Multithreaded Programming in Cilk

ACM Supercomputing 2007

Workshop on Manycore and Multicore Computing

November 11, 2007

Charles E. Leiserson

CILK ARTS

- Incorporated in 2006 to commercialize 15 years of MIT research on **Cilk** (pronounced "**SILK**").
- Headquartered in Lexington, Massachusetts.
- Venture funded led by Stata Venture Partners. Additional support from an NSF SBIR Award.
- Currently seeking alpha and beta design partners for our first product, **Cilk++**.

The Multicore Software Problem

- 950,000 software engineers and programmers work in the United States.
  — *U.S. Bureau of Labor Statistics, 2006*
- A negligible fraction know how to program parallel computers.
- Enormous legacy investment in serial programming technology and training.

"[Multicore] could become the biggest software *remediation* task of this decade."
— *Gartner Group, January 31, 2007*

Three Key Issues

Development Time

- How can we get our product out in time?
- Where will we be able to find enough parallel-programming talent?
- Will we be forced to redesign our applications?

Application Performance

- How can we minimize response time?
- Will our solution scale as the number of processor cores increases?

Software Reliability

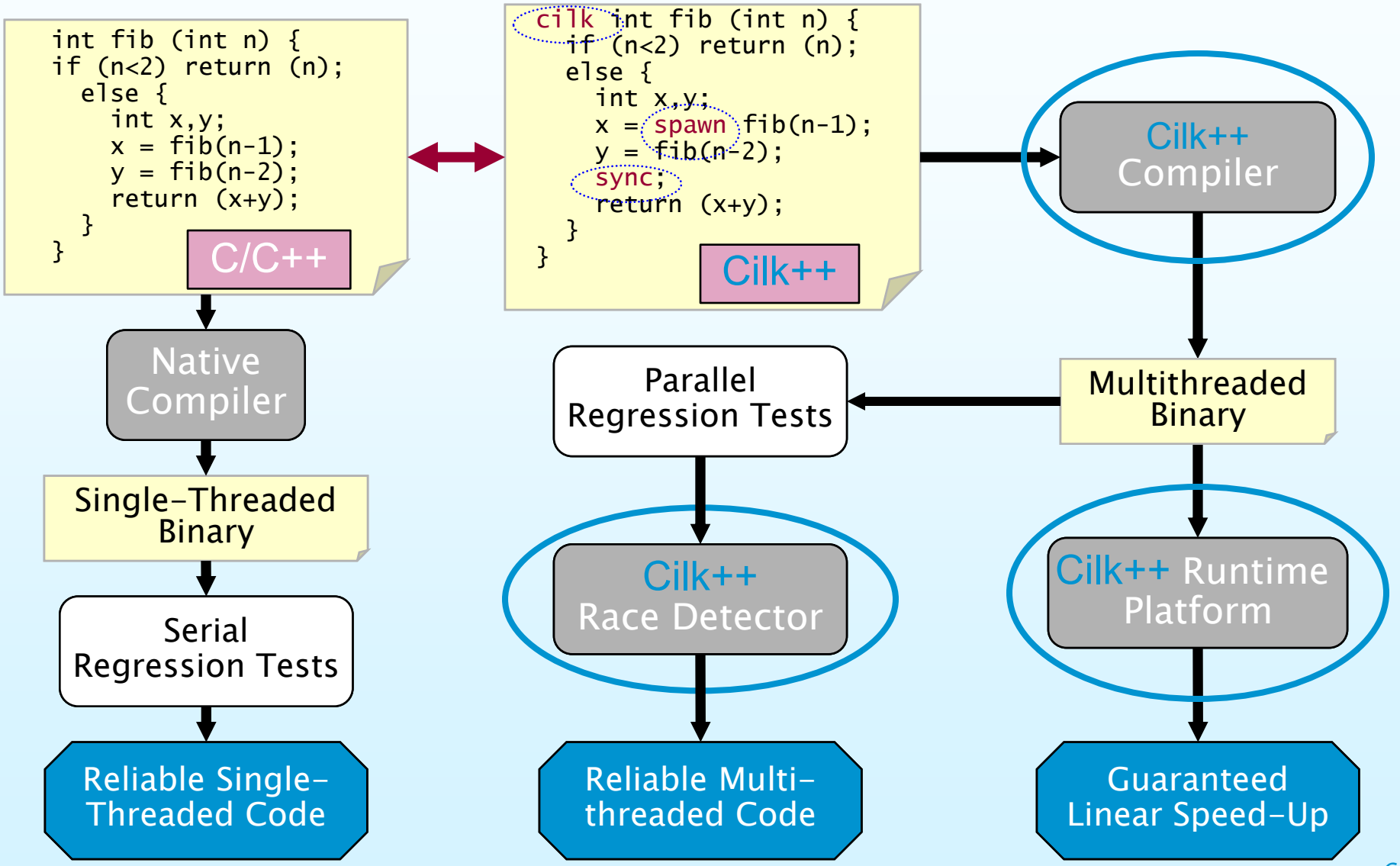
- How can we debug and maintain our applications?
- How will we regression-test before release?

What is Cilk?

Cilk is a ***remarkably simple*** set of extensions for C/C++ and other languages and a powerful ***runtime platform*** for multicore applications.

*Cilk provides a smooth **evolution** from serial programming to parallel programming, because Cilk parallel programs retain serial semantics.*

The Cilk++ Solution



Outline

- Introduction
- **Cilk++** Extensions
- Runtime Platform
- Race Detector
- Case Study
- Conclusion

Cilk++ Keywords

```
template <typename T>
cilk void qsort(T begin, T end) {
  if (begin != end) {
    T middle = partition(
                  begin,
                  end,
                  bind2nd( less<typename iterator_traits<T>::value_type>(),
                           *begin )
                );
    spawn qsort(begin, middle);
    qsort(max(begin + 1, middle), end);
    sync;
  }
}
```

The function contains parallel control constructs.

The named *child* **Cilk++** function can execute in parallel with the *parent* caller.

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

SP-reciprocity

Cilk++ provides two ways to invoke a function:

- calling
- spawning

```
template <typename T>
cilk void qsort(T begin, T end) {
  if (begin != end) {
     T middle = partition(
                    begin,
                    end,
                    bind2nd( less<typename iterator_traits<T>::value_type>(),
                             *begin )
                  );
     spawn qsort(begin, middle);
     qsort(max(begin + 1, middle), end);
     sync;
  }
}
```

Cilk++ and C/C++ interoperate seamlessly.
Arbitrary statement blocks can also be spawned.

Cilk++ Loops

```
cilk for ( T v = begin; v < end; v++)
  {
    statement1;
    statement2;
    ...
  }
```

- A Cilk++ loop's iterations execute in parallel.
- The loop index can be an arbitrary C++ random-access iterator.
- A P-processor execution consumes at most P times the stack space of a 1-processor execution, no matter how many iterations in the loop.

Global Variables

- Global variables inhibit parallelism by inducing *data races*.
- *Locking* can "solve" data races, but *lock contention* can destroy all parallelism.
- Making *local copies* of the global variables can remove contention, but at the cost of restructuring program logic.
- Cilk++ provides a feature to handle races on global variables efficiently without locking or code restructuring.

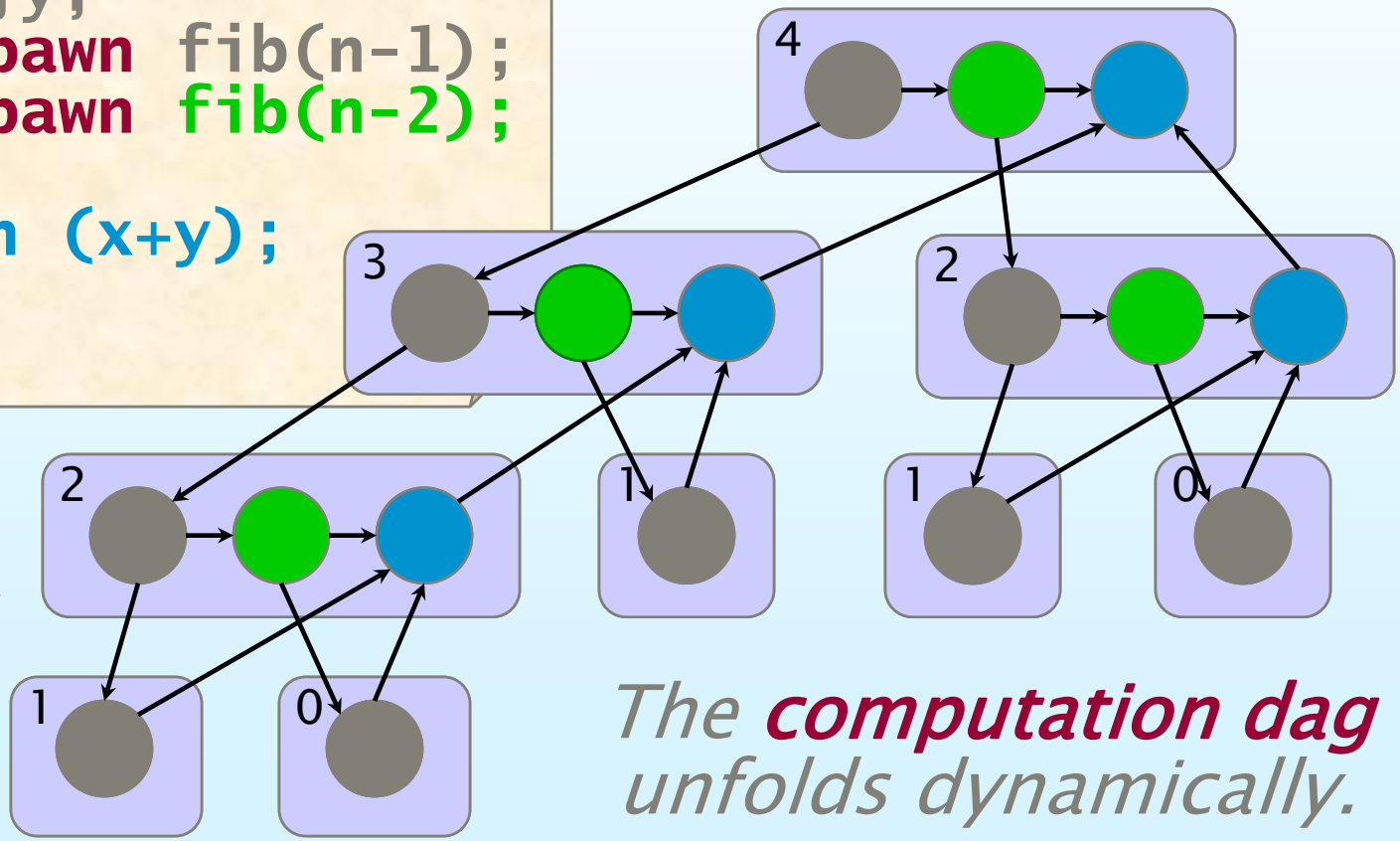
Outline

- Introduction
- Cilk++ Extensions
- Runtime Platform
- Race Detector
- Case Study
- Conclusion

Dynamic Multithreading

```
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);
  }
}
```

Example: fib(4)

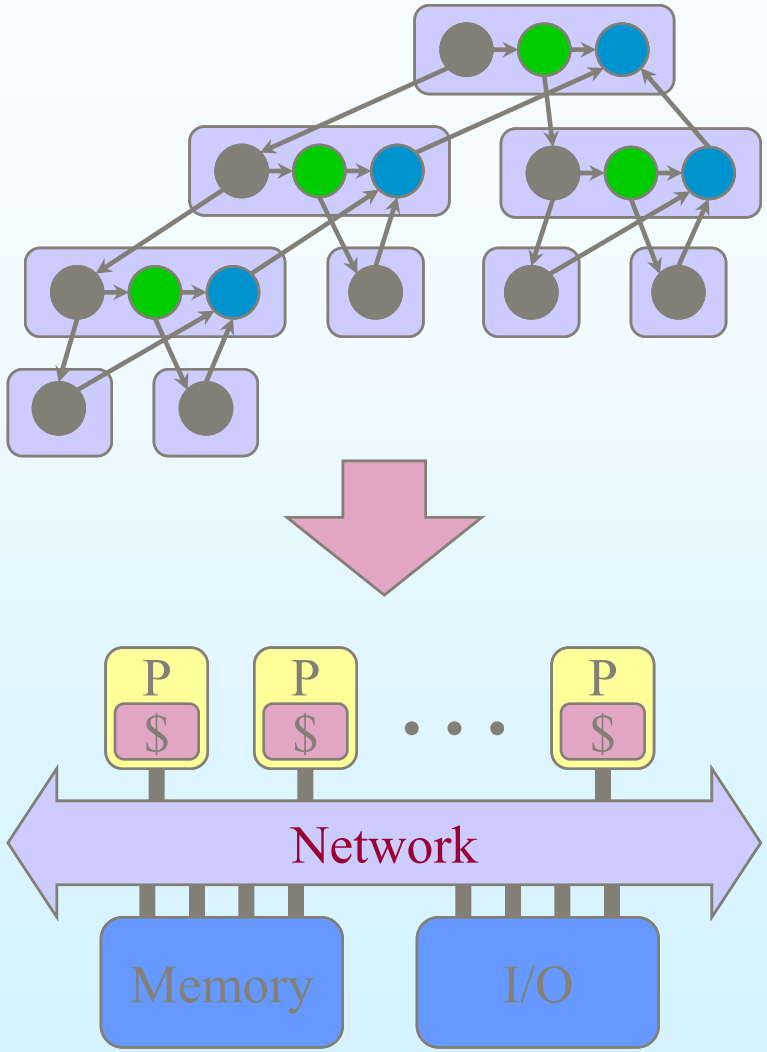


"Processor oblivious"

*The **computation dag** unfolds dynamically.*

Scheduling

- Cilk++ allows the programmer to express *potential* parallelism in an application.
- The Cilk++ *runtime platform* maps Cilk threads onto available processors dynamically as the application executes.



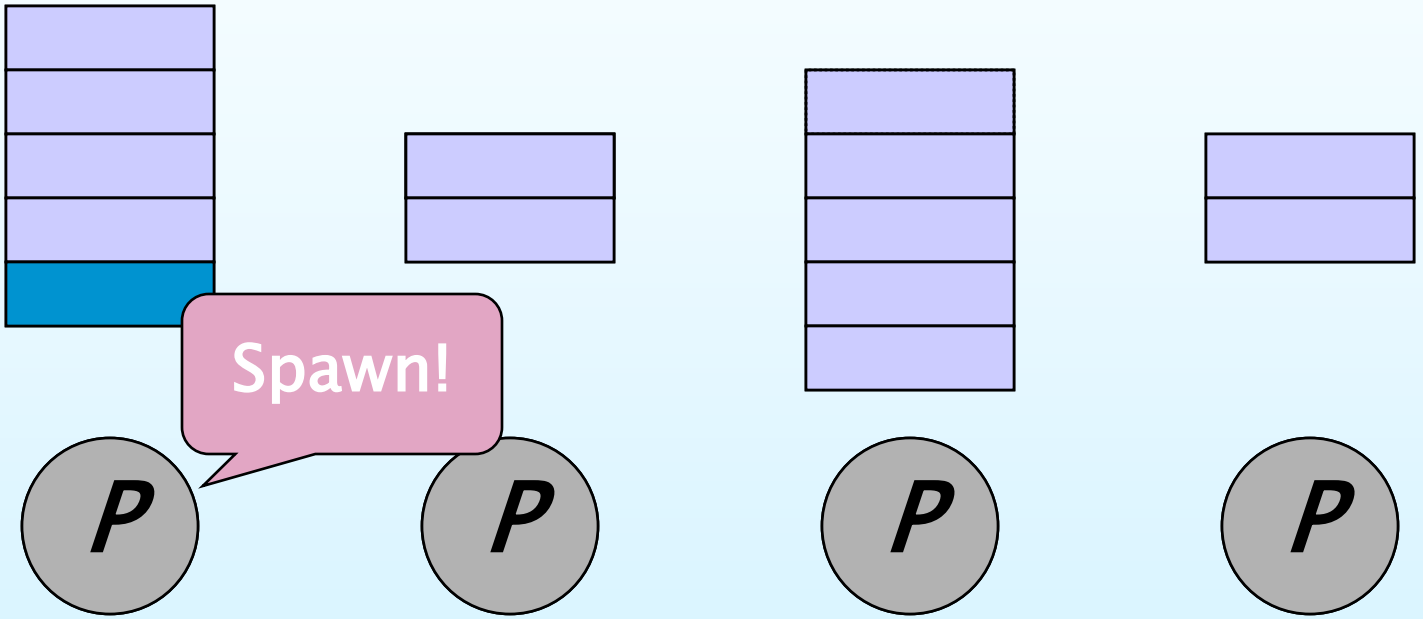
Cilk++ Runtime Overheads

```
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);
  }
}
```

A `spawn/return` is over 450 times faster than a Pthread `create/exit` — less than 3 times slower than an ordinary C function call. On one processor, Cilk++ overhead typically measures less than 1–2%.

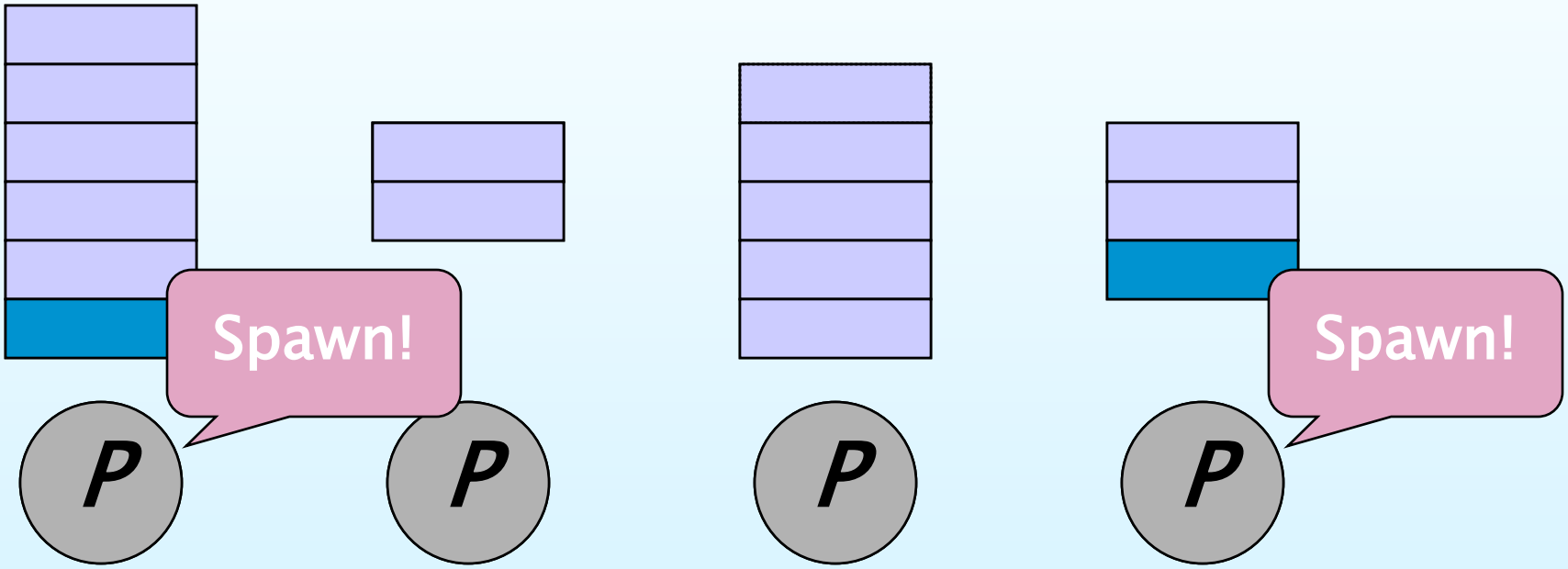
Work-Stealing Scheduler

Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



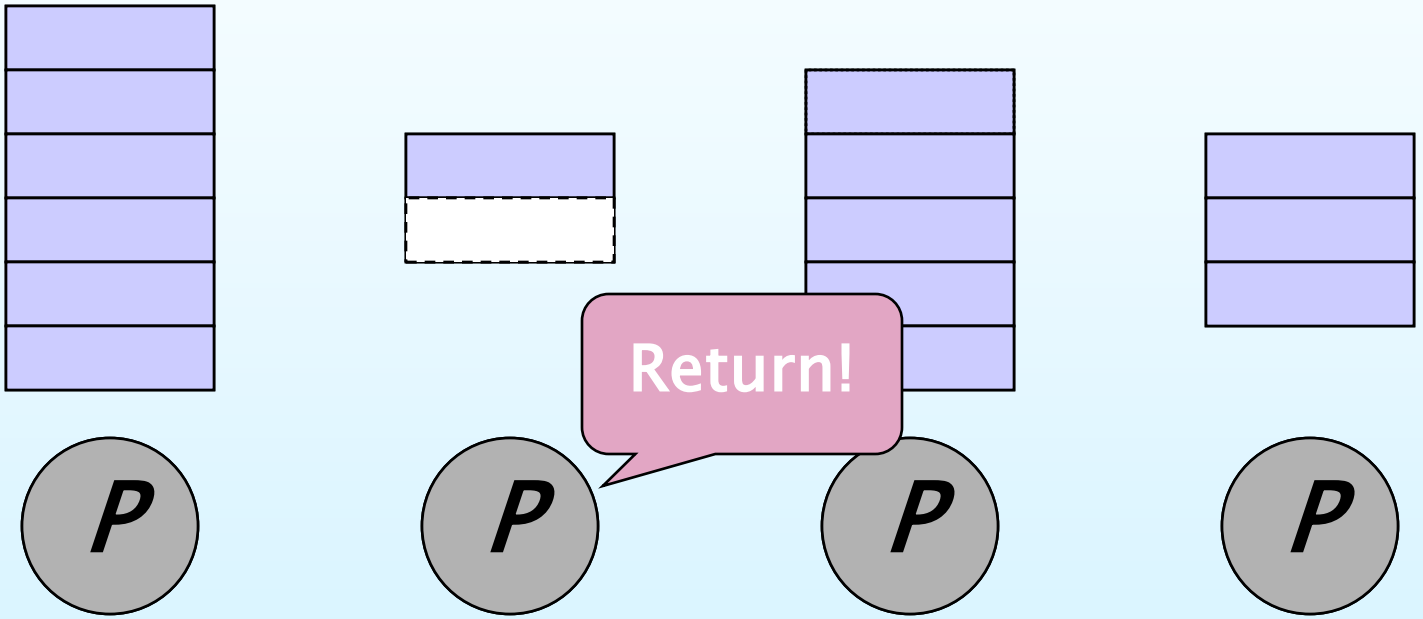
Work-Stealing Scheduler

Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



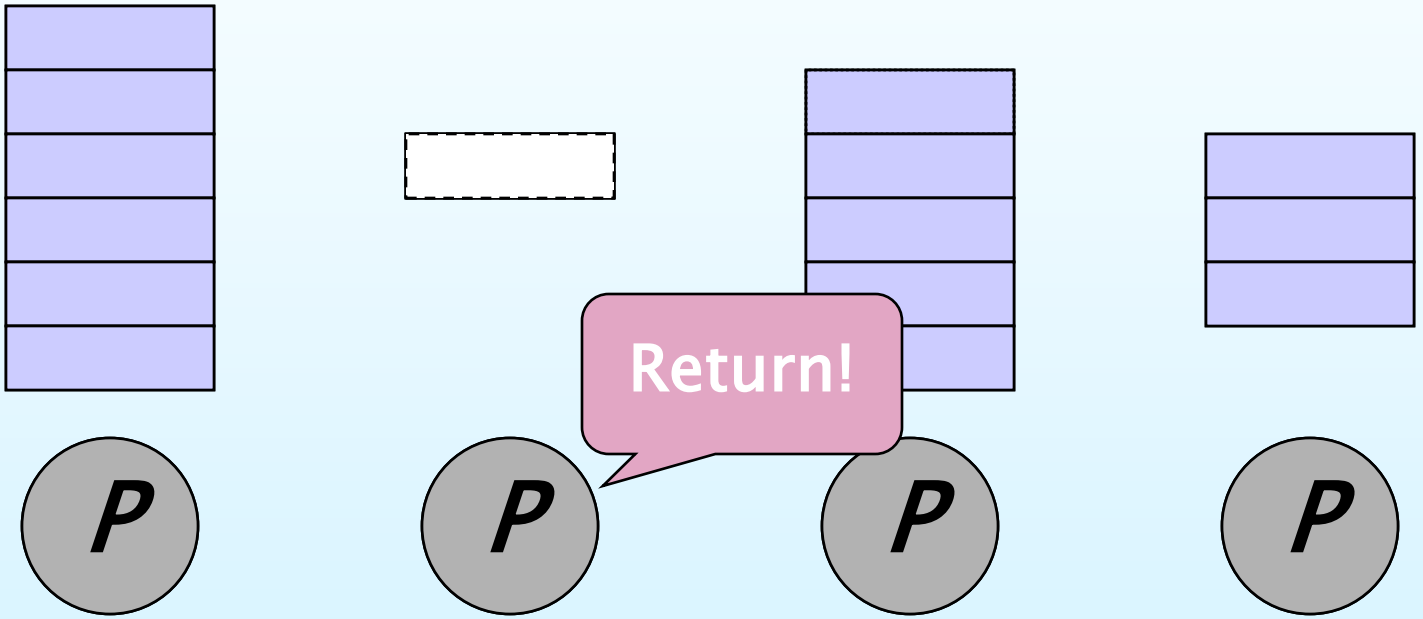
Work-Stealing Scheduler

Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



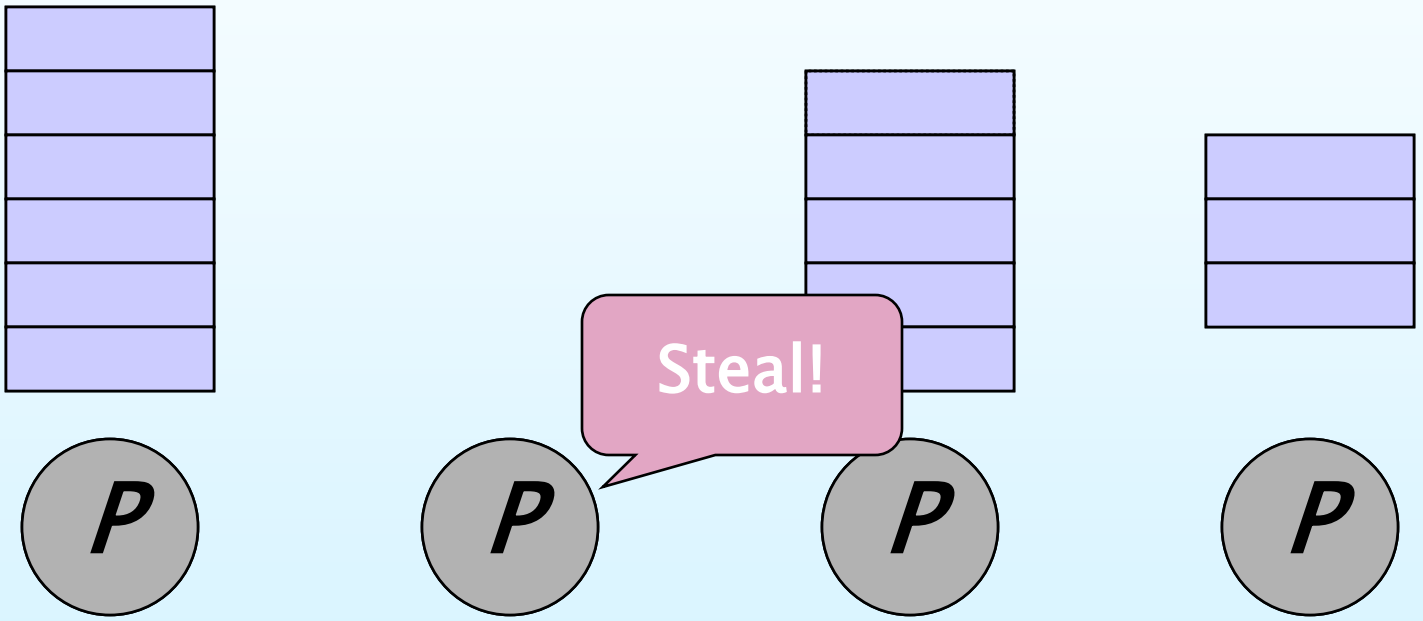
Work-Stealing Scheduler

Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



Work-Stealing Scheduler

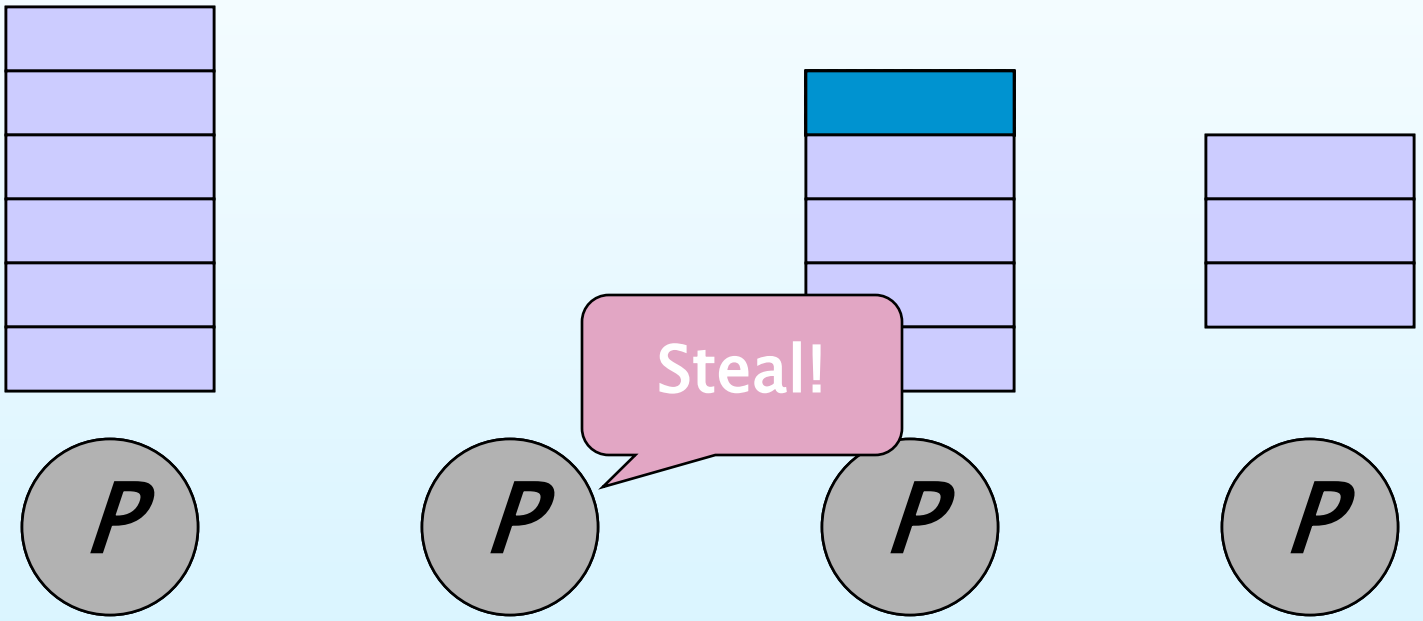
Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



When a worker runs out of work, it ***steals*** from the top of a victim's deque.

Work-Stealing Scheduler

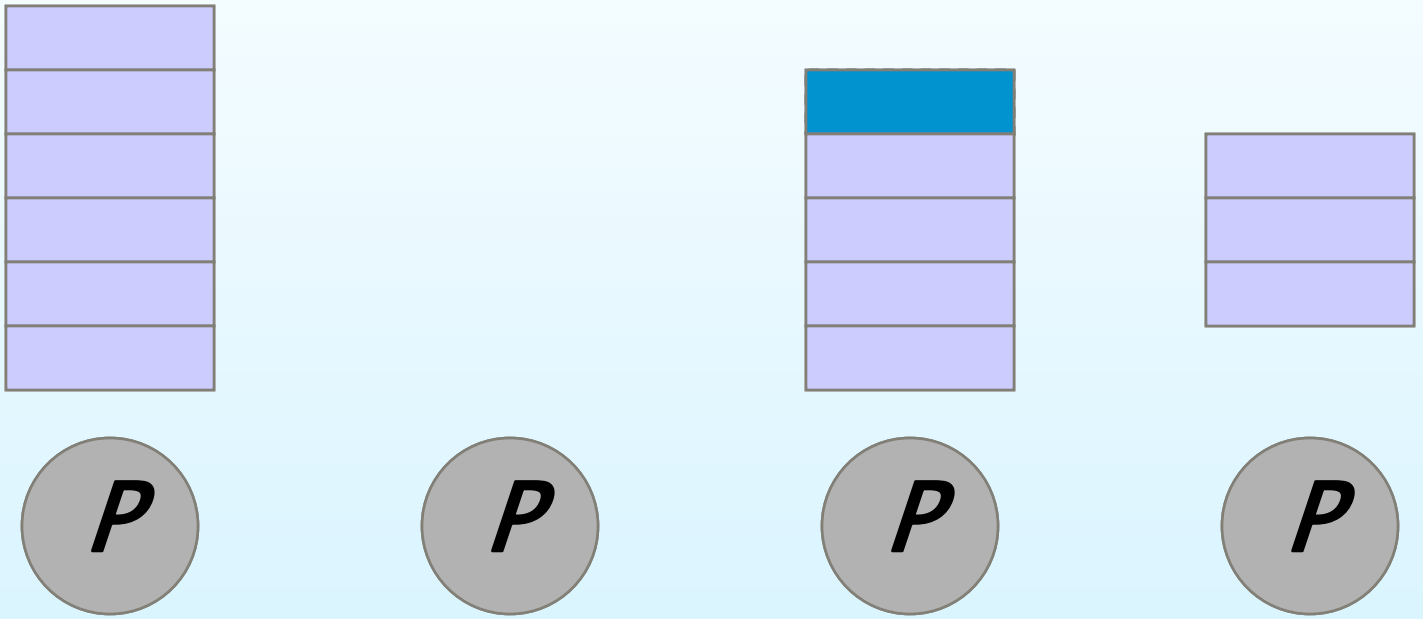
Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



When a worker runs out of work, it ***steals*** from the top of a victim's deque.

Work-Stealing Scheduler

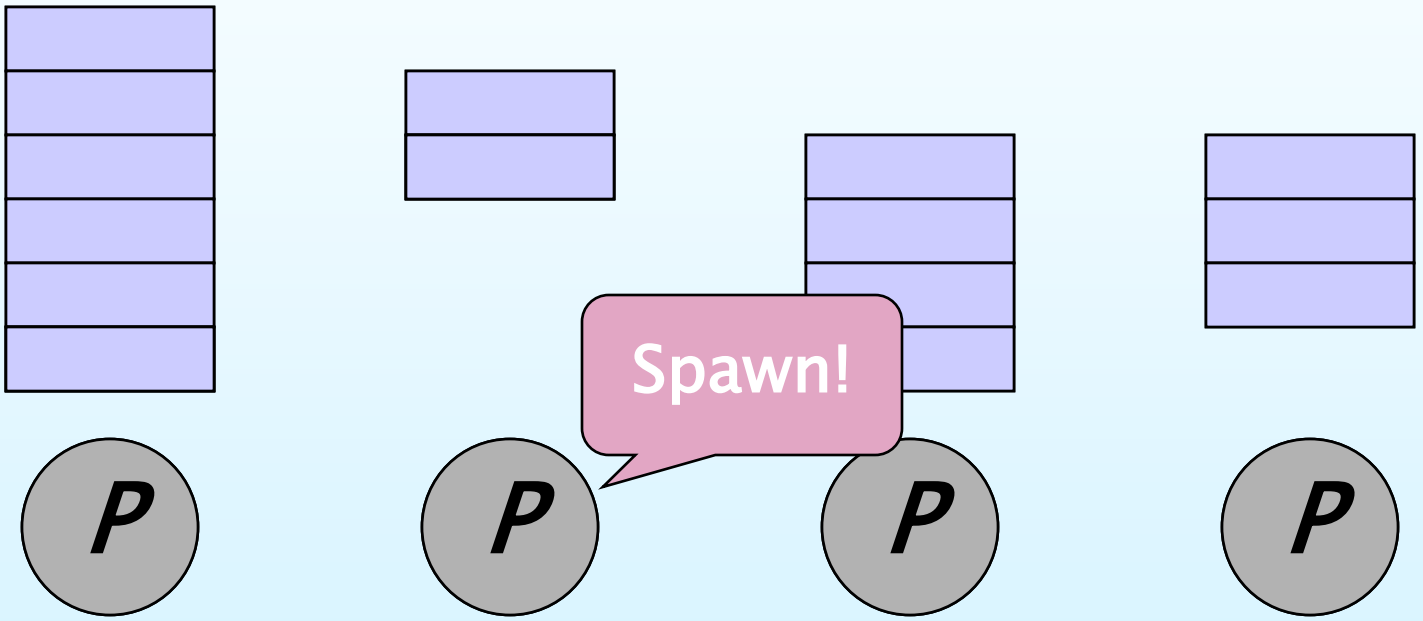
Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



When a worker runs out of work, it ***steals*** from the top of a victim's deque.

Work-Stealing Scheduler

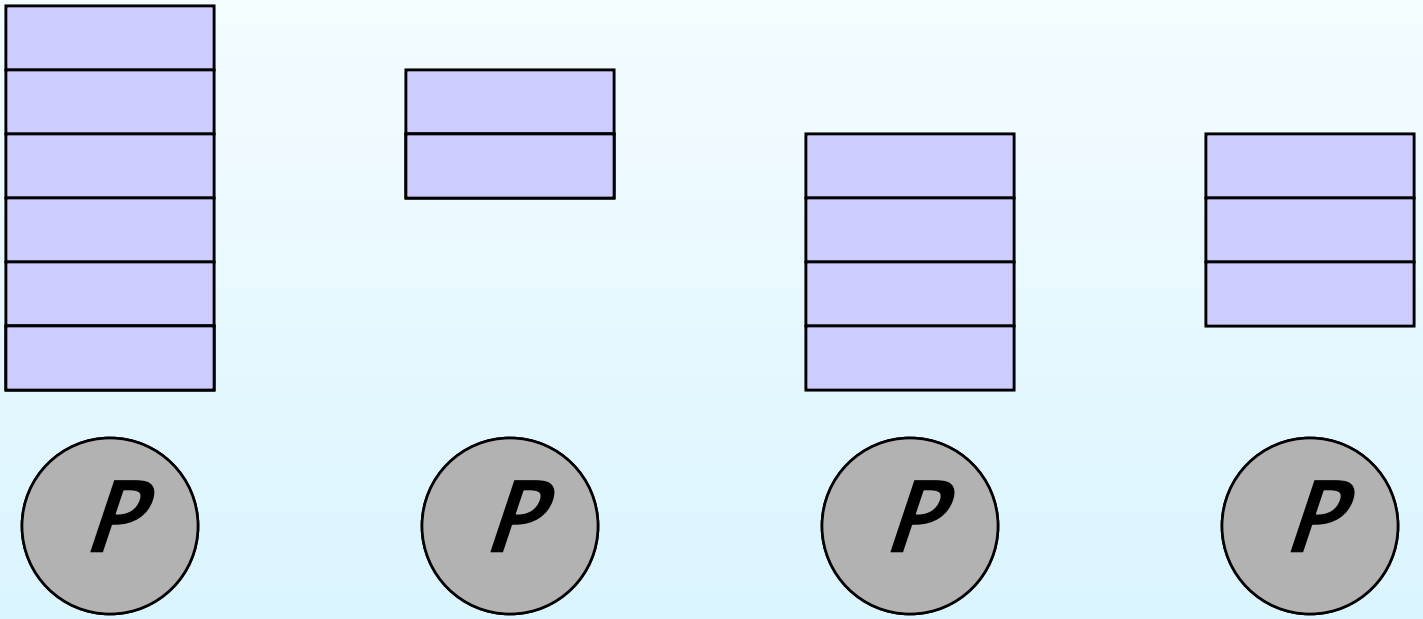
Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



When a worker runs out of work, it ***steals*** from the top of a victim's deque.

Work-Stealing Scheduler

Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.

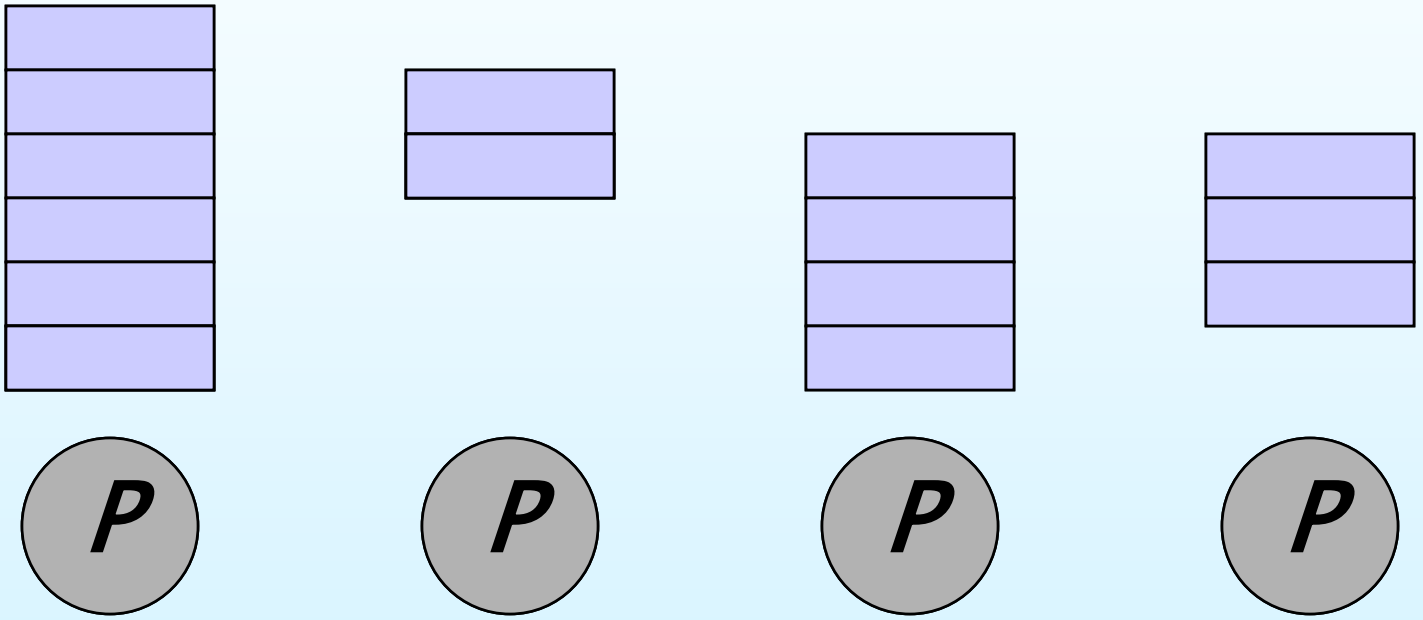


When a worker runs out of work, it ***steals*** from the top of a victim's deque.

When a worker has lots of parallelism, its deque never empties ⇒ ***linear speed-up***.

Work-Stealing Scheduler

Each ***worker*** (processor) maintains a ***work deque*** of ready threads, and it manipulates the bottom of the deque like a stack.



With sufficient parallelism, workers steal infrequently $\Rightarrow$ ***linear speed-up***.

Outline

- Introduction
- Cilk++ Extensions
- Runtime Platform
- Race Detector
- Case Study
- Conclusion

Serial Correctness

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

C/C++

Native Compiler

Single-Threaded Binary

Serial Regression Tests

Reliable Single-Threaded Code

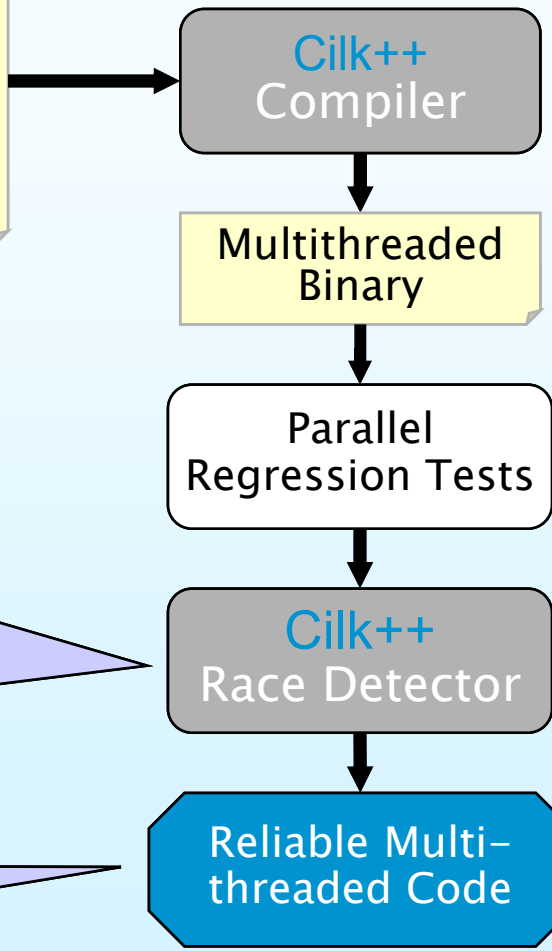
The ***serial elision*** is the code with the Cilk++ keywords removed or "nulled" out.

Serial correctness can be debugged and verified with standard regression tests on the serial elision.

Parallel Correctness

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

Cilk++



Parallel correctness can be debugged and verified with the Cilk++ data-race detector, which guarantees to find inconsistencies with the serial code quickly.

The Cilk++ code is as reliable as the original serial code.

Cilk++ Race Detector

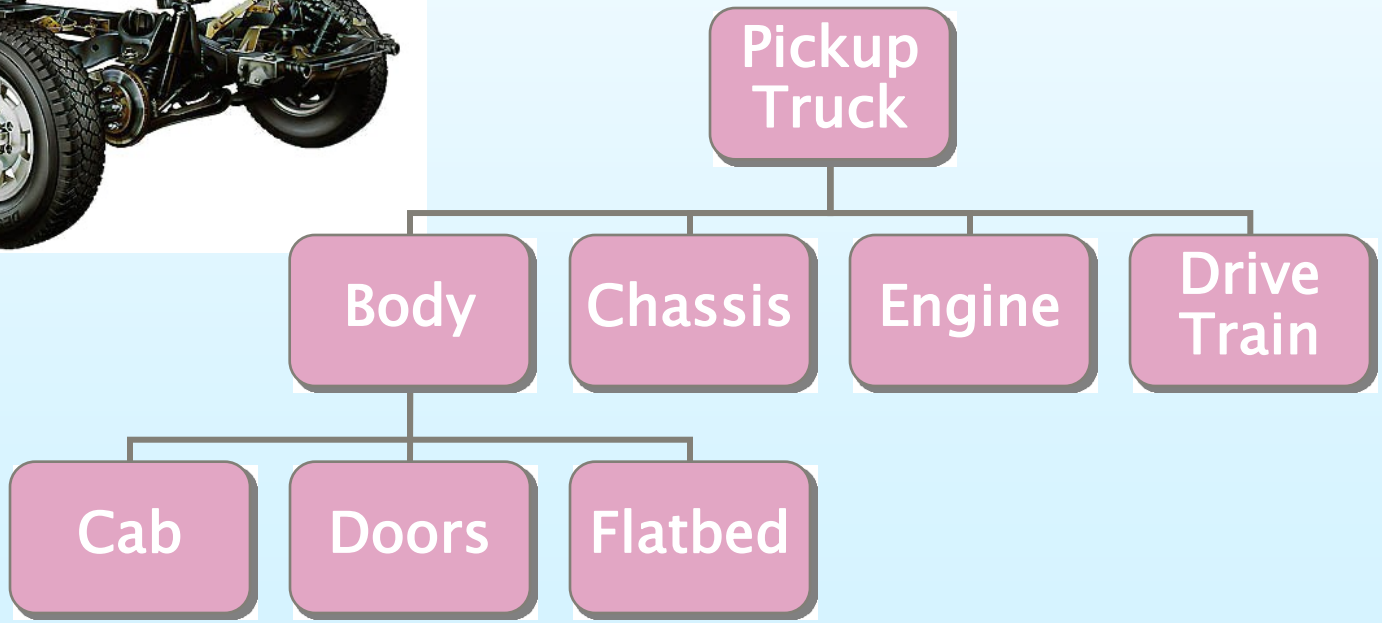
- Runs off the binary executable using dynamic instrumentation.
- Employs a regression-test methodology, where the customer provides test inputs.
- Mathematically guarantees to find races in ostensibly deterministic programs.
- Identifies filenames, lines, and variables involved in offending races, including stack traces.
- Understands mutual-exclusion locks.
- Runs about 10-50 times slower than real-time.

Outline

- Introduction
- Cilk++ Extensions
- Runtime Platform
- Race Detector
- Case Study
- Conclusion

Collision Detection

A CILK ARTS alpha design partner represents a mechanical assembly as a tree of subassemblies down to individual parts.



Parallelization Effort

Since the Cilk++ compiler was not yet working when this evaluation was performed, we used the MIT Cilk distribution.

Task	*MIT Cilk Time*	*Cilk++ Est. Time*
Convert from C++ to C (~3000 SLOC)	5 days	0
Eliminate global variables	1.5 days	30 min
"Cilkify"	30 min	30 min

All work was performed by a Brown University summer intern majoring in computer science with no experience in C, C++, or Cilk.

Keyword Count

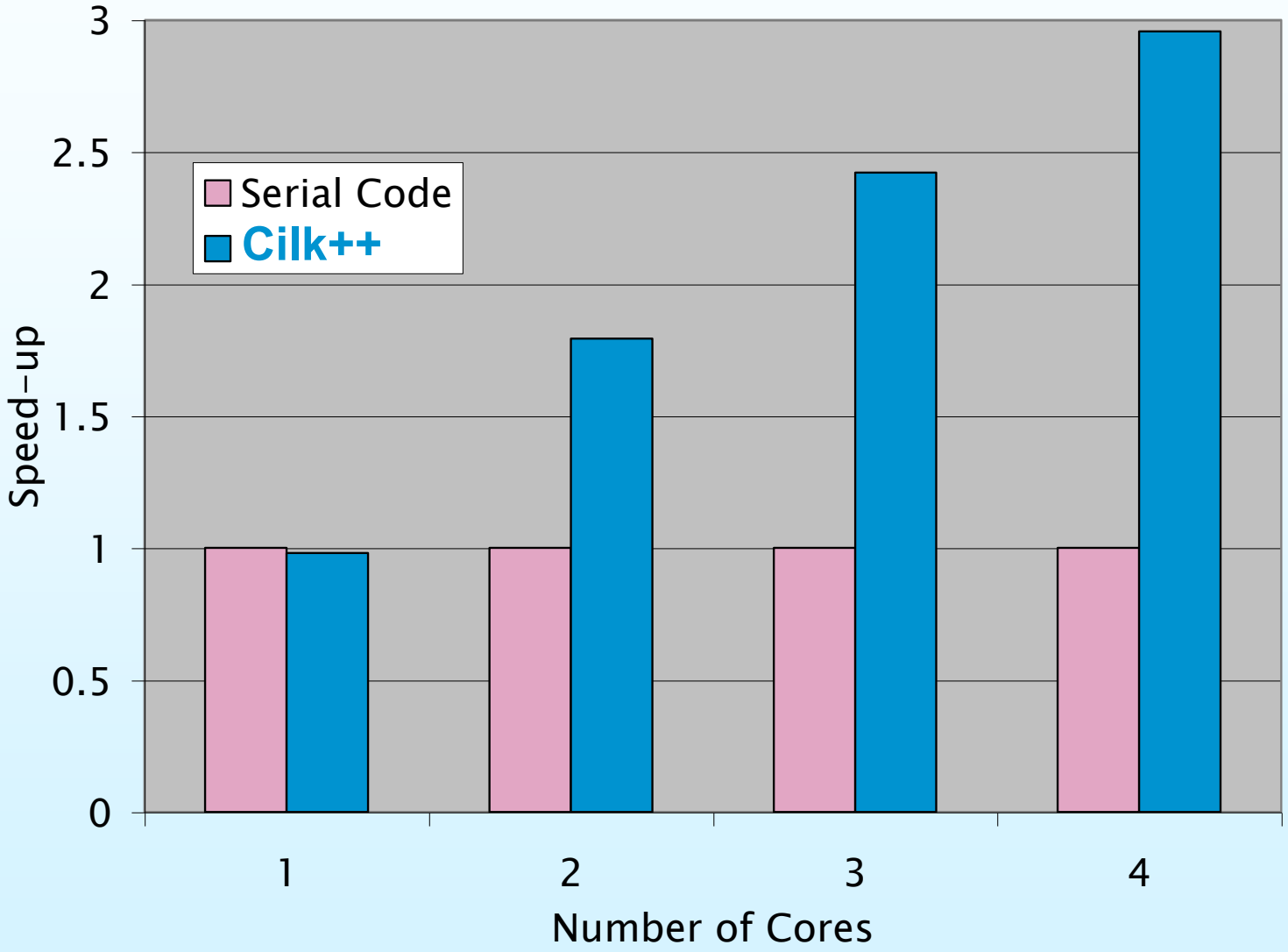
Mesh creation

Statement	*MIT Cilk*	*Cilk++*
cilk	7	3
spawn	11	6
sync	3	3

Detection

Statement	*MIT Cilk*	*Cilk++*
cilk	2	1
spawn	5	5
sync	2	3

Performance



Outline

- Introduction
- Cilk++ Extensions
- Runtime Platform
- Race Detector
- Case Study
- Conclusion

Comparison of Approaches

	Pthreads	*MPI*	*OpenMP*	*Data Parallel*	**Cilk**
Scales up	no	yes	yes	yes	**yes**
Scales down	yes	no	no	no	**yes**
Seamless	some	no	some	no	**yes**
Simple	no	no	no	no	**yes**
Safe release	no	no	no	yes	**yes**
Cache friendly	some	no	no	no	**yes**
Load balancing	manual	no	poor	poor	**yes**

CILK ARTS Is Hiring

Talk to me, or send your resume to **jobs@cilk.com**.

CILK **ARTS** celebrates BEAUTY in engineering, EMPATHY in business, and INTEGRITY and FAIRNESS in all we do.