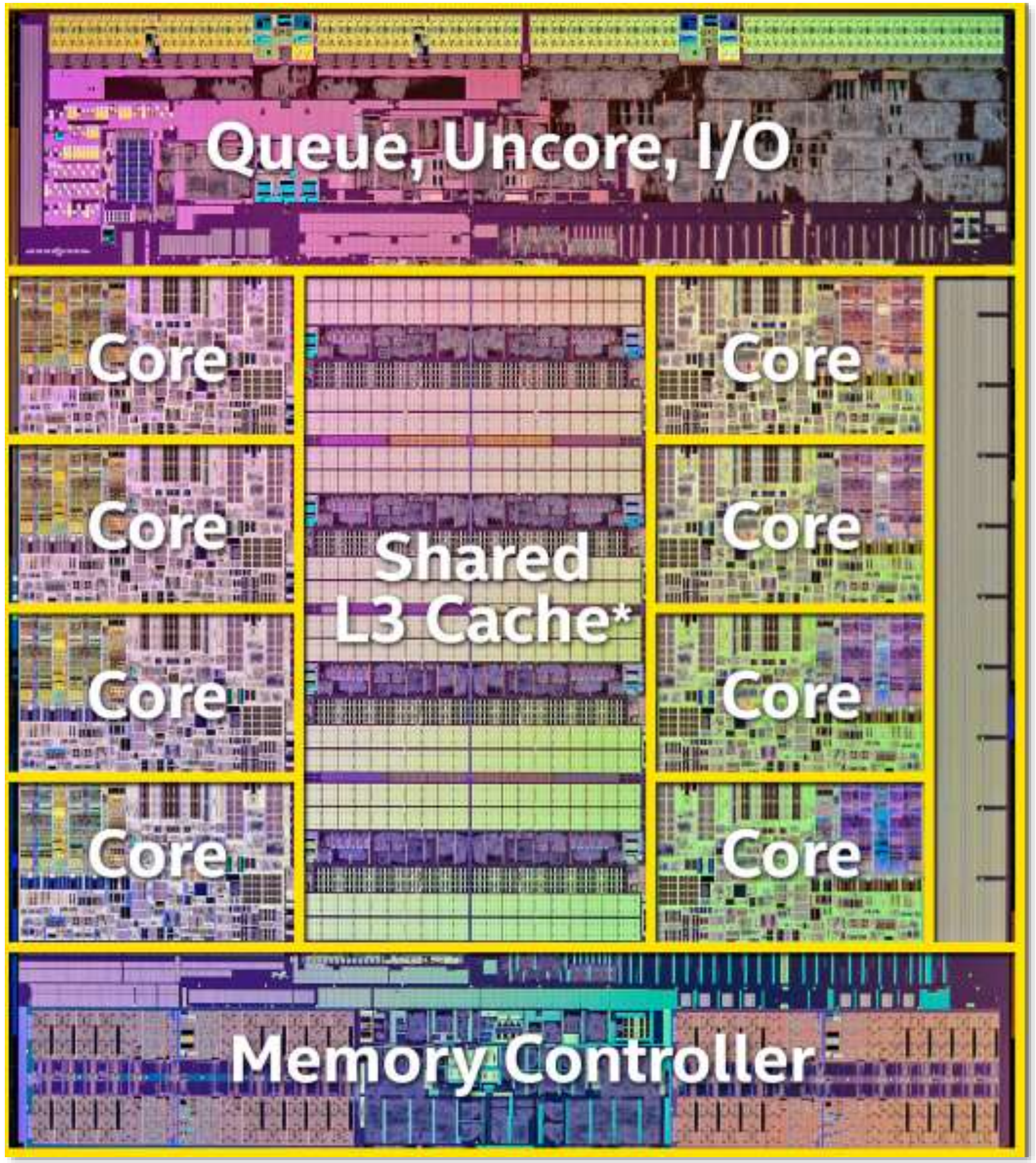
Performance Engineering of Software Systems

Lecture 7
Multicore Programming

Srini Devadas

September 29, 2022

Multicore Processors

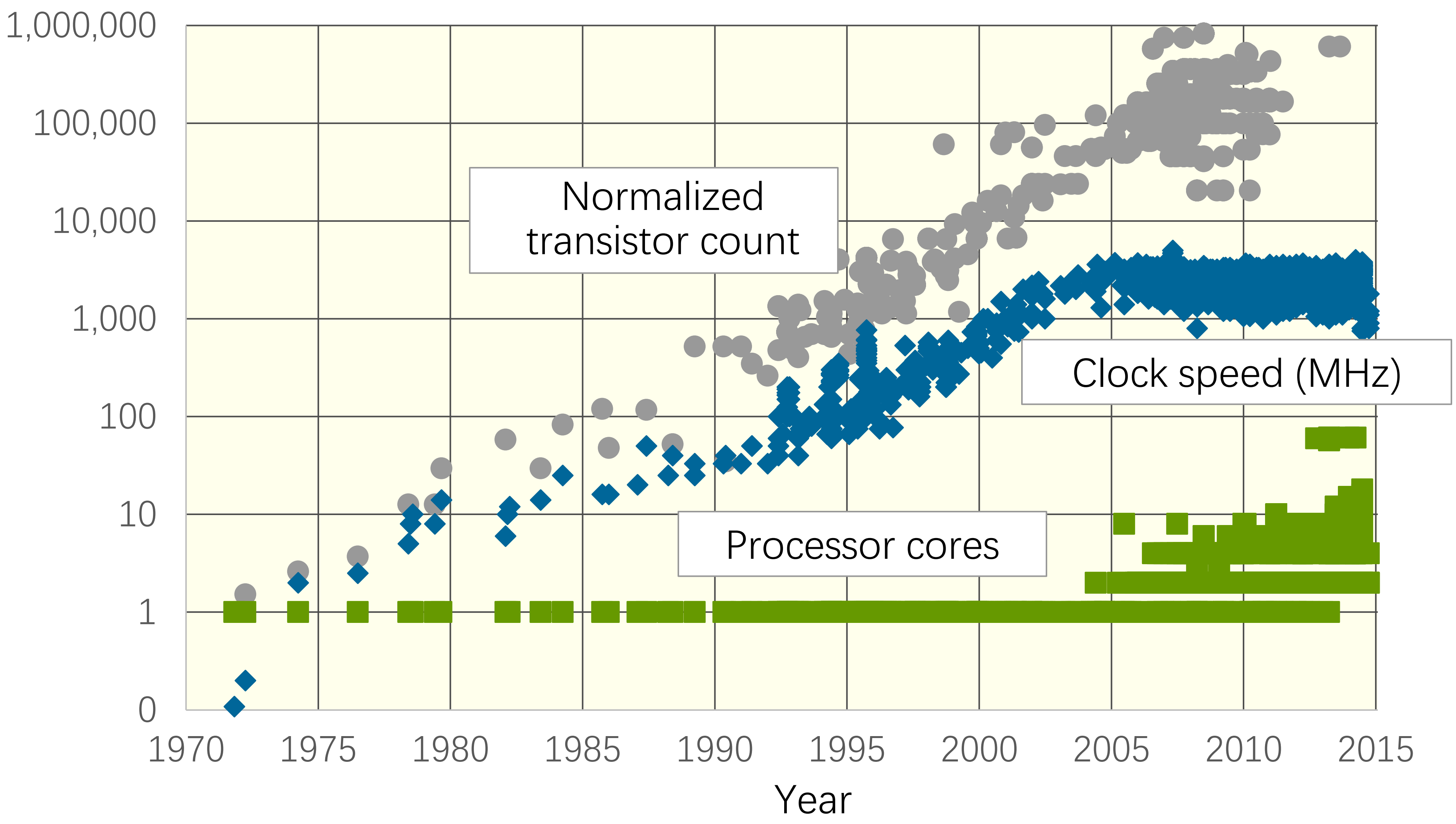


Intel Haswell-E

Q Why do modern chips have multiple processor cores?

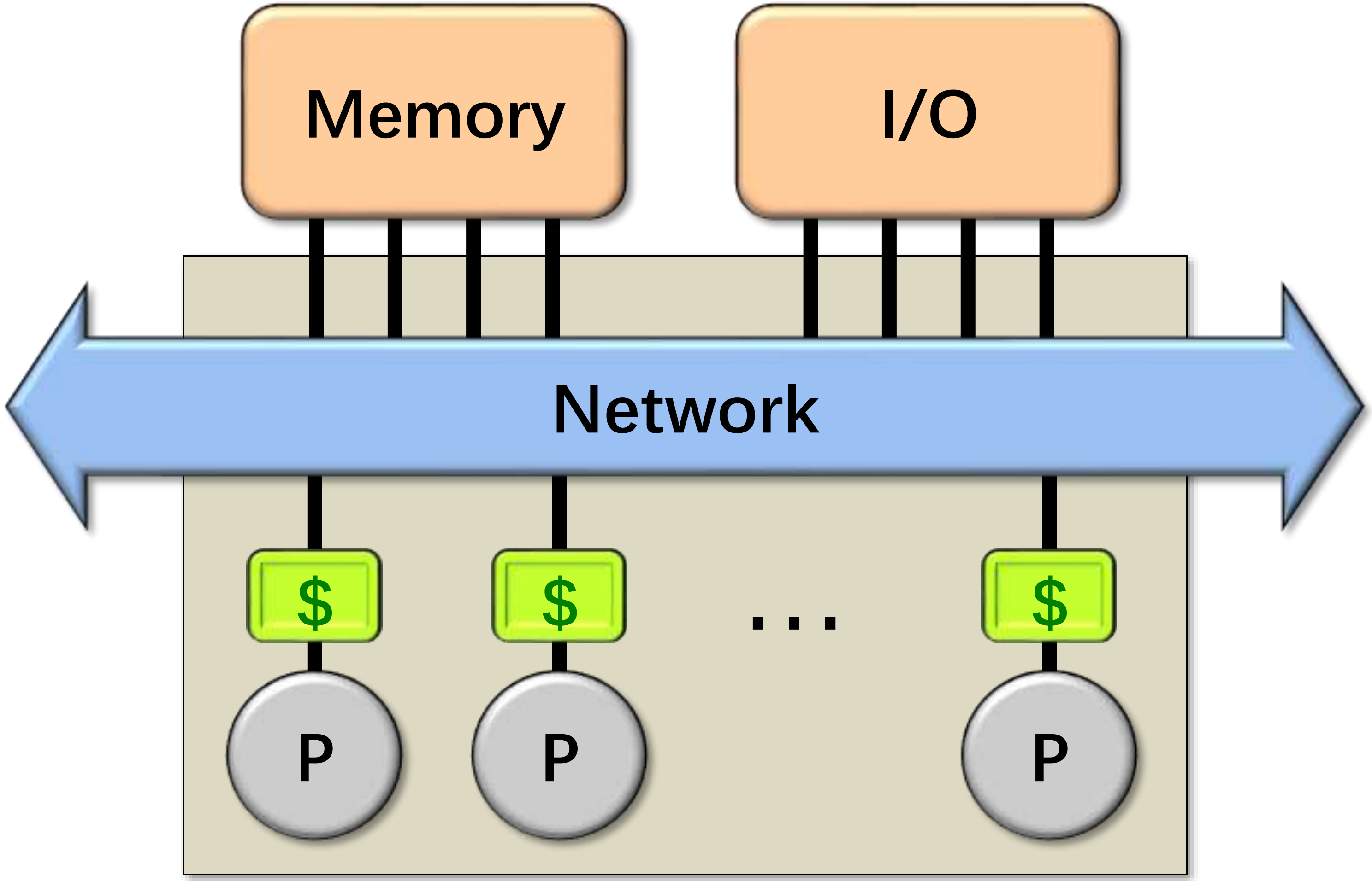
A Because of Moore's Law, the end of the scaling of clock frequency, and diminishing returns to ILP.

Technology Scaling



Processor data from Stanford's CPU DB [DKM12].

Abstract Multicore Architecture

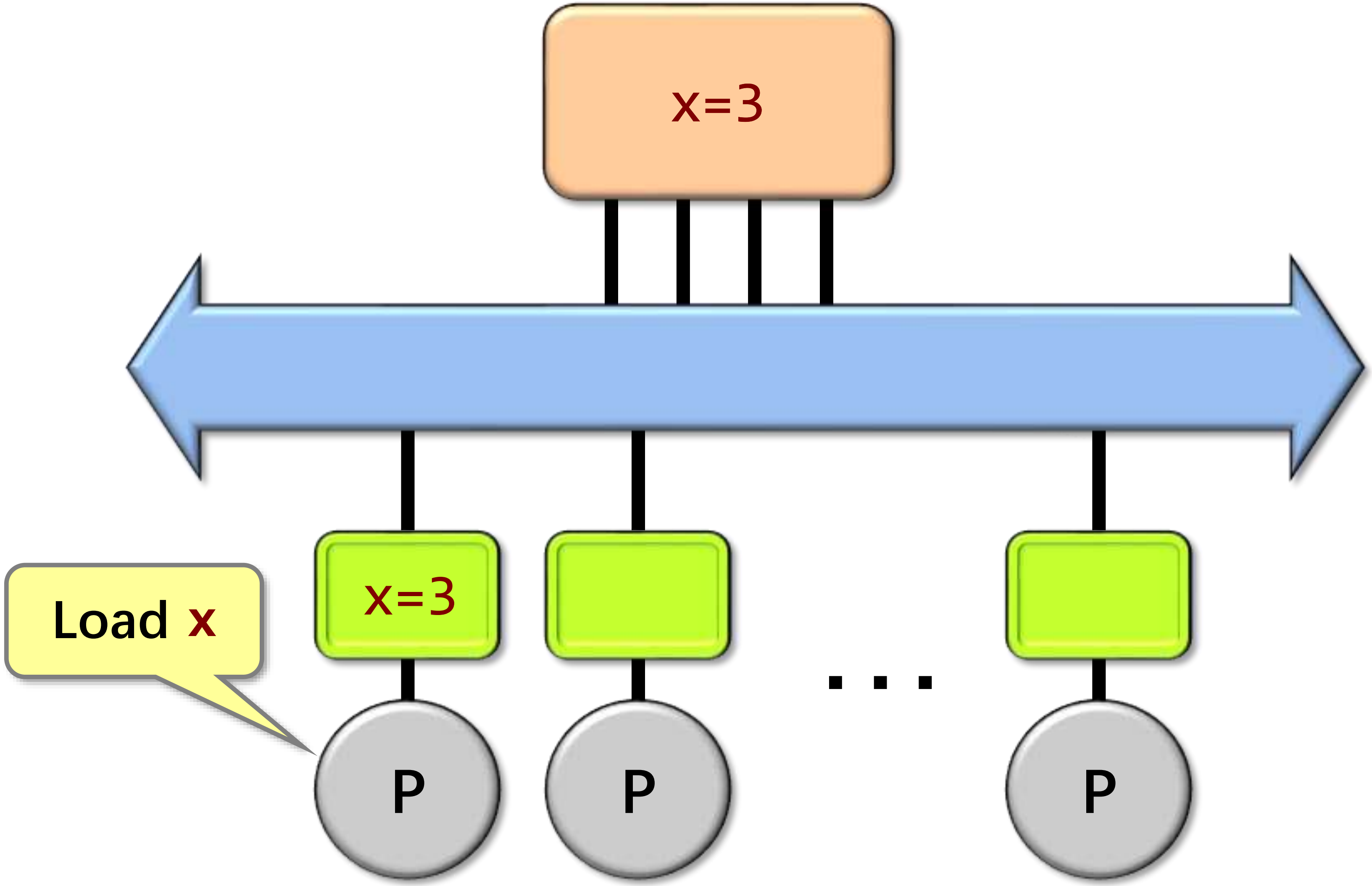


Chip Multiprocessor (CMP)

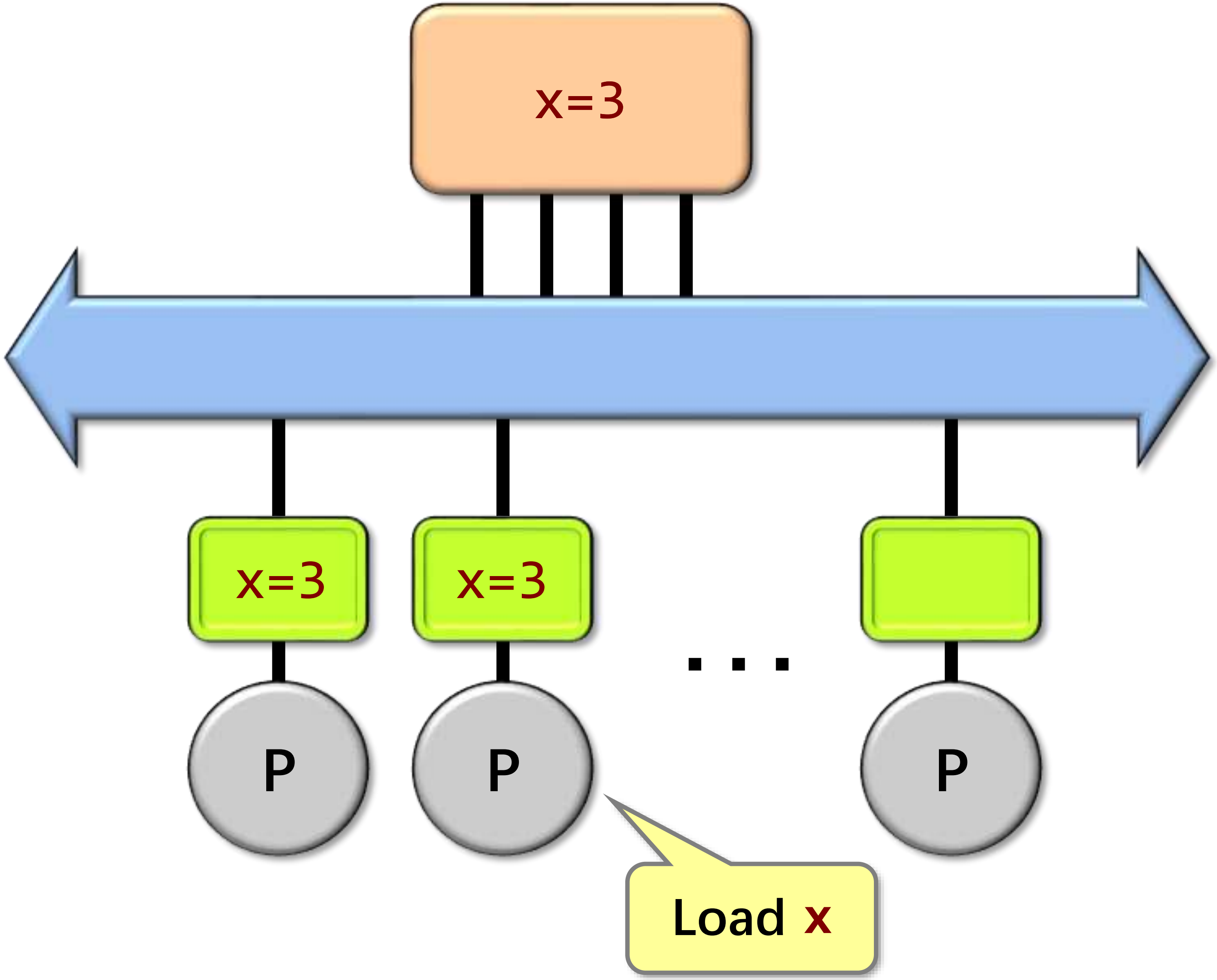
OUTLINE

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

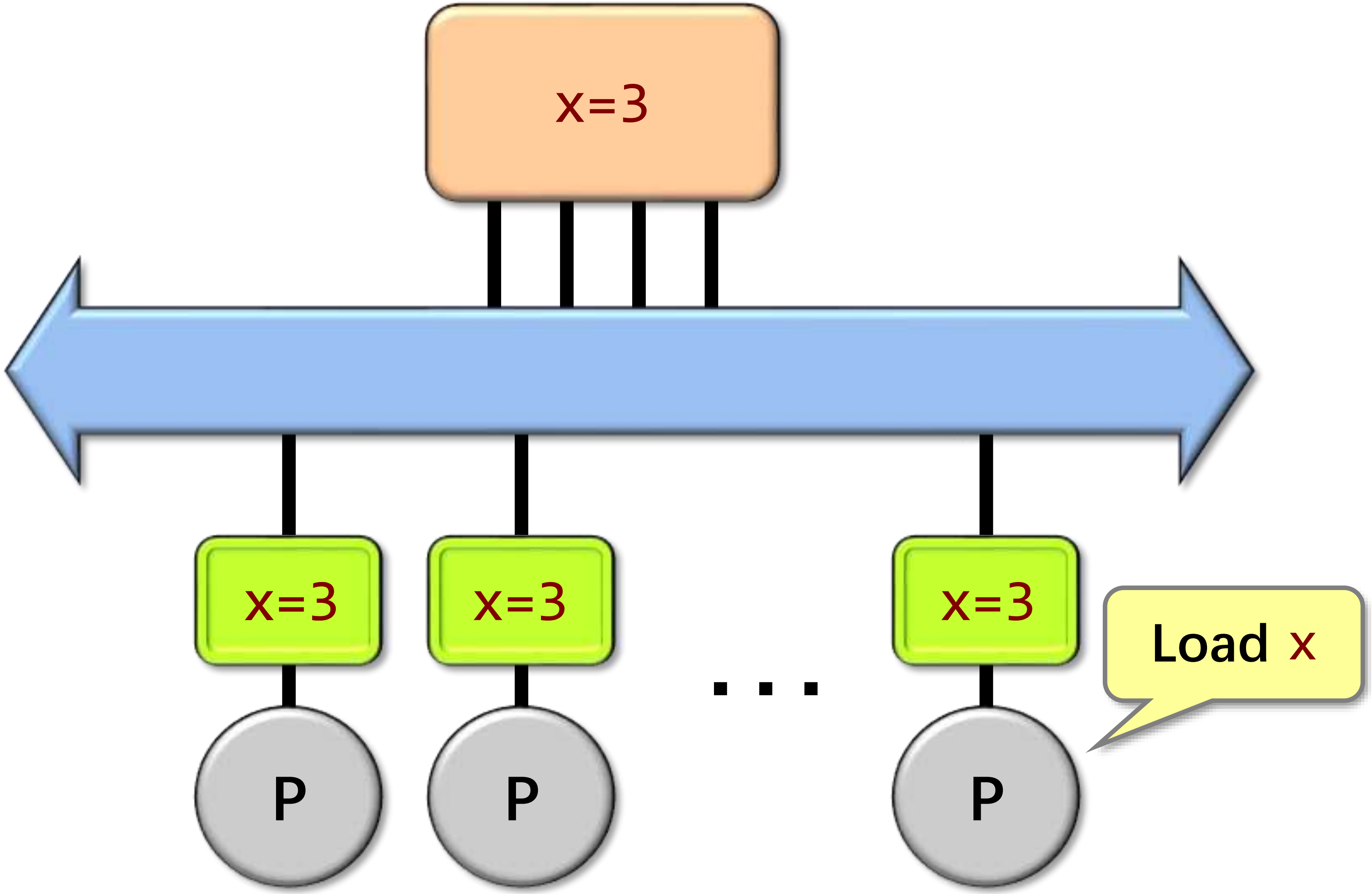
Cache Coherence



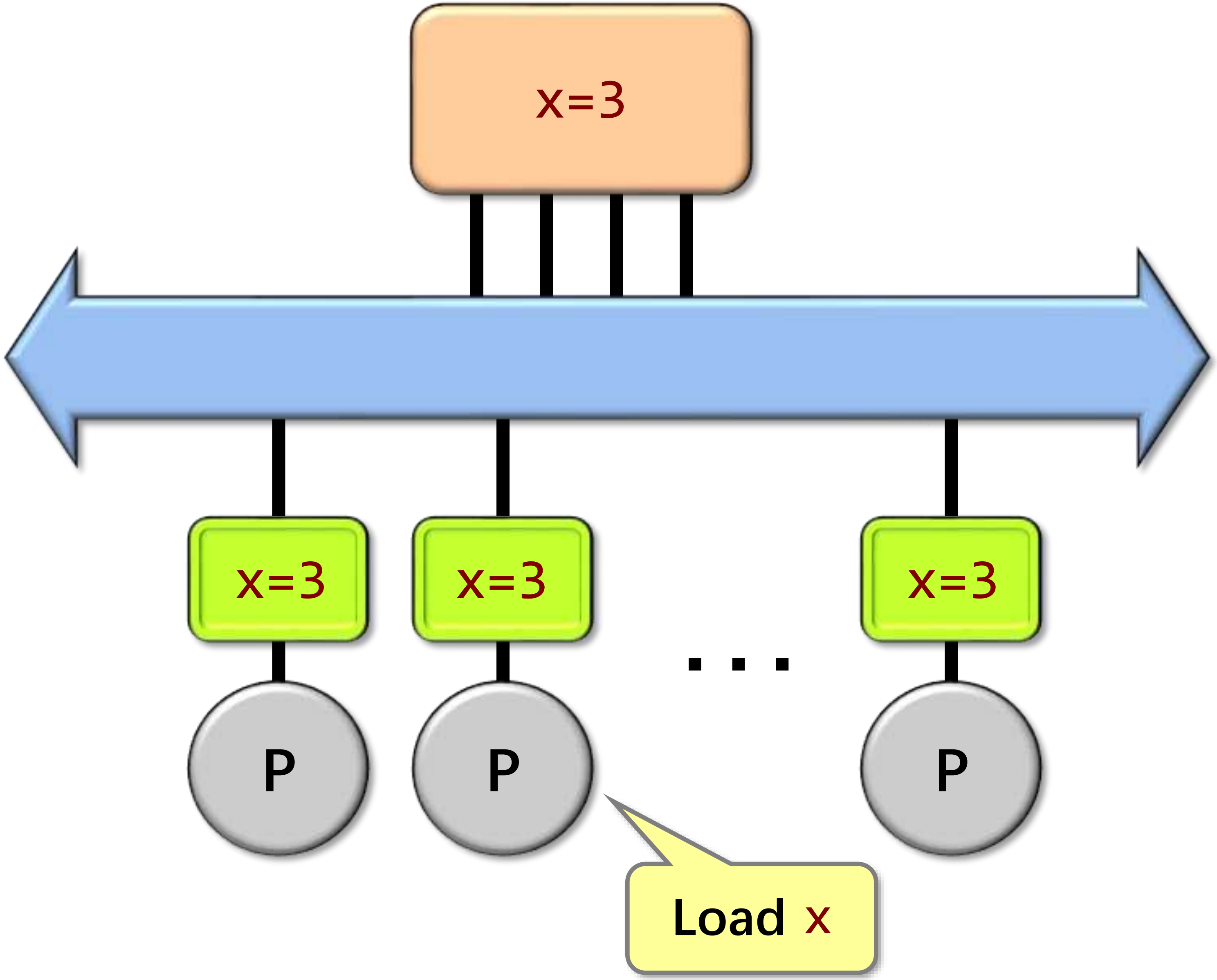
Cache Coherence



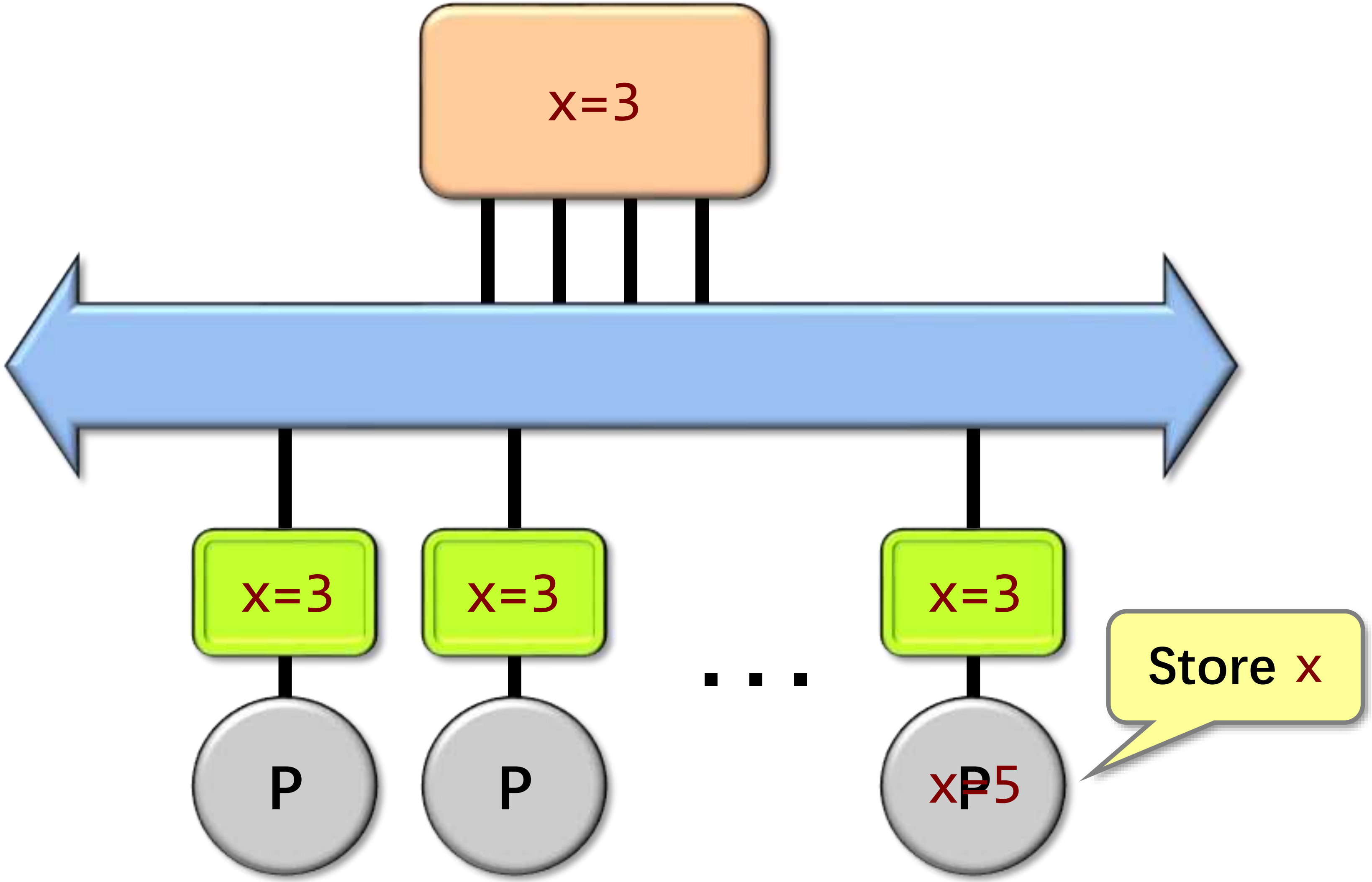
Cache Coherence



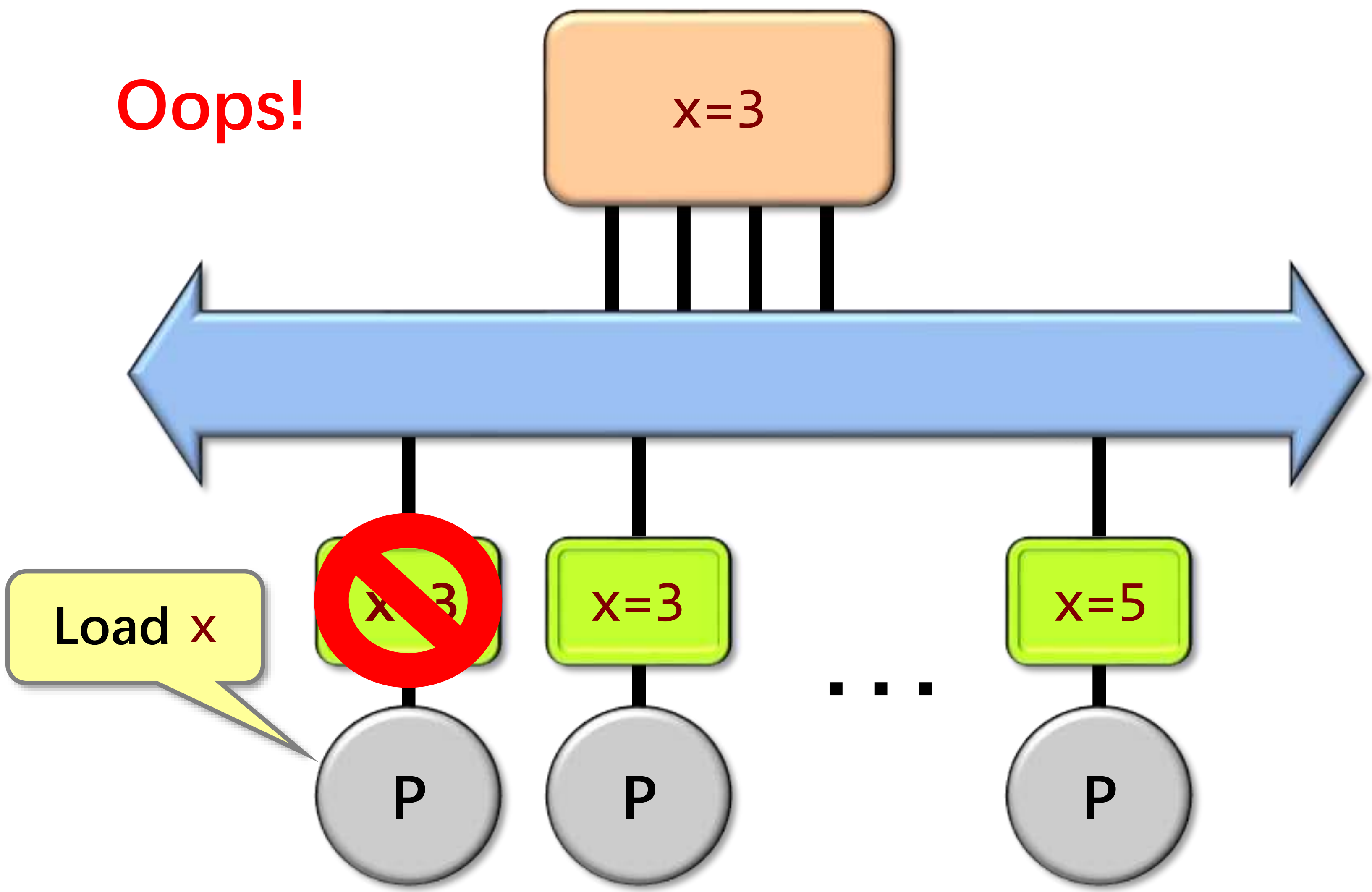
Cache Coherence



Cache Coherence



Cache Coherence



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).

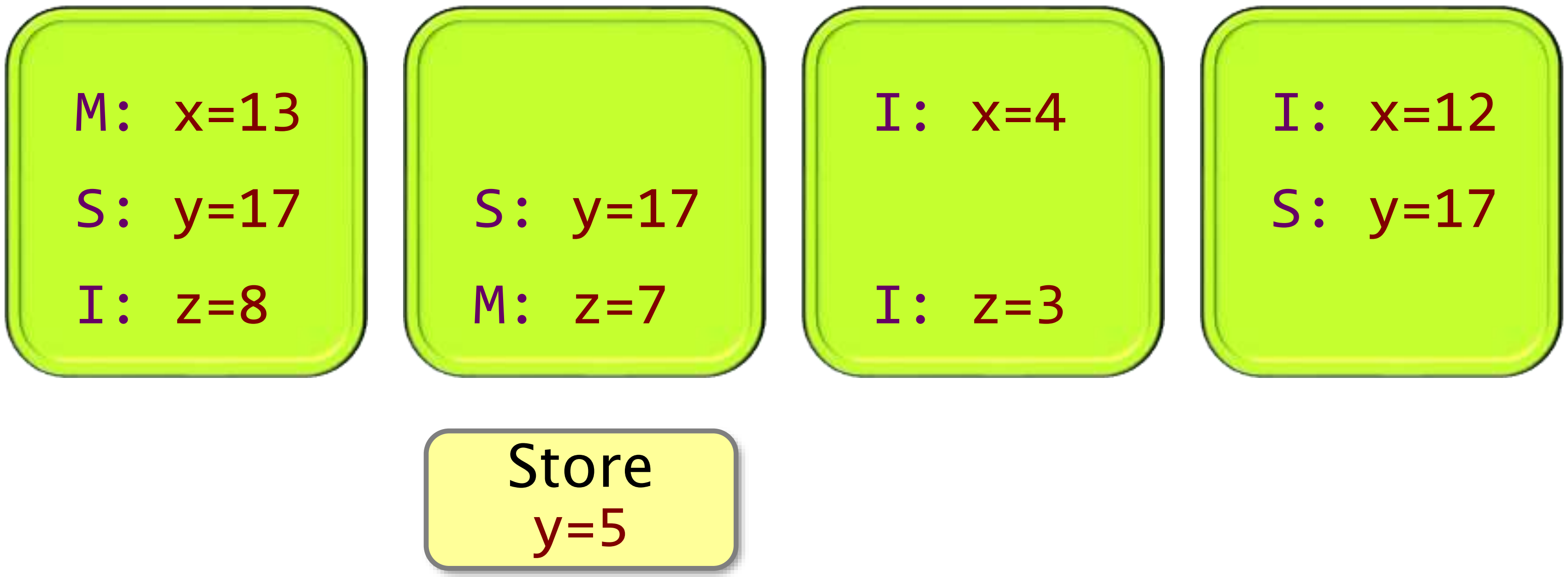
M: x=13		I: x=4	I: x=12
S: y=17	S: y=17		S: y=17
I: z=8	M: z=7	I: z=3	

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

MSI Protocol

Each cache line is labeled with a state:

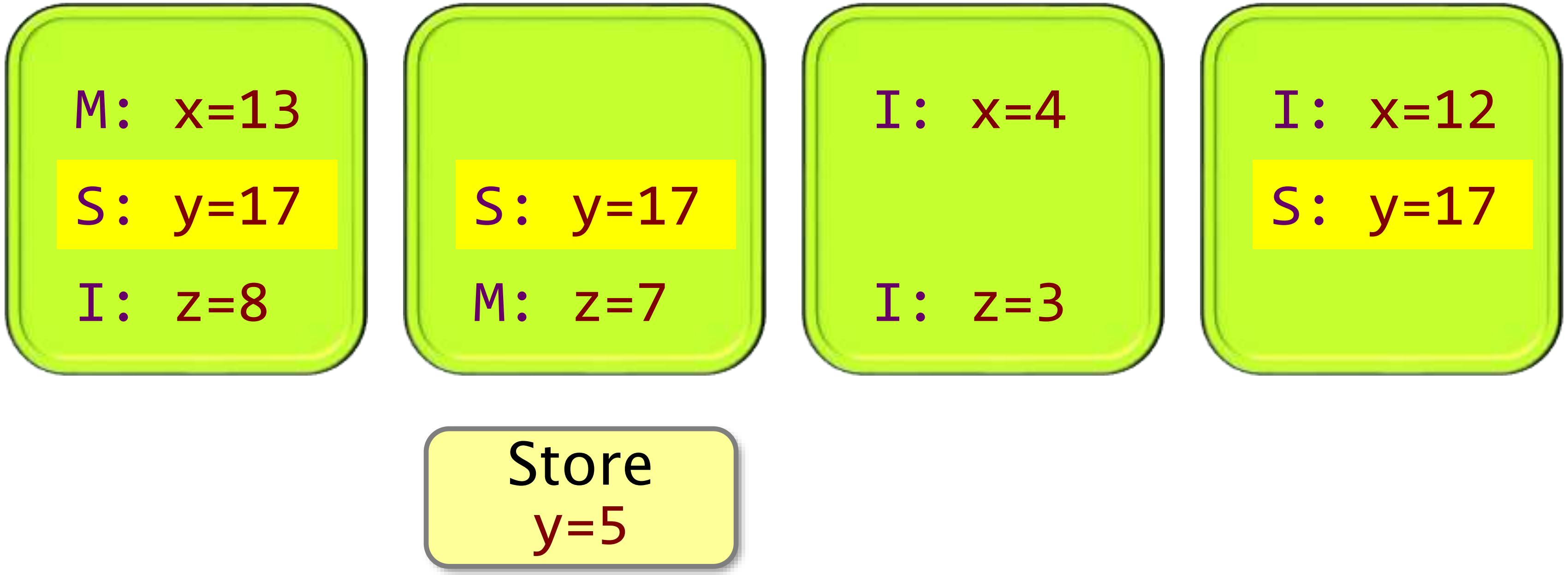
- **M:** cache block has been modified. No other caches contain this block in M or S states.
- **S:** other caches may be sharing this block.
- **I:** cache block is invalid (same as not there).



MSI Protocol

Each cache line is labeled with a state:

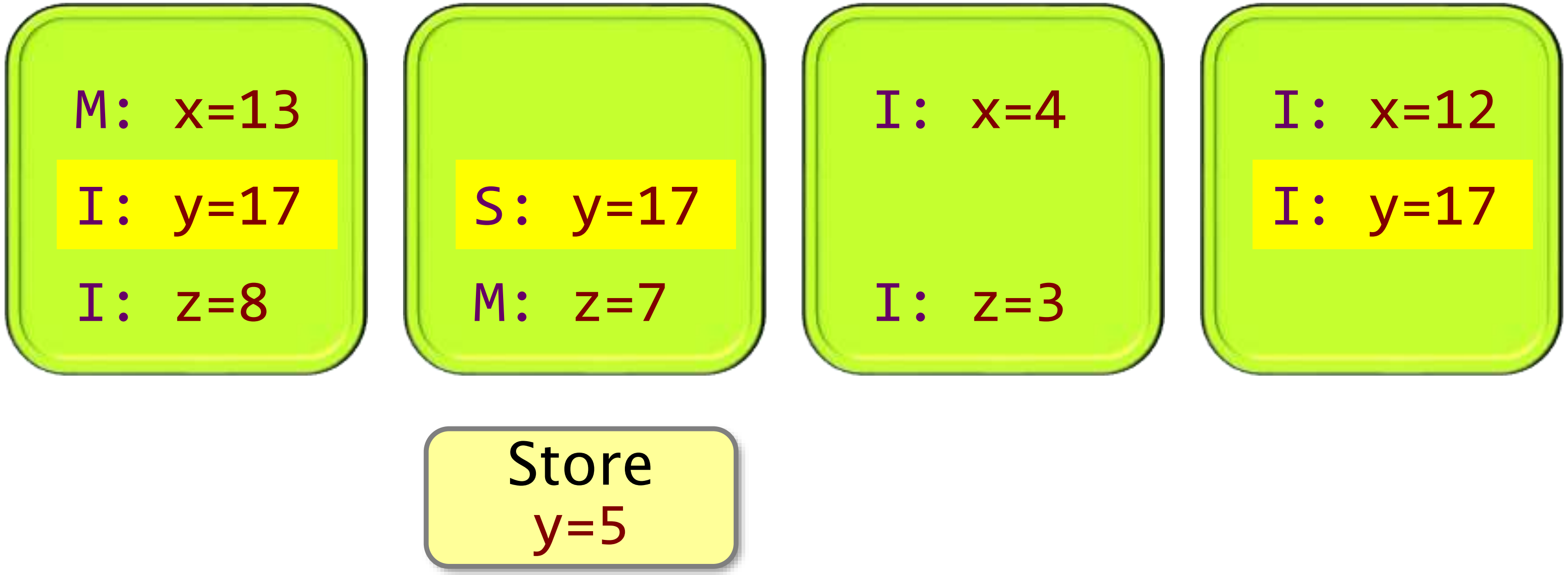
- **M:** cache block has been modified. No other caches contain this block in M or S states.
- **S:** other caches may be sharing this block.
- **I:** cache block is invalid (same as not there).



MSI Protocol

Each cache line is labeled with a state:

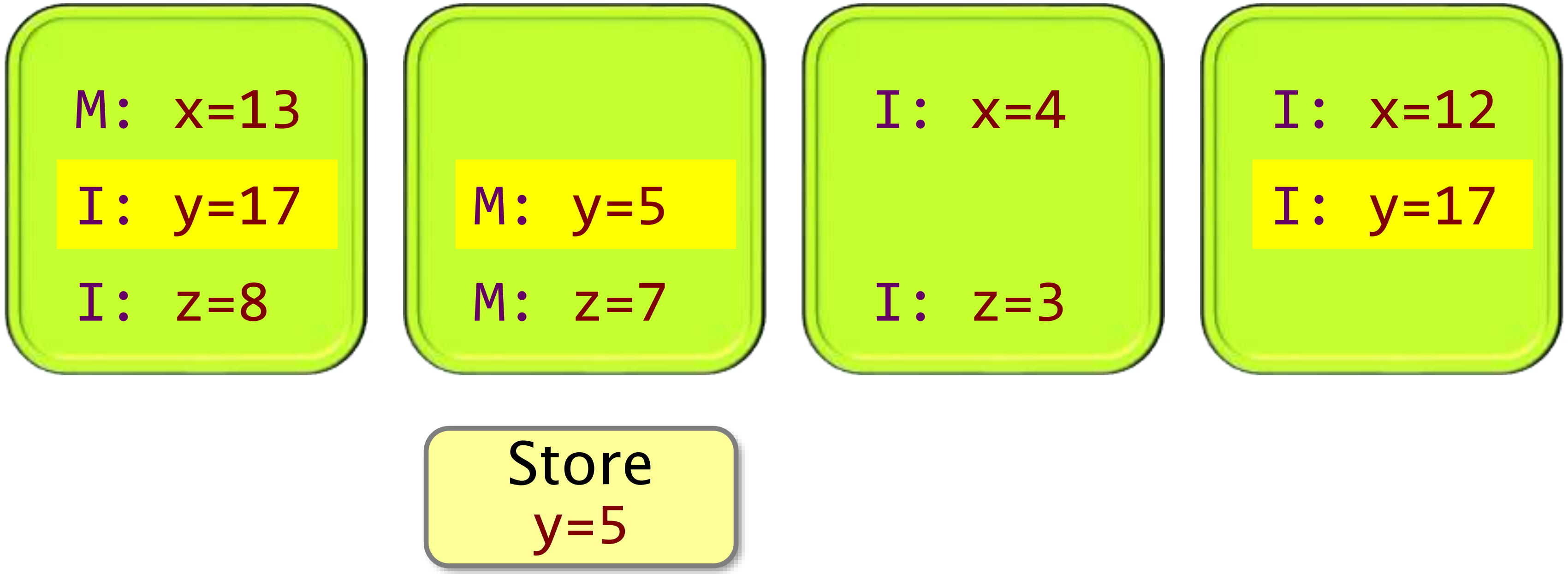
- **M:** cache block has been modified. No other caches contain this block in M or S states.
- **S:** other caches may be sharing this block.
- **I:** cache block is invalid (same as not there).



MSI Protocol

Each cache line is labeled with a state:

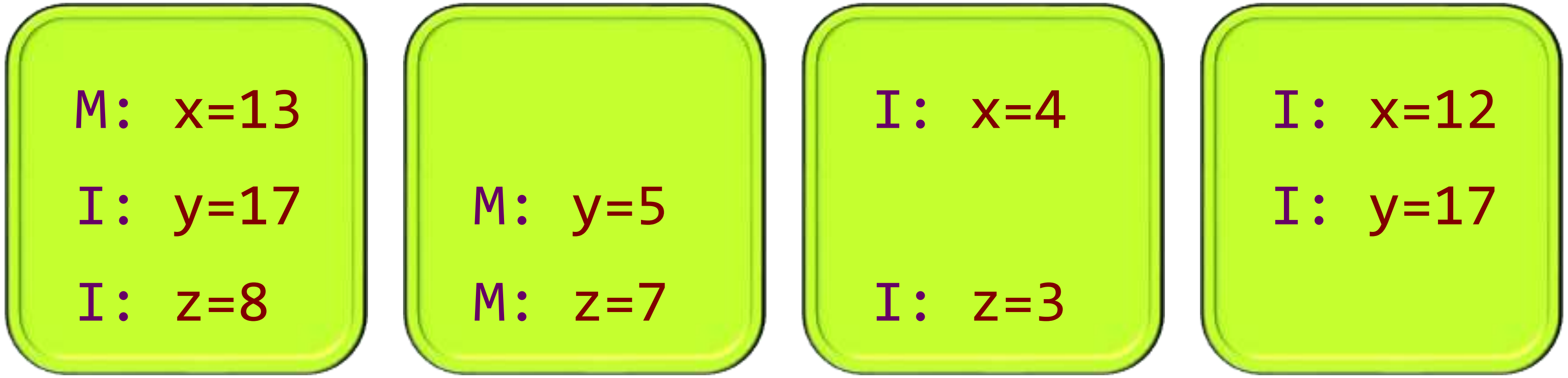
- **M:** cache block has been modified. No other caches contain this block in M or S states.
- **S:** other caches may be sharing this block.
- **I:** cache block is invalid (same as not there).



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).



Outline

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

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**

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:

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

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.

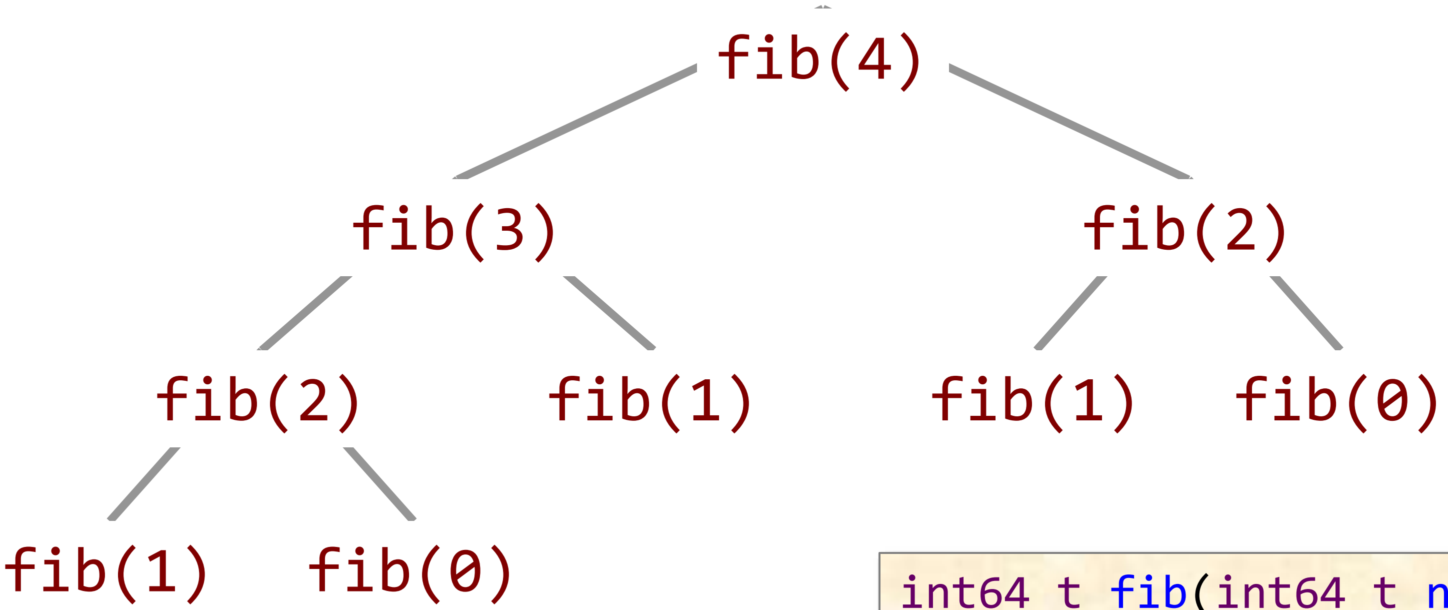
Fibonacci Program

```
#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 a good didactic example.

Fibonacci Execution



★ Key idea for parallelization

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

```c
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);
  }
}
```

OUTLINE

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

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 inter-thread coordination.

---

*WinAPI threads provide similar functionality.

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
```

Pthread Fib 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;
}
```

Pthread Fib Implementation

Original code.

```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;
}
```

Pthread Fib 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;
}
```

Structure for thread arguments.

```
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;
}
```

Pthread Fib 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;
}
```

Pthread Fib Implementation

```c
#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.

```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;
}
```

Pthread Fib 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], 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.

Pthread Fib 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;
}
```

Create thread to execute fib(n–1)

Pthread Fib 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(voi
  int64_t i = ((thread
  ((thread_args *) ptr)->output = fib(i);
  return NULL;
}
```

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

```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;
}
```

Pthread Fib 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.

Pthread Fib 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.

Issues with Pthreads

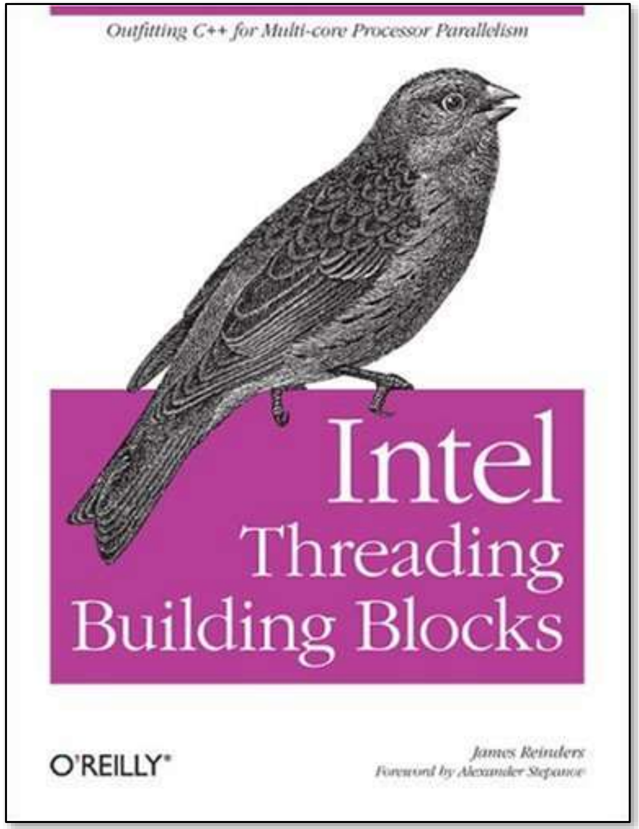
Overhead	The cost of creating a thread >10^4 cycles ⇒ coarse-grained concurrency. (Thread pools can help.)
Scalability	Fibonacci code gets at most about 1.5 speedup for 2 cores. Need a rewrite for more cores.
Modularity	The Fibonacci logic is no longer neatly encapsulated in the `fib()` function.
Code Simplicity	Programmers must marshal arguments (shades of 1957!) and engage in error-prone protocols in order to load-balance.

Outline

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

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 on Cilk at MIT
- Focus on **performance**



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;
}
```

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;
}
```

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;
}
```

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* su
          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;
}
```

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;
}
```

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     ld() )
                         FibTas   -1, &x);
      FibTask& b = *new( allo   ate_child() )
                         F   Task(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).

```
#inc      stdint>
#         <iostream>
     ude "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;
}
```

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;
}
```

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>
#         e "tbb/task.h"

  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;
}
```

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(      &x);
      FibTask& b = *new( alloca   child() )
                          FibT    (n-2, &y);
      set_ref_count(3);
      spawn(b);
      spawn_and_wait_f   _all(a);
      *sum = x + y;
    }
    return NULL;
  }
};
```

Add the results together to produce the final output.

```
#i         cstdint>
        e <iostream>
      ude "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;
}
```

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_chil
                         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;
}
```

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**.

Outline

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

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);
  }
}
```

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.

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);
  }
}
```

Sharing of memory is managed explicitly.

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);
  }
}
```

Wait for the two tasks to complete before continuing.

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,
  - Mutual-exclusion (mutex) locks

Outline

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

Cilk

- Award-winning multithreading language developed at MIT
- Provides a small set of **linguistic extensions to C/C++** to support **fork-join parallelism**.
- Features a provably efficient **work-stealing** scheduler.
- Provides a **reducer** linguistic interface for parallelizing code with global variables.
- Ecosystem includes a **race detector** and a **scalability analyzer**

OpenCilk

This class will be using the **OpenCilk** compiler

- OpenCilk is based on **Tapir/LLVM**, which was developed at MIT
  - By Tao B. Schardl, William S. Moses, and Charles E. Leiserson
- OpenCilk's compiler can generally produce **better code** than can other compilers for parallel-language constructs.
- The OpenCilk **runtime system** is based on **Cheetah**
  - By I-Ting Angelina Lee at Washington University in St. Louis
- OpenCilk includes the **Cilksan** race detector and the **Cilkscale** scalability analyzer

Nested Parallelism in Cilk

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

Loop Parallelism in Cilk

Example:
In-place matrix transpose

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

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;
  }
}
```

Serial Semantics

Cilk source

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

serial projection

```
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 projection** 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.

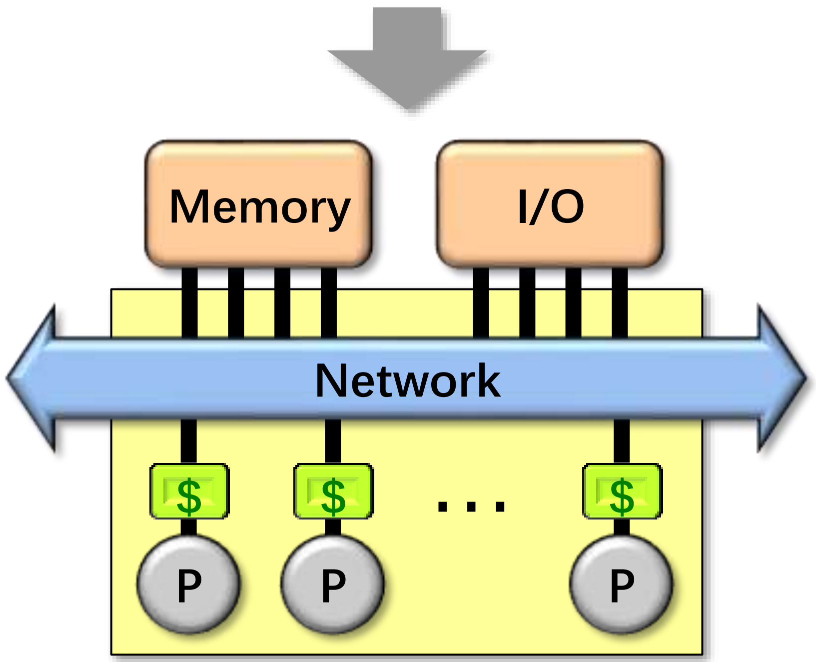
To obtain the serial projection:

```
#define cilk_for for
#define cilk_spawn
#define cilk_scope
```

Scheduling

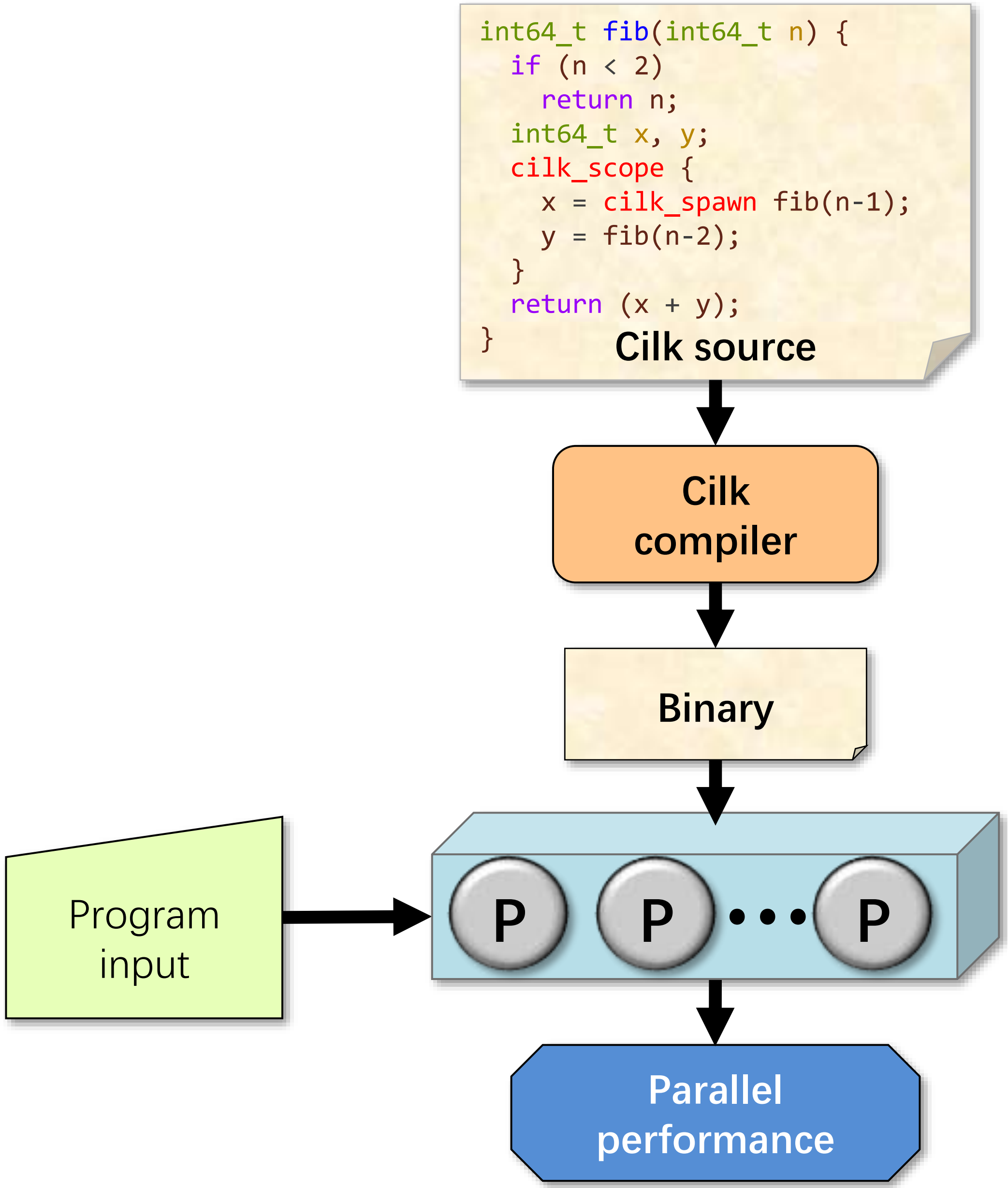
- The Cilk concurrency platform allows the programmer to express **logical parallelism** in an application.
- The Cilk **scheduler** maps the executing program onto the processor cores dynamically at runtime.
- Cilk's **work-stealing scheduling algorithm** is provably efficient.

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

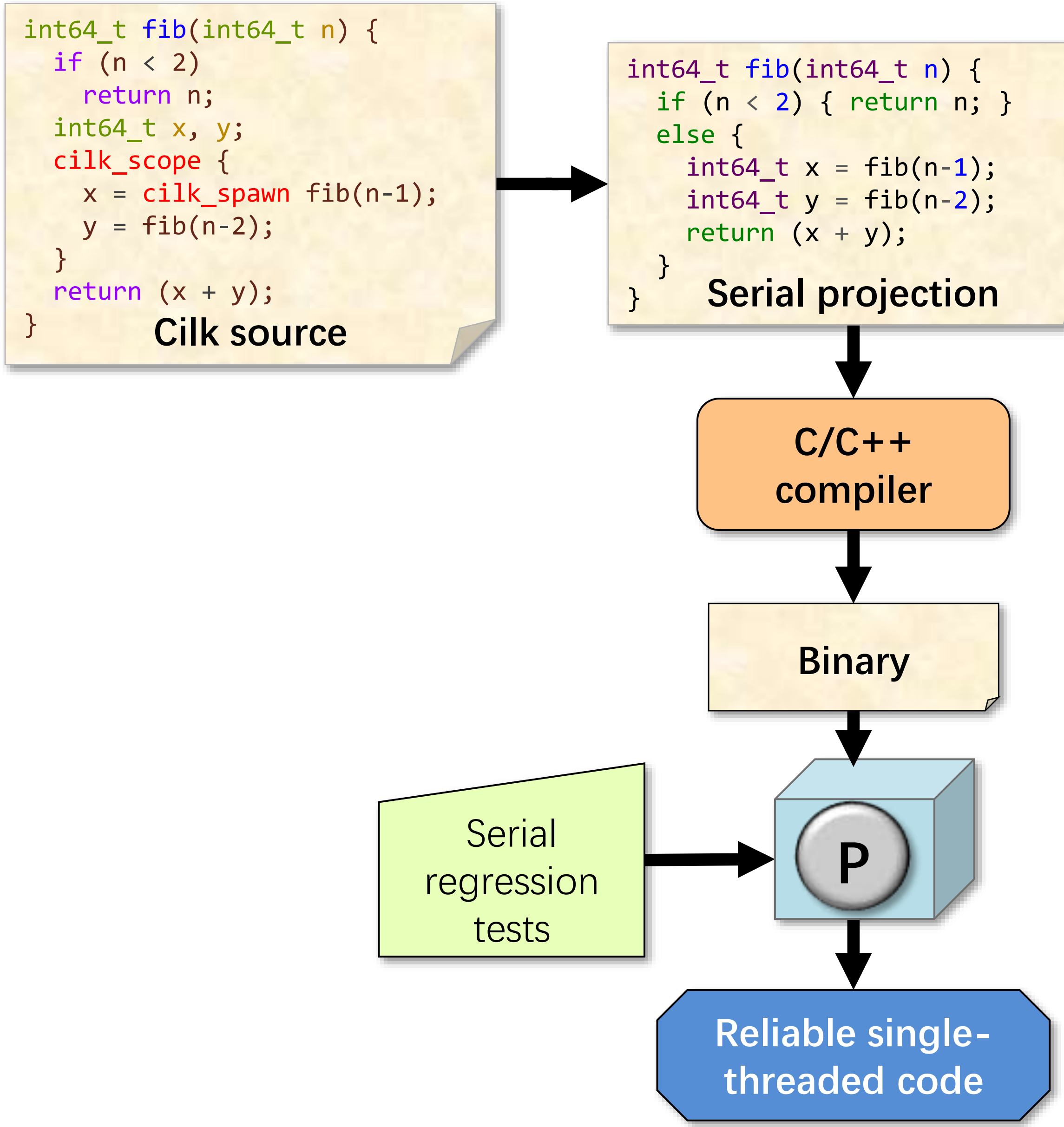


Cilk Platform

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

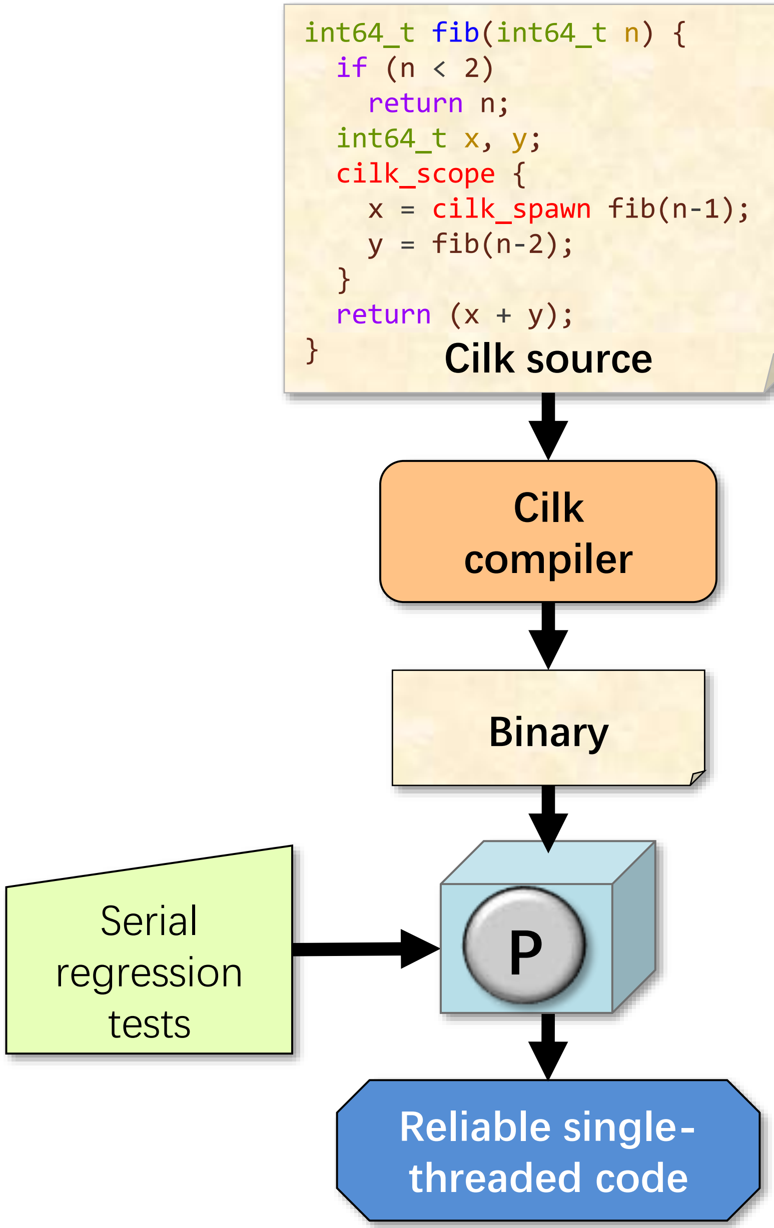


Serial Testing



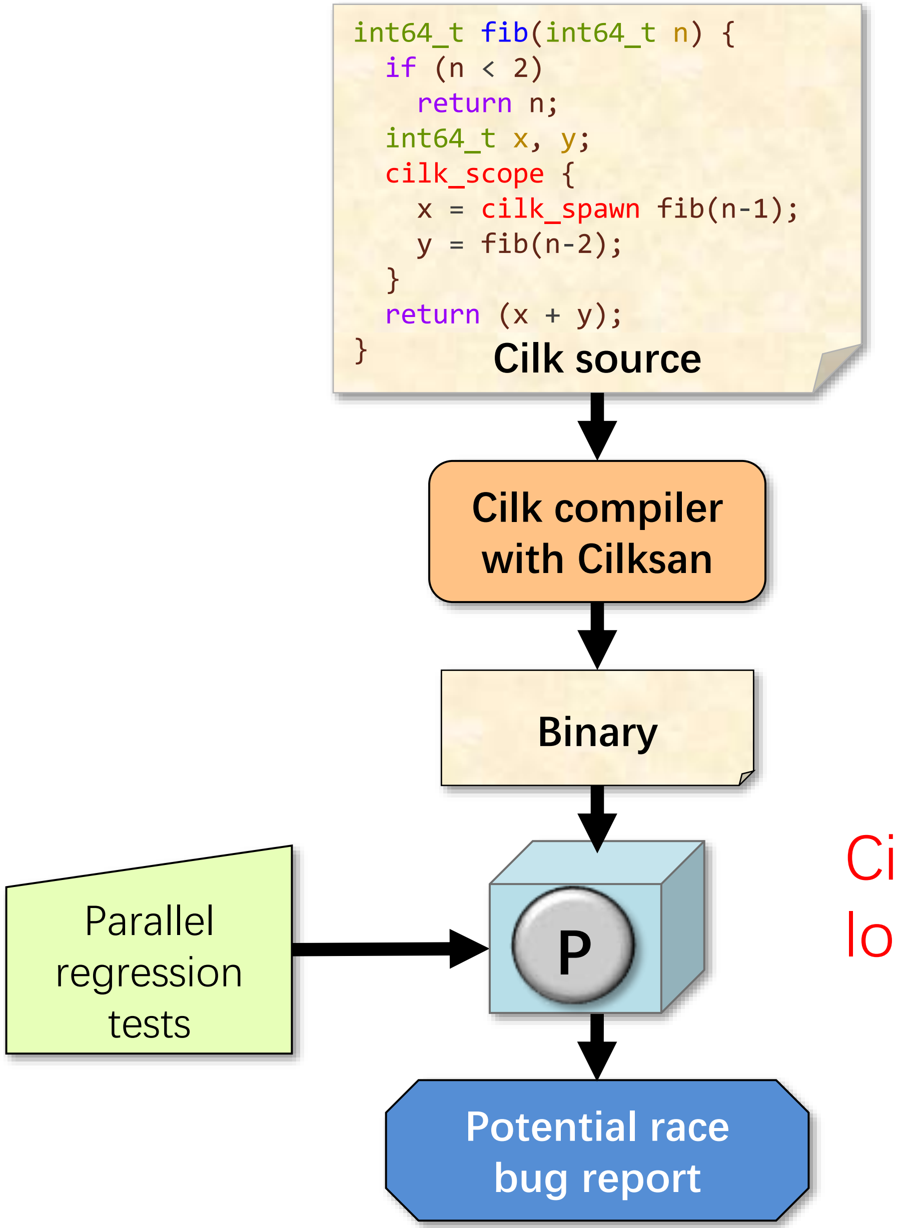
Alternative Serial Testing

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



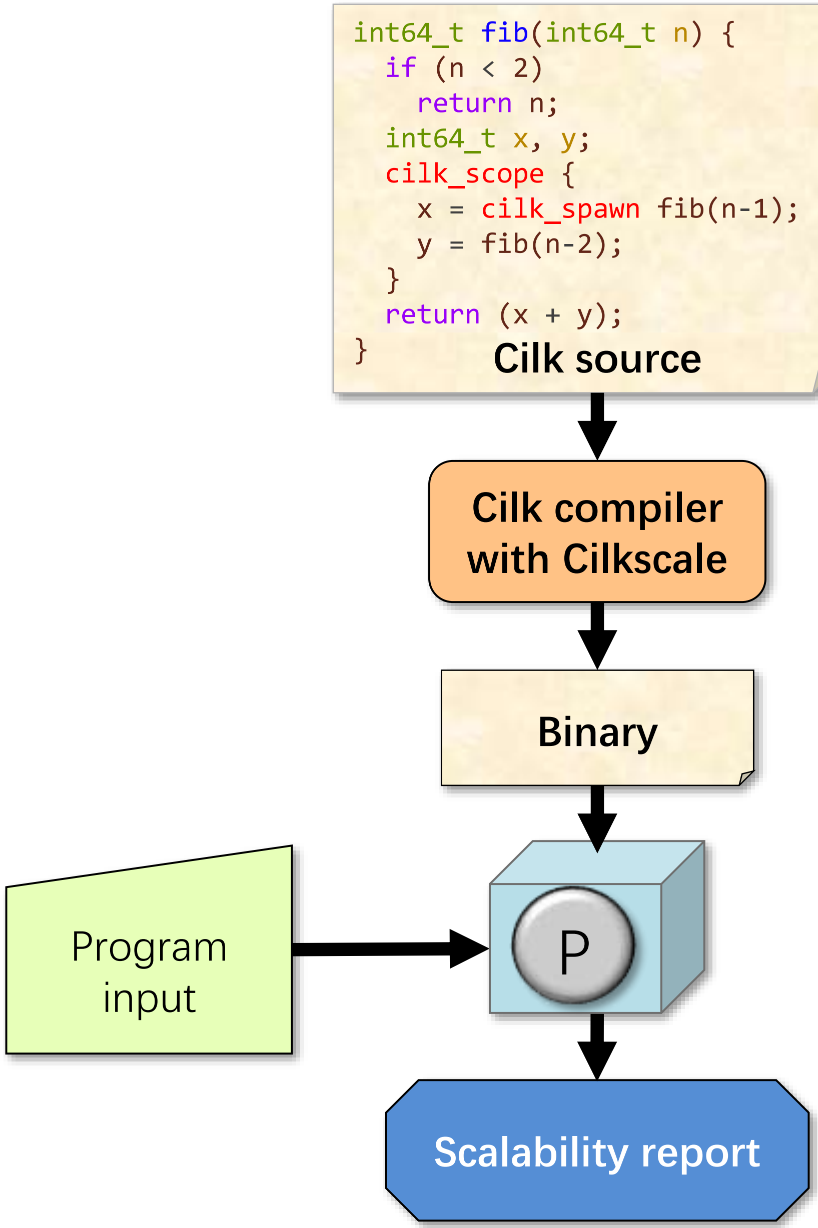
The parallel program executing on one core should behave exactly the same as the execution of the serial projection.

Parallel Testing



Cilksan finds and localizes race bugs

Scalability Analysis



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

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 simulation & rendering using Cilk.