CSE 613: Parallel Programming

Lecture 3
(The Cilk++ Concurrency Platform)

(inspiration for many slides comes from talks given
by Charles Leiserson and Matteo Frigo)

Rezaul A. Chowdhury

Department of Computer Science
SUNY Stony Brook
Spring 2019

The Cilk++ Concurrency Platform

— Supports *dynamic multithreading*

— Includes a small set of *linguistic extensions* to C++ to support *fork-join* parallelism

— Based on multithreaded language technology developed at MIT and MIT spin-off *Cilk Arts* (acquired by *Intel* in 2009)

— Includes

- A provably efficient scheduler
- Hyperobject library for parallelizing code with global variables
- Race detector (*Cilkscreen*)
- Scalability analyzer (*Cilkview*)

The Cilk++ Concurrency Platform

Download URL

- Open Cilk @ MIT Cilk Hub:

  http://cilk.mit.edu/

- Intel® Cilk Plus:

  https://www.cilkplus.org/

Serial to Parallel using Three Keywords

Nested Parallelism in Cilk++

$$^{n}C_{r} = {}^{n-1}C_{r-1} + {}^{n-1}C_{r}$$

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = comb( n – 1, r – 1 );
  y = comb( n – 1, r );

  return ( x + y );
}
```

Serial C++ code

Grant permission to execute the called (spawned) function in parallel with the caller.

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

```
  comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r – 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```

Function return enforces implicit synchronization.

Oblivious of the number of cores / processors!

Cilk++ code

Loop Parallelism in Cilk++

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \xrightarrow[\text{transpose}]{\text{in-place}} \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}$$

```
for ( int i = 1; i < n; ++i )
   for ( int j = 0; j < i; ++j )
      {
        double t = A[ i ][ j ];
        A[ i ][ j ] = A[ j ][ i ];
        A[ j ][ i ] = t;
      }
```

Serial C++ code

Allows all iterations of the loop to be executed in parallel.

Converted to spawns and syncs using recursive divide-and-conquer.

```
cilk_for ( int i = 1; i < n; ++i )
   for ( int j = 0; j < i; ++j )
      {
        double t = A[ i ][ j ];
        A[ i ][ j ] = A[ j ][ i ];
        A[ j ][ i ] = t;
      }
```

Cilk++ code

Measuring
Parallel Performance

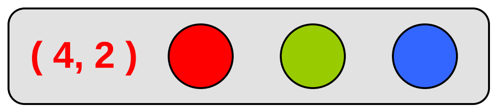
Cilk++ Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```

Cilk++ Execution Model

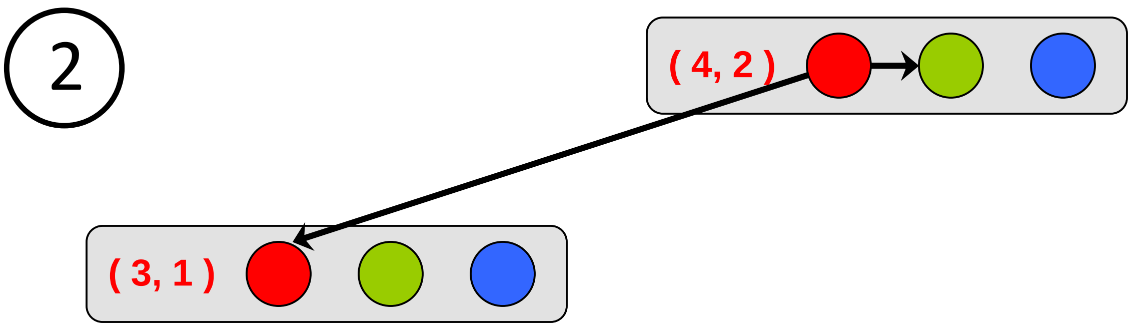
```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r – 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```

1



Cilk++ Execution Model

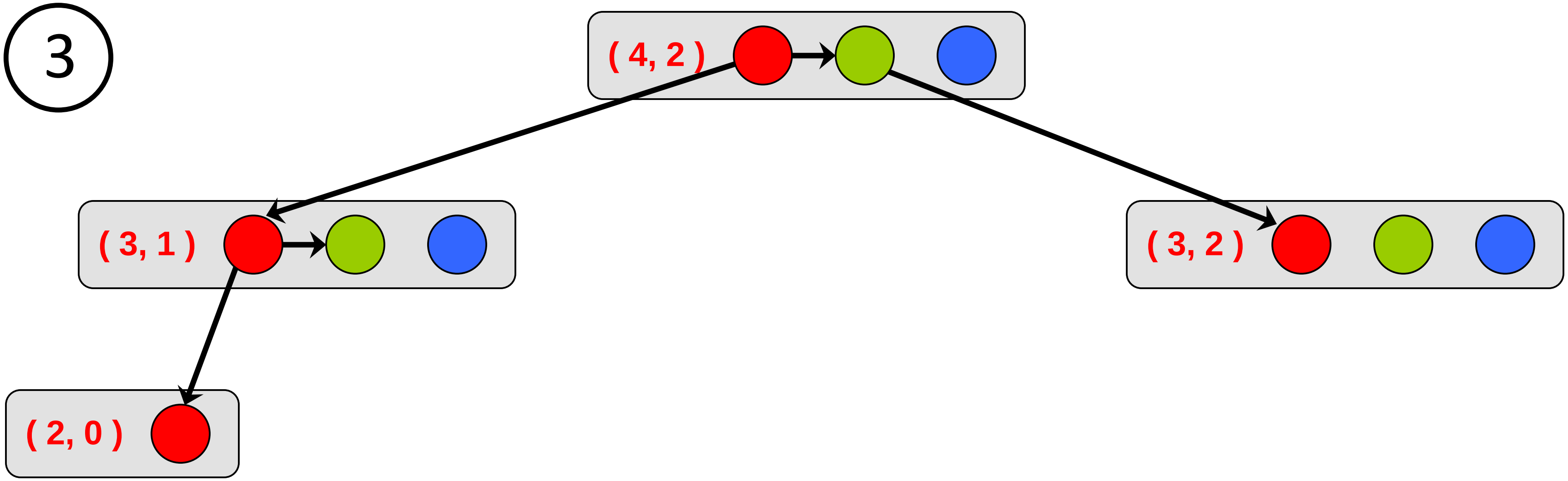
```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r – 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```



Cilk++ Execution Model

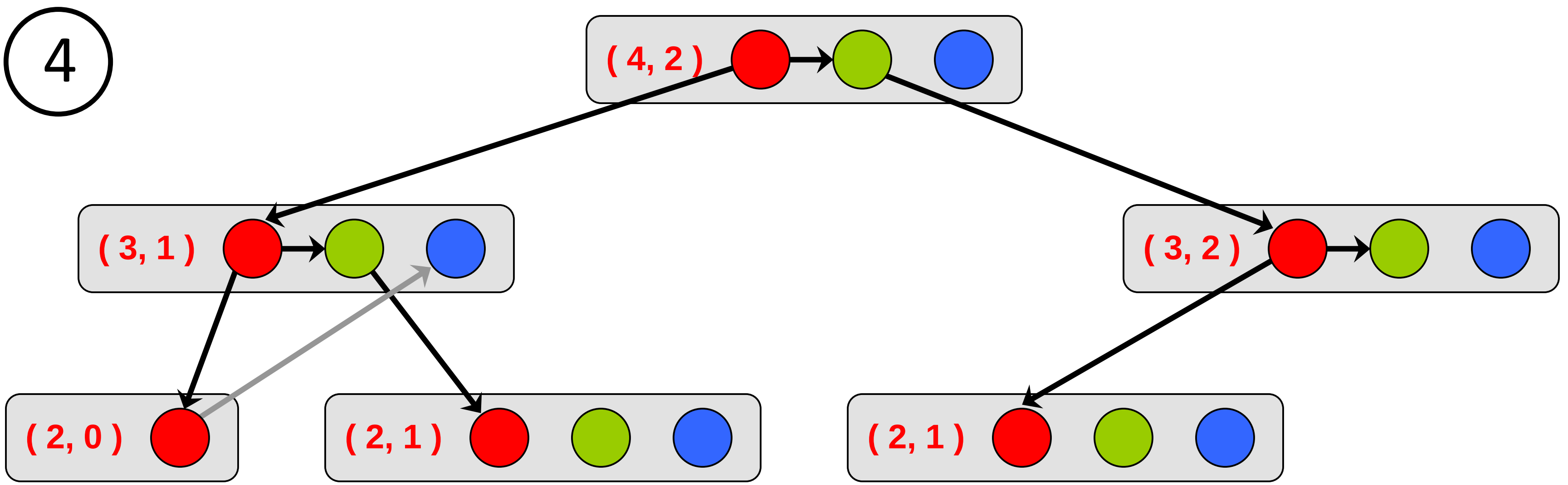
```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```

3



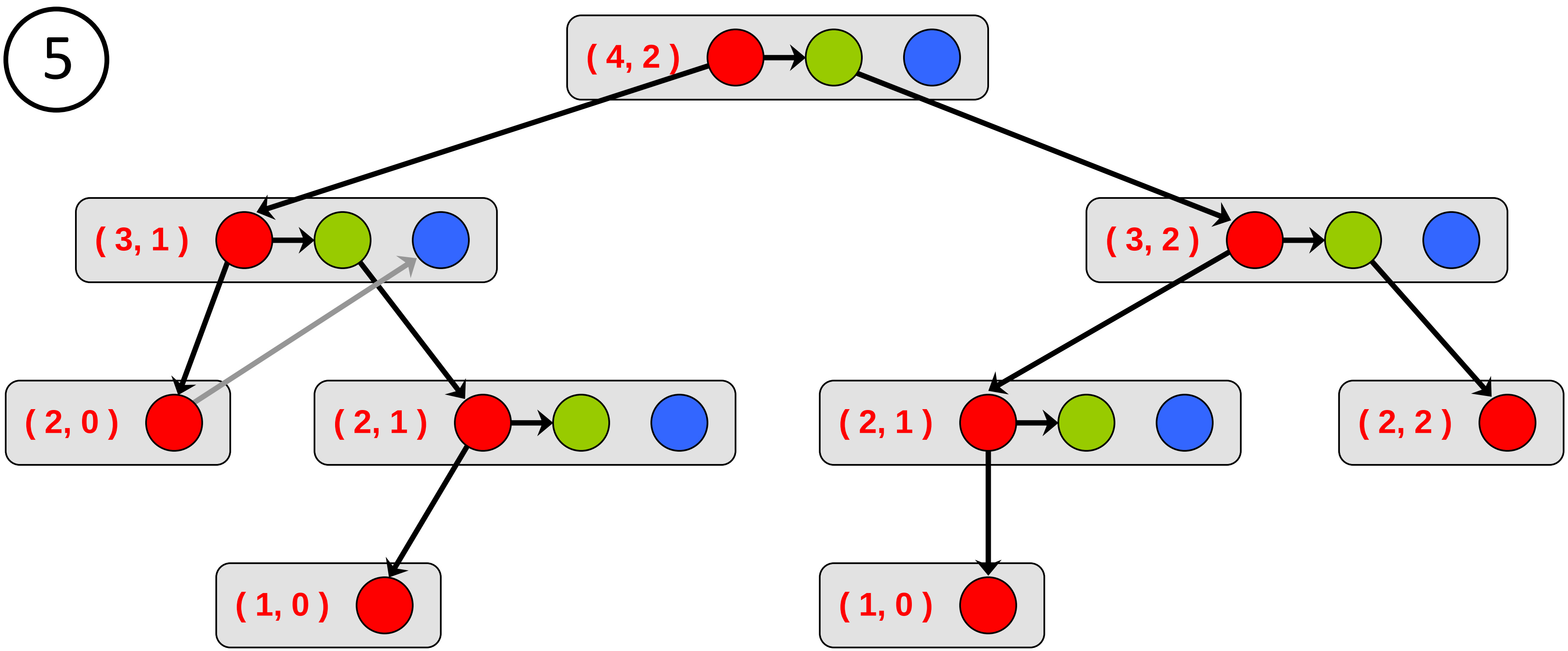
Cilk++ Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r – 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```



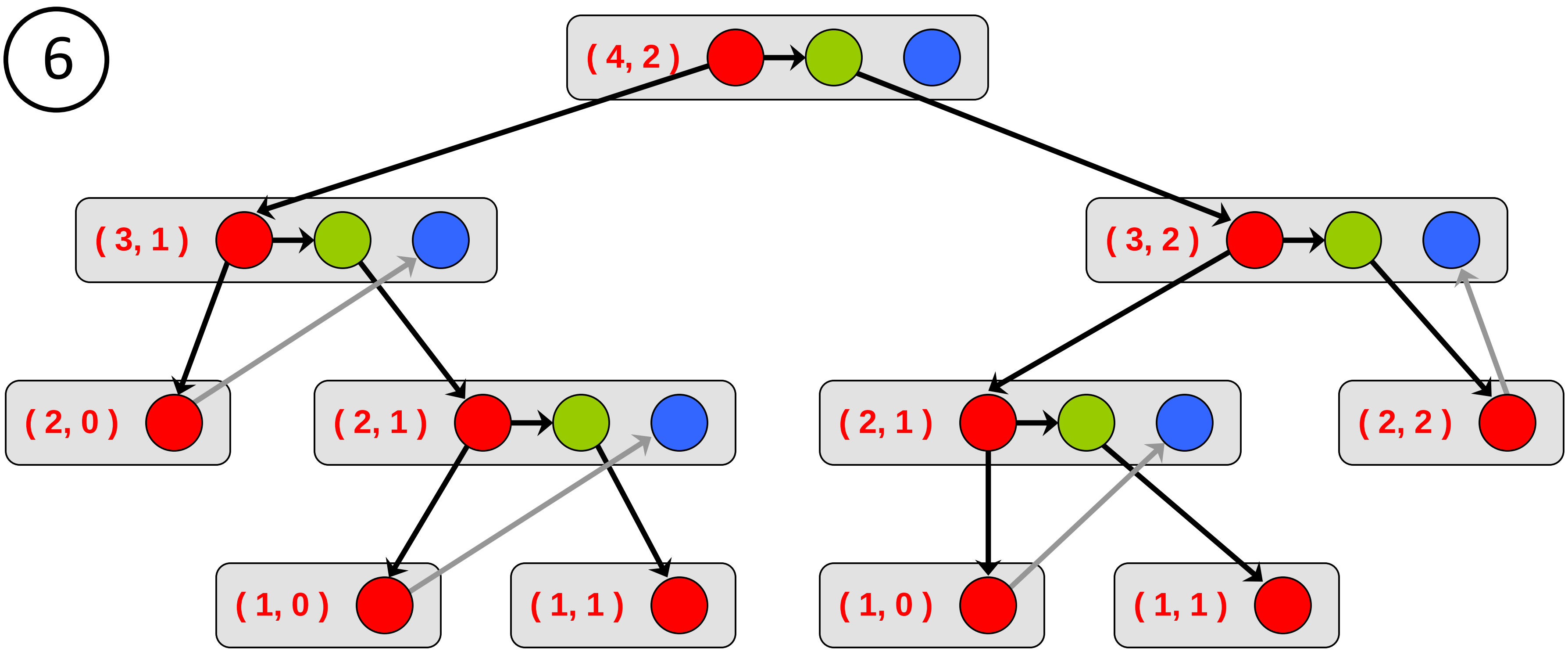
Cilk++ Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r – 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```



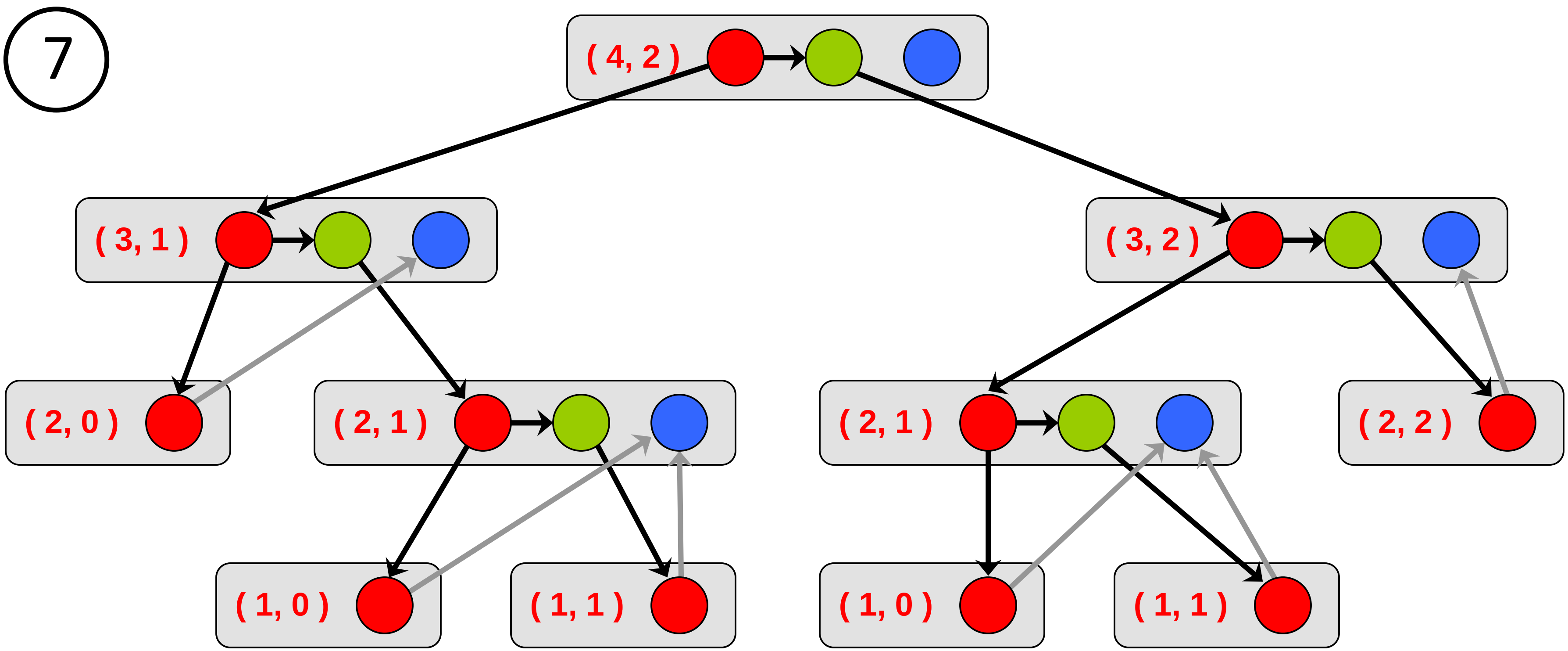
Cilk++ Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```



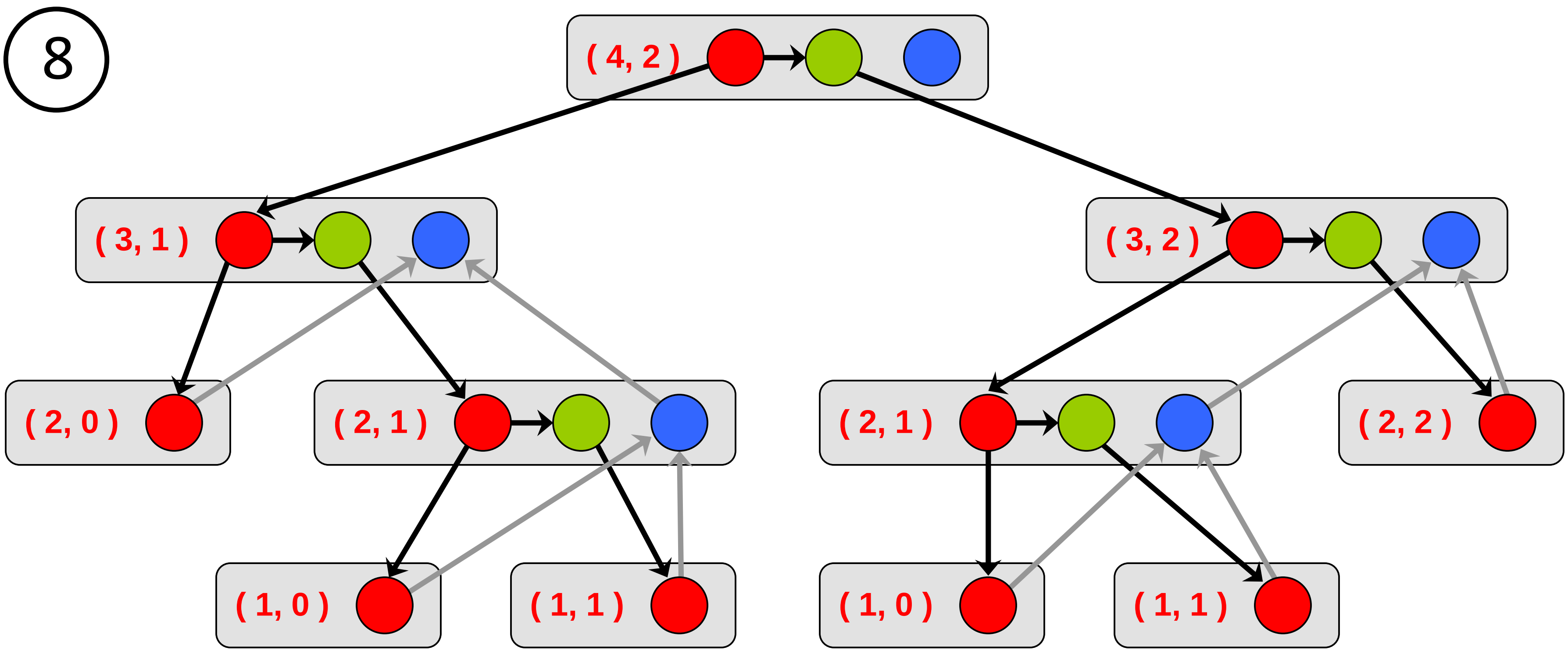
Cilk++ Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r – 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```



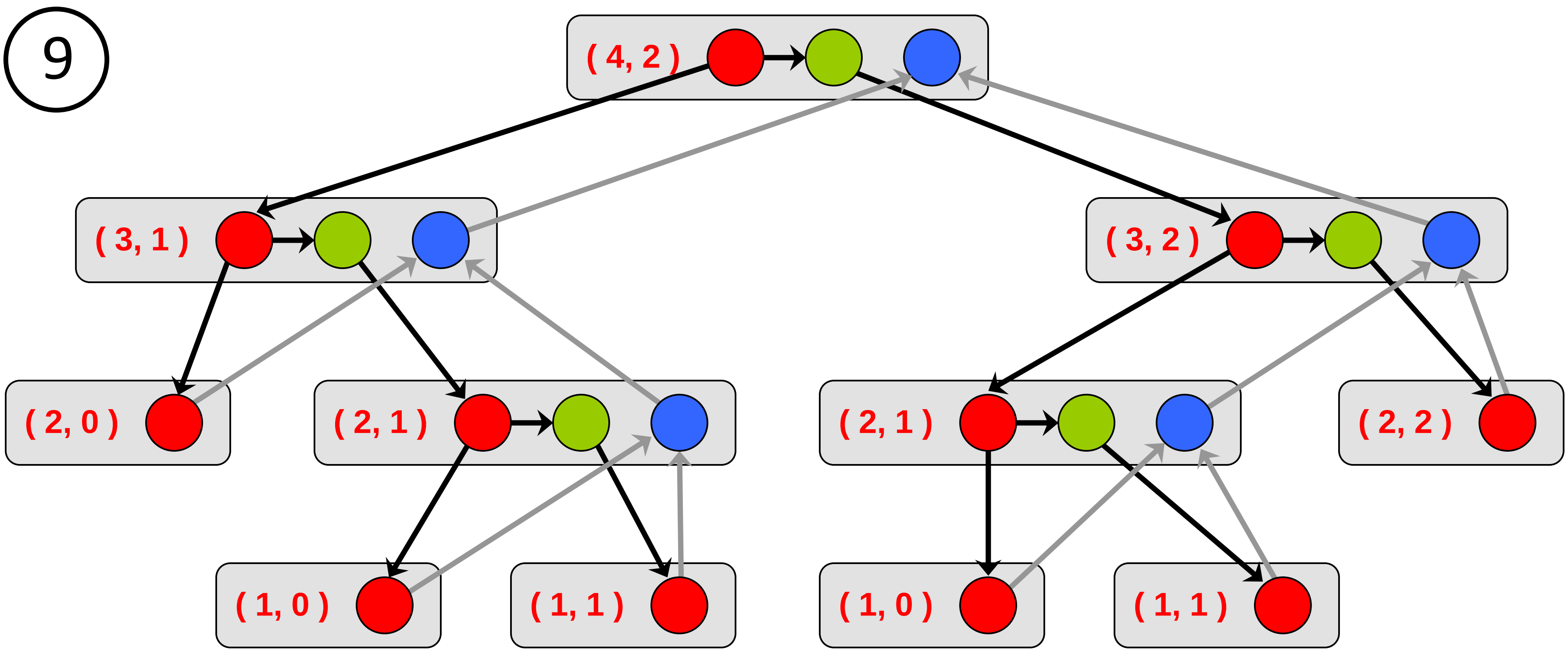
Cilk++ Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n – 1, r - 1 );
  y = comb( n – 1, r );

  cilk_sync;

  return ( x + y );
}
```

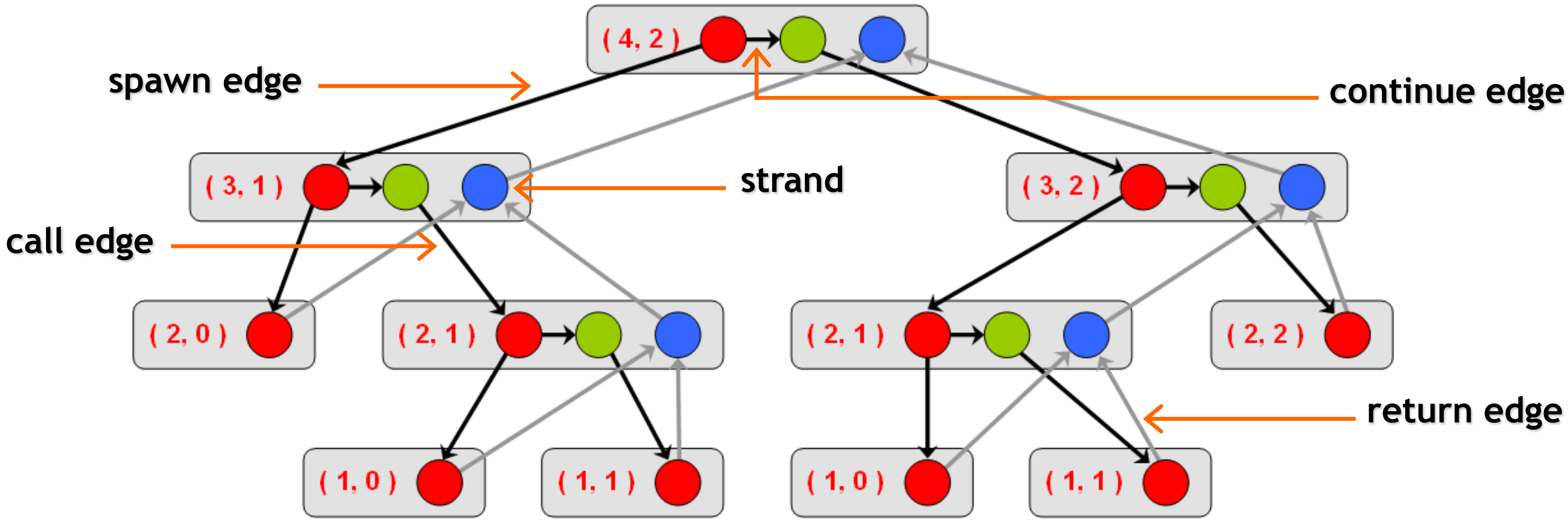


Cilk++ Execution Model

```
int comb ( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

  x = cilk_spawn comb( n - 1, r - 1 );
  y = comb( n - 1, r );

  cilk_sync;

  return ( x + y );
}
```



Computation DAG

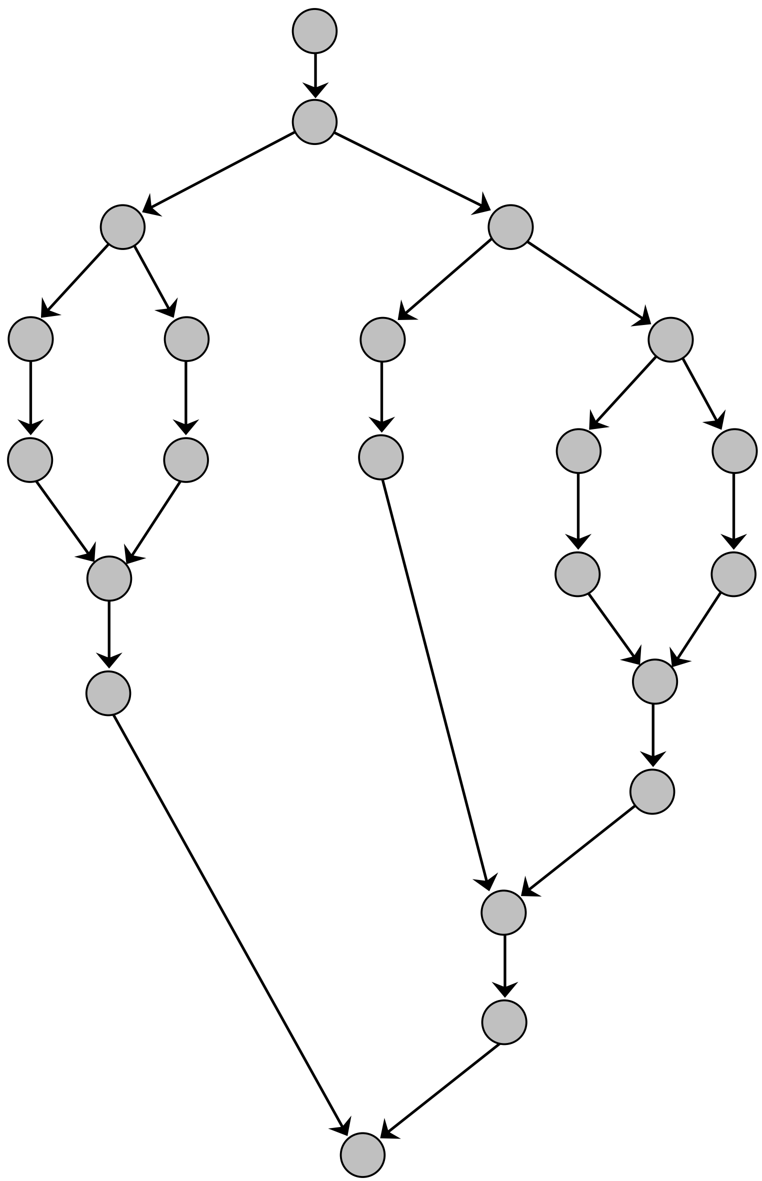


– A parallel instruction stream is represented by a DAG $G = (V, E)$.

– Each vertex $v \in V$ is a *strand* which is a sequence of instructions without a spawn, call, return or exception.

– Each edge $e \in E$ is a *spawn*, *call*, *continue* or *return* edge.

Parallel Performance



T_p = execution time on *p* cores

work = T_1

span = T_∞

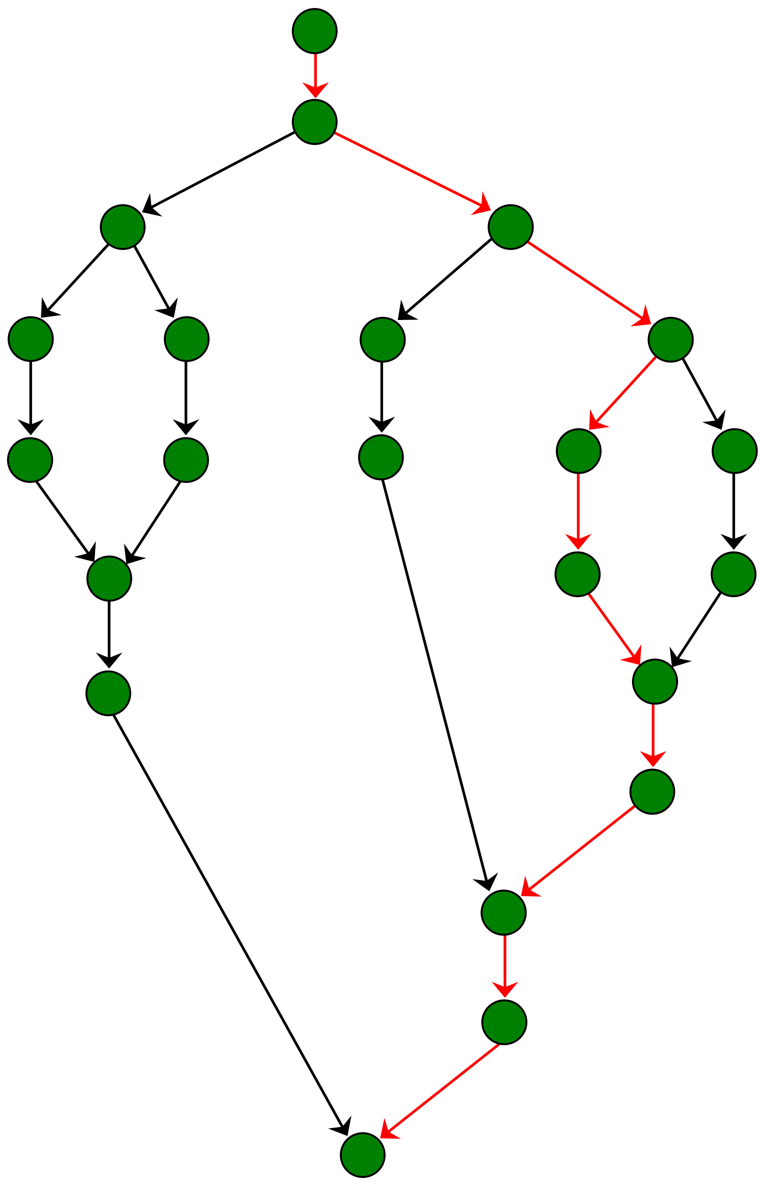
Work Law

$$T_p \geq T_1 / p$$

Span Law

$$T_p \geq T_\infty$$

Speedup & Parallelism



T_p = execution time on p cores

work = T_1

Work Law

$T_p \geq T_1 / p$

span = T_∞

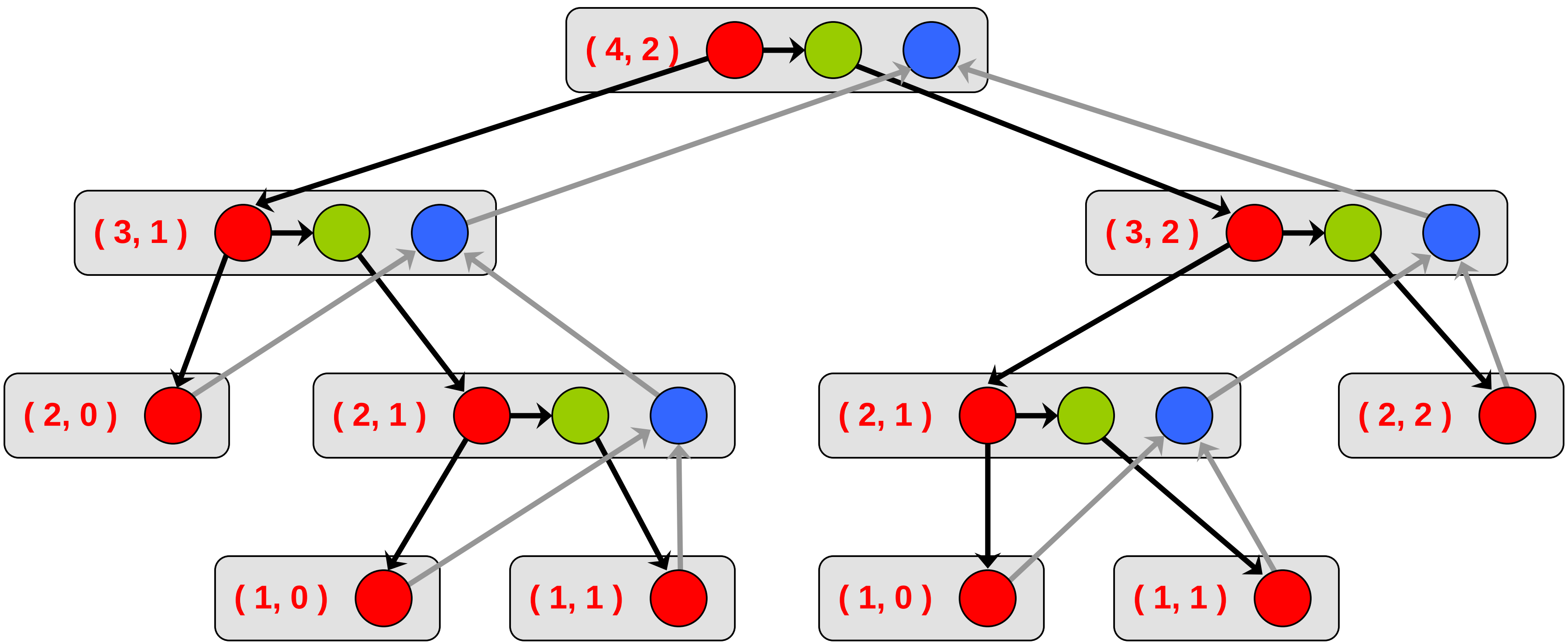
Span Law

$T_p \geq T_\infty$

speedup = T_1 / T_p

parallelism = T_1 / T_∞

Parallelism in comb(4, 2)



Only marginal performance gains with more than 2 cores!

work: $T_1 = 21$

span: $T_\infty = 9$

parallelism $= T_1 / T_\infty = 21 / 9 \approx 2.33$

Implementation of Parallel Loops in Cilk++

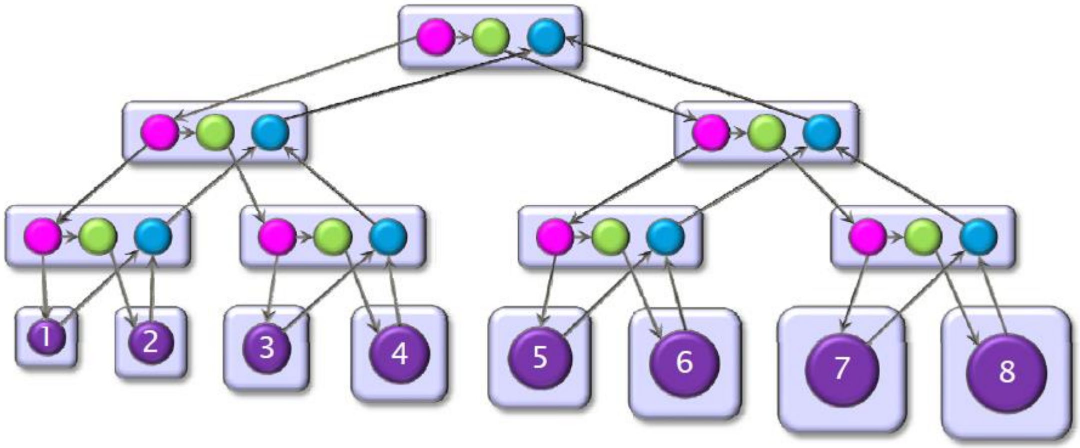
```
cilk_for ( int i = s; i < t; ++i )
      BODY( i );
```

divide-and-conquer implementation

```
void recur( int lo, int hi )
{
  if ( hi - lo > GRAINSIZE )
    {
      int mid = lo + ( hi - lo ) / 2;
      cilk_spawn recur( lo, mid );
      recur( mid, hi );
    }
  else
    {
      for ( int i = lo; i < hi; ++i )
         BODY( i );
    }
}

recur( s, t );
```

Analysis of Parallel Loops

```
cilk_for ( int i = 1; i < n; ++i )
   for ( int j = 0; j < i; ++j )
      {
         double t = A[ i ][ j ];
         A[ i ][ j ] = A[ j ][ i ];
         A[ j ][ i ] = t;
      }
```



Source: Charles Leiserson

— Span of loop control $= \Theta(\log n)$

— Maximum span of an iteration $= \Theta(n)$

— Work, $T_1(n) = \Theta(n^2)$

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

— Parallelism $= \frac{T_1(n)}{T_\infty(n)} = \Theta(n)$

Analysis of Parallel Loops

```
cilk_for ( int i = 1; i < n; ++i )
   cilk_for ( int j = 0; j < i; ++j )
      {
        double t = A[ i ][ j ];
        A[ i ][ j ] = A[ j ][ i ];
        A[ j ][ i ] = t;
      }
```

— Span of outer loop control $= \Theta(\log n)$

— Maximum span of inner loop control $= \Theta(\log n)$

— Span of body $= \Theta(1)$

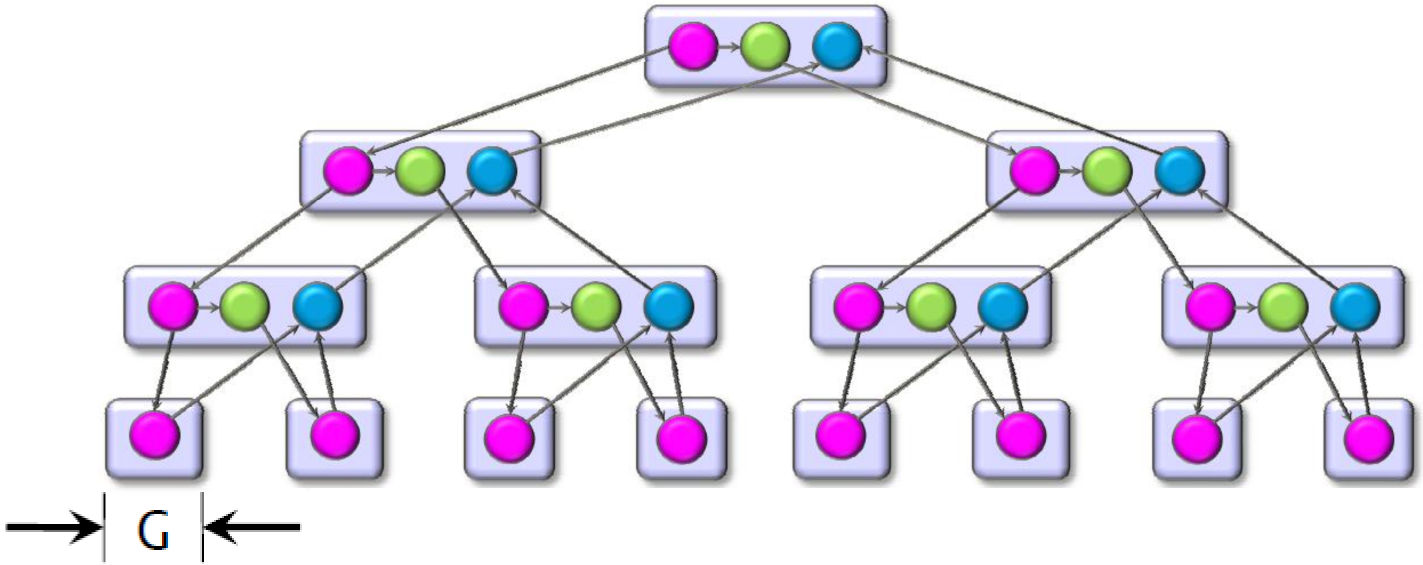
— Work, $T_1(n) = \Theta(n^2)$

— Span, $T_\infty(n) = \Theta(\log n)$

— Parallelism $= \frac{T_1(n)}{T_\infty(n)} = \Theta\left(\frac{n^2}{\log n}\right)$

Analysis of Parallel Loops

```
#pragma cilk_grainsize = G
cilk_for ( int i = 0; i < n; ++i )
      A[ i ] += B[ i ];
```



Source: Charles Leiserson

— Work, $T_1(n) = n \cdot t_{iter} + \frac{n}{G} \cdot t_{spawn}$

— Span, $T_\infty(n) = G \cdot t_{iter} + \log\left(\frac{n}{G}\right) \cdot t_{spawn}$

— Parallelism $= \frac{T_1(n)}{T_\infty(n)} = \frac{n}{G} \cdot \frac{1+\frac{r}{G}}{1+\frac{r}{G}\cdot\log\left(\frac{n}{G}\right)}$, where, $r = \frac{t_{spawn}}{t_{iter}}$

Implementation of Parallel Loops in Cilk++

Default GRAINSIZE: $min\left\{\frac{N}{8p}, 512\right\}$

— p = number of processing elements

— N = number of loop iterations

— Works well for loops that are reasonably balanced

```
void cilk_for_custom_grainsize( int s, int t )
{
   int p = cilk::current_worker_count( );
#pragma cilk_grainsize = ( t - s ) / ( 4 * p )
   cilk_for ( int i = s; i < t; ++i )
      BODY( i );
}
```

Custom GRAINSIZE

— small $\Rightarrow$ high overhead

— large $\Rightarrow$ less parallelism

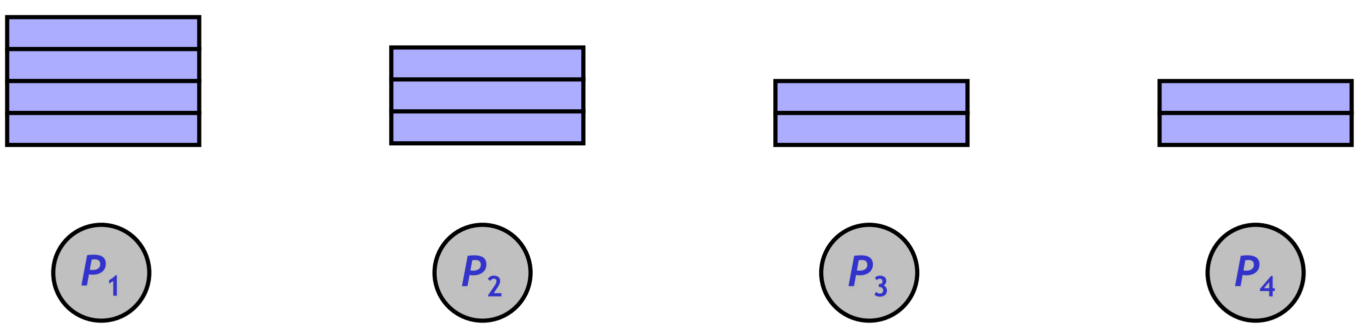
Cilk++'s
Work-Stealing Scheduler

Cilk++'s Work-Stealing Scheduler

- A *randomized distributed* scheduler
- Achieves
  - $T_p = \frac{T_1}{p} + \mathrm{O}(T_\infty)$ time (provably)
  - $T_p \approx \frac{T_1}{p} + T_\infty$ time (empirically)
- Near-perfect linear speedup as long as parallelism, $\frac{T_1}{T_\infty} \gg p$
- Uses at most *p* times the space used by a serial execution
- Has provably good *cache performance*

Cilk++'s Work-Stealing Scheduler

- Each core maintains a *work dqueue* of ready threads
- A core manipulates the bottom of its dqueue like a stack
  - Pops ready threads for execution
  - Pushes new/spawned threads
- Whenever a core runs out of ready threads it *steals* one from the top of the dqueue of a *random* core



The Cilkview
Scalability Analyzer

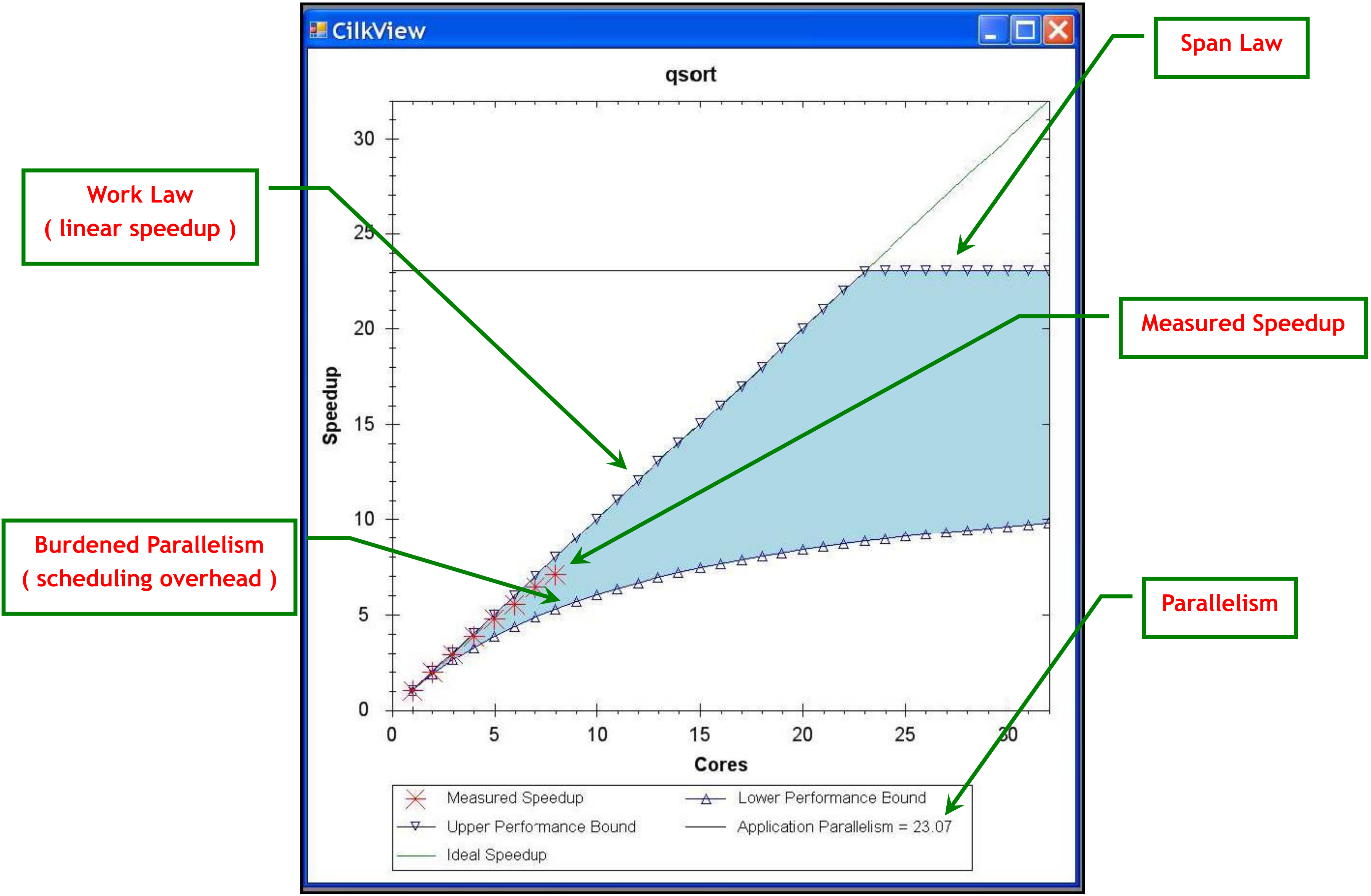
Cilkview Scalability Analyzer

- Measures *work* and *span* using *dynamic instrumentation*.
- Derives *upper bounds* on parallel performance using work and span.
- Estimates *scheduling overhead* to compute a *burdened span* for lower bounds.

Cilkview Scalability Analyzer

```
template < typename T >
void qsort( T p, T r )
{
  if ( p != r )
    {
      T q = partition( p, r, bind2nd( less< typename
                       iterator_traits< T >::value_type >( ), *p ) );
      cilk_spawn qsort( p, q );
      qsort( max( p + 1, q ), r );
      cilk_sync;
    }
}

int cilk_main( )
{
  int n = 10000000;
  double a[ n ];

  cilk::cilkview cv;
  cilk_for ( int i = 0; i < n; i++ )
      a[ i ] = sin( ( double ) i );
  cv.start( );
  qsort( a, a + n );
  cv.stop( );
  cv.dump( ``qsort'' );

  return 0;
}
```

Cilkview Scalability Analyzer



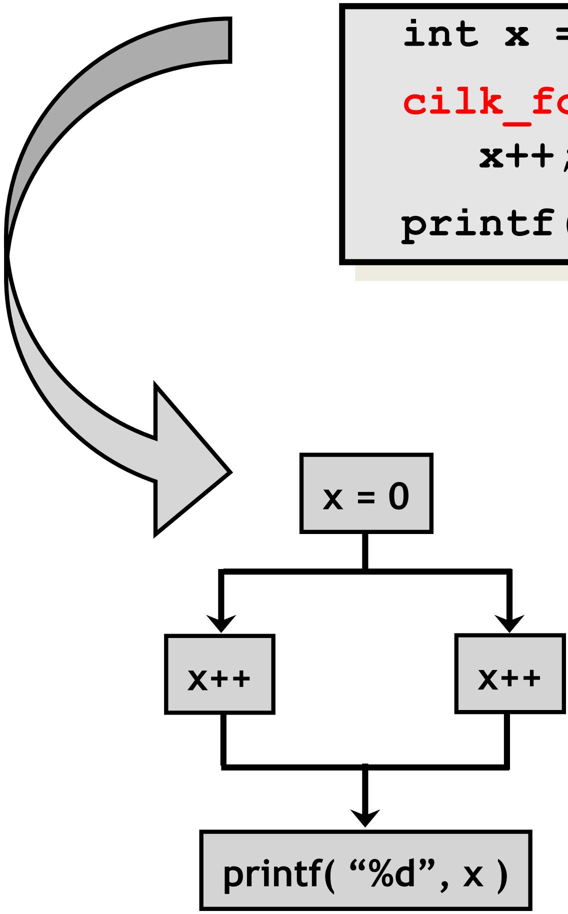
Source: He, Leiserson & Leiserson, 2009

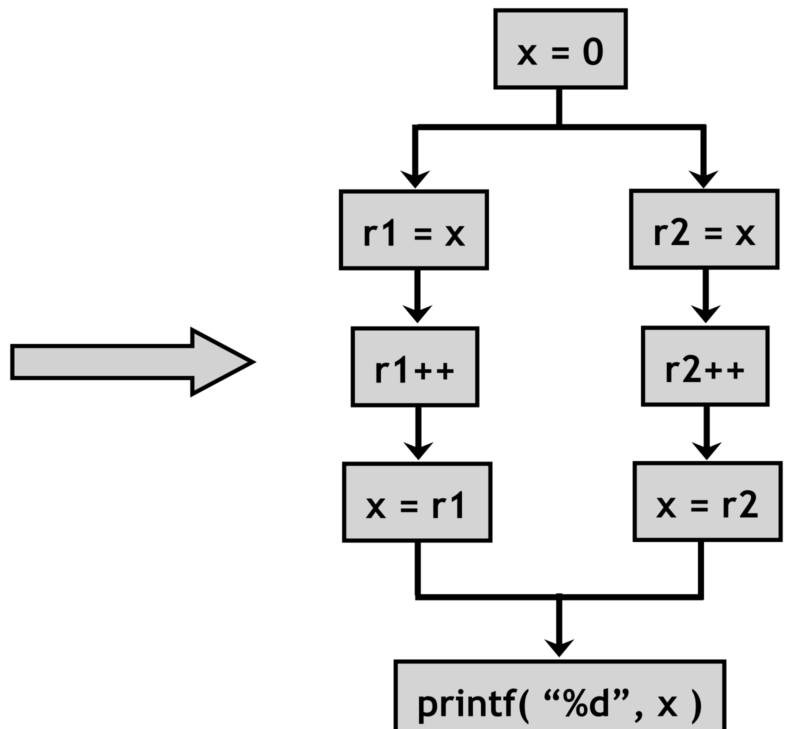
Race Bugs
and
the Cilkscreen Race Detector

Race Bugs

A *determinacy race* occurs if two logically parallel instructions access the same memory location and at least one of them performs a write.

```
int x = 0;
cilk_for ( int i = 0; i < 2; i++ )
   x++;
printf( "%d", x );
```





Critical Sections and Mutexes

race

```
int r = 0;

cilk_for ( int i = 0; i < n; i++ )
    r += eval( x[ i ] );
```

```
cilk::mutex mtx;

cilk_for ( int i = 0; i < n; i++ )
    mtx.lock( );
    r += eval( x[ i ] );
    mtx.unlock( );
```

critical section

two or more strands must not access at the same time

mutex (mutual exclusion)

an attempt by a strand to lock an already locked mutex causes that strand to block (i.e., wait) until the mutex is unlocked

Problems

- lock overhead
- lock contention

Critical Sections and Mutexes

race

```
int r = 0;

cilk_for ( int i = 0; i < n; i++ )
    r += eval( x[ i ] );
```

```
cilk::mutex mtx;

cilk_for ( int i = 0; i < n; i++ )
    mtx.lock( );
    r += eval( x[ i ] );
    mtx.unlock( );
```

```
cilk::mutex mtx;

cilk_for ( int i = 0; i < n; i++ )
    int y = eval( x[ i ] );
    mtx.lock( );
    r += y;
    mtx.unlock( );
```

- slightly better solution
- but lock contention can still destroy parallelism

Cilkscreen Race Detector

- If determinacy data races exist in an ostensibly deterministic program (e.g., a program with no mutexes), *Cilkscreen* guarantees to find such a race.
- Uses *regression tests* on user-provided test inputs
- *Reports* filenames, line and variables involved in races as well as stack traces.
- Runs the binary executable using *dynamic instrumentation*.
- Runs about 20 times *slower* than real-time.

Race Bugs and the Cilk++ Reducers

Race Bugs and Cilk++ Reducer Hyperobjects

- Cilk++ provides *reducer hyperobjects* to mitigate data races on nonlocal variables without locks and code restructuring
- A variable *x* can be declared a Cilk++ *reducer* over an *associative* operation such as addition, list concatenation etc.
- Strands can update *x* as if it were an ordinary local variable, but *x* is, in fact, maintained as a collection of different *views*.
- Clik++ runtime system coordinates the views and combines them when appropriate.

a summing reducer over int

updates are resolved automatically without races or contention

```
cilk::reducer_opadd< int > r;
cilk_for ( int i = 0; i < n; i++ )
   r += eval( x[ i ] );
cout << r.get_value( );
```

at the end the final int value can be extracted

Race Bugs and Cilk++ Reducer Hyperobjects

original

```
x = 0;
x += 2;
x++;
x += 3;
x += 4;
x += 7;
x += 5;
x += 4;
x += 2;
x++;
x += 6;
x += 9;
x += 3;
x++;
x += 8;
```

raceless parallel execution

equivalent

```
x1 = 0;
x1 += 2;
x1++;
x1 += 3;
x1 += 4;
x1 += 7;
x1 += 5;
x1 += 4;
```

```
x2 = 0;
x2 += 2;
x2++;
x2 += 6;
x2 += 9;
x2 += 3;
x2++;
x2 += 8;
```

```
x = x1 + x2;
```

equivalent

```
x1 = 0;
x1 += 2;
x1++;
x1 += 3;
x1 += 4;
```

```
x2 = 0;
x2 += 7;
x2 += 5;
x2 += 4;
x2 += 2;
x2++;
```

```
x3 = 0;
x3 += 6;
x3 += 9;
x3 += 3;
x3++;
x3 += 8;
```

```
x = x1 + x2 + x3;
```

raceless parallel execution

If you do not need to look at intermediate values the result is *determinate* because addition is *associative*.

Cilk++ Reducer Library

- Many commonly used reducers
  - reducer_list_append
  - reducer_list_prepend
  - reducer_max
  - reducer_max_index
  - reducer_min
  - reducer_min_index
  - reducer_opadd
  - reducer_ostream
  - reducer_basic_string
  - …
- One can also make one’s own reducers using cilk::monoid_base and cilk::reducer

Some Concluding Remarks

Cilk++ seems to have several major advantages

- very easy to use (compared to DIY platforms like pthreads)
- portable code (e.g., core-/processor-oblivious)
- produces efficient executables
  (efficient scheduler, cache-efficiency)
- useful toolkit (cilkview, cilkscreen)