Software Performance Engineering

LECTURE 9

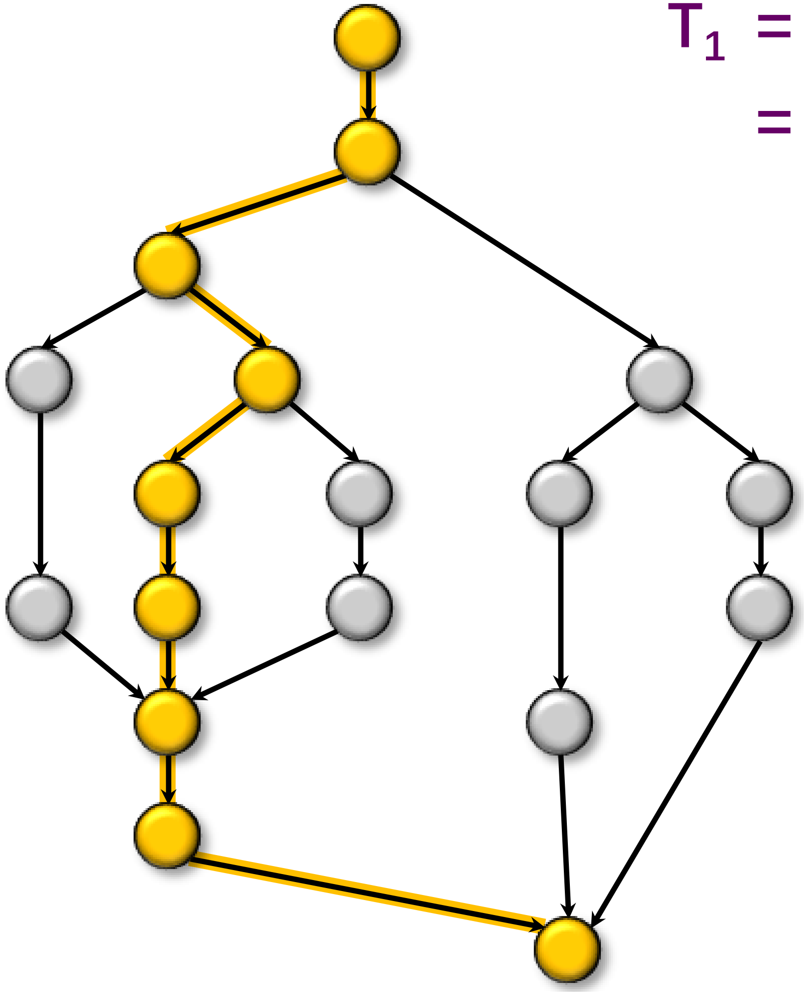
Scheduling Theory and Parallel Loops

Xuhao Chen

October 14, 2025

Performance Measures

T_P = execution time on P processors



T_1 = work
= 18

T_∞ = span
= 9

WORK LAW
$T_P \geqslant T_1/P$

SPAN LAW
$T_P \geqslant T_\infty$

Parallelism

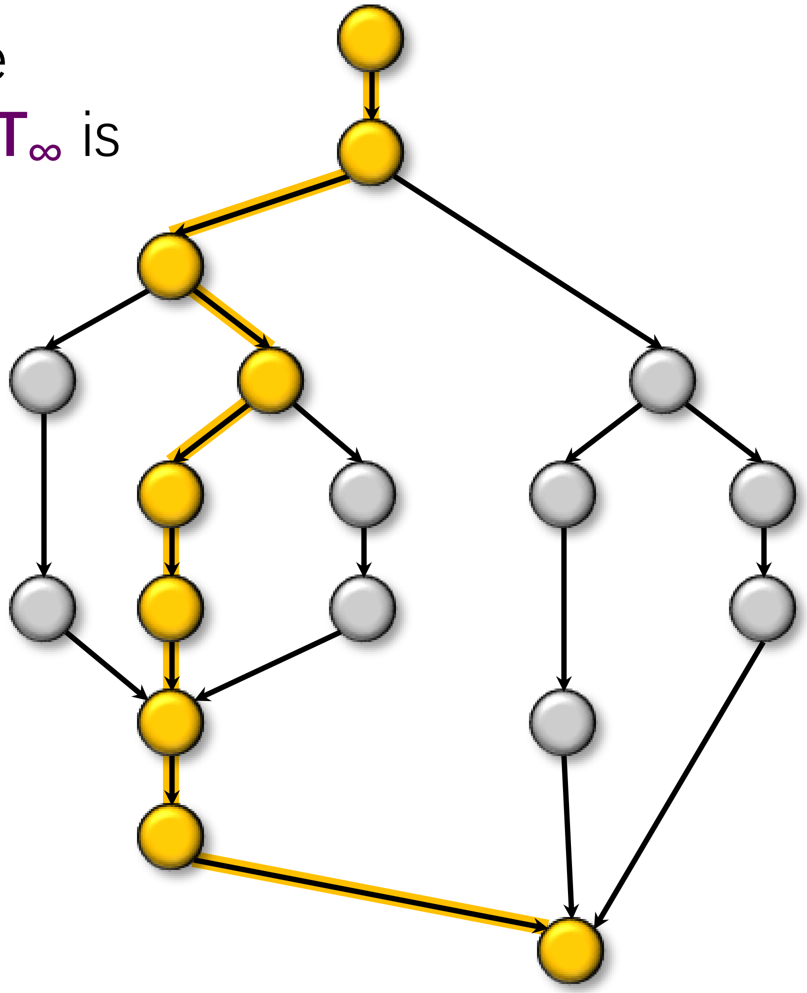
As the **Span Law** dictates that $T_P \geq T_\infty$, the maximum possible speedup given T_1 and T_∞ is

T_1/T_∞ = parallelism

= the average amount of work per step along the span

= 18/9

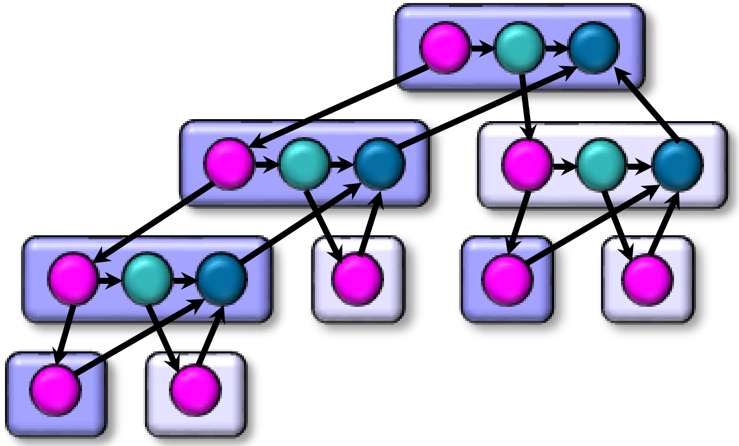
= 2 .

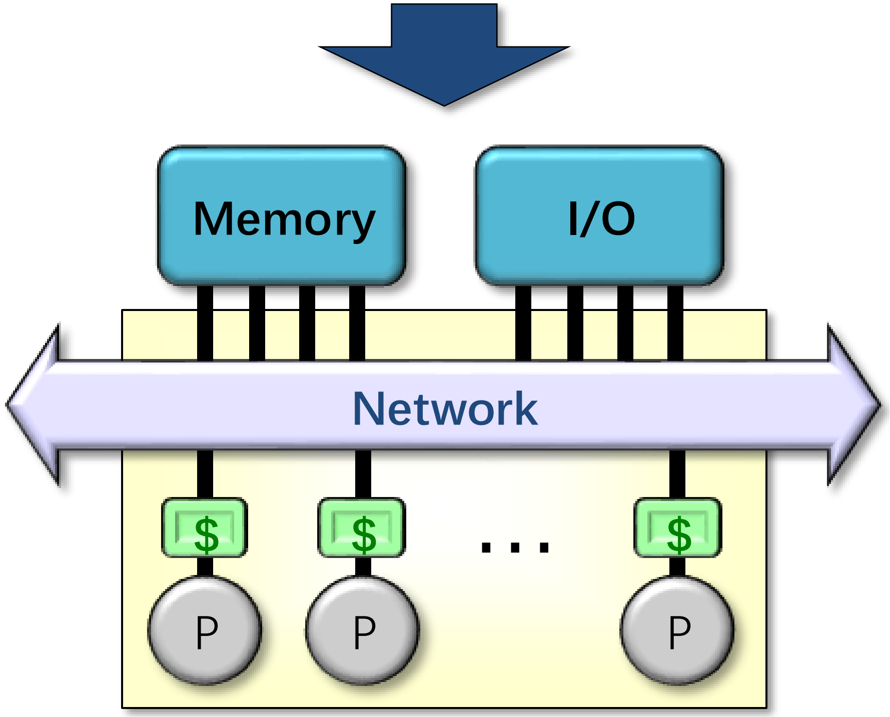


Scheduling Theory

Scheduling

- Cilk allows the programmer to express potential parallelism in an application
- The Cilk scheduler maps strands onto processors dynamically at runtime
- Since the theory of distributed schedulers is complicated, we'll explore the ideas with a simple, centralized scheduler

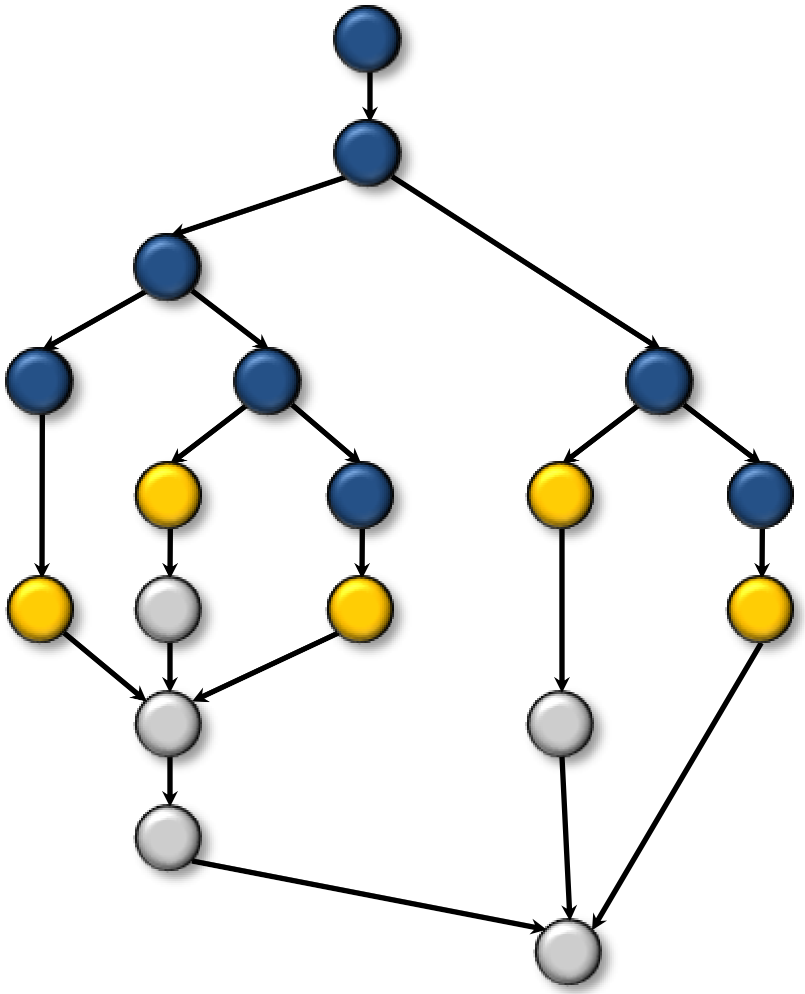




Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition. A strand is ready if all its predecessors have executed.



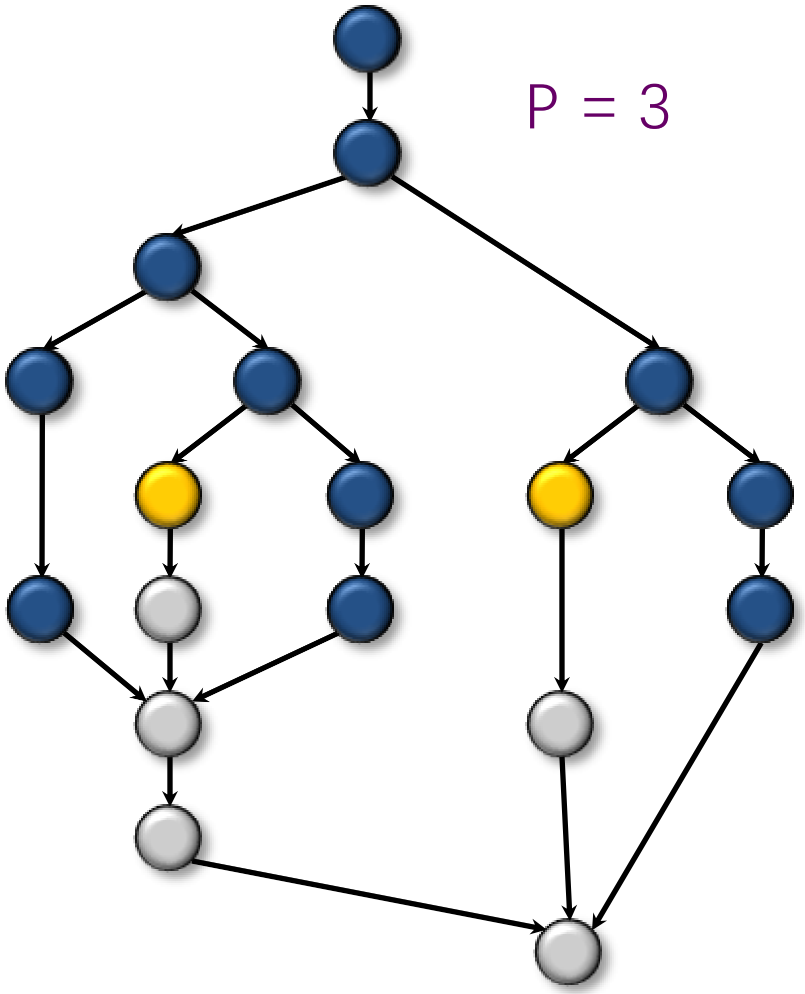
Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition. A strand is ready if all its predecessors have executed.

Complete step

- $\geqslant$ P strands ready.
- Run any P.



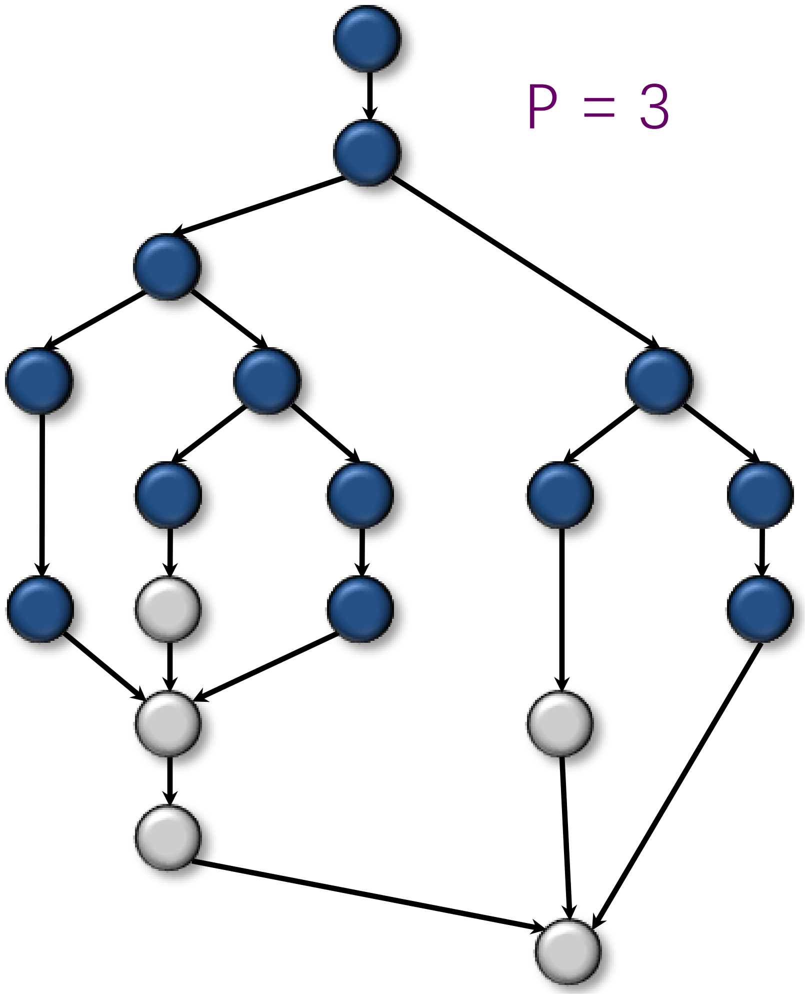
Greedy Scheduling

IDEA: Do as much as possible on every step.

Definition. A strand is ready if all its predecessors have executed.

Complete step
- $\geqslant$ P strands ready.
- Run any P.

Incomplete step
- $<$ P strands ready.
- Run all of them.



Analysis of Greedy

Greedy Scheduling Theorem [G68, B75, EZL89].
Any greedy scheduler achieves

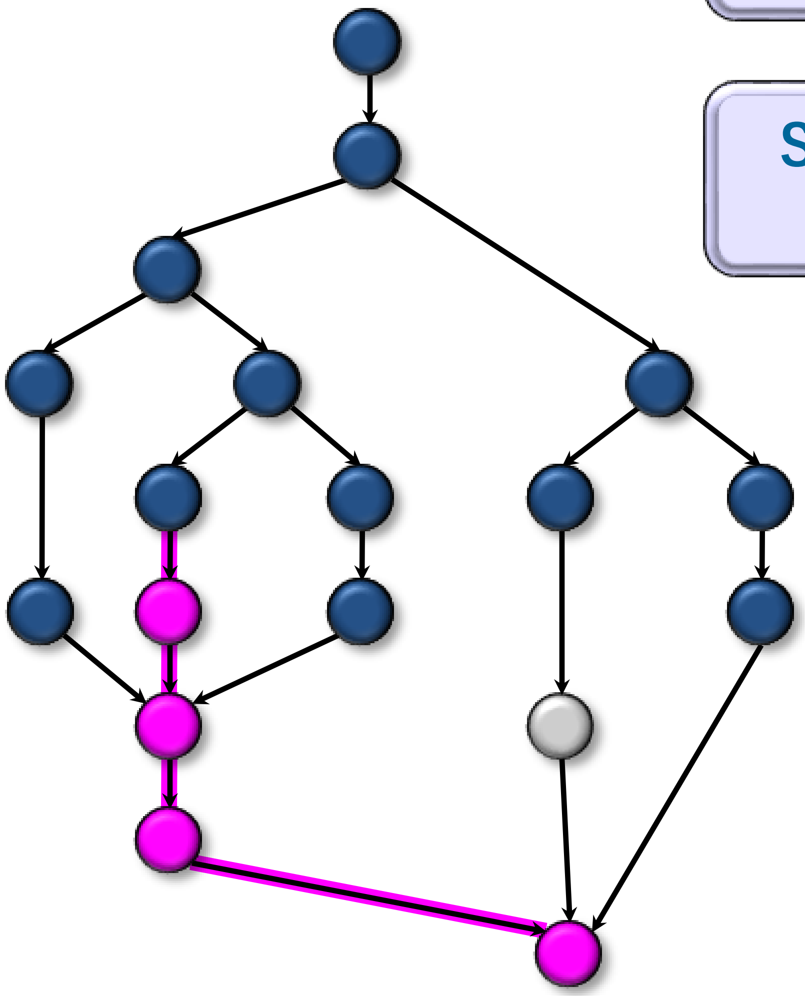
$$T_P \leqslant T_1/P + T_\infty.$$

WORK LAW
$T_P \geqslant T_1/P$

SPAN LAW
$T_P \geqslant T_\infty$

Proof.

- # complete steps $\leqslant T_1/P$ since each complete step performs P work.
- # incomplete steps $\leqslant T_\infty$ 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 $T_P* \geqslant \max\{T_1/P, T_\infty\}$ by the WORK and SPAN LAWS, we have

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

Linear Speedup

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

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

$$\begin{aligned} T_P &\leqslant T_1/P + T_\infty \\ &\approx T_1/P . \end{aligned}$$

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

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

Cilk Performance

- Cilk's randomized 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 as long as $P \ll T_1/T_\infty$.
- Instrumentation in Cilkscale allows you to measure T_1 and T_∞ .

Parallel Loops

Loop Parallelism in Cilk

Example:
In-place matrix transpose

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

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

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

Loop Parallelism in Cilk

Example:
In-place matrix transpose

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

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;
  }
}
```

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;
  }
}
```

Original code

Compiler-generated recursion

```
void p_loop(int lo, int hi)  //half open
{
  if (hi > lo + 1) {
    int mid = lo + (hi - lo)/2;
    cilk_scope {
      cilk_spawn p_loop(lo, mid);
      p_loop(mid, hi);
    }
    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;
  }
}
⋮
p_loop(1, n);
```

Divide-and-conquer

The OpenCilk compiler implements `cilk_for` loops using divide and conquer at optimization levels `-O1` and 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;
  }
}
```

Original code

Compiler-generated recursion

```
void p_loop(int lo, int hi)  //half open
{
  if (hi > lo + 1) {
    int mid = lo + (hi - lo)/2;
    cilk_scope {
      cilk_spawn p_loop(lo, mid);
      p_loop(mid, hi);
    }
    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;
  }
}
⋮
p_loop(1, n);
```

Divide-and-conquer

The OpenCilk compiler implements `cilk_for` loops using divide and conquer at optimization levels `-O1` and 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 p_loop(int lo, int hi)  //half open
{
  if (hi > lo + 1) {
    int mid = lo + (hi - lo)/2;
    cilk_scope{
      cilk_spawn p_loop(lo, mid);
      p_loop(mid, hi);
    }
    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;
  }
}
⋮
p_loop(1, n);
```

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;
  }
}
```

```
void p_loop(int lo, int hi)  //half open
{
  if (hi > lo + 1) {
    int mid = lo + (hi - lo)/2;
    cilk_scope {
      cilk_spawn p_loop(lo, mid);
      p_loop(mid, hi);
    }
    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;
  }
}
⋮
p_loop(1, n);
```

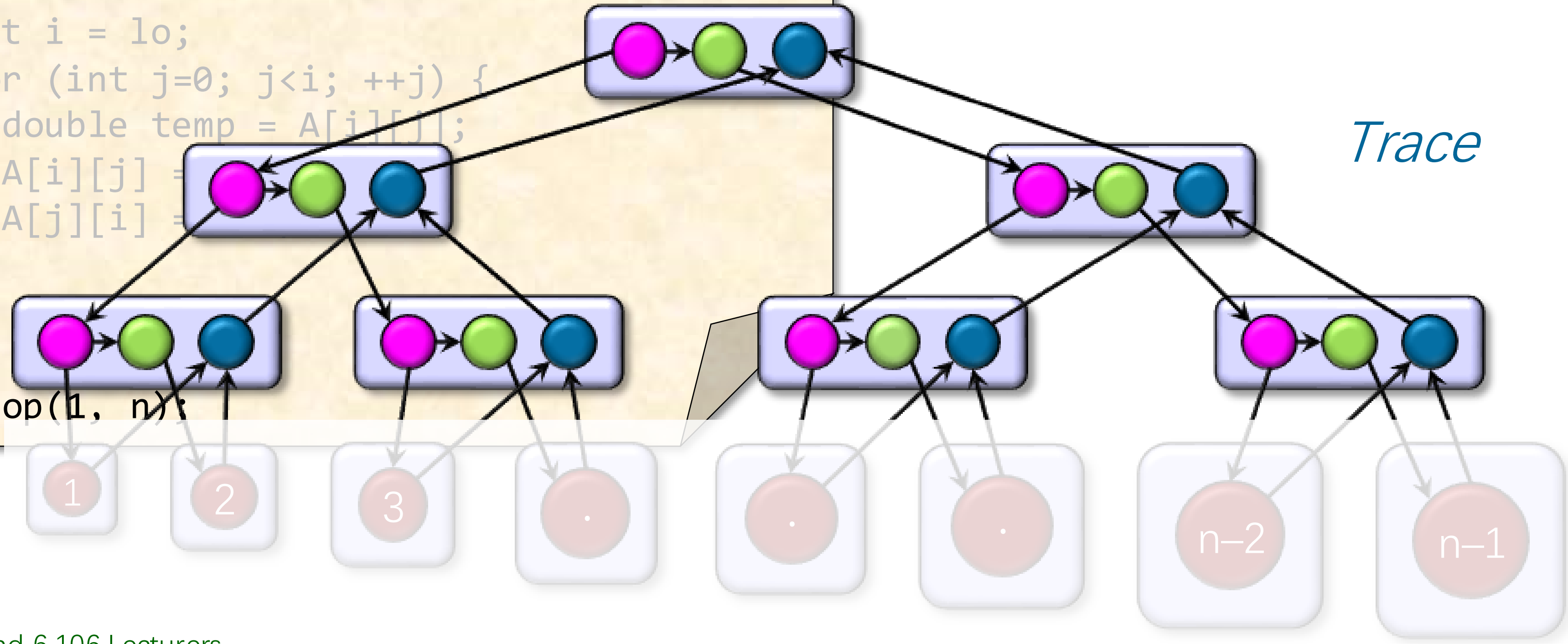
lifted
loop body

Execution of Parallel Loops

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

Divide-and-conquer implementation

`cilk_for` loop control (internal nodes)

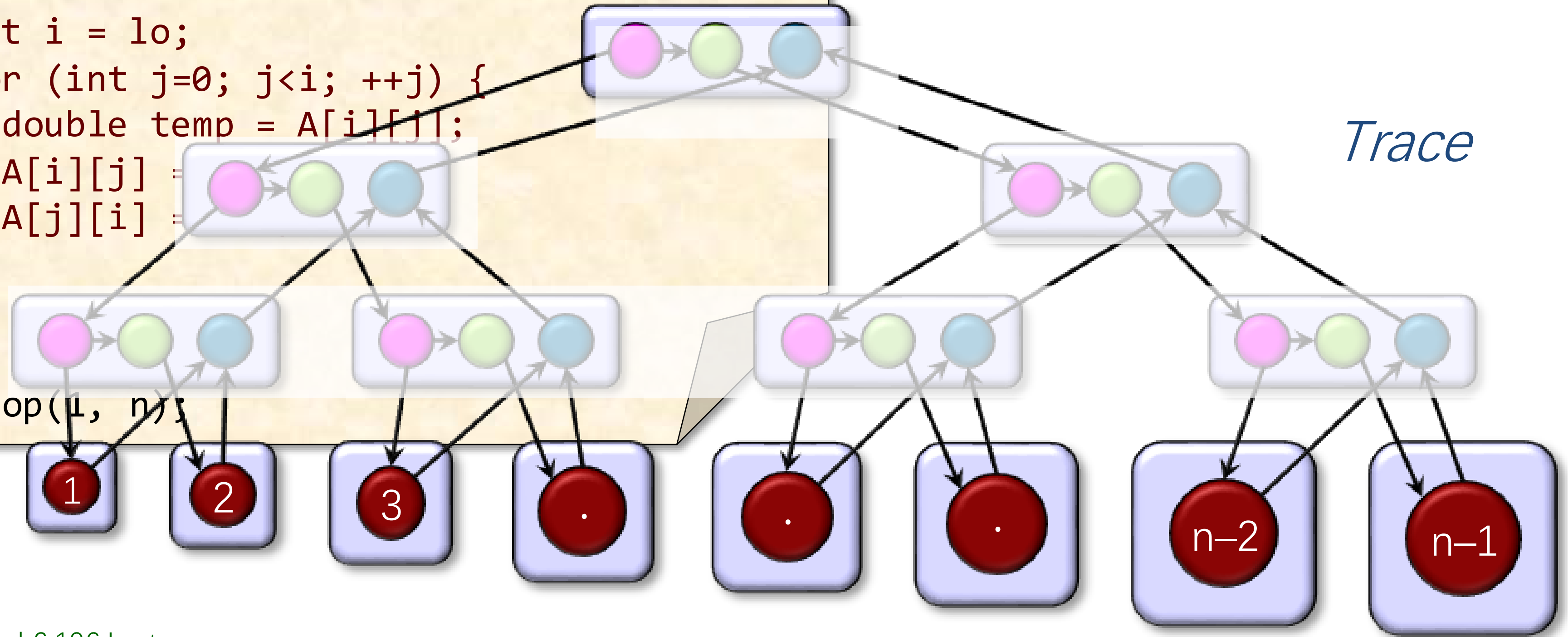


Execution of Parallel Loops

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

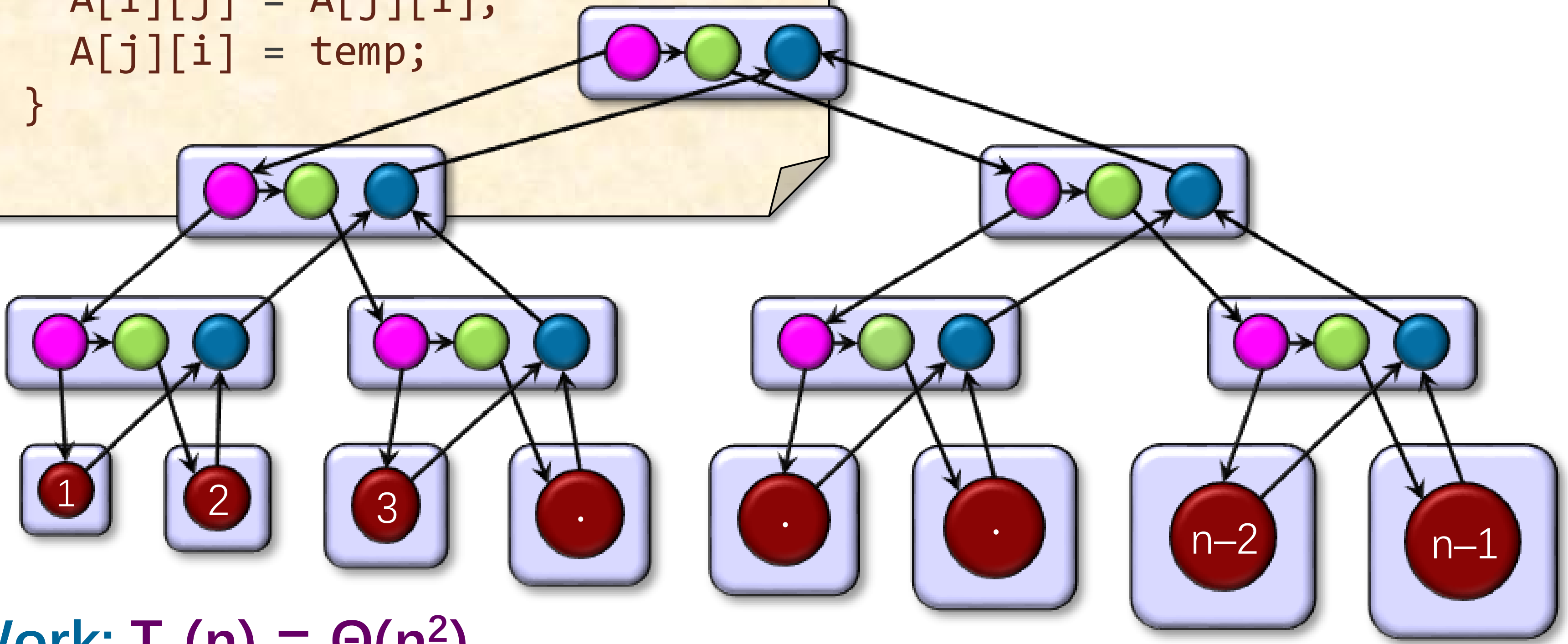
Divide-and-conquer implementation

`cilk_for` body
(leaves)



Analysis of Parallel Matrix Transpose

```
// 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;
  }
}
```



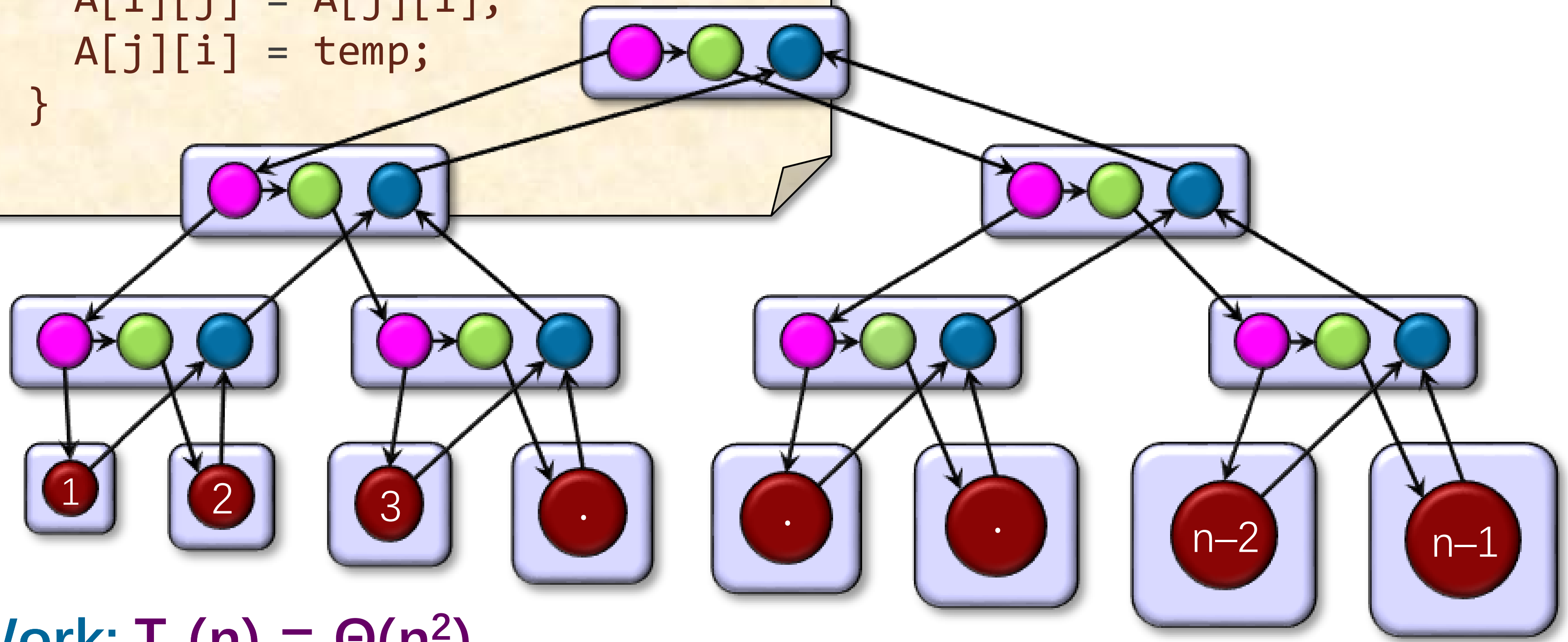
Work: $T_1(n) = \Theta(n^2)$

Analysis of Parallel Matrix Transpose

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

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

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 = $\Theta(1)$.

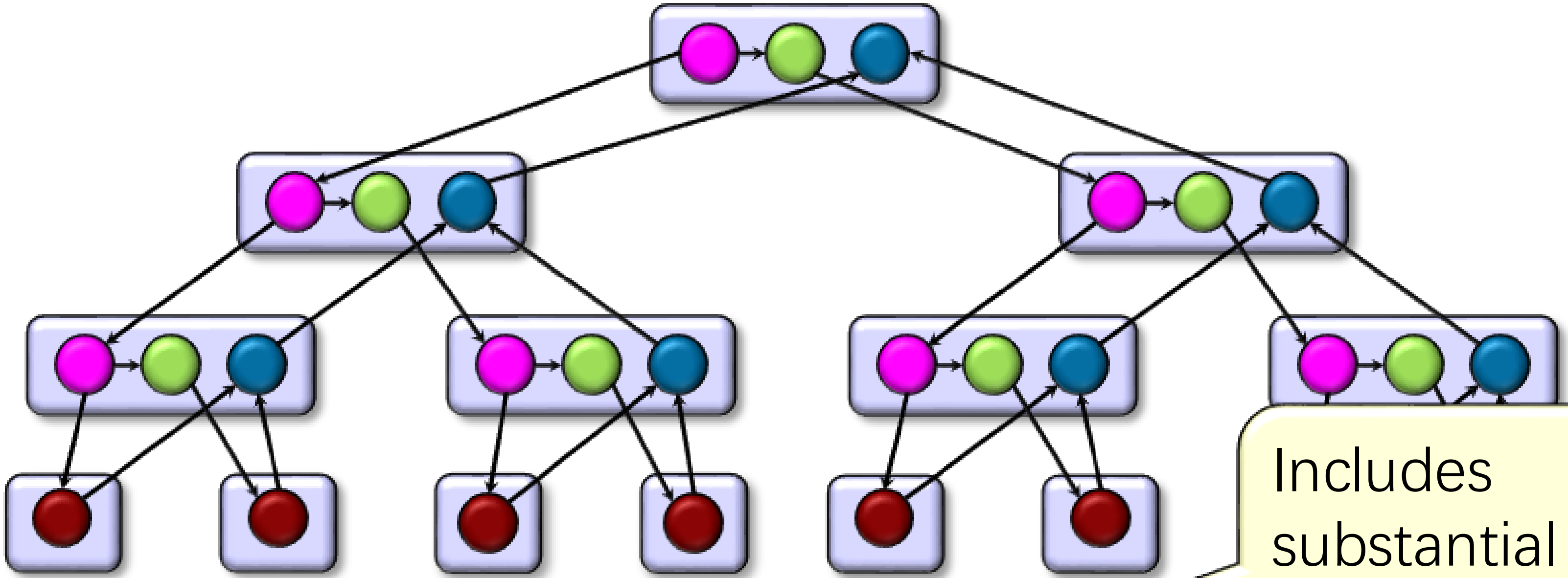
How much loop control overhead?

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];
}
```



Includes substantial overhead!

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

Optimizing Parallel-Loop Control

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

Original code

Compiler-generated recursion

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

Coarsening Parallel Loops

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

Compiler-generated recursion

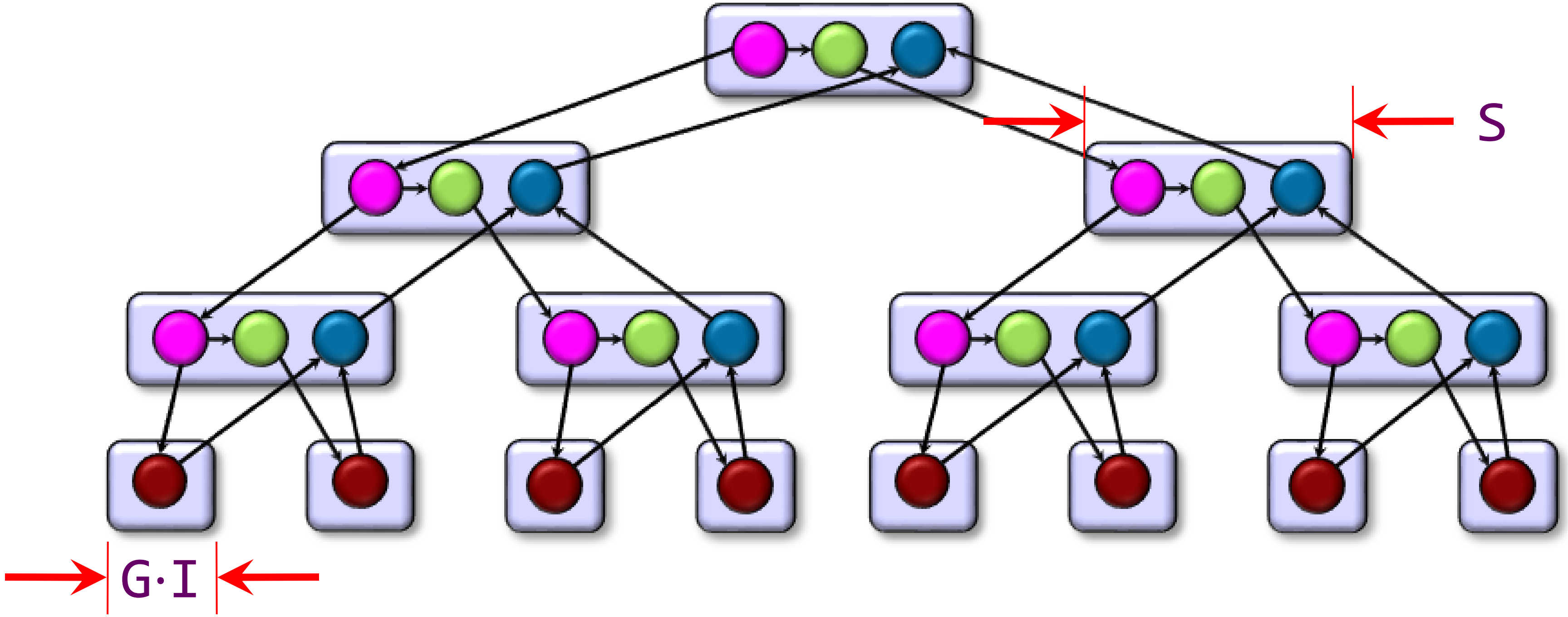
```
void p_loop(int lo, int hi) { //half open
  if (hi > lo + G) {
    int mid = lo + (hi - lo)/2;
    cilk_scope {
      cilk_spawn p_loop(lo, mid);
      p_loop(mid, hi);
    }
    return;
  }
  for (int i=lo; i<hi; ++i) {
    A[i] += B[i];
  }
}
⋮
p_loop(0, n);
```

If a grain-size pragma is not specified, the Cilk runtime system heuristically guesses **G** to minimize overhead.

Loop Grain Size

Vector addition

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



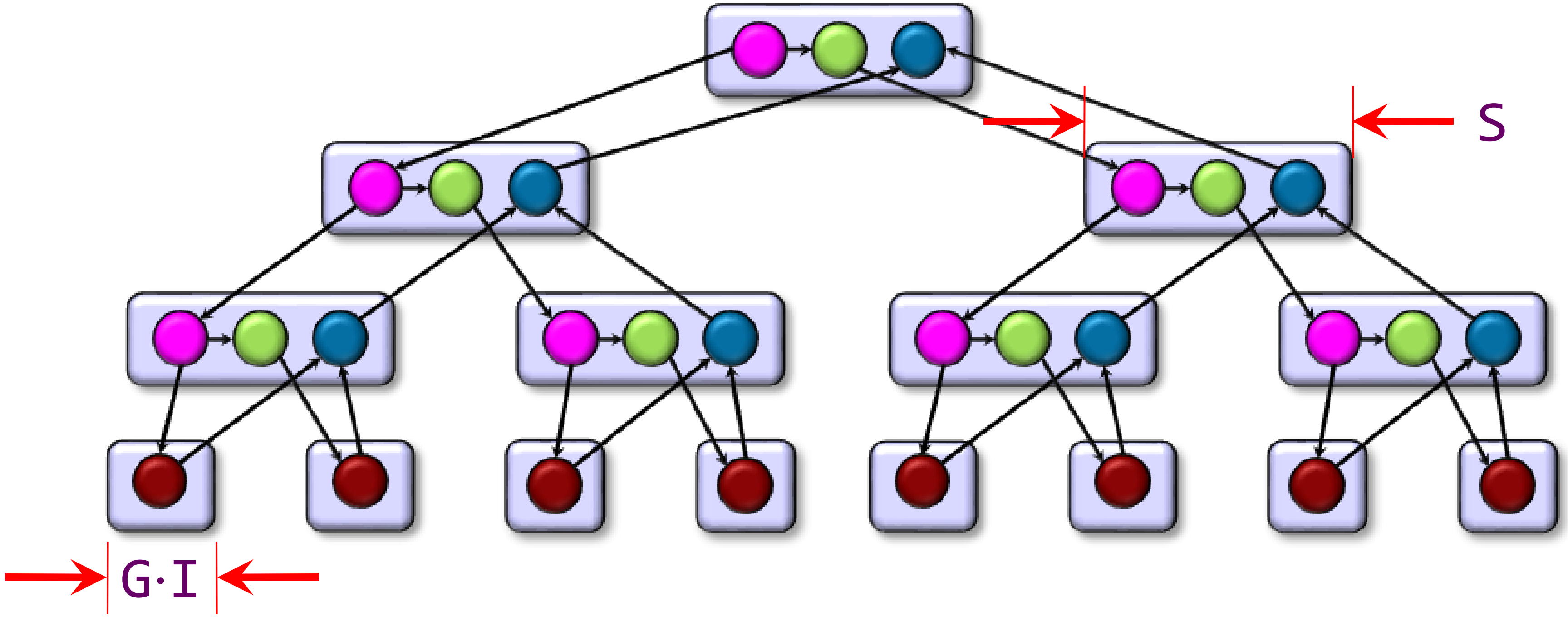
Let I be the time for one iteration of the loop body.

Let S be the time to perform a level of the recursion.

Loop Grain Size

Vector addition

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

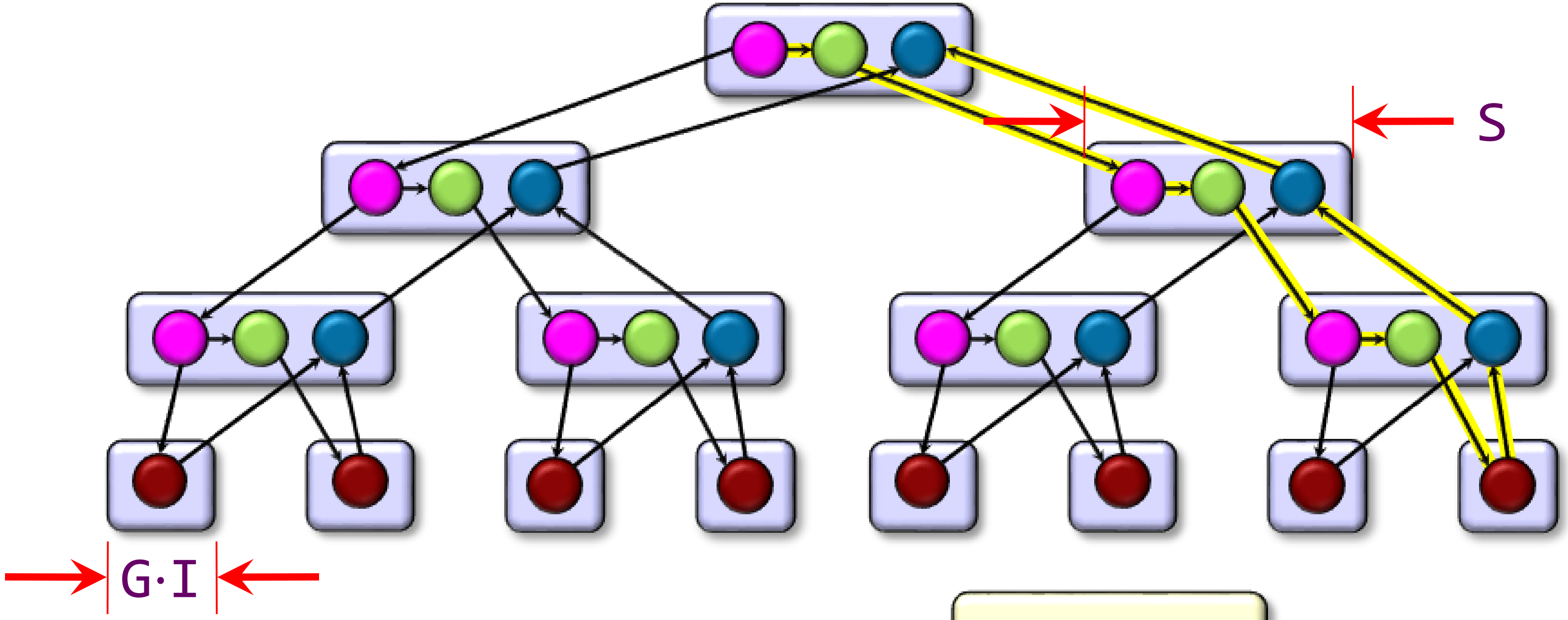


$$\textit{Work: } T_1 = \underbrace{n \cdot I}_{\textbf{real work}} + \underbrace{(n/G - 1) \cdot S}_{\textbf{work overhead}}$$

Loop Grain Size

Vector addition

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



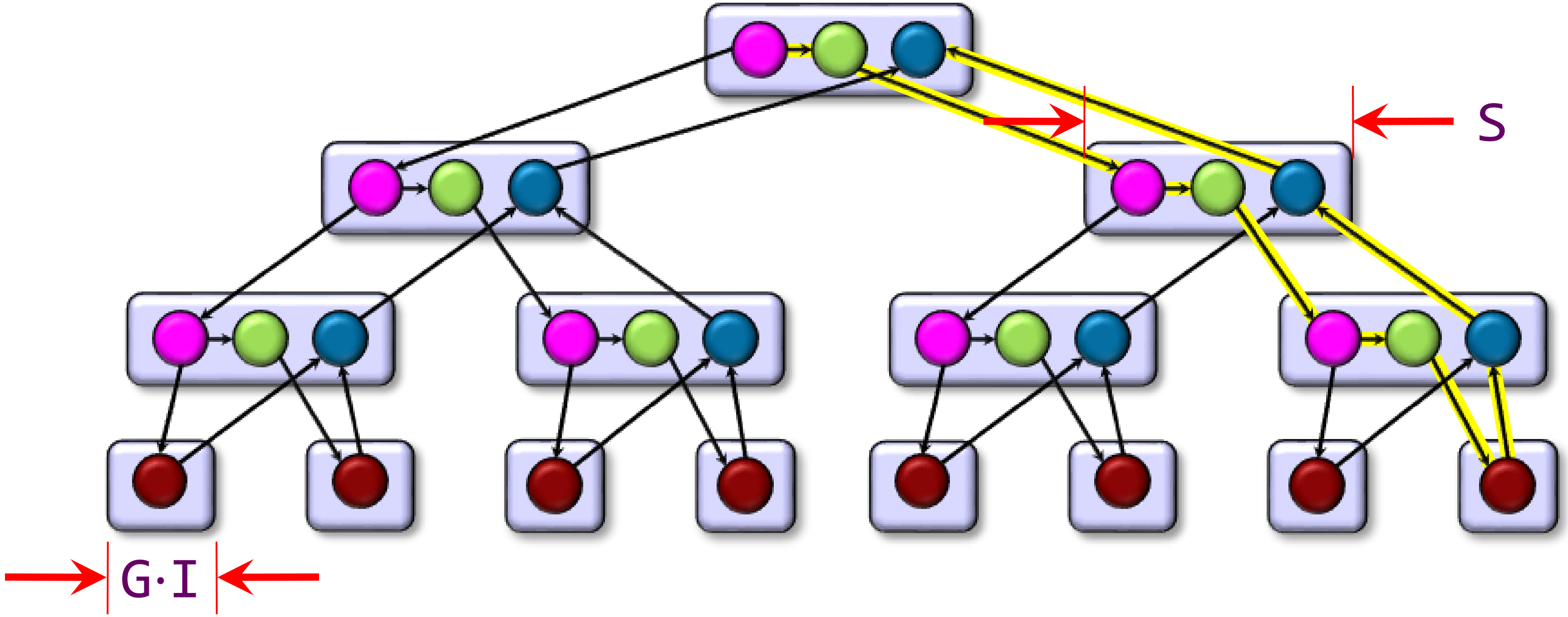
Work: $T_1 = n\cdot I + (n/G - 1)\cdot S$ — G be large

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

Loop Grain Size

Vector addition

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



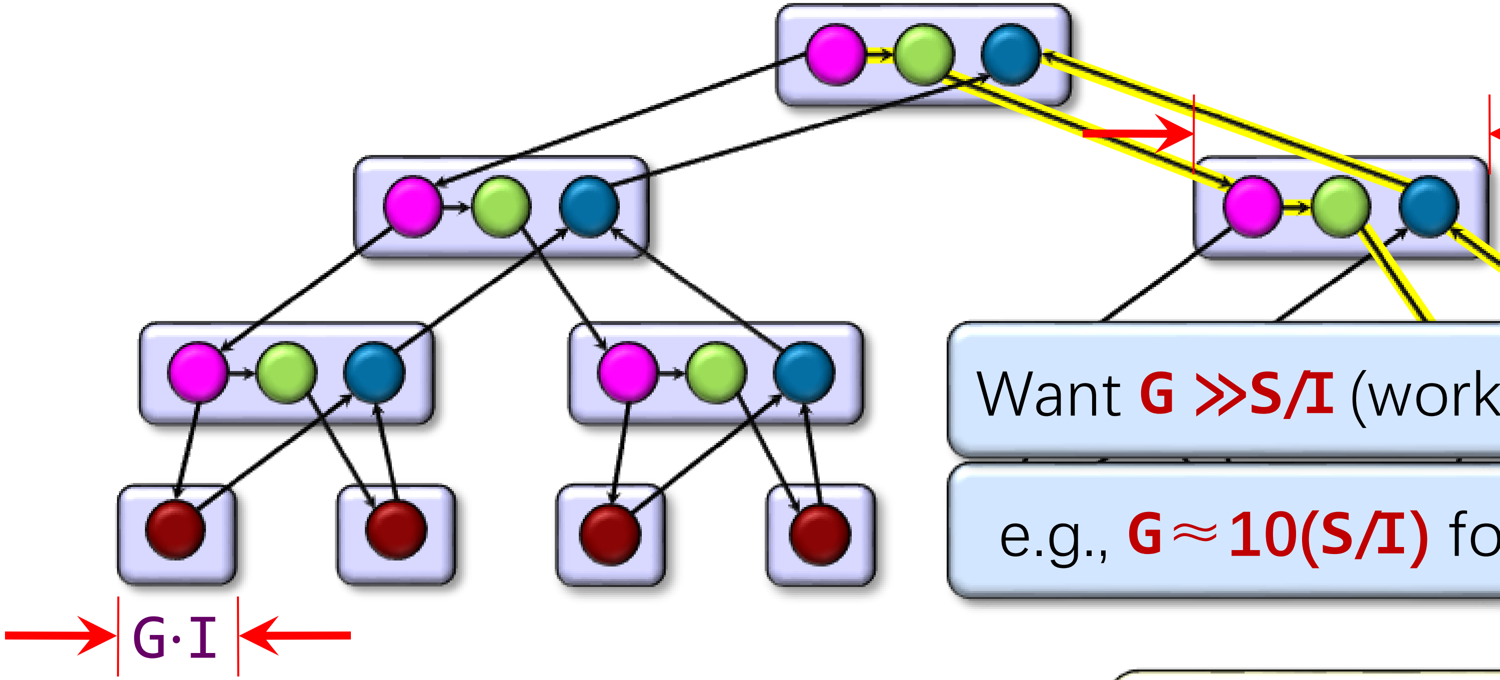
Work: $T_1 = n \cdot I + (n/G - 1) \cdot S$

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

Loop Grain Size

Vector addition

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



Want **G ≫S/I** (work) and **G** small (span)

e.g., **G ≈ 10(S/I)** for **10%** work overhead

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

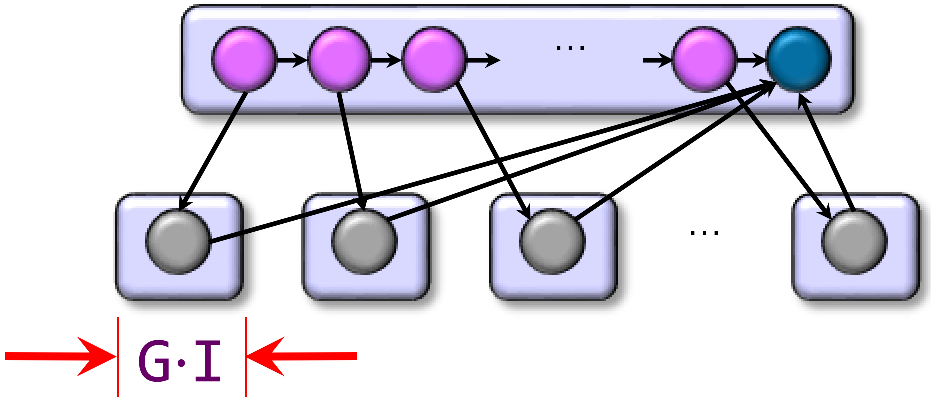
Span: $T_\infty = G \cdot I + \lg(n/G) \cdot S$

Parallelism

$T_1/T_\infty \approx \Theta(n/\lg n)/(S/I)$

Another Implementation

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



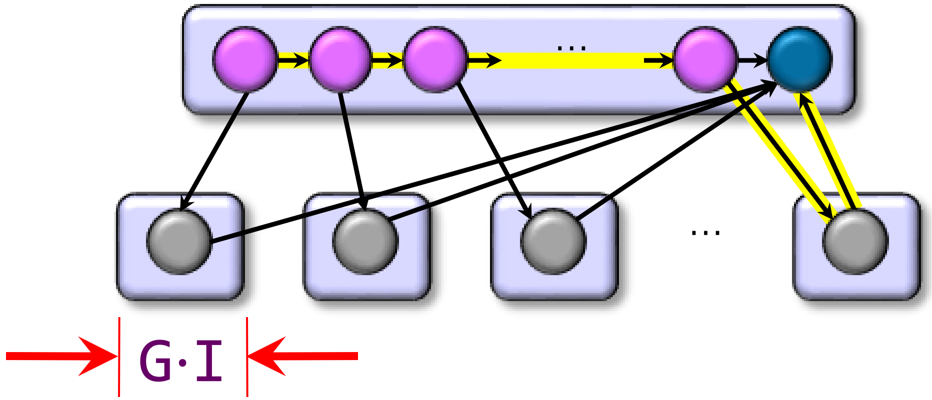
Assume that **G** = 1.

Work: $T_1 = \Theta(n)$

Span: $T_\infty =$

Another Implementation

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



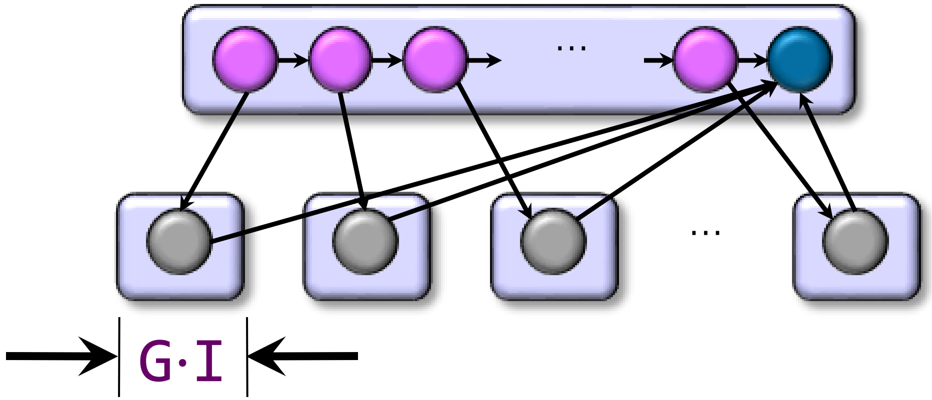
puny

Assume that $\mathbf{G} = 1$.

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){
  cilk_scope {
    for (int j=0; j<n; j+=G) {
      cilk_spawn {
        for (int i=j; i<MIN(j+G,n); i++)
          A[i] += B[i];
} } } }
```



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

Analysis 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 asymptotic parallelism of Code A compare to that 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:

Span:

Parallelism:

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:

Parallelism:

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:

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

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);
}
```

Take-Aways

- Any greedy scheduler provides linear speedup on computations having sufficient **parallel slackness**
- The OpenCilk runtime system incorporates a **randomized work-stealing** scheduler that has strong theoretical bounds on its running time which are similar to those for greedy scheduling
- Loops in Cilk are synthesized using **divide-and-conquer** spawning, which incurs linear work and logarithmic span
- **Coarsening recursion** can reduce loop overhead