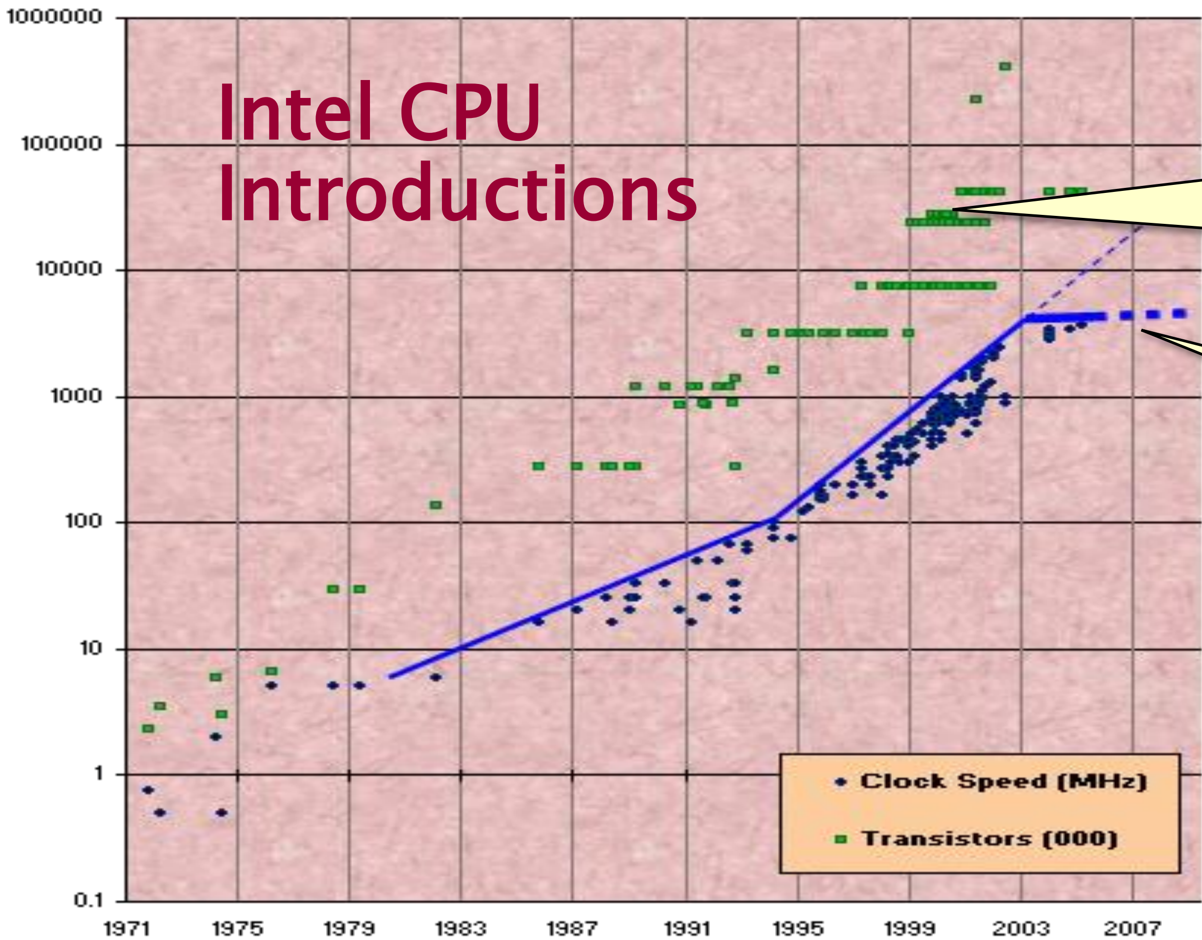
6.172 Performance Engineering of Software Systems

LECTURE 12
Multicore Programming

Charles E. Leiserson

October 21, 2010

Moore's Law

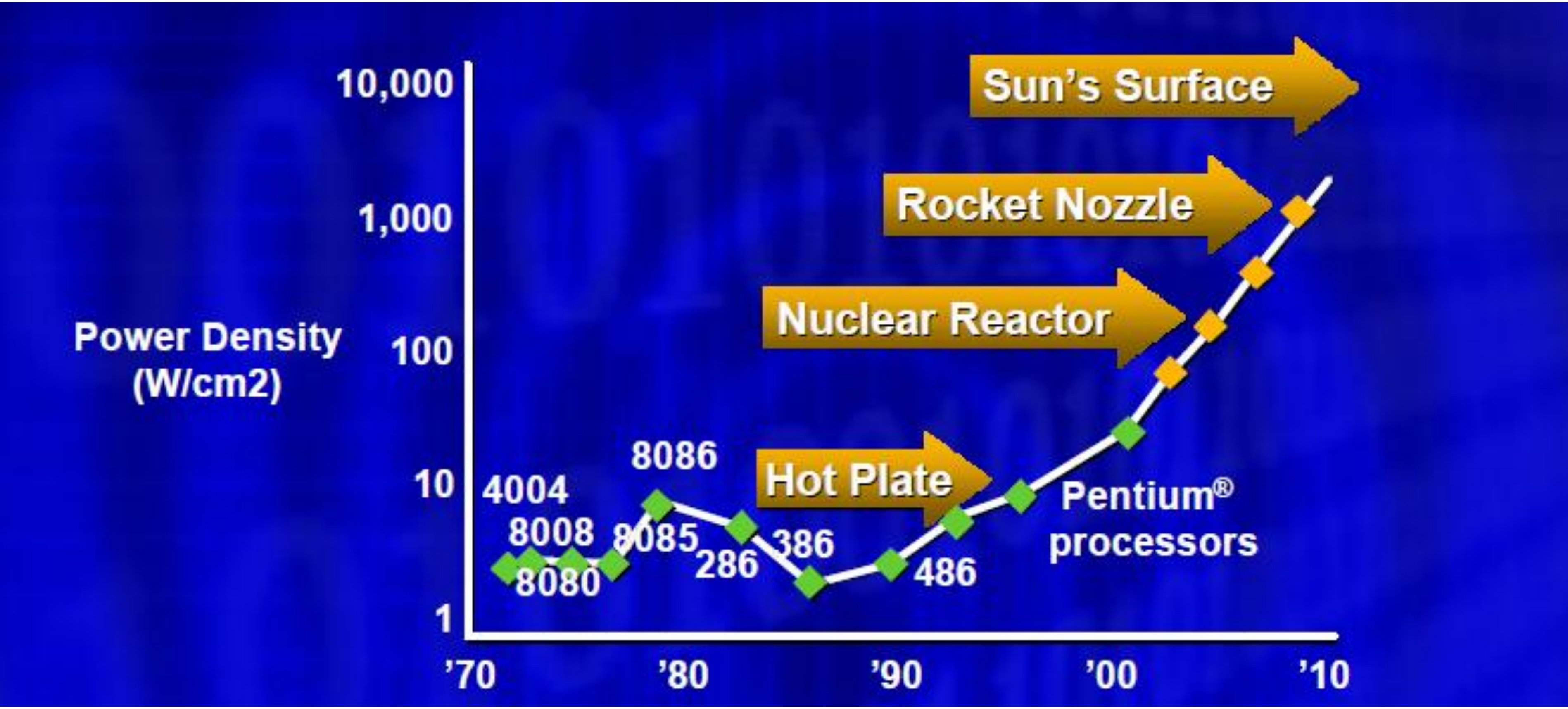


Transistor count is still rising, …

but clock speed is bounded at ~5GHz.

Source: Herb Sutter, "The free lunch is over: a fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, 30(3), March 2005.

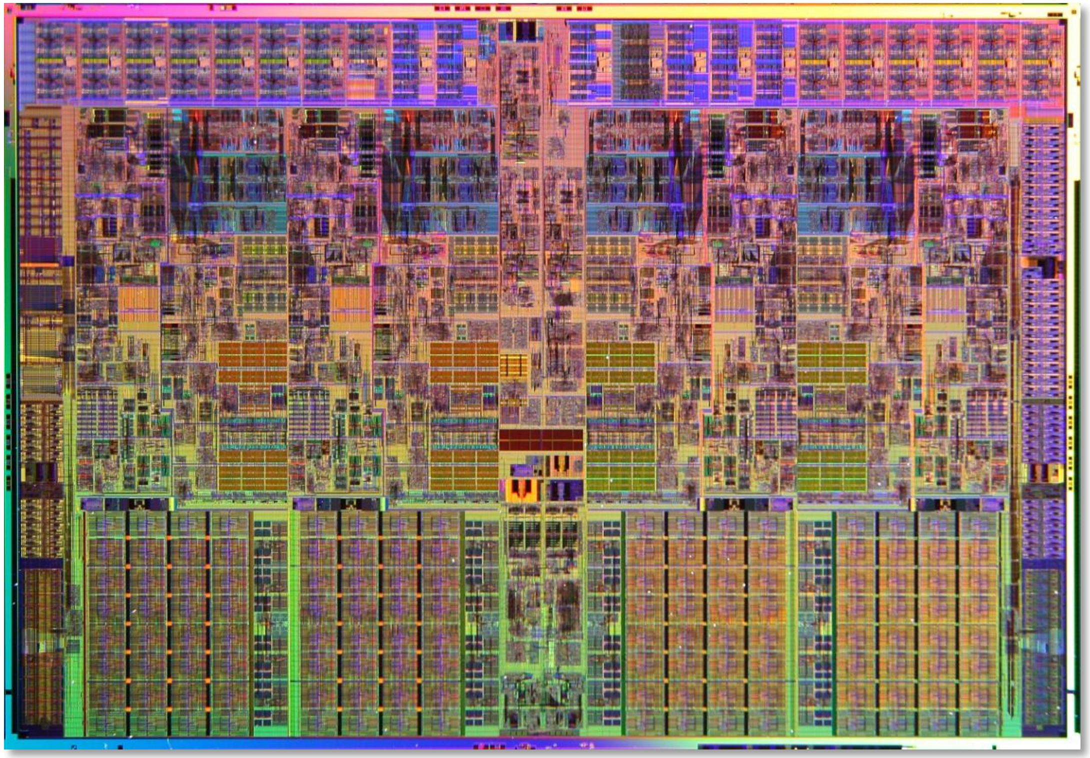
Power Density



Reprinted with permission of Intel Corporation.

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

Vendor Solution

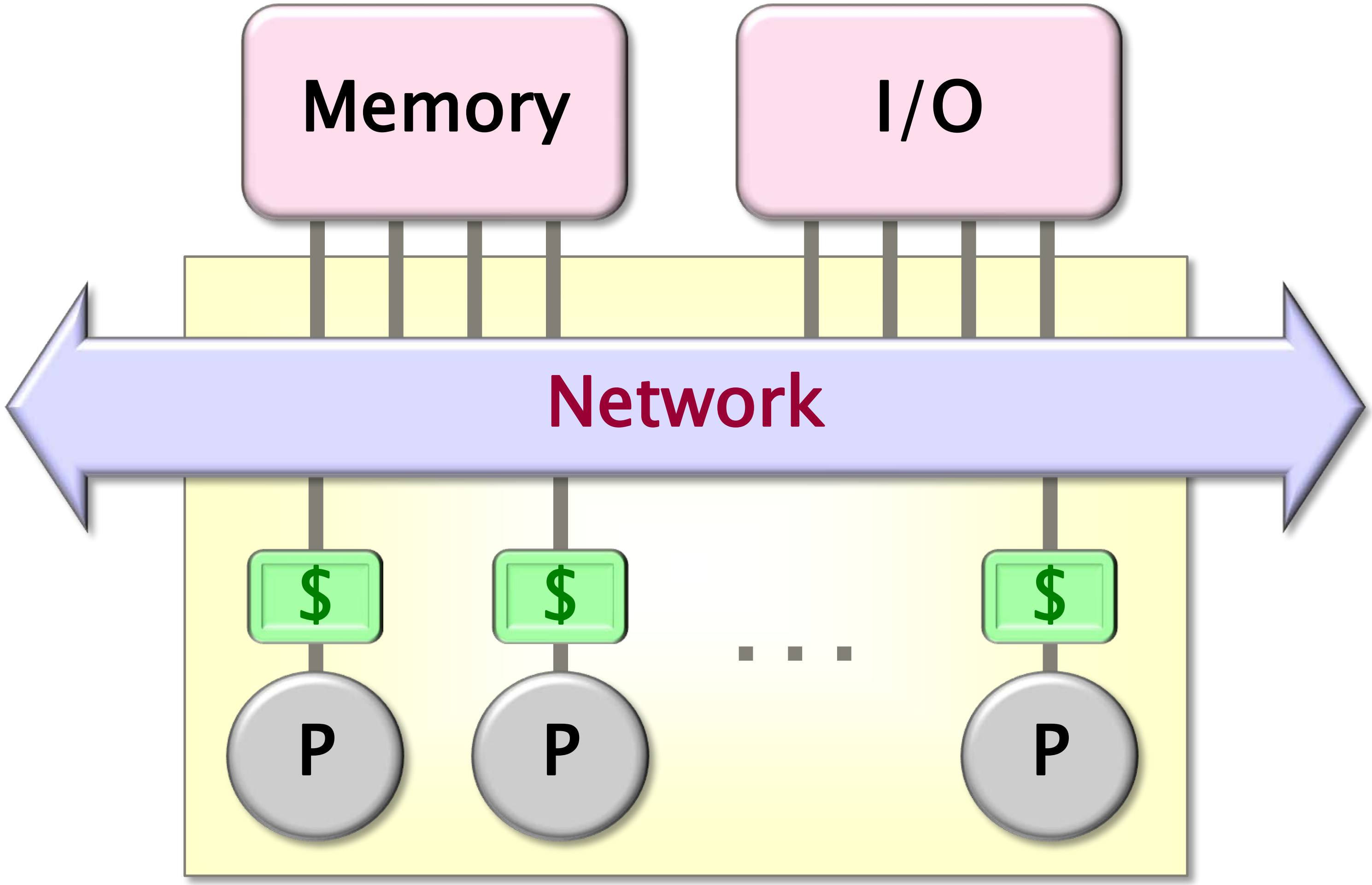


Intel Core i7 processor

Reprinted with permission of Intel Corporation.

- To scale performance, put many processing cores on the microprocessor chip.
- Each generation of Moore's Law potentially doubles the number of cores.

Abstract Multicore Architecture



Chip Multiprocessor (CMP)

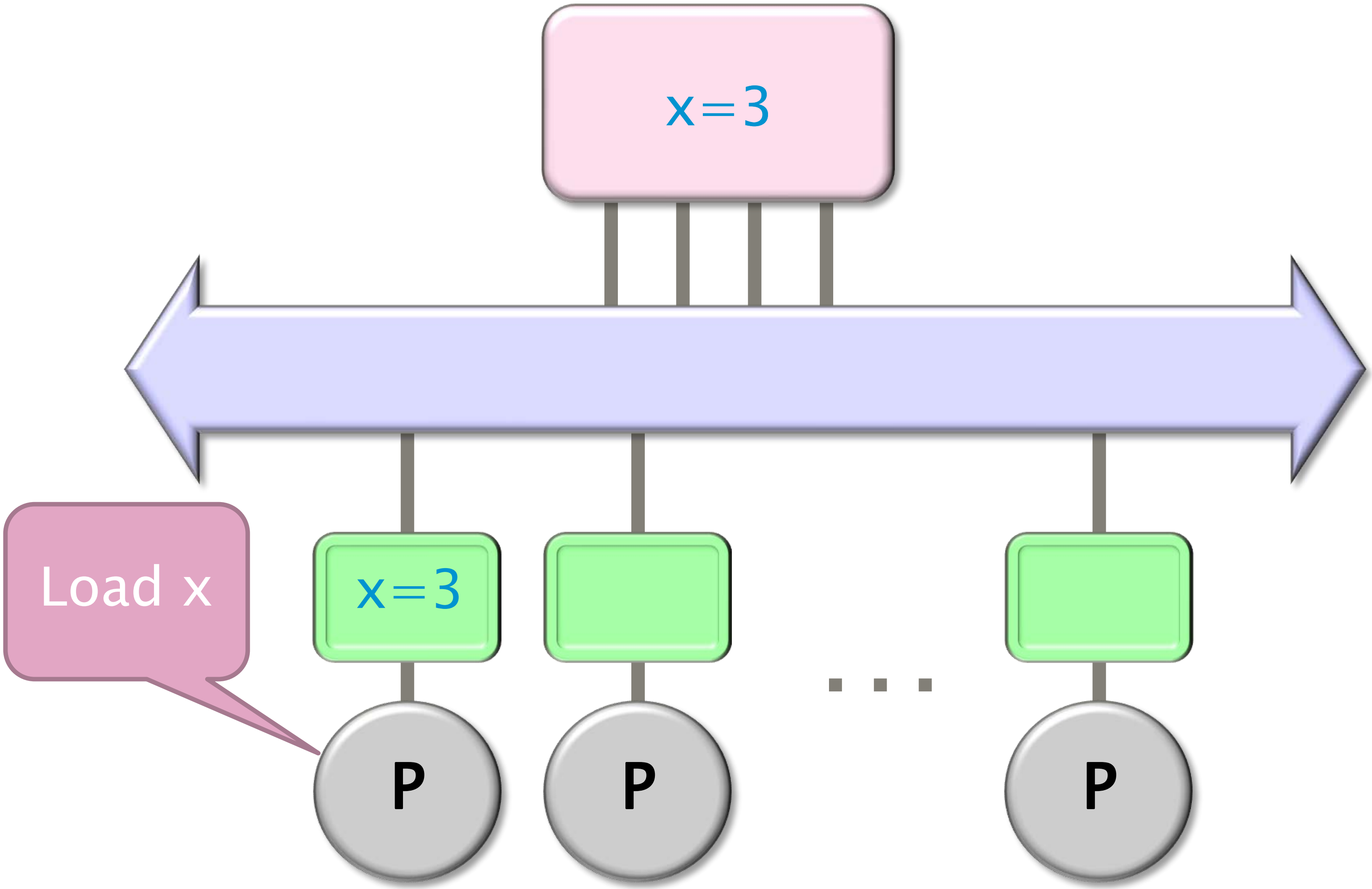
OUTLINE

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

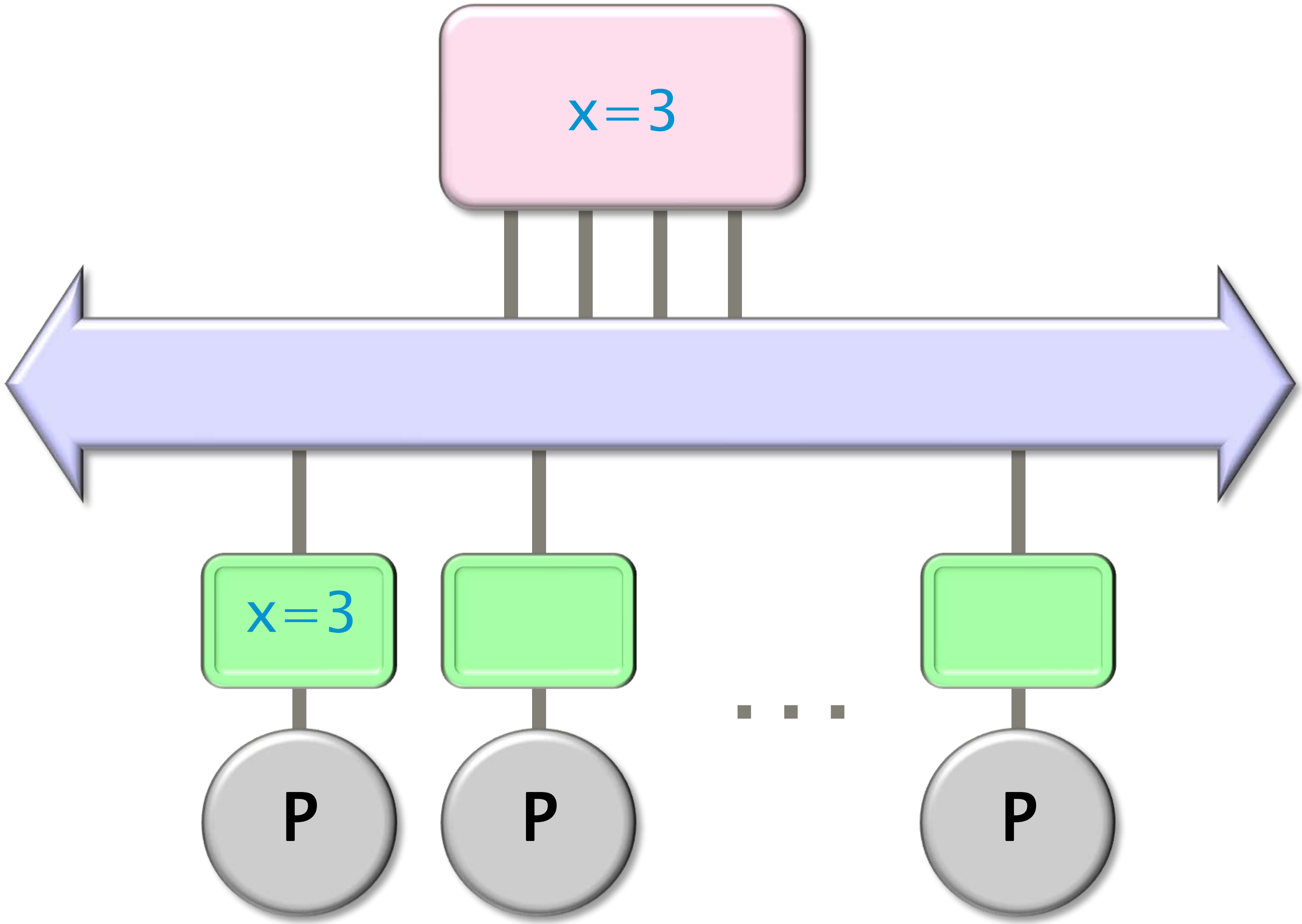
OUTLINE

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

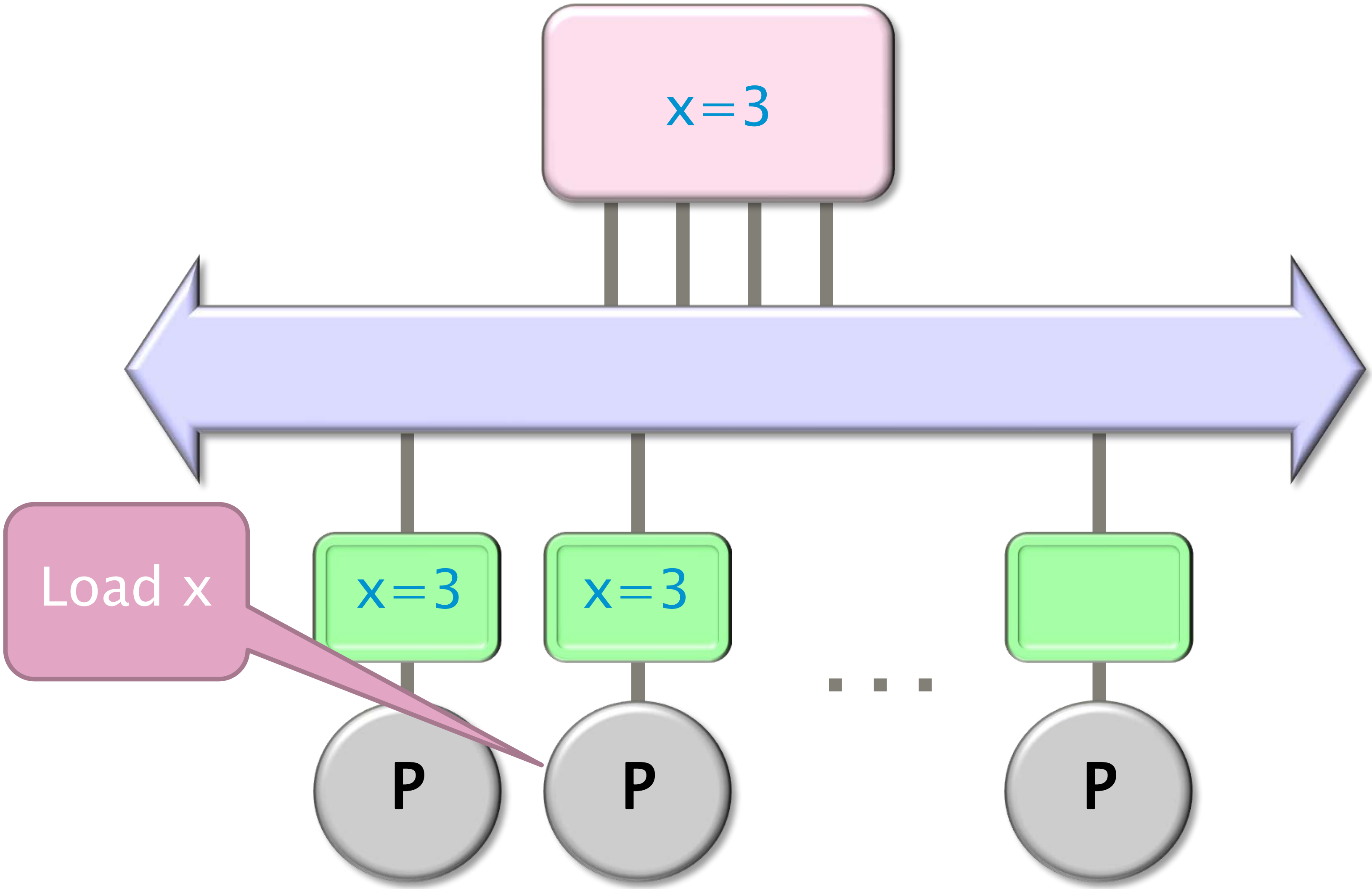
Cache Coherence



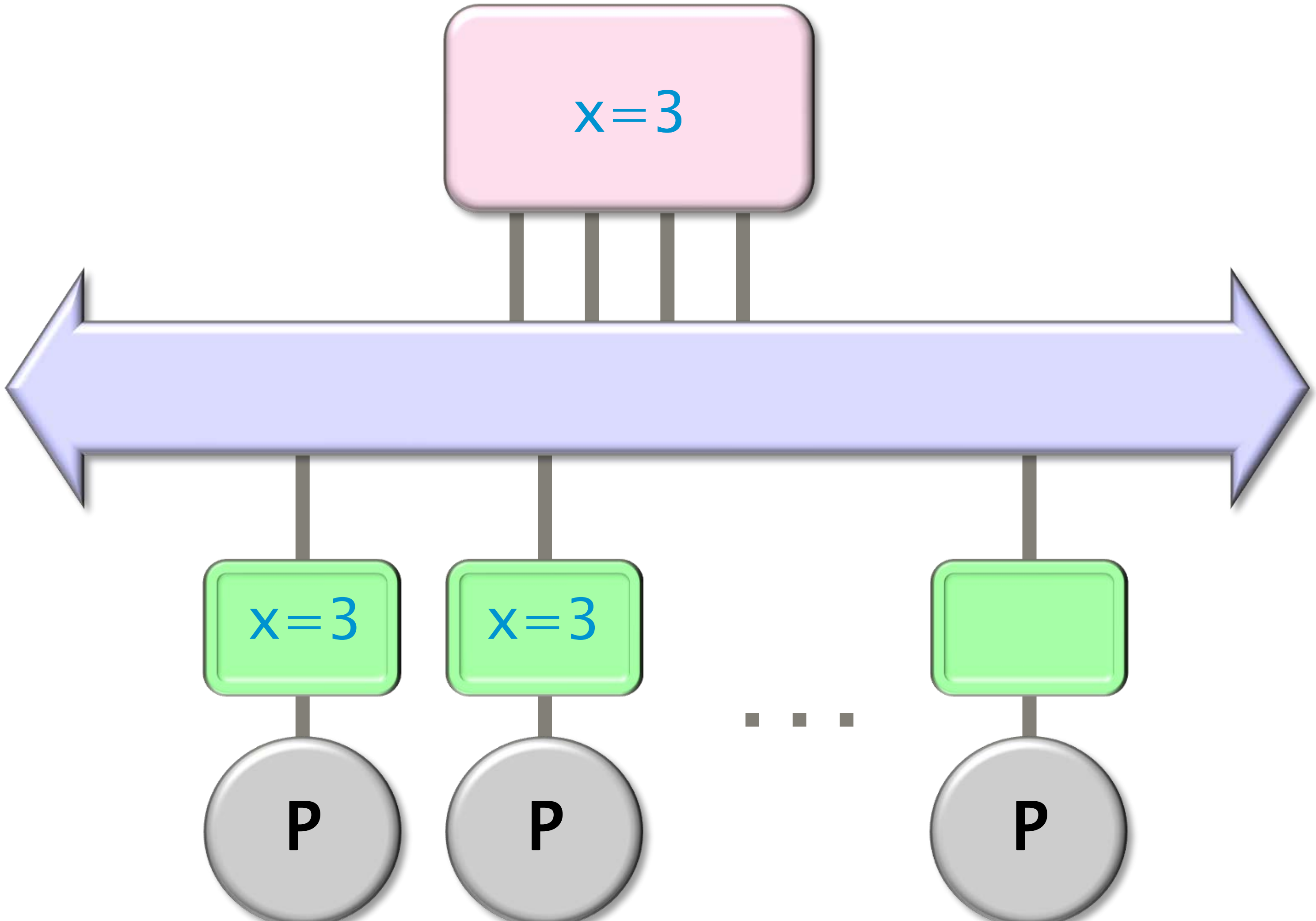
Cache Coherence



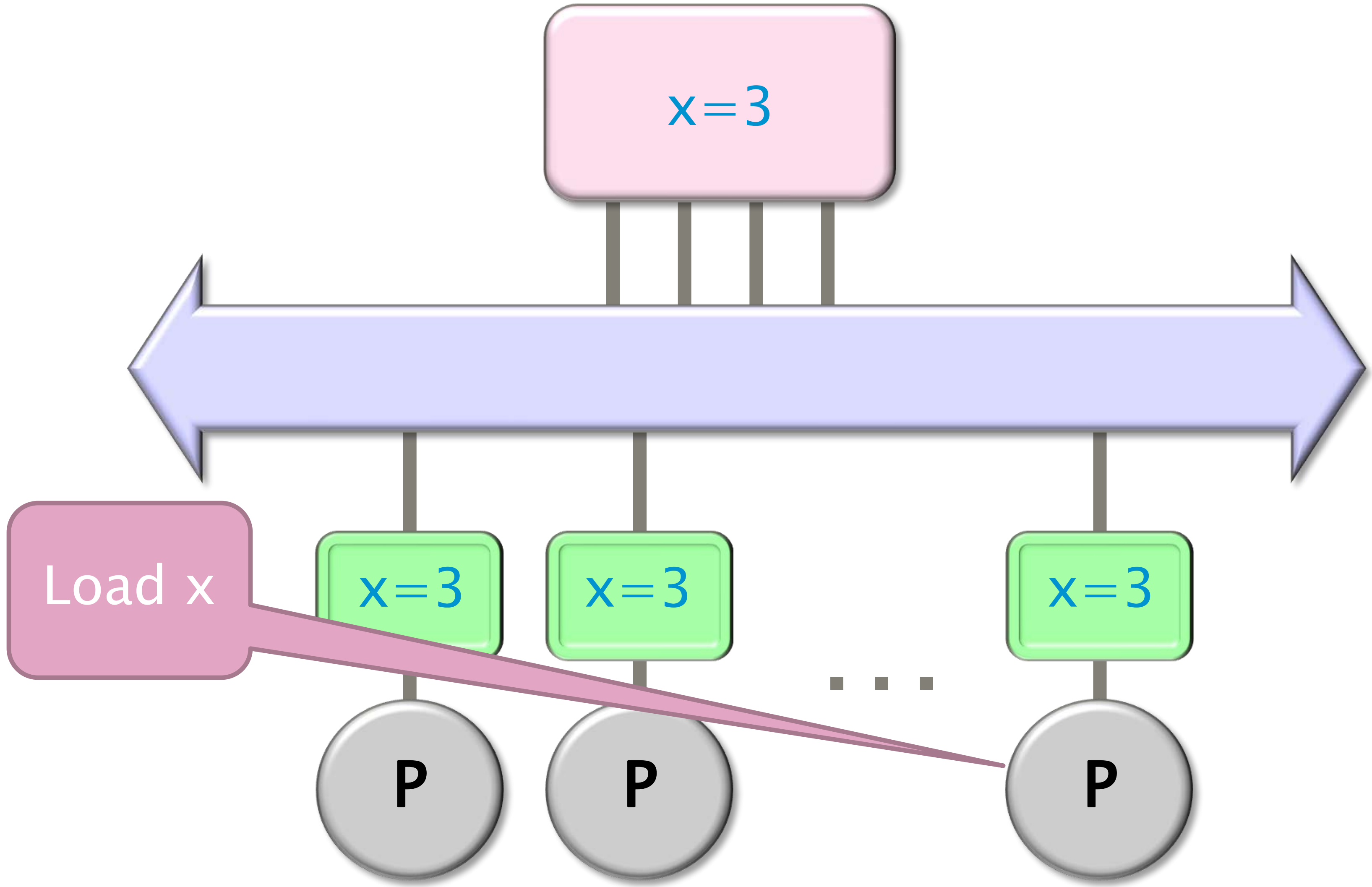
Cache Coherence



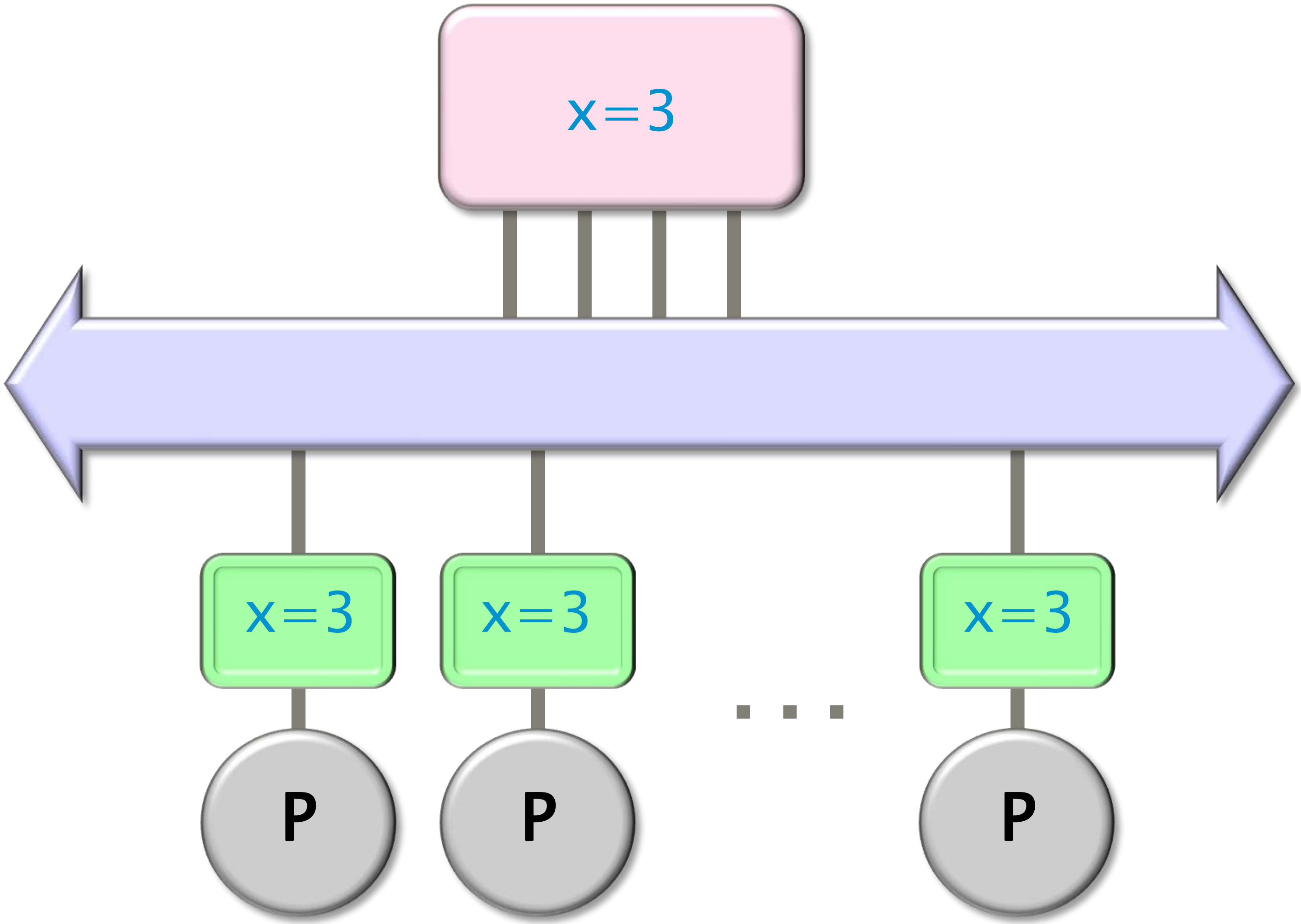
Cache Coherence



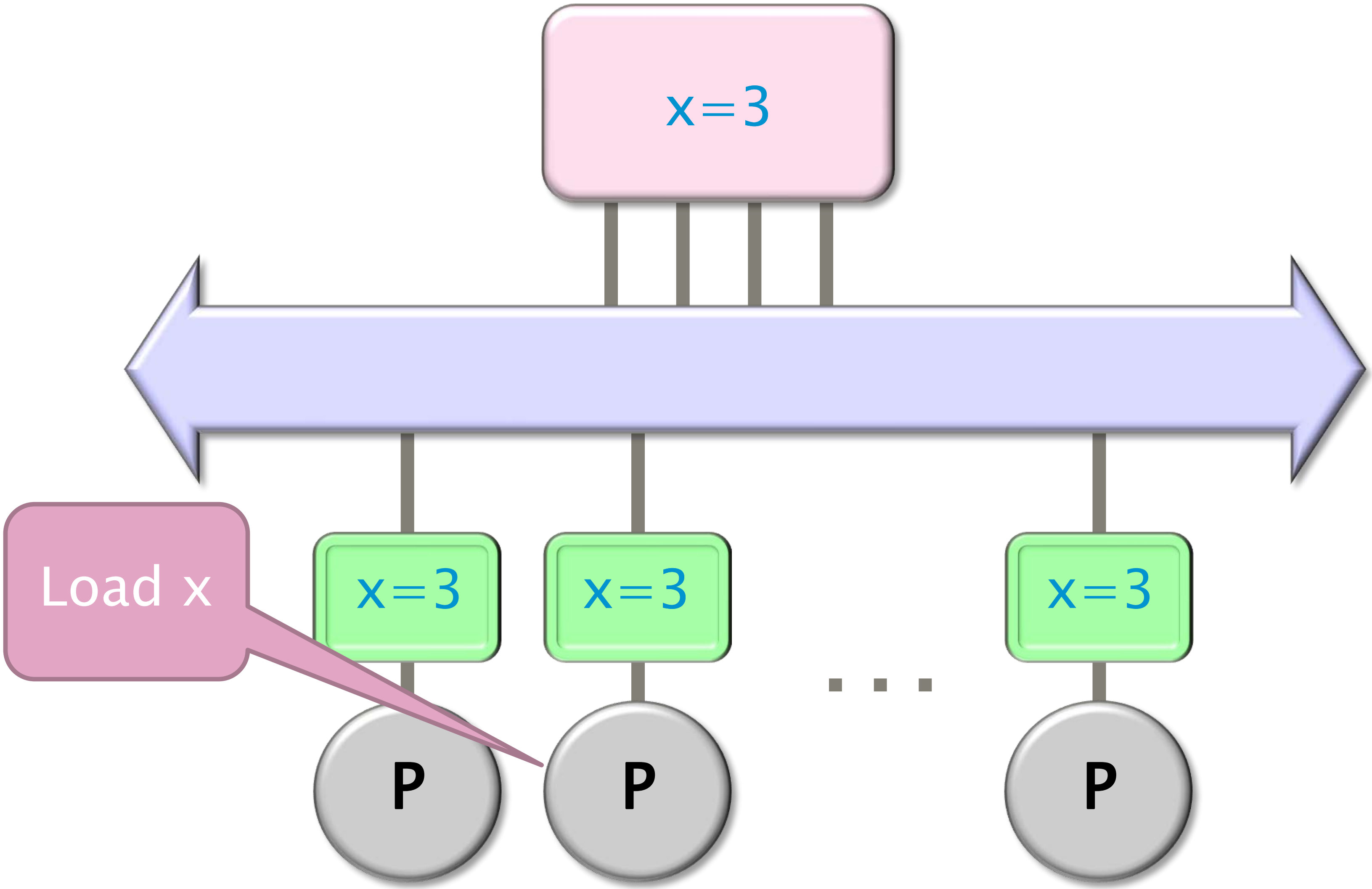
Cache Coherence



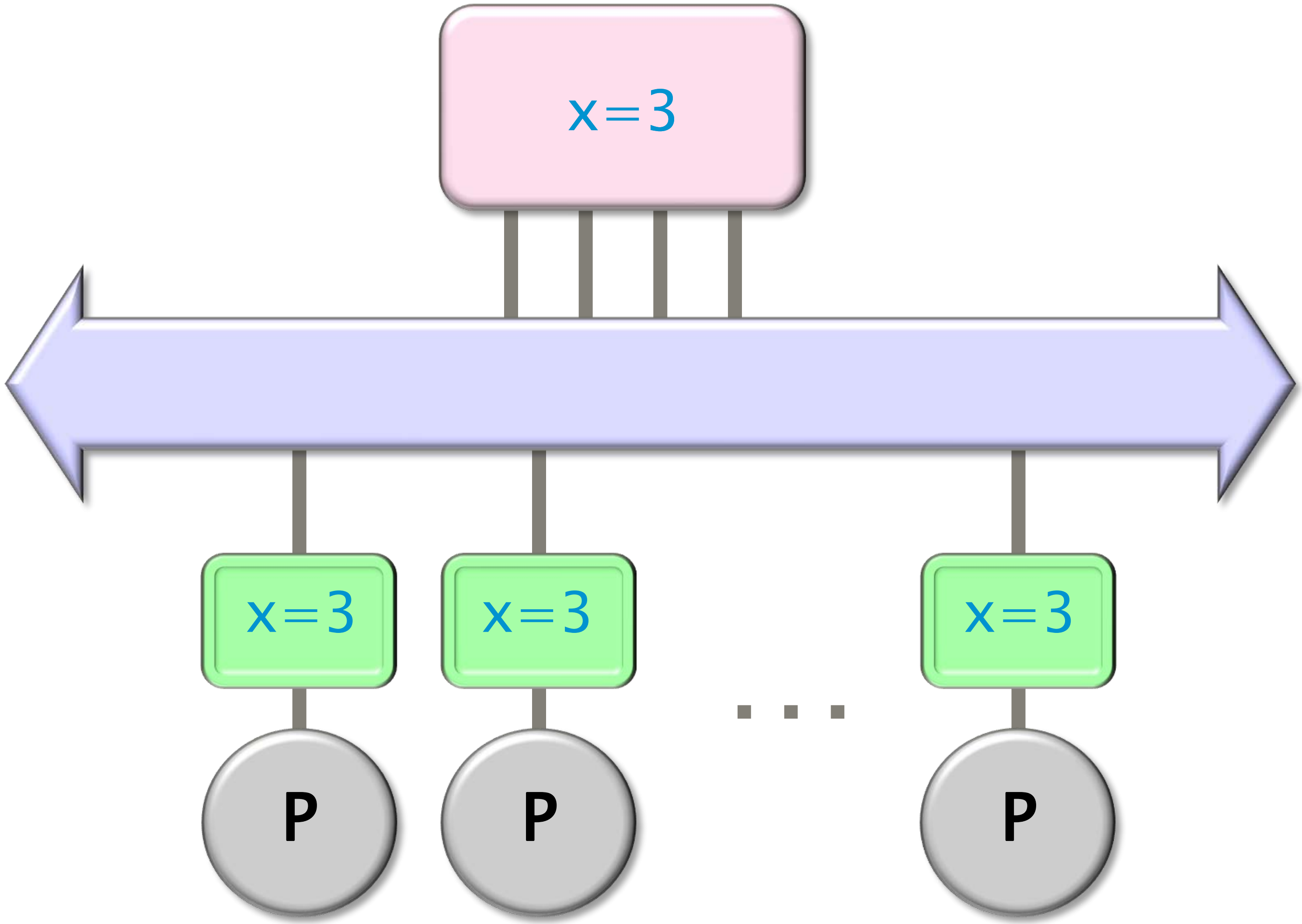
Cache Coherence



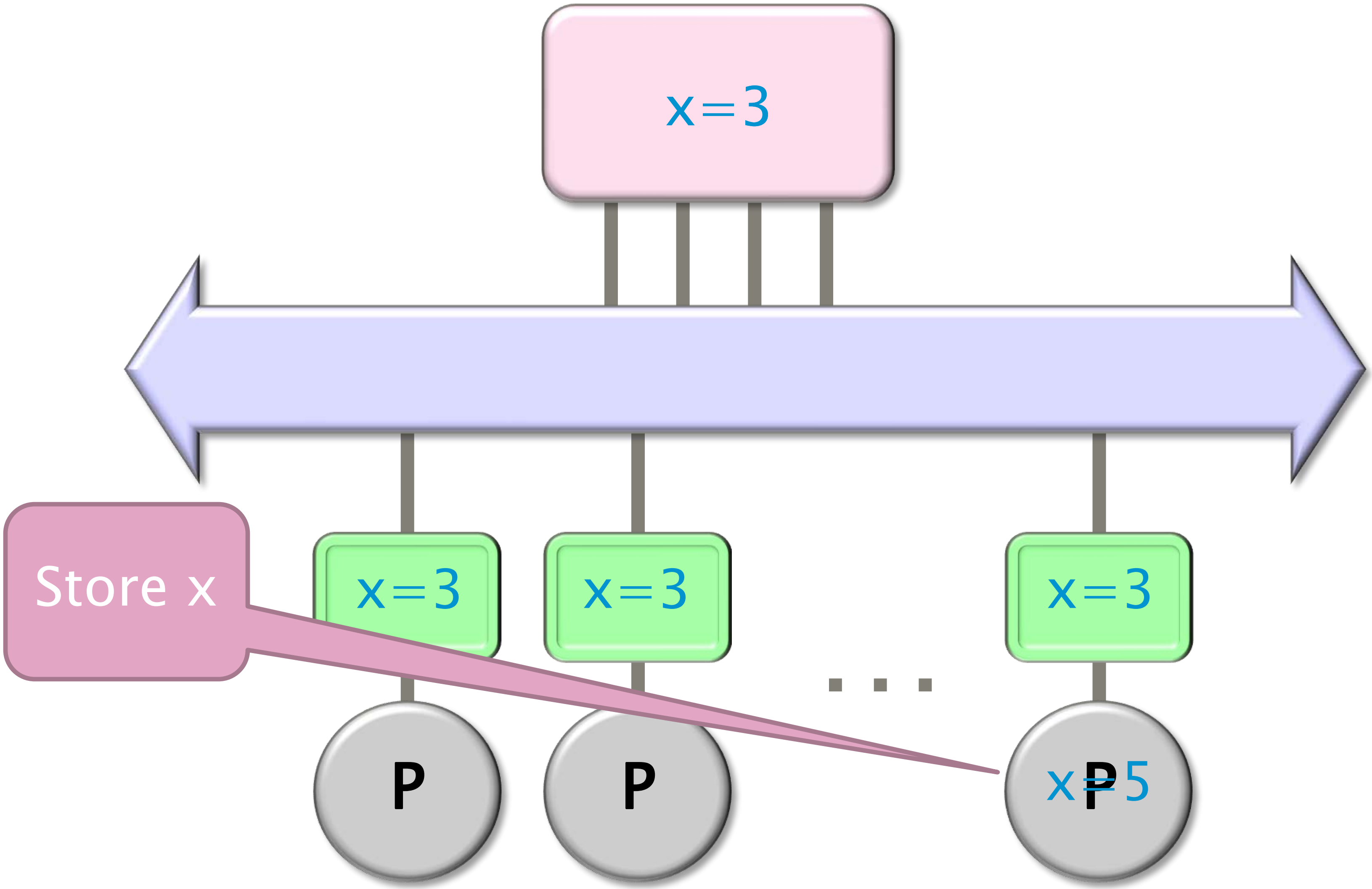
Cache Coherence



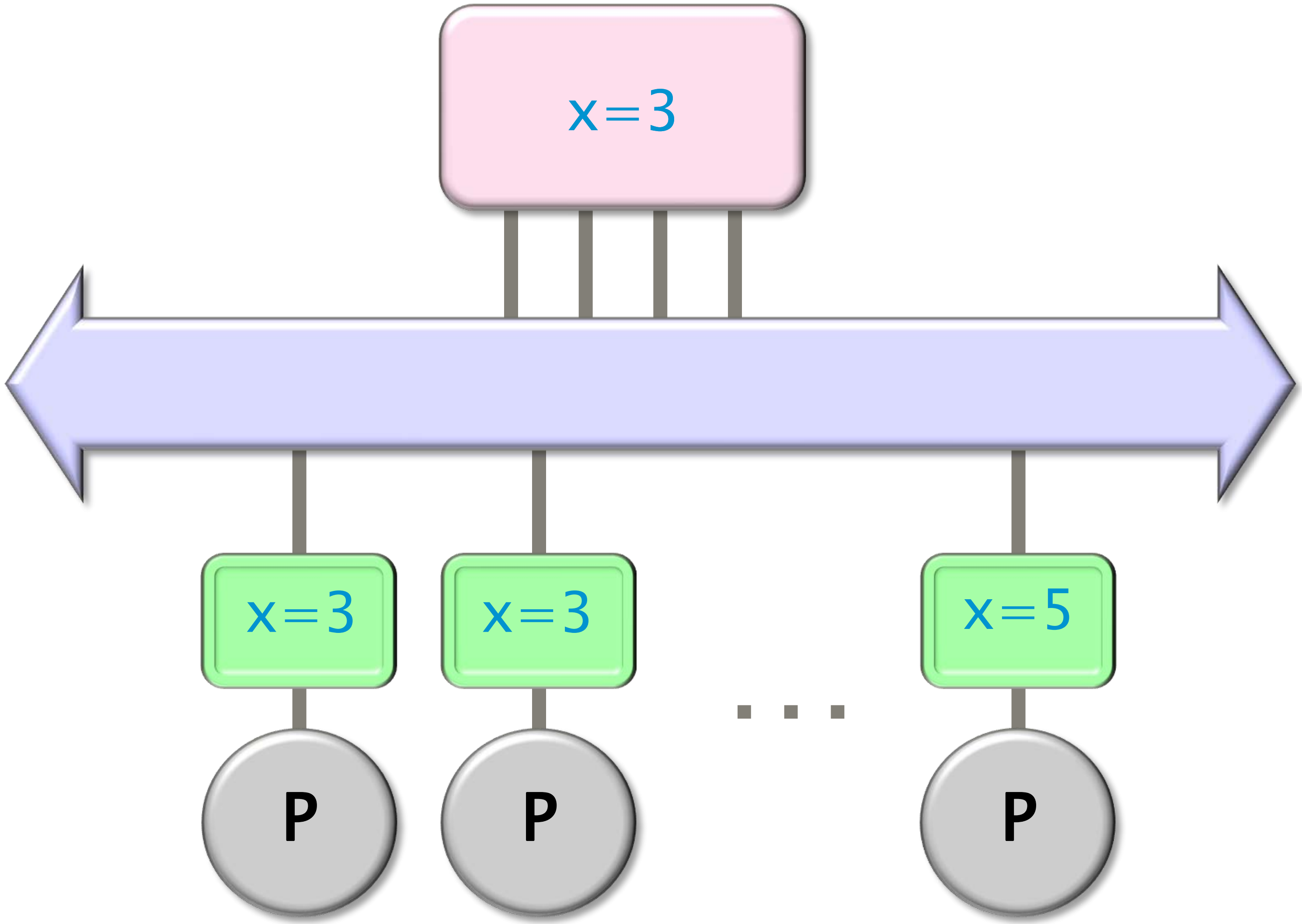
Cache Coherence



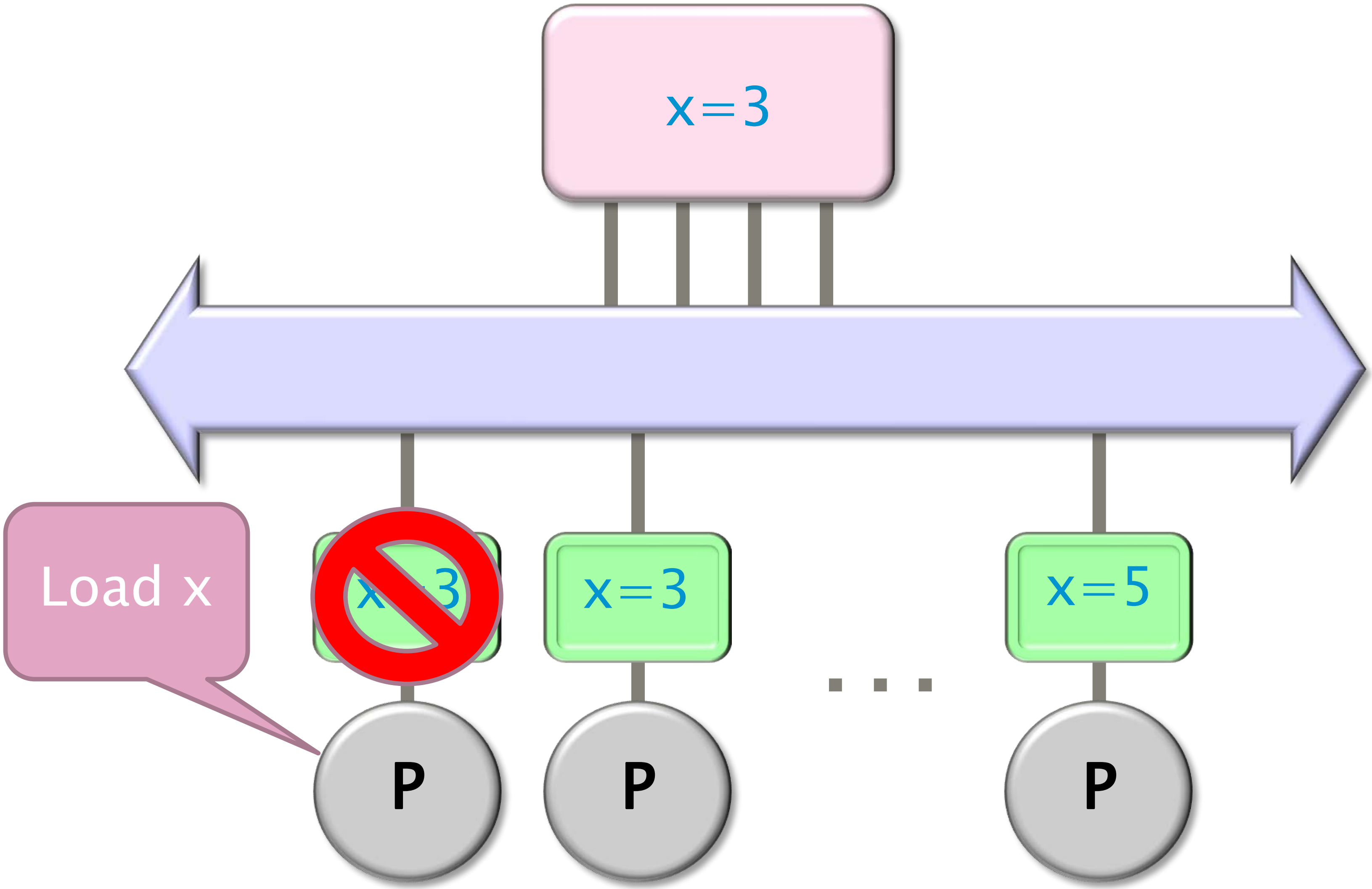
Cache Coherence



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.

OUTLINE

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

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:

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

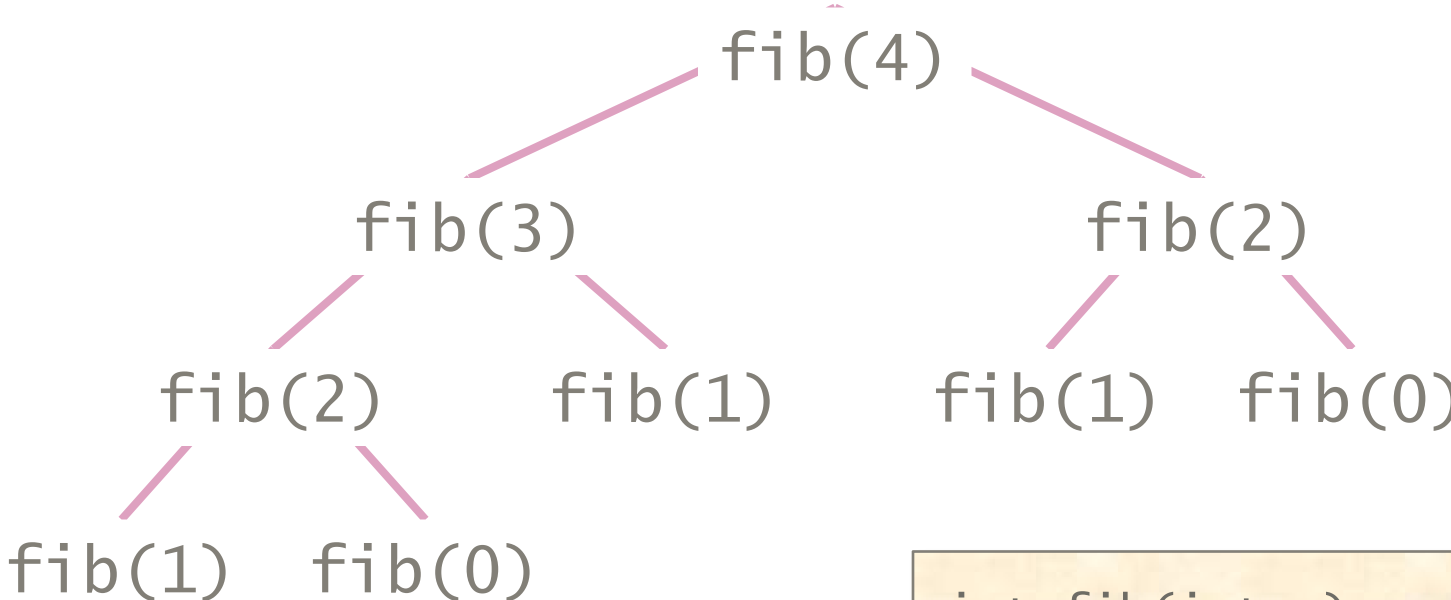
Fibonacci Program

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

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

int main(int argc, char *argv[])
{
  int n = atoi(argv[1]);
  int result = fib(n);
  printf("Fibonacci of %d is %d.\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.

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

OUTLINE

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

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.

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 Implementation

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

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

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

void *thread_func ( void *ptr )
{
  int 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;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\n", n, result);
  return 0;
}
```

Pthread Implementation

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

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

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

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

Original code.

```c
int main(int argc, char *argv[])
{
  pthread_t thread;
  thread_args args;
  int status;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\n", n, result);
  return 0;
}
```

Pthread Implementation

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

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

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

void *thread_func ( void *ptr )
{
  int 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;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\n", n, result);
  return 0;
}
```

Pthread Implementation

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

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

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

void *thread_func ( void *ptr )
{
  int 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;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\n", n, result);
  return 0;
}
```

Pthread Implementation

```
#include <stdio.h>
#include <stdli
#include <pthre

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

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

void *thread_func ( void *ptr )
{
  int 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;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\n", n, result);
  return 0;
}
```

Pthread Implementation

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

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

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

void *thread_func ( void *ptr )
{
  int 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;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\n", n, result);
  return 0;
}
```

Marshal input argument to thread.

Pthread Implementation

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

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

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

void *thread_func ( void *ptr )
{
  int 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;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\n", n, result);
  return 0;
}
```

Create thread to execute fib(n-1).

Pthread Implementation

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

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

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

void *thread_func (
{
  int i = ((thread_
  ((thread_args *) 
  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;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\n", n, result);
  return 0;
}
```

Pthread Implementation

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

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

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

void *thread_func
{
  int i = ((threac
  ((thread_args *)
  return NULL;
}
```

Block until the auxiliary thread finishes.

```c
int main(int argc, char *argv[])
{
  pthread_t thread;
  thread_args args;
  int status;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\n", n, result);
  return 0;
}
```

Pthread Implementation

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

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

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

void *thread_func ( void *ptr )
{
  int 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;
  int result;
  int thread_result;
  if (argc < 2) return 1;
  int n = atoi(argv[1]);
  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 %d is %d.\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 $\Rightarrow$ coarse-grained concurrency. (Thread pools can help.)

Scalability Fibonacci code gets 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.

OUTLINE

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

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 **work-stealing**.
- Focus on **performance**.

Reinders, James. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

Fibonacci in TBB

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
             n(n_), sum(sum_) {}

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long 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;
    }
};
```

Fibonacci in TBB

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
             n(n_), sum(sum_) {}

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long 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.

Fibonacci in TBB

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
             n(n_), sum(sum_) {}

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long 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.

Fibonacci in TBB

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
             n(n_), sum(sum_) {}

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long 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.

Fibonacci in TBB

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
             n(n_), sum(sum_) {}

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long 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.

Fibonacci in TBB

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
             n(n_), sum(sum_) {}

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long 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).

Fibonacci in TBB

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
             n(n_), sum(sum_) {}

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long 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.

Fibonacci in TBB

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
             n(n_), sum(sum_) {}

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long 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 a and wait for both a and b to finish.

Fibonacci in TBB

```
class FibTask: public task {
public:
    const long n;
    long* const sum;
    FibTask( long n_, long* sum_ ) :
             n(n_), sum(sum_) {}

    task* execute() {
        if( n < 2 ) {
            *sum = n;
        } else {
            long 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;
    }
};
```

Add the results together to produce the final output.

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++
- Race Conditions

OpenMP

- Specification produced by an industry consortium.
- Several compilers available, both open-source and proprietary, including gcc and Visual Studio.
- Linguistic extensions to C/C++ or Fortran in the form of compiler pragmas.
- Runs on top of native threads.
- Supports loop parallelism and, more recently in Version 3.0, task parallelism.

Fibonacci in OpenMP 3.0

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

Fibonacci in OpenMP 3.0

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

Compiler directive.

Fibonacci in OpenMP 3.0

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

The following statement is an independent task.

Fibonacci in OpenMP 3.0

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

Sharing of memory is managed explicitly.

Fibonacci in OpenMP 3.0

```
int fib(int n)
{
   if (n < 2) return n;
   int x, y;
#pragma omp task shared(x)
   x = fib(n - 1);
#pragma omp task shared(y)
   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 provides a variety of synchronization constructs, such as barriers, atomic updates, and mutual-exclusion locks.

OUTLINE

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

Cilk++

- Small set of linguistic extensions to C++ to support fork-join parallelism.
- Developed 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.
- Includes the Cilkscreen race detector and Cilkview scalability analyzer.

Nested Parallelism in Cilk++

```
int fib(int n)
{
   if (n < 2) return n
   int 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.

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

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

```
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++ serialization

The C++ *serialization* 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 serialization:

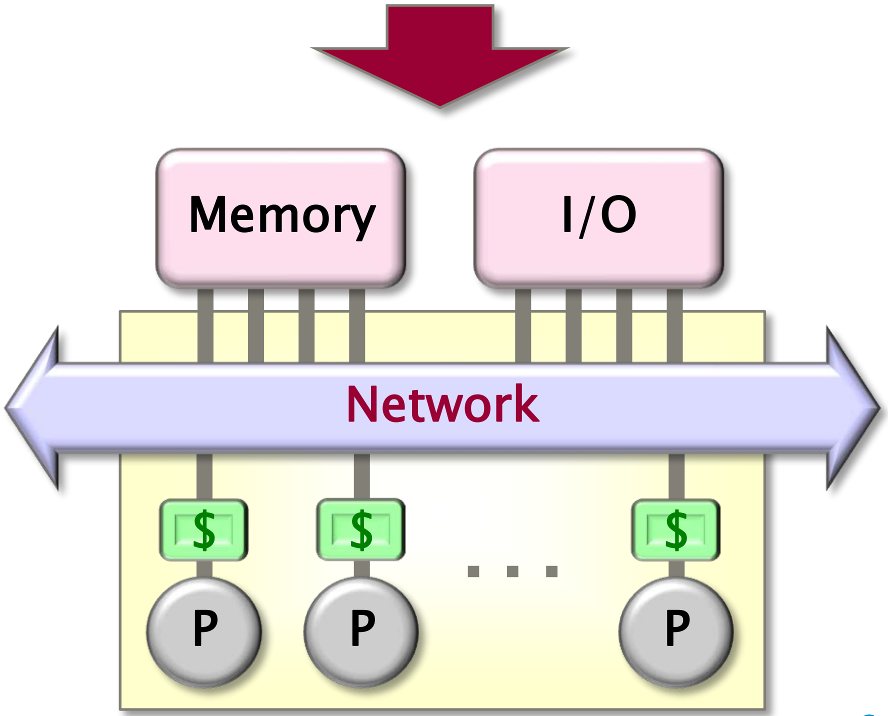
```
#define cilk_for for
#define cilk_spawn
#define cilk_sync
```

Or, specify a switch to the Cilk++ compiler.

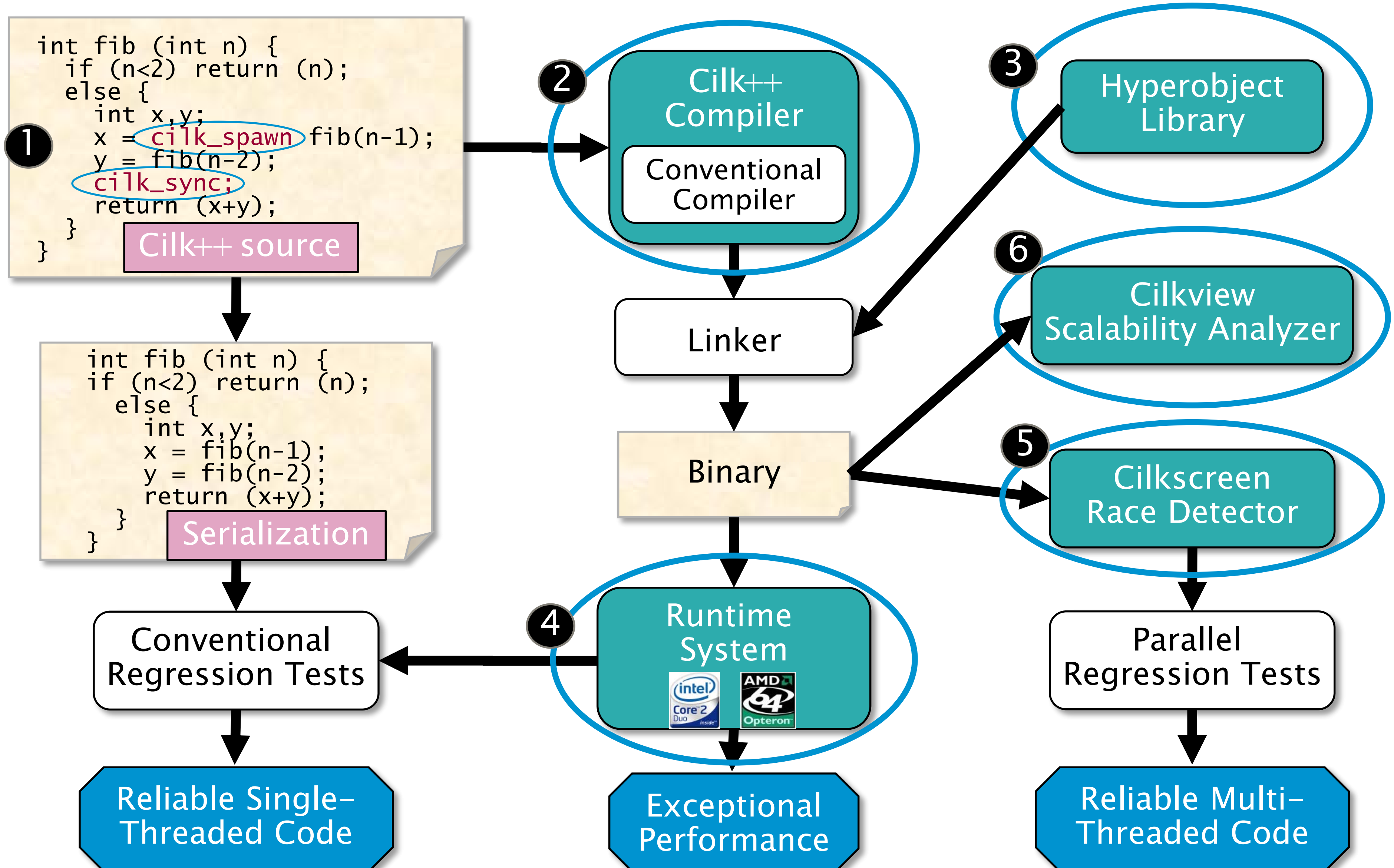
Scheduling

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

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



Cilk++ Platform



OUTLINE

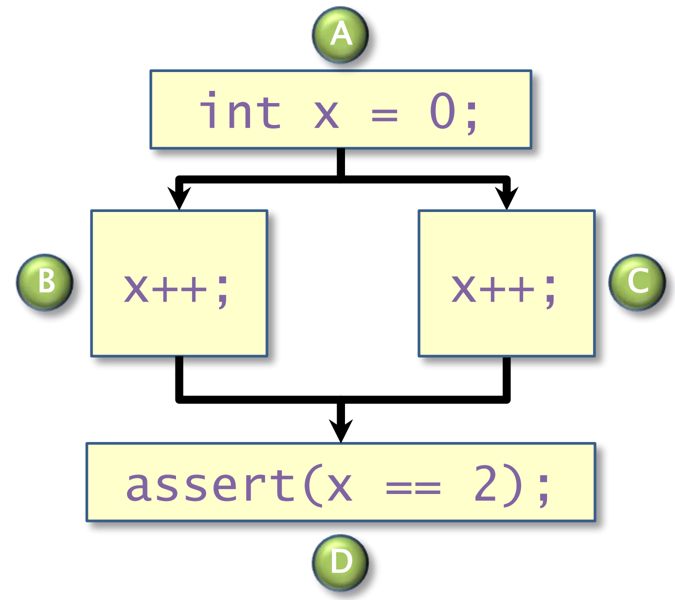
- Shared-Memory Hardware
- Concurrency Platforms
  - Pthreads (and WinAPI Threads)
  - Threading Building Blocks
  - OpenMP
  - Cilk++
- Race Conditions

Determinacy Races

Definition. A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

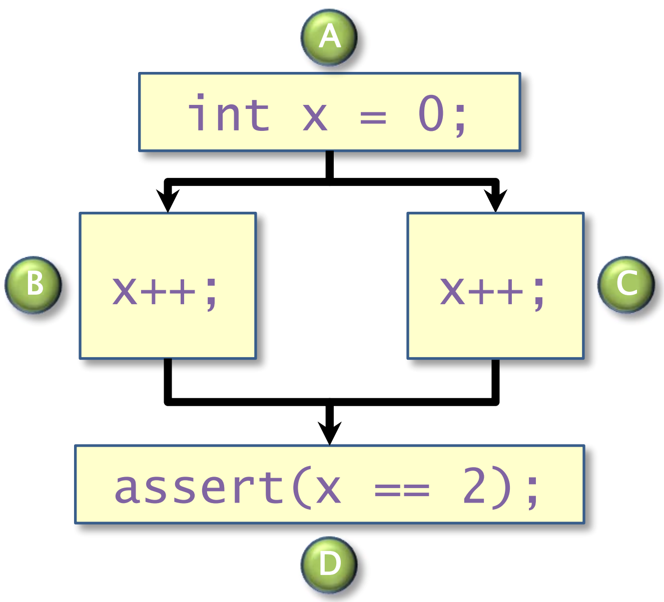
Example

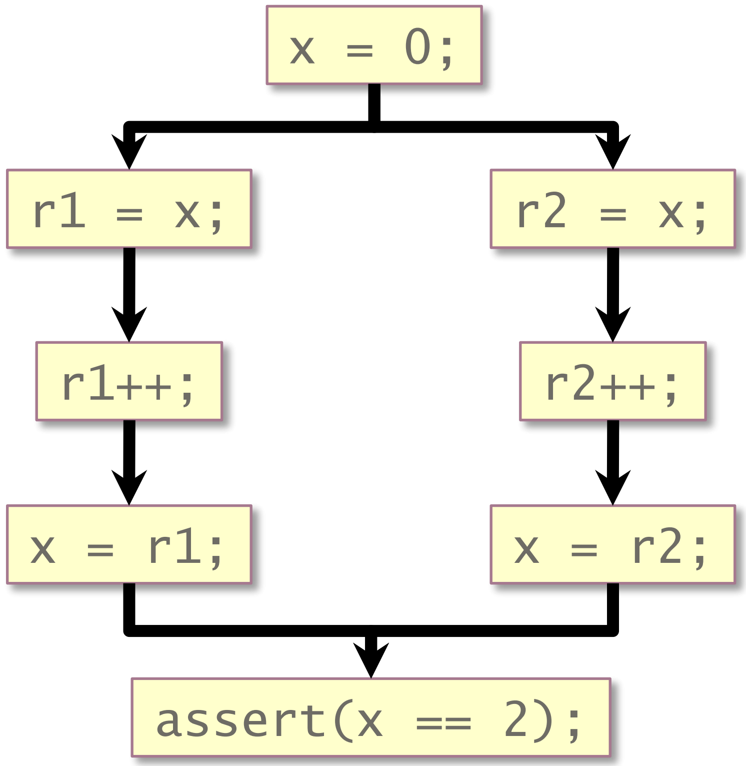
```
int x = 0;
cilk_for (int i=0, i<2, ++i) {
    x++;
}
assert(x == 2);
```



Dependency Graph

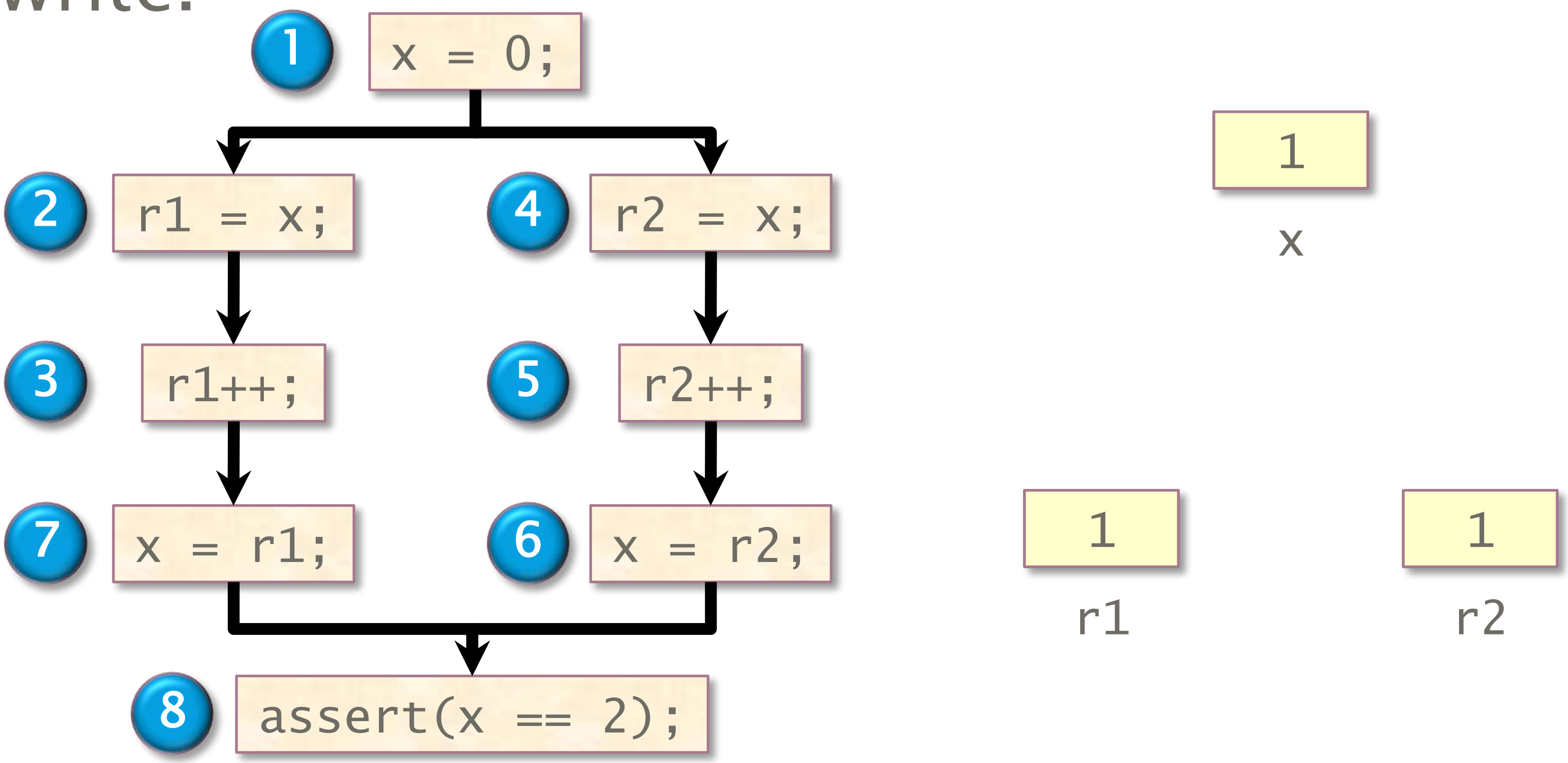
A Closer Look





Race Bugs

Definition. A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.



Types of Races

Suppose that instruction A and instruction B both access a location x, and suppose that A∥B (A is parallel to B).

A	B	Race Type
read	read	none
read	write	read race
write	read	read race
write	write	write race

Two sections of code are *independent* if they have no determinacy races between them.

Avoiding Races

- Iterations of a `cilk_for` should be independent.
- Between a `cilk_spawn` and the corresponding `cilk_sync`, the code of the spawned child should be independent of the code of the parent, including code executed by additional spawned or called children.
  Note: The arguments to a spawned function are evaluated in the parent before the spawn occurs.
- Machine word size matters. Watch out for races in packed data structures:

```
struct{
    char a;
    char b;
} x;
```

Ex. Updating `x.a` and `x.b` in parallel may cause a race! Nasty, because it may depend on the compiler optimization level. (Safe on Intel.)

Cilkscreen Race Detector

- If an ostensibly deterministic Cilk++ program run on a given input could possibly behave any differently than its serialization, Cilkscreen **guarantees** to report and localize the offending race.
- Employs a **regression-test** methodology, where the programmer provides test inputs.
- **Identifies** filenames, lines, and variables involved in races, including stack traces.
- Runs off the binary executable using **dynamic instrumentation**.
- Runs about **20** times slower than real-time.

6.172 Performance Engineering of Software Systems
Fall 2010