6.172
Performance
Engineering
of Software
Systems

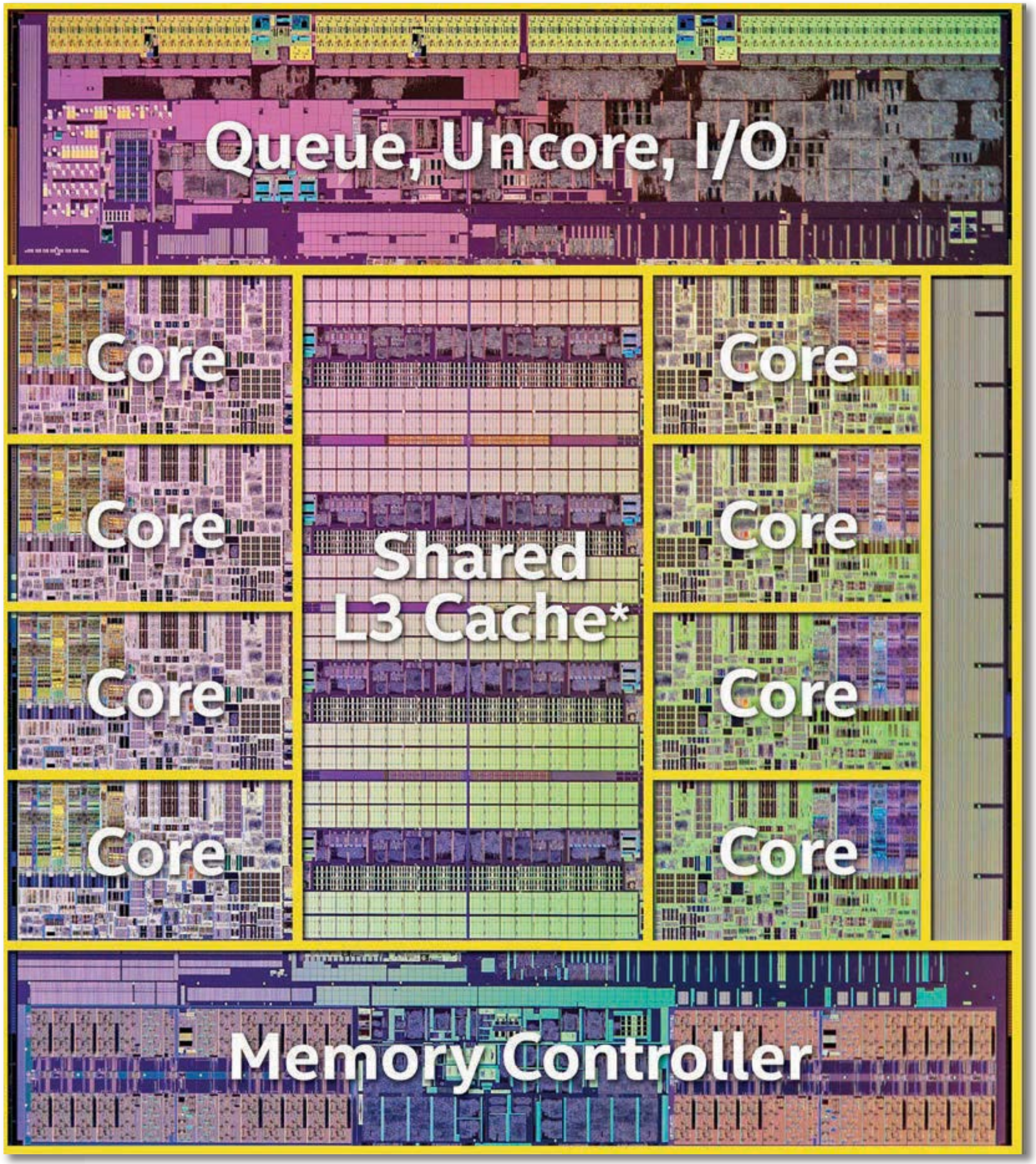
LECTURE 6

Multicore Programming

Julian Shun

© 2008–2018 by the MIT 6.172 Lecturers

Multicore Processors



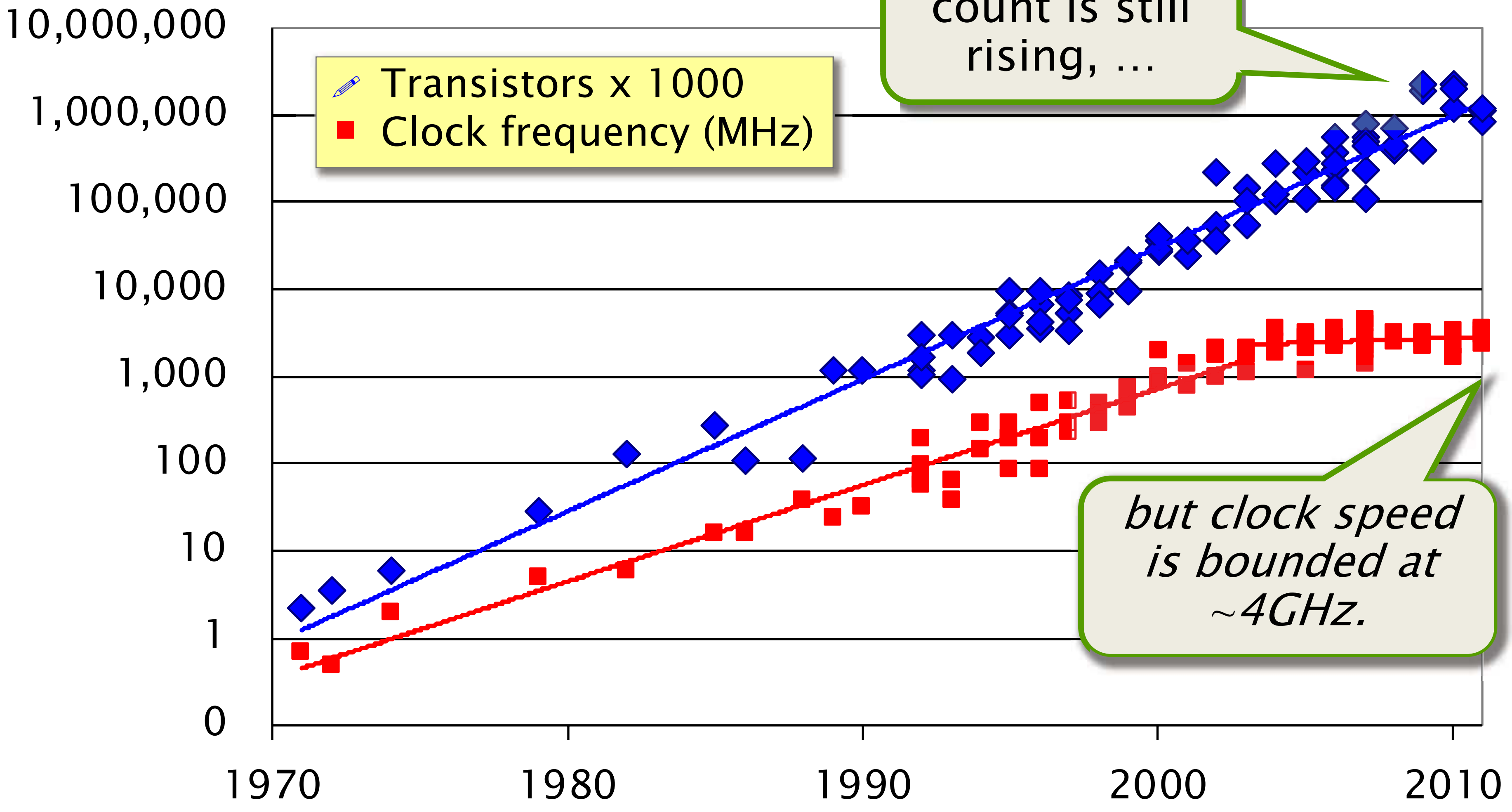
Q Why do semiconductor vendors provide chips with multiple processor cores?

A Because of Moore's Law and the end of the scaling of clock frequency.

Intel Haswell-E

© Intel. All rights reserved. This content is excluded from our Creative Commons license. For more information, see https://ocw.mit.edu/help/faq-fair-use/

Technology Scaling

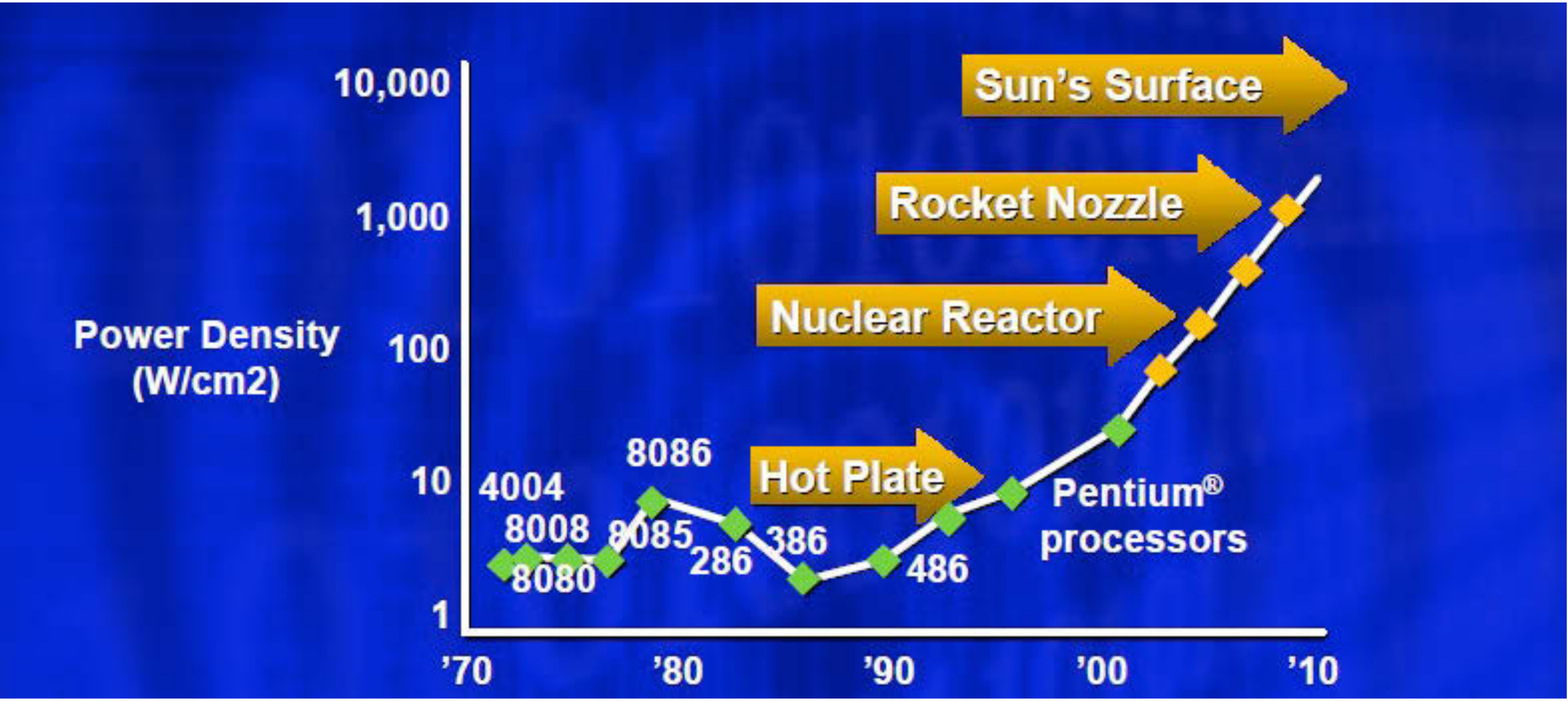


Transistor count is still rising, …

but clock speed is bounded at ~4GHz.

© 2008–2018 by the MIT 6.172 Lecturers

Power Density

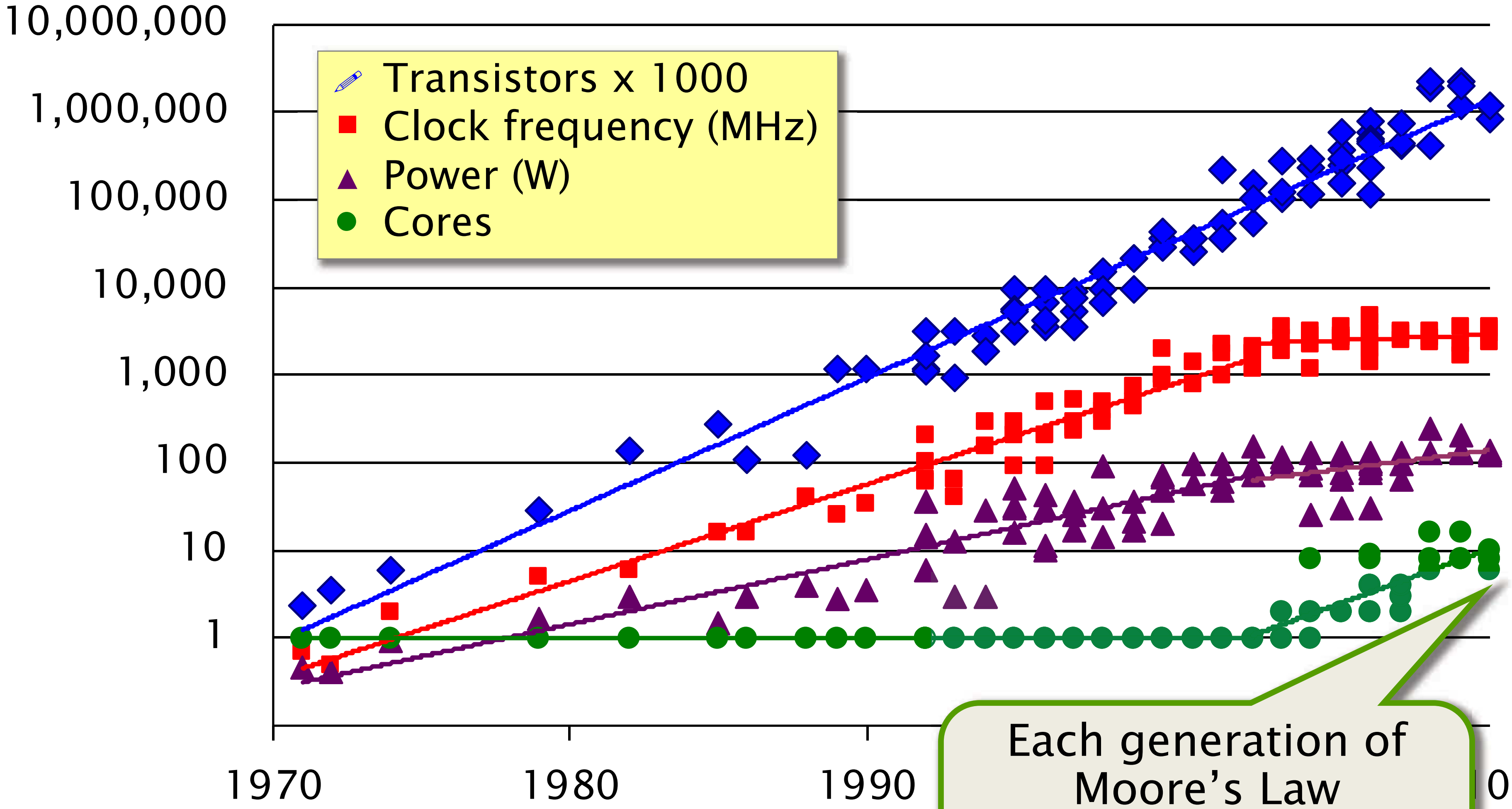


Source: Patrick Gelsinger, *Intel Developer's Forum*, Intel Corporation, 2004.

Projected power density, if clock frequency had continued its trend of scaling 25%–30% per year.

© Paul Gelsinger at Intel Corporation. All rights reserved. This content is excluded from our Creative Commons license. For more information, see https://ocw.mit.edu/help/faq-fair-use/

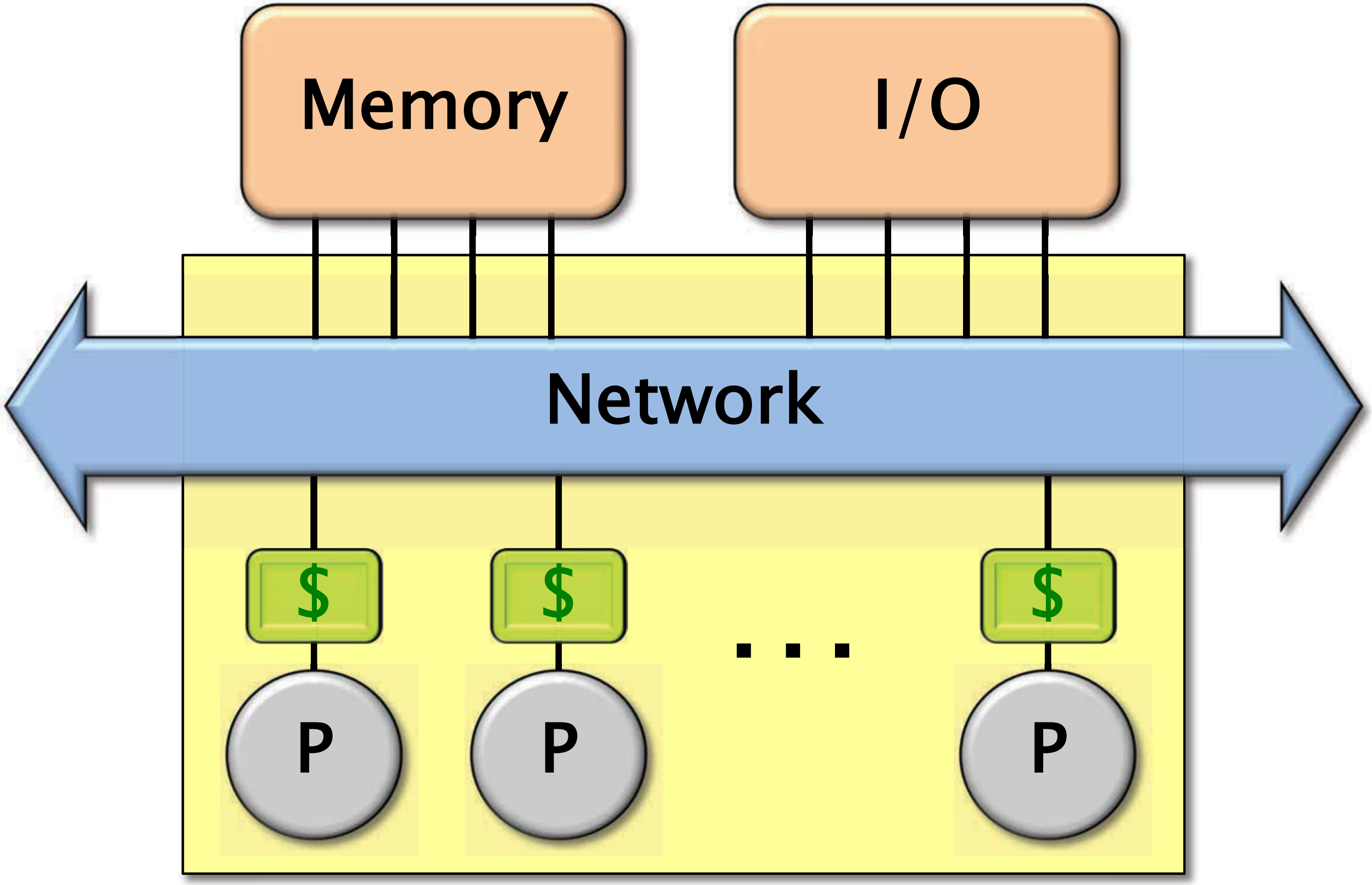
Technology Scaling



Each generation of Moore's Law potentially doubles the number of cores.

© 2008–2018 by the MIT 6.172 Lecturers

Abstract Multicore Architecture



Chip Multiprocessor (CMP)

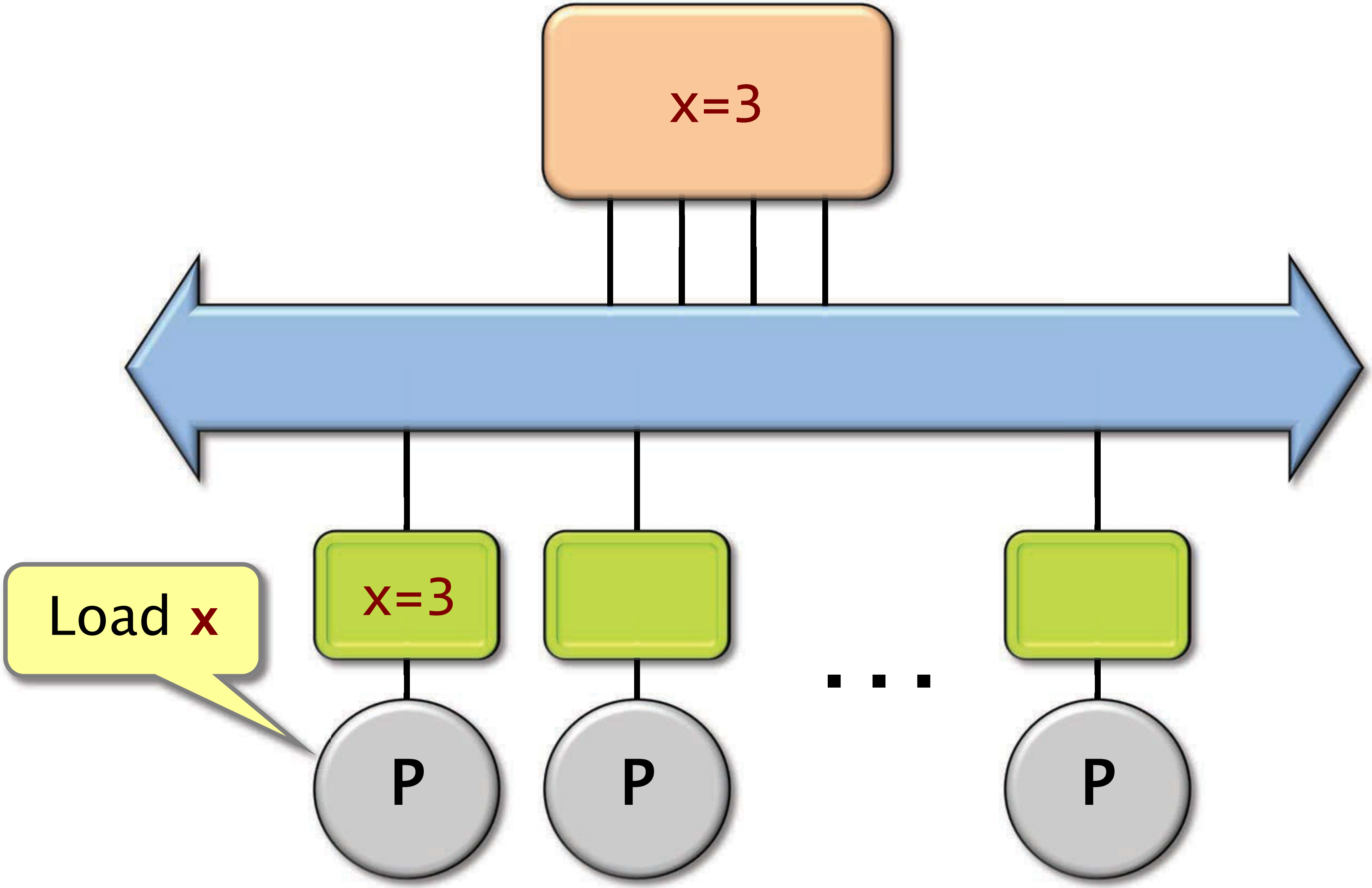
© 2008–2018 by the MIT 6.172 Lecturers

OUTLINE

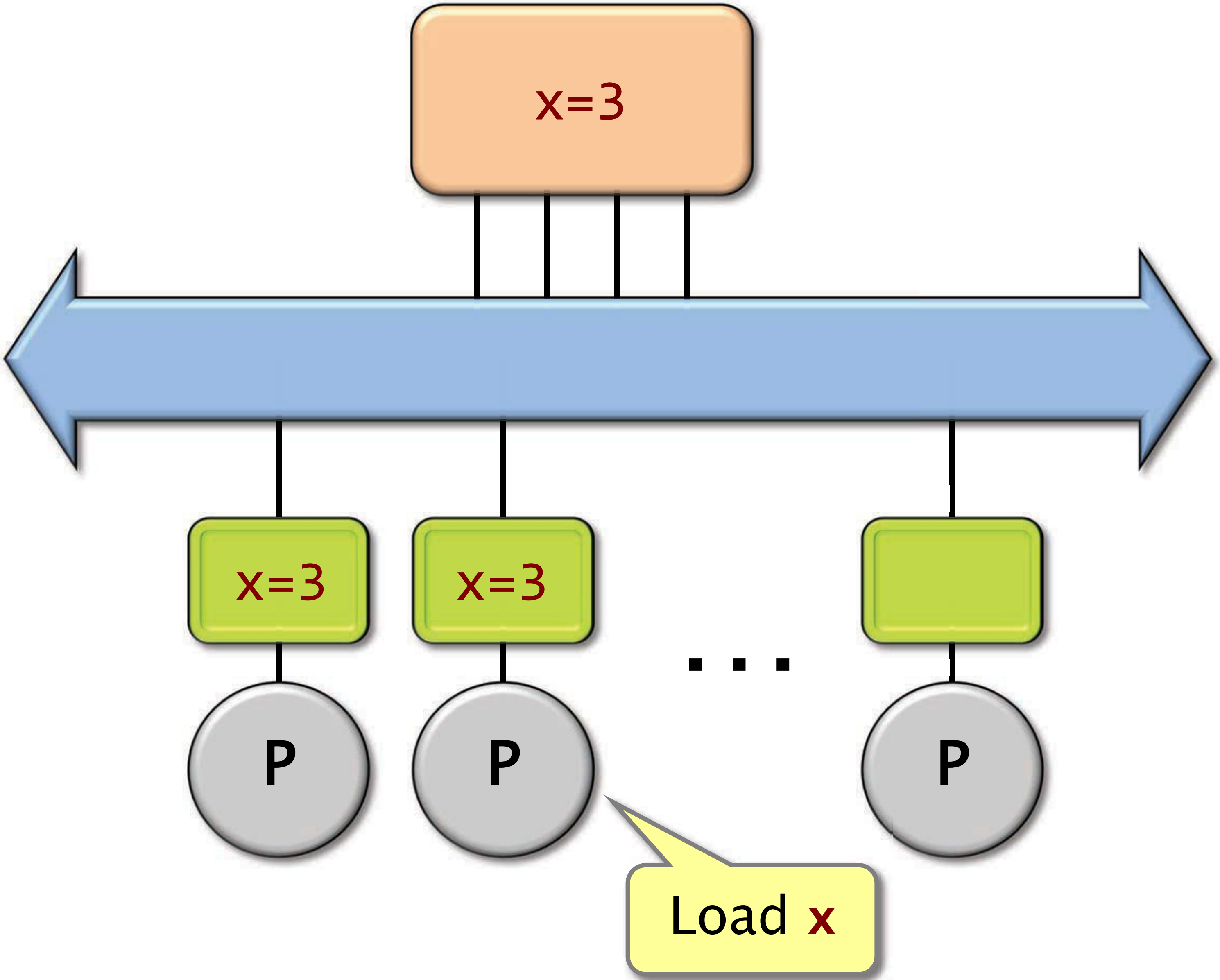
- Shared-Memory Hardware
- Concurrency Platforms
  - Pthreads (and WinAPI Threads)
  - Threading Building Blocks
  - OpenMP
  - Cilk Plus

© 2008–2018 by the MIT 6.172 Lecturers

Cache Coherence



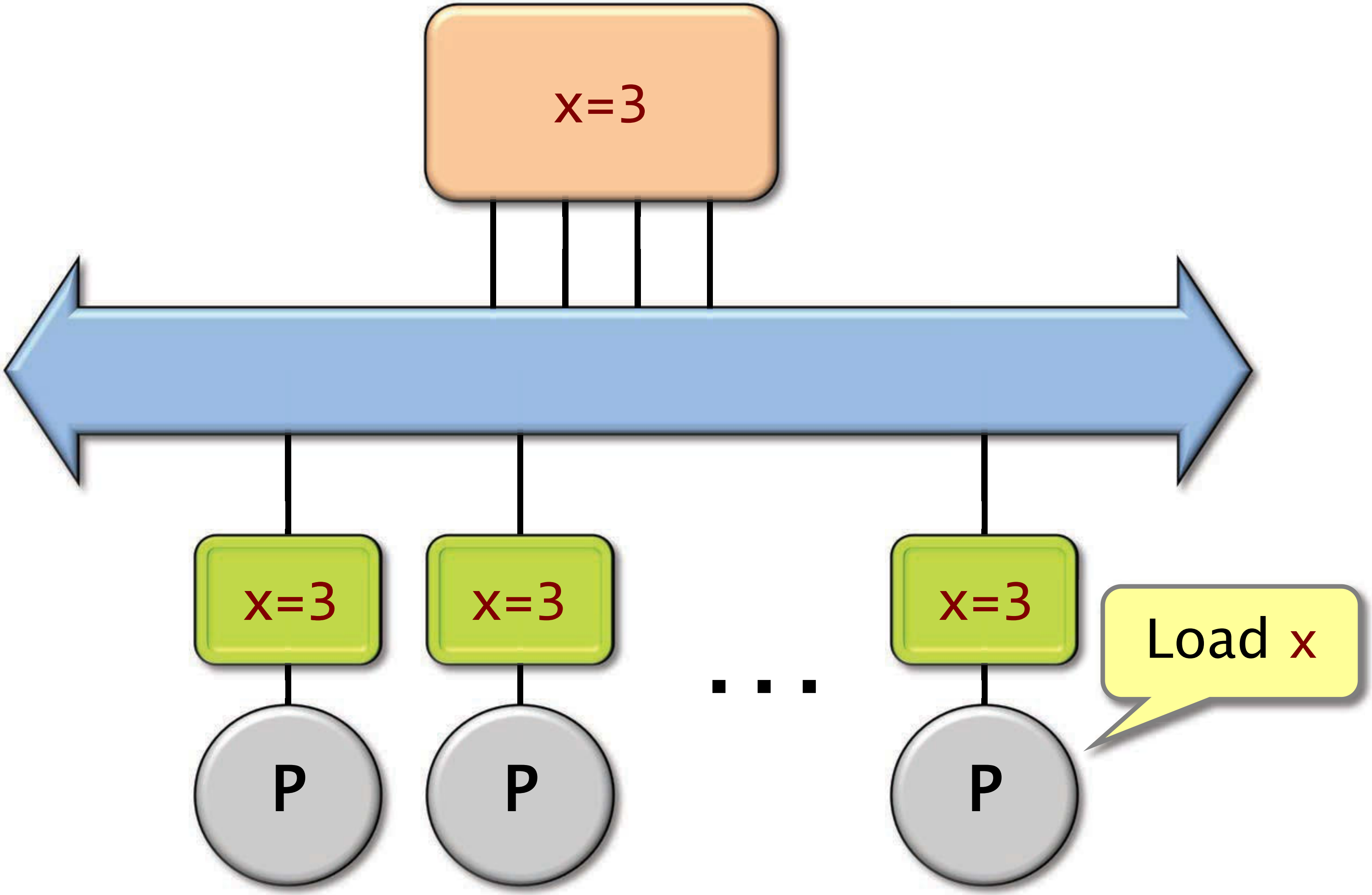
© 2008–2018 by the MIT 6.172 Lecturers



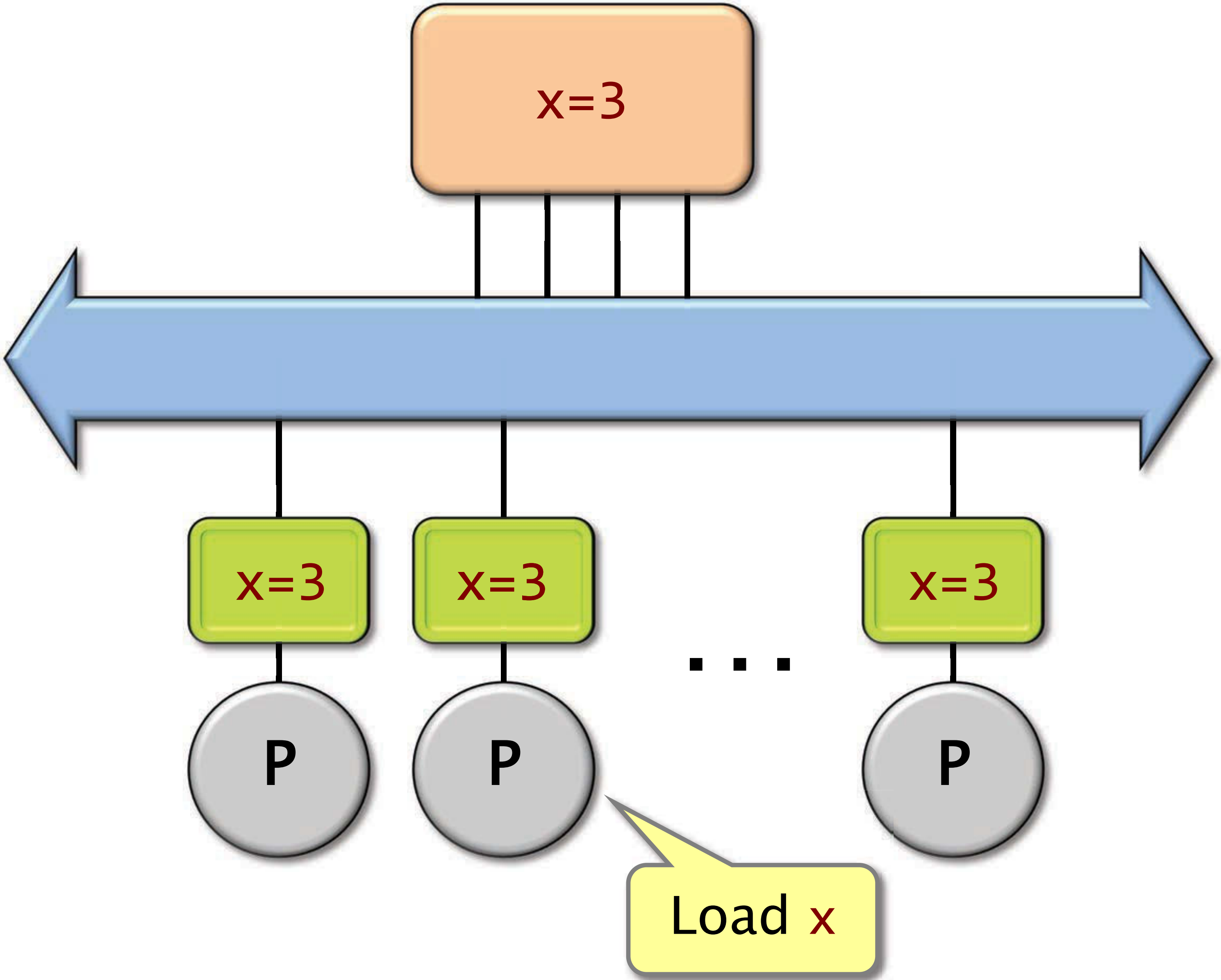
Load **x**

© 2008–2018 by the MIT 6.172 Lecturers

Cache Coherence

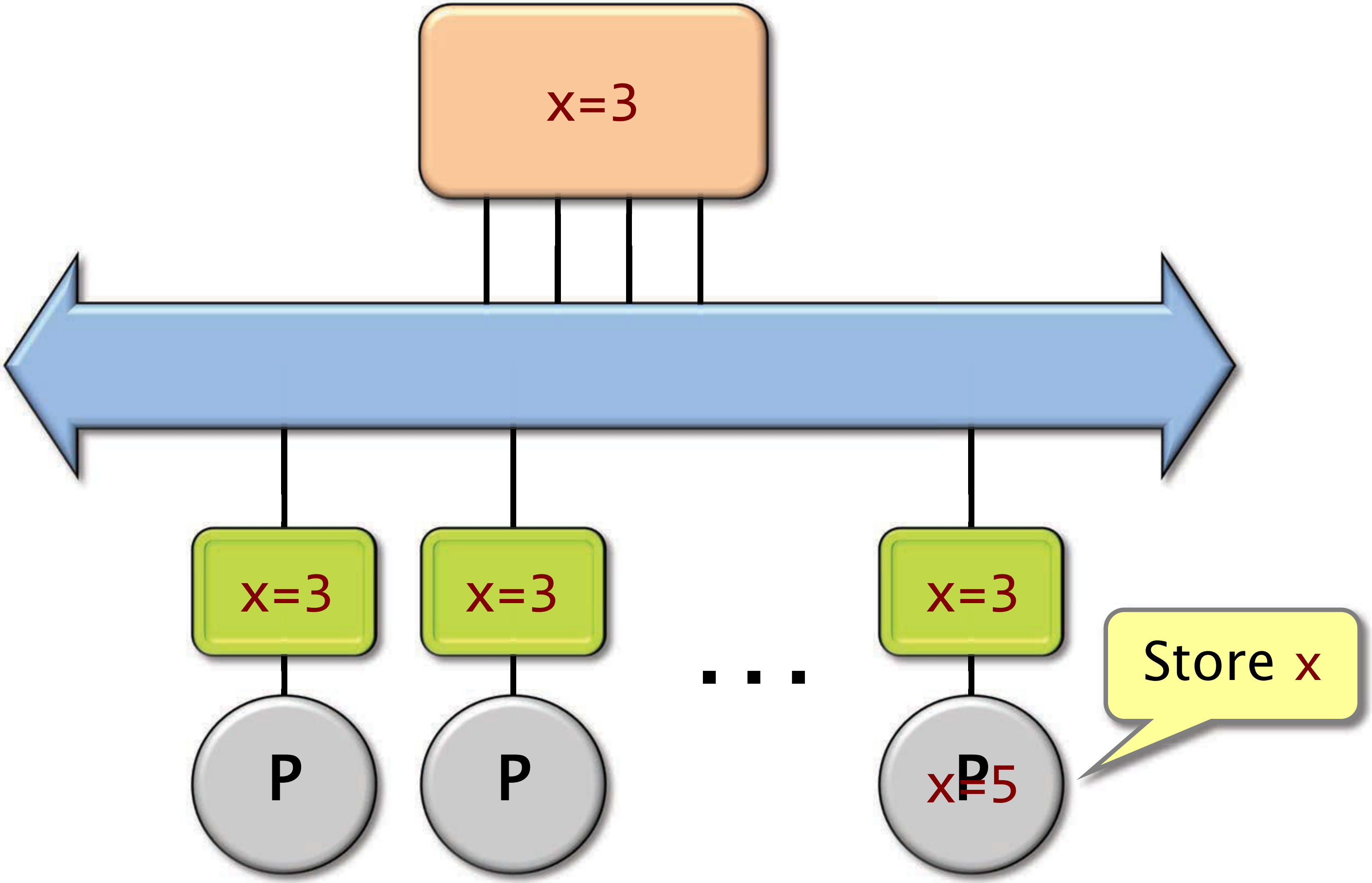


© 2008–2018 by the MIT 6.172 Lecturers



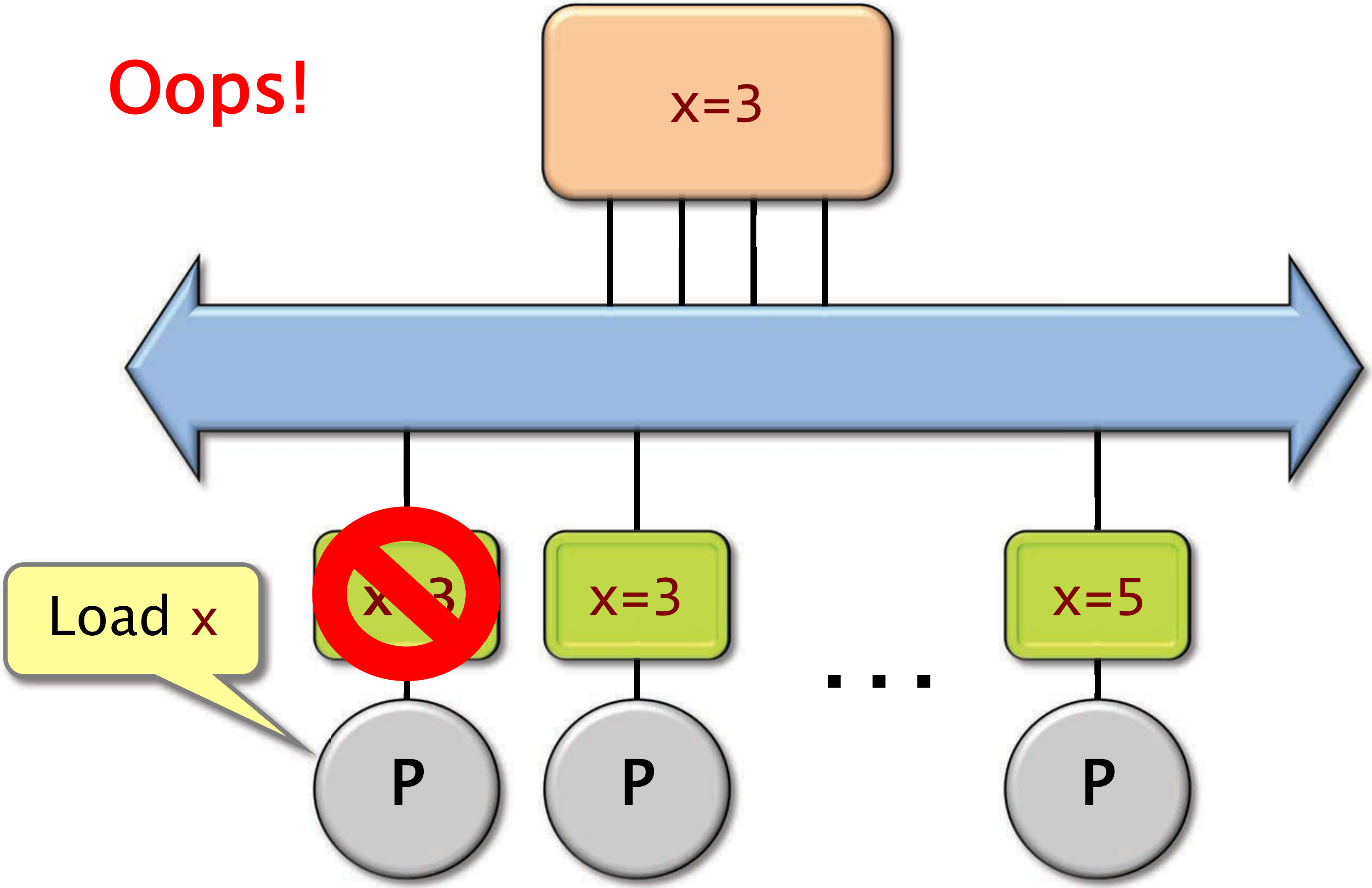
© 2008–2018 by the MIT 6.172 Lecturers

Cache Coherence



Cache Coherence

Oops!



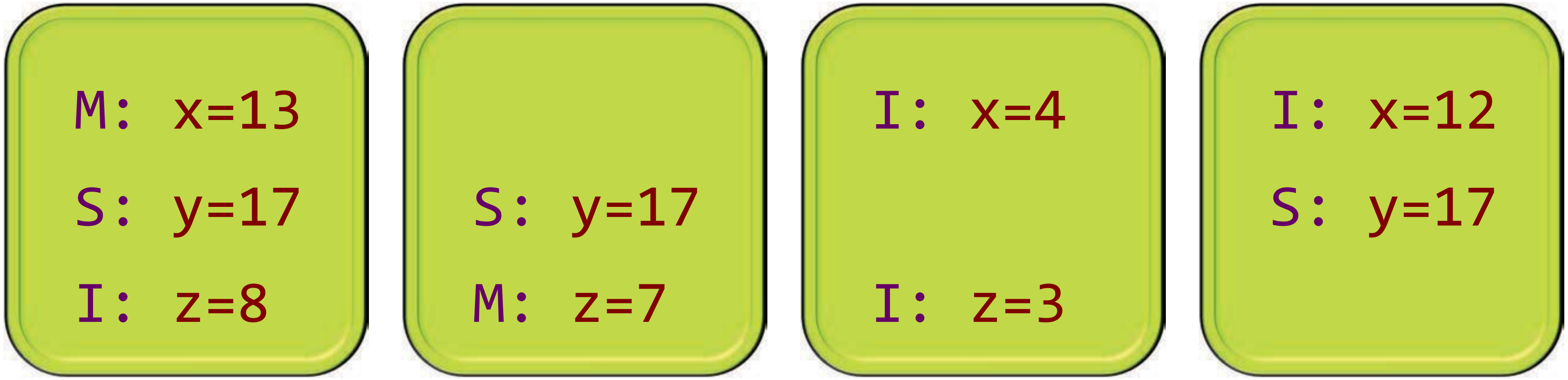
Load x

© 2008–2018 by the MIT 6.172 Lecturers

MSI Protocol

Each cache line is labeled with a state:

- M: cache block has been modified. No other caches contain this block in M or S states.
- S: other caches may be sharing this block.
- I: cache block is invalid (same as not there).



M: x=13
S: y=17
I: z=8

S: y=17
M: z=7

I: x=4
I: z=3

I: x=12
S: y=17

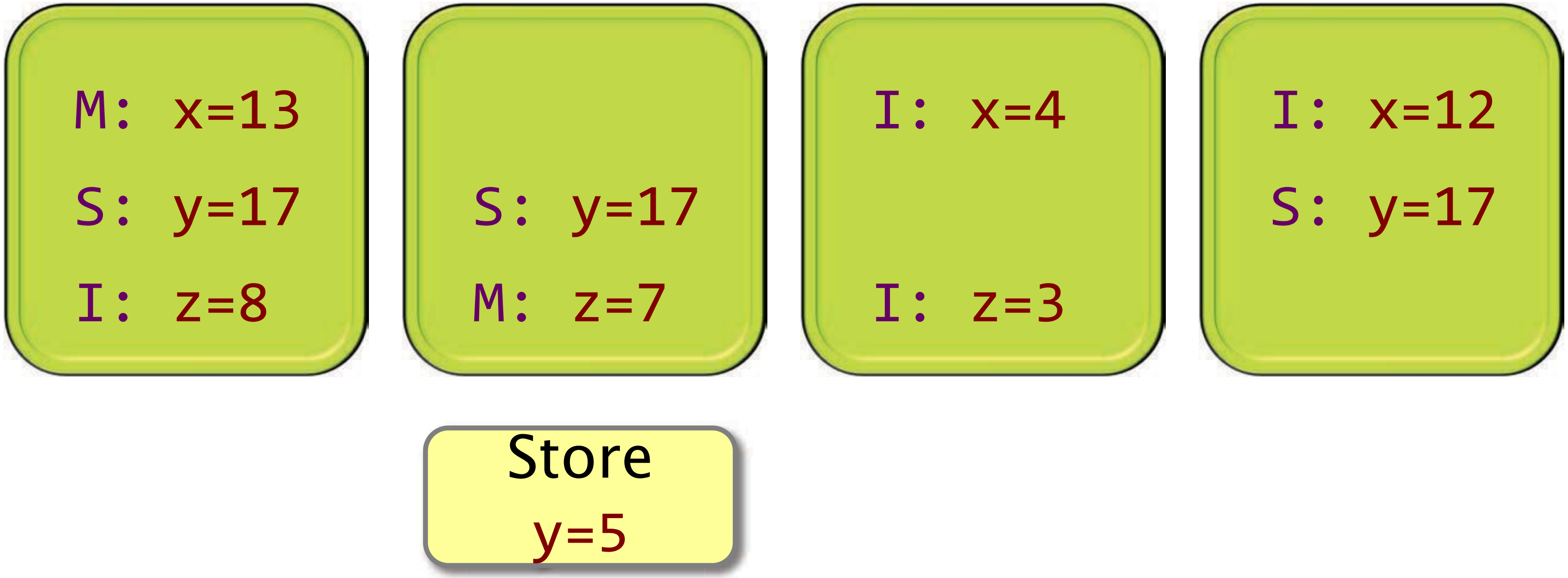
Before a cache modifies a location, the hardware first invalidates all other copies.

© 2008–2018 by the MIT 6.172 Lecturers

MSI Protocol

Each cache line is labeled with a state:

- **M**: cache block has been modified. No other caches contain this block in M or S states.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid (same as not there).



M: x=13
S: y=17
I: z=8

S: y=17
M: z=7

I: x=4
I: z=3

I: x=12
S: y=17

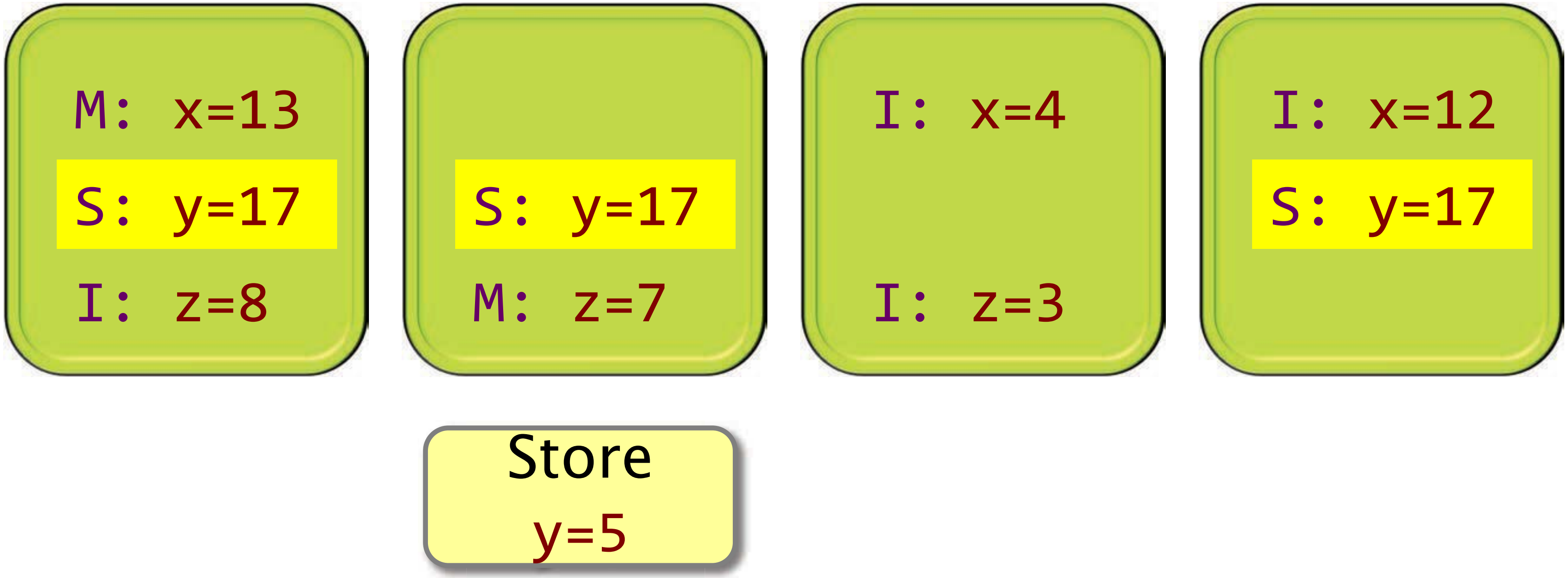
Store
y=5

© 2008–2018 by the MIT 6.172 Lecturers

MSI Protocol

Each cache line is labeled with a state:

- **M**: cache block has been modified. No other caches contain this block in M or S states.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid (same as not there).



M: x=13
S: y=17
I: z=8

S: y=17
M: z=7

I: x=4
I: z=3

I: x=12
S: y=17

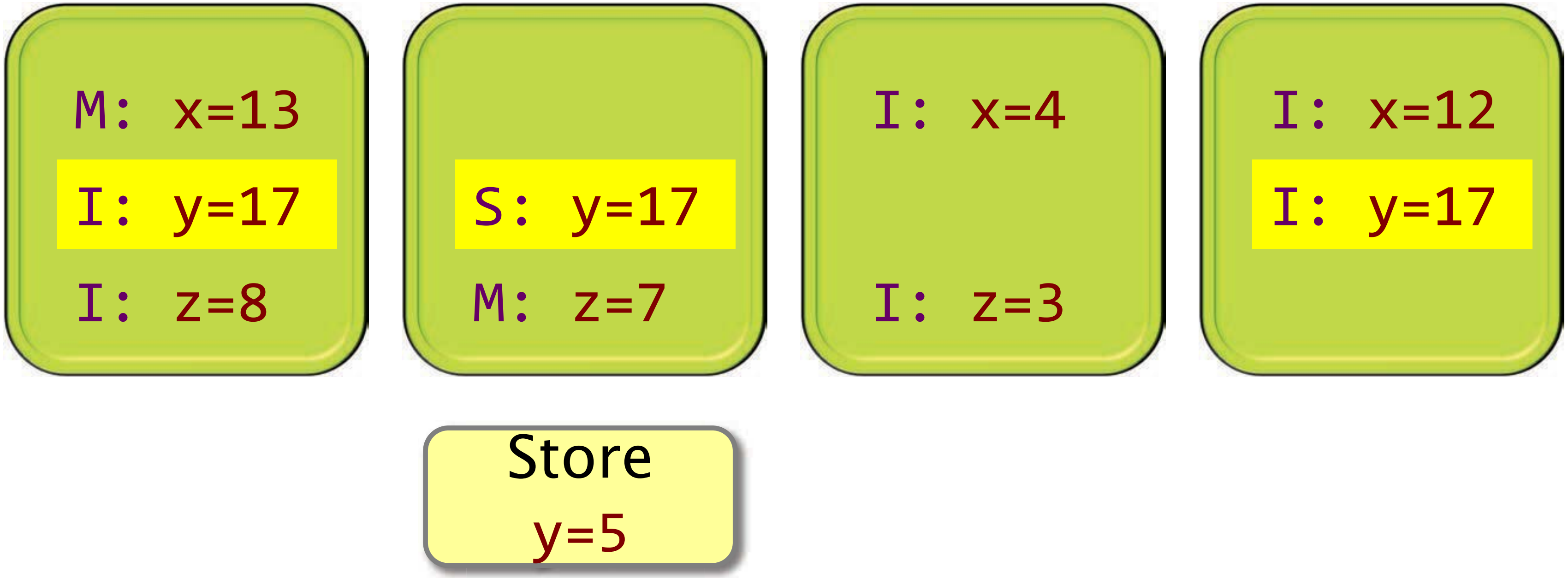
Store
y=5

© 2008–2018 by the MIT 6.172 Lecturers

MSI Protocol

Each cache line is labeled with a state:

- **M**: cache block has been modified. No other caches contain this block in M or S states.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid (same as not there).



M: x=13
I: y=17
I: z=8

S: y=17
M: z=7

I: x=4
I: z=3

I: x=12
I: y=17

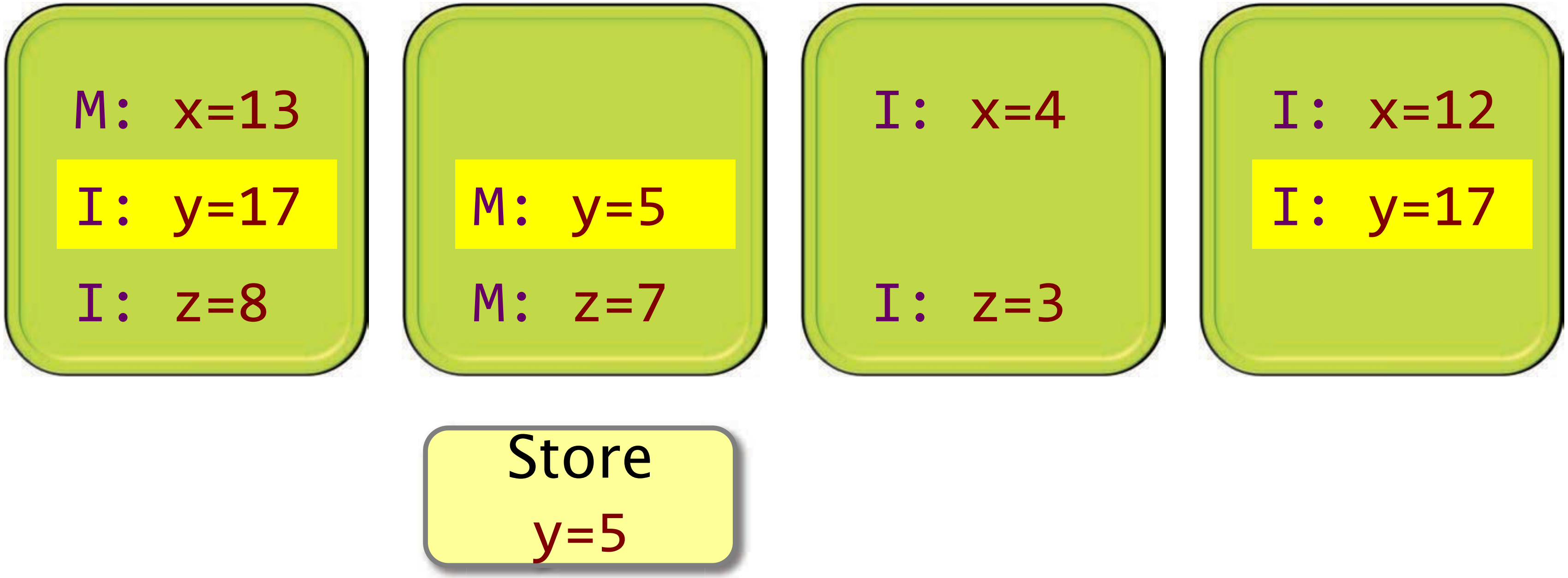
Store
y=5

© 2008–2018 by the MIT 6.172 Lecturers

MSI Protocol

Each cache line is labeled with a state:

- M: cache block has been modified. No other caches contain this block in M or S states.
- S: other caches may be sharing this block.
- I: cache block is invalid (same as not there).



M: x=13
I: y=17
I: z=8

M: y=5
M: z=7

I: x=4
I: z=3

I: x=12
I: y=17

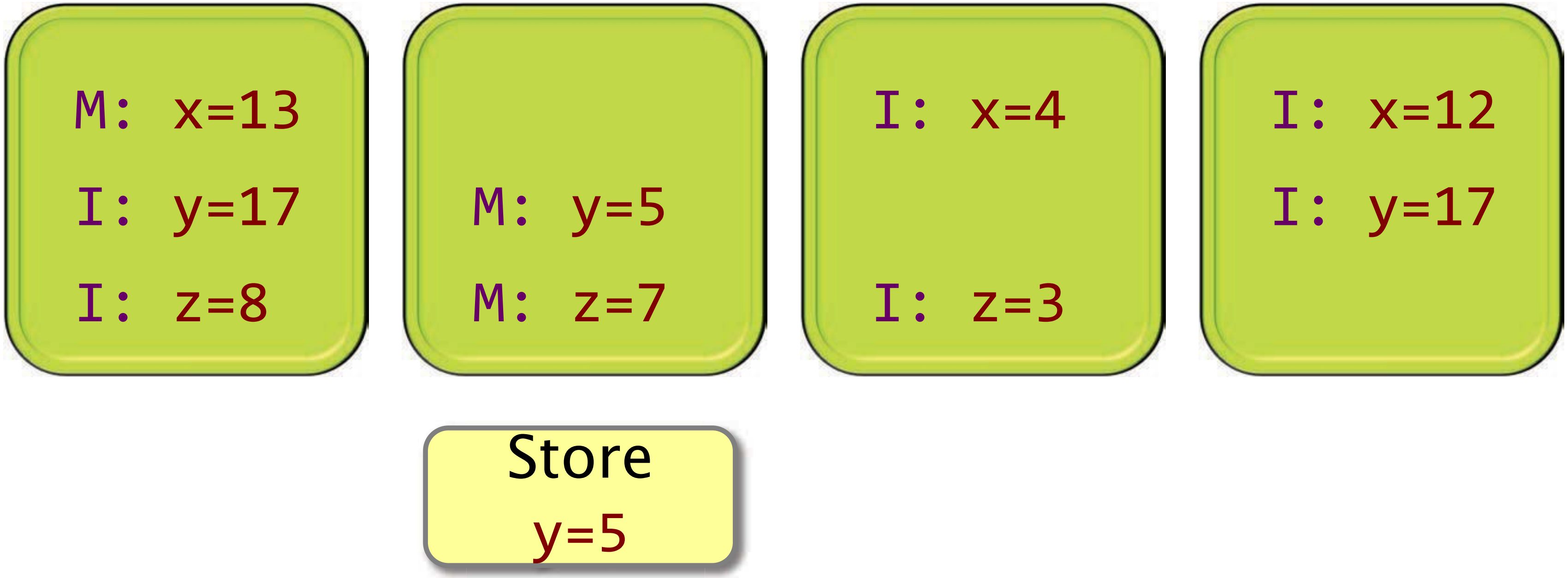
Store
y=5

© 2008–2018 by the MIT 6.172 Lecturers

MSI Protocol

Each cache line is labeled with a state:

- **M**: cache block has been modified. No other caches contain this block in M or S states.
- **S**: other caches may be sharing this block.
- **I**: cache block is invalid (same as not there).



M: x=13
I: y=17
I: z=8

M: y=5
M: z=7

I: x=4
I: z=3

I: x=12
I: y=17

Store
y=5

© 2008–2018 by the MIT 6.172 Lecturers

Outline

- Shared-Memory Hardware
- Concurrency Platforms
  - Pthreads (and WinAPI Threads)
  - Threading Building Blocks
  - OpenMP
  - Cilk

© 2008–2018 by the MIT 6.172 Lecturers

Concurrency Platforms

- Programming directly on processor cores is painful and error-prone.
- A concurrency platform abstracts processor cores, handles synchronization and communication protocols, and performs load balancing.
- Examples
  - Pthreads and WinAPI threads
  - Threading Building Blocks (TBB)
  - OpenMP
  - Cilk

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci Numbers

The Fibonacci numbers are the sequence ⟨0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …⟩, where each number is the sum of the previous two.

Recurrence:

$F_0 = 0,$
$F_1 = 1,$
$F_n = F_{n-1} + F_{n-2}$ for $n > 1$.

Image is in the public domain.

The sequence is named after Leonardo di Pisa (1170–1250 A.D.), also known as Fibonacci, a contraction of *filius Bonaccii* —“son of Bonaccio.” Fibonacci’s 1202 book *Liber Abaci* introduced the sequence to Western mathematics, although it had previously been discovered by Indian mathematicians.

© 2008–2018 by the MIT 6.172 Lecturers

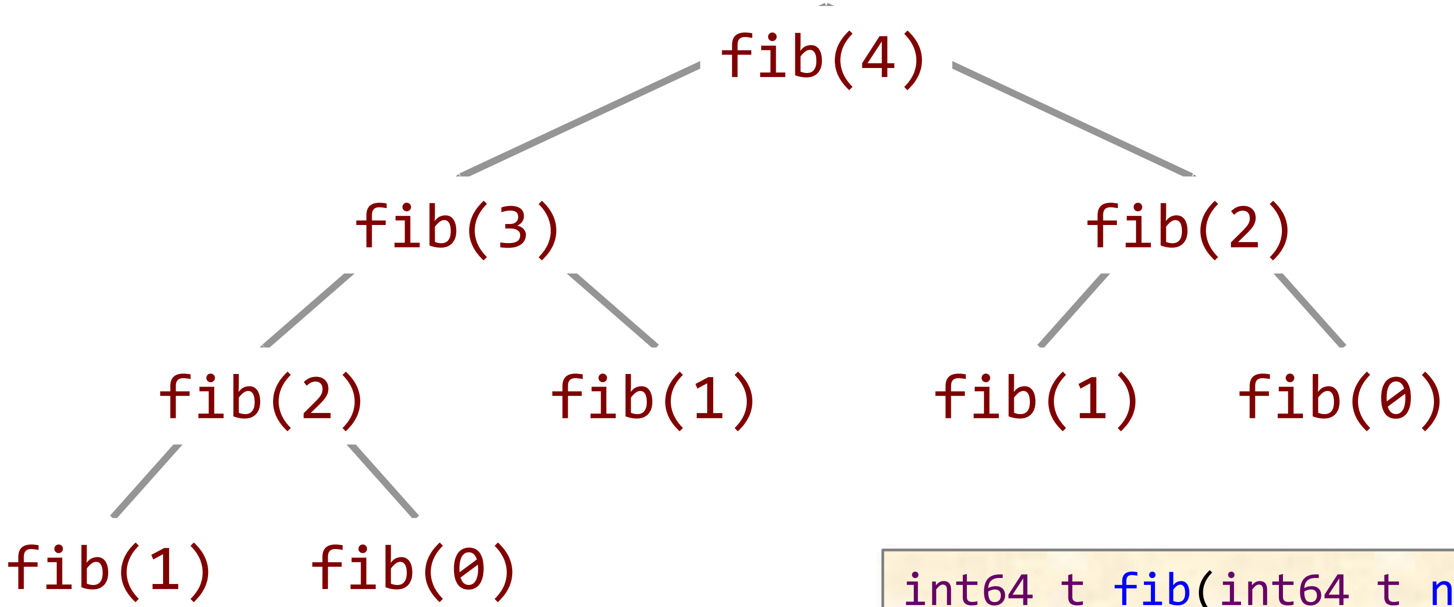
Fibonacci Program

```c
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

int main(int argc, char *argv[]) {
  int64_t n = atoi(argv[1]);
  int64_t result = fib(n);
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

Disclaimer to Algorithms Police

This recursive program is a poor way to compute the nth Fibonacci number, but it provides for a good didactic example.

Fibonacci Execution



fib(4)

fib(3) fib(2)

fib(2) fib(1) fib(1) fib(0)

fib(1) fib(0)

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}
```

Key idea for parallelization

The calculations of fib(n-1) and fib(n-2) can be executed simultaneously without mutual interference.

© 2008–2018 by the MIT 6.172 Lecturers

OUTLINE

- Shared-Memory Hardware
- Concurrency Platforms
  - Pthreads (and WinAPI Threads)
  - Threading Building Blocks
  - OpenMP
  - Cilk

© 2008–2018 by the MIT 6.172 Lecturers

Pthreads*

- Standard API for threading specified by ANSI/IEEE POSIX 1003.1–2008.
- Do-it-yourself concurrency platform.
- Built as a library of functions with “special” non-C semantics.
- Each thread implements an abstraction of a processor, which are multiplexed onto machine resources.
- Threads communicate though shared memory.
- Library functions mask the protocols involved in interthread coordination.

*WinAPI threads provide similar functionality.

© 2008–2018 by the MIT 6.172 Lecturers

Key Pthread Functions

```
int pthread_create(
  pthread_t *thread,
    //returned identifier for the new thread
  const pthread_attr_t *attr,
    //object to set thread attributes (NULL for default)
  void *(*func)(void *),
    //routine executed after creation
  void *arg
    //a single argument passed to func
) //returns error status
```

```
int pthread_join(
  pthread_t thread,
    //identifier of thread to wait for
  void **status
    //terminating thread’s status (NULL to ignore)
) //returns error status
```

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(void *ptr) {
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

Original code.

```
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(void *ptr) {
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(void *ptr) {
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Structure for thread arguments.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(void *ptr) {
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Function called when thread is created.

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

```
#include <inttypes.h>
#include <pthread
#include <stdio.h
#include <stdlib.

int64_t fib(int64
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(void *ptr) {
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

No point in creating thread if there isn't enough to do.

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

```
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(void *ptr) {
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], N
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

Marshal input argument to thread.

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

```
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(void *ptr) {
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

Create thread to execute `fib(n-1)`.

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

```
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(voi
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

Main program executes fib(n-2) in parallel.

```
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(void *ptr) {
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

Block until the auxiliary thread finishes.

© 2008–2018 by the MIT 6.172 Lecturers

Pthread Implementation

```c
#include <inttypes.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}

typedef struct {
  int64_t input;
  int64_t output;
} thread_args;

void *thread_func(void *ptr) {
  int64_t i = ((thread_args *) ptr)->input;
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

```c
int main(int argc, char *argv[]) {
  pthread_t thread;
  thread_args args;
  int status;
  int64_t result;

  if (argc < 2) { return 1; }
  int64_t n = strtoul(argv[1], NULL, 0);
  if (n < 30) {
    result = fib(n);
  } else {
    args.input = n-1;
    status = pthread_create(&thread,
                            NULL,
                            thread_func,
                            (void*) &args);
    // main can continue executing
    if (status != NULL) { return 1; }
    result = fib(n-2);
    // wait for the thread to terminate
    status = pthread_join(thread, NULL);
    if (status != NULL) { return 1; }
    result += args.output;
  }
  printf("Fibonacci of %" PRId64 " is %" PRId64 ".\n",
         n, result);
  return 0;
}
```

Add the results together to produce the final output.

© 2008–2018 by the MIT 6.172 Lecturers

Issues with Pthreads

Overhead	The cost of creating a thread $>10^4$ cycles ⇒ coarse-grained concurrency. (Thread pools can help.)
Scalability	Fibonacci code gets at most about 1.5 speedup for 2 cores. Need a rewrite for more cores.
Modularity	The Fibonacci logic is no longer neatly encapsulated in the `fib()` function.
Code Simplicity	Programmers must marshal arguments (shades of 1958!) and engage in error-prone protocols in order to load-balance.

© 2008–2018 by the MIT 6.172 Lecturers

Outline

- Shared-Memory Hardware
- Concurrency Platforms
  - Pthreads (and WinAPI Threads)
  - Threading Building Blocks
  - OpenMP
  - Cilk

© 2008–2018 by the MIT 6.172 Lecturers

Threading Building Blocks

- Developed by Intel.
- Implemented as a C++ library that runs on top of native threads.
- Programmer specifies tasks rather than threads.
- Tasks are automatically load balanced across the threads using a work-stealing algorithm inspired by research at MIT.
- Focus on performance.

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_) :
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_child() )
                         FibTask(n-1, &x);
      FibTask& b = *new( allocate_child() )
                         FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

```
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                    FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_) :
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_child() )
                         FibTask(n-1, &x);
      FibTask& b = *new( allocate_child() )
                         FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

A computation organized as explicit tasks.

```
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                    FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_) :
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_child() )
                     FibTask(n-1, &x);
      FibTask& b = *new( allocate_child() )
                     FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

FibTask has an input parameter n and an output parameter sum.

```
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                 FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_ )
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_child() )
                   FibTask(n-1, &x);
      FibTask& b = *new( allocate_child() )
                   FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

The execute() function performs the computation when the task is started.

```
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                   FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_) :
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_child() )
                   FibTask(n-1, &x);
      FibTask& b = *new( allocate_child() )
                   FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

Recursively create two child tasks, a and b.

```
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                    FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_) :
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_child() )
                   FibTask(n-1, &x);
      FibTask& b = *new( allocate_child() )
                   FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

Set the number of tasks to wait for (2 children + 1 implicit for bookkeeping).

```
#include <stdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                 FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_) :
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_child() )
                           FibTask(n-1, &x);
      FibTask& b = *new( allocate_child() )
                           FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

Start task b.

```
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                 FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_) :
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_child()
                     FibTask(n-1, &x
      FibTask& b = *new( allocate_chil  ) )
                     FibTask(n-2,  y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

Start task a and wait for both a and b to finish.

```
#inc        stdint>
#i          <iostream>
#       ue "tbb/task.h"

 t main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                 FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_) :
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_c    () )
                         FibTask(n    &x);
      FibTask& b = *new( allocat  child() )
                         FibT  k(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_f r_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

Add the results together to produce the final output.

```
#i        cstdint>
#      e <iostream>
    lude "tbb/task.h"

int main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                 FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in TBB

```
using namespace tbb;
class FibTask: public task {
public:
  const int64_t n;
  int64_t* const sum;
  FibTask(int64_t n_, int64_t* sum_) :
          n(n_), sum(sum_) {}

  task* execute() {
    if( n < 2 ) {
      *sum = n;
    } else {
      int64_t x, y;
      FibTask& a = *new( allocate_child
                         FibTask(n-1, &x);
      FibTask& b = *new( allocate_child() )
                         FibTask(n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_for_all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

Create root task;
spawn and wait.

```
#include <cstdint>
#include <iostream>
#include "tbb/task.h"

int main(int argc, char *argv[]) {
  int64_t res;
  if (argc < 2) { return 1; }
  int64_t n =
    strtoul(argv[1], NULL, 0);
  FibTask& a = *new(task::allocate_root())
                 FibTask(n, &res);
  task::spawn_root_and_wait(a);

  std::cout << "Fibonacci of " << n
       << " is " << res << std::endl;
  return 0;
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Other TBB Features

- TBB provides many C++ templates to express common patterns simply, such as
  - `parallel_for` for loop parallelism,
  - `parallel_reduce` for data aggregation,
  - `pipeline` and `filter` for software pipelining.
- TBB provides concurrent container classes, which allow multiple threads to safely access and update items in the container concurrently.
- TBB also provides a variety of mutual-exclusion library functions, including locks and atomic updates.

© 2008–2018 by the MIT 6.172 Lecturers

Outline

- Shared-Memory Hardware
- Concurrency Platforms
  - Pthreads (and WinAPI Threads)
  - Threading Building Blocks
  - OpenMP
  - Cilk

© 2008–2018 by the MIT 6.172 Lecturers

OpenMP

- Specification by an industry consortium.
- Several compilers available, both open-source and proprietary, including GCC, ICC, Clang, and Visual Studio.
- Linguistic extensions to C/C++ and Fortran in the form of compiler pragmas.
- Runs on top of native threads.
- Supports loop parallelism, task parallelism, and pipeline parallelism.

Fibonacci in OpenMP

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
#pragma omp task shared(x,n)
    x = fib(n-1);
#pragma omp task shared(y,n)
    y = fib(n-2);
#pragma omp taskwait
    return (x + y);
  }
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in OpenMP

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
#pragma omp task shared(x,n)
    x = fib(n-1);
#pragma omp task shared(y,n)
    y = fib(n-2);
#pragma omp taskwait
    return (x + y);
  }
}
```

Compiler directive.

Fibonacci in OpenMP

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
#pragma omp task shared(x,n)
    x = fib(n-1);
#pragma omp task shared(y,n)
    y = fib(n-2);
#pragma omp taskwait
    return (x + y);
  }
}
```

The following statement is an independent task.

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in OpenMP

```c
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
#pragma omp task shared(x,n)
    x = fib(n-1);
#pragma omp task shared(y,n)
    y = fib(n-2);
#pragma omp taskwait
    return (x + y);
  }
}
```

Sharing of memory is managed explicitly.

© 2008–2018 by the MIT 6.172 Lecturers

Fibonacci in OpenMP

```c
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
#pragma omp task shared(x,n)
    x = fib(n-1);
#pragma omp task shared(y,n)
    y = fib(n-2);
#pragma omp taskwait
    return (x + y);
  }
}
```

Wait for the two tasks to complete before continuing.

© 2008–2018 by the MIT 6.172 Lecturers

Other OpenMP Features

- OpenMP provides many pragma directives to express common patterns, such as
  - `parallel for` for loop parallelism,
  - `reduction` for data aggregation,
  - directives for scheduling and data sharing.
- OpenMP supplies a variety of synchronization constructs, such as barriers, atomic updates, and mutual-exclusion (mutex) locks.

© 2008–2018 by the MIT 6.172 Lecturers

Outline

- Shared-Memory Hardware
- Concurrency Platforms
  - Pthreads (and WinAPI Threads)
  - Threading Building Blocks
  - OpenMP
  - Cilk

© 2008–2018 by the MIT 6.172 Lecturers

Intel Cilk Plus

- The "Cilk" part is a small set of linguistic extensions to C/C++ to support fork-join parallelism. (The "Plus" part supports vector parallelism.)
- Developed originally by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009.
- Based on the award-winning Cilk multithreaded language developed at MIT.
- Features a provably efficient work-stealing scheduler.
- Provides a hyperobject library for parallelizing code with global variables.
- Ecosystem includes the Cilkscreen race detector and Cilkview scalability analyzer.

© 2008–2018 by the MIT 6.172 Lecturers

Tapir/LLVM and Cilk

6.172 will be using the Tapir/LLVM compiler, which supports the Cilk subset of Cilk Plus.

- Tapir/LLVM was developed at MIT by Tao B. Schardl, William Moses, and Charles Leiserson.
- Tapir/LLVM generally produces better code relative to its base compiler than other implementations of Cilk.
- Tapir/LLVM uses Intel's Cilk Plus runtime system.
- Tapir/LLVM also supports more general features, such as the spawning of code blocks.

© 2008–2018 by the MIT 6.172 Lecturers

Nested Parallelism in Cilk

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

The named child function may execute in parallel with the parent caller.

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

Cilk keywords grant permission for parallel execution. They do not command parallel execution.

© 2008–2018 by the MIT 6.172 Lecturers

Loop Parallelism in Cilk

Example: In-place matrix transpose

$$\underset{A}{\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}} \Longrightarrow \underset{A^T}{\begin{pmatrix} a_{11} & a_{21} & \cdots & a_{n1} \\ a_{12} & a_{22} & \cdots & a_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{pmatrix}}$$

The iterations of a cilk_for loop execute in parallel.

```
// indices run from 0, not 1
cilk_for (int i=1; i<n; ++i) {
  for (int j=0; j<i; ++j) {
    double temp = A[i][j];
    A[i][j] = A[j][i];
    A[j][i] = temp;
  }
}
```

Reducers in Cilk

Example:
Parallel summation

```
unsigned long sum = 0;
for (int i=0; i<n; ++i) {
  sum += i;
}
printf("%d\n",sum);
```

```
CILK_C_REDUCER_OPADD(sum, unsigned long, 0);
CILK_C_REGISTER_REDUCER(sum);
cilk_for(int i=0; i<n; ++i) {
  REDUCER_VIEW(sum) += i;
}
printf("The sum is %f\n", REDUCER_VIEW(sum));
CILK_C_UNREGISTER_REDUCER(sum);
```

© 2008–2018 by the MIT 6.172 Lecturers

Reducers in Cilk

Reducers can be created for monoids (algebraic structures with an associative binary operation and an identity element)

Cilk has several predefined reducers (add, multiply, min, max, and, or, xor, etc.)

© 2008–2018 by the MIT 6.172 Lecturers

Serial Semantics

Cilk source

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

serial elision

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
    x = fib(n-1);
    y = fib(n-2);

    return (x + y);
  }
}
```

The serial elision of a Cilk program is always a legal interpretation of the program’s semantics.

Remember, Cilk keywords grant permission for parallel execution. They do not command parallel execution.

To obtain the serial elision:

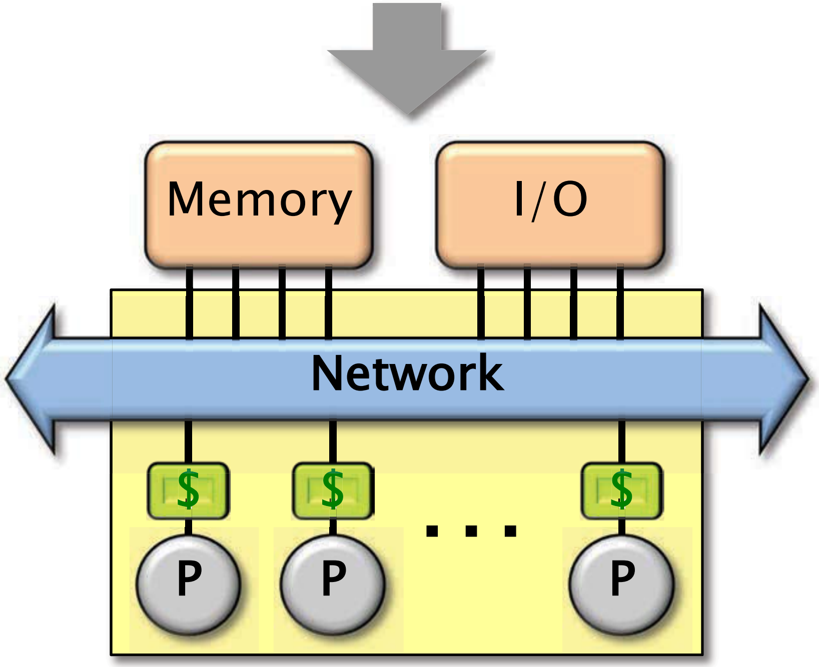
```
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```

© 2008–2018 by the MIT 6.172 Lecturers

Scheduling

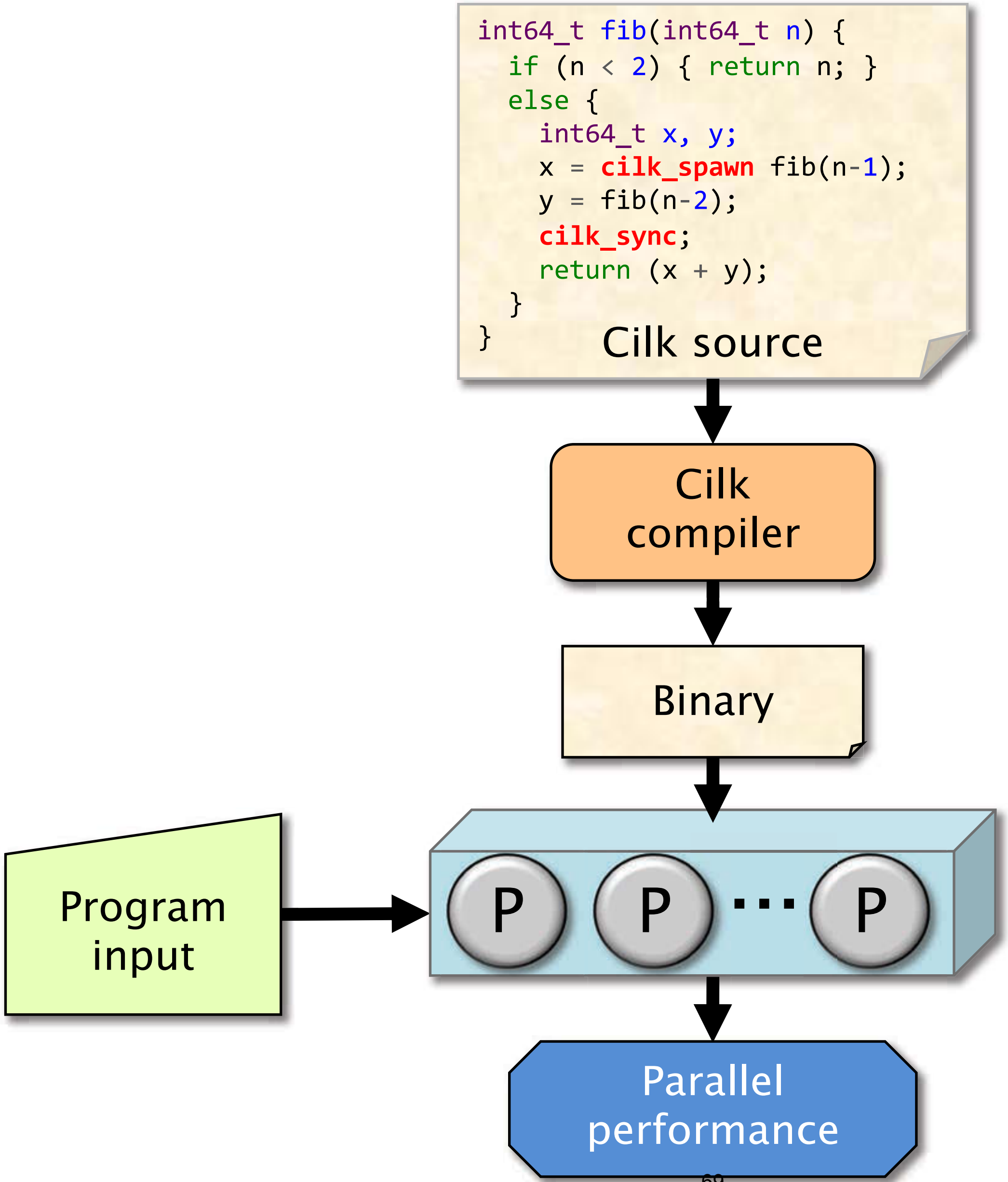
- The Cilk concurrency platform allows the programmer to express logical parallelism in an application.
- The Cilk scheduler maps the executing program onto the processor cores dynamically at runtime.
- Cilk’s work-stealing scheduling algorithm is provably efficient.

```
int64_t fib(int64_t n) {
  if (n < 2) {
    return n;
  } else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```



© 2008–2018 by the MIT 6.172 Lecturers

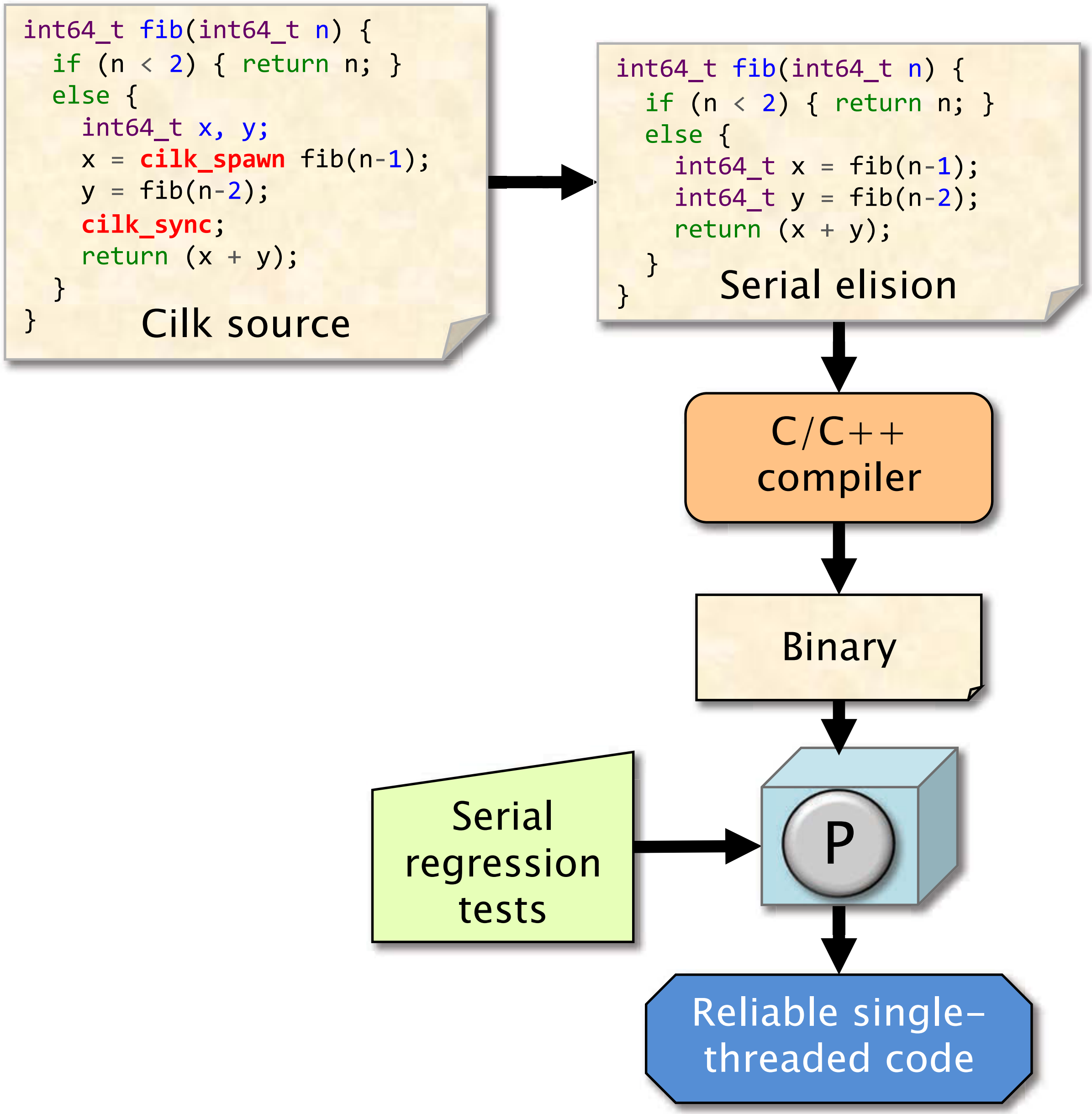
Cilk Platform



```
int64_t fib(int64_t n) {
  if (n < 2) { return n; }
  else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

© 2008–2018 by the MIT 6.172 Lecturers

Serial Testing



```
int64_t fib(int64_t n) {
  if (n < 2) { return n; }
  else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

Cilk source

```
int64_t fib(int64_t n) {
  if (n < 2) { return n; }
  else {
    int64_t x = fib(n-1);
    int64_t y = fib(n-2);
    return (x + y);
  }
}
```

Serial elision

C/C++ compiler

Binary

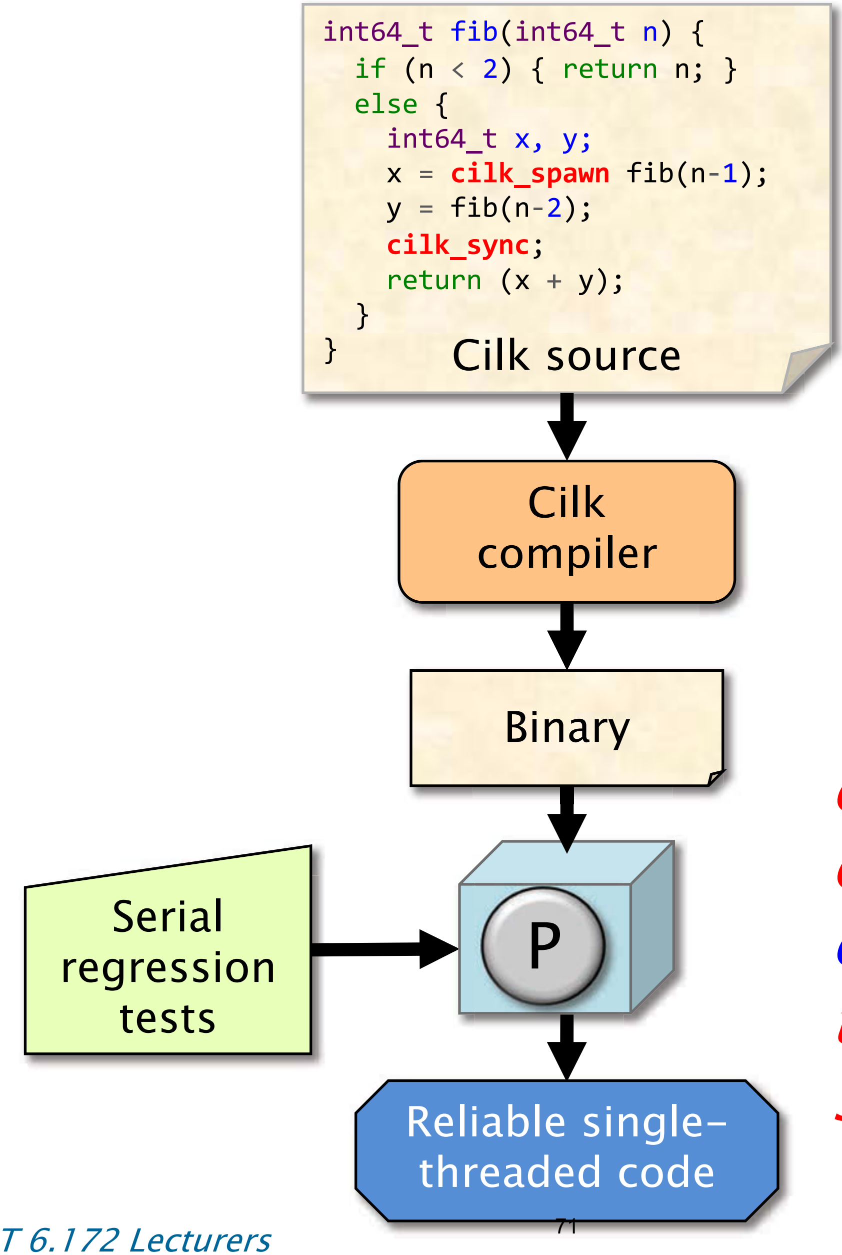
Serial regression tests

P

Reliable single-threaded code

© 2008–2018 by the MIT 6.172 Lecturers

Alternative Serial Testing

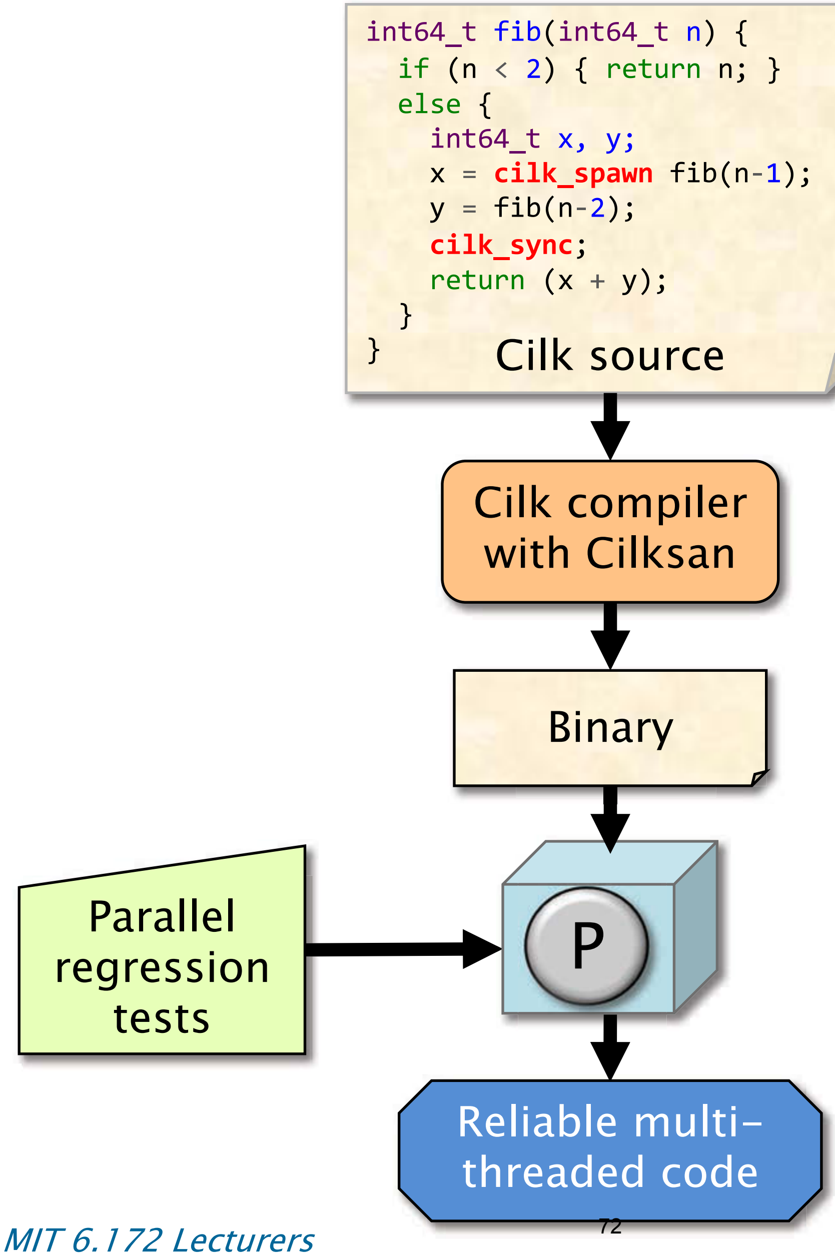


```
int64_t fib(int64_t n) {
  if (n < 2) { return n; }
  else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

The parallel program executing on one core should behave exactly the same as the execution of the serial elision.

© 2008–2018 by the MIT 6.172 Lecturers

Parallel Testing



```
int64_t fib(int64_t n) {
  if (n < 2) { return n; }
  else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

Cilk source

Cilk compiler with Cilksan

Binary

Parallel regression tests

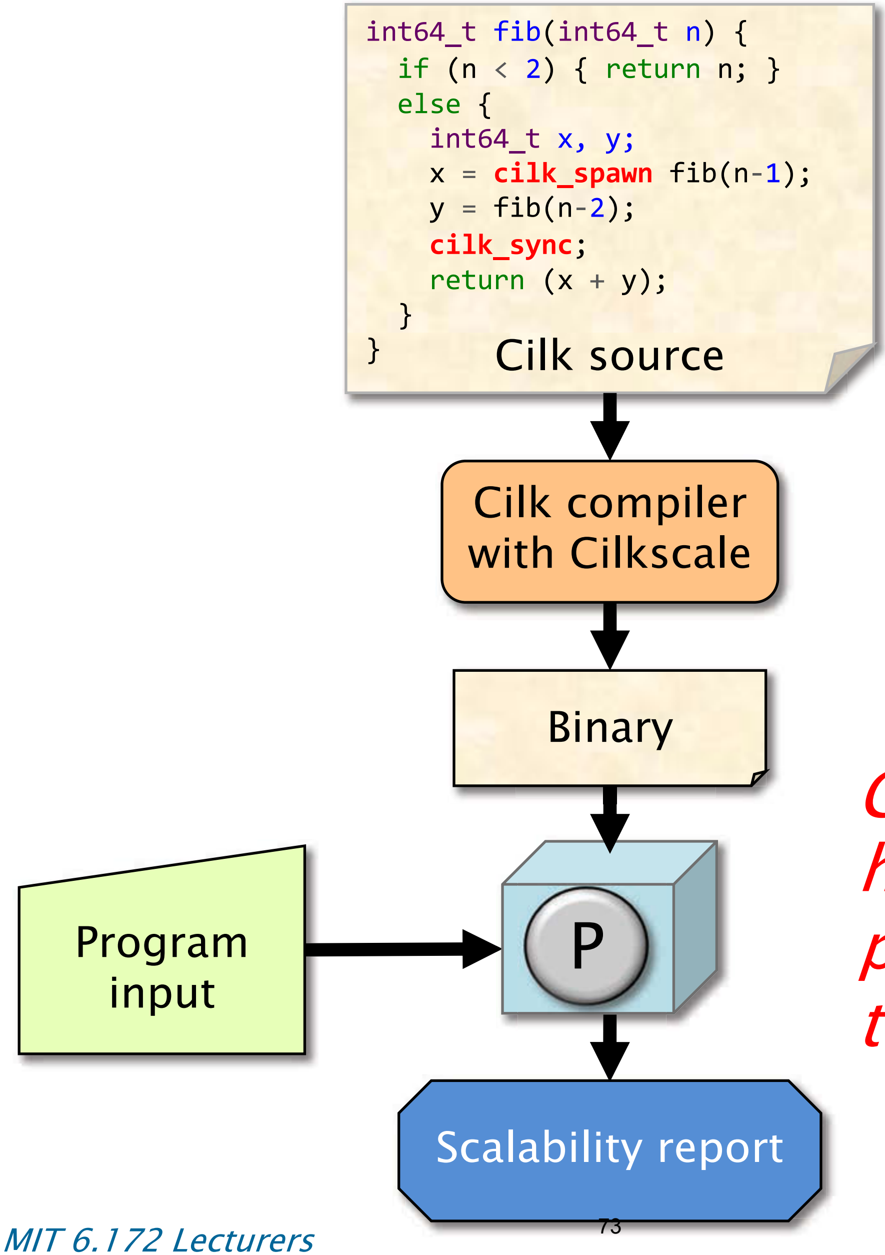
P

Reliable multi-threaded code

Cilksan finds and localizes determinacy races.

© 2008–2018 by the MIT 6.172 Lecturers

Scalability Analysis



```
int64_t fib(int64_t n) {
  if (n < 2) { return n; }
  else {
    int64_t x, y;
    x = cilk_spawn fib(n-1);
    y = fib(n-2);
    cilk_sync;
    return (x + y);
  }
}
```

Cilkscale analyzes how well your program will scale to larger machines.

© 2008–2018 by the MIT 6.172 Lecturers

Summary

- Processors today have multiple cores, and obtaining high performance requires parallel programming
- Programming directly on processor cores is painful and error-prone.
- Cilk abstracts processor cores, handles synchronization and communication protocols, and performs provably efficient load balancing.
- Project 2: Parallel screen saver using Cilk

© 2008–2018 by the MIT 6.172 Lecturers

MIT OpenCourseWare
https://ocw.mit.edu

6.172 Performance Engineering of Software Systems
Fall 2018

For information about citing these materials or our Terms of Use, visit: https://ocw.mit.edu/terms.