Introduction to Cilk Programming

COS597C

Feng Liu

Outline

- Introduction
- Basic Cilk programming
- Cilk runtime system support
- Conclusion

Parallel Programming Models

Libraries	Pthread, MPI, TBB, …
New Languages and Extensions	Erlang, Cilk, Haskell, CUDA, OpenCL, NESL, StreamIt, Smalltalk, Unified Parallel C (UPC), F#, …
Unsorted	OpenMP, Global Arrays, X10

Parallel programming is really ***hard*****…..**

- task partition / data partition
- synchronization
- communication
- new syntax

Source: http://en.wikipedia.org/wiki/Parallel_programming_model

TIOBE Programming Languages Ranking

- Java / C / C++ dominate the market!

Position Sep 2010	Position Sep 2009	Delta in Position	Programming Language	Ratings Sep 2010	Delta Sep 2009	Status
1	1	=	Java	17.915%	-1.47%	A
2	2	=	C	17.147%	+0.29%	A
3	4	↑	C++	9.812%	-0.18%	A
4	3	↓	PHP	8.370%	-1.79%	A
5	5	=	(Visual) Basic	5.797%	-3.40%	A
6	7	↑	C#	5.016%	+0.83%	A
7	8	↑	Python	4.583%	+0.65%	A
8	18	↑↑↑↑↑↑↑↑↑↑	Objective-C	3.368%	+2.78%	A
9	6	↓↓↓	Perl	2.447%	-2.08%	A
10	10	=	Ruby	1.907%	-0.47%	A
11	9	↓↓	JavaScript	1.665%	-1.33%	A

Source: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

Cilk Motivation (1)

- **The programmer** should focus on structuring his program to expose parallelism and exploit locality

- **The compiler and runtime system** are with the responsibility of scheduling the computation task to run efficiently on the given platform.

- Two key elements:
  - The program language should be **"simple"**.
  - The runtime system can guarantee an efficient and provable **performance** on multi-processors.

Cilk Motivation (2)

- Cilk is a C/C++ extensions to support **nested data and task parallelisms**

- The Programmers identify elements that can safely be executed in parallel
  - Nested loops ➔ data parallelism➔ cilk threads
  - Divide-and-conquer algorithms ➔ task parallelism➔ cilk threads

- The run-time environment decides how to actually divide the work between processors
  - can run without rewriting on any number of processors

Important Features of Cilk

- Extends C/C++ languages with **six** new keywords
  - cilk, spawn & sync
  - inlet & abort
  - SYNCHED
- Has a serial semantics
- Provides performance guarantees based on performance abstractions.
- Automatically manages low-level aspects of parallel execution by Cilk’s runtime system.
  - Speculation
  - Workload balancing (work stealing)

Outline

- Introduction
- Basic Cilk programming
- Cilk runtime system support
- Conclusion

Basic Cilk Programming (1)

- C/C++ extensions to support nested task and data parallelism
- Fibonacci example

Sequential version

```
int fib (int n) {
    if (n < 2) return 1;
    else {
        int rst = 0;
        rst += fib (n-1);
        rst += fib (n-2);
        return rst;
    }
}
```

Basic Cilk Programming (2)

- C/C++ extensions to support nested task and data parallelism
- Fibonacci example

Sequential version

```
int fib (int n) {
    if (n < 2) return 1;
    else {
        int rst = 0;
        rst += fib (n-1);
        rst += fib (n-2);
        return rst;
    }
}
```

Pthread version

```
arg_structure;

void * fib(void * arg);

  pthread_t tid;
  pthread_create(tid, fib, arg);
…
  phread_join(tid);
  pthread_exit;
}
```

Basic Cilk Programming (3)

- C/C++ extensions to support nested task and data parallelism
- Fibonacci example

Sequential version

```
int fib (int n) {
    if (n < 2) return 1;
    else {
        int rst = 0;
        rst += fib (n-1);
        rst += fib (n-2);
        return rst;
    }
}
```

OpenMP version

```
int fib (int n) {
    if (n<2) return 1;
    else {
        int rst = 0;
        #pragma omp task
        {#pragma omp atomic
            rst += fib(n-1);}
         #pragma omp task
        {#pragma omp atomic
            rst += fib(n-2);}
    }
    #pragma omp taskwait
    return rst;
}
```

Basic Cilk Programming (4)

- C/C++ extensions to support nested task and data parallelism
- Fibonacci example

Sequential version

```
int fib (int n) {
    if (n < 2) return 1;
    else {
        int rst = 0;
        rst += fib (n-1);
        rst += fib (n-2);
        return rst;
    }
}
```

Cilk version

```
cilk int fib (int n) {
    if (n < 2) return 1;
    else {
        int rst = 0;
        rst += spawn fib (n-1);
        rst += spawn fib (n-2);
        sync;
        return rst;
    }
}
```

Basic Cilk Programming (4')

- C/C++ extensions to support nested task and data parallelism
- Fibonacci example

Sequential version

```
int fib (int n) {
    if (n < 2) return 1;
    else {
        int rst = 0;
        rst += fib (n-1);
        rst += fib (n-2);
        return rst;
    }
}
```

Cilk version

```
cilk int fib (int n) {
    if (n < 2) return 1;
    else {
        int rst = 0;
        rst += spawn fib (n-1);
        rst += spawn fib (n-2);
        return rst;
    }
}
```

Basic Cilk Programming (5)

- C/C++ extensions to support nested task and data parallelism
- Fibonacci example

•a cilk procedure
•capable of being spawned in parallel
•"main" function also needs to start with cilk keyword;

Cilk version

```
cilk int fib (int n) {
    if (n < 2) return 1;
    else {
        int rst = 0;
        rst += spawn fib (n-1);
        rst += spawn fib (n-2);
        sync;
        return rst;
    }
}
```

Basic Cilk Programming (6)

- C/C++ extensions to support nested task and data parallelism
- Fibonacci example

Cilk version

```
cilk int fib (int n) {
   if (n < 2) return 1;
   else {
      int rst = 0;
      rst += spawn fib (n-1);
      rst += spawn fib (n-2);
      sync;
      return rst;
   }
}
```

•**Child** cilk procedures, which can execute in parallel with the **parent** procedure.

Basic Cilk Programming (7)

- C/C++ extensions to support nested task and data parallelism
- Fibonacci example

Cilk version

```
cilk int fib (int n) {
   if (n < 2) return 1;
   else {
      int rst = 0;
      rst += spawn fib (n-1);
      rst += spawn fib (n-2);
      sync;
      return rst;
   }
}
```

•Wait until all the children have returned.
•only for the children of current parent; not global
•compiler would like to add an explicit sync before return.

Basic Cilk Programming (8)

- C/C++ extensions to support nested task and data parallelism
- Fibonacci example

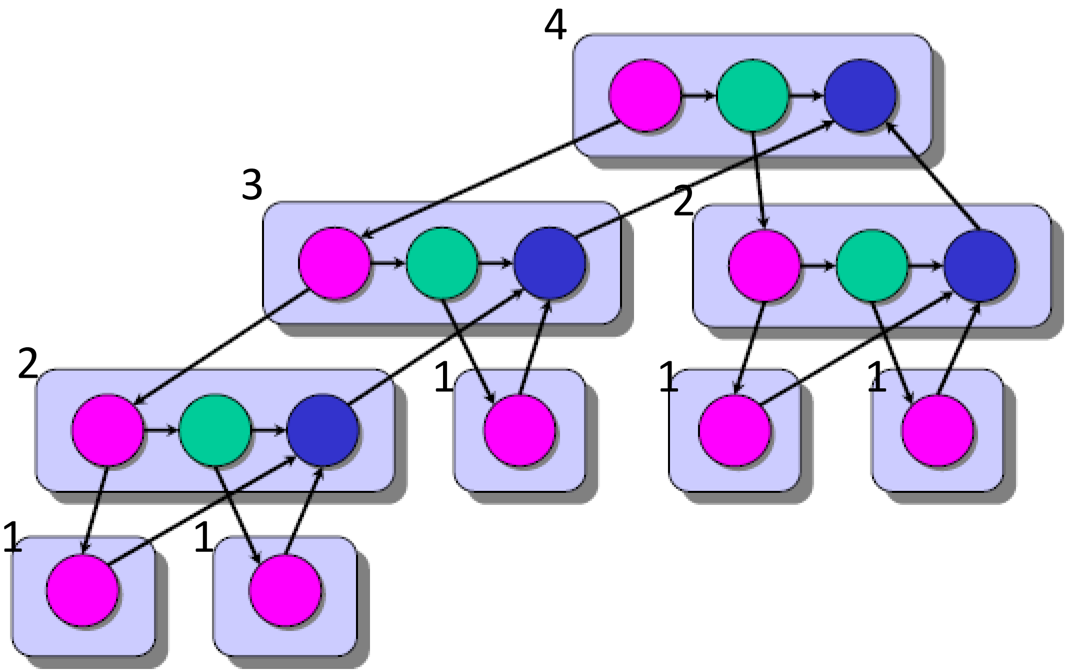
Cilk version

```
cilk int fib (int n) {
   if (n < 2) return 1;
   else {
      int rst = 0;
      rst += spawn fib (n-1);
      rst += spawn fib (n-2);
      sync;
      return rst;
   }
}
```

Execution Plan

Compute: fib (4)

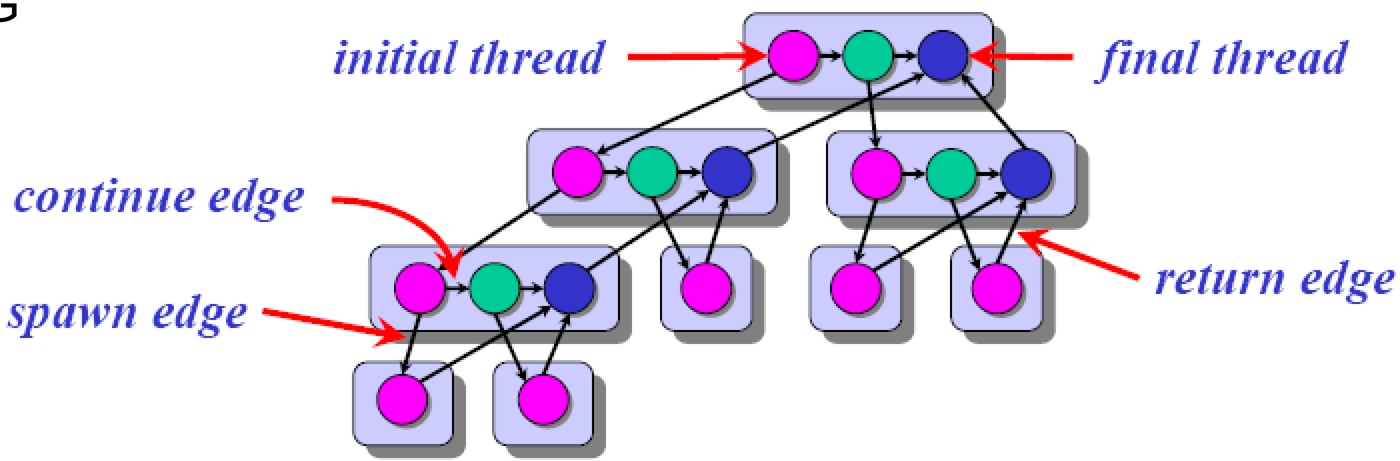
```
cilk fib (int n) {
 if (n < 2) return 1;
   else {
      int rst = 0;
      rst += spawn fib (n-1);
      rst += spawn fib (n-2);
     sync;
     return rst;
   }
}
```



Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

Performance Measurement (1)

- DAG

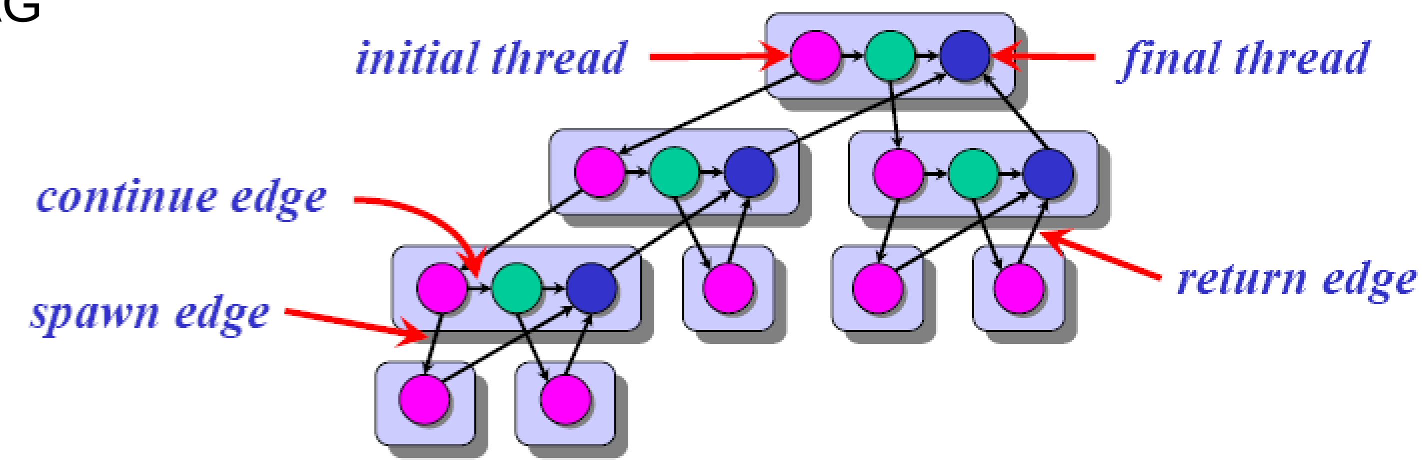


- **Cilk thread**: A maximal sequence of instructions that ends with a spawn, sync and return.

Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

Performance Measurement (2)

- DAG



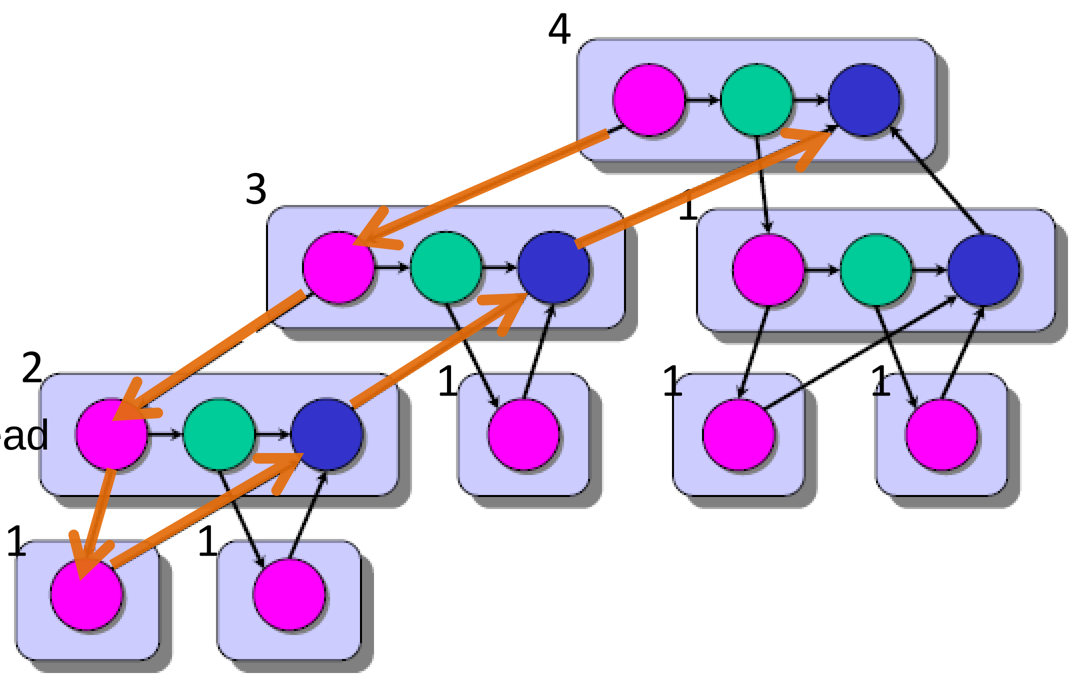
- Work: **W** = all the spawned Cilk threads
- Depth: **D** = critical path length; maximal sequential of instructions not containing parallel control (spawn, sync, return).
- Parallelism: W/D
- Execution time: **T**
  - T > W/P (P is the processors number)
  - T > D

Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

Performance Measurement (3)

- Performance Measurement
  - Work: 17
  - Depth: 8
  - Parallelism: 17/8 = 2.125
- execution time: >8
  - Runtime scheduling overhead

Compute: fib (4)



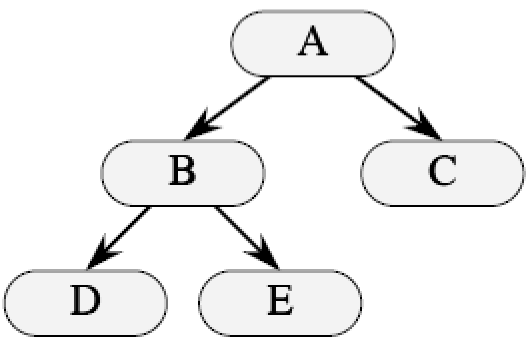
Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

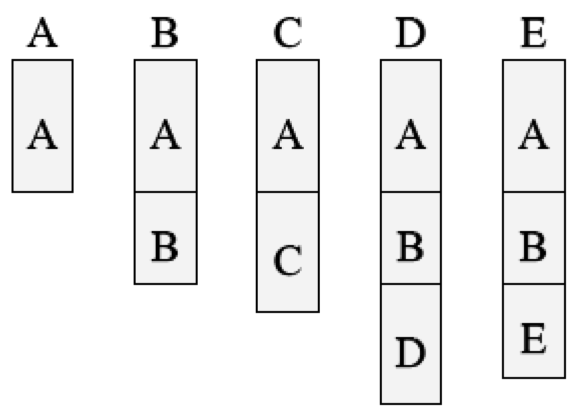
Storage Allocation

- Cilk supports both stacks and heaps.
- For stack, Cilk supports C's rule of pointers.
- Parents' pointer can be passed to children
- Children's pointers can not be passed to parents

```
ptr = Cilk_alloca(size);
```

- For heap, it works exactly as same a C
  - malloc(size); free()





Storage Allocation and Locks

- Cilk also supports global variables, like C.
- Cilk also has locks.

```
#include <cilk-lib.h>
:
Cilk_lockvar mylock;
:
{
    Cilk_lock_init(mylock);
:
    Cilk_lock(mylock); /* begin critical section */
:
:
    Cilk_unlock(mylock); /* end critical section */
}
```

inlet keyword (1)

- **Motivation**
  - children procedures need to return the value to the parent
  - guarantee those return values are incorporated into the parent's frame **atomically**
  - No lock is required to avoid data race

```
cilk int fib (int n)
{
        int rst = 0;
        inlet void summer (int result)
        {
                rst += result;
                return;
        }
        if (n<2) return n;
        else {
                summer(spawn fib (n-1));
                summer(spawn fib (n-2));
                sync;
                return (rst);
        }
}
```

inlet keyword (2)

- Some restrictions of “inlet”:
  - It can not contain spawn or sync statements.
  - Only the first argument to an inlet is spawned
  - Implicit inlets can be inserted by the compiler
  -

abort keyword

- **Motivation**
  - a procedure spawns off parallel work which it later discovers is unnecessary
  - Will not abort future spawned threads
  - Parallel search
  - Multiply zero

```
int product(int *A, int n) {
	int i, p=1;
	for (i=0; i<n; i++) {
		p *= A[i];
	}
	return p;
}
```

```
int product(int *A, int n) {
	int i, p=1;
	for (i=0; i<n; i++) {
		p *= A[i];
		if (p == 0) return 0;
	}
	return p;
}
```

Quit early if the partial product is 0

Programming Example

```
int product(int *A, int n) {
        int i, p=1;
        for (i=0; i<n; i++) {
                p *= A[i];
        }
        return p;
}
```

Programming Example

```
int product(int *A, int n) {
	int i, p=1;
	for (i=0; i<n; i++) {
		p *= A[i];
	}
	return p;
}
```

```
int prod(int *A, int n) {
	int p = 1;
	if (n == 1) {
		return A[0];
	} else {
		p *= prod(A, n/2);
		p *= prod(A+n/2, n-n/2);
		return p;
	}
}
```

Programming Example

```
int product(int *A, int n) {
    int i, p=1;
    for (i=0; i<n; i++) {
        p *= A[i];
    }
    return p;
}
```

```
cilk int prod(int *A, int n) {
    int p = 1;
    if (n == 1) {
        return A[0];
    } else {
        p *= spawn prod(A, n/2);
        p *= spawn prod(A+n/2, n-n/2);
        sync;
        return p;
    }
}
```

Programming Example

```
int product(int *A, int n) {
	int i, p=1;
	for (i=0; i<n; i++) {
		p *= A[i];
	}
	return p;
}
```

```
cilk int prod(int *A, int n) {
	int p = 1;
	inlet void mult(int x) {
		p *= x;
		return;
	}
	if (n == 1) {
		return A[0];
	} else {
		mult(spawn prod(A, n/2));
		mult(spawn prod(A+n/2, n-n/2));
		sync;
		return p;
	}
}
```

Programming Example

```
int product(int *A, int n) {
        int i, p=1;
        for (i=0; i<n; i++) {
                p *= A[i];
        }
        return p;
}
```

```
cilk int prod(int *A, int n) {
        int p = 1;
        inlet void mult(int x) {
                p *= x;
                if (p == 0) abort;
                return;
        }
        if (n == 1) {
                return A[0];
        } else {
                mult(spawn prod(A, n/2));
                mult(spawn prod(A+n/2, n-n/2));
                sync;
                return p;
        }
}
```

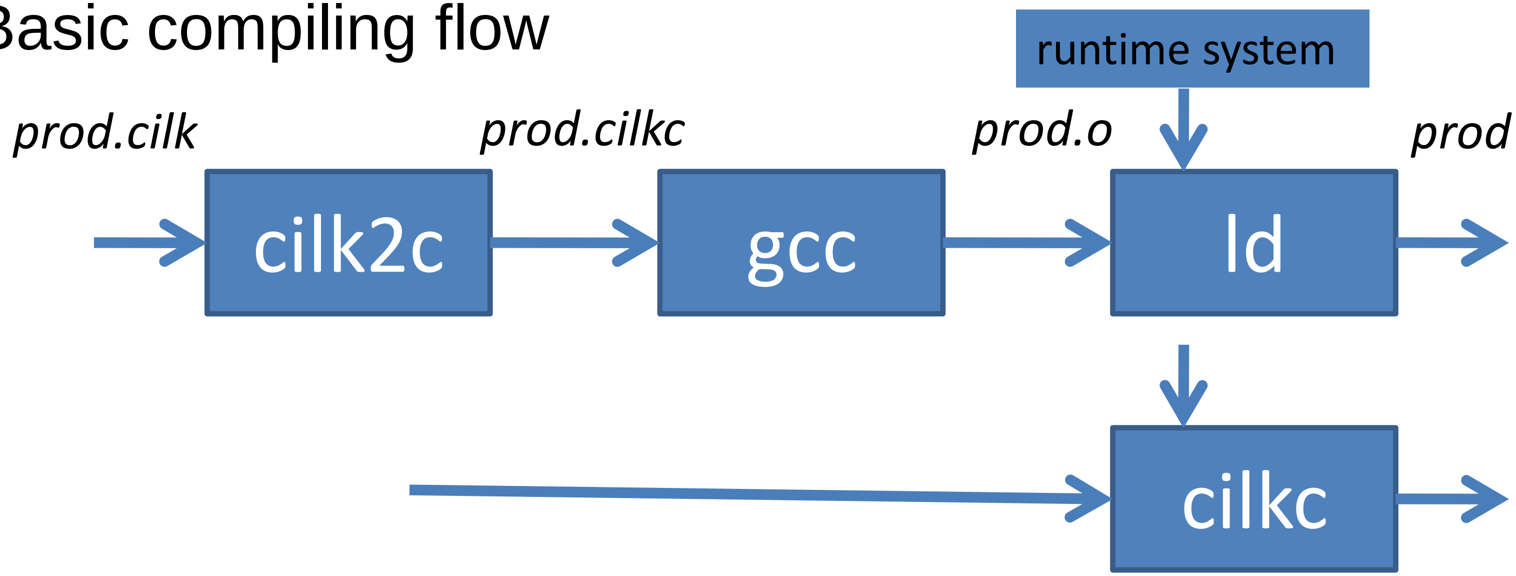
Programming Example

```
int product(int *A, int n) {
        int i, p=1;
        for (i=0; i<n; i++) {
                p *= A[i];
        }
        return p;
}
```

```
cilk int prod(int *A, int n) {
        int p = 1;
        inlet void mult(int x) {
                p *= x;
                if (p == 0) abort;
                return;
        }
        if (n == 1) {
                return A[0];
        } else {
                mult(spawn prod(A, n/2));
                if (p == 0) return 0;
                mult(spawn prod(A+n/2, n-n/2));
                sync;
                return p;
        }
}
```

How compile and execute?

- Basic compiling flow



- ./cilkc -[options] filename.cilk

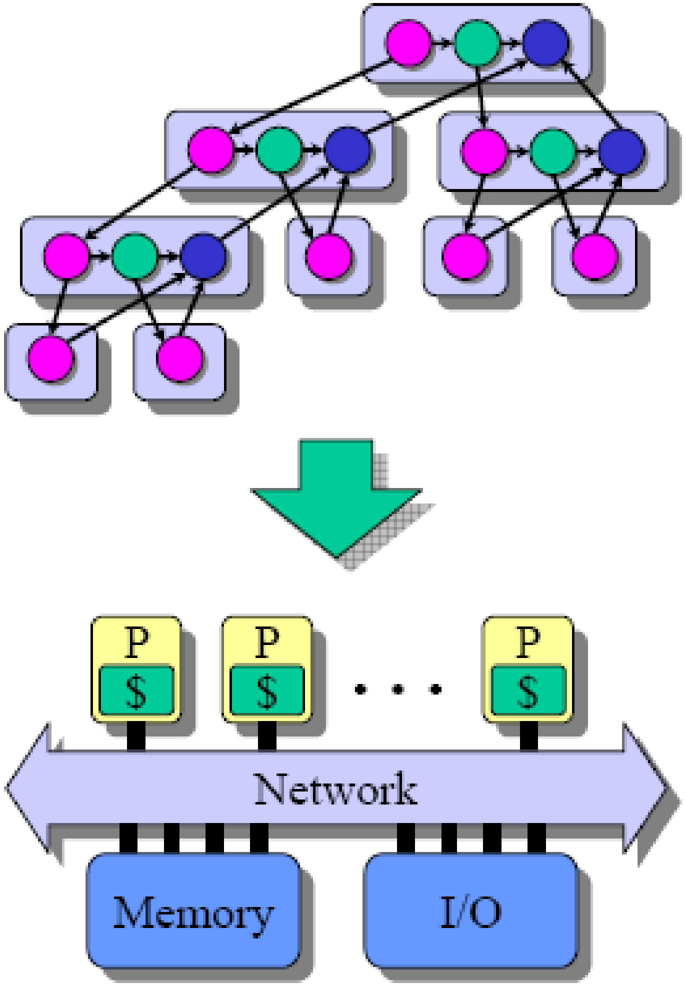
- ./filename --nproc THRDNUM <arguments>

Outline

- Introduction
- Basic Cilk programming
- Cilk runtime system support
- Conclusion

Scheduling

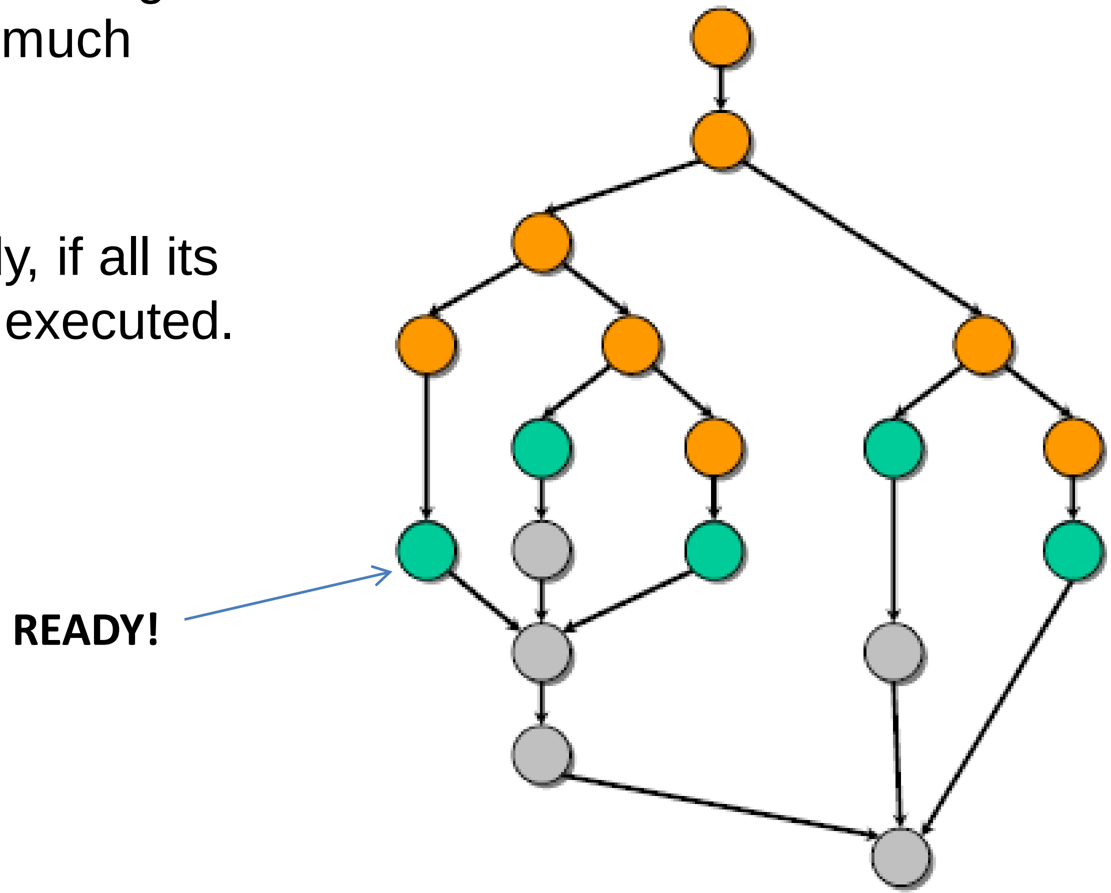
- The **cilk scheduler** maps Cilk threads onto processors dynamically at run-time.



Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

Run-time Schedule (1)

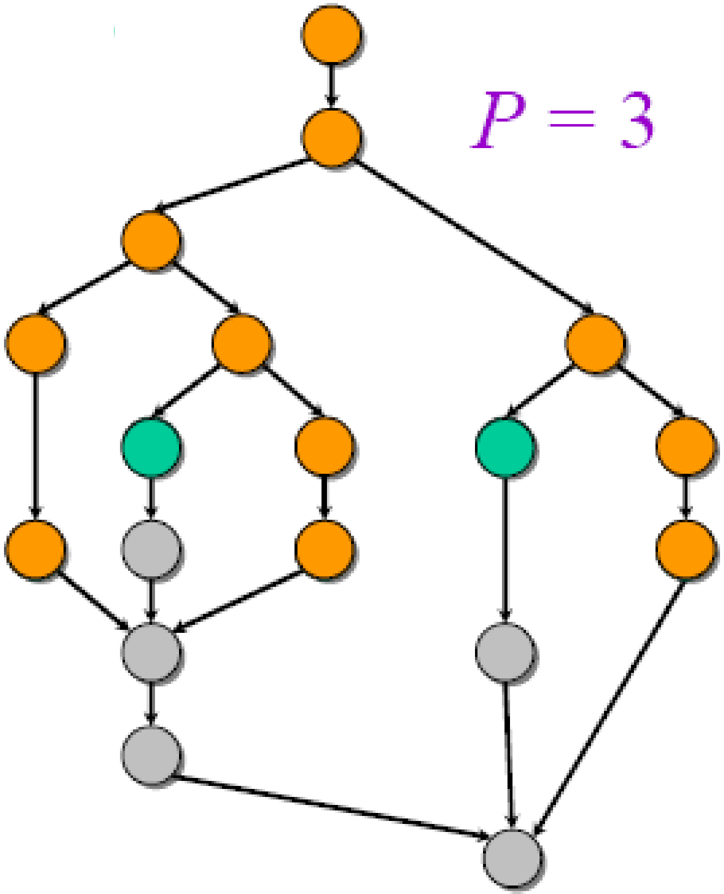
- Using a greedy scheduling – in each step, do as much work as possible
- A cilk thread is ready, if all its predecessors have executed.



Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

Run-time Schedule (2)

- Complete step:
- >= P threads is ready
- Pick up any **P threads** to run



Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

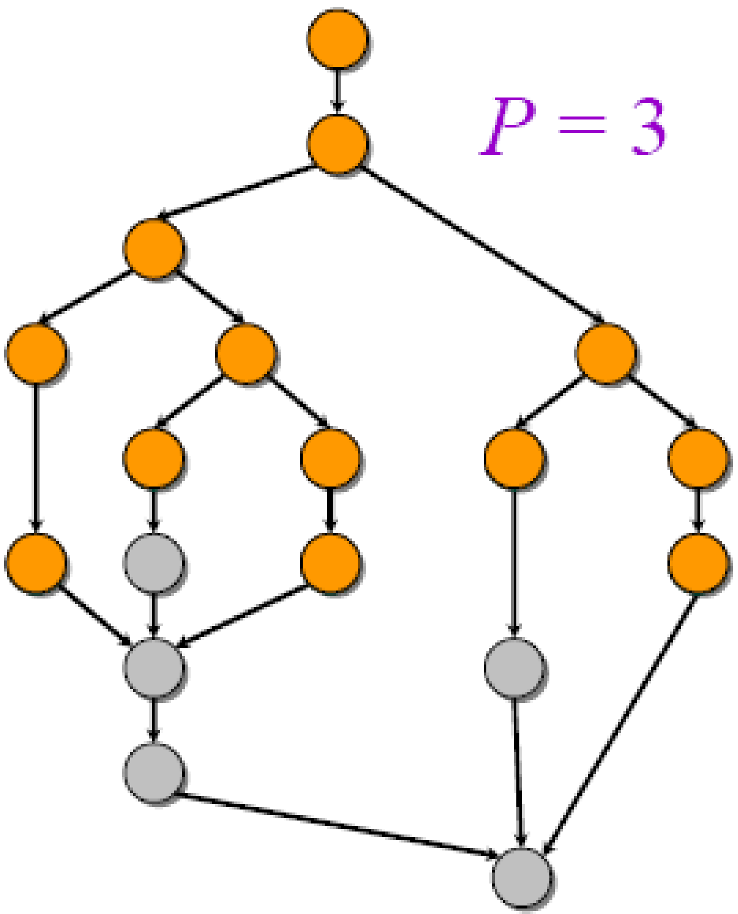
Run-time Schedule (3)

- Complete step:
- >= P threads is ready
- Pick up any **P threads** to run

- Incomplete step:
- < P threads is ready
- Run all of them

- Theoretically, a greedy scheduling can achieve performance:

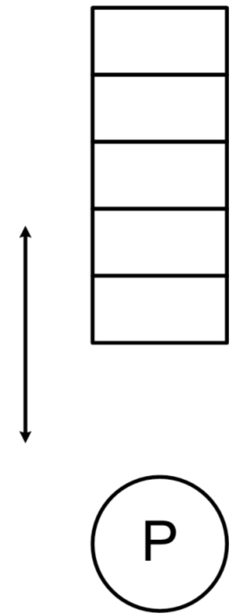
  T = W/P + D



Source: http://supertech.csail.mit.edu/cilk/lecture-1.pdf

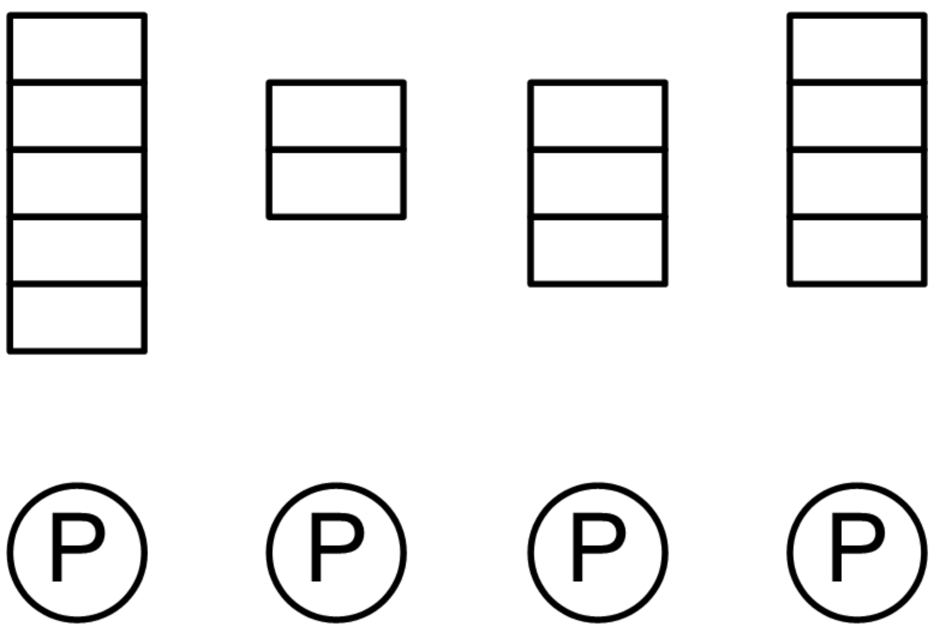
How does it implement?

- The **cilk2c** generates two clones for one cilk procedure
  - **Fast clone**
    - Initials a frame structure
    - Saves live variables in the frame
    - Pushes it on the bottom of a **deque**
    - Does the computation
    - Always spawned
  - **Slow clone**
    - Executes if a thread is stolen
    - Restores the live variables
    - Does the computation
  - A checker is made whenever a procedure returns to see if the resuming parent has been stolen.



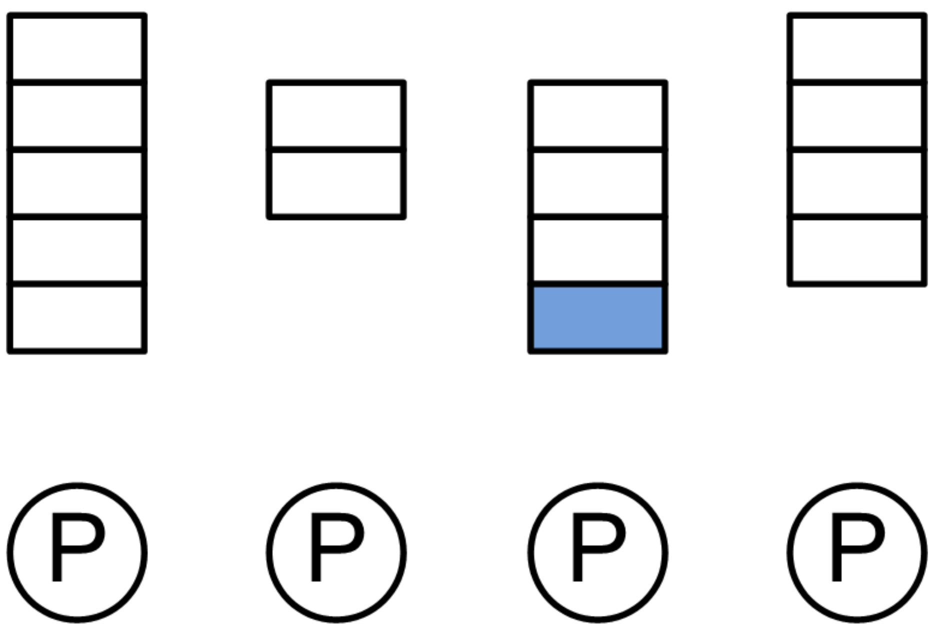
Scheduling - deques

- Each processor maintains a work (dual-end queue) deque of ready threads, and it manipulates the bottom of the deque like a stack.



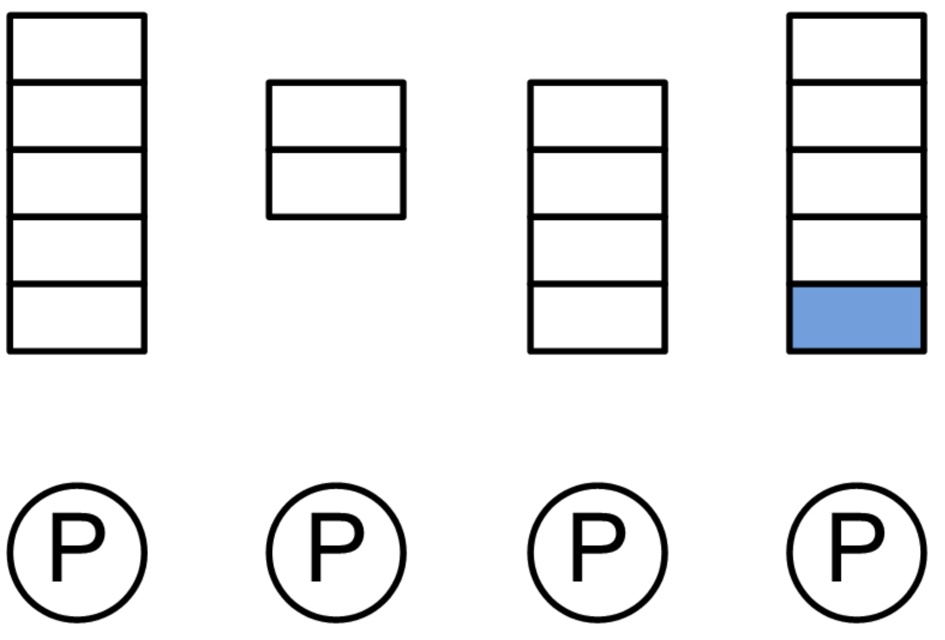
Scheduling - spawn

- Each processor maintains a work (dual-end queue) deque of ready threads, and it manipulates the bottom of the deque like a stack.



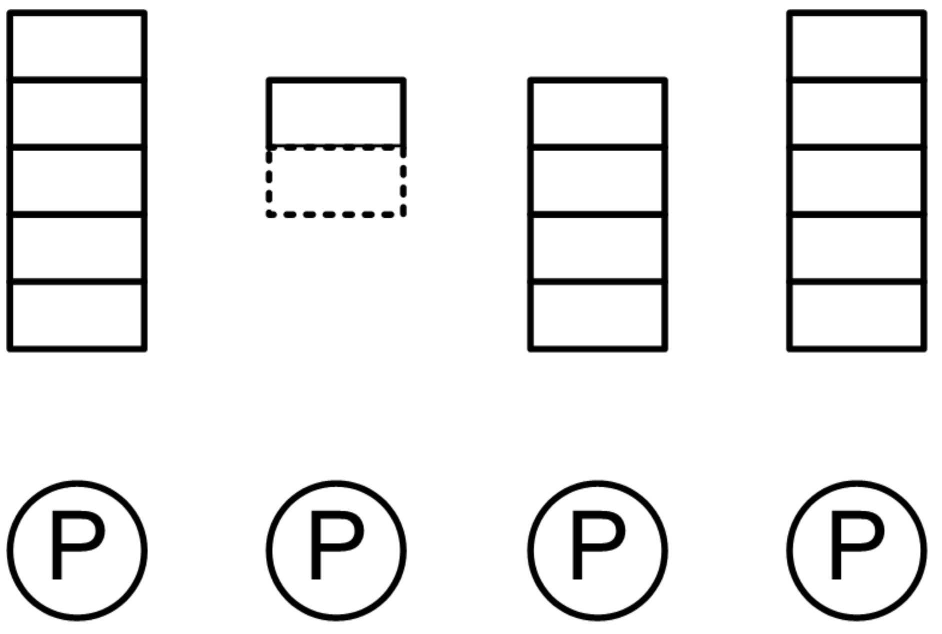
Scheduling - spawn

- Each processor maintains a work (dual-end queue) deque of ready threads, and it manipulates the bottom of the deque like a stack.



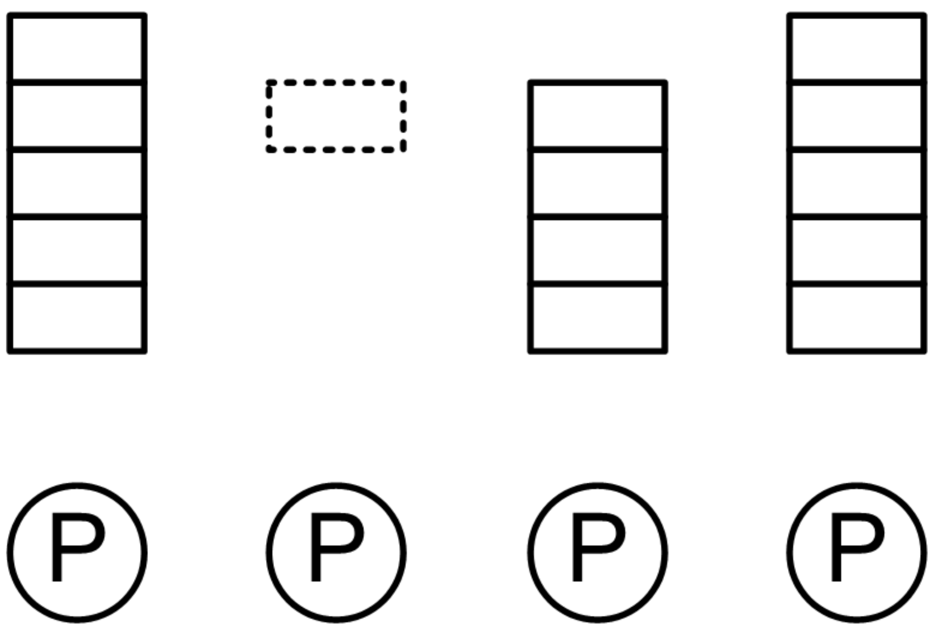
Scheduling - return

- Each processor maintains a work (dual-end queue) deque of ready threads, and it manipulates the bottom of the deque like a stack.



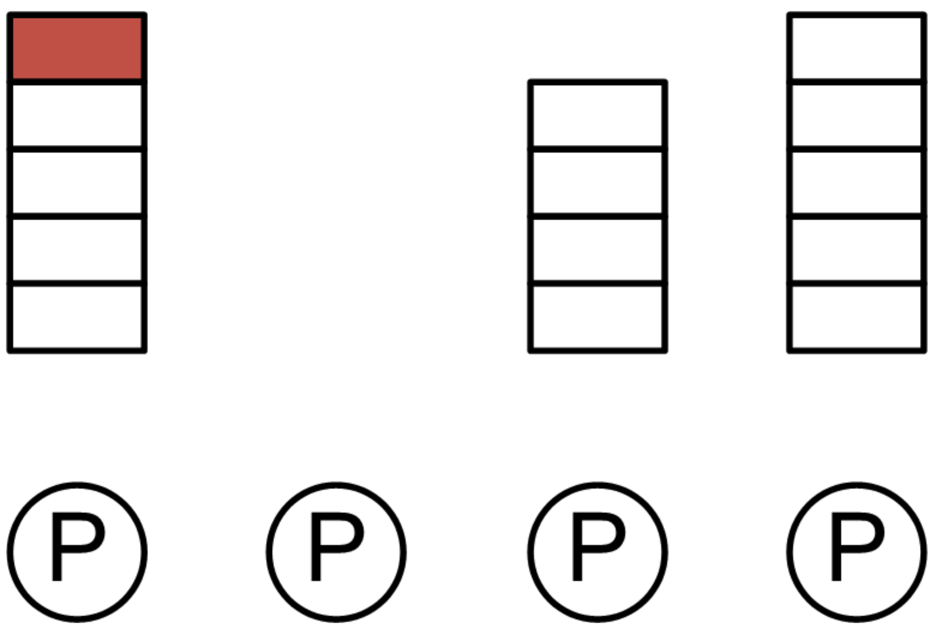
Scheduling - return

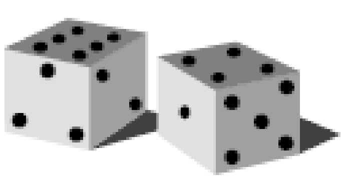
- Each processor maintains a work (dual-end queue) deque of ready threads, and it manipulates the bottom of the deque like a stack.



Scheduling - stealing

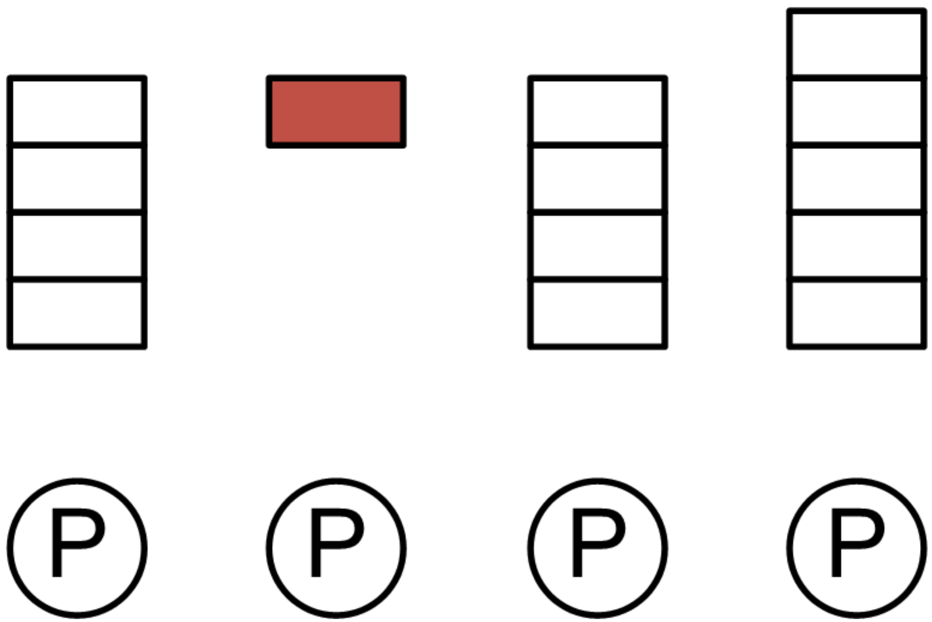
- Each processor maintains a work (dual-end queue) deque of ready threads, and it manipulates the bottom of the deque like a stack.





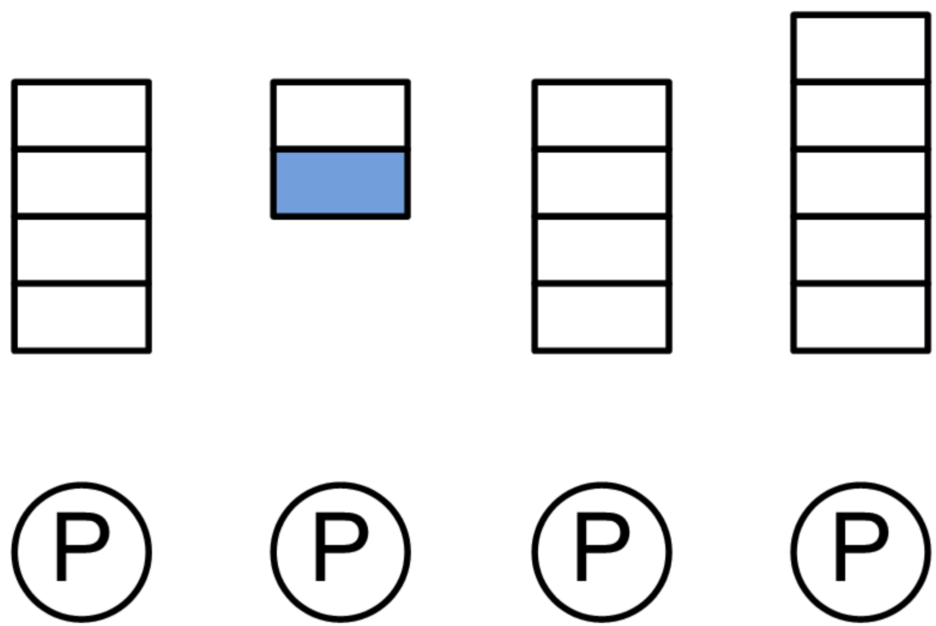
Scheduling - stealing

- Each processor maintains a work (dual-end queue) deque of ready threads, and it manipulates the bottom of the deque like a stack.

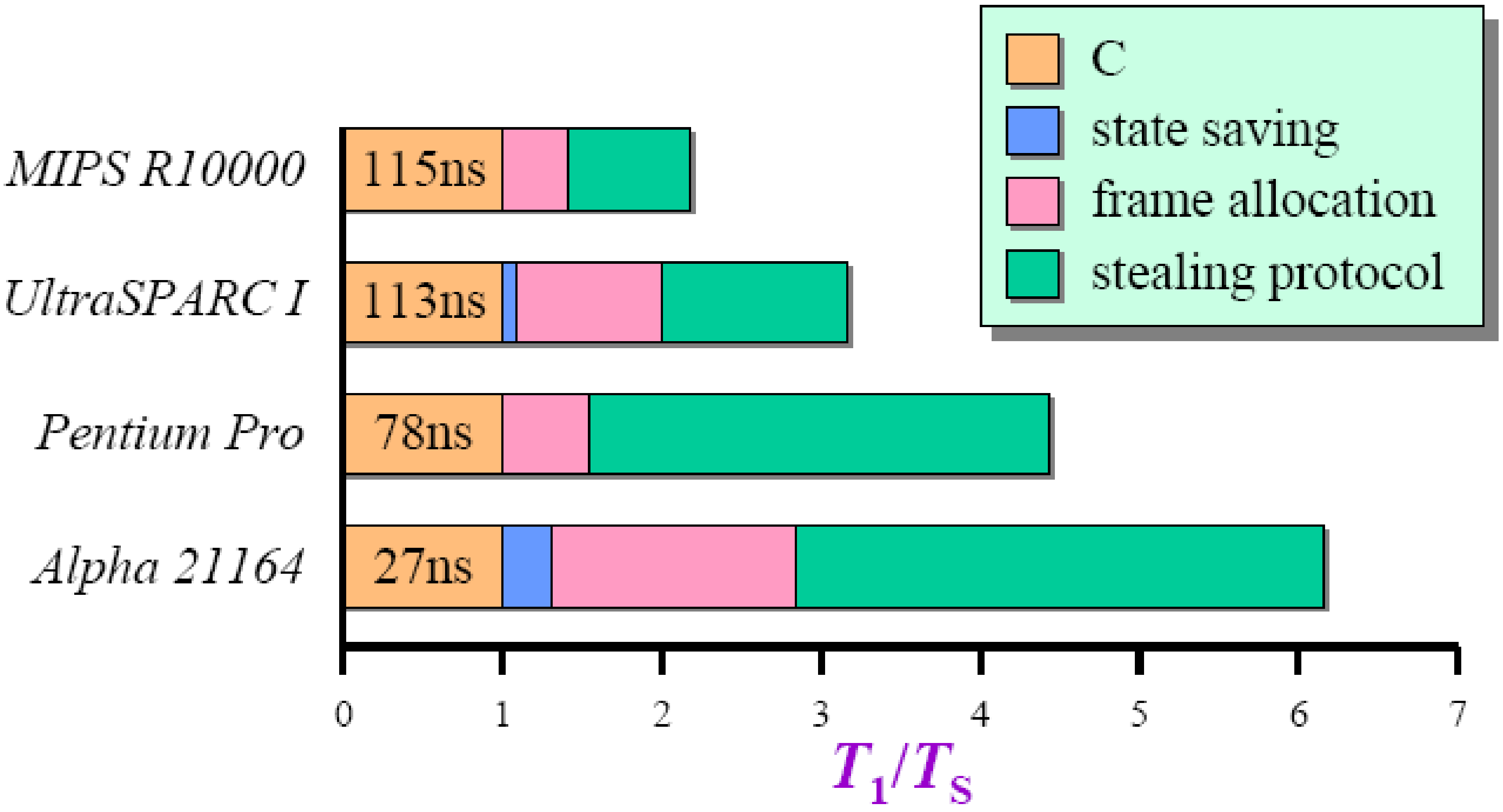


Scheduling - spawn

- Each processor maintains a work (dual-end queue) deque of ready threads, and it manipulates the bottom of the deque like a stack.



Work overhead



Source: http://supertech.csail.mit.edu/cilk/lecture-3.pdf

Conclusion

- Cilk programming is simple
- Cilk compiler translates the .cilk source code to a .c code
- Cilk runtime system can guarantee the performance

Reference

[1] Cilk: An Efficient Multithreaded Runtime System - R. D. Blumofe et al.
http://supertech.csail.mit.edu/papers/PPoPP95.pdf

[2] The latest Cilk-5.4.6 Reference Manual-
http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf

[3] Cilk lectures notes - Charles Leiserson and Bradley Kuszmaul
http://supertech.csail.mit.edu/cilk/lecture-1.pdf
http://supertech.csail.mit.edu/cilk/lecture-2.pdf
http://supertech.csail.mit.edu/cilk/lecture-3.pdf