From

Modern Algorithms Workshop

Parallel Algorithms

Prof. Charles E. Leiserson

Dr. Tao B. Schardl

September 19, 2018

Outline

- Introduction
- Cilk Model
- Detecting Nondeterminism
- What Is Parallelism?
- Scheduling Theory Primer
- *Lunch Break*
- Analysis of Parallel Loops
- Case Study: Matrix Multiplication
- Case Study: Jaccard Similarity
- Post-Moore Software

ANALYSIS OF PARALLEL LOOPS

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;
  }
}
```

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

Divide-and-conquer implementation

The Tapir/LLVM compiler implements `cilk_for` loops this way at optimization level `-O1` or 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 recur(int lo, int hi)  //half open
{
  if (hi > lo + 1) {
    int mid = lo + (hi - lo)/2;
    cilk_spawn recur(lo, mid);
               recur(mid, hi);
    cilk_sync;
    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;
  }
}
⋮
recur(1, n);
```

Divide-and-conquer implementation

lifted
loop body

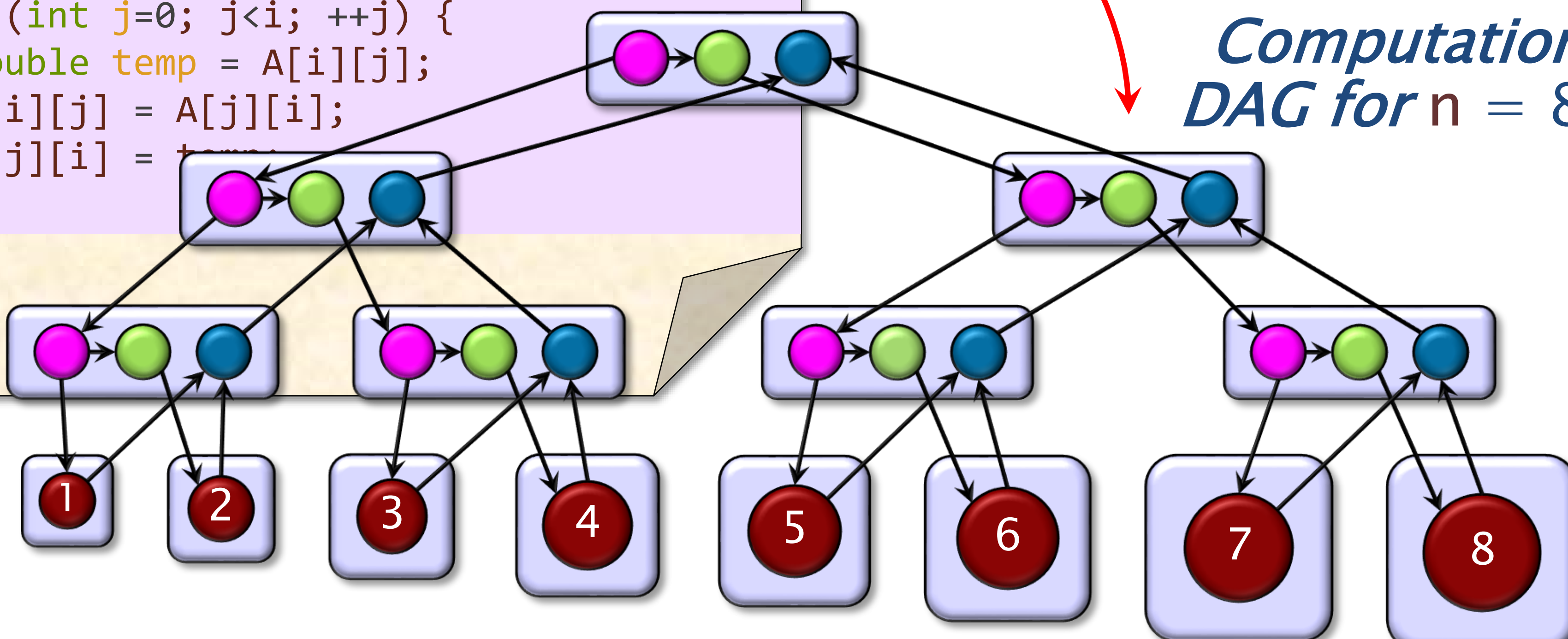
Execution of Parallel Loops

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

Divide-and-conquer implementation

cilk_for
loop control

Computation DAG for $n = 8$

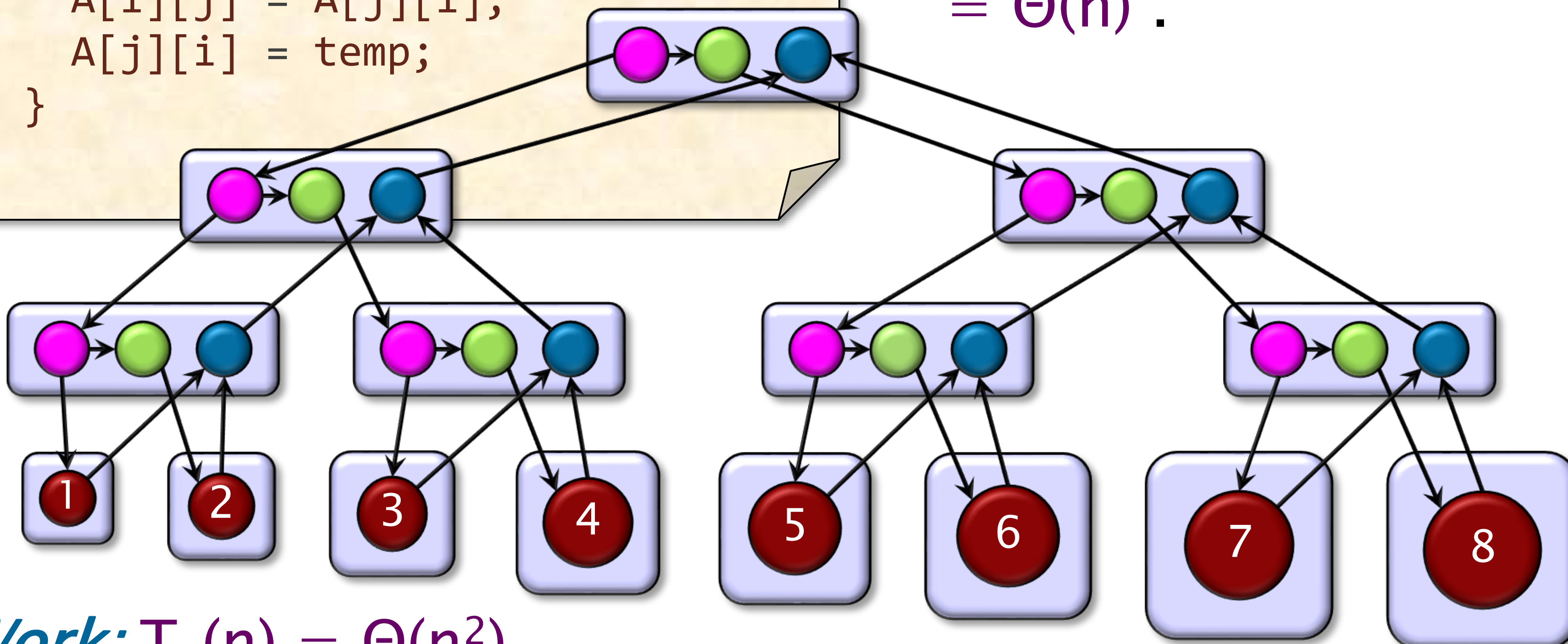


Analysis of Parallel Loops

```
// 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;
  }
}
```

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 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 $= \Theta(\lg n)$.

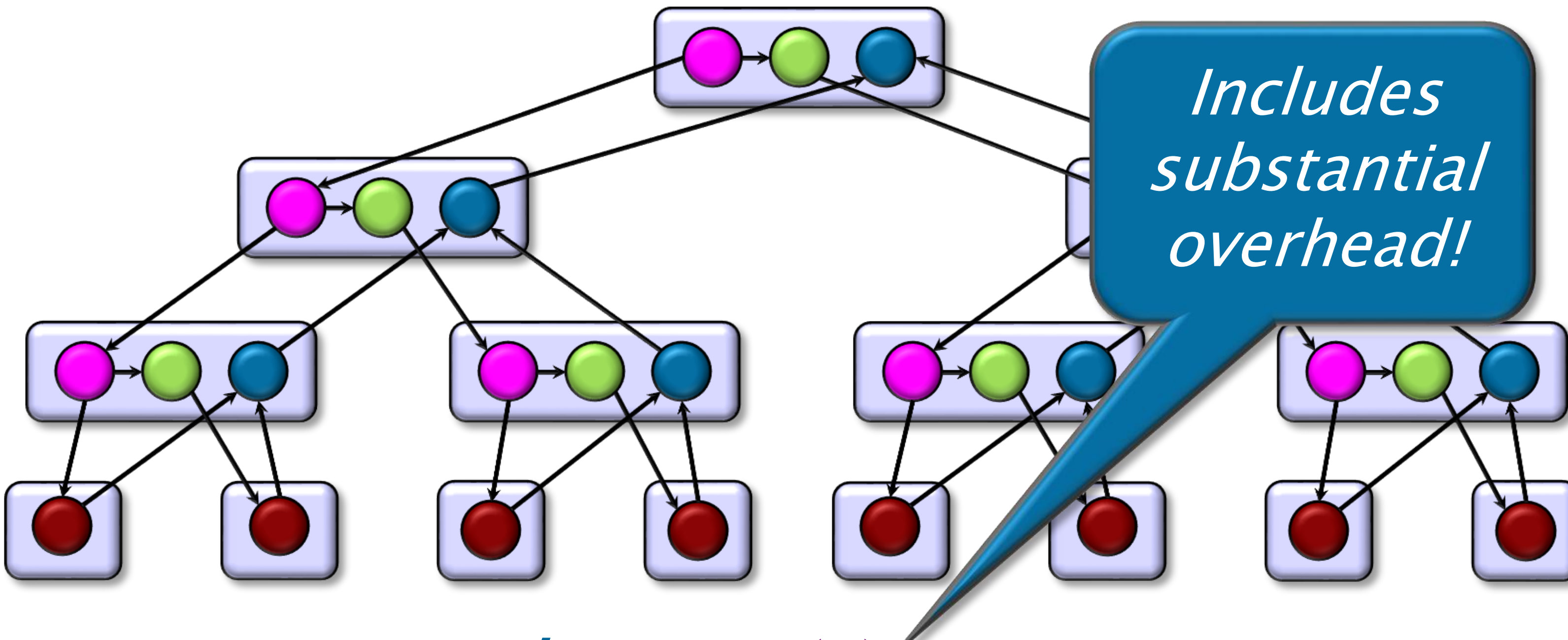
Span of body $= \Theta(1)$.

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

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

Coarsening Parallel Loops

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

Implementation with coarsening

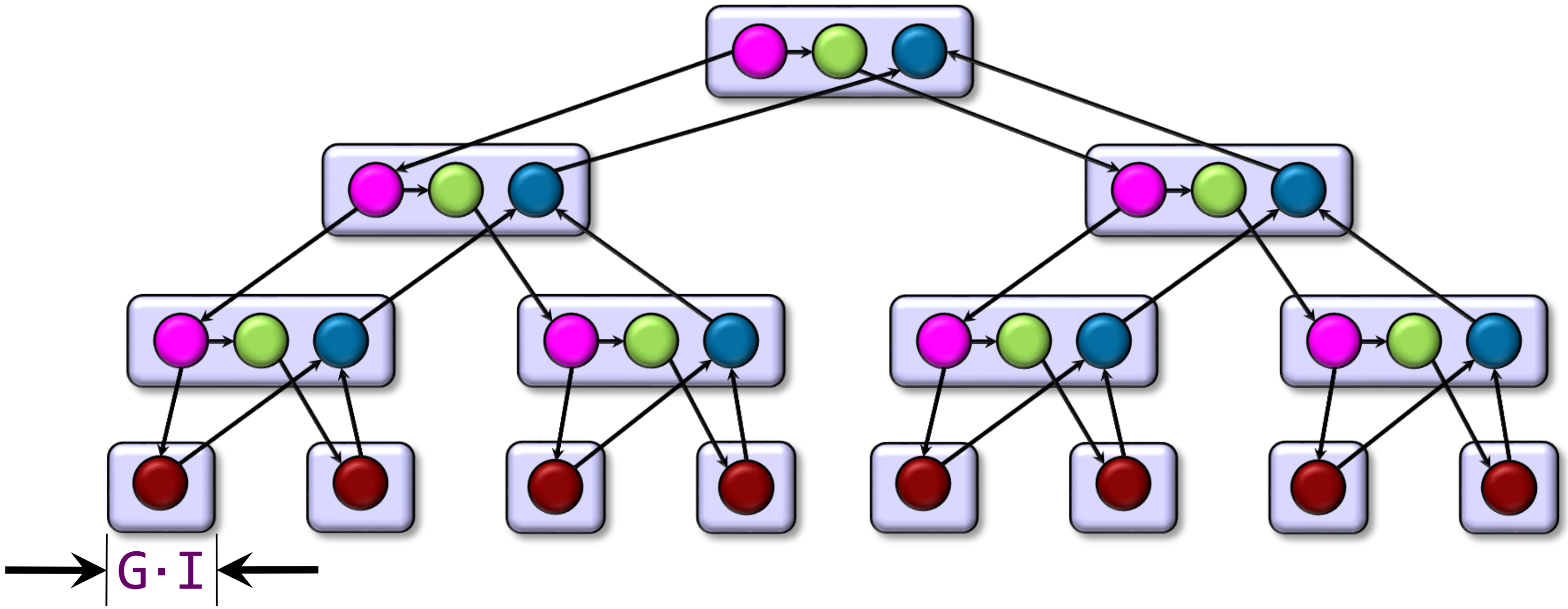
If a grainsize pragma is not specified, the Cilk runtime system makes its best guess to minimize overhead.

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

Loop Grain Size

Vector addition

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

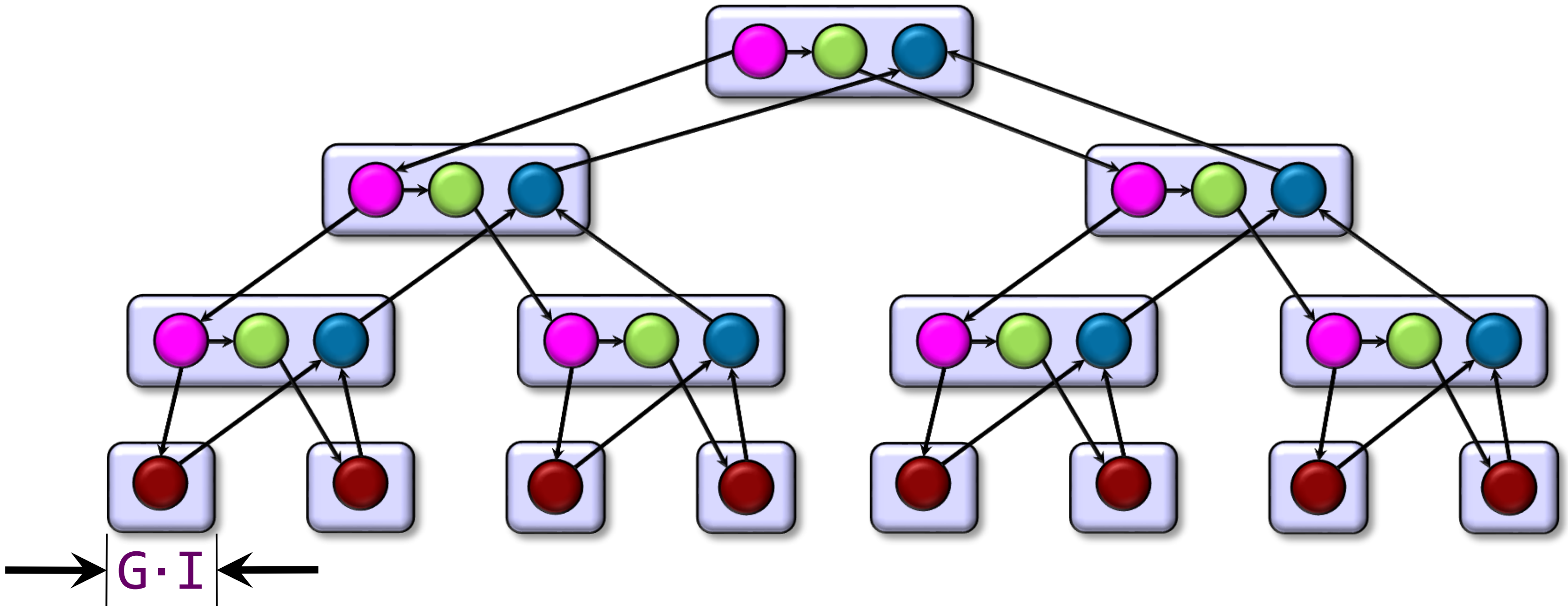


Let I be the time for one iteration of the loop body.
Let S be the time to perform a spawn and return.

Loop Grain Size

Vector addition

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



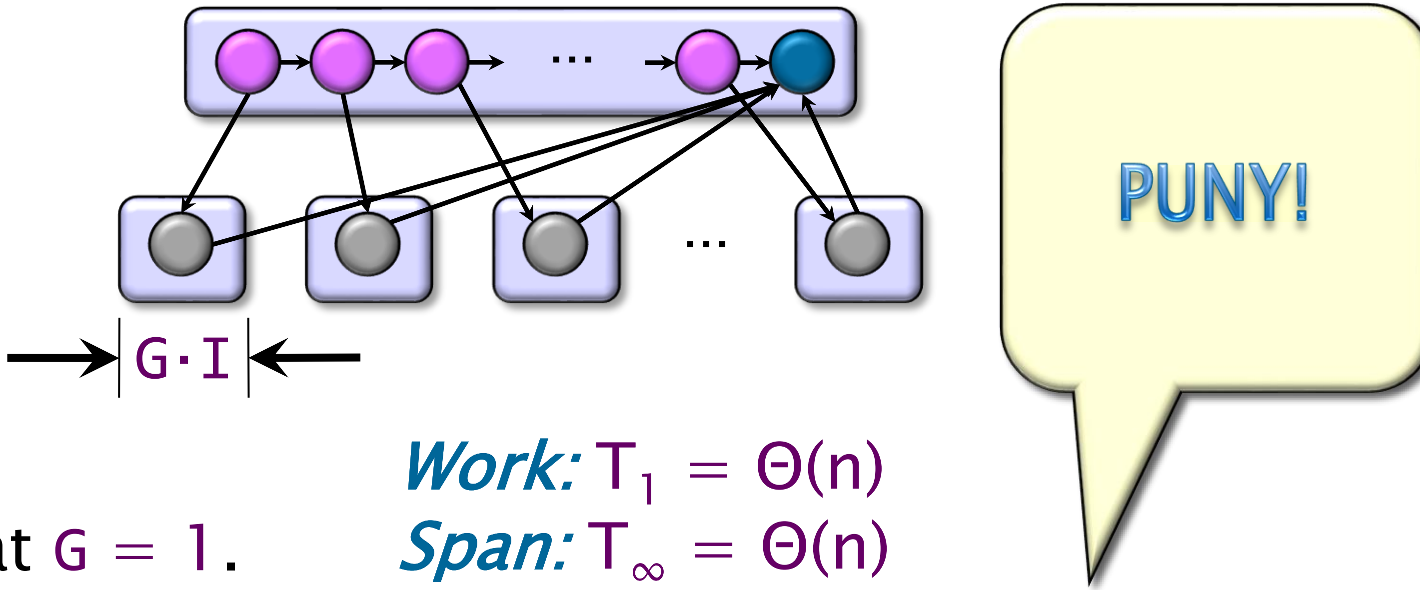
Work: $T_1 = n \cdot I + (n/G - 1) \cdot S$

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

Want $G \gg S/I$ 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;
```

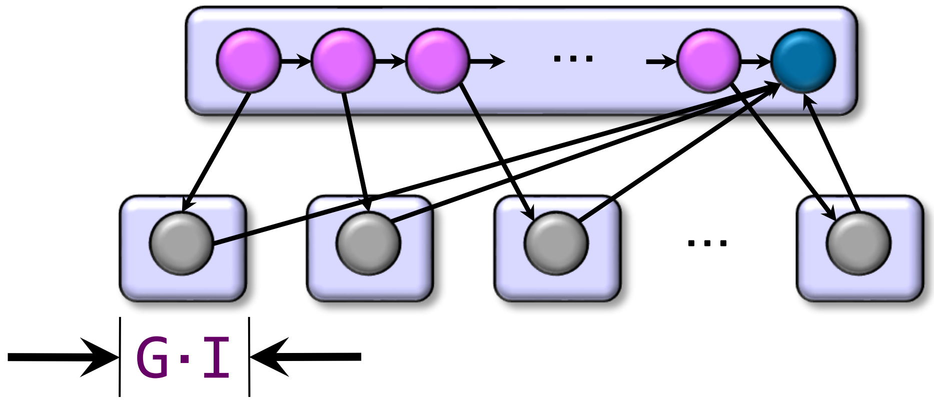


Assume that G = 1.

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

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

Quiz on Parallel Loops

Question: Let P be the number of workers on the system. How does the parallelism of the first code compare to that of the second? (Differences highlighted.)

```
#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: $T_\infty = \Theta(\lg(n/32) + 32)$
$= \Theta(\lg n)$

Parallelism:
T_1/T

PUNY!

Want $T_1/T_\infty \gg P$.

```
#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: $T_\infty = \Theta(\lg P +$ $P)$
$= \Theta(n/P)$

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:

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

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);
}
```