Cilk and Cilk++
Lecture 1

- Language extensions, rather than entirely new languages
- Assume you already know C and/or C++

- Three parallelism keywords

- Execution model (multithreading)
- A few small examples
- How to use on the cs-parallel server
- Race conditions

Cilk (and Cilk++) Concurrency Platform

- Supports *dynamic multithreading*
- Includes a small set of *linguistic extensions* to C/C++ to support *fork-join* parallelism
- Based on multithreaded language technology developed at MIT and MIT spin-off *CilkArts* (acquired by *Intel* in 2009)
- Includes
  - *A provably efficient scheduler*
  - *Hyperobject library for parallelizing code with global variables*
  - *Race detector (Cilkscreen) - we will not cover this*
  - *Scalability analyzer (Cilkview) - we will not cover this*

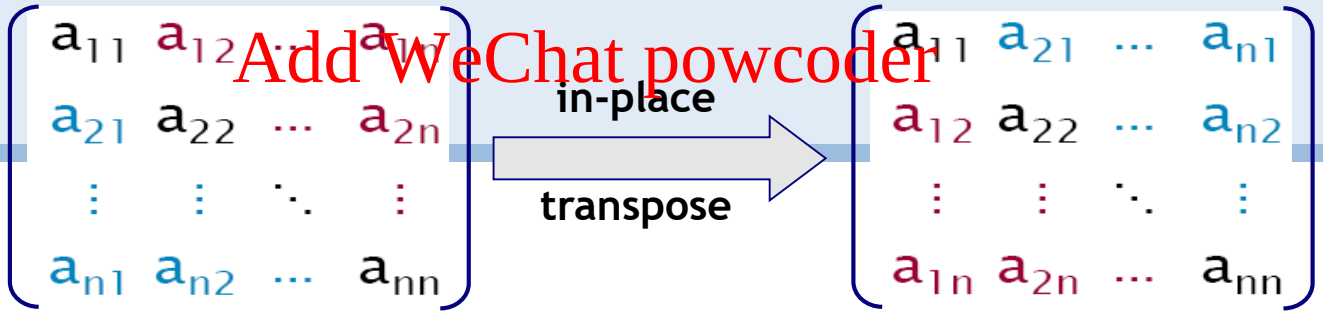
Serial to Parallel using Three Keywords

cilk_for : all iterations of this for-loop may be performed in parallel using multiple threads

cilk_spawn : fork a new thread to run in parallel with the current thread

cilk_sync : all spawned threads synchronize here, then continue sequentially in a single thread

Loop Parallelism




```
for ( i = 2; i <= n; ++i )
   for ( j = 1; j <= i-1; ++j )
      {
        double t = A[ i ][ j ];
        A[ i ][ j ] = A[ j ][ i ];
        A[ j ][ i ] = t;
      }
```


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

Serial code

```
cilk_for ( i = 2; i <= n; ++i )
   cilk_for ( j = 1; j <= i-1; ++j )
      {
        double t = A[ i ][ j ];
        A[ i ][ j ] = A[ j ][ i ];
        A[ j ][ i ] = t;
      }
```

Multithreaded code

Nested Parallelism

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

```
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 code

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

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


```
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 );
}
```

Function return enforces implicit synchronization.

Oblivious of the number of cores / processors!

Multithreaded code

Cilk Execution Model

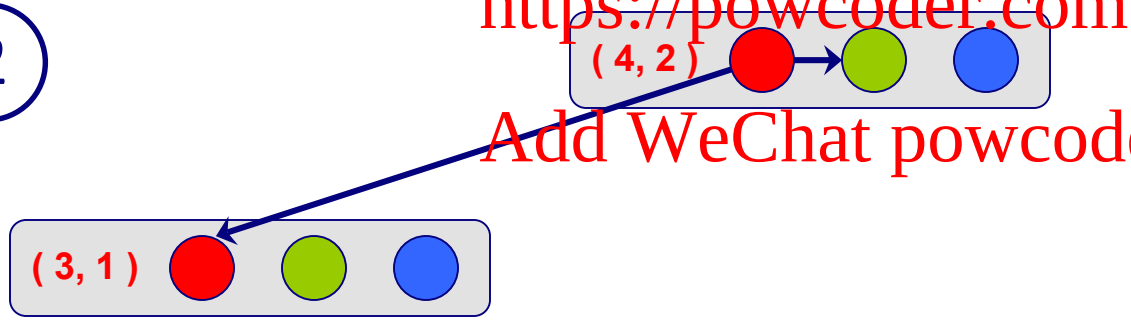

```
int comb( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  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

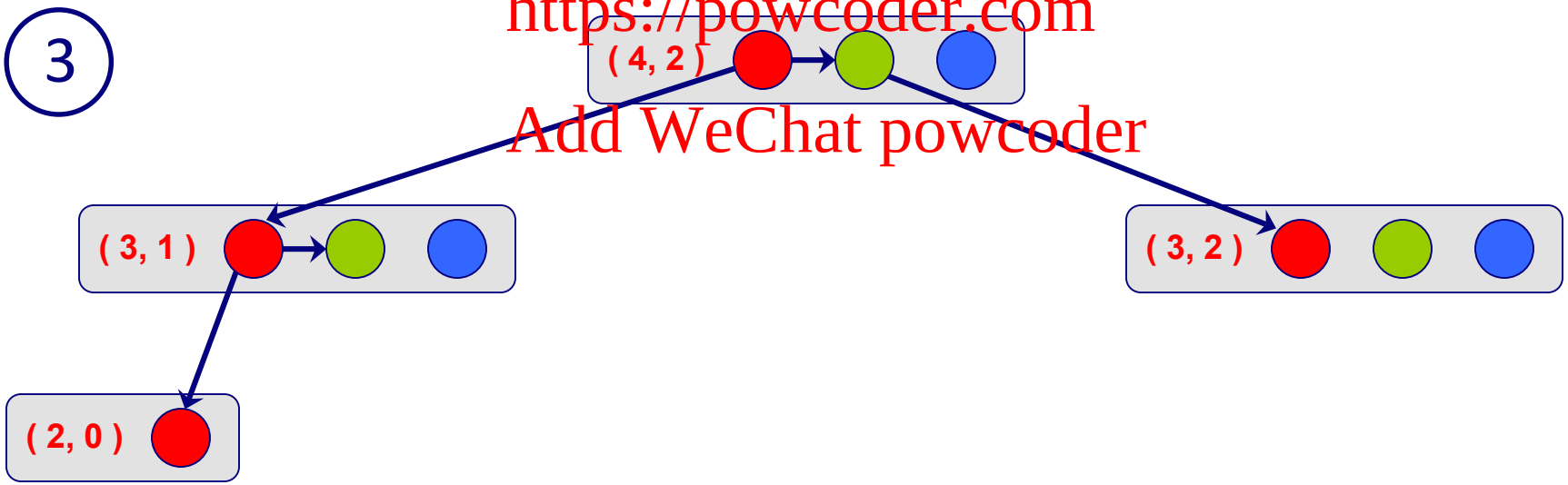

```
int comb( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  int x, y;

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

  cilk_sync;

  return ( x + y );
}
```


2



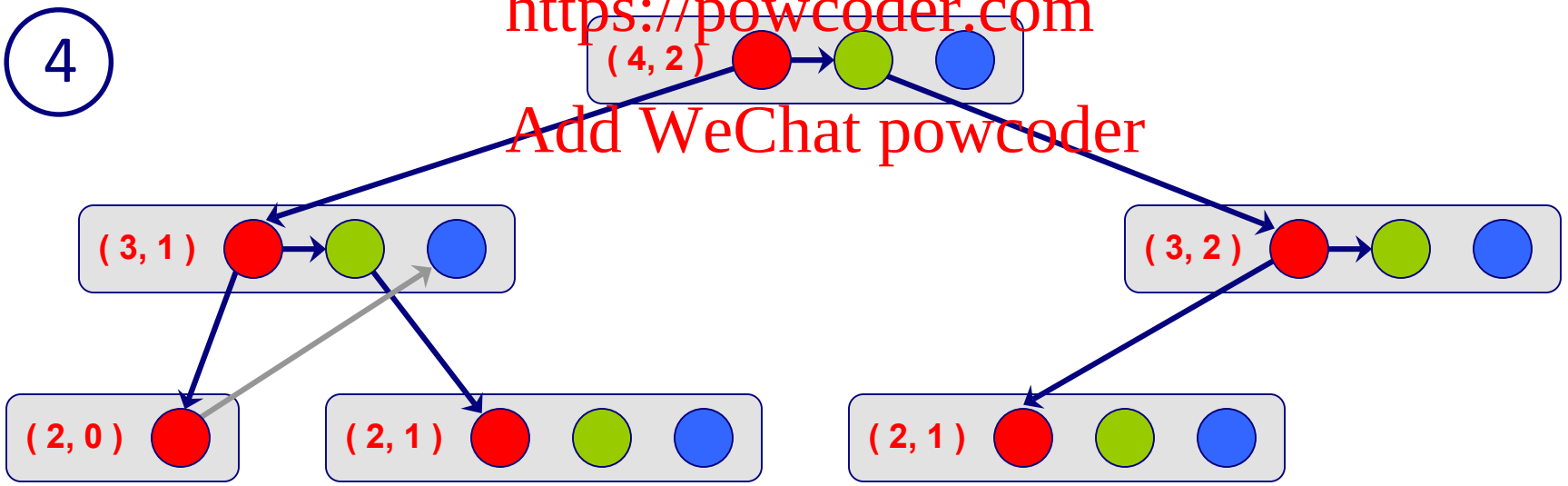
Cilk Execution Model


```
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

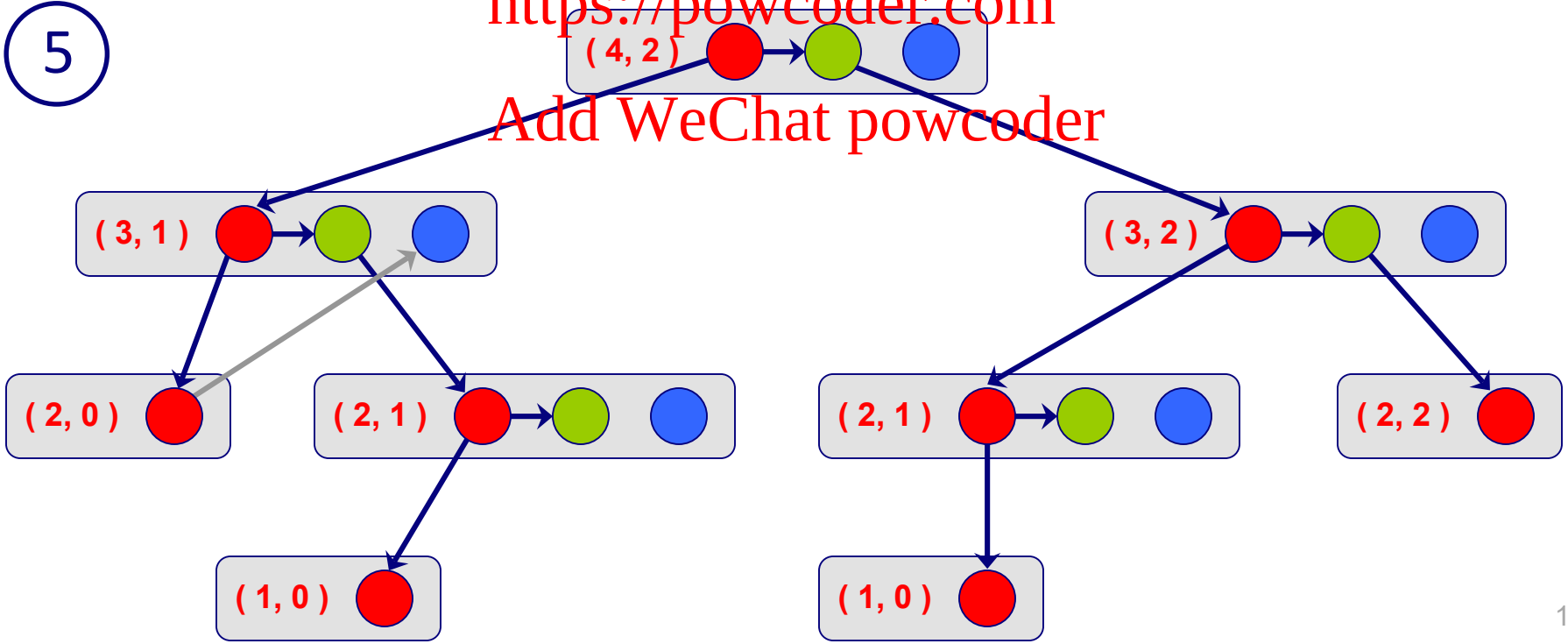

```
int comb( int n, int r )
{
  if ( r > n ) return 0;
  if ( r == 0 || r == n ) return 1;

  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 );
}
```

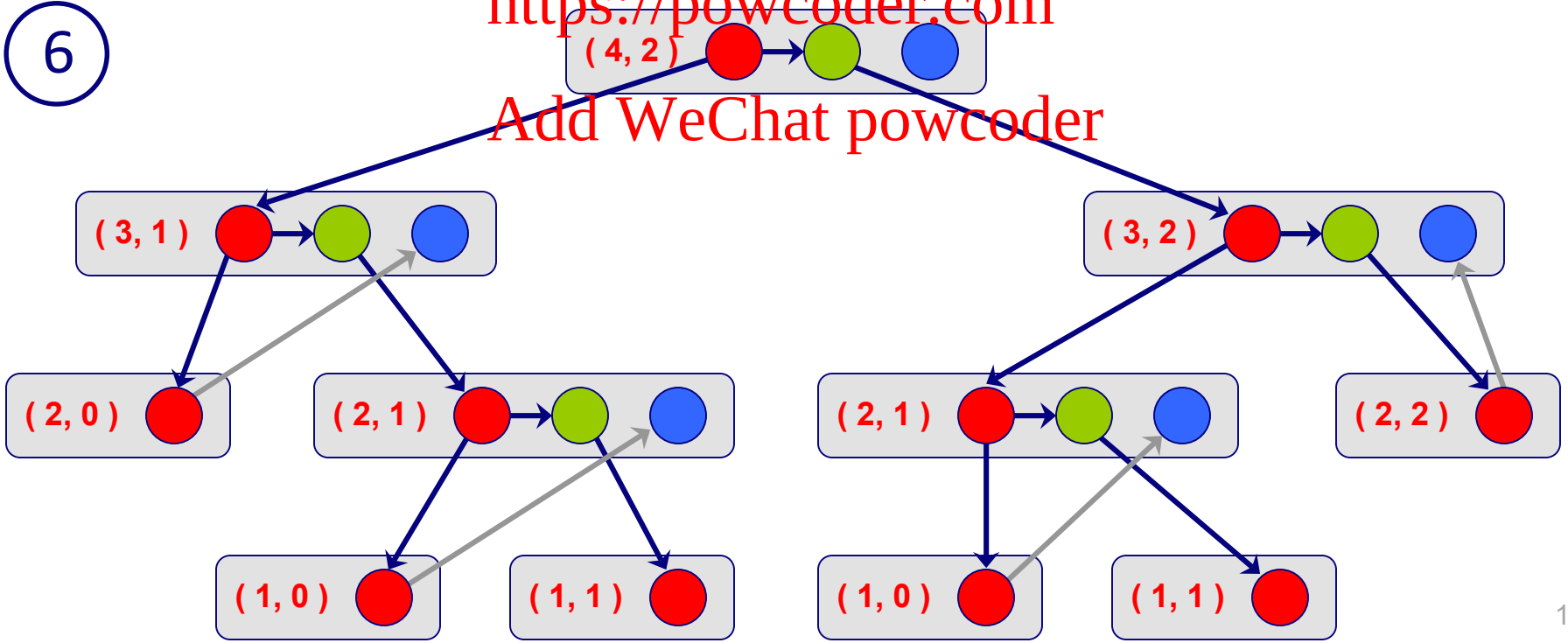

5



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 );
}
```

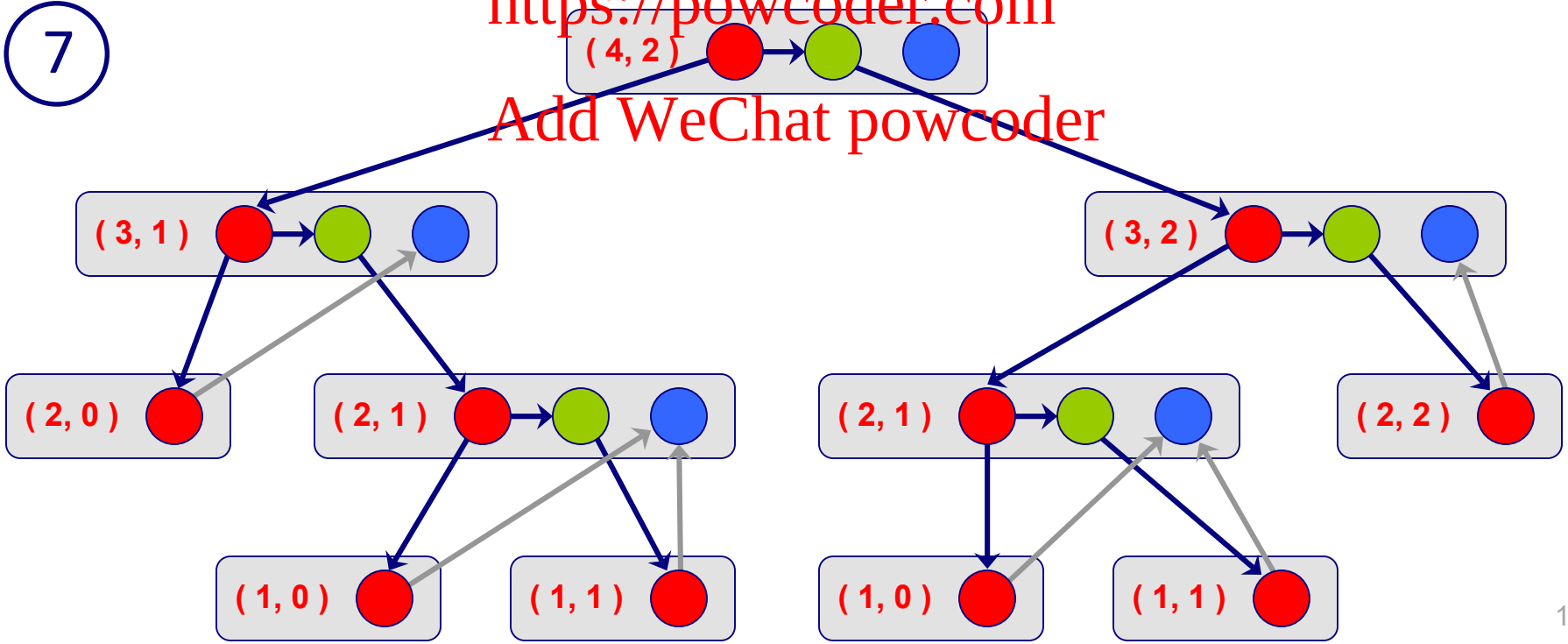

6



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 );
}
```

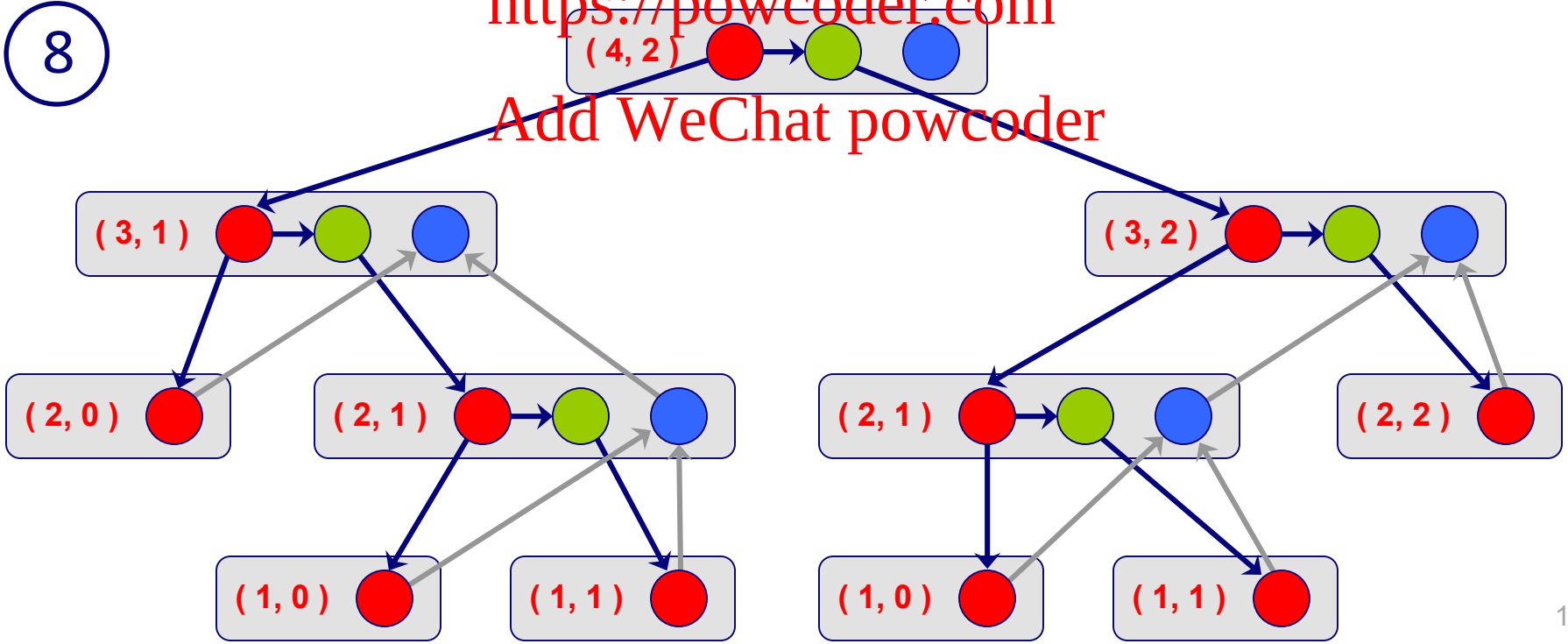

7



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 );
}
```

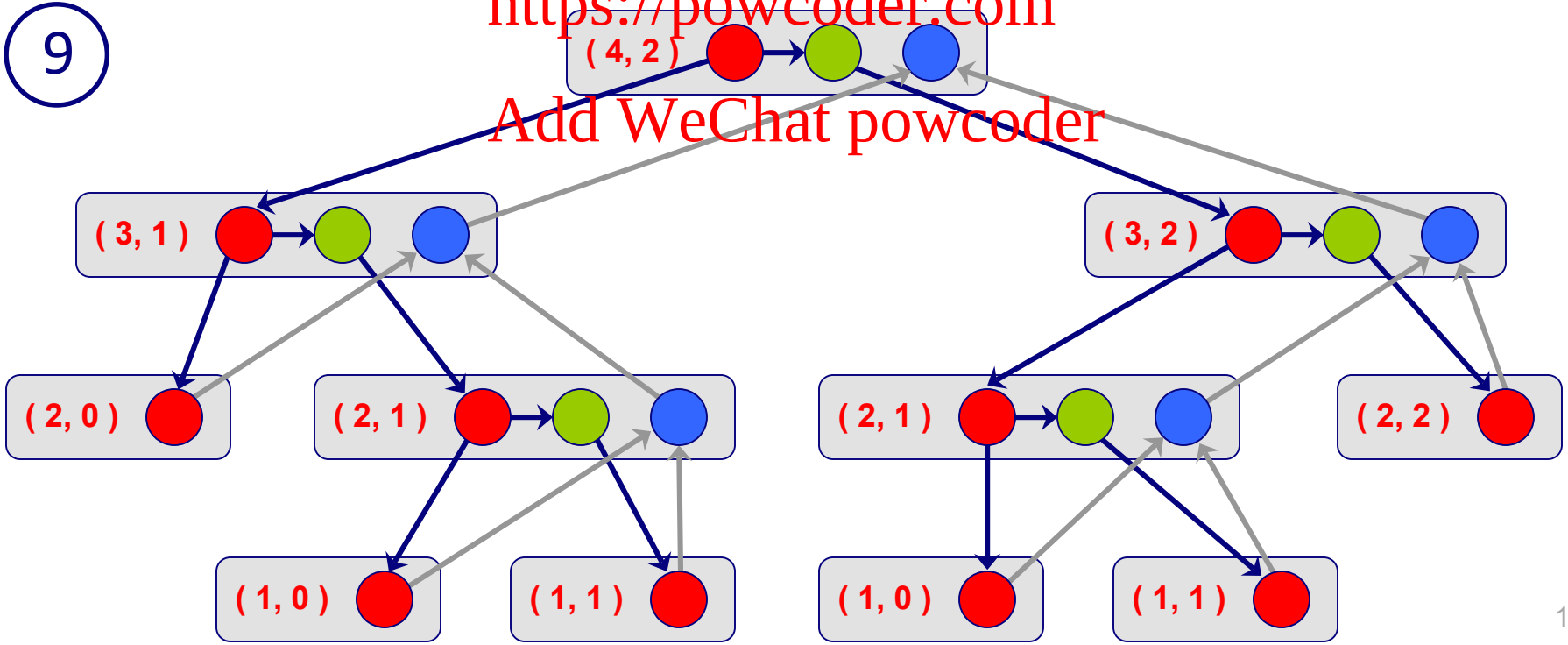

8



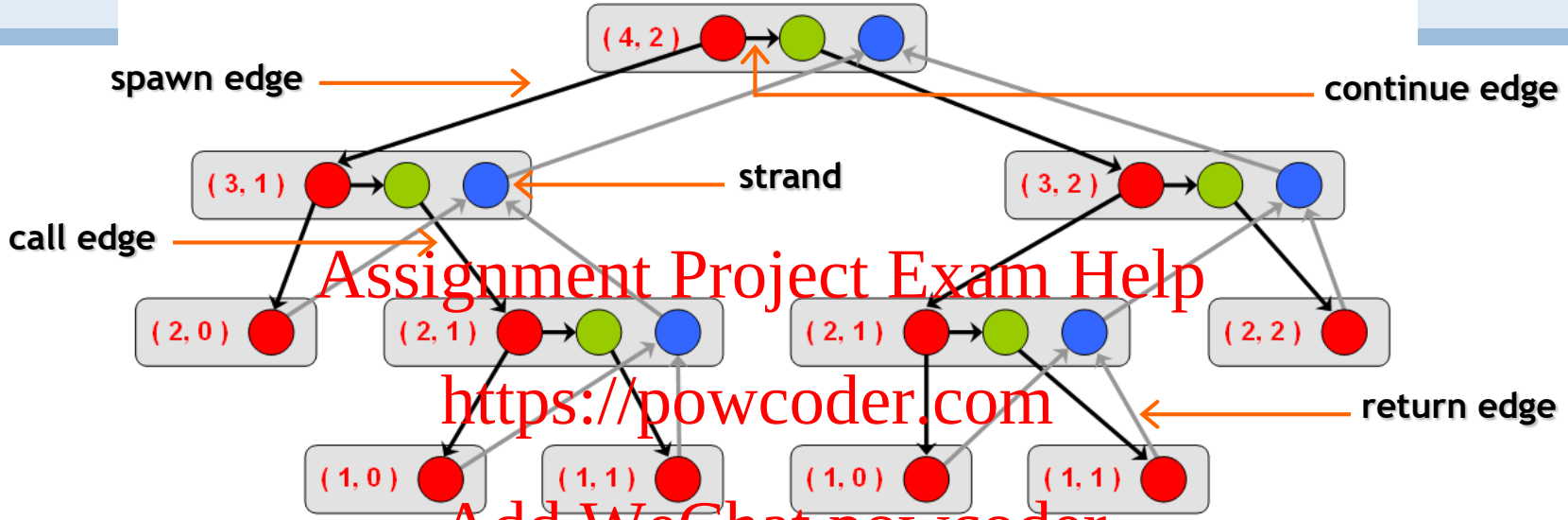
Cilk Execution Model


```
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 );
}
```


9



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.

cilk_for.c


```
#include <stdio.h>
#include <stdlib.h>
#include <cilk/cilk.h>

int main (int argc, char *argv[ ]) {
        int num=10;
        if (argc>1) num=atoi(argv[1]);
        cilk_for (int k=0; k<num; k++) {
                printf ("thread %d\n", k);
        }
        return 0;
}
```

How to compile and run Cilk on cs-parallel


```
> cilk  cilk_for.c  –o  cilk_for
> ./cilk_for
```



```
thread 0
thread 5
thread 7
thread 8
thread 3
thread 9
thread 6
thread 1
thread 4
thread 2
```

Non-deterministic behavior


```
> ./cilk_for  10
thread 1
thread 0
thread 5
thread 6
thread 8
thread 7
thread 3
thread 4
thread 2
thread 9
```

cilk_for.cpp


```
#include <iostream>
#include <sstream>
#include <cstdlib>
#include <cilk/cilk.h>
using namespace std;

int main (int argc, char *argv[ ]) {
        int num=10;
        if (argc>1) num=atoi(argv[1]);
        cilk_for (int k=0; k<num; k++) {
                ostringstream oss;
                oss << "thread " << k << endl;
                cout << oss.str( );
        }
}
```

How to compile and run Cilk++ on cs-parallel


```
> cilk++  cilk_for.cpp  –o  cilk_for
> ./cilk_for  20
```



```
thread 0
thread 5
thread 10
thread 7
thread 6
thread 11
thread 1
thread 15
thread 17
thread 8
thread 16
thread 19
thread 2
thread 3
thread 18
thread 12
thread 9
thread 13
thread 4
thread 14
```

comb.c


```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <cilk/cilk.h>
#include <cilk/cilk_api.h>

int comb (int n, int r) {
        if (r<0 || r>n) return 0;
        if (r==0 || r==n) return 1;
        int x = cilk_spawn comb (n-1, r-1);
        int y = comb (n-1, r);
        cilk_sync;
        return x + y;
}
```

comb.c (continued)


```
int main (int argc, char* argv[ ] ) {
        int n = argc==1 ? 30 : atoi(argv[1]);
        int r = argc<=2 ? (n/2) : atoi(argv[2]);
        clock_t start = clock( );
        int result = comb (n, r);
        clock_t finish = clock( );
        double duration = (double)(finish-start) / CLOCKS_PER_SEC;
        printf ("C(%d,%d) = %d\n", n, r, result);
        printf ("Calculated in %lf seconds using %d workers\n",
                                duration, __cilkrts_get_nworkers( ));
        return 0;
}
```

How to compile and run Cilk on cs-parallel


```
> cilk  comb.c  –o  comb
> ./comb  30  15
> ./comb  30
> ./comb
```


C(30,15) = 155117520

Calculated in 12.139764 seconds using 24 workers

Actually just takes about 0.5 seconds in real time

comb.cpp


```
#include <iostream>
#include <cstdlib>
#include <time.h>
#include <cilk/cilk.h>
#include <cilk/cilk_api.h>
using namespace std;

int comb (int n, int r) {
        if (r<0 || r>n) return 0;
        if (r==0 || r==n) return 1;
        int x = cilk_spawn comb (n-1, r-1);
        int y = comb (n-1, r);
        cilk_sync;
        return x + y;
}
```

comb.cpp


```
int main (int argc, char* argv[ ] ) {
        int n = argc==1 ? 30 : atoi(argv[1]);
        int r = argc<=2 ? (n/2) : atoi(argv[2]);
        clock_t start = clock( );
        int result = comb (n, r);
        clock_t finish = clock( );
        double duration = (double)(finish-start) / CLOCKS_PER_SEC;
        cout << "C(" << n << "," << r << ") = " << result << endl;
        cout << "Calculated in " << duration << " seconds using "
                << __cilkrts_get_nworkers( ) << " workers" << endl;
        return 0;
}
```

How to compile and run Cilk++ on cs-parallel


```
> cilk++  comb.cpp  –o  comb
> ./comb  30  15
> ./comb  30
> ./comb
```


C(30,15) = 155117520

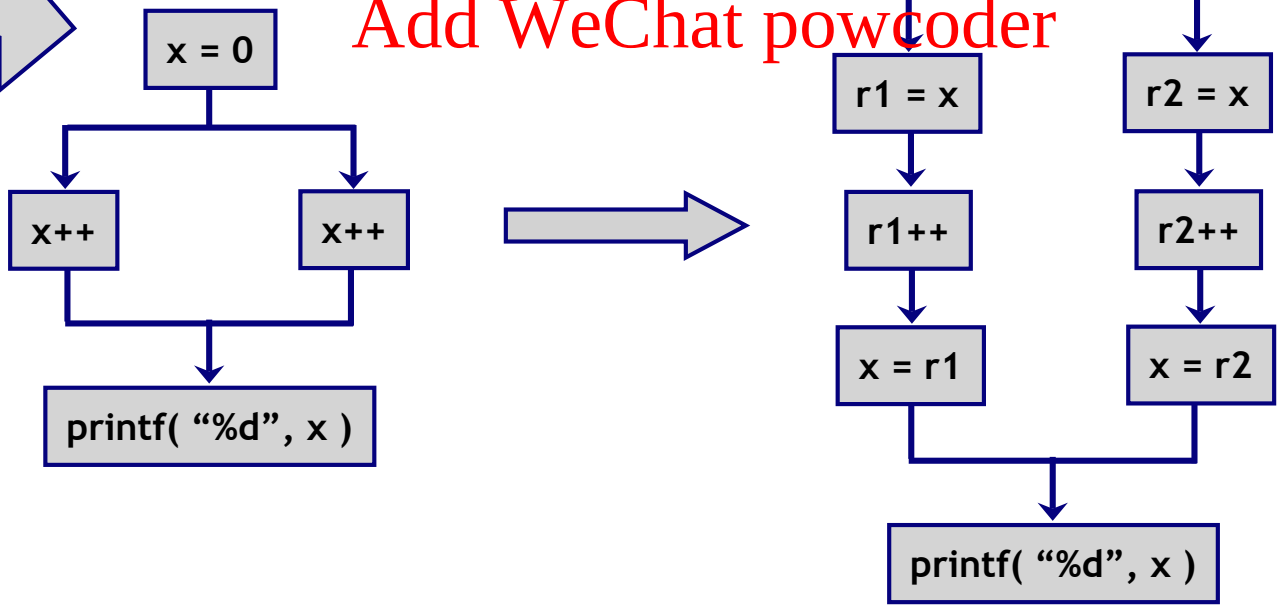
Calculated in 11.978701 seconds using 24 workers

Actually just takes about 0.5 seconds in real time

Race Conditions or 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


```
int r = 0;

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

race

```
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

Some Concluding Remarks

Cilk and Cilk++ seem 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) – *we will not cover these*

Remaining lectures will show many more examples!