n/2);
         spawn  vadd (A+n/2, B+n/2, n/2);
         sync;
}
```

Parallelization strategy:

1. Convert loops to recursion.
2. Insert Cilk keywords.

Parallelizing Vector Addition

C

```
void vadd(float *A, float *B, int N)
     int i; for (i=0; i<n; i++) A[i]+=B[i];
}
```

Cilk

```
cilk void vadd(float *A, float *B, int N){

   if (n<=BASE) {
     int i; for (i=0; i<n; i++) A[i]+=B[i];
   } else {
        spawn  vadd (A, B, n/2);
        spawn  vadd (A+n/2, B+n/2, n/2);
        sync;
}
```

Parallelization strategy:

1. Convert loops to recursion.
2. Insert Cilk keywords.

Side benefit:

D&C is generally good for caches!

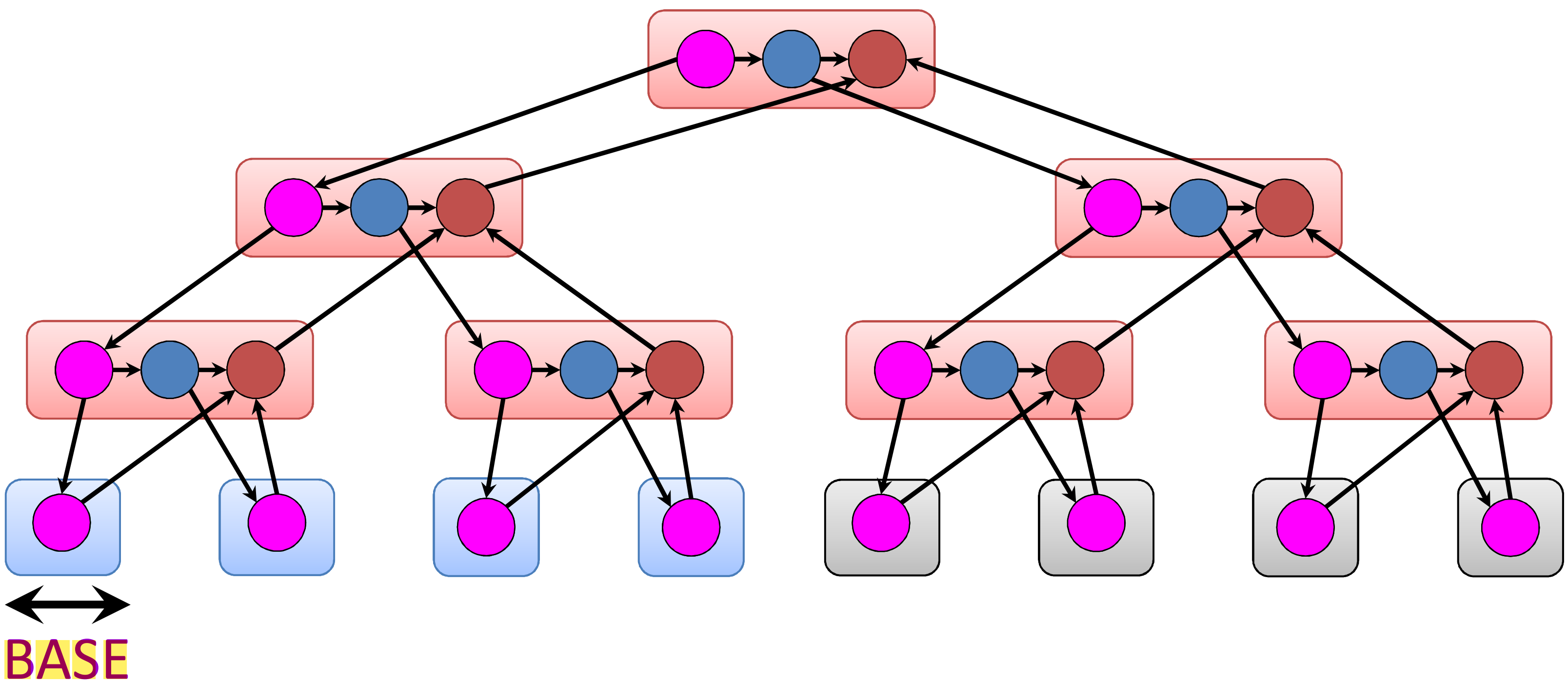
Vector Addition Analysis

To add two vectors of length n, where BASE = $\Theta(1)$:

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

Span: T_∞ = ? $\Theta(\lg n)$

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



Square-Matrix Multiplication

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

C A B

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

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

Recursive Matrix Multiplication

Divide and conquer —

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \mathrm{X} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

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

8 multiplications of $(n/2)$ X $(n/2)$ matrices.
1 addition of n X n matrices.

Matrix Multiply in Pseudo-Cilk

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

$C = A \times B$

Absence of type declarations.

Matrix Multiply in Pseudo-Cilk

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

$C = A \times B$

Coarsen base cases for efficiency.

Matrix Multiply in Pseudo-Cilk

```
cilk void Mult(*C, *A, *B, n) {
  float *T = Cilk_alloca(n*n*sizeof(float));
  h base case & partition matrices i
  spawn Mult(C11,A11,B11,n/2);
  spawn Mult(C12,A11,B12,n/2)
  spawn Mult(C22,A21,B12,n/2)
  spawn Mult(C21,A21,B  n/2)
  spawn Mult(T11,A12,B2  /2)
  spawn Mult(T12,A12,B22,
  spawn Mult(T22,A22,B22,n
  spawn Mult(T21,A22,B21,n/
  sync;
  spawn Add(C,T,n);
  sync;
  return;
}
```

C = A X B

Also need a row-size argument for array indexing.

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

Work of Matrix Addition

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

Work: $A_1(n) = 4\,A_1(n/2) + \Theta(1)$

$= \Theta(n^2)$

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

Span of Matrix Addition

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

maximum

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

$= \Theta(\lg n)$

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

Work of Matrix Multiplication

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

8

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

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

$= \Theta(n^3)$

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

Span of Matrix Multiplication

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

8

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

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

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

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

Parallelism of Matrix Multiply

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

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

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

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

Stack Temporaries

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

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

IDEA: Trade off parallelism for less storage.

No-Temp Matrix Multiplication

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

Saves space, but at what expense?

Work of No-Temp Multiply

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

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

$= \Theta(n^3)$

Span of No-Temp Multiply

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

maximum

maximum

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: $\dfrac{M_1(n)}{M_\infty(n)} = \Theta(n^2)$

For 1000 X 1000 matrices,

Parallelism = $(10^3)^3/10^3 = 10^6$.

Faster in practice!

Tableau Construction

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

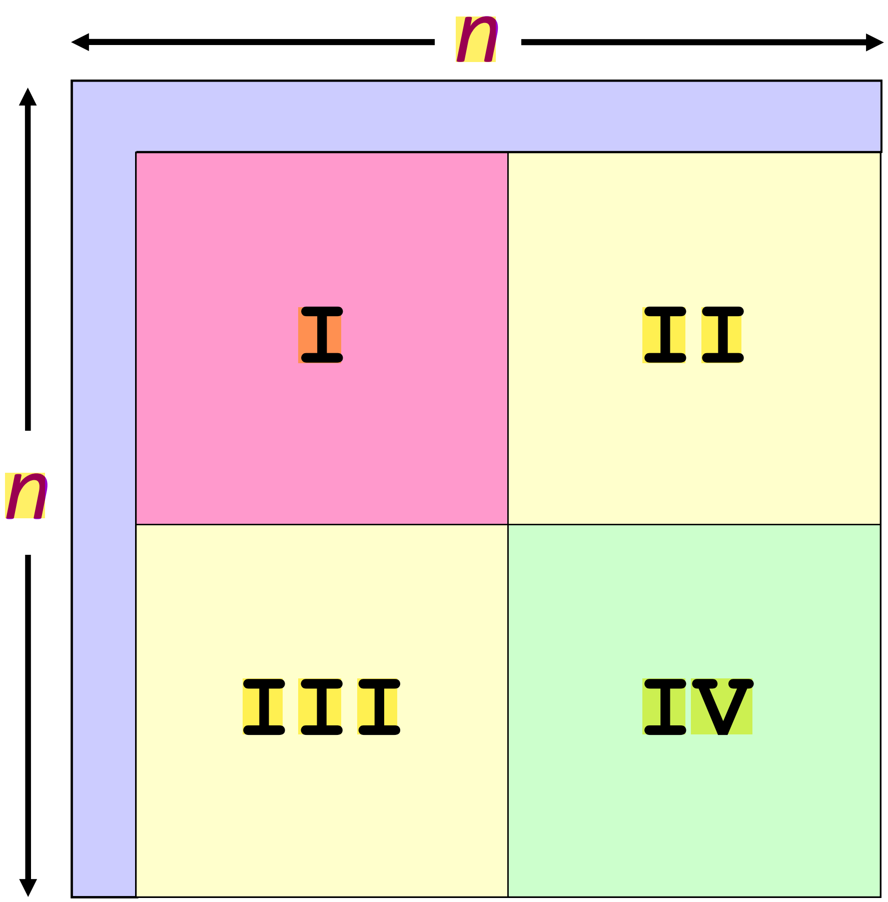
00	01	02	03	04	05	06	07
10	11	12	13	14	15	16	17
20	21	22	23	24	25	26	27
30	31	32	33	34	35	36	37
40	41	42	43	44	45	46	47
50	51	52	53	54	55	56	57
60	61	62	63	64	65	66	67
70	71	72	73	74	75	76	77

Dynamic programming

- Longest common subsequence
- Edit distance
- Time warping

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

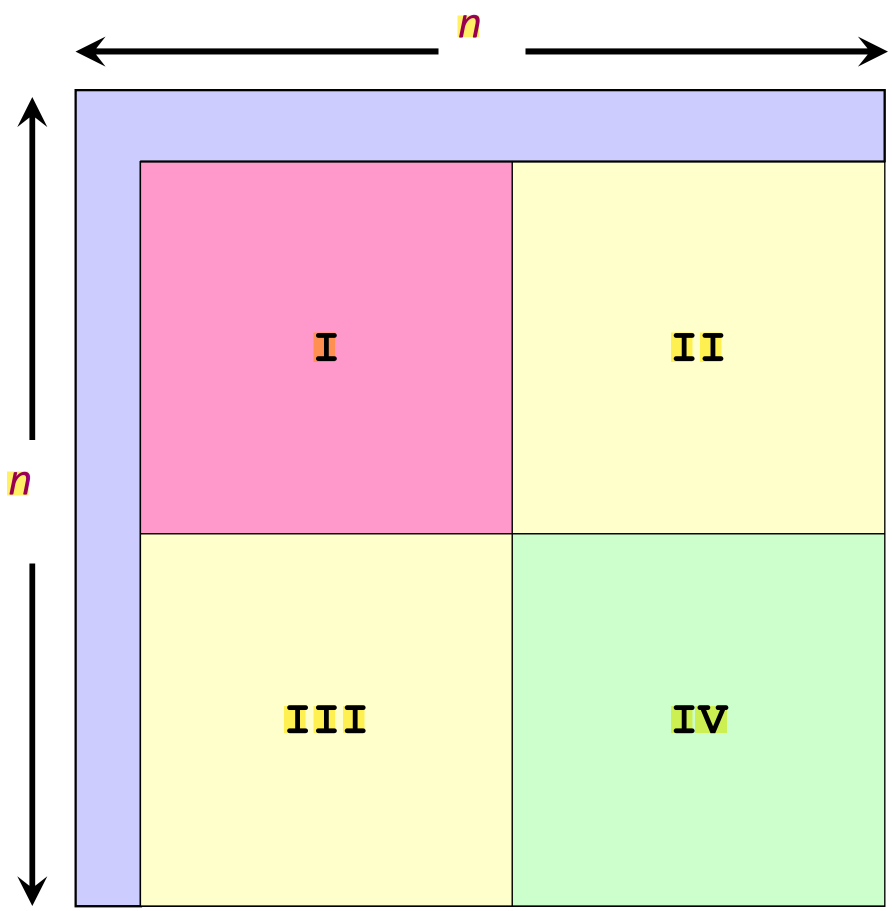
Recursive Construction



Cilk code

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

Recursive Construction



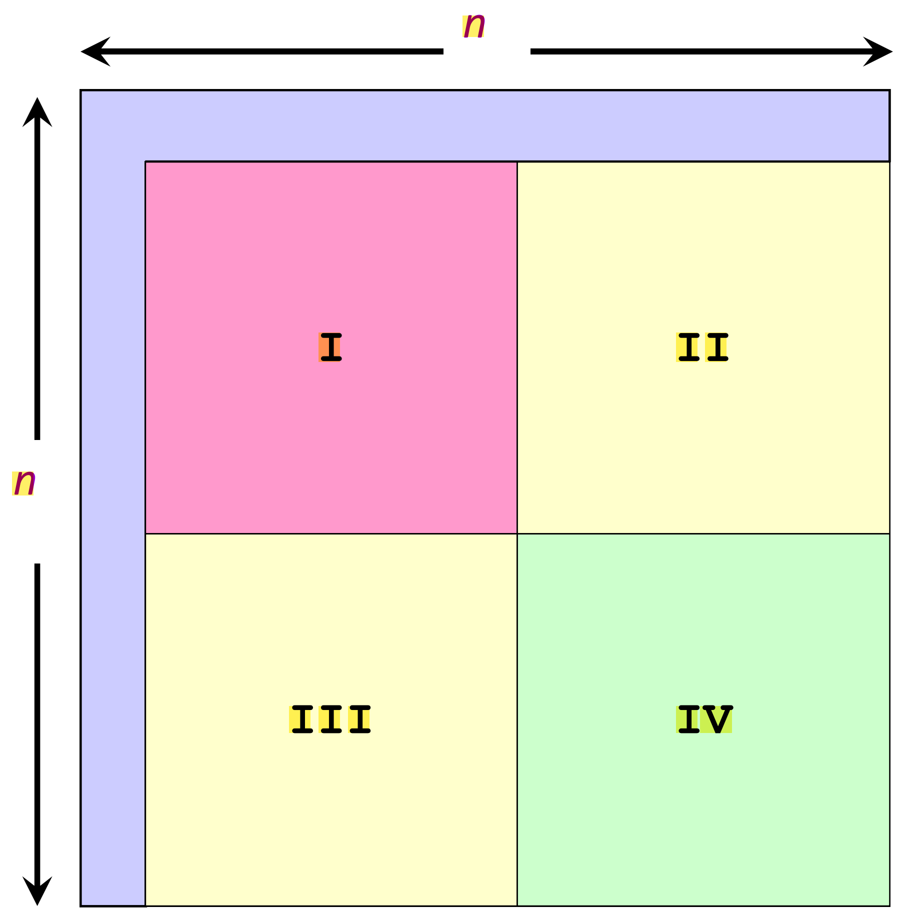
Cilk code

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

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

$= \Theta(n^2)$

Recursive Construction



Cilk code

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

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

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

Analysis of Tableau Construction

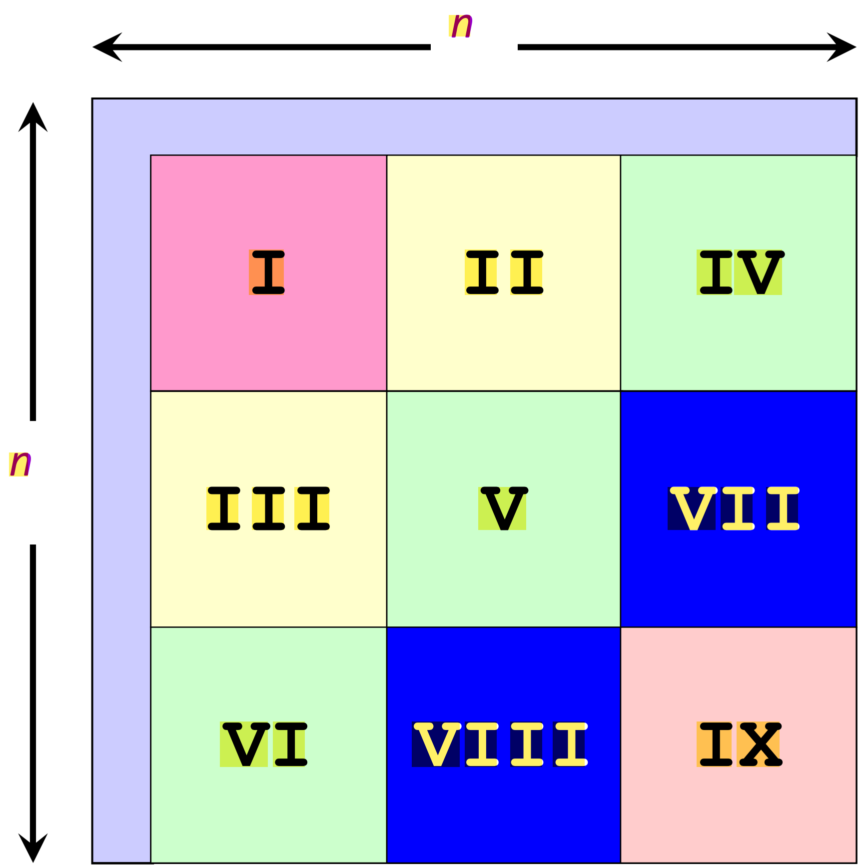
Work: $T_1(n) = \Theta(n^2)$

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

$= \Theta(n^{1.58})$

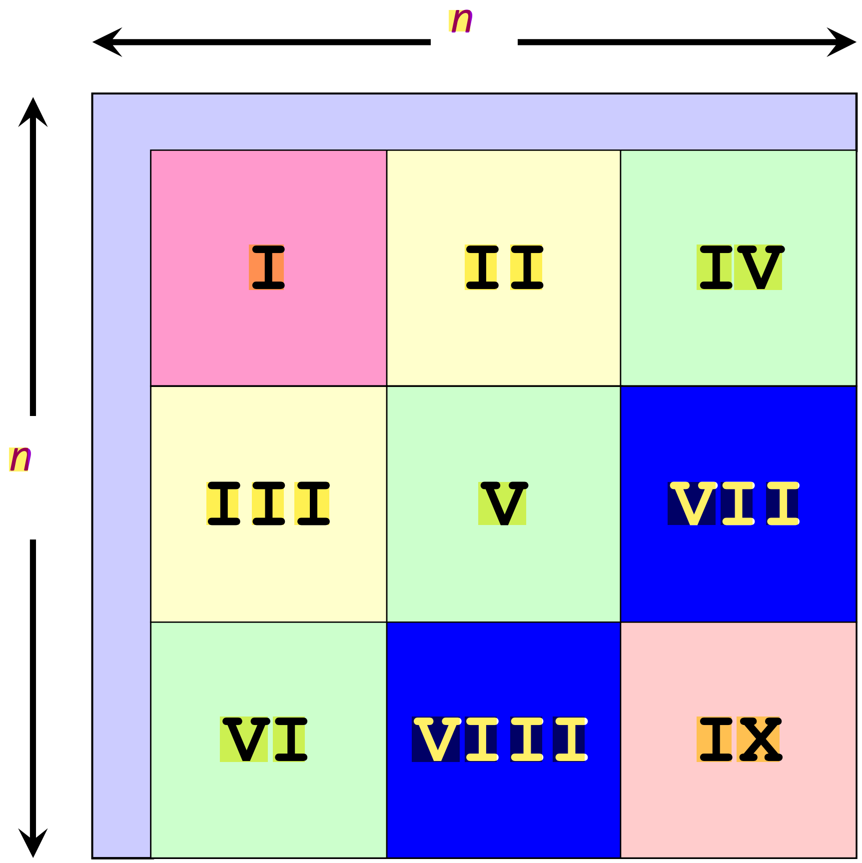
Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(n^{0.42})$

A More-Parallel Construction




```
spawn I;
sync;
spawn II;
spawn III;
sync;
spawn IV;
spawn V;
spawn VI
sync;
spawn VII;
spawn VIII;
sync;
spawn IX;
sync;
```

A More-Parallel Construction

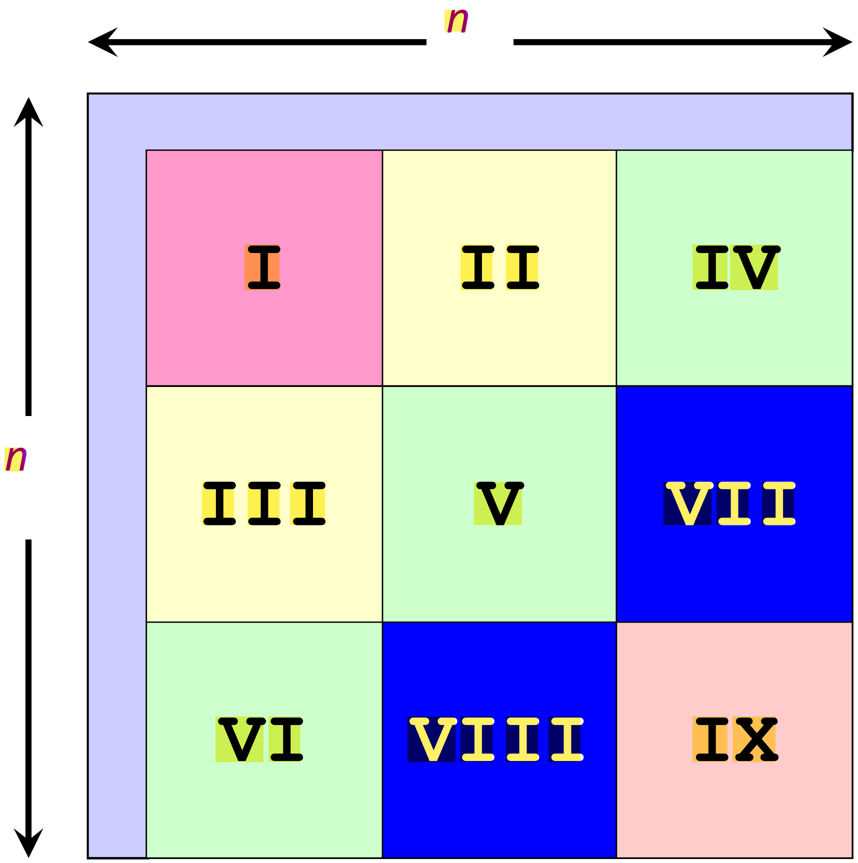



```
spawn I;
sync;
spawn II;
spawn III;
sync;
spawn IV;
spawn V;
spawn VI
sync;
spawn VII;
spawn VIII;
sync;
spawn IX;
sync;
```

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

$= \Theta(n^2)$

A More-Parallel Construction




```
spawn I;
sync;
spawn II;
spawn III;
sync;
spawn IV;
spawn V;
spawn VI
sync;
spawn VII;
spawn VIII;
sync;
spawn IX;
sync;
```

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

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

Analysis of Revised Construction

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

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

$= \Theta(n^{1.46})$

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(n^{0.54})$

More parallel by a factor of

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