

CSE 260

Lecture 19

Parallel Programming Languages

Announcements

- Thursday's office hours are cancelled
- Office hours on Weds 2p to 4pm
- Jing will hold OH, too, see Moodle

Today's lecture

- Parallel Programming Languages
 - ♦ Cilk
 - ♦ X10

Dynamic parallelism

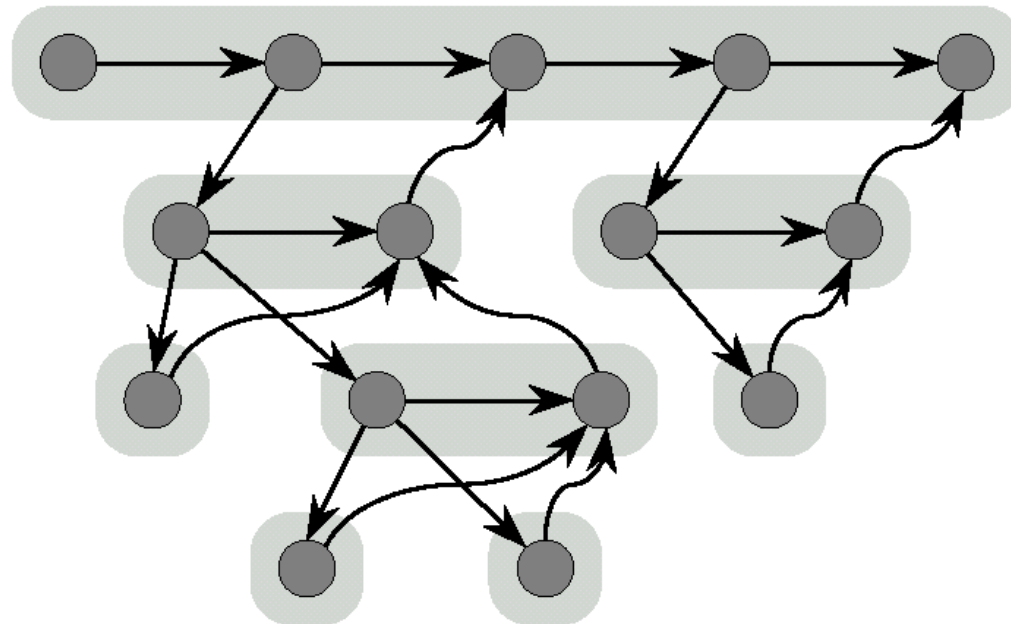
- How to support dynamic creation of parallelism, while hiding the details
- Dynamic parallelism is much harder to manage than static parallelism
 - ◆ How to keep the processors equally busy?
 - ◆ How to avoid excessive overhead costs?

Managing application complexity

- Focus on on thread-based parallelism
- Threads communicate anonymously
 - ◆ Correctness and synchronization
 - ◆ Workload distribution
- Scalability
- Task granularity

An alternative

- Let's think of a computation in terms of a graph, more precisely, a DAG
- Nodes denote computation, edges data dependence



CILK

- CILK is a programming language that supports a constrained model of thread-based parallelism with *guarantees* about *performance*
- Useful in implementing divide and conquer algorithms
- See <http://supertech.lcs.mit.edu/cilk>
- Cilk Plus: an extension to C and C++
 - ♦ Supported by Intel compilers and GCC 4.7
 - ♦ See <http://software.intel.com/en-us/articles/intel-cilk-plus>

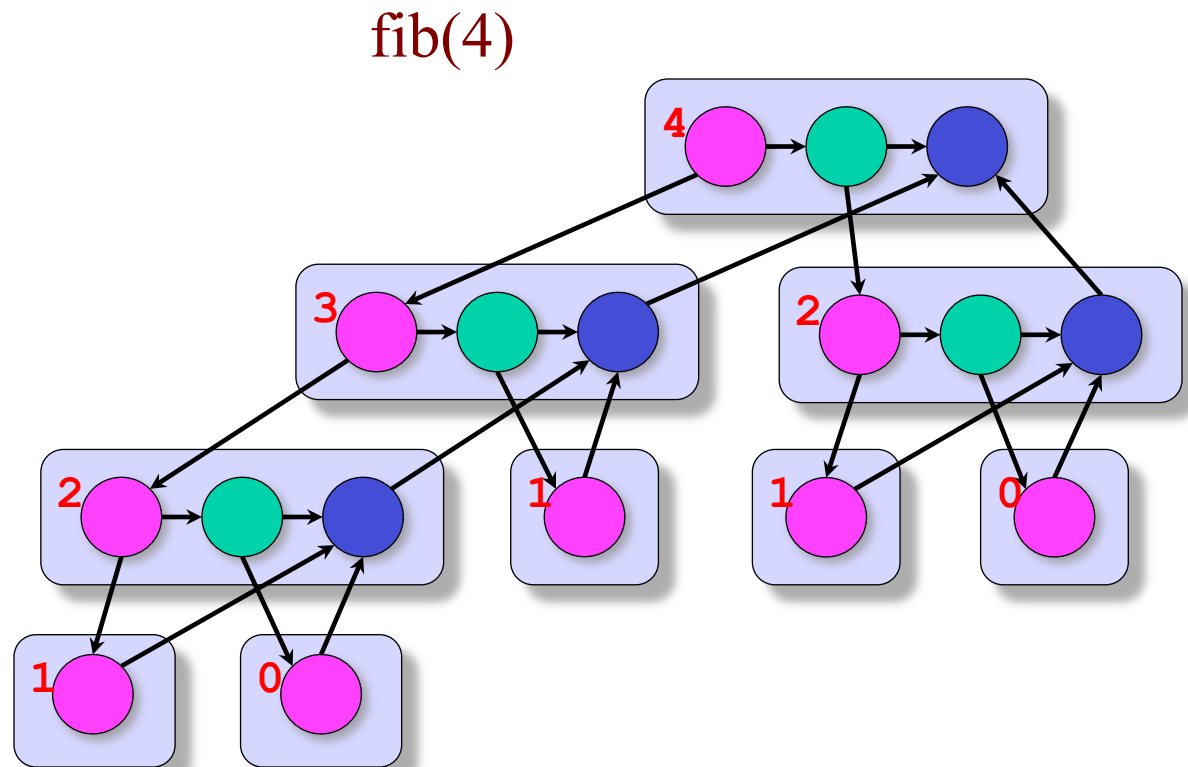
A first CILK program

- `fib()` is called from a dynamically **spawned** thread
- Non-blocking call
- Calls to `fib()` execute concurrently
- Parent continues until it reaches a **sync** barrier, and waits for children to return

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else {
        int x, y;
        x = spawn fib (n-1);
        y = spawn fib (n-2);
        sync;
        return (x+y);
    }
}
```

Call graph for Fibonacci

```
cilk int fib (int n) {  
    if (n < 2) return n;  
    else {  
        int x, y;  
        x = spawn fib (n-1);  
        y = spawn fib (n-2);  
        sync;  
        return (x+y);  
    }  
}
```

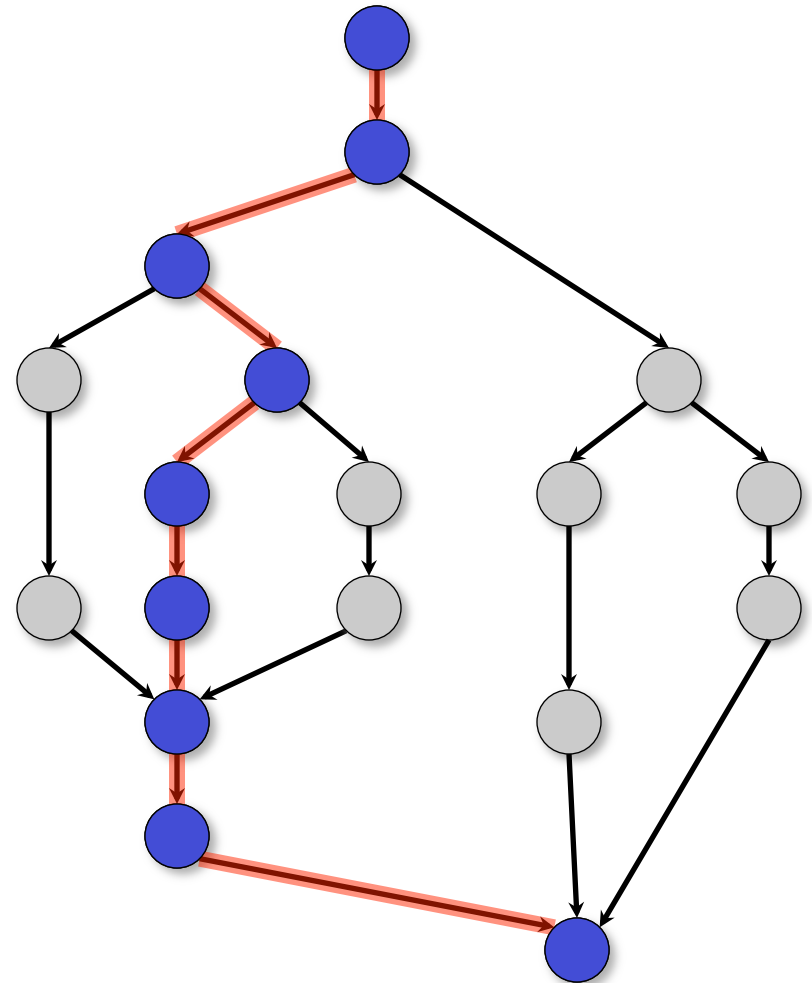


Spawn,
continue and
return edges

© 2006 Charles E. Leiserson

Performance Metrics

- Define *work* as the total time to execute the entire computation on one processor (T_1)
- *Critical path*: the longest time to execute the threads along any dependence path (T_∞)
- Assume P processors
- Define T_p = time on P processors



© 2006 Charles E. Leiserson

Lower bounds on performance

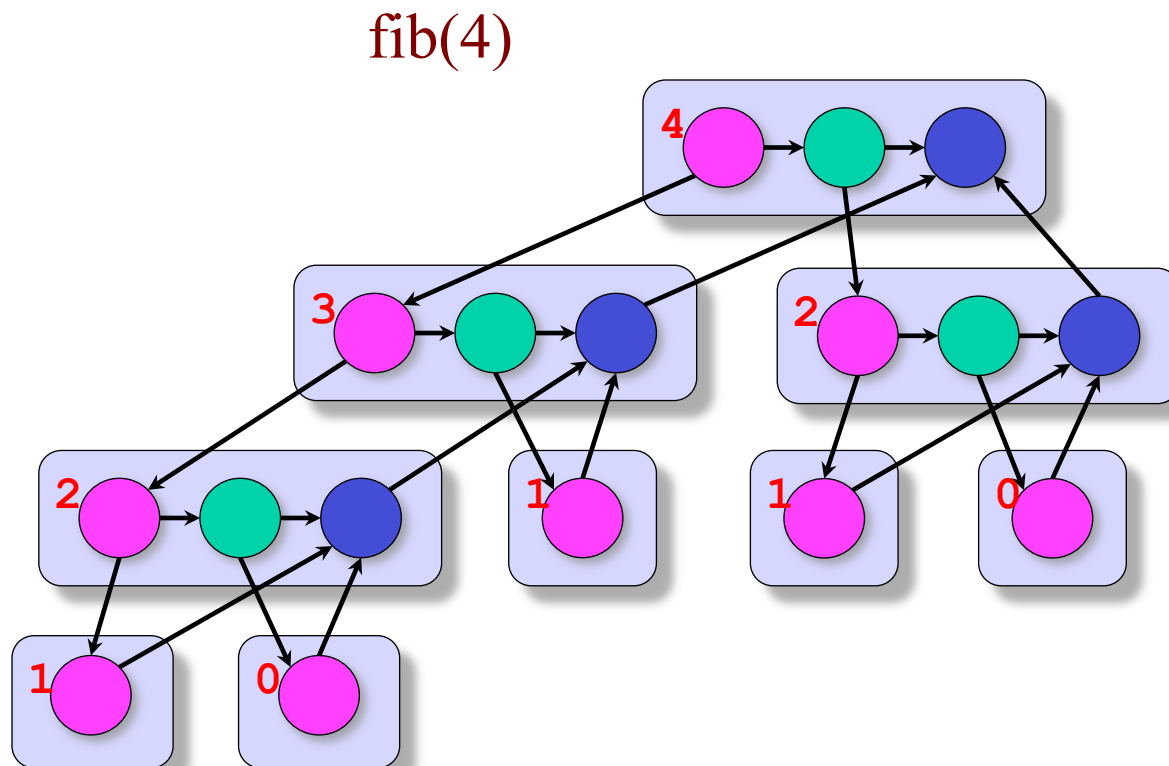
- $T_P \geq T_1 / P$
 - ♦ In one step, P processors can do at most P units of work
 - ♦ May not be true in some search problems
- $T_P \geq T_\infty$
 - ♦ In one step, P processors can do no more work than an infinite number of processors can
- $T_1 / T_P = \text{speedup}$, cannot be superlinear, why?
- $T_1 / T_\infty = \text{available parallelism}$
= average work available along every step along the critical path

Performance of Fibonacci

Work = 17

Critical path: 8

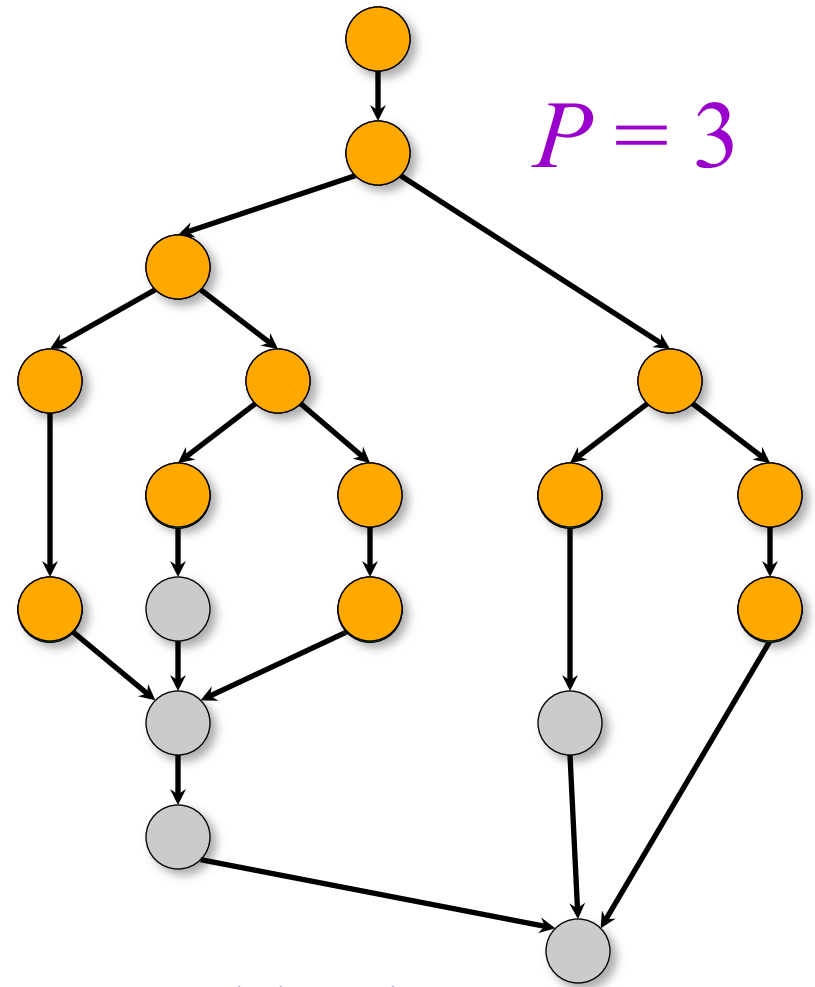
Parallelism: 2.125



© 2006 Charles E. Leiserson

A greedy scheduler

- In each step, perform as much work as possible at most P threads
- A thread is *ready* if all predecessors have completed
- The step is *complete* if $\geq P$ threads are ready:
at most T_1/P complete steps
- Else it is *incomplete*
at most T_∞ incomplete steps



© 2006 Charles E. Leiserson

Theorem

- Theorem due to Graham and Brent

A greedy scheduler executes a computation with work T_1 and critical-path length T_∞ in time

$$T_P \leq T_1 / P + T_\infty$$

- Informal proof (Demmel)

A processor is either *working* or *stealing*.

The total time all processors spend working is T_1 . Each steal has a $1/P$ chance of reducing the span by 1. Thus, the expected cost of all steals is $O(PT_\infty)$.

Since there are P processors, the expected time is

$$(T_1 + O(PT_\infty))/P = T_1/P + O(T_\infty) \blacksquare$$

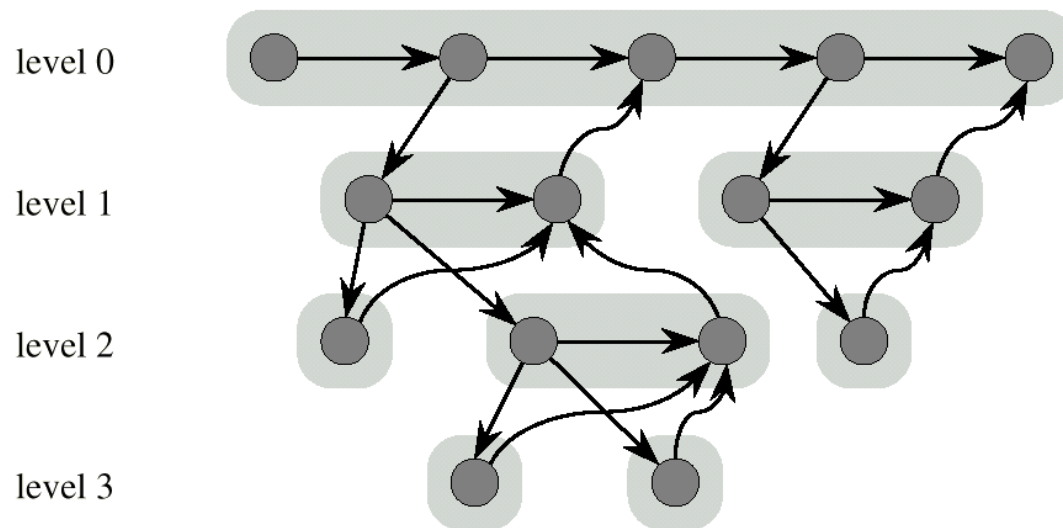
- Corollary: any greedy scheduler is optimal to within a factor of 2

Performance

- CILK's scheduler is provably optimal
$$T_P \leq T_1 / P + O(T_\infty)$$
- Near perfect speedup when $P \ll T_1 / T_\infty$
- Empirically $T_P \approx c_1 T_1 / P + c_\infty T_\infty$
 $c_1 \approx 1.07$, $c_\infty \approx 1.04$, provided $T_1 / T_\infty > 10P$
- $T_P \geq \text{Max}(T_1 / P, T_\infty)$
 - ♦ The critical path is a stronger lower bound on T_P exceeds the average parallelism T_1 / T_∞
 - ♦ Otherwise, T_1 / P is the stronger bound
- Depends on the ability to have good scheduler

Cilk's work stealing scheduler

- When a processor runs out of work it steals a thread from a *victim* taken at random
- The *thief* takes the shallowest thread from the top of the victim's queue. Why?
 - ♦ Low overhead when all are busy, mostly incurred by thief
 - ♦ Little effect on T_1/P term



Matrix multiplication

- HPC Challenge (2006)
- Uses divide and conquer algorithm, performs 8 matrix multiplications on $n/2 \times n/2$ matrices and 1 matrix addition

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} * \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$
$$= \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} + \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix}$$

Matrix multiply in Cilk

```
Cilk Void MM(*C, *A, *B, n){  
    float *T Cilk_alloca(n*n*sizeof(float));  
    spawn MM(C11,A11,B11,n/2)  
    spawn MM(C12,A11,B12,n/2)  
    ...  
    spawn MM(T11,A12,B21,n/2)  
    ...  
    sync  
    spawn Add (C,T,n)  
    sync;  
}
```

$$T_1 = \Theta(n^3)$$

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

$$\text{Parallelism} = n^3 / \lg^2 n$$

```
Cilk Void Add(*C, *T, n){  
    spawn Add(C11,T11,n/2)  
    spawn Add(C12,T12,n/2)  
    spawn Add(C21,T21,n/2)  
    spawn Add(C22,T22,n/2)  
    sync;  
}
```

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

$$T_\infty = \Theta(\lg n)$$

Today's lecture

- Parallel Programming Languages
 - ♦ Cilk
 - ♦ X10