Dynamic Multi-threaded Programming for Shared-Memory Multicore Processors

SC3260/5260 High-Performance Computing

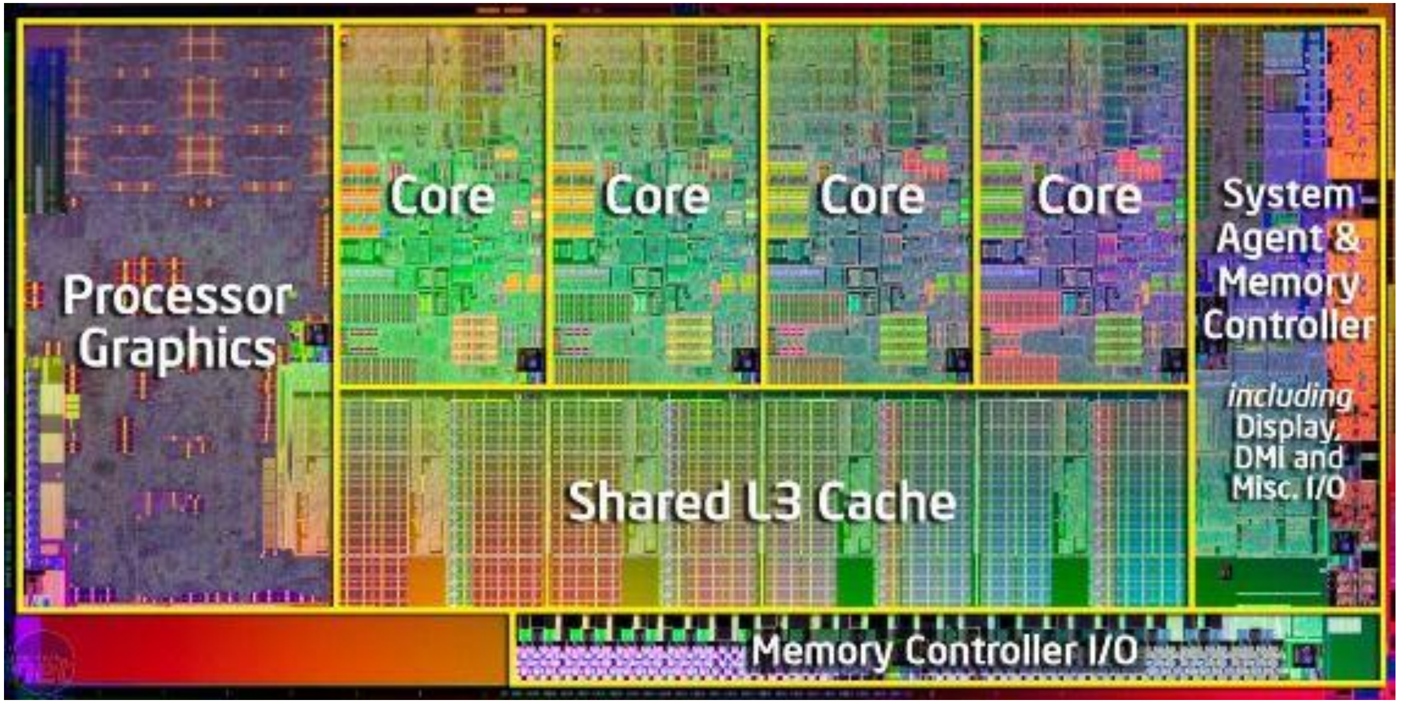
Hongyang Sun
(hongyang.sun@vanderbilt.edu)

Vanderbilt University

Spring 2020

Shared-Memory Multicore Processors

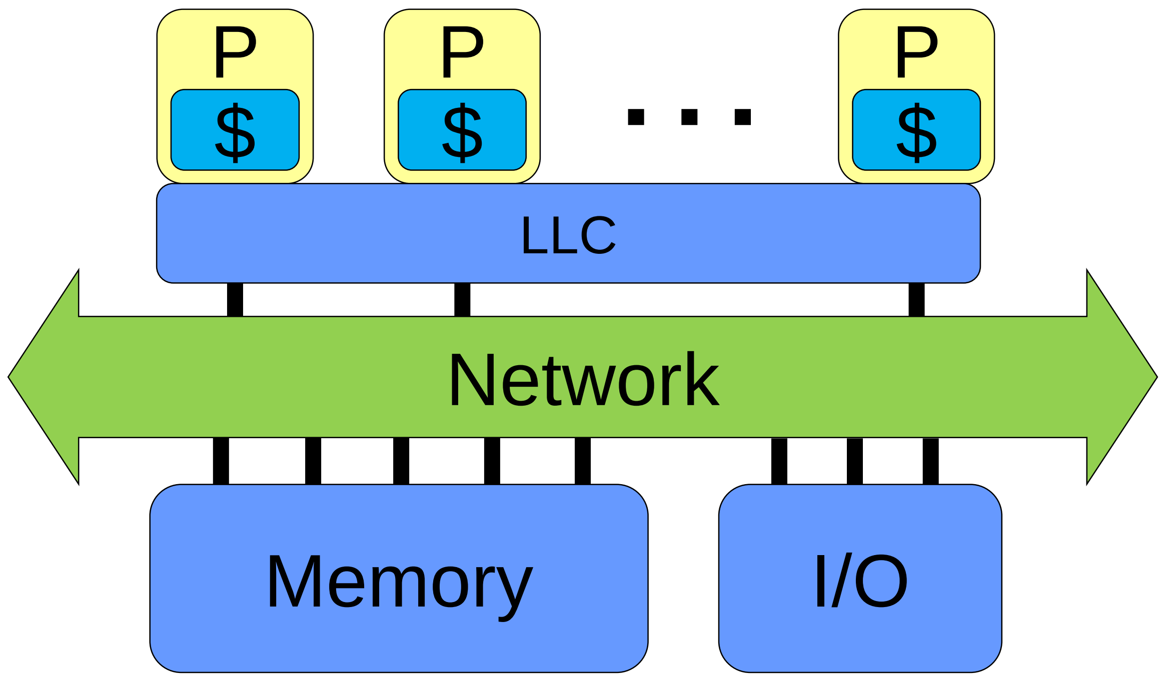
- Very few single-core processors nowadays…



Intel Sandy Bridge Quad-core Processors

Abstract Multicore Architecture

- Multiple processor cores share the same memory space, I/O (or last-level cache), but have private cache.



Multicore Programming

- Programming directly on multicore processors is difficult, due to issues like synchronization, load balancing, etc.
- A concurrency platform provides an abstract programming model for multicore processors
  - POSIX Threads (Pthreads)
  - Intel Threading Building Blocks (TBB)
  - OpenMP
  - Cilk/CilkPlus

Cilk/CilkPlus

- Simple language extension to C/C++ to support dynamic task parallelism and data parallelism with just a few keywords (cilk_spawn, cilk_sync, cilk_for).
- Distinct features of Cilk/CilkPlus
  - Simple design and implementation.
  - Provably-efficient work-stealing scheduler.
  - Hyperobject library to perform parallel reduction.
  - Suite of tools to detect determinacy-race and to analyze scalability of code.

History of Cilk Development

- Originally developed by the Supertech research group led by Professor Charles Leiserson at MIT in the 1990’s;
- Later commercialized as a spinoff company, Cilk Arts., in 2006;
- Subsequently acquired by Intel in 2009;
- Available through open-source software, maintained by Cilk Hub (http://cilk.mit.edu/) since 2017.

Example: Fibonacci Numbers

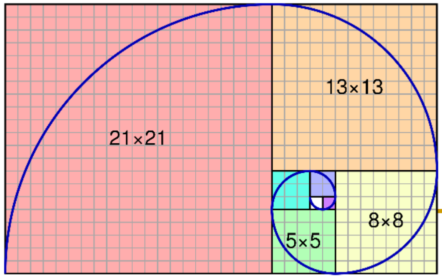
- Fibonacci numbers form a sequence of numbers (0, 1, 1, 2, 3, 5, 8, 13,…) such that each number is sum of two preceding ones, starting from 0 and 1.

Fibonacci, *filius Bonacci* ("son of Bonacci"), Italian mathematician. c. 1170–c. 1240–50

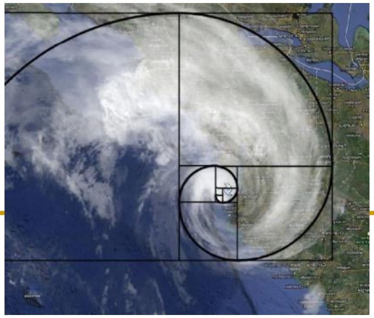
- Recurrence:

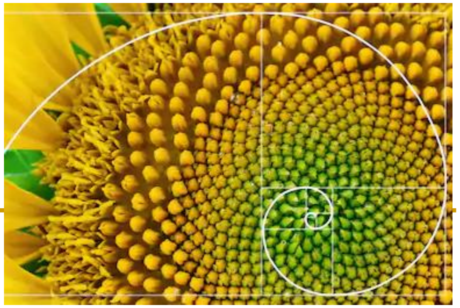
$$F_n = \begin{cases} n, & \text{for } n \le 1 \\ F_{n-1} + F_{n-2}, & \text{for } n > 1 \end{cases}$$

- Many interesting occurrences in nature & maths



The Fibonacci spiral



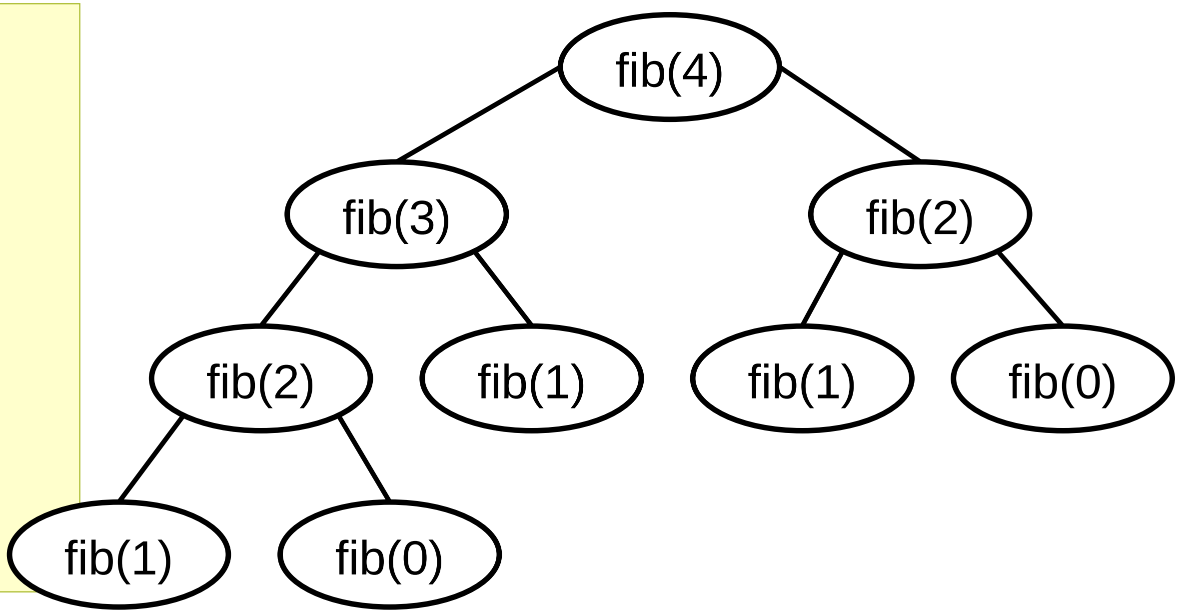


Computing *n*-th Fibonacci number

Serial C program

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

Recursion tree



- ***Remark***: This is a very inefficient (i.e., exponential time) algorithm to compute the *n*-th Fibonacci number, but a good example to illustrate multi-threaded programming.

Computing *n*-th Fibonacci number

Parallel Cilk program

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

"Cilk_spawn" creates a child thread that ***may*** be executed in parallel with the parent (main) thread

"Cilk_sync" requires that all children threads ***must*** return before the parent thread can continue

- Serial elision: removing the cilk keywords (cilk_spawn and cilk_sync) of a cilk program always returns a legal serial C program.

Example: Matrix-Vector Multiply

- Compute $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ a_{1,0} & \cdots & a_{1,n-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

Serial C program

```
Mat_Vec_Mut(A, x, y, n){
  for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
         y[i] += A[i][j]*x[j];
     }
  }
}
```

Example: Matrix-Vector Multiply

- Compute $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ a_{1,0} & \cdots & a_{1,n-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

Parallel Cilk program

```
Mat_Vec_Mut(A, x, y, n){
  cilk_for (int i=0;i<n;i++){
        for (int j=0;j<n;j++){
            y[i] += A[i][j]*x[j];
        }
  }
}
```

"Cilk_for" allows the iterations within the for-loop to be executed in parallel

Can we parallelize the inner for-loop as well?

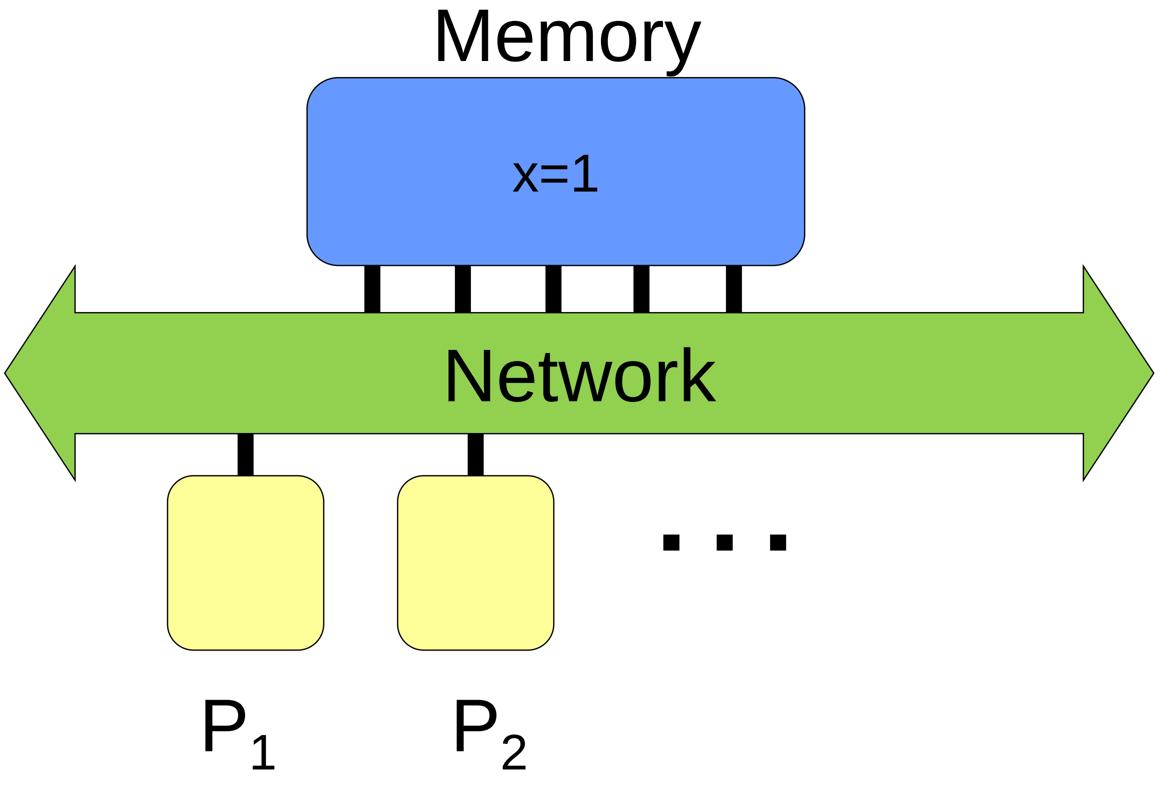
Issues with Shared-Memory Processors and Multicore Programming

- Cache Coherence
- Race Conditions
- Performance Models
- Load Balancing/Scheduling

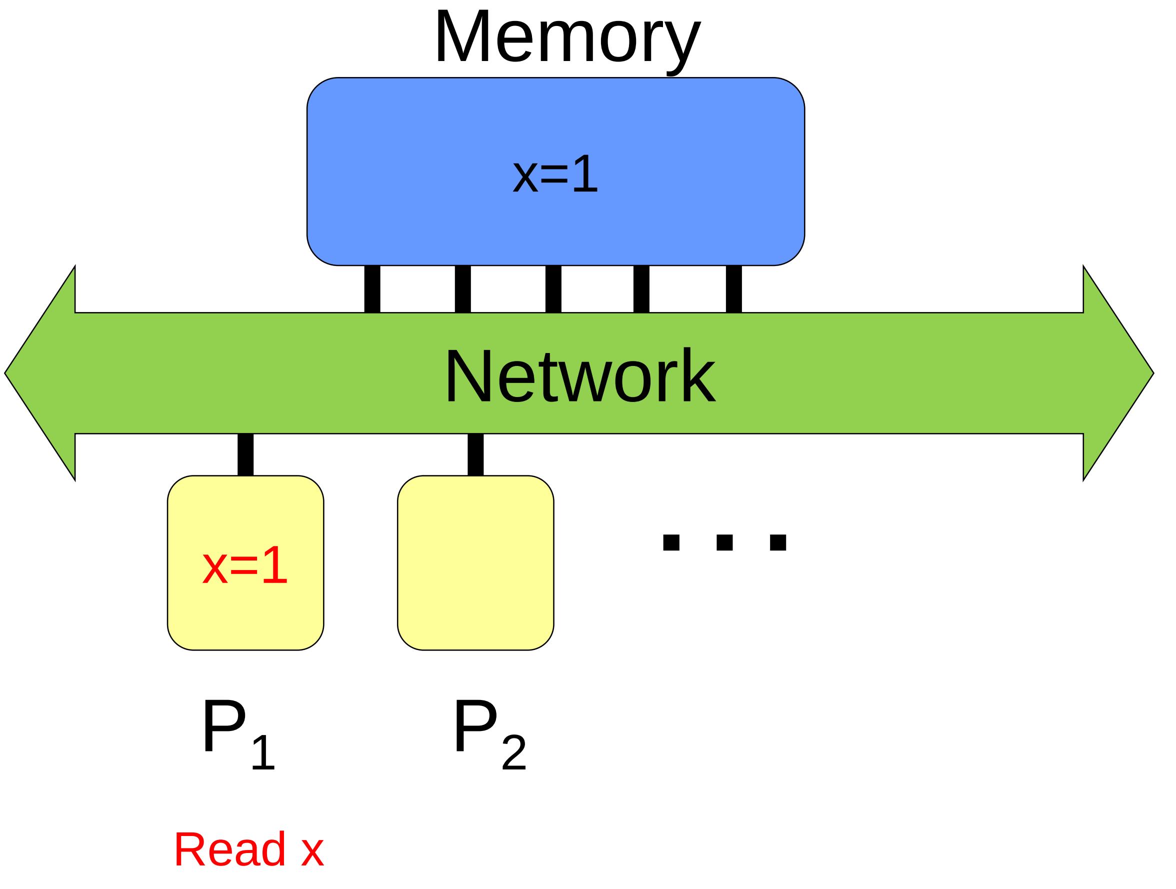
Cache Coherence

- In a shared-memory system where each processor has a separate private cache (e.g., L1 or L2), it is possible to have multiple copies of shared data: one in main memory and one in local cache of each processor.
- Cache coherence ensures that different copies of the shared data are consistent with each other (i.e., when one copy changed, the other copies must reflect the change).

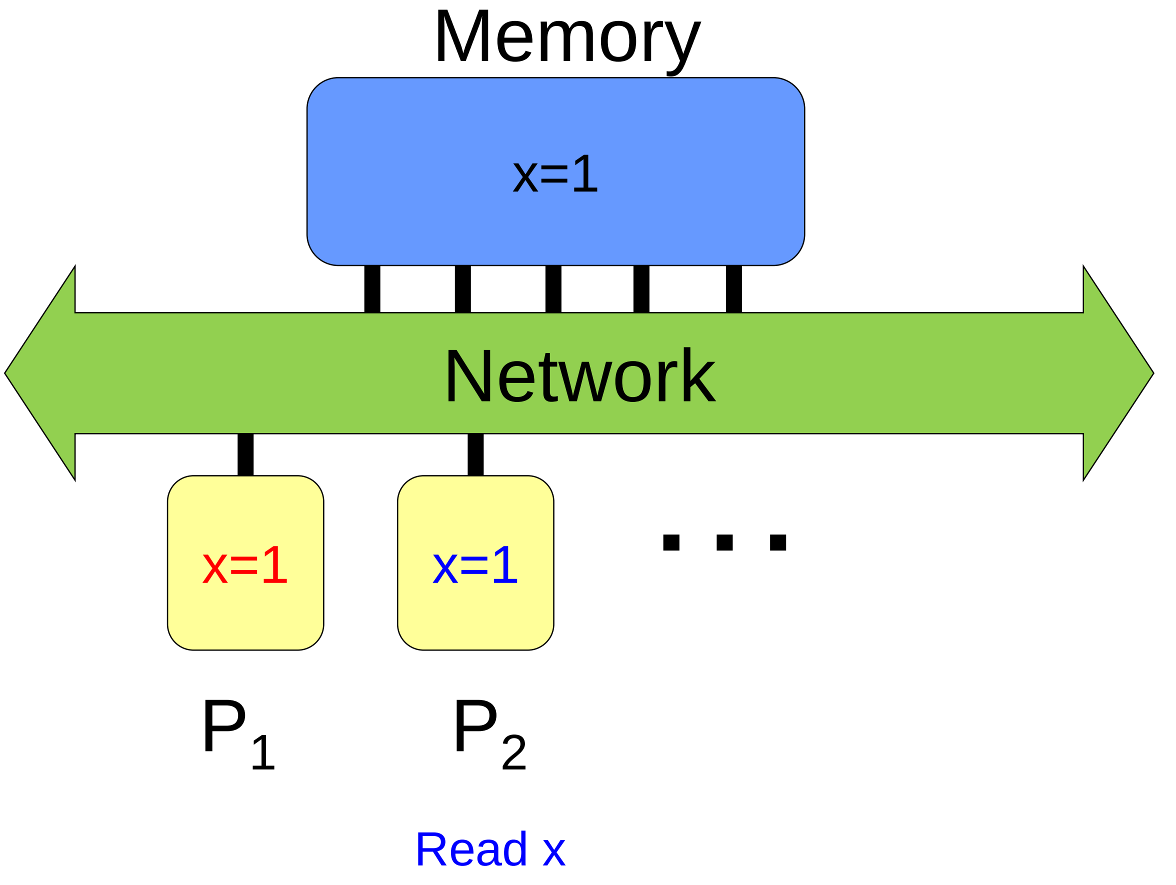
Example



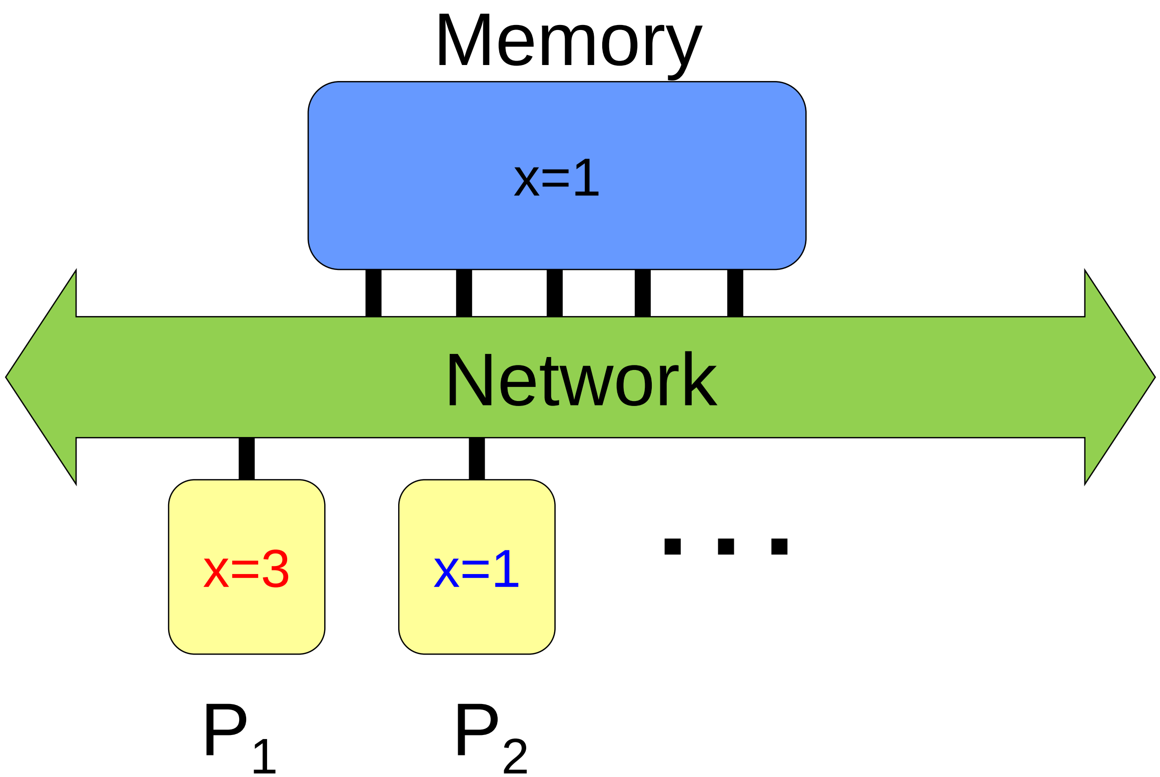
Example



Example

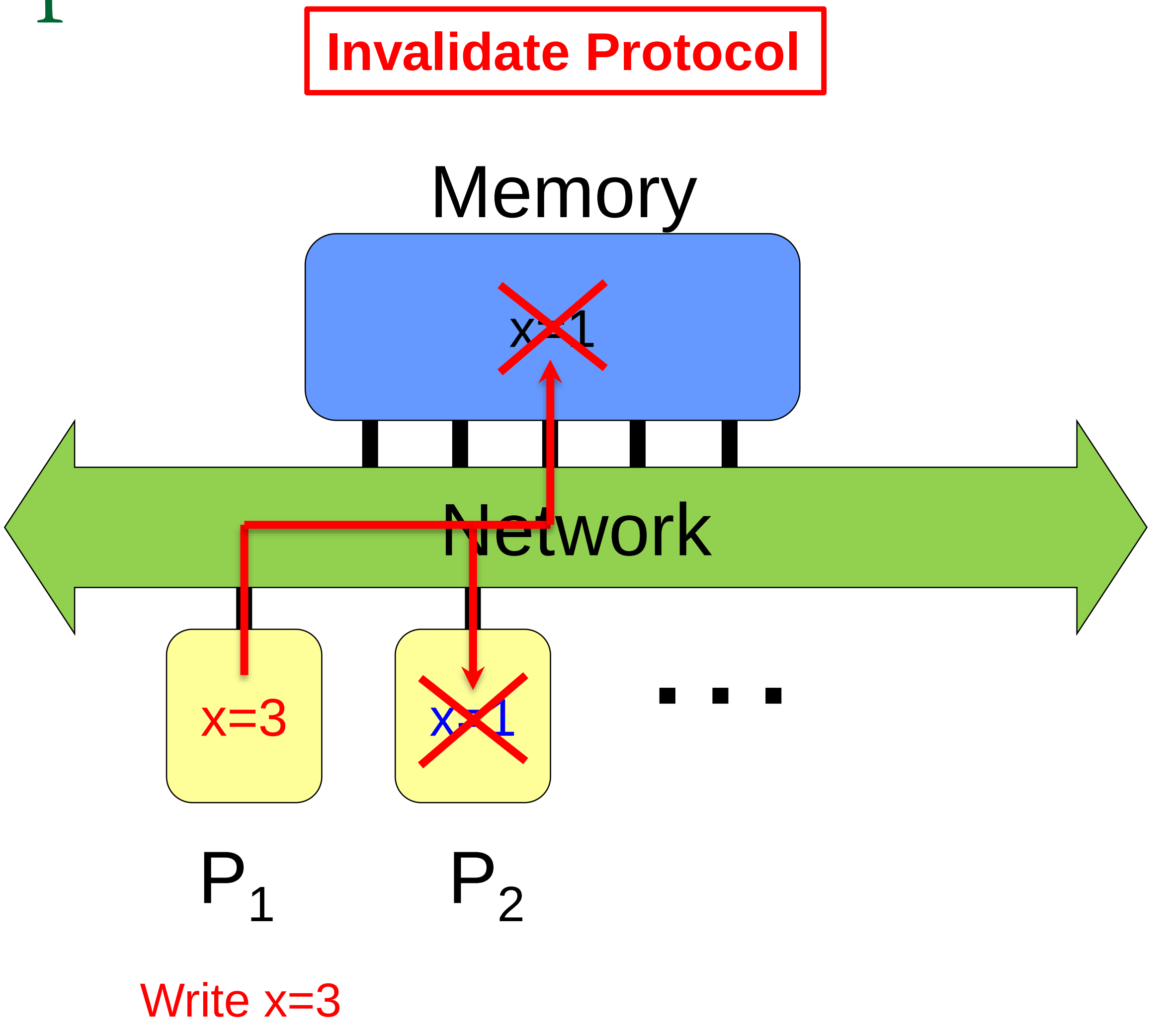


Example

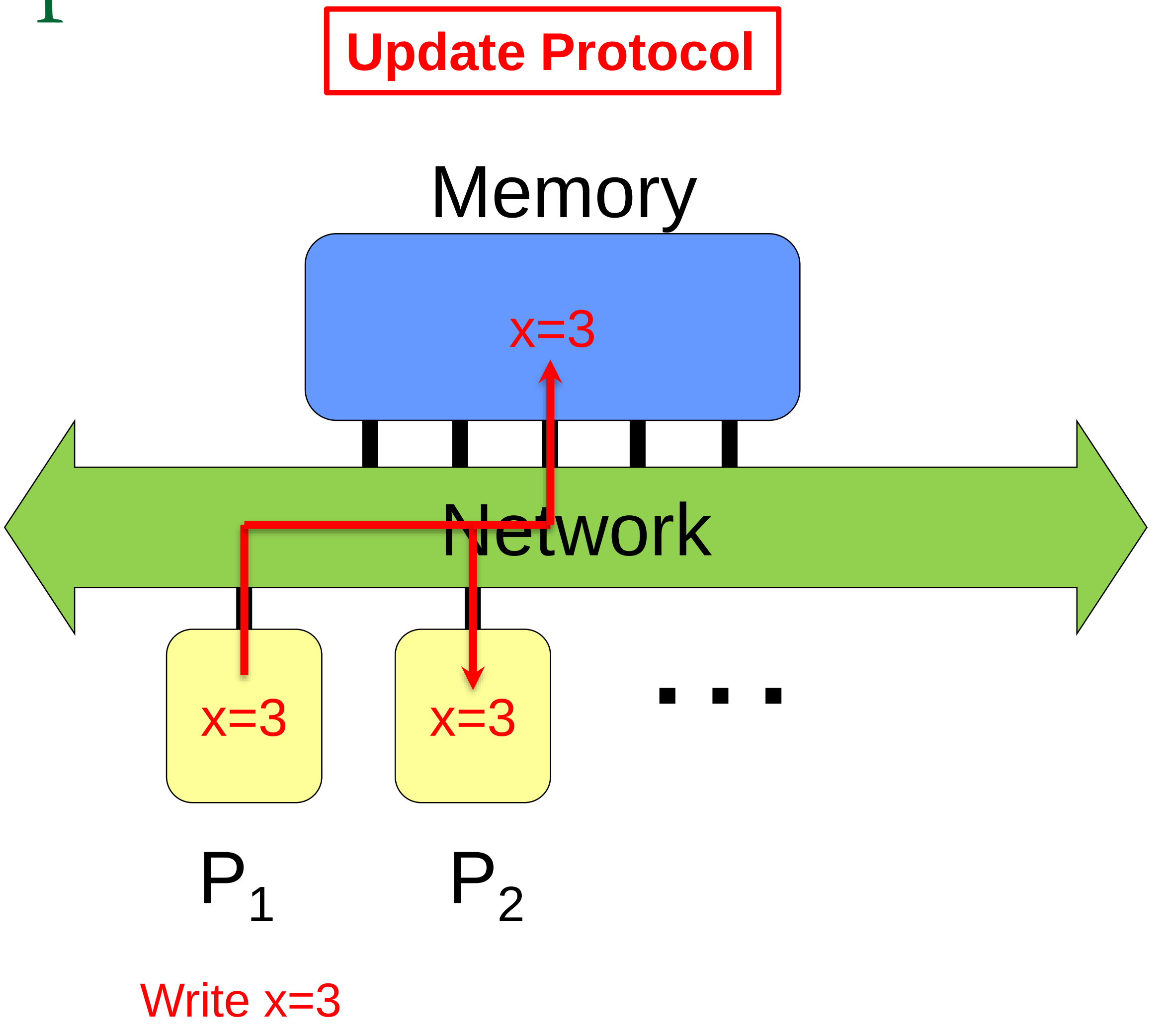


Write x=3

Example



Example



Cache Coherence

- Various different mechanism (e.g., using snoopy cache or directory) have been devised based on either invalidate protocol or update protocol.
- Examples of cache coherence protocols include: MSI, MESI, MOSI, MOESI, etc. by defining a set of states (e.g., Modified, Shared, Invalidate) for each cache block/line, and events that trigger transitions among these states.
- *Cache coherence is typically handled by the hardware (with performance penalty), and the programmer needs not worry about this issue* ☺

Issues with Shared-Memory Processors and Multicore Programming

- Cache Coherence
- Race Conditions
- Performance Models
- Load Balancing/Scheduling

Race Conditions

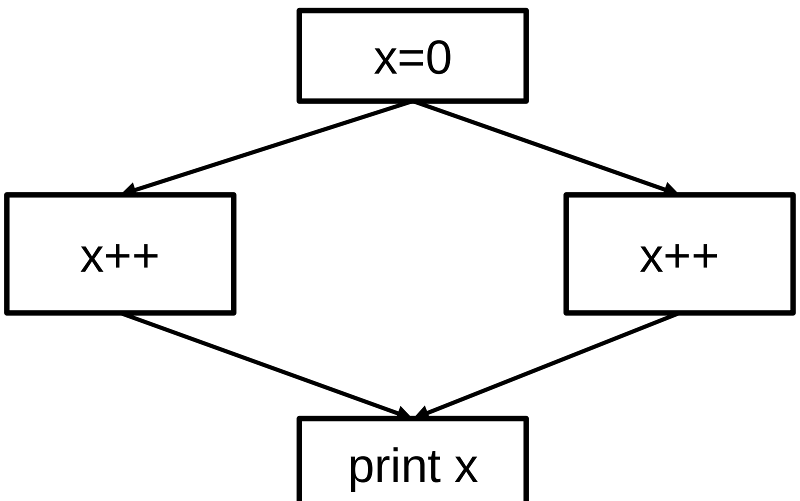
- A multi-threaded program is deterministic if it always does the same thing on the same input. Otherwise, it is nondeterministic if its behavior might vary from run to run.
- A (determinacy) race condition occurs when two logically parallel instructions access the *same memory location* and at least one of the instructions performs a *write*. The output depends on which instruction "*wins the race*".

A Simple Race Example

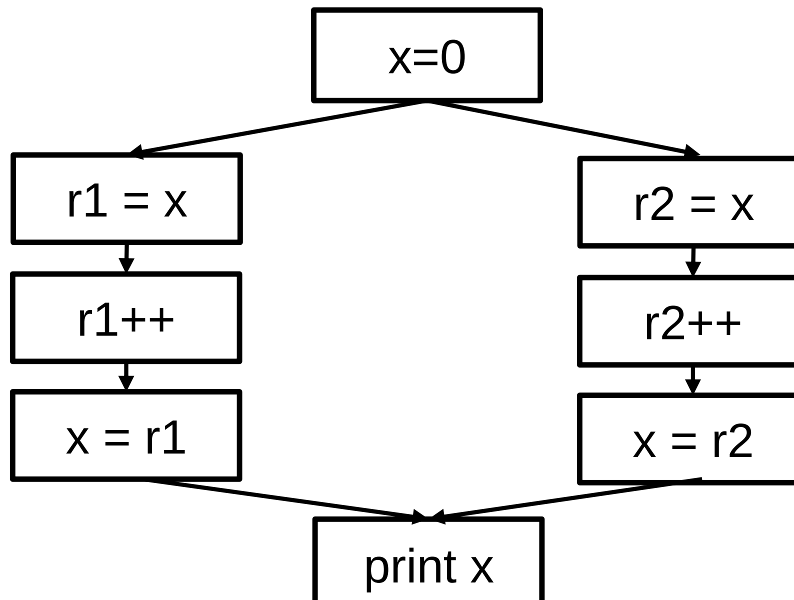
```
race_example(){
   int x = 0;
   cilk_for (int i=0;i<2;i++){
      x++;
   }
   print x;
}
```

What is the output of "**print x**"?

Dependency graph

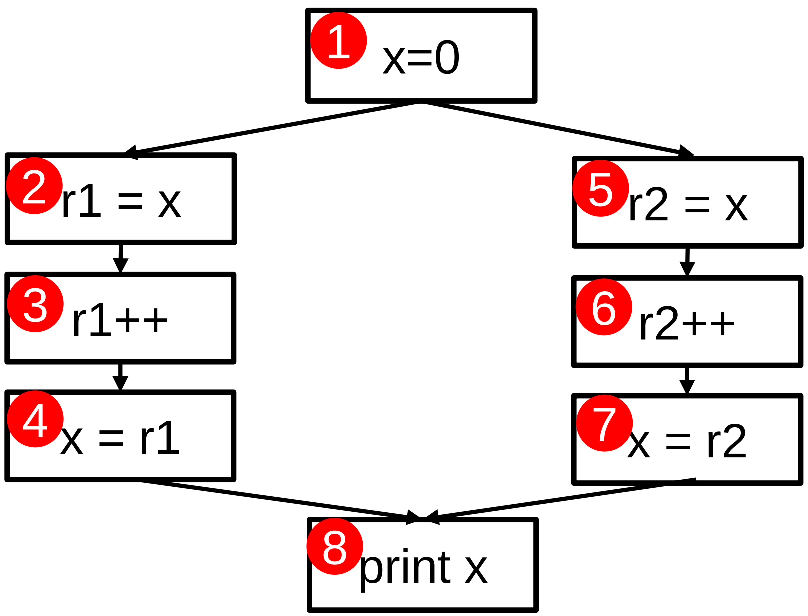


Detailed dependency graph



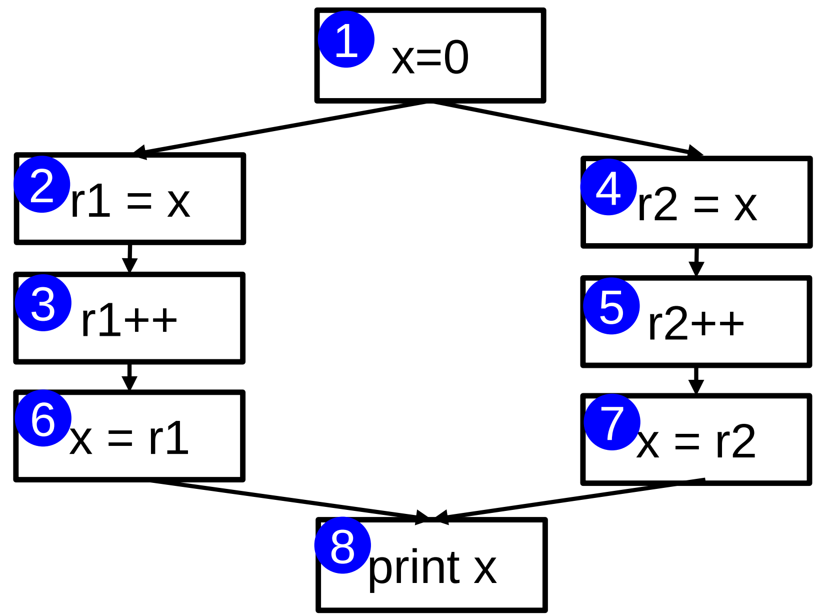
Determinacy Race

One possible execution sequence



Output: x=2

Another execution sequence



Output: x=1

Race Example: Matrix-Vector Multiply

- Compute $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$

$$\begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} a_{0,0} & \cdots & a_{0,n-1} \\ a_{1,0} & \cdots & a_{1,n-1} \\ \vdots & \ddots & \vdots \\ a_{n-1,0} & \cdots & a_{n-1,n-1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

Parallel Cilk program

```
Mat_Vec_Mut(A, x, y, n){
   cilk_for (int i=0;i<n;i++){
      cilk_for (int j=0;j<n;j++){
          y[i] += A[i][j]*x[j];
      }
   }
}
```

Parallelizing the inner for-loop can create a **race condition**!

Race Bugs

- Famous computer bugs due to race conditions include:
  - Therac-25 radiation therapy machine, which killed three people and injured several others in 1980s.
  - North American Blackout of 2003, which left over 50 million people without power.
- These race bugs are notoriously difficult to find; You can run tests in the lab for days without a failure only to discover that your software sporadically crashes in the field.

Avoiding Race Bugs

- Use locks (that serialize access of the same memory location), but it can degrade performance.
- Cilk provides hyperobject that maintain local views of the same variable in different threads, and merge upon sync.
- Be careful in control construct and make sure that:
  - Iterations of a cilk_for loop are independent;
  - Logically parallel threads created between cilk_spawn and cilk_sync are independent.
- Use Cilksan race detection tool, which is guaranteed to report race conditions given a Cilk program and an input at compile time. (see http://cilk.mit.edu/tools/)

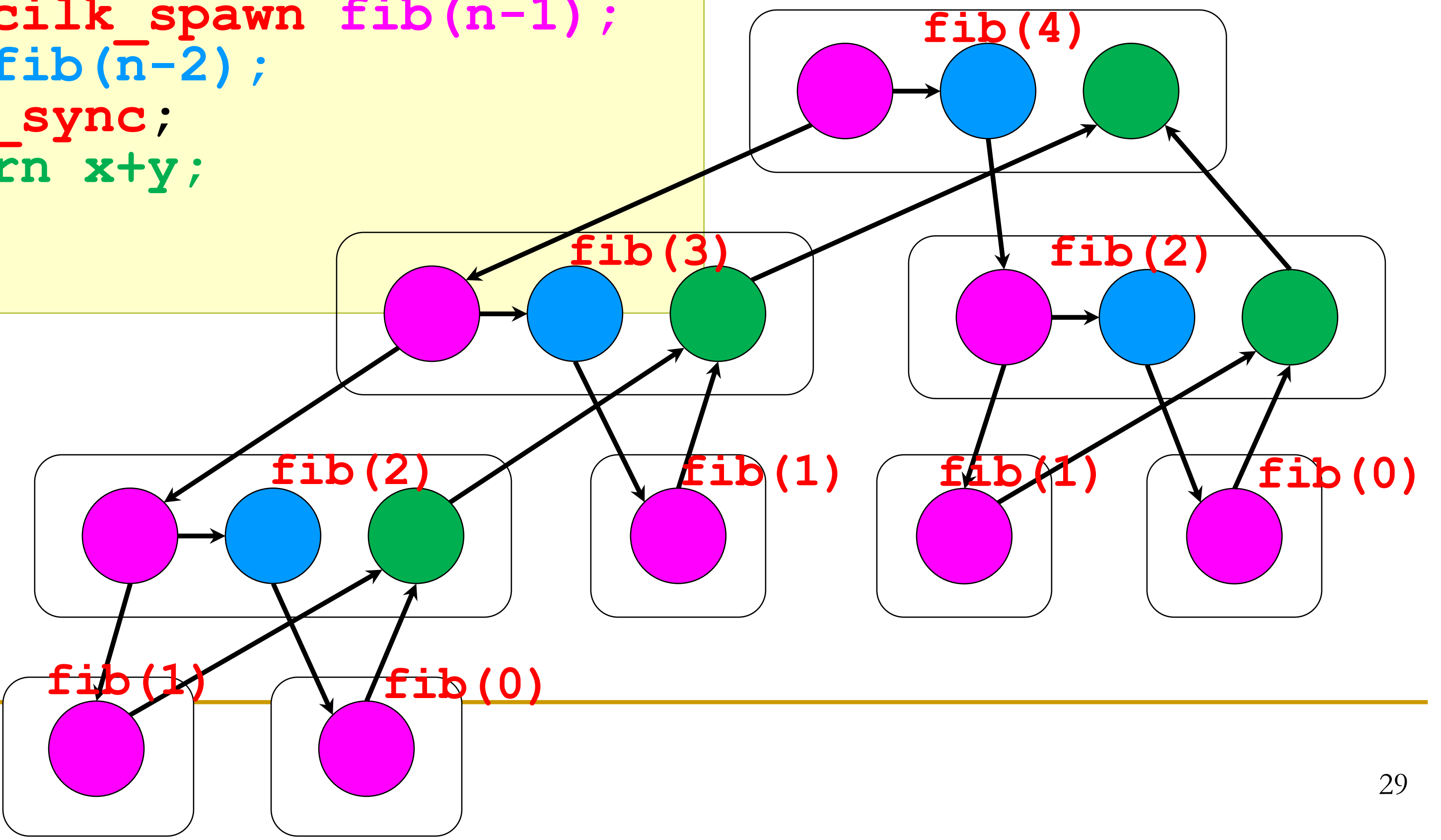
Issues with Shared-Memory Processors and Multicore Programming

- Cache Coherence
- Race Conditions
- Performance Models
- Load Balancing/Scheduling

Computation Model

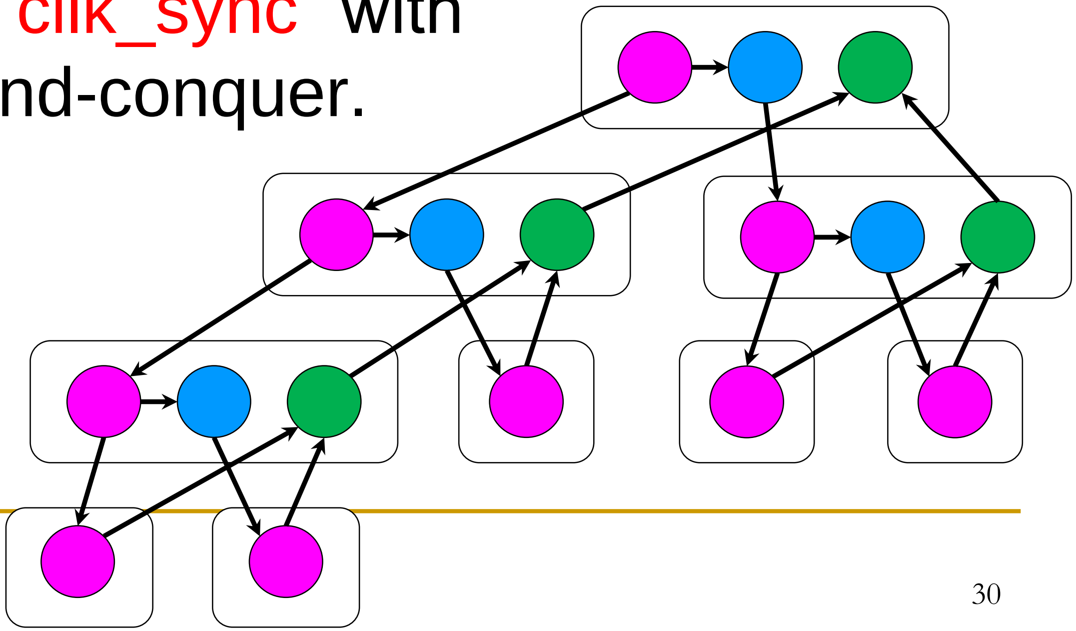
```
int fib(int n){
   if (n<=1){
      return n;
   }
   else{
      int x, y;
      x = cilk_spawn fib(n-1);
      y = fib(n-2);
      cilk_sync;
      return x+y;
   }
}
```

Computation modeled as a dynamically unfolding directed acyclic graph (DAG)



Computation Model

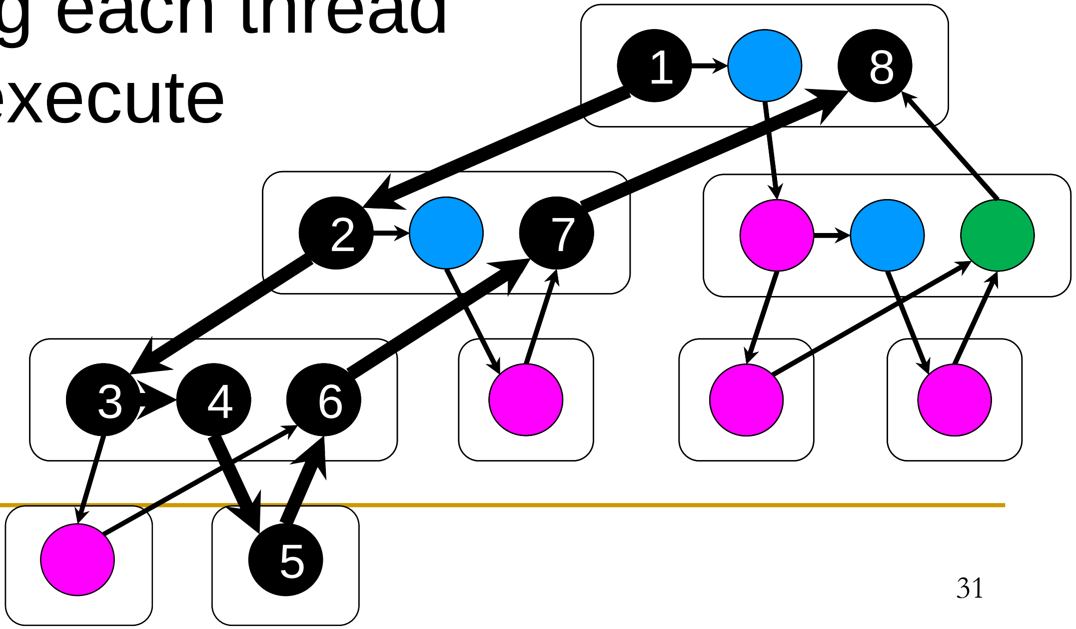
- Computational DAG $G = (V, E)$.
  - Each vertex $v \in V$ represents a thread.
  - Each edge $e \in E$ represents a forward progress (e.g., due to spawn, return, or continue).
  - "cilk_for" loops are converted to "cilk_spawn" and "cilk_sync" with recursive divide-and-conquer.



Program Characteristics

- Intrinsic characteristics of a program
  - T_1: total work of computation.
  - T_∞: length of critical path or span of computation.
  - $\bar{P} = T_1/T_\infty$: average parallelism of computation.

- For fib(4), assuming each thread takes unit time to execute
  - $T_1 = 17$
  - $T_\infty = 8$
  - $\bar{P} = 2.125$



Performance Measures

- Runtime performance of a program (depending on the scheduler)
  - T_P: execution time on P processors

    Work Law: $T_P \geq T_1/P$

    Span Law: $T_P \geq T_\infty$

  - $S_P = T_1/T_P$: speedup on P processors

    If $S_P = P$, then perfect linear speedup

    If $S_P = \Theta(P)$, then linear speedup

    If $S_P < P$, then sublinear speedup

    If $S_P > P$, then superlinear speedup

Speedup vs. Parallelism

- In this performance model, maximum possible speedup on P processors is

$$S_P \leq \min(P, \bar{P})$$

Due to “Work Law”

Due to “Span Law”

- Using an efficient (e.g., greedy or work stealing) scheduler, a program achieves near perfect linear speedup (i.e., $S_P \approx P$) if

$$P \ll \bar{P}$$

Optimize Parallel Performance

- Careful design of parallel programs
  - Conserve work T_1
  - Reduce span T_∞

  $\Big\}\rightarrow$ Increased parallelism $\bar{P}$
- Choose $P < \bar{P}$ processors/cores to run the program, otherwise wasting resources.
- Use Cilkscale scalability analysis tool, which analyzes the work and span of a program to derive upper bounds on parallel performance (see http://cilk.mit.edu/tools/).

Issues with Shared-Memory Processors and Multicore Programming

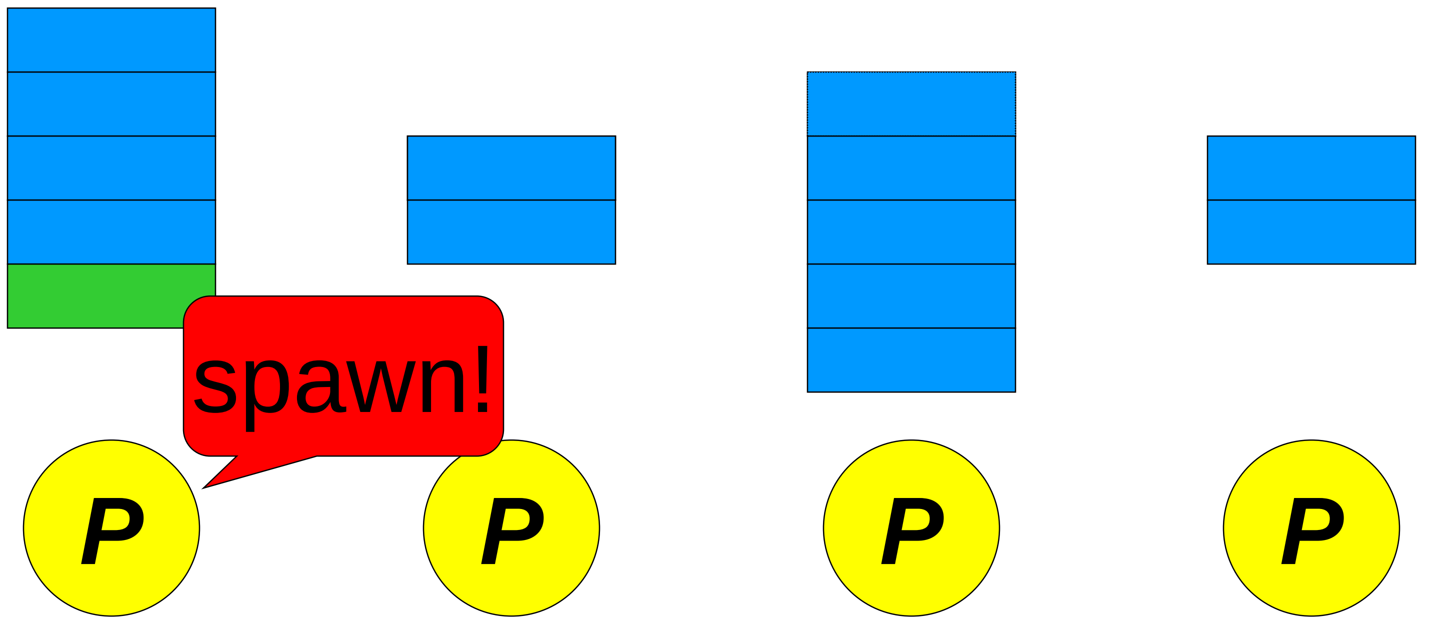
- Cache Coherence
- Avoiding Race Conditions
- Performance Models
- Load Balancing/Scheduling

Load Balancing/Scheduling

- Mapping logical threads (e.g., created by cilk_spawn) onto physical processors/cores to balance the loads of different workers.
- Cilk features a provably-efficient work-stealing scheduler that maps threads onto processors dynamically at runtime.
- *The programmer needs not worry about load balancing/scheduling; it is automatically handled by the runtime system ☺*

Work-Stealing Scheduler

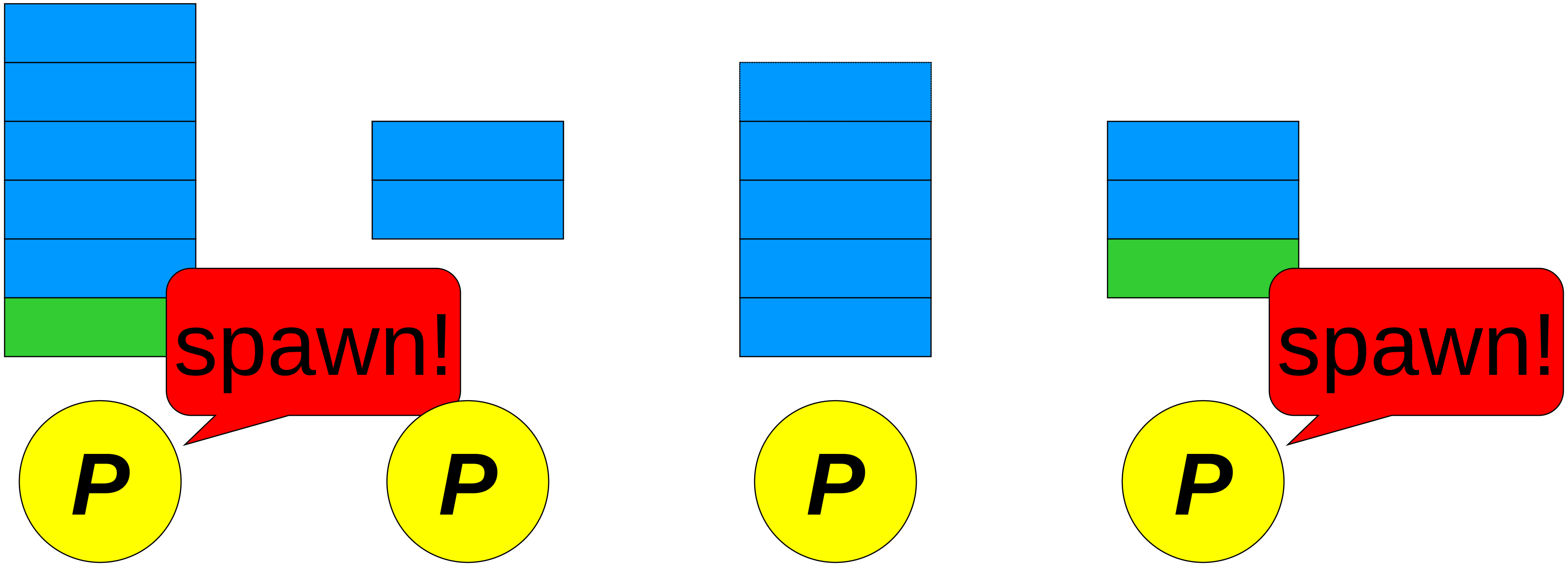
Each processor maintains a *double-ended queue (deque)*



- When a parent thread ***spawns*** a child thread, the parent thread is pushed onto the *bottom* of deque, and the processor works on the child thread.

Work-Stealing Scheduler

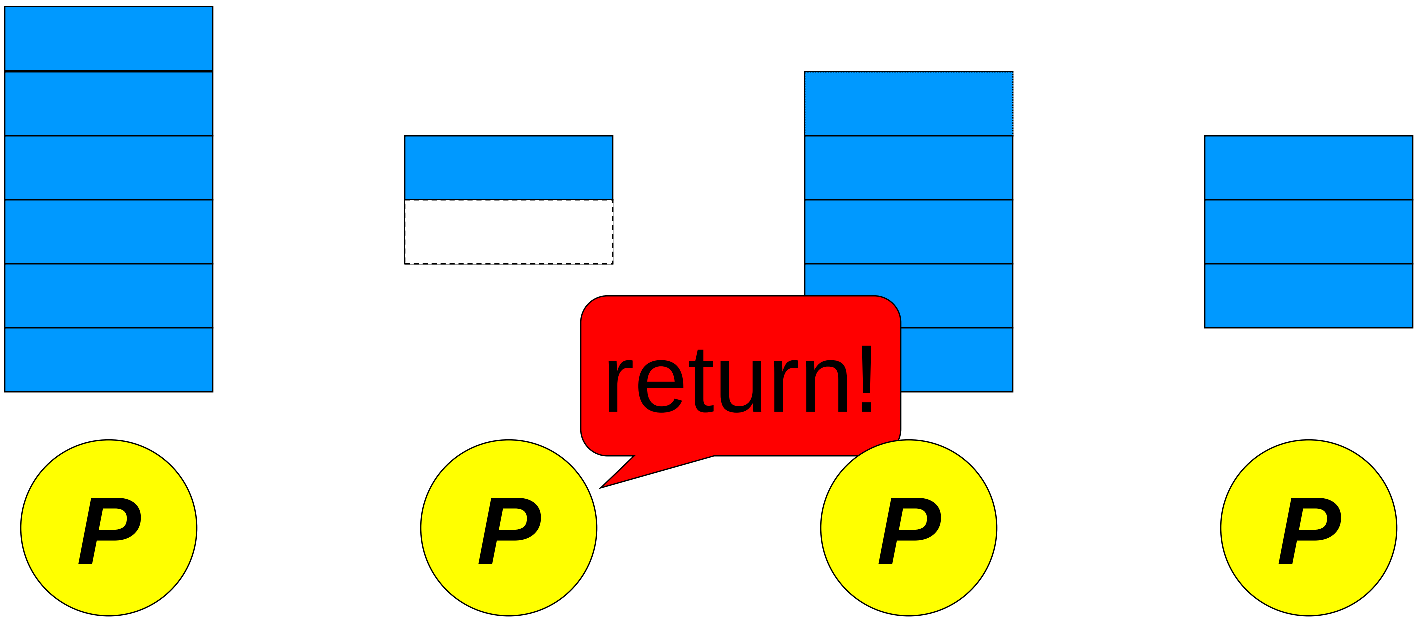
Each processor maintains a *double-ended queue (deque)*



- When a parent thread ***spawns*** a child thread, the parent thread is pushed onto the *bottom* of deque, and the processor works on the child thread.

Work-Stealing Scheduler

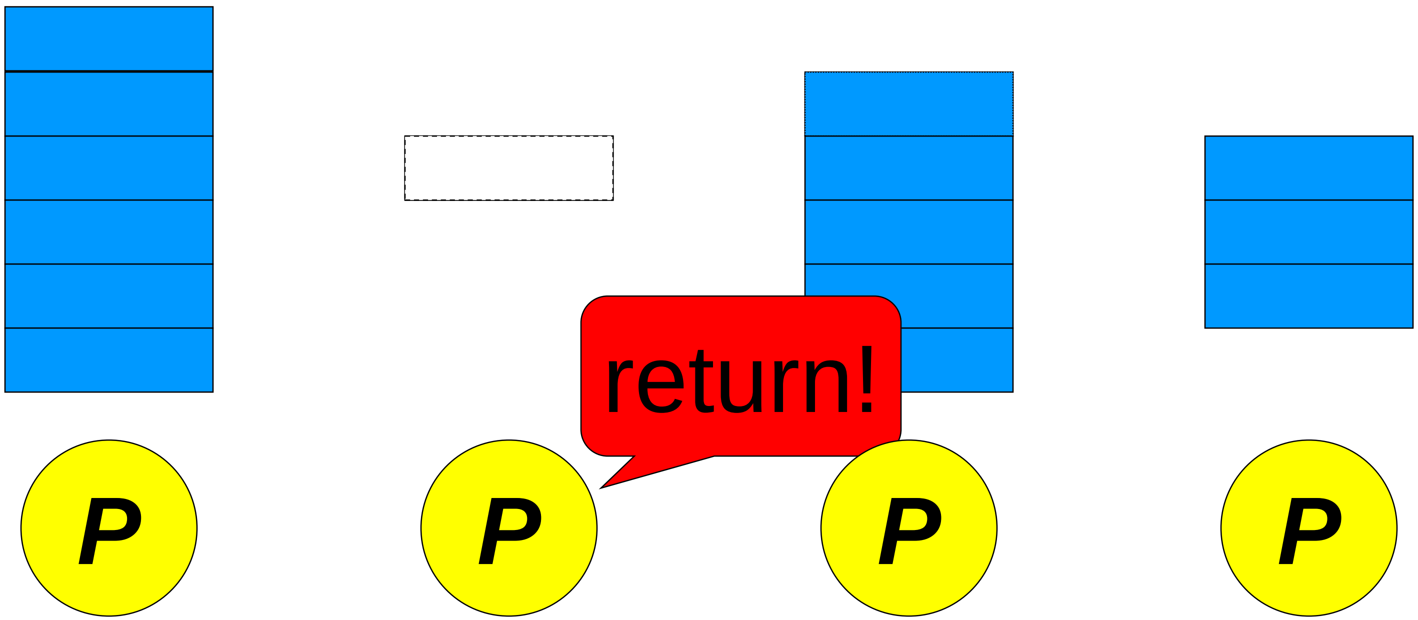
Each processor maintains a *double-ended queue (deque)*



- When a current thread ***returns***, the processor fetches the *bottom* thread from the deque and works on it.

Work-Stealing Scheduler

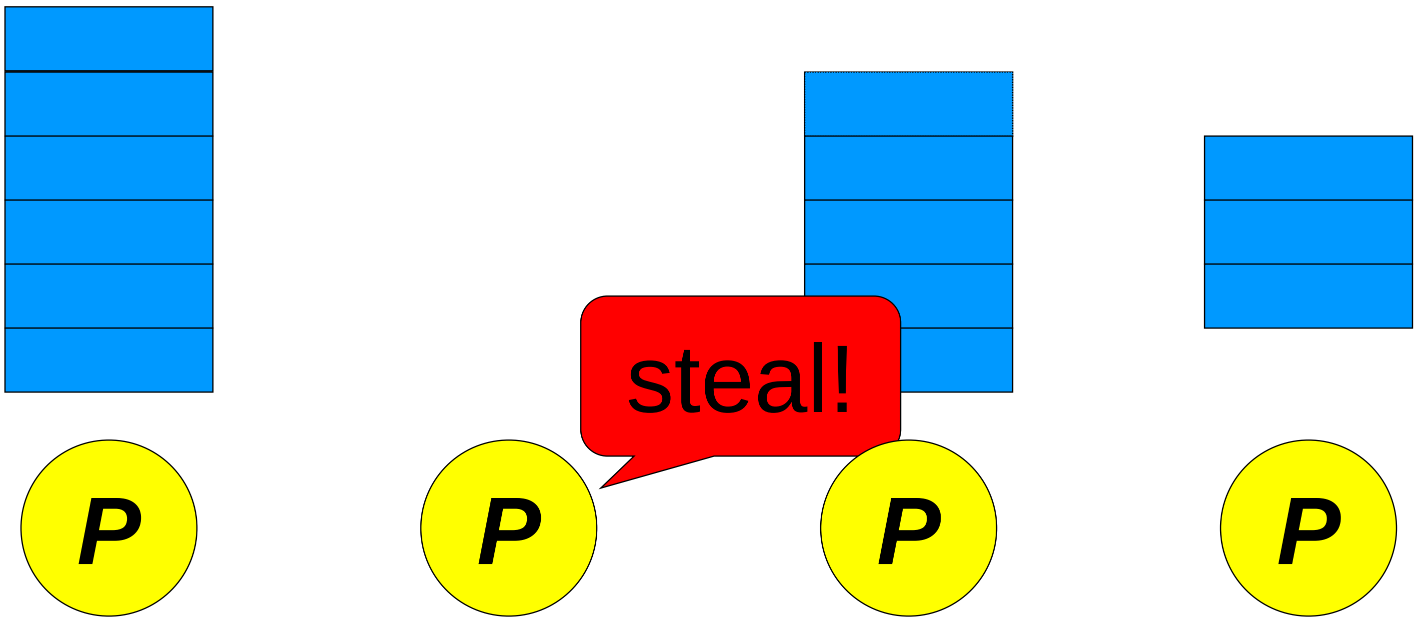
Each processor maintains a *double-ended queue (deque)*



- When a current thread ***returns***, the processor fetches the *bottom* thread from the deque and works on it.

Work-Stealing Scheduler

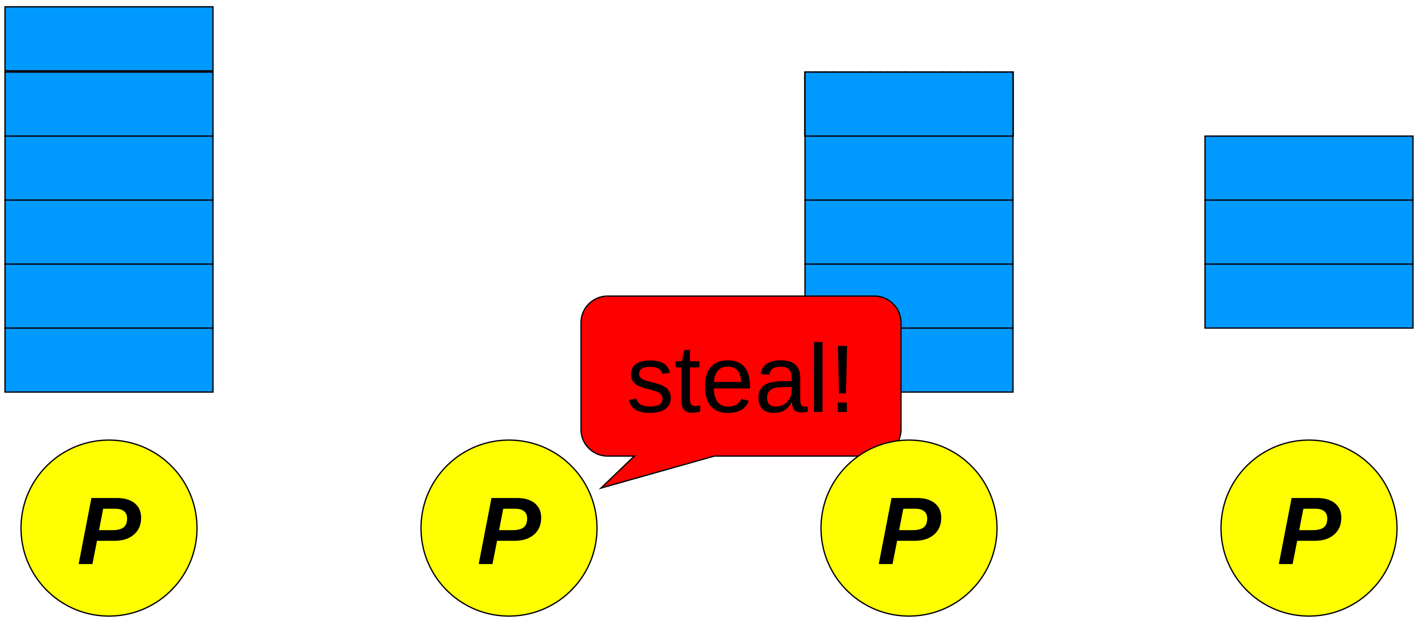
Each processor maintains a *double-ended queue (deque)*



- When a processor runs out of work, it randomly selects a "*victim*", and ***steals*** the thread from the *top* of the victim's deque (if victim has no work, select another victim).

Work-Stealing Scheduler

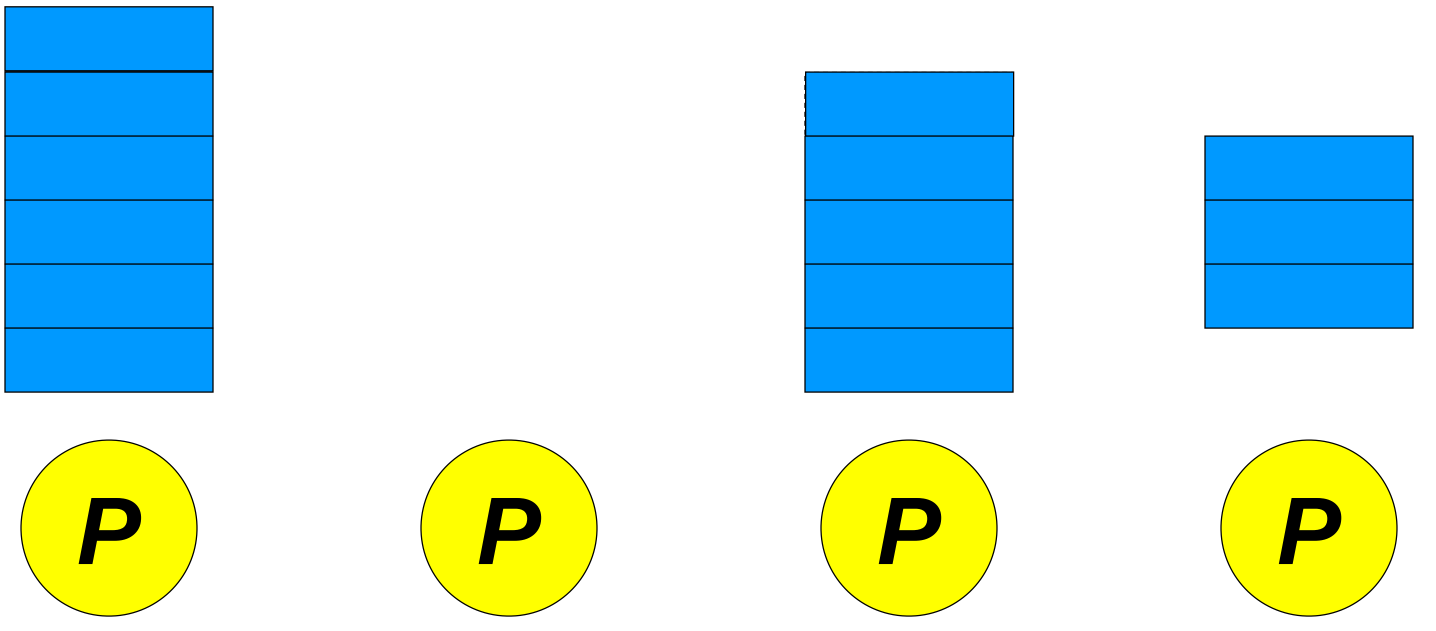
Each processor maintains a *double-ended queue (deque)*



- When a processor runs out of work, it randomly selects a "*victim*", and ***steals*** the thread from the *top* of the victim's deque (if victim has no work, select another victim).

Work-Stealing Scheduler

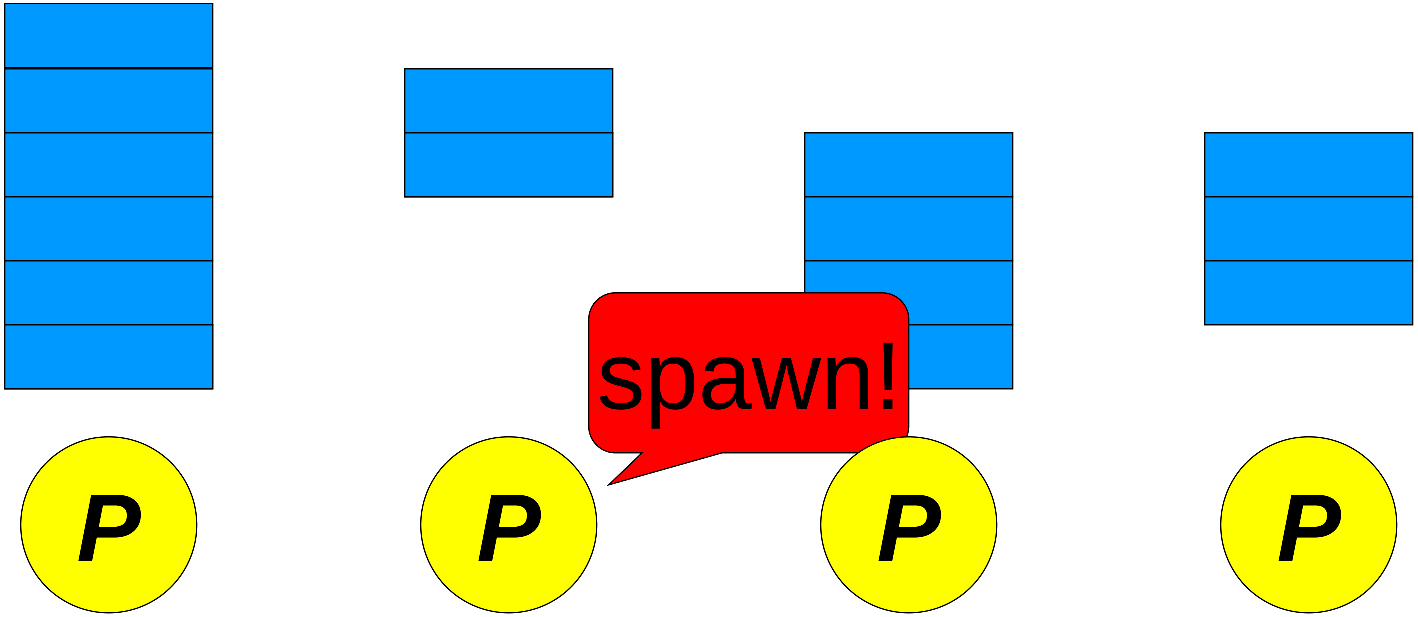
Each processor maintains a *double-ended queue (deque)*



- When a processor runs out of work, it randomly selects a "*victim*", and ***steals*** the thread from the *top* of the victim's deque (if victim has no work, select another victim).

Work-Stealing Scheduler

Each processor maintains a *double-ended queue (deque)*



- When a parent ***spawns*** a child thread, parent is pushed onto the *bottom* of deque, and the processor works on child.

Performance of Work Stealing

Theorem [BL94] Suppose a program with T_1 work and T_∞ span executes on P processors. Then, a work-stealing scheduler achieves an expected running time:

$$T_P \approx T_1/P + O(T_\infty)$$

and uses a stack space at most:

$$M_P \leq P \cdot M_1$$

where M_1 is the stack space required by a serial execution of the program.

Installing Cilk/CilkPlus

- Open-source implementation of Cilk uses the Tapir/LLVM compiler (based on Clang&LLVM), which compiles a Cilk program more efficiently.
- Install Tapir/LLVM and Cilk Plus runtime (See http://cilk.mit.edu/download/ for instructions).
- Compile:

```
$ clang -fcilkplus fib.c -o fib
```

- Run:

```
$ CILK_NWORKERS=4 ./fib 41
```

Dynamic Multi-threaded Algorithms

- We will study dynamic multi-threaded algorithms for the following problems:
  - Matrix Multiplication, Matrix Transpose, Reduction, Prefix Sum (Scan), Sorting, Stencil.
- Homework 1 (due on Feb. 14):
  - Choose one: Matrix Multiplication (implementation of one algorithm) or Stencil (design of two algorithms).
- Possible Problems for Term Project:
  - Matrix Multiplication or Sorting (Email us if you would like to work on any of the two problems for the term project).

References

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms (3rd ed.), Chapter 27. *MIT Press and McGraw-Hill*, 2009.
- C. E. Leiserson. Cilk. *In D. A. Padua, editor, Encyclopedia of Parallel Computing. Springer*, 2011.
- C. E. Leiserson. Multithreaded Programming in Cilk Lectures.
- Cilk Hub. http://cilk.mit.edu/