CPSC 4240/5240: Parallel Programming Techniques

Lecture 16: Scheduling, Parallel Loops, and Cilk

Lecturer: Quanquan C. Liu

Spring 2026

Announcements

- Homework 3 Posted
  - Due **Monday, March 30 11:59pm**
  - Late due date: **Thursday, April 2, 11:59pm**
- Final Project
  - Due **April 24**
  - 5-10 min final project presentation **April 21/23**

Final Project: Additional Details

- Final Project
  - Due **April 24**
  - 5-10 min final project presentation **April 21, 23**
- Grade Breakdown for Project:
  - **Originality: 30% (for 524 students: half of this grade will consist of research potential)**
  - Execution: 30%
  - Written Report: 20%
  - Final Presentation: 20%

Scheduling Parallel Operations and Cilk

Slides courtesy of MIT 6.106 Instructors

Introduction to Cilk

- **Cilk** is a **multithreaded parallel extension** for C/C++ that simplifies parallel programming
- Originally developed at **MIT**; now part of several open-source and commercial implementations (e.g., **Intel Cilk Plus**, **Tapir/LLVM**)
- Provides simple **keywords** (cilk_spawn, cilk_sync, cilk_for) and a **work-stealing scheduler** to achieve efficient parallelism

What is a work-stealing scheduler?

What is a work-stealing scheduler?

- Each thread (worker) runs its own **deque** of tasks (activation frames)
- **Normal execution**: pop tasks from top of its own deque
- **If a worker is idle**: tries to **steal** a task from the **bottom** of another worker's deque
- **Efficiency**:
  - **Expected time** to find tasks is good when there is enough parallelism (the DAG is wide)
  - Stealing overhead is typically minimal compared to manual thread management

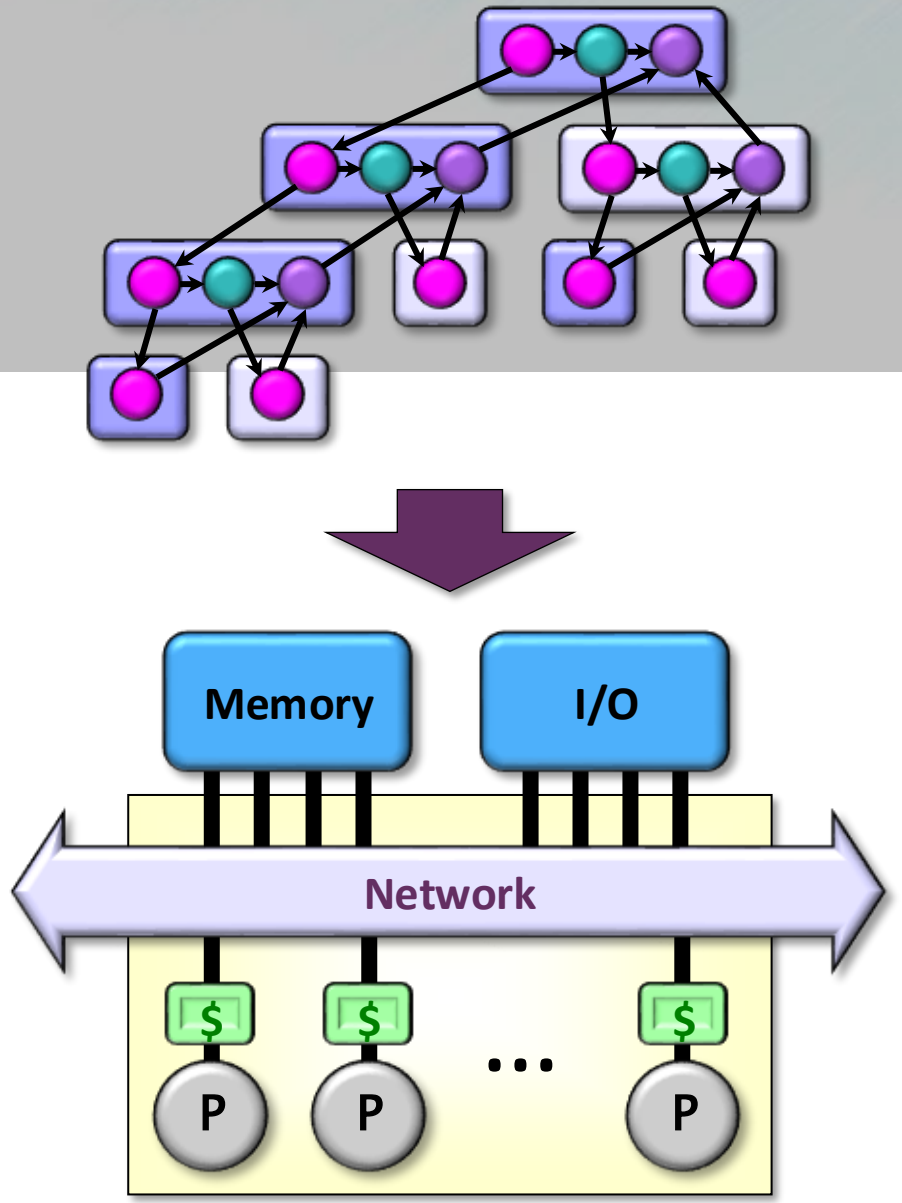
Cilk versus OpenMP

Cilk compared to OpenMP

- Includes language extensions (Cilk_for etc.) whereas OpenMP is **only compiler directives**
- Uses **work-stealing scheduler** vs. compiler specific scheduler
  - OpenMP can specify scheduling policy
- Uses **divide-and-conquer parallelism (what is this?—later this lecture!)** instead of fork-join
  - Recall fork-join in OpenMP; parallel for loop:
    - Splits the for loop into chunks and assigns chunks to threads
    - Joins the threads via a join

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

Greedy Scheduling

- IDEA: Do as much as possible on every step.

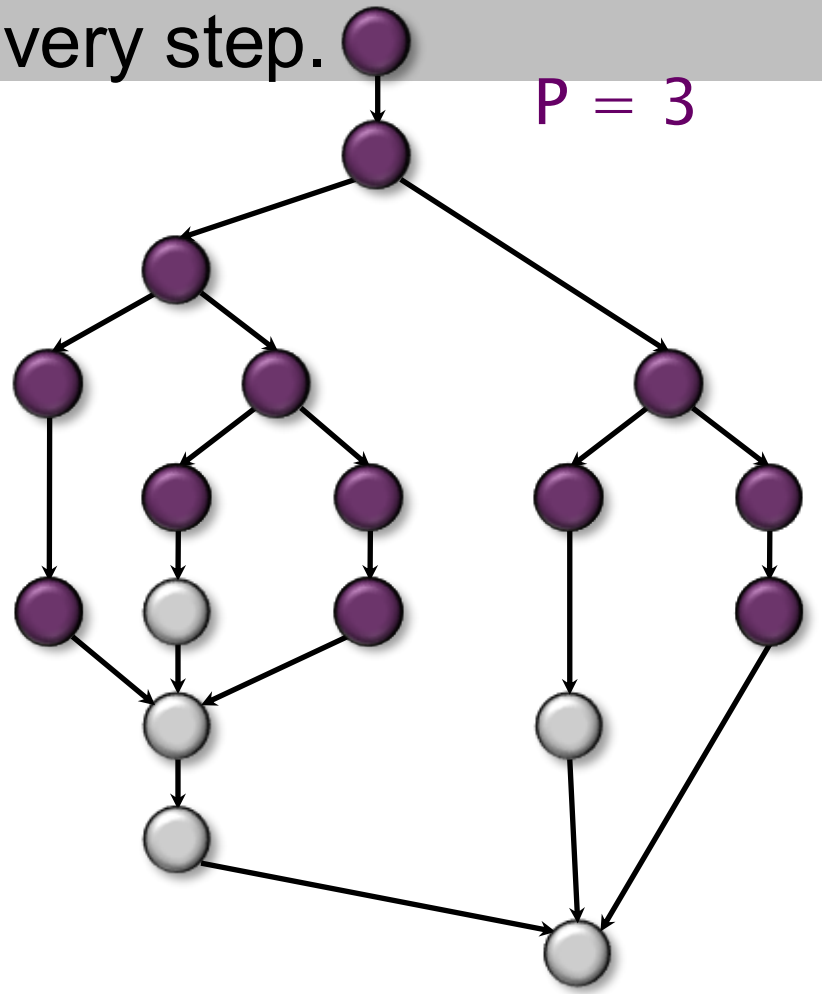
Definition. A strand is *ready* if all its predecessors have executed.

Complete step

- ≥ P strands ready.
- Run any P.

Incomplete step

- < P strands ready.
- Run all of them.



Analysis of Greedy Scheduling

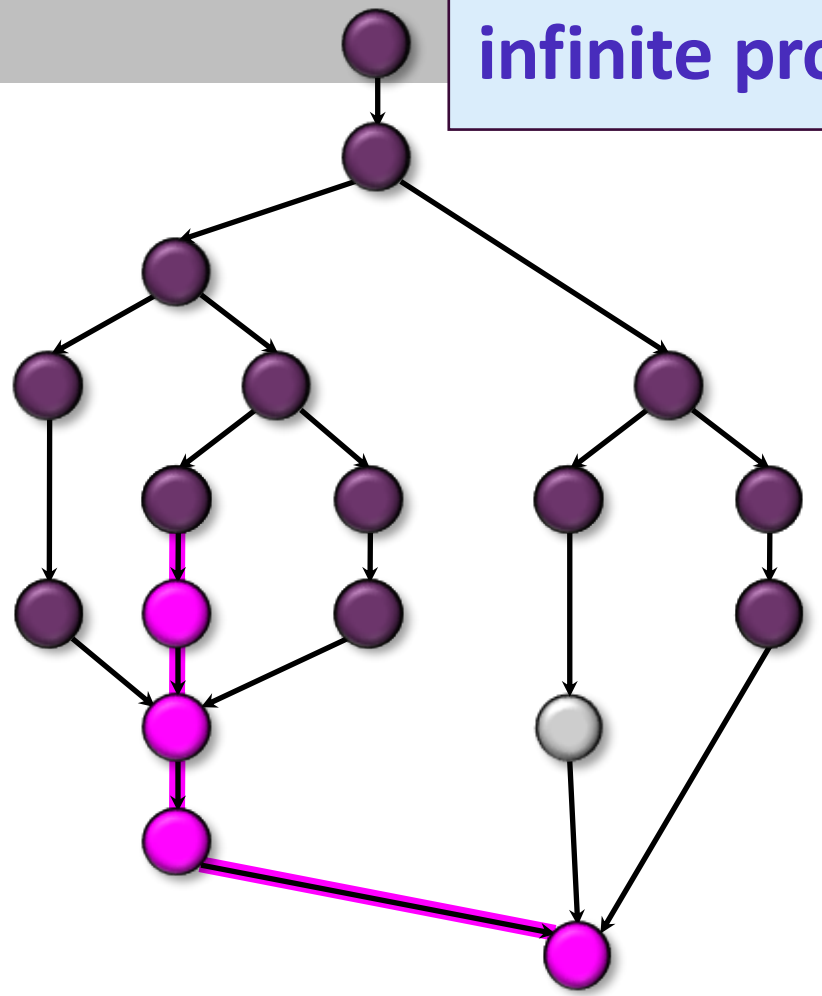
T_1 time on 1 processor
T_∞ time on infinite processors

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

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

Proof.

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

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

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

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

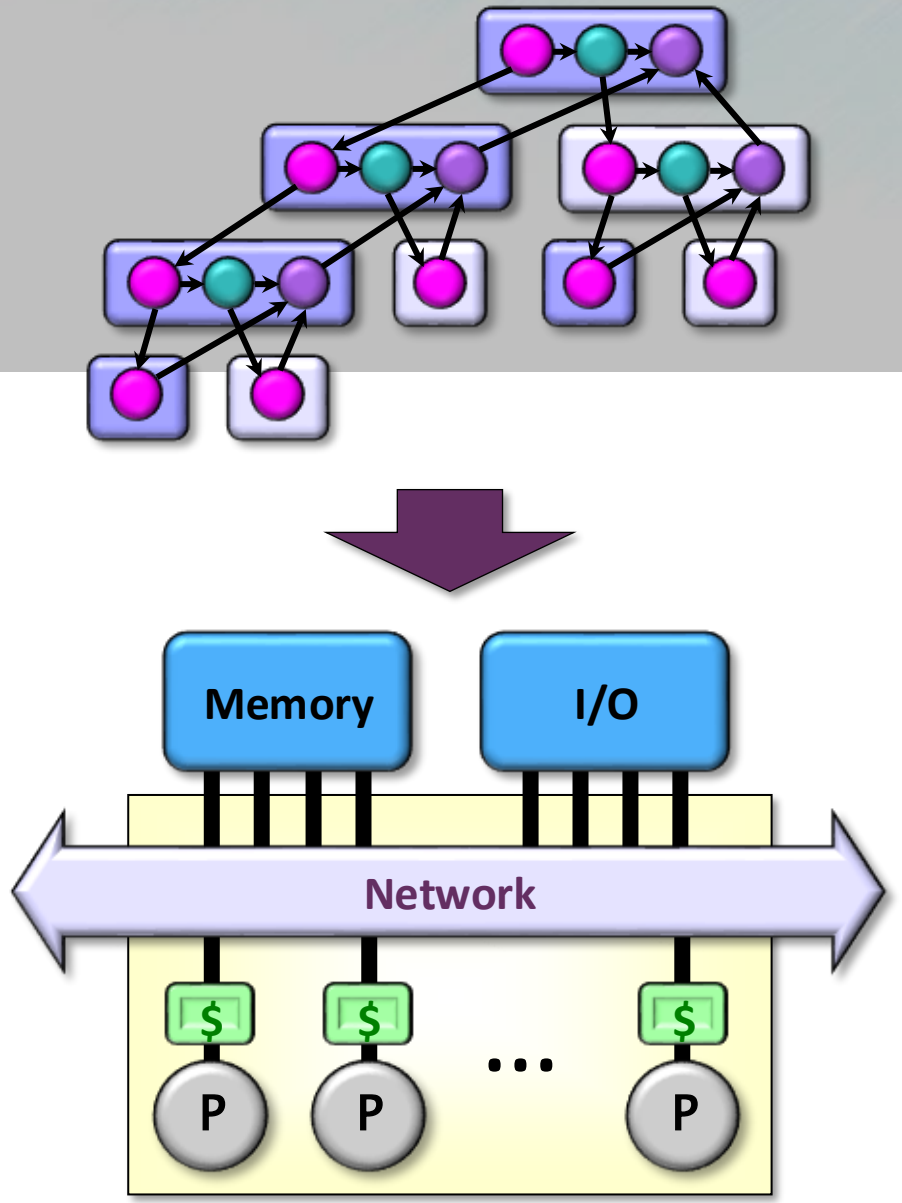
Thus, the speedup is $T_1/T_P \approx P$. ■

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

Scheduling Parallel Computation

Scheduling

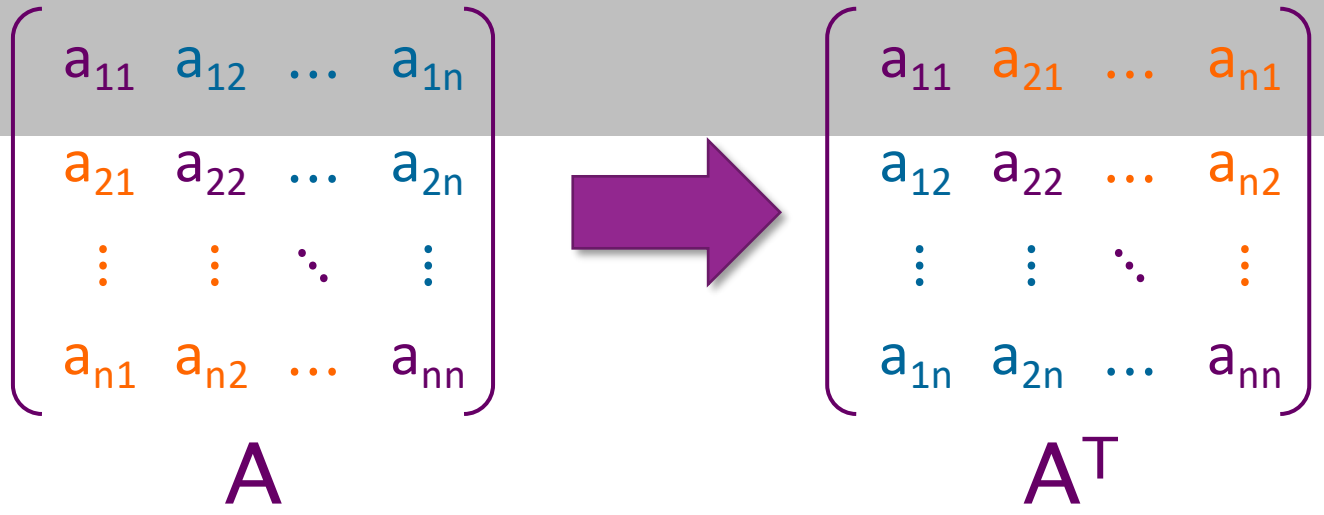
- Cilk allows the programmer to express potential parallelism in an application.
- The Cilk scheduler maps **strands** onto processors dynamically at runtime.
- Since the theory of distributed schedulers is complicated, we'll explore the ideas with a simple, centralized scheduler.



Fine-Grained Parallel Loops

Loop Parallelism in Cilk

Example: In-place matrix transpose

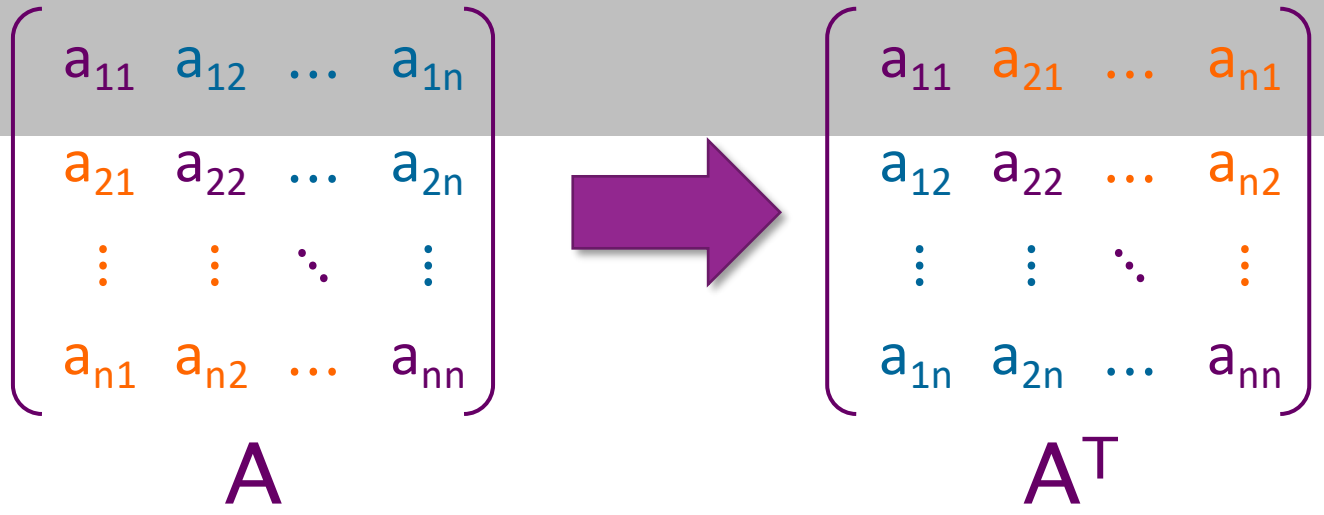


The iterations of a `cilk_for` loop execute in parallel.

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



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

Divide-and-conquer

The OpenCilk compiler implements `cilk_for` loops using divide and conquer at optimization levels `–O1` and higher.

Compiler-generated recursion

```
void p_loop(int lo, int hi)
{
  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;
  }
}
```

Original code

Divide-and-conquer

The OpenCilk compiler implements `cilk_for` loops using divide and conquer at optimization levels `–O1` and higher.

Compiler-generated recursion

```
void p_loop(int lo, int hi)
{
  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;
  }
}
```

Original code

cilk_for

loop control

```
void p_loop(int lo, int hi)
{
  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;
  }
}
```

Original code

lifted
loop body

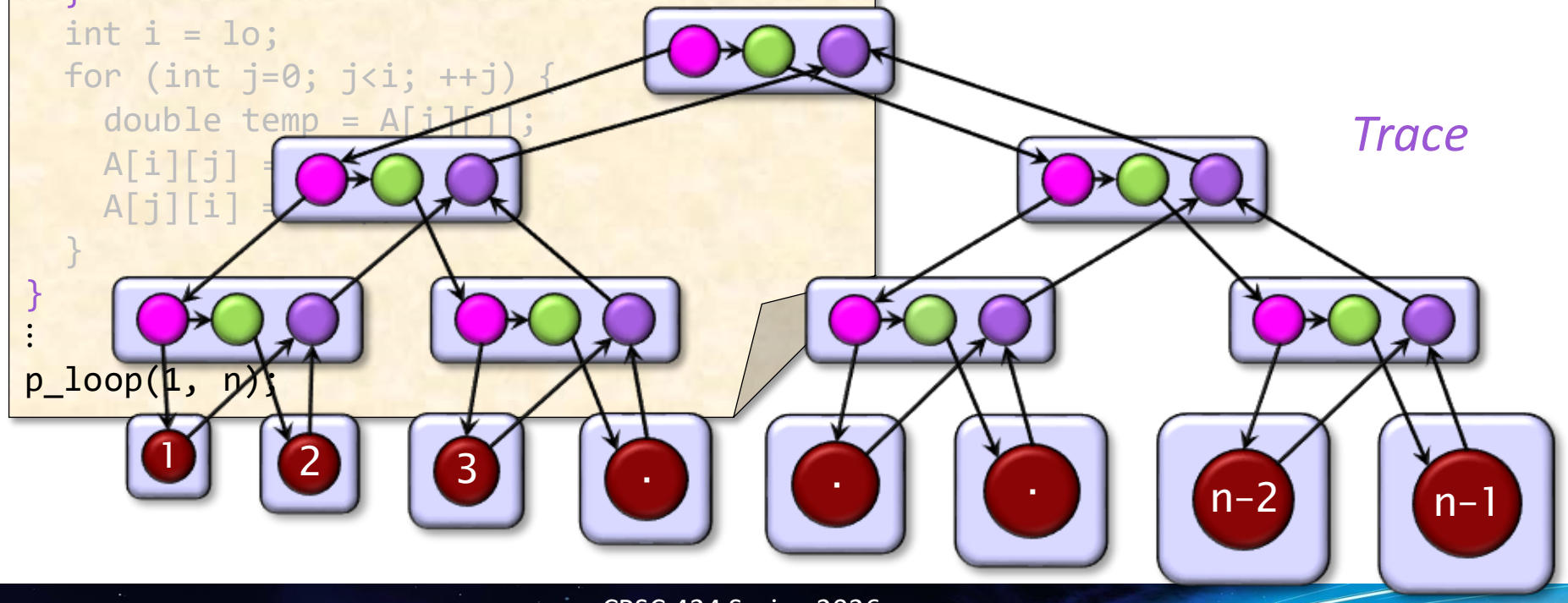
```
void p_loop(int lo, int hi)
{
  if (hi > lo + 1) {
    int mid = lo + (hi - lo)/2;
    cilk_scope {
      cilk_spawn p_loop(lo, mid);
      p_loop(mid, hi);
    }
    return;
  }
  int i = lo;
  for (int j=0; j<i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}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

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)



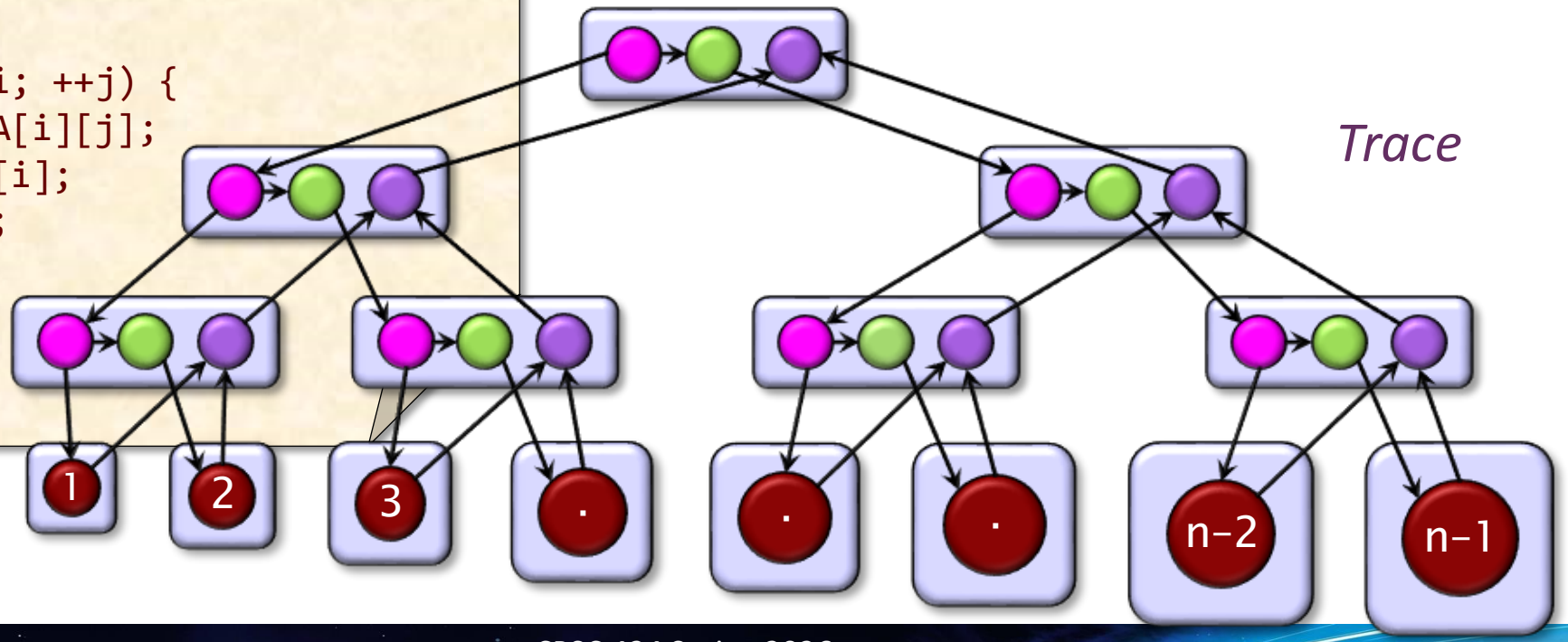
Execution of Parallel Loops

```
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];
    A[j][i] = temp;
  }
}
⋮
p_loop(1, n);
```

Divide-and-conquer implementation

`cilk_for` body (leaves)

Trace

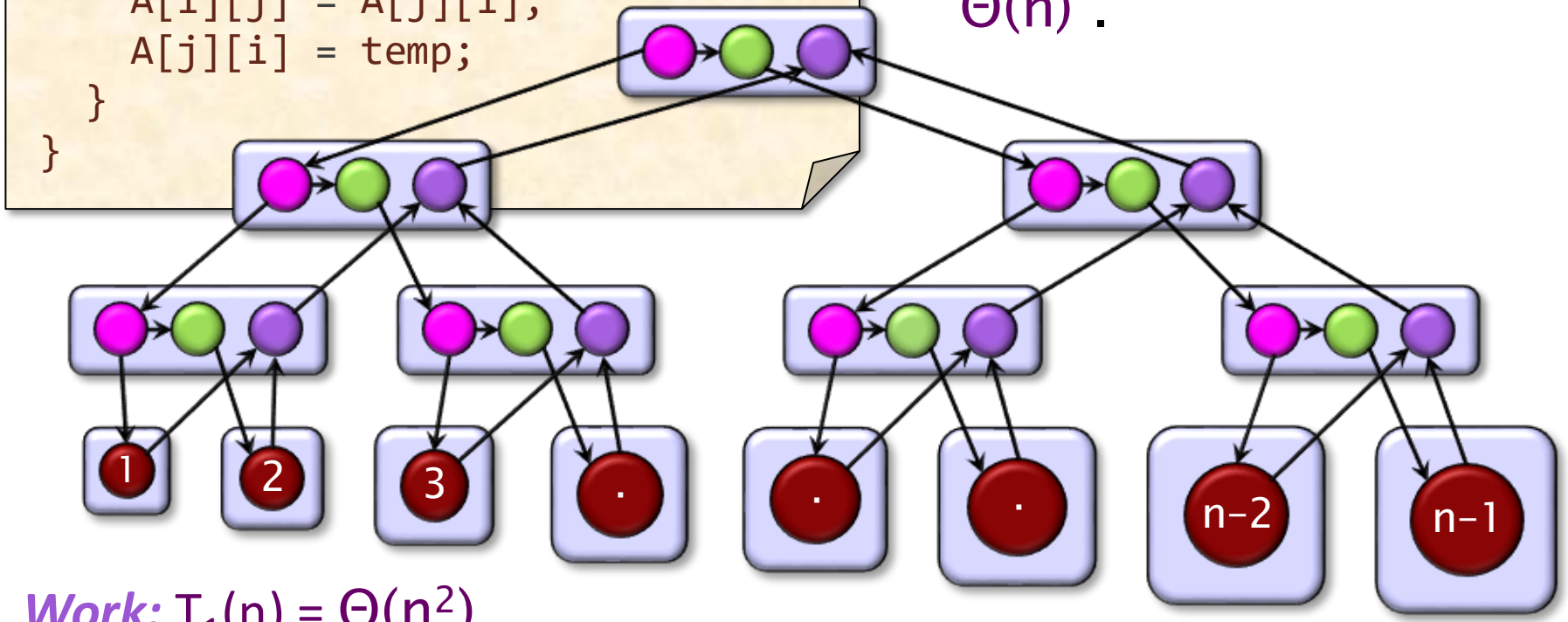


Analysis of Parallel Matrix Transpose

```
// 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 = $\Theta(\log n)$.

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



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

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

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

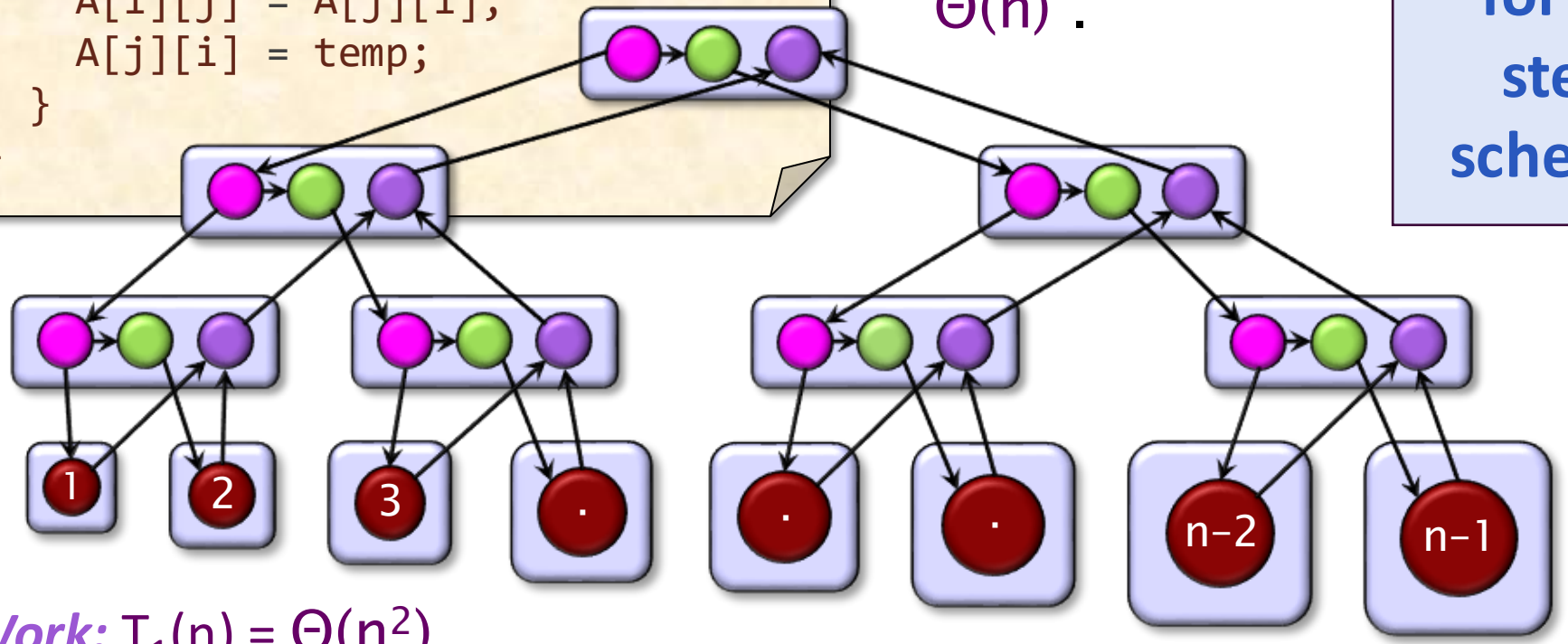
Analysis of Parallel Matrix Transpose

```
// 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 = $\Theta(\log n)$.

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

Ideal format for work-stealing scheduling!



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

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

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 = $\Theta(\log n)$.

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

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

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

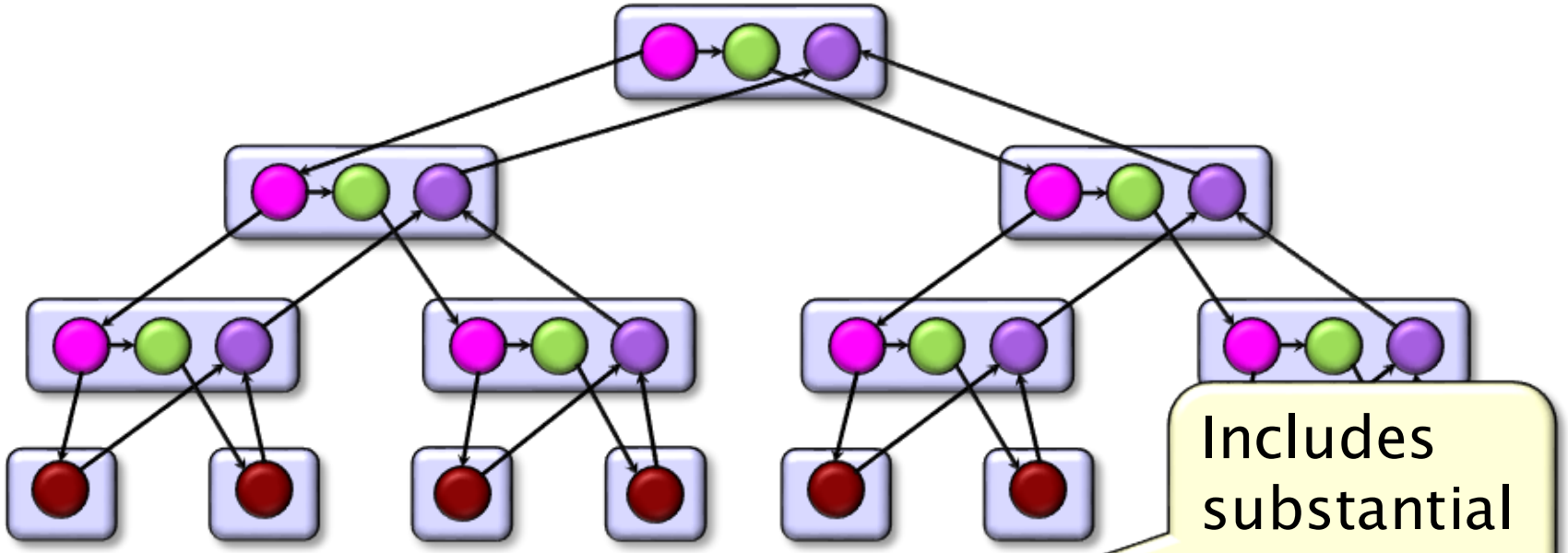
Span: $T_\infty(n) = \Theta(\log n)$

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

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

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

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);
```

Optimizing Parallel-Loop Control

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

Original code

Recall: $T_P \leq \frac{T_1}{P} + T_\infty$

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];
}
```

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

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);
```

Coarsening Parallel Loops

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

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

$$\textbf{Chunk size} \approx \frac{total\ iters}{C \times cores}$$

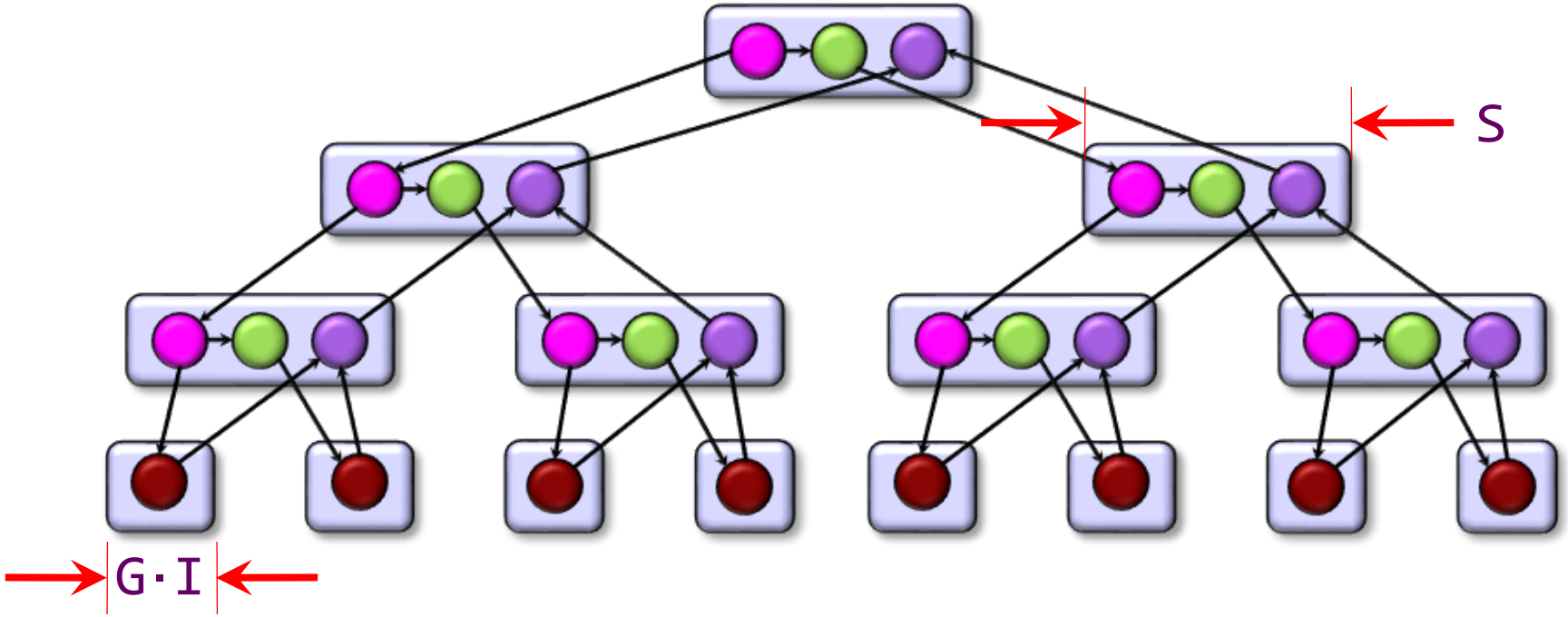
Compiler-generated recursion

```
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);
```

Loop Grain Size

Vector addition

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

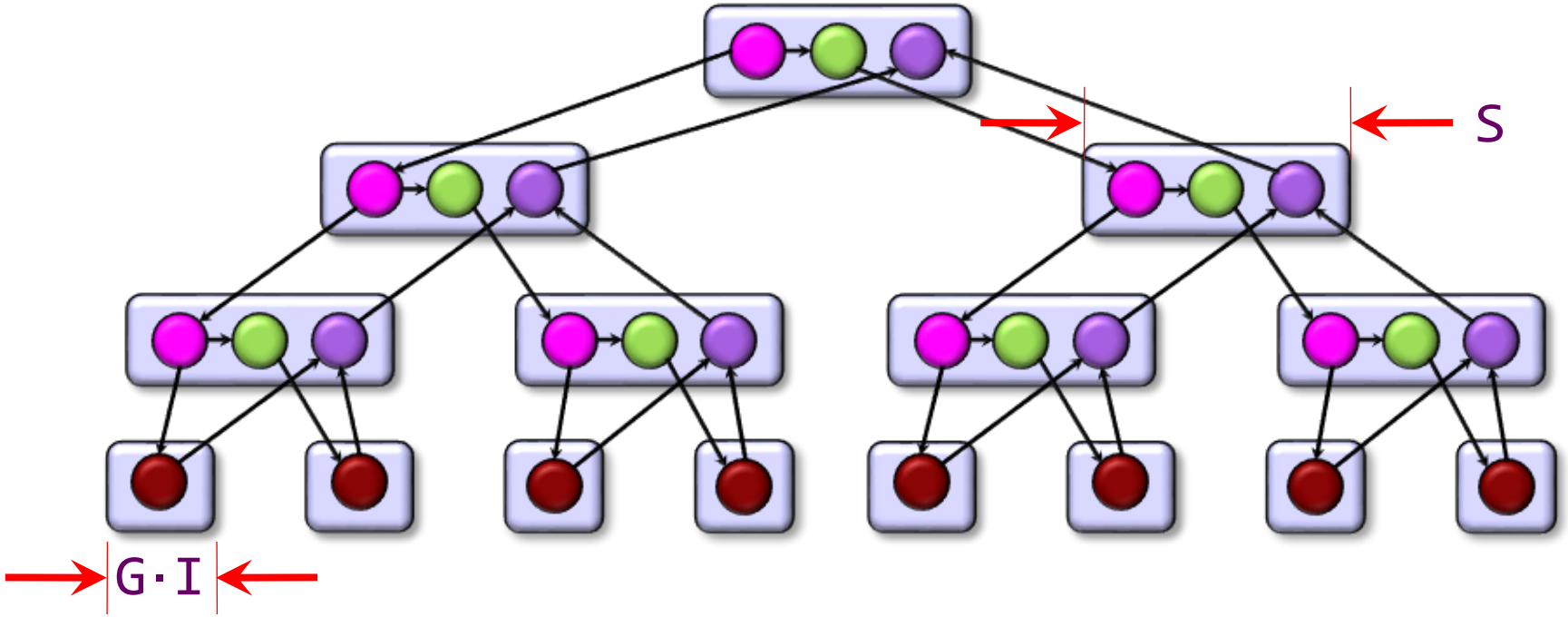


Let I be the time for one iteration of the loop body.
Let S be the time to perform a level of the recursion.

Loop Grain Size

Vector addition

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

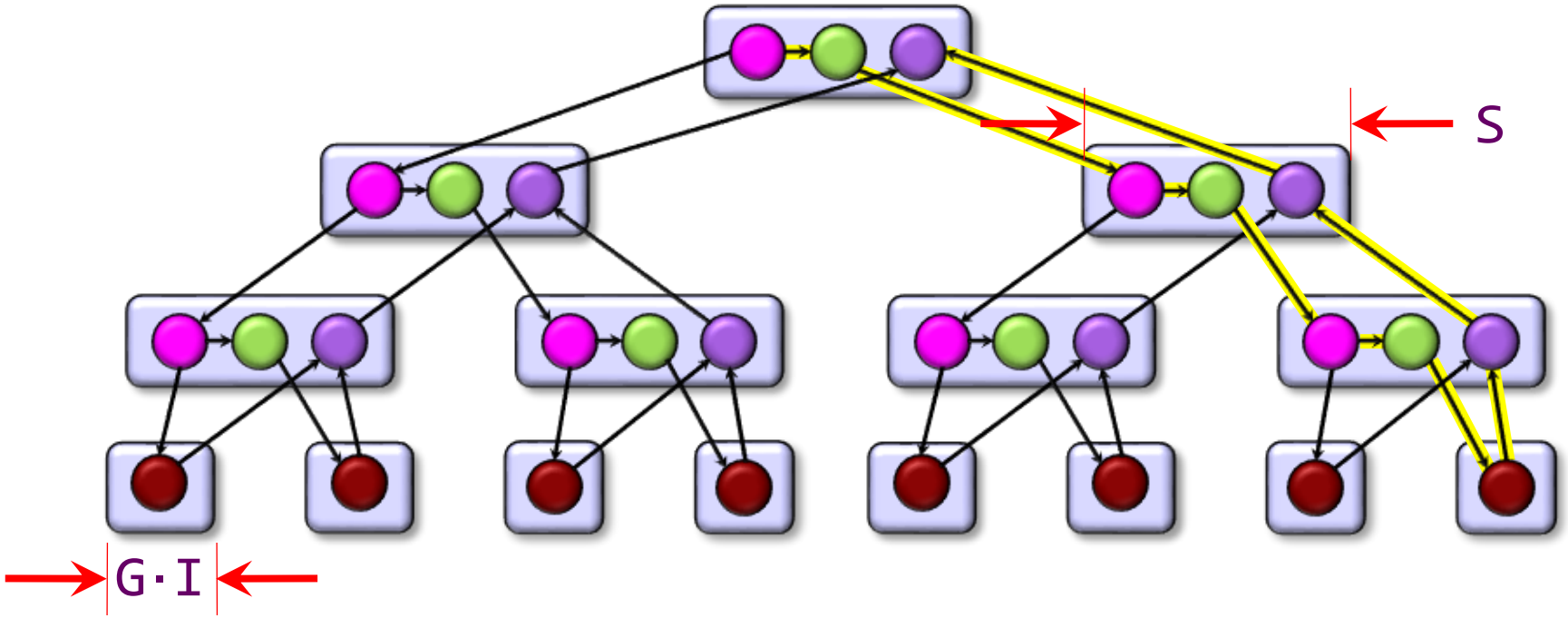


Work: $T_1 = n \cdot I + (n/G - 1) \cdot S$

Loop Grain Size

Vector addition

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



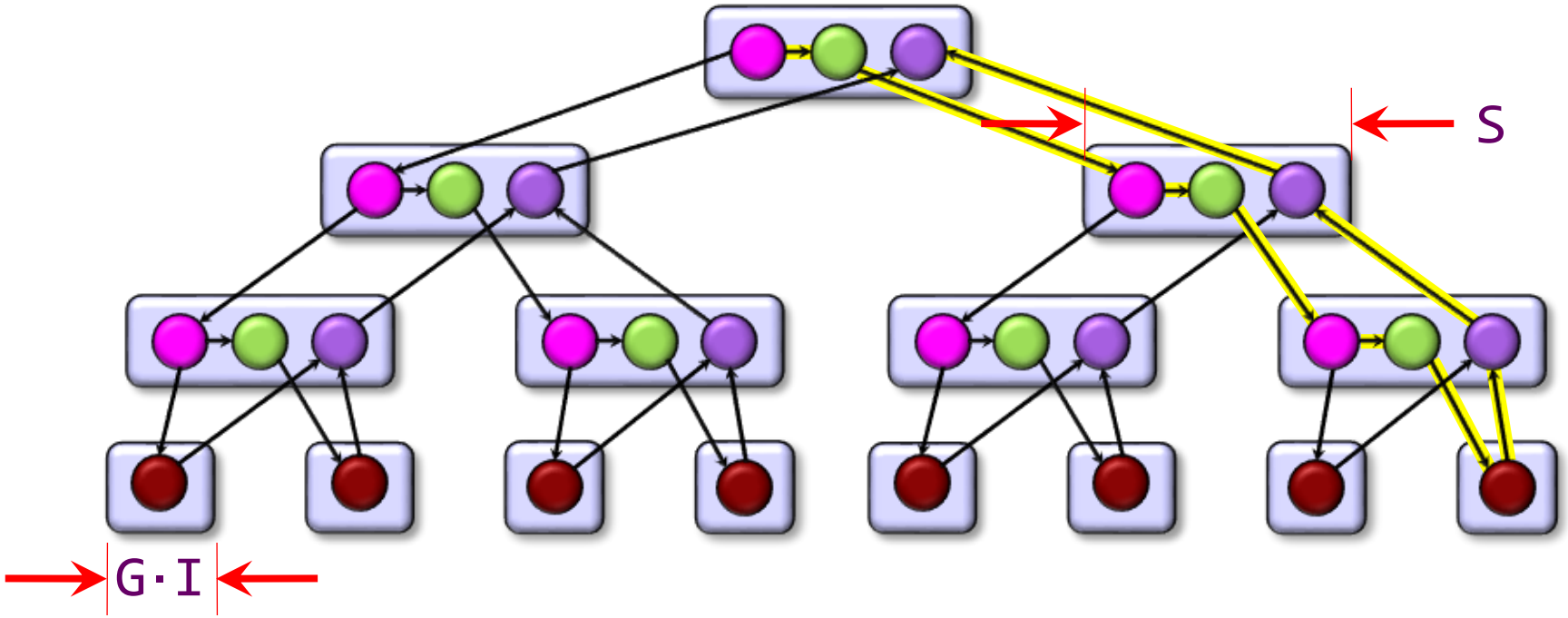
Work: $T_1 = n \cdot I + (n/G - 1) \cdot S$

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

Loop Grain Size

Vector addition

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



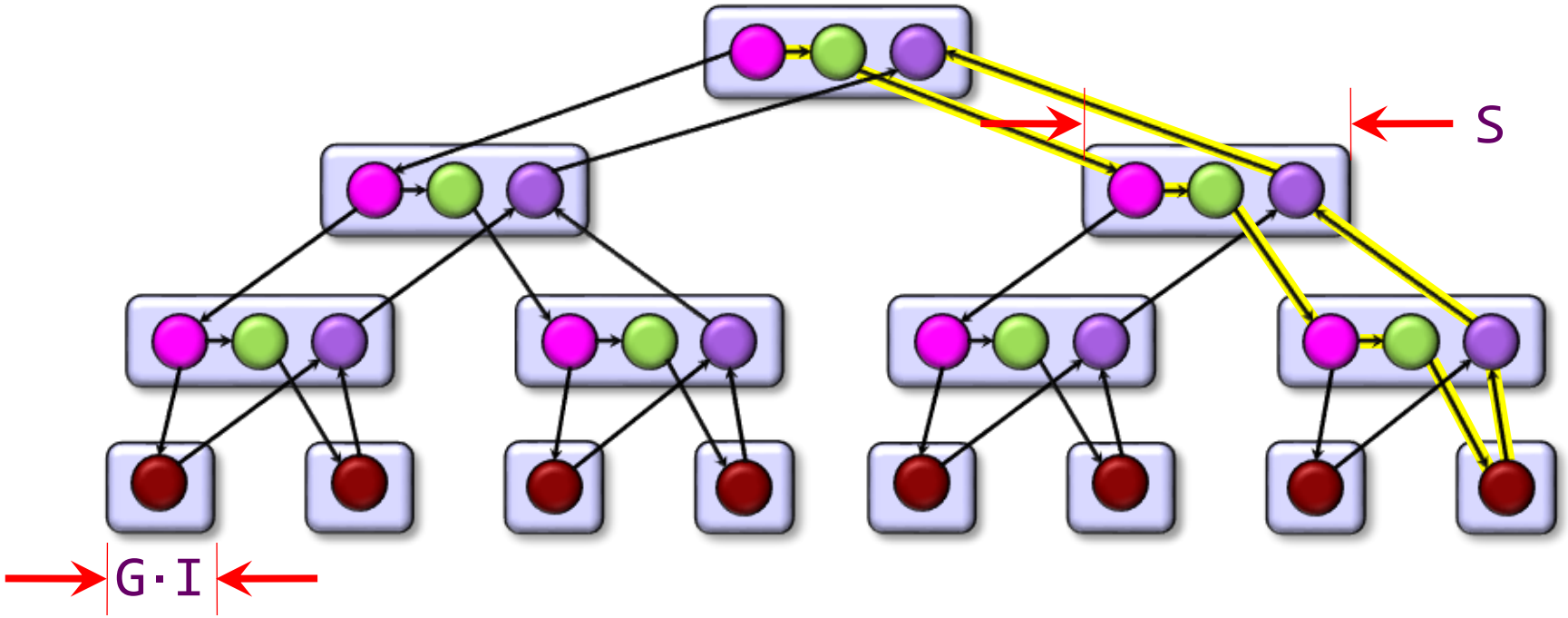
Work: $T_1 = n \cdot I + (n/G - 1) \cdot S$

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

Loop Grain Size

Vector addition

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



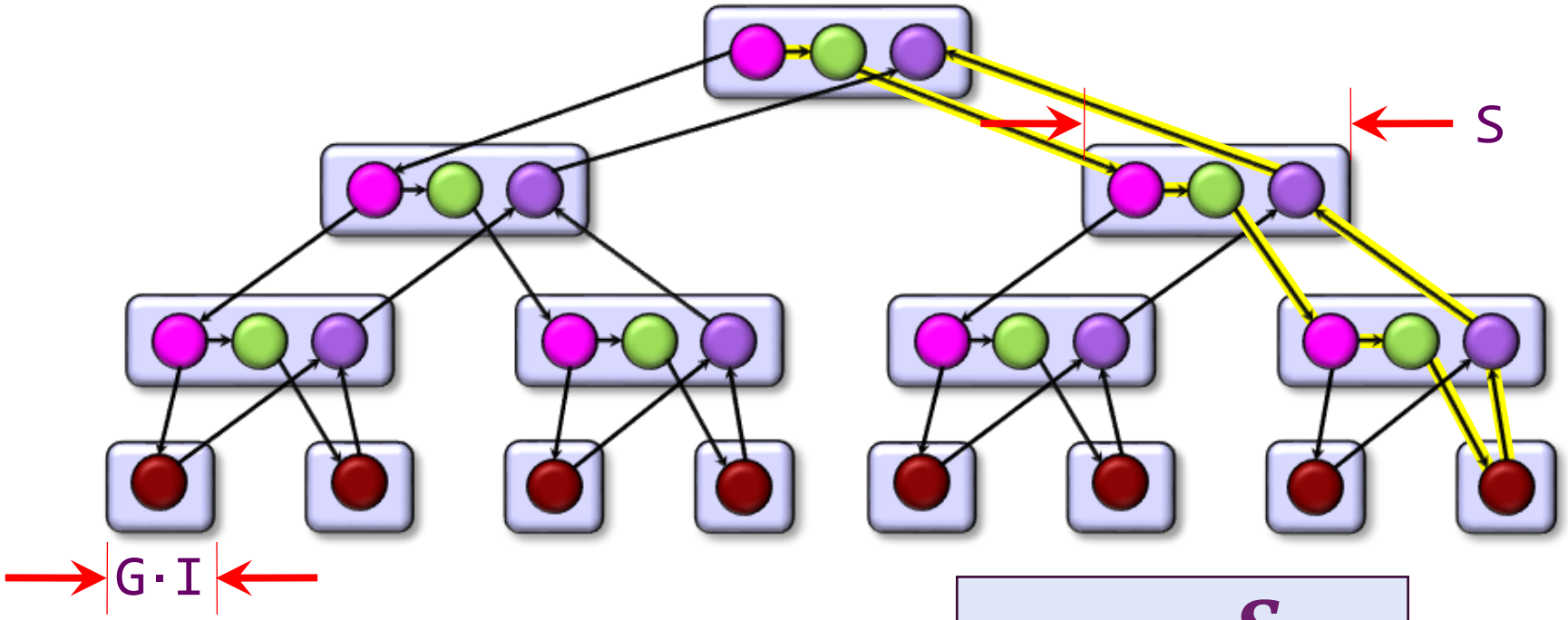
Work: $T_1 = n \cdot I + (n/G - 1) \cdot S$

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

Loop Grain Size

Vector addition

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



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

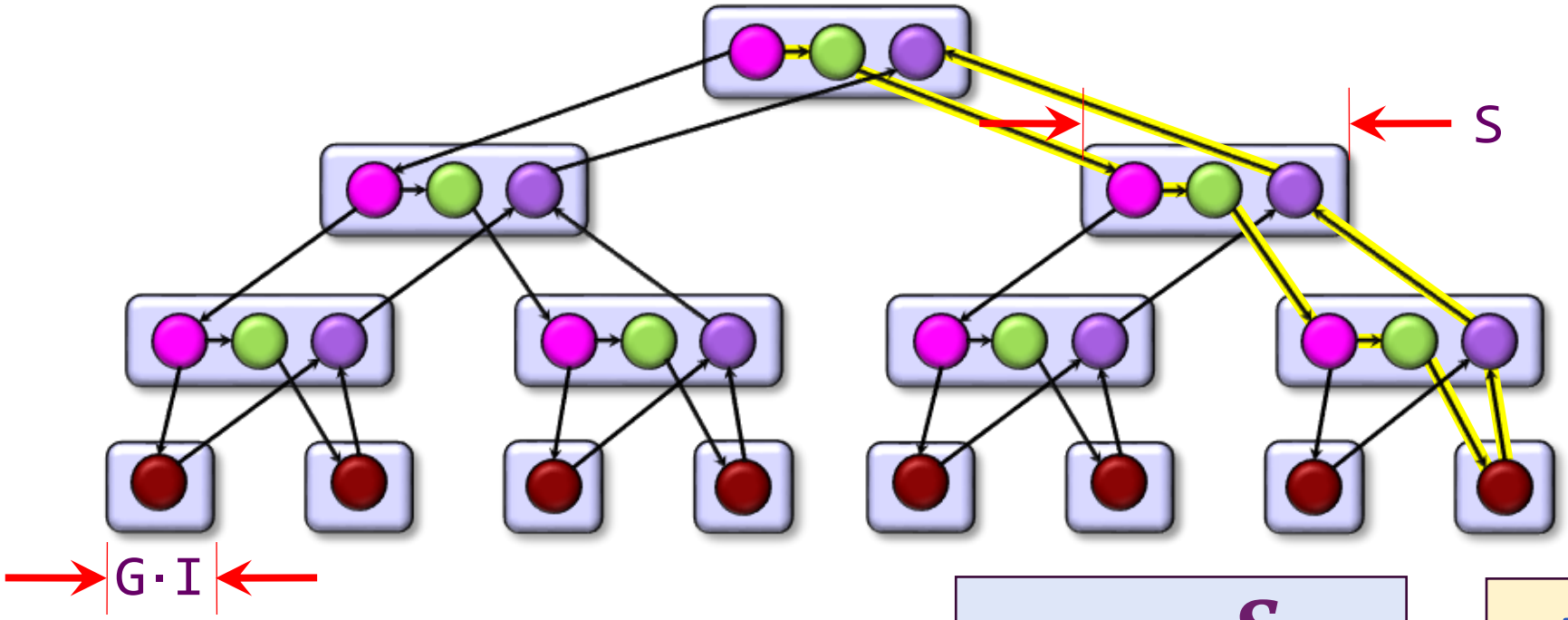
Span: $T_\infty = G \cdot I + \log(n/G) \cdot S$

$$I \gg \frac{S}{G}$$

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

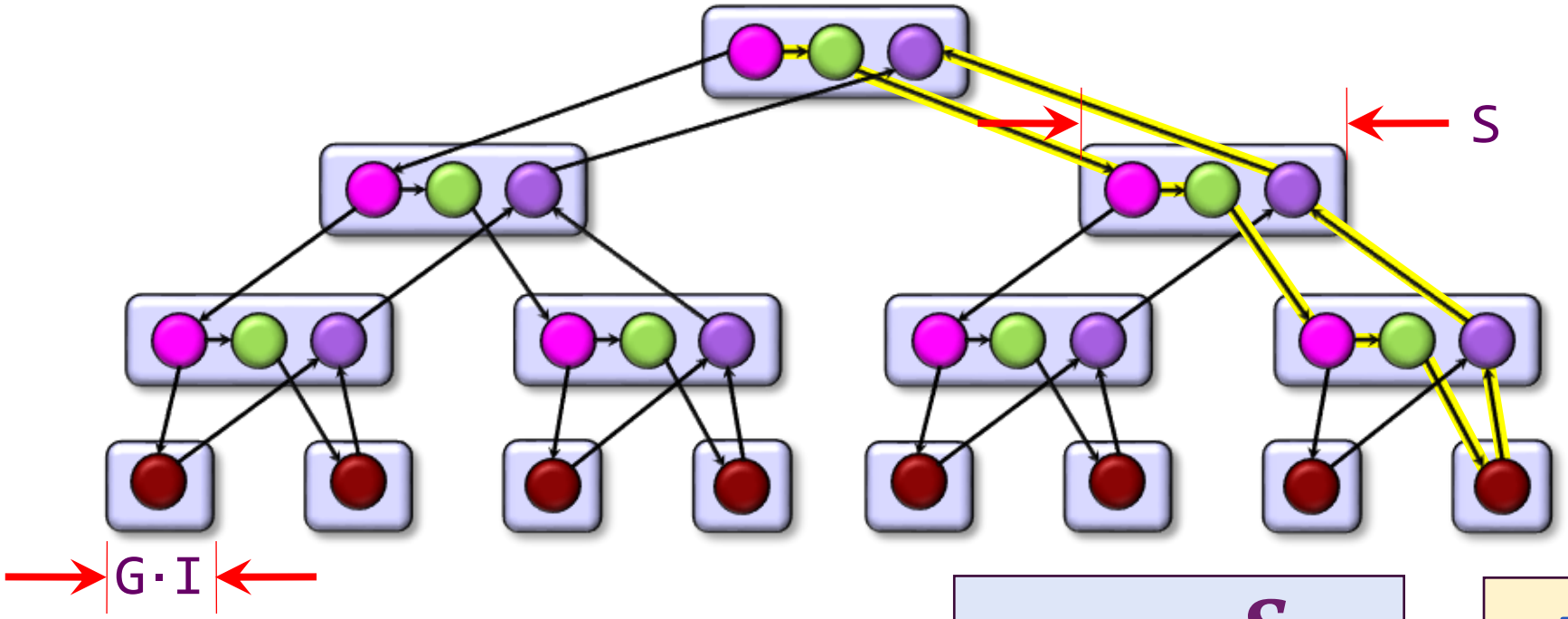
$$I \gg \frac{S}{G}$$

First term dominates

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

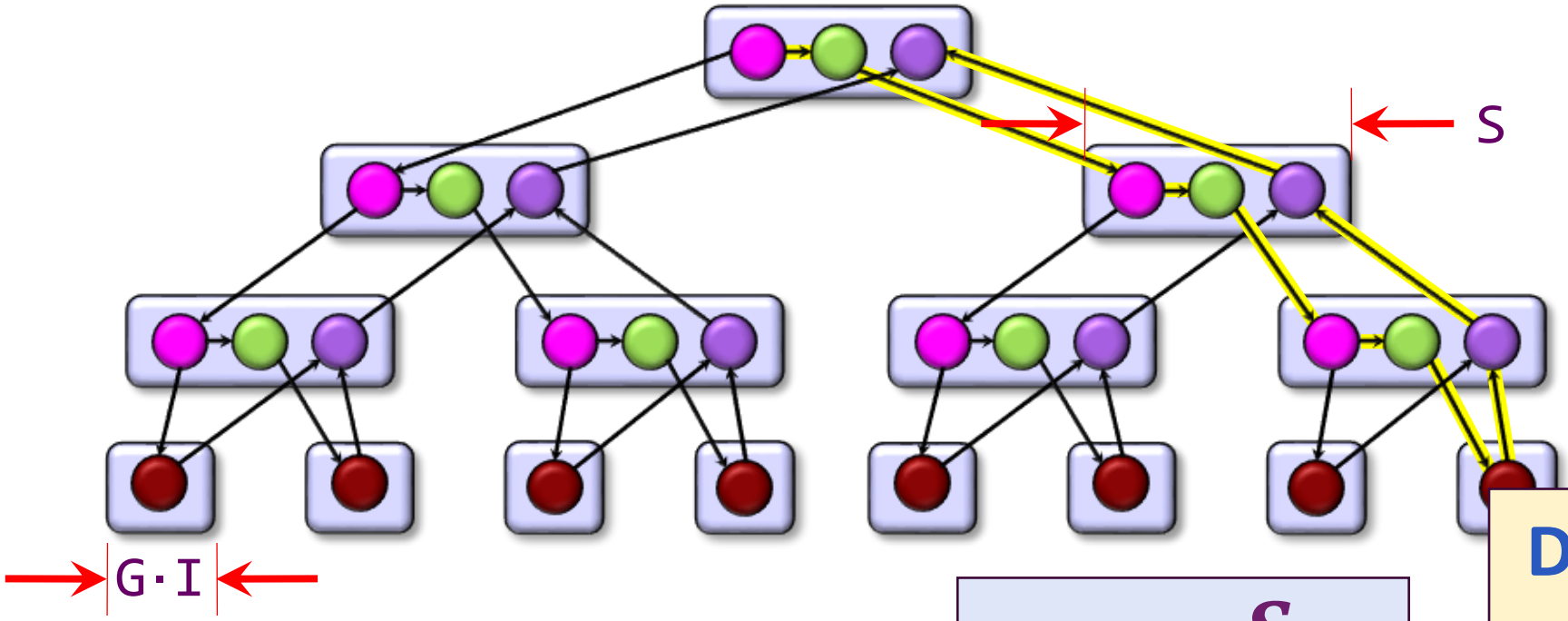
$$G \gg \frac{S}{I}$$

First term dominates

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

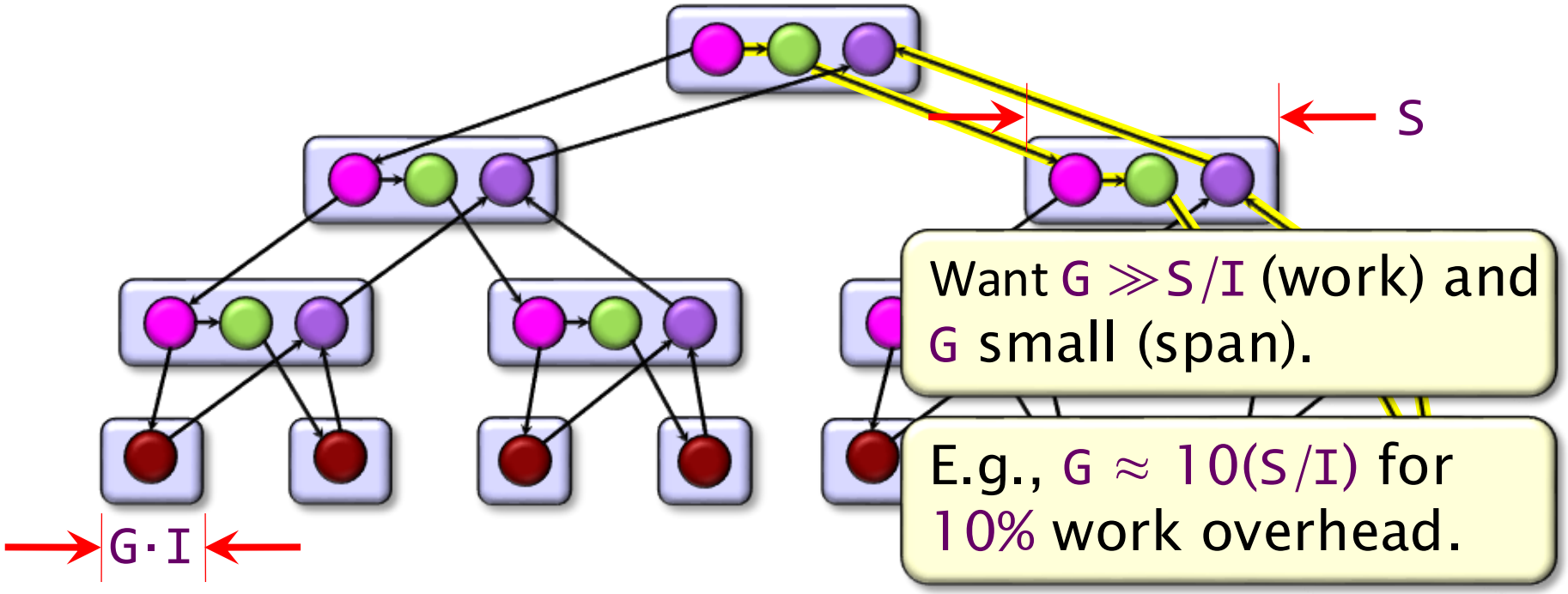
$$G \gg \frac{S}{I}$$

Diminishing returns as G increases

Loop Grain Size

Vector addition

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



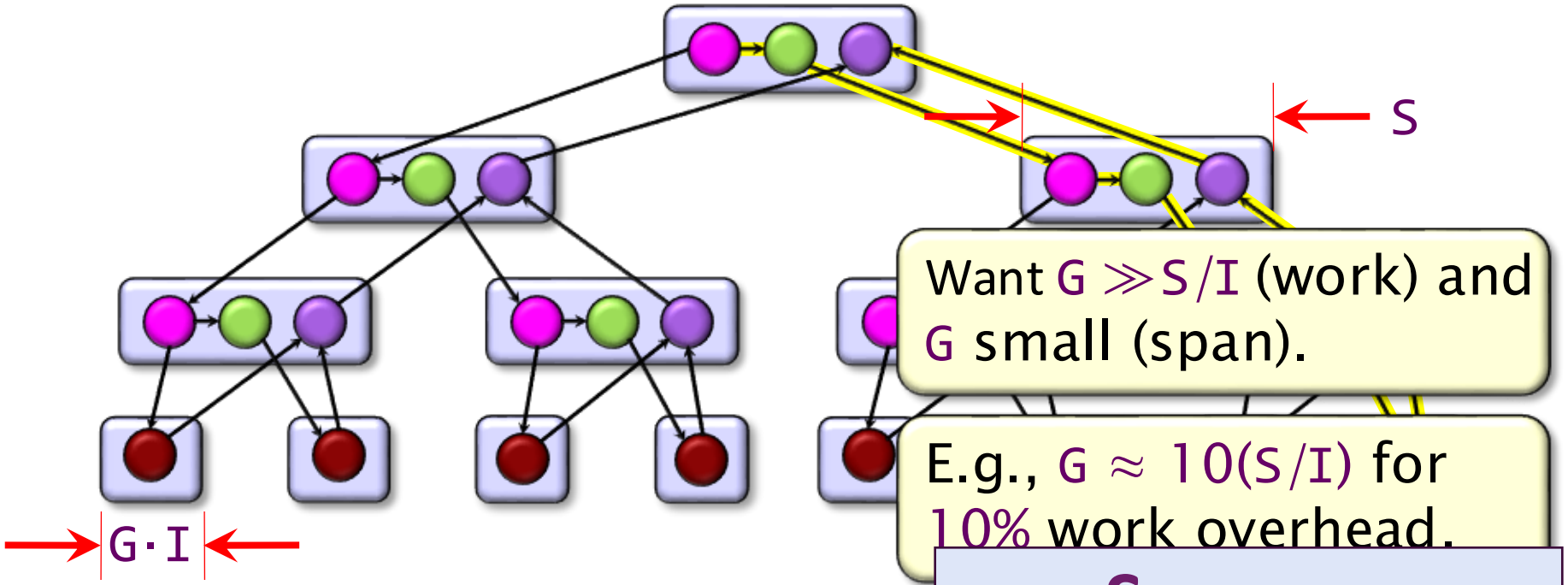
Work: $T_1 = n \cdot I + (n/G - 1) \cdot S$

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

Loop Grain Size

Vector addition

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



Want $G \gg S/I$ (work) and G small (span).

E.g., $G \approx 10(S/I)$ for 10% work overhead.

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

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

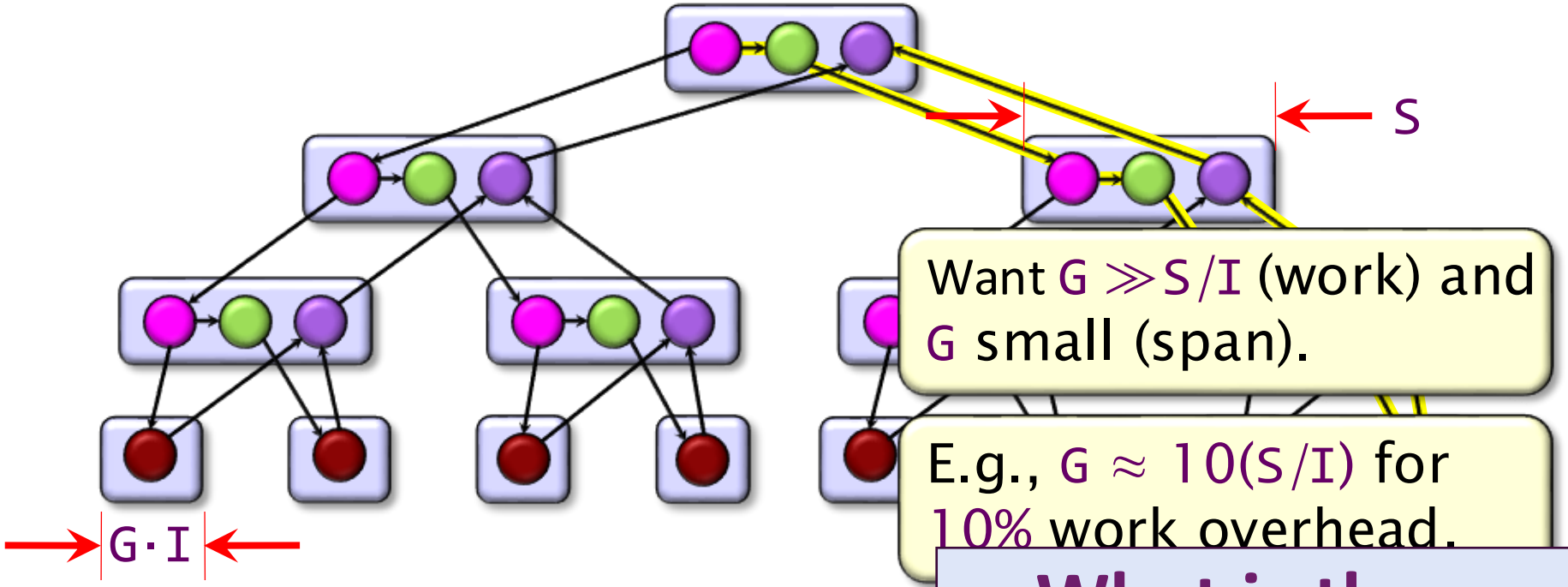
Suppose

$$S = O(\log n)$$

Loop Grain Size

Vector addition

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



Want $G \gg S/I$ (work) and G small (span).

E.g., $G \approx 10(S/I)$ for 10% work overhead.

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

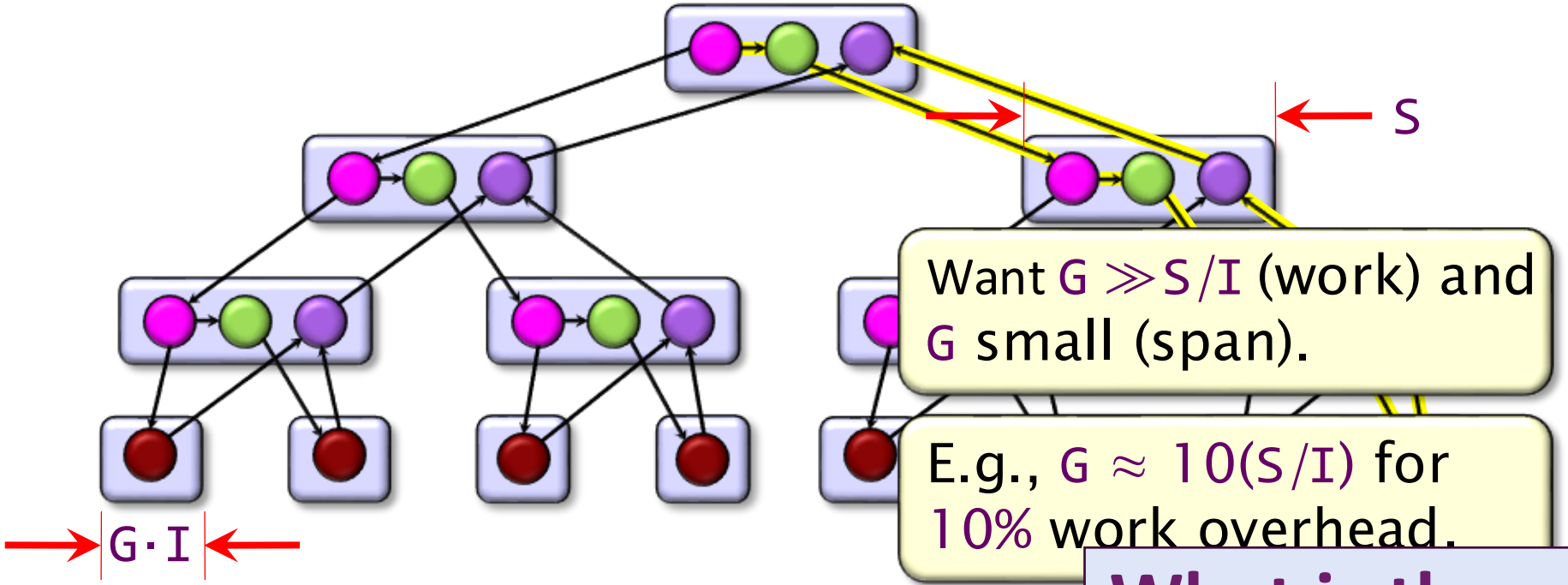
Span: $T_\infty = G \cdot I + \log(n/G) \cdot S$

What is the resulting work and span?

Loop Grain Size

Vector addition

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



Want $G \gg S/I$ (work) and G small (span).

E.g., $G \approx 10(S/I)$ for 10% work overhead.

Work: $T_1 = \Theta(n)$

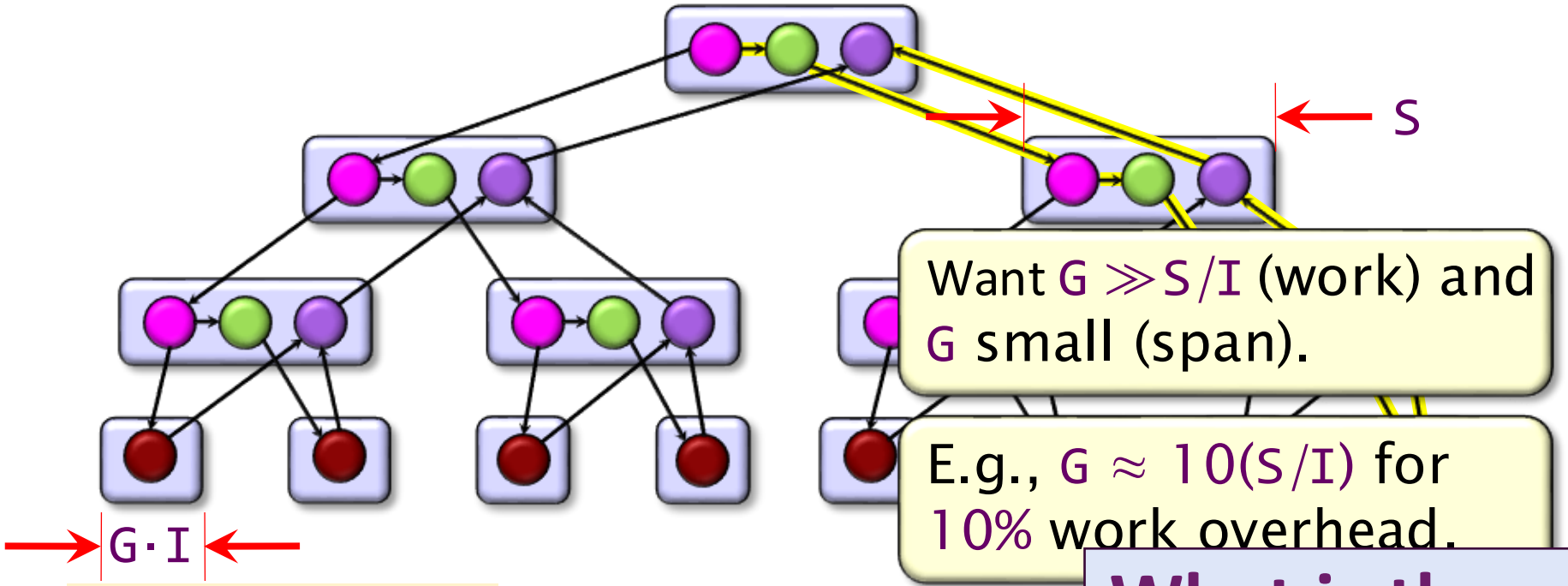
Span: $T_\infty = \Theta(\log n) + \Theta\left(\log\left(\frac{n}{\log(n)}\right)\right) \cdot \log(n)$

What is the resulting work and span?

Loop Grain Size

Vector addition

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



Want $G \gg S/I$ (work) and G small (span).

E.g., $G \approx 10(S/I)$ for 10% work overhead.

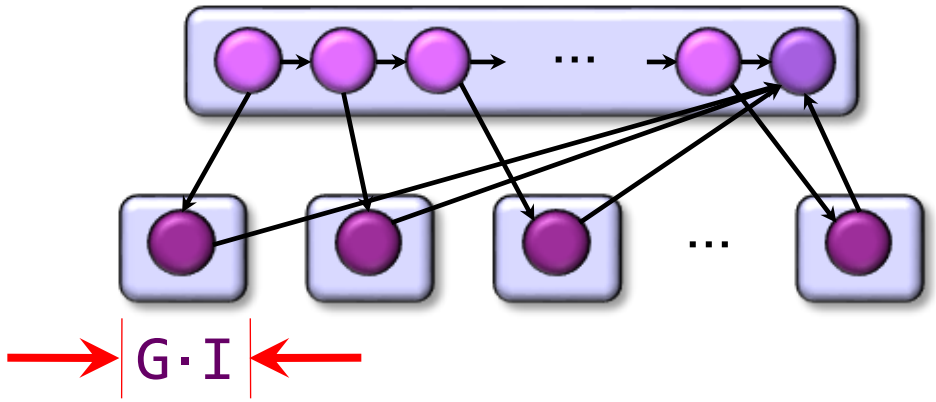
Work: $T_1 = \Theta(n)$

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

What is the resulting work and span?

Another Implementation

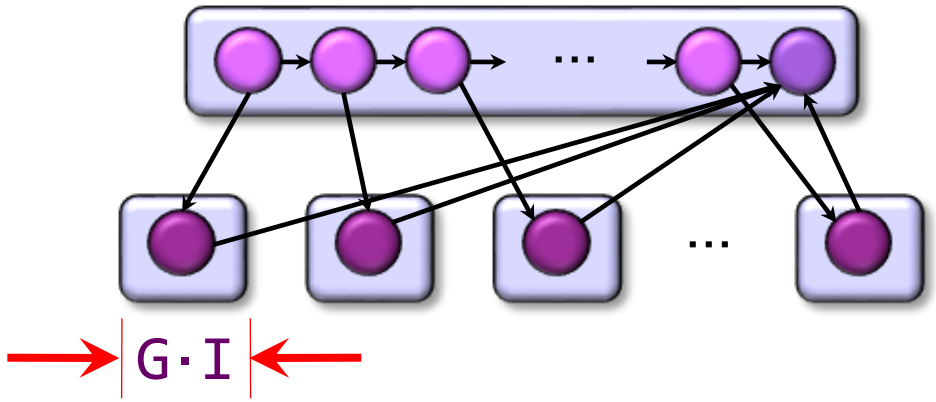
```
void vadd (double *A, double *B, int n){
  cilk_scope {
    for (int j=0; j<n; j+=G) {
      cilk_spawn {
        for (int i=j; i<MIN(j+G,n); i++)
          A[i] += B[i];
} } } }
```



cilk_scope: spawned threads must complete before leaving

Another Implementation

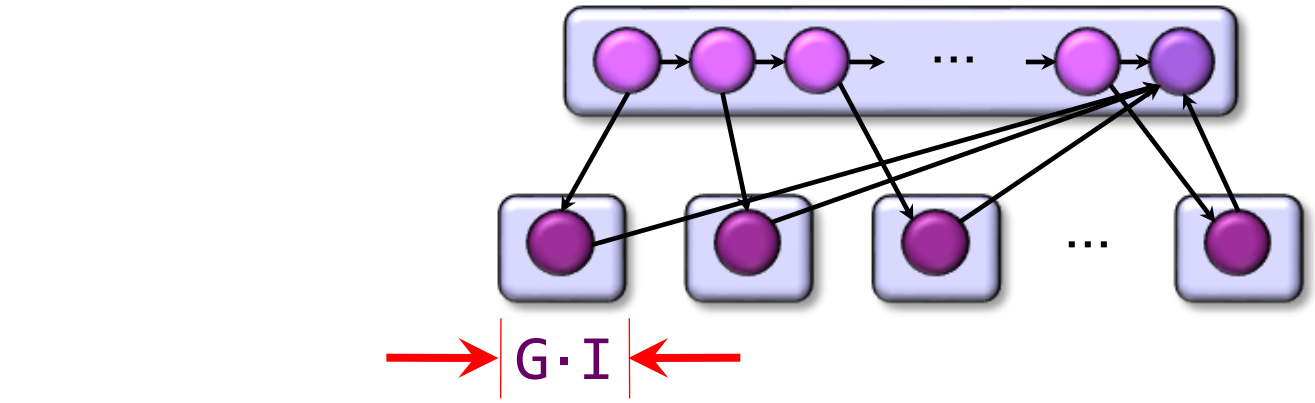
```
void vadd (double *A, double *B, int n){
  cilk_scope {
    for (int j=0; j<n; j+=G) {
      cilk_spawn {
        for (int i=j; i<MIN(j+G,n); i++)
          A[i] += B[i];
} } } }
```



cilk_spawn: spawns a task that can be completed in parallel

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];
} } } }
```



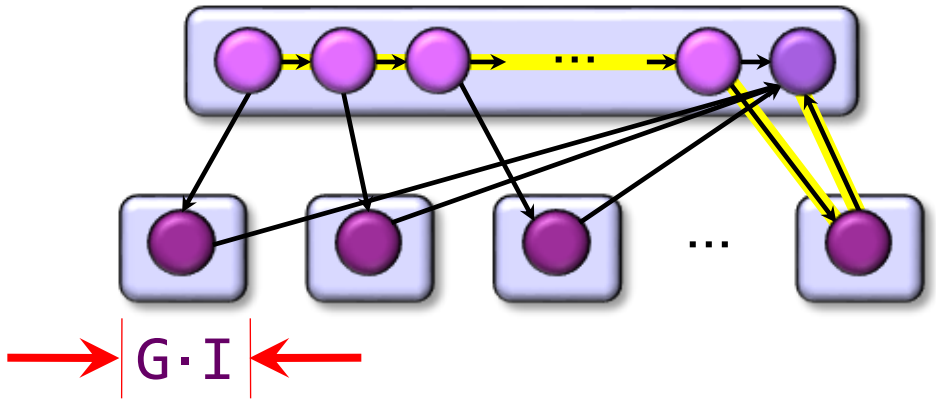
Assume that $G = 1$.

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];
} } } }
```



No parallelism!!

Assume that $G = 1$.

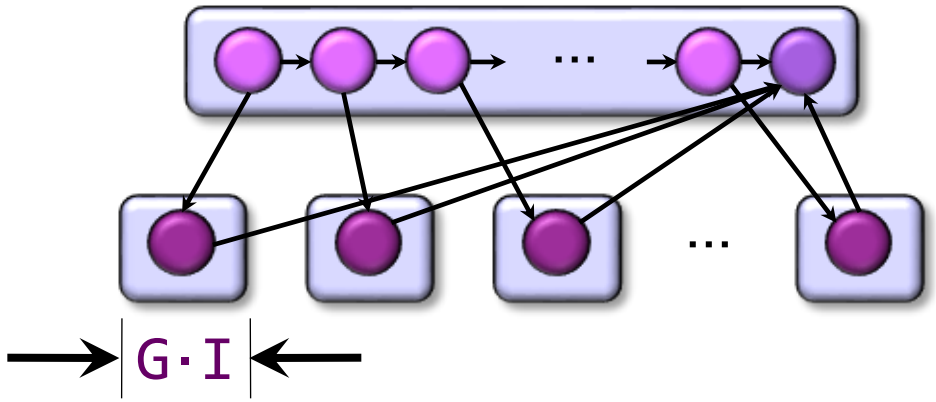
Work: $T_1 = \Theta(n)$

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

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

Another Implementation

```
void vadd (double *A, double *B, int n){
  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:

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: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(\log(n/32)+32)$
$= \Theta(\log n)$

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

Code B

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

Work: $T_1 = \Theta(n)$

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

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 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: $T_1 = \Theta(n)$

Span: $T_\infty = \Theta(\log(n/32)+32)$
$= \Theta(\log n)$

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

Still Small!

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: $T_1 = \Theta(n)$

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

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:

```
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 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.