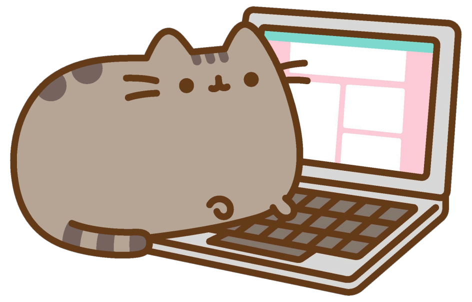
CSE 6230:
HPC Tools and Applications

+

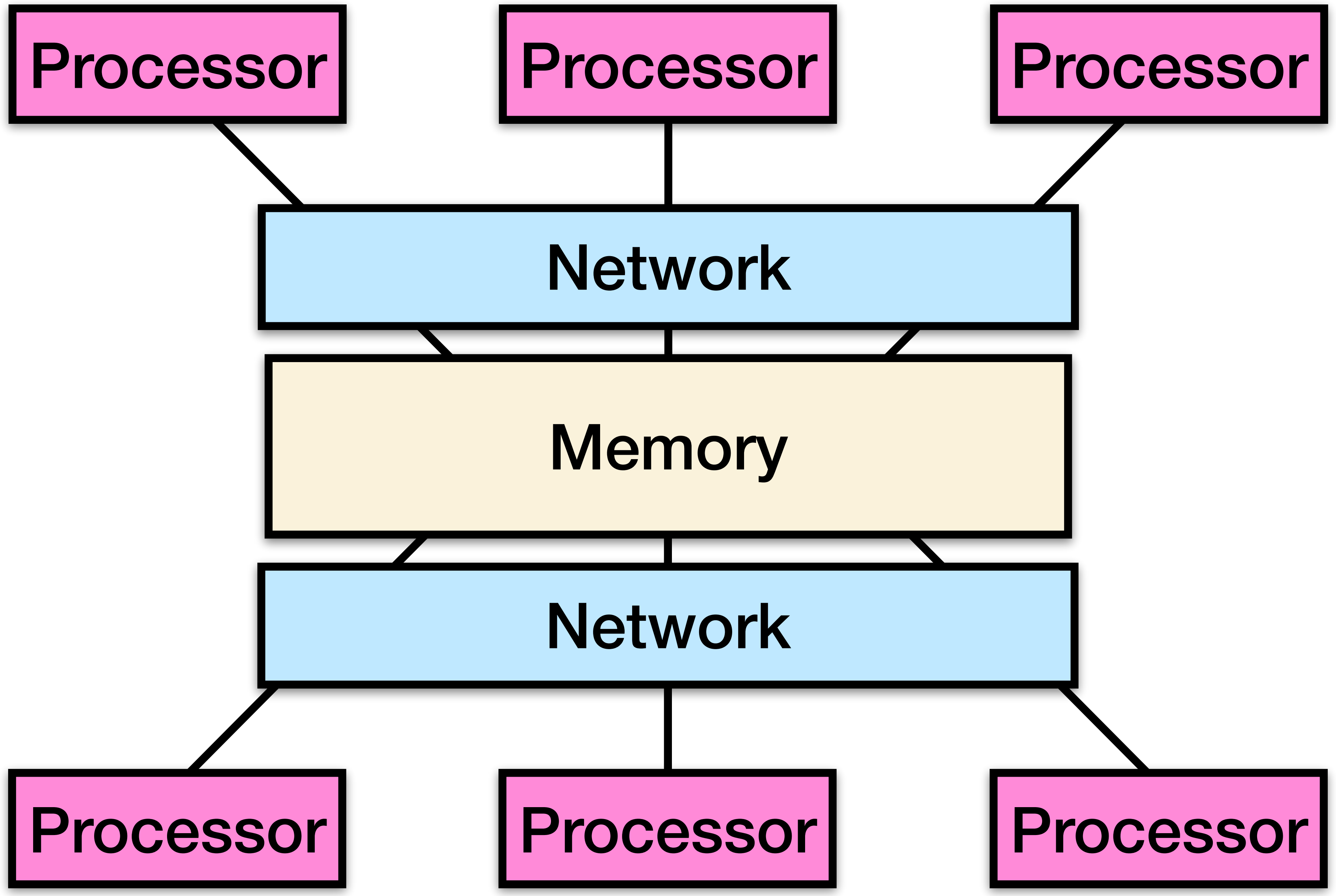


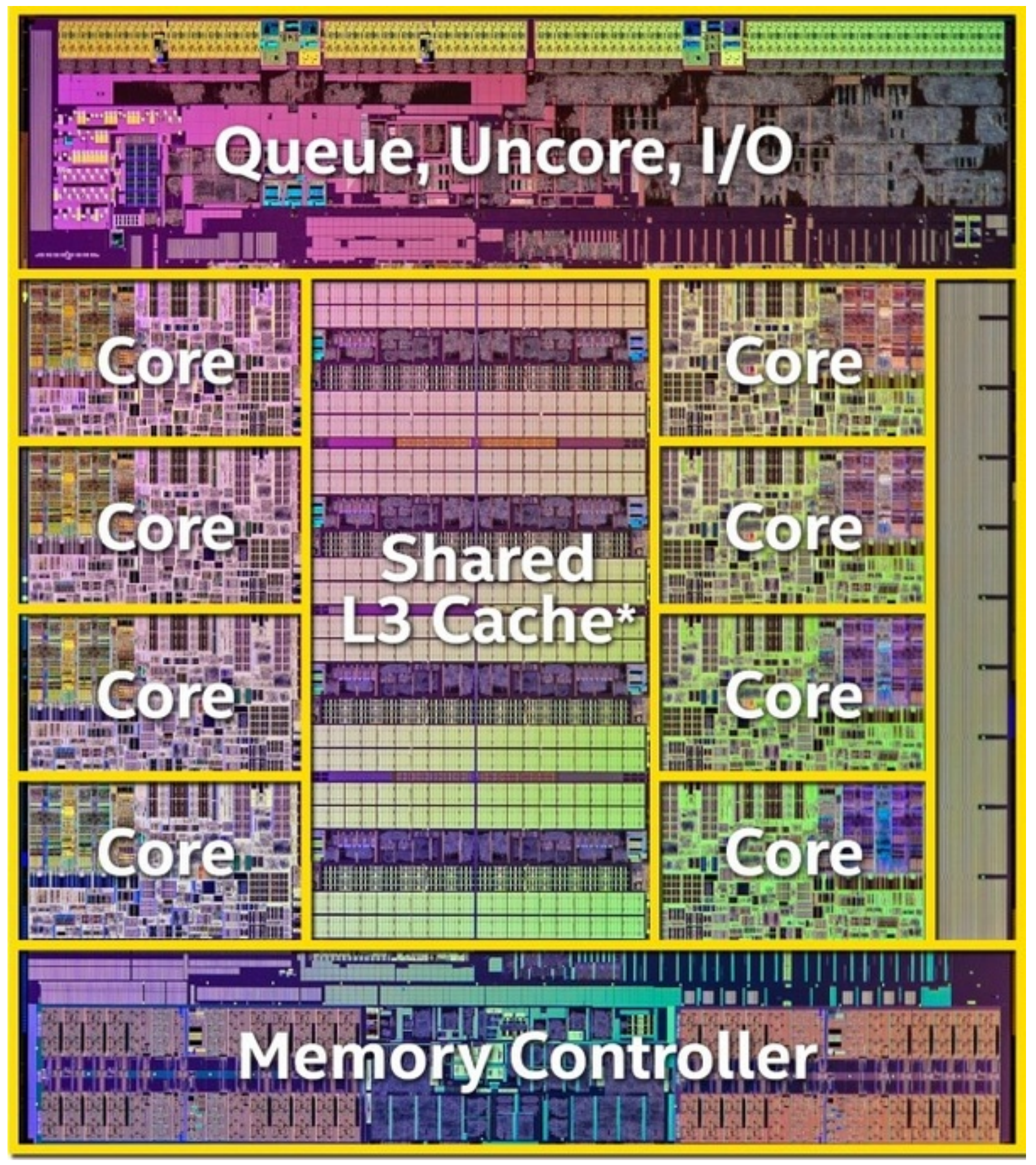
Lecture 6: Shared-Memory Programming

Helen Xu
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

Recall: Shared-memory multiprocessors



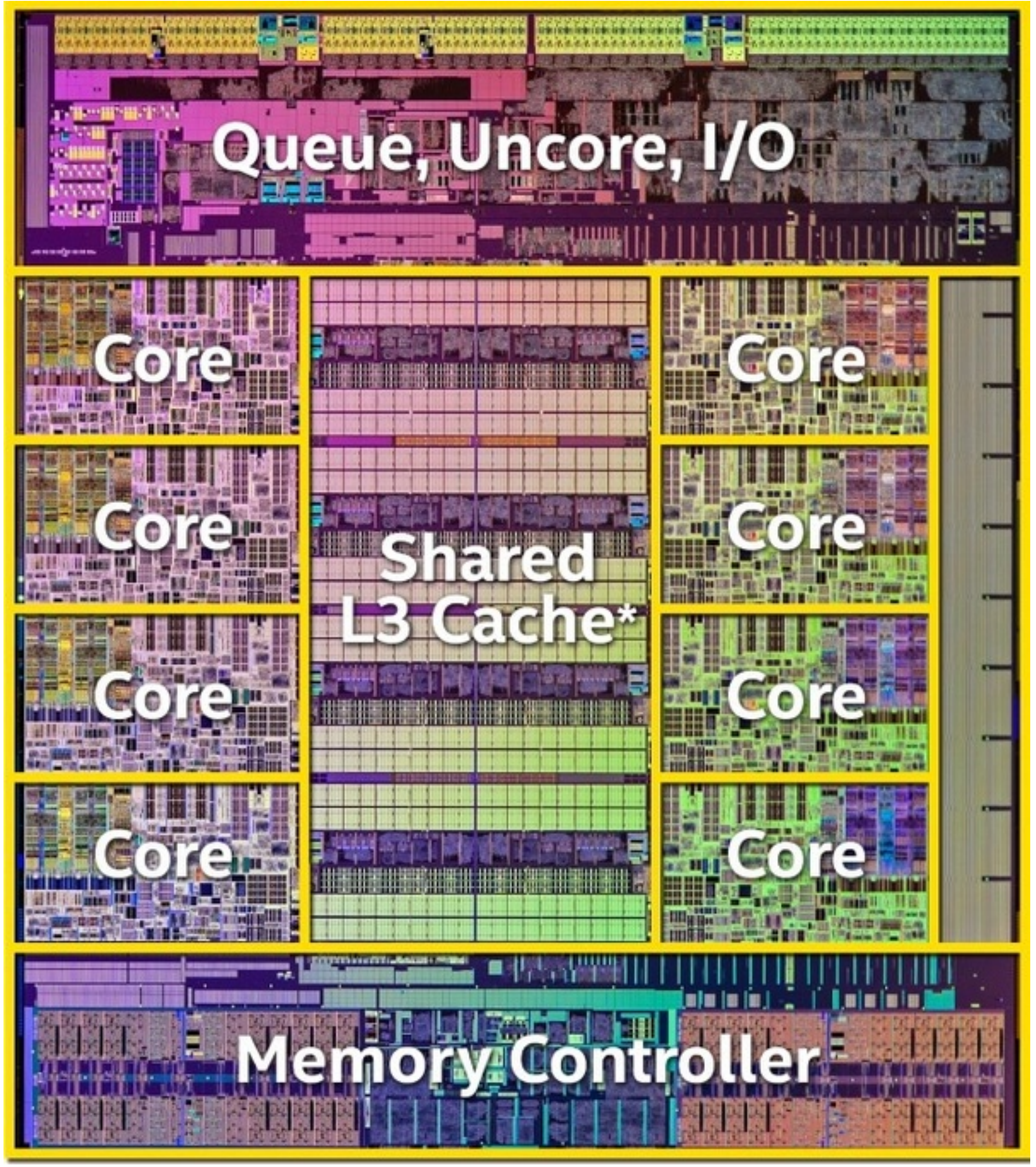


E.g. Intel Haswell

A **shared-memory multiprocessor** (SMP) connects multiple processors to a single memory system.

All threads can access the global memory space.

Multicore processors



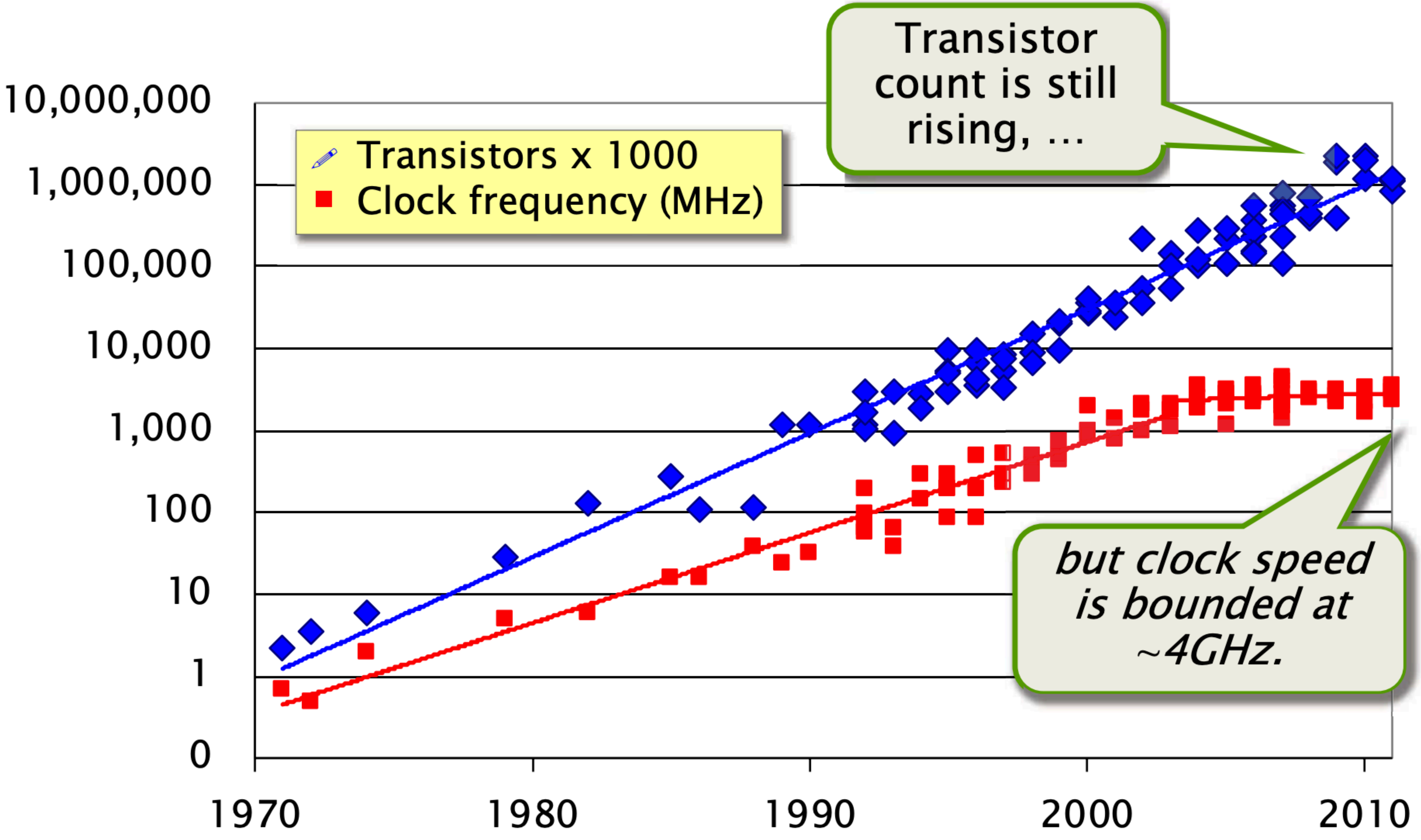
E.g. Intel Haswell

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

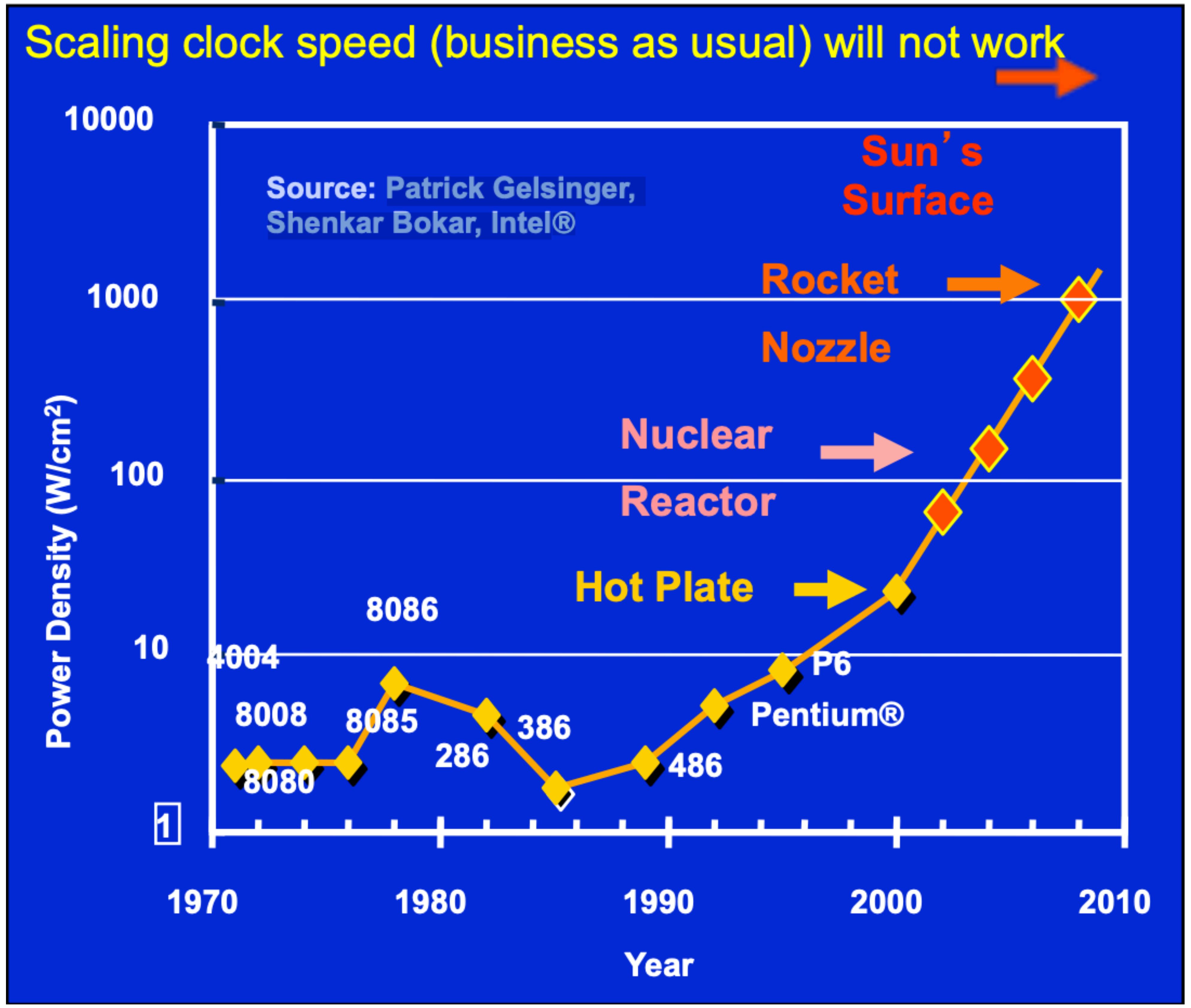
A: Because of **Moore's law** and the end of the scaling of **clock frequency**.

From MIT OCW 6.172

Technology scaling



Recall: Power density



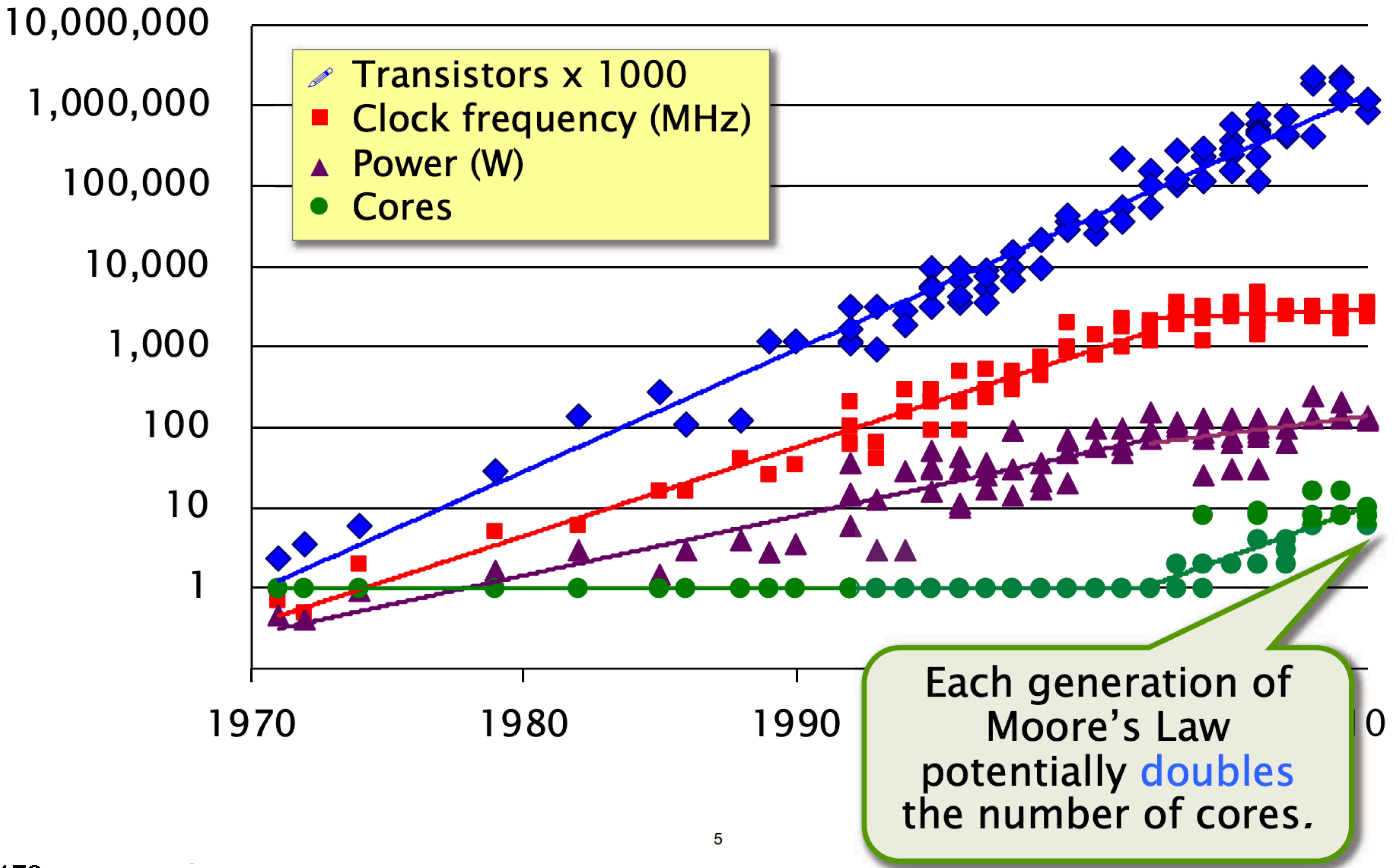
The growth of power density, as seen in 2004, if the scaling of clock frequency had continued its trend of 25%-30% increase per year.

Is it better to increase speed by doubling frequency or cores?

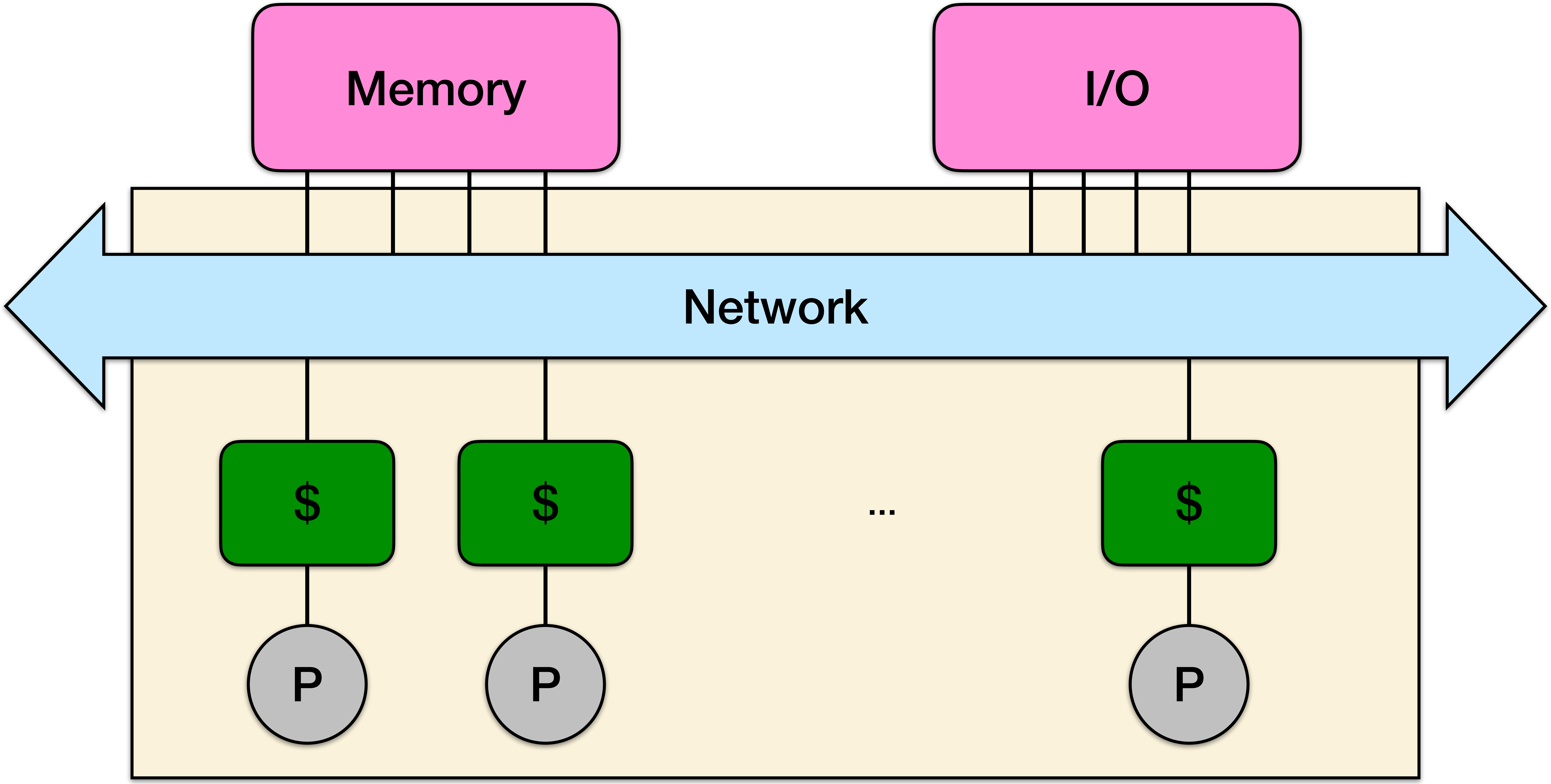
Performance ∝ (cores) x (freq)

Power ∝ (cores) x ($freq^{2.5}$)

Technology scaling



Abstract multicore architecture

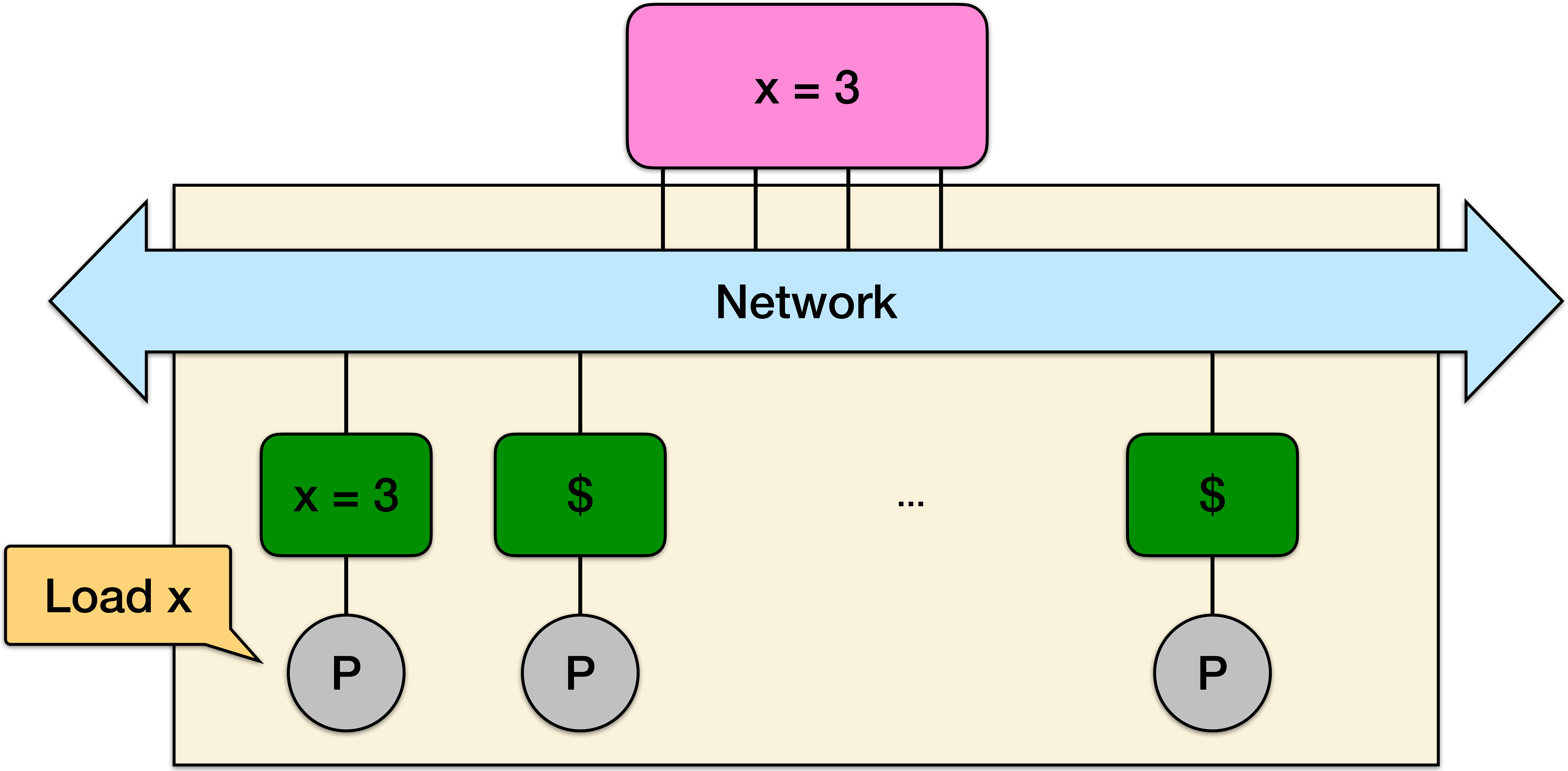


Chip Multiprocessor (CMP)

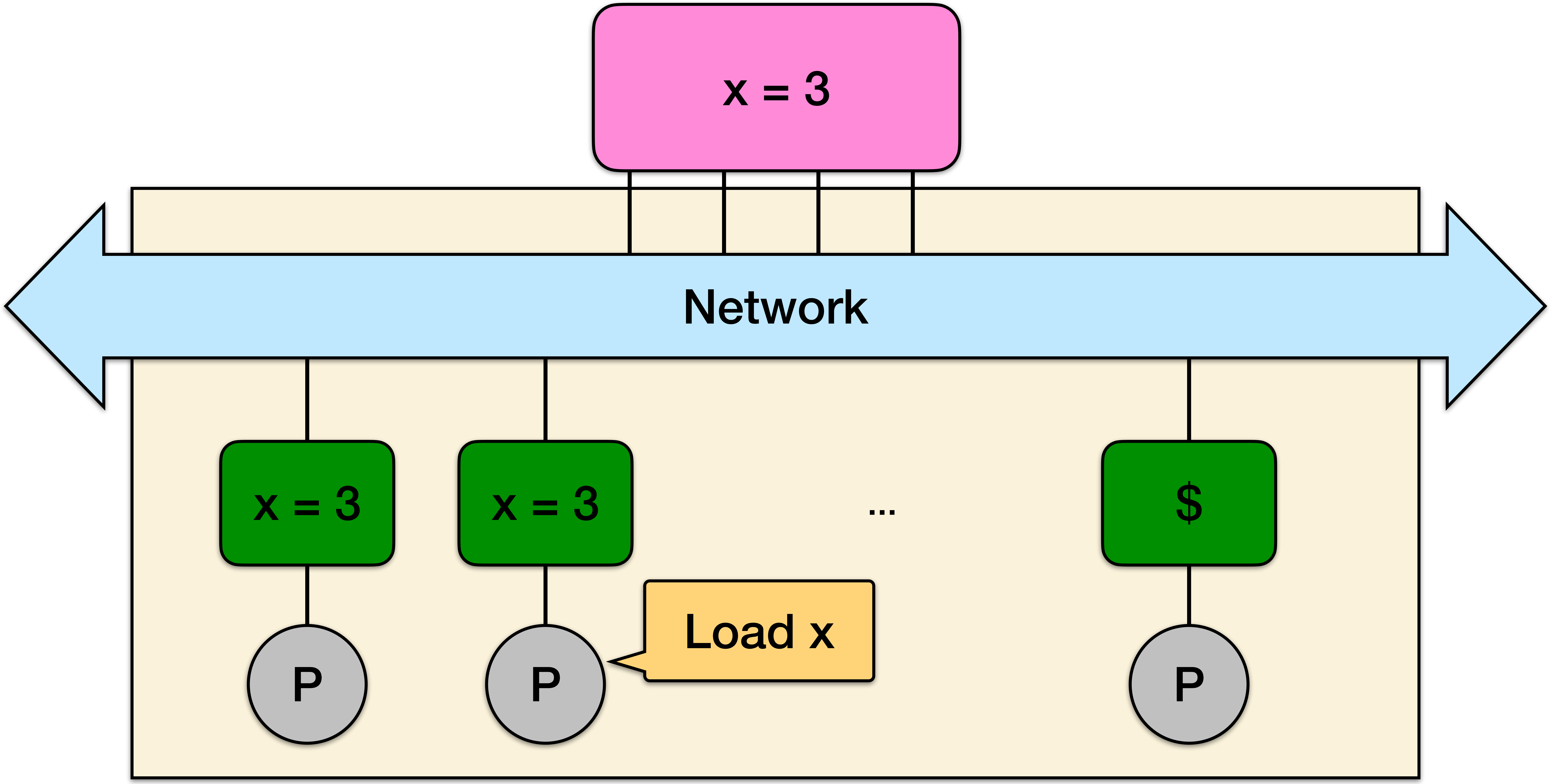
From MIT OCW 6.172

Shared-Memory Hardware

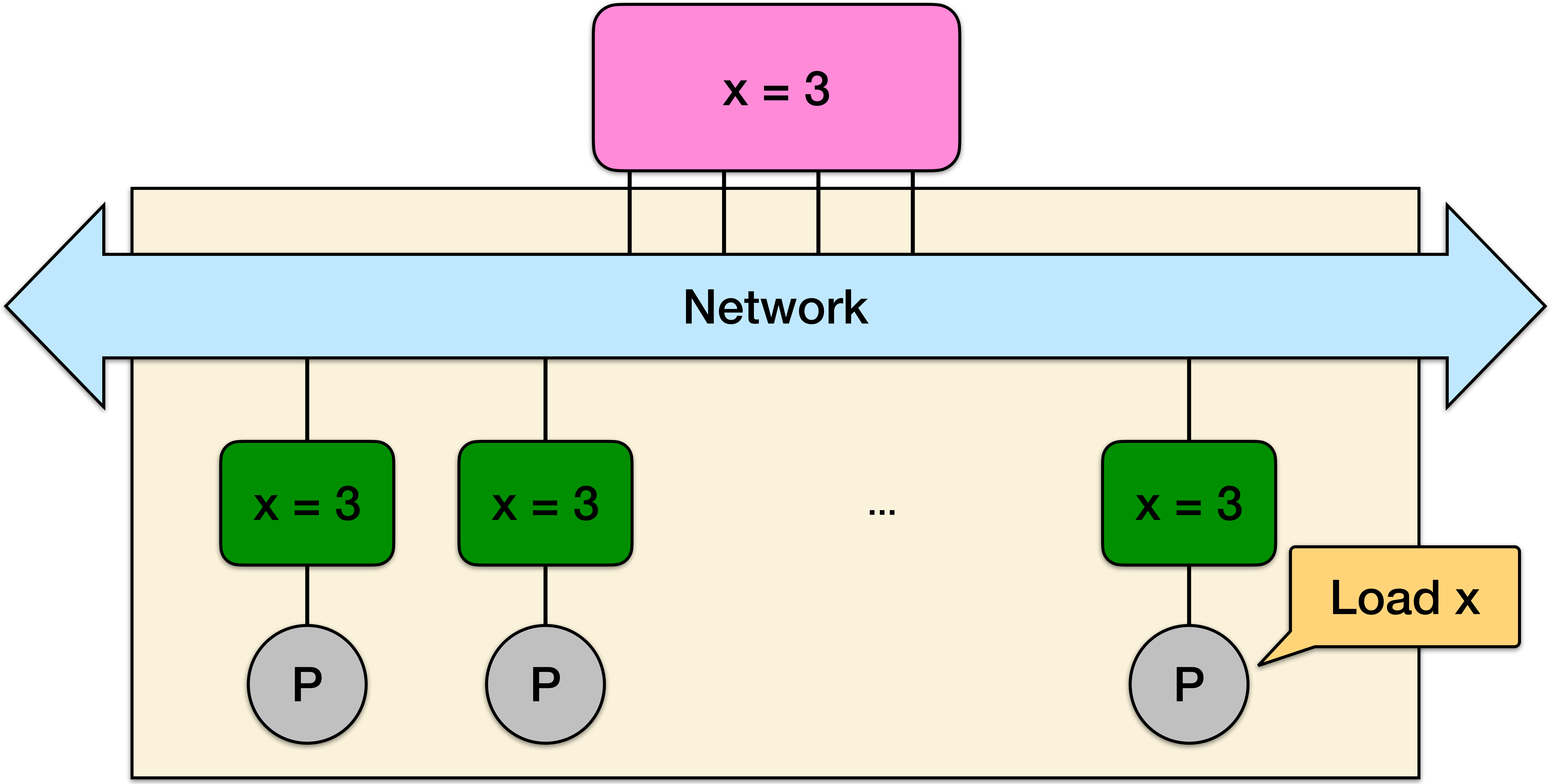
Cache coherence



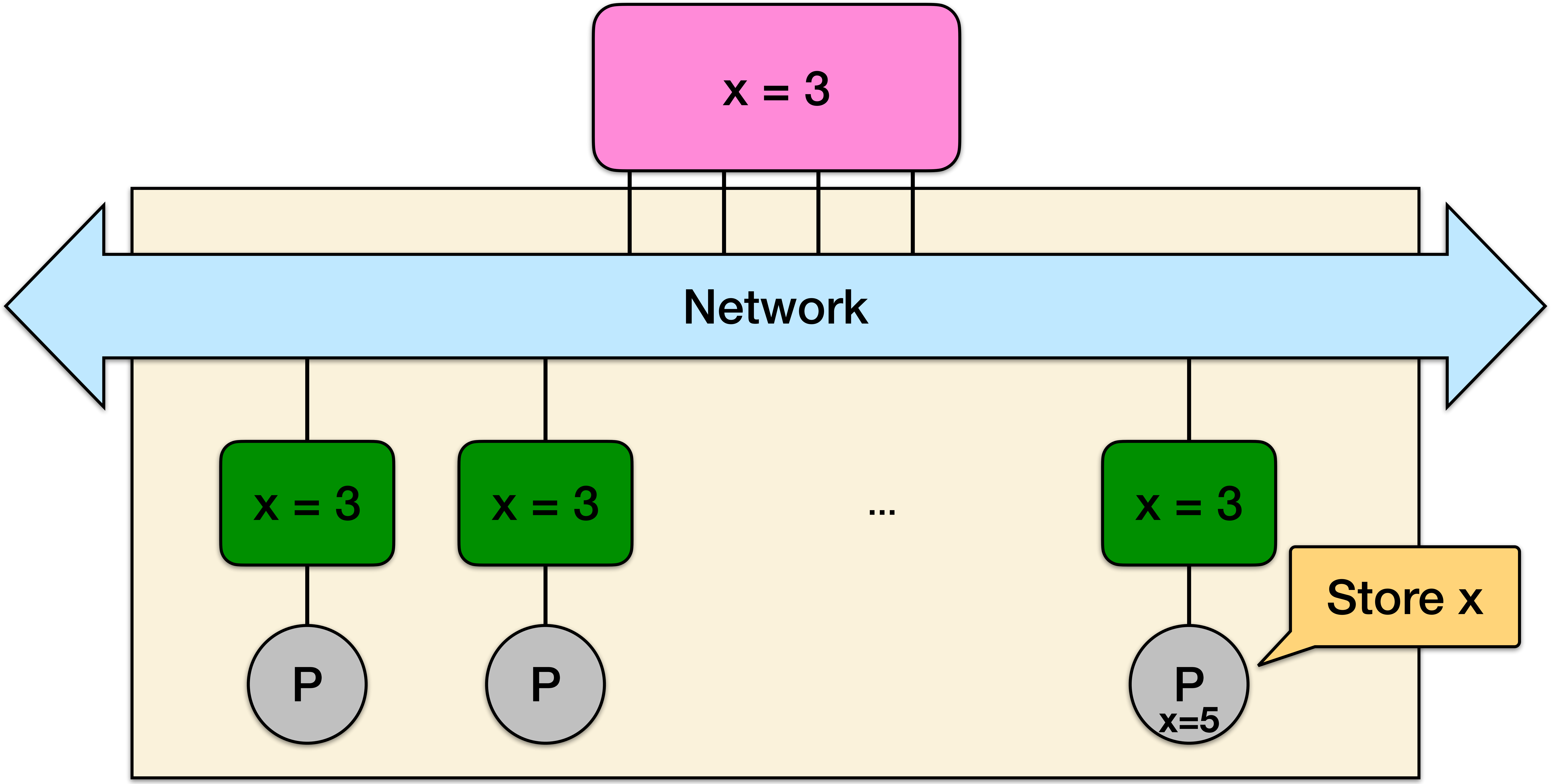
Cache coherence



Cache coherence

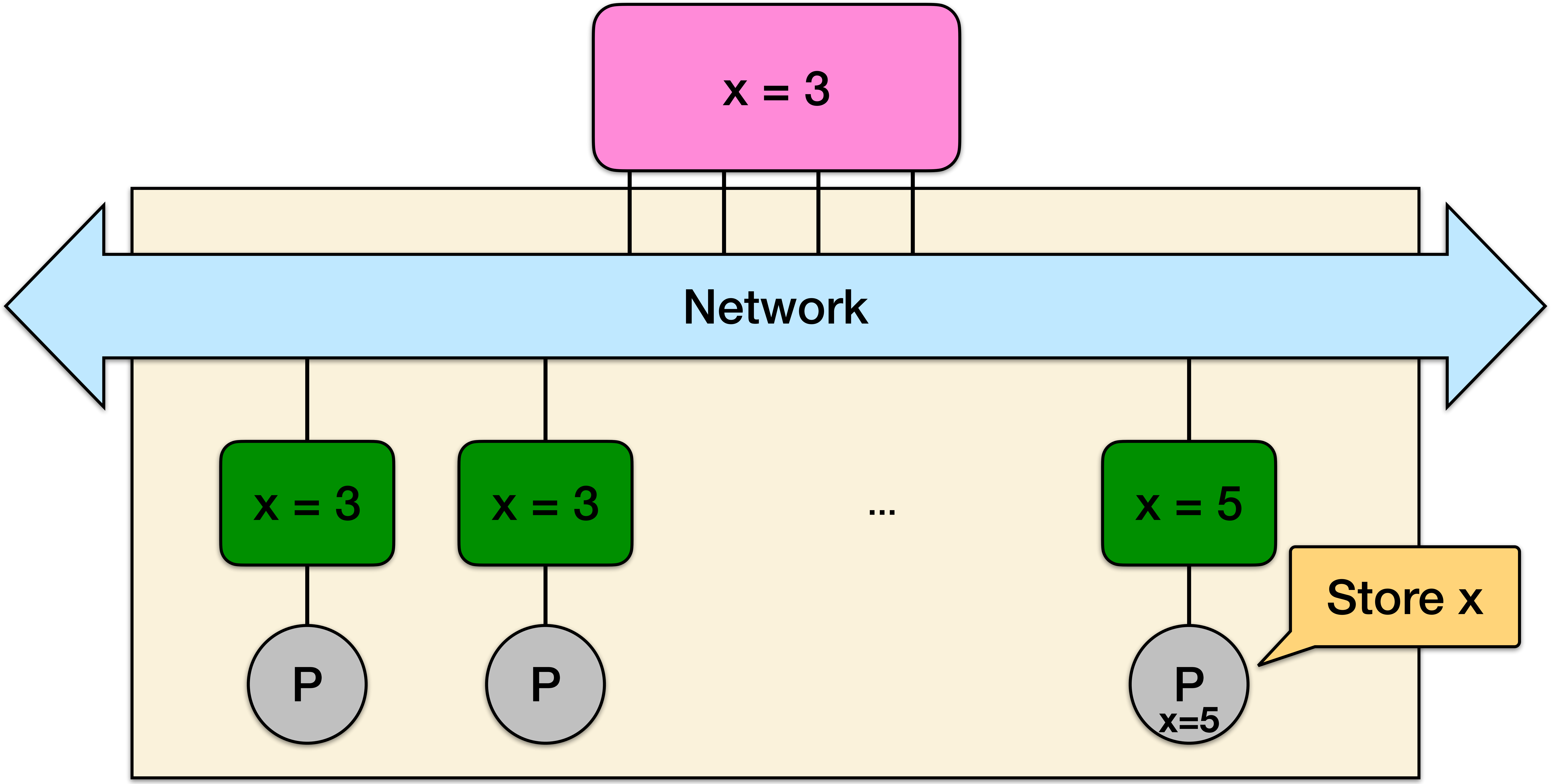


Cache coherence

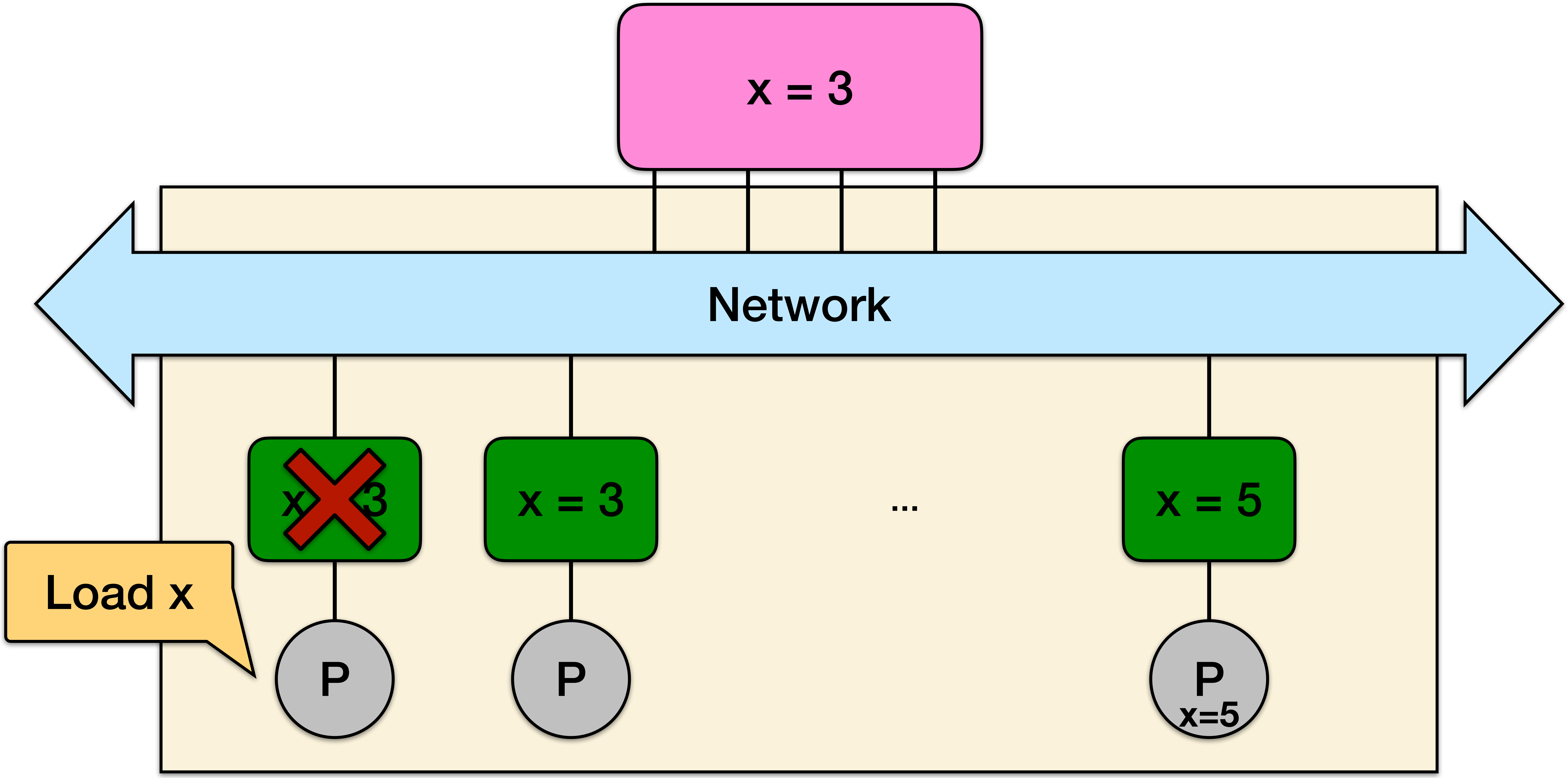


From MIT OCW 6.172

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** (the 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
```

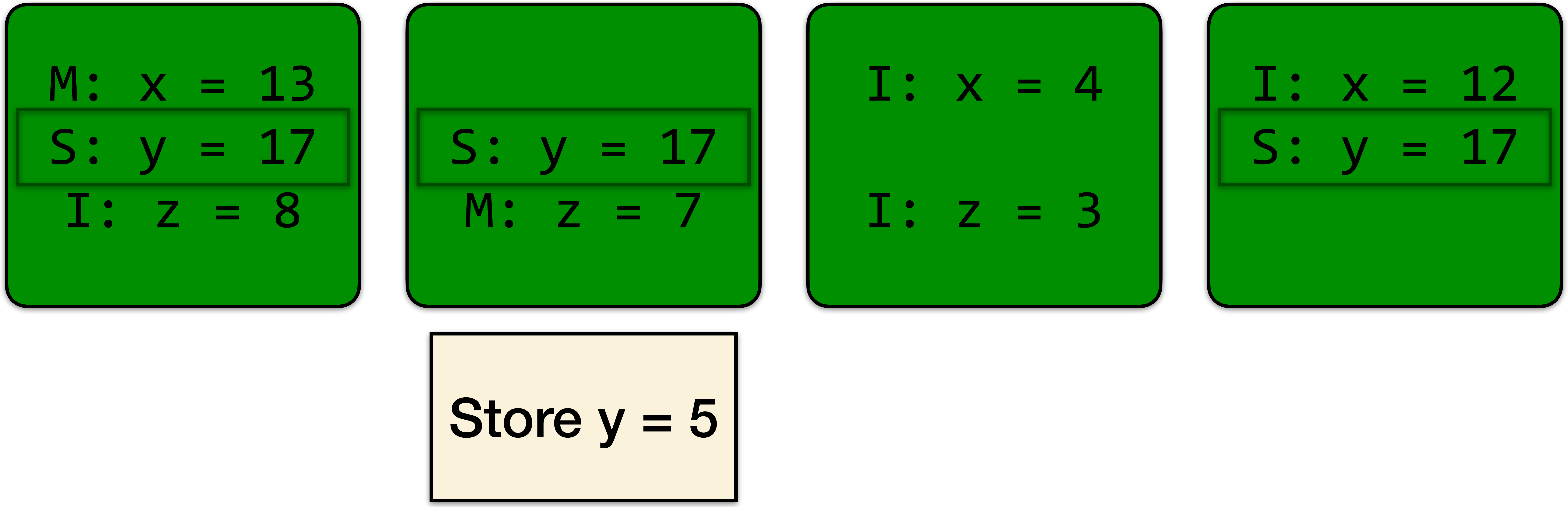
```
I: x = 12
S: y = 17
```

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

MSI Protocol

Each cache line is labeled with a state:

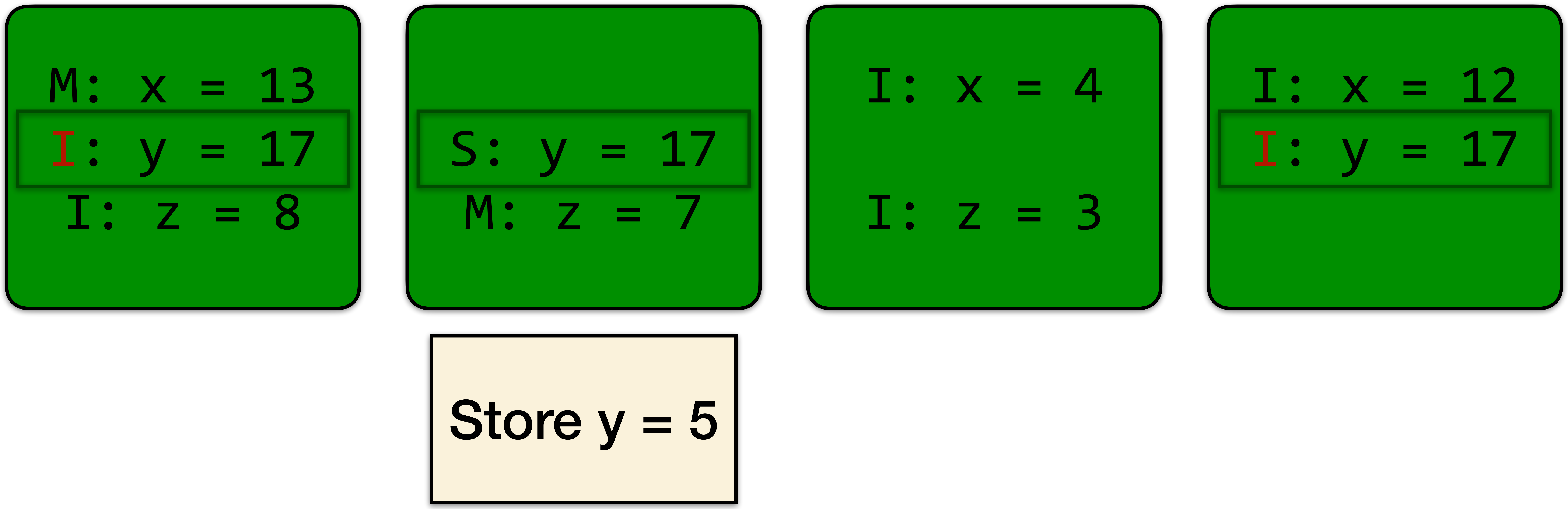
- M: cache block has been **modified**. No other caches contain this block in M or S states.
- S: Other caches may be **sharing** this block
- I: cache block is **invalid** (the same as not there)



MSI Protocol

Each cache line is labeled with a state:

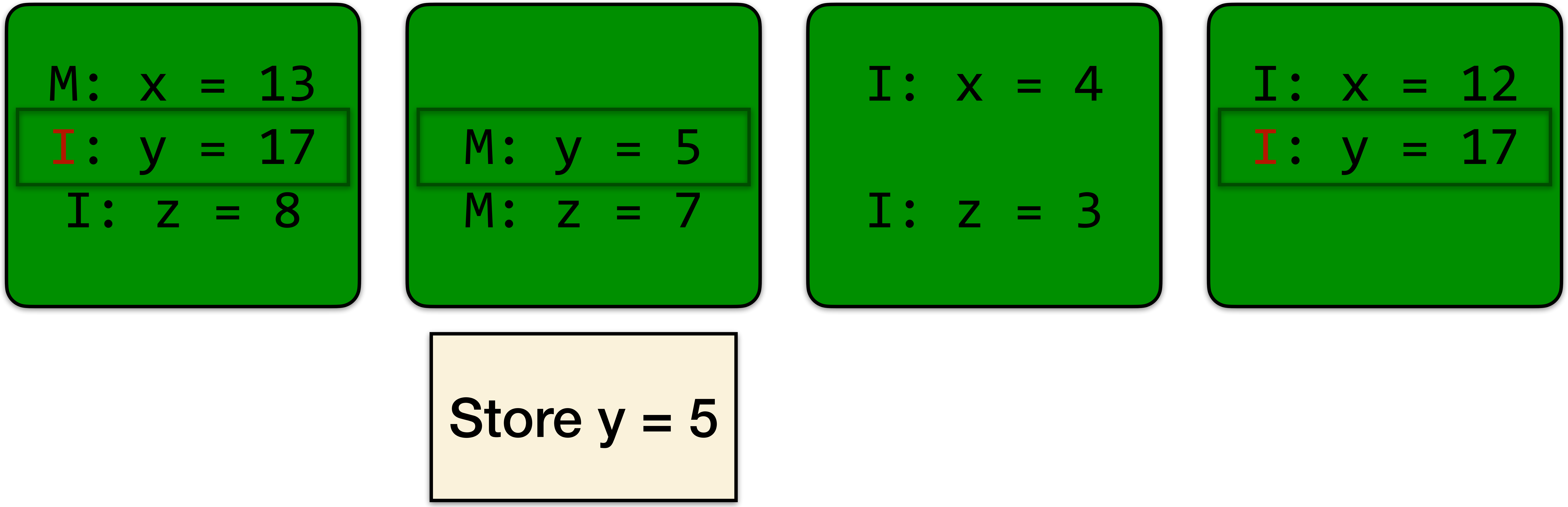
- M: cache block has been **modified**. No other caches contain this block in M or S states.
- S: Other caches may be **sharing** this block
- I: cache block is **invalid** (the same as not there)



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** (the same as not there)



Concurrency Platforms

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 include: Pthreads, Cilk, OpenMP.

Multithreading

Theory **Practice**

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 $n > 1$.

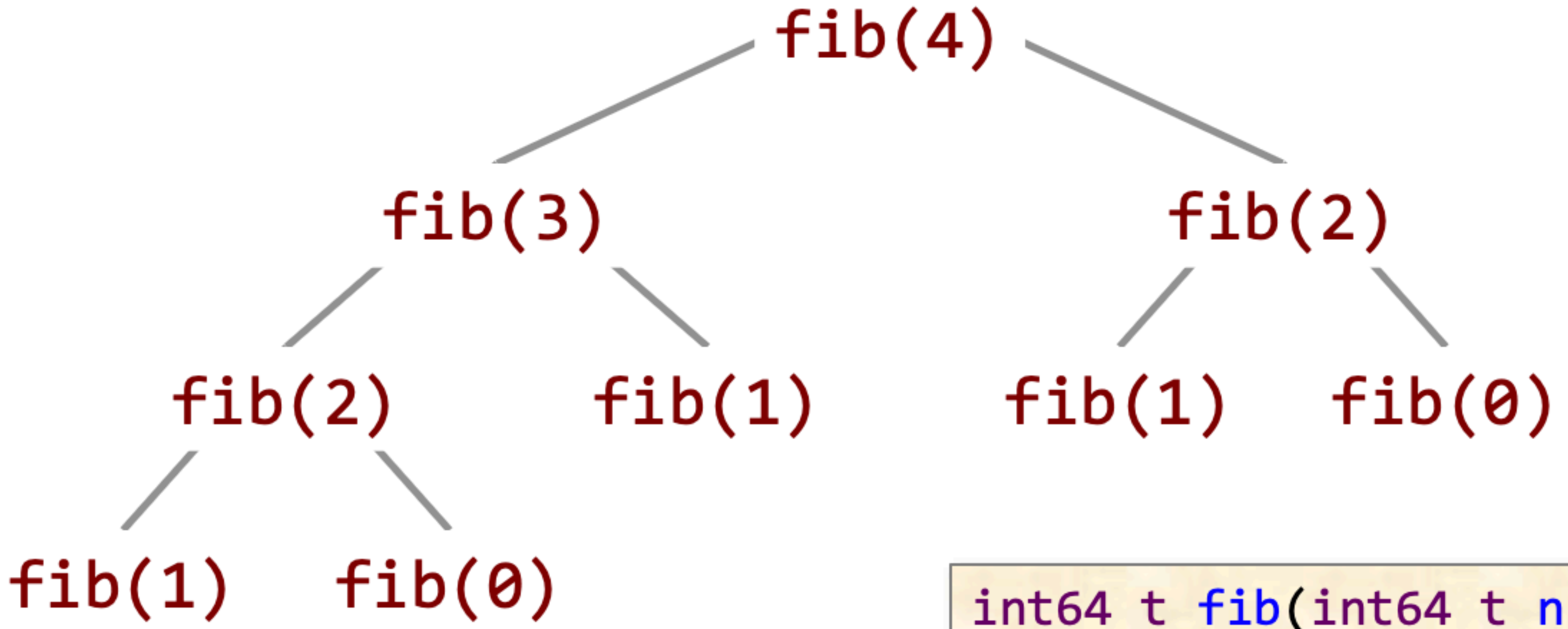
Fibonacci program

```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

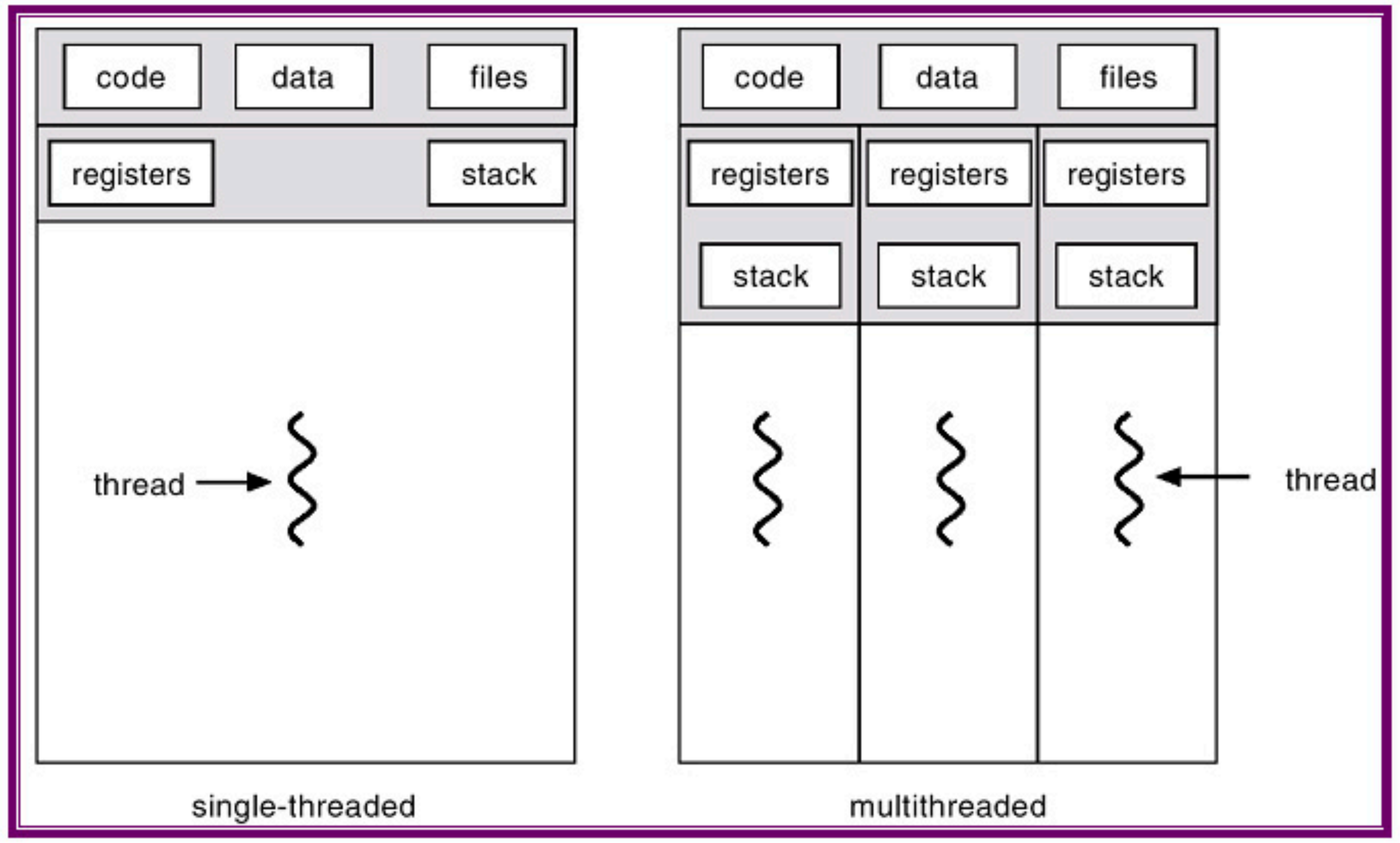


Key idea for parallelization
The calculations of fib(n-1) and fib(n-2) can be executed simultaneously without mutual interference.

```
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);
  }
}
```

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.

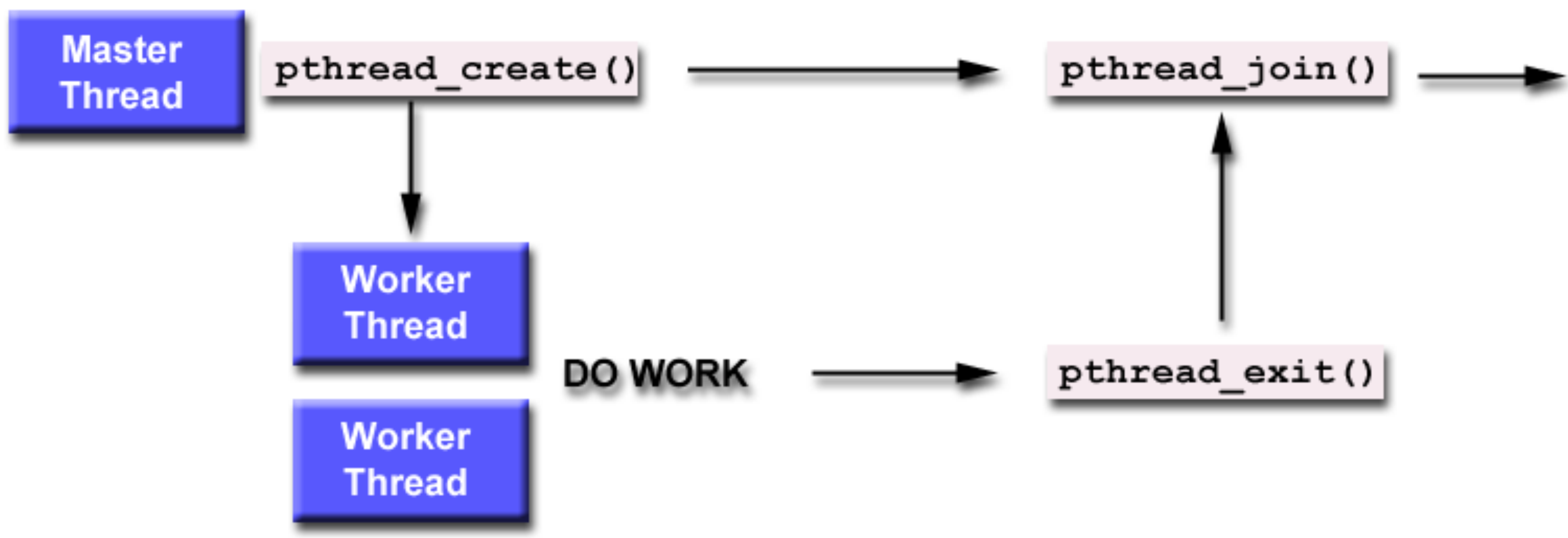


http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html

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
```



https://hpc-tutorials.llnl.gov/posix/joining_and_detaching/

From MIT OCW 6.172

Pthread implementation

Original (serial) 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 implementation

Original (serial) code

Structure for thread arguments

```
#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;
}
```

Pthread implementation

Original (serial) code

Structure for thread arguments

Function called when thread is created

```
#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;
}
```

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;
}
```

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 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], NU
  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;
}
```

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

Marshal input argument to thread

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;
}
```

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

Create thread to execute fib(n-1)

```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], NU
  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 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;
}
```

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

Create thread to execute fib(n-1)

```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], NU
  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

Main thread executes fib(n-2) in parallel

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;
}
```

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

Create thread to execute fib(n-1)

Block until auxiliary thread finishes

```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], NU
  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

Main thread executes fib(n-2) in parallel

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], NU
  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
         n, result);
  return 0;
}
```

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

Marshal input argument to thread

Create thread to execute fib(n-1)

Main thread executes fib(n-2) in parallel

Block until auxiliary thread finishes

Add results together for 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 1958!) and engage in error-prone protocols in order to load-balance.

OpenMP

- First introduced in 1997.
- 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**.

OpenMP®

https://www.openmp.org/

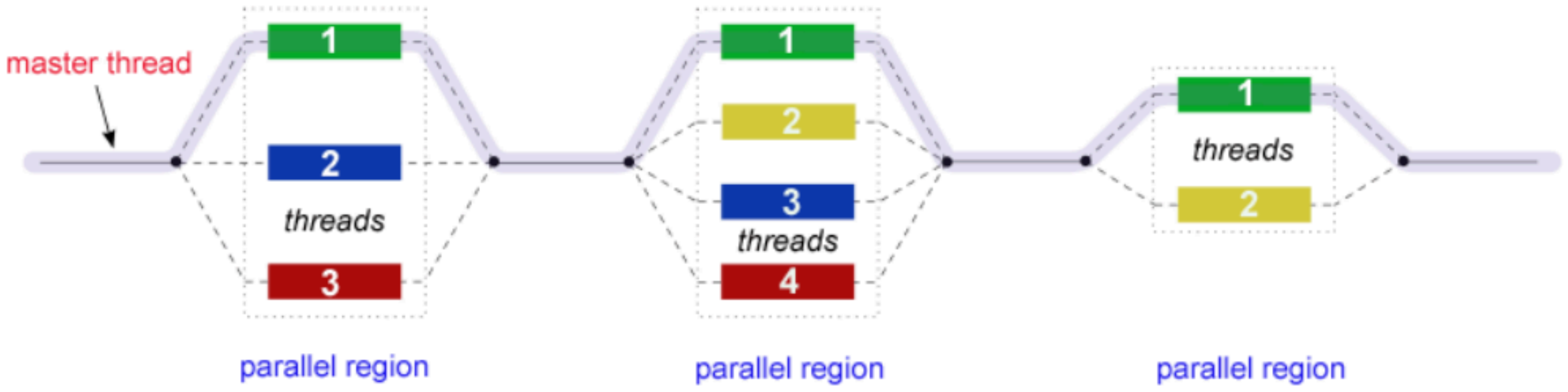
Three OpenMP building blocks

- **Compiler directives** - e.g.,
  - variable types: private, shared
  - parallel tasks, parallel for
- **Runtime libraries / APIs** - e.g.,
  - omp_set/get_num_threads, omp_get_thread_num, etc.
- **Environment variables** - e.g.,
  - OMP_NUM_THREADS, OMP_SCHEDULE, etc.

From Ramki Kannan

Fork-join model

- OpenMP programs **begin with a single process**: the master thread.
- The master thread executes sequentially until the first parallel region construct is encountered.
- **Fork**: the master thread then creates a team of parallel threads to execute the code in the parallel region.
- **Join**: When the team threads complete the statements in the parallel region, they synchronize and terminate, leaving only the master thread.



OpenMP usage in C/C++

- Add `#include <omp.h>` at the top of your file with the other includes
- Usage: `#pragma omp directive [clauses] newline`
- Compile with the `-fopenmp` flag

Fibonacci in OpenMP

Compiler directive

```
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);
  }
}
```

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

The following statement is an independent task

Fibonacci in OpenMP

Compiler directive

```
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

Sharing of memory is managed explicitly

Shared variables have one version for all the threads.

Most variables are **shared by default**: with a few exceptions e.g., iteration variables

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

The following statement is an independent task

Sharing of memory is managed explicitly

Wait for the two tasks to complete before continuing

Shared variables have one version for all the threads.

Most variables are **shared by default**: with a few exceptions e.g., iteration variables

Parallel for example- Saxpy

```
#pragma omp parallel for
for (i = 0; i < n; i++) {
  y[i] = a*x[i] + y[i]
}
```

OpenMP compiler directive

$$\begin{array}{c} a \cdot \\ a \cdot \\ \\ a \cdot \end{array} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \begin{array}{c} + \\ + \\ \\ + \end{array} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_d \end{bmatrix} \begin{array}{c} = \\ = \\ \\ = \end{array} \begin{bmatrix} a \cdot x_1 + y_1 \\ a \cdot x_2 + y_2 \\ \vdots \\ a \cdot x_d + y_d \end{bmatrix}$$

OMP Load Balancing

OpenMP provides different methods to divide iterations among threads, indicated by the **schedule** clause: schedule (<method>, [chunk size])

Methods include:

- Static: the default schedule; divide iterations into chunks according to size, then distribute chunks to each thread in **a round-robin manner**.
- Dynamic: each thread grabs a chunk of iterations, then **requests another chunk upon completion of the current one**, until all iterations are executed.

4 threads, 100 iterations

Schedule	Iterations mapped onto thread			
	0	1	2	3
Static	1-25	26-50	51-75	76-100
Static,20	1-20, 81-100	21-40	41-60	61-80
Dynamic	1, ···	2, ···	3, ···	4, ···
Dynamic,10	1 – 10, ···	11 – 20, ···	21 – 30, ···	31 – 40, ···

From Ramki Kannan

OMP Variables Scope - private

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

int main(void){
  int i;
  int x;
  x=44;
  #pragma omp parallel for private(x)
  for(i=0;i<=10;i++){
    x=i;
    printf("Thread number: %d     x:
    %d\n”,omp_get_thread_num(),x);
  }
  printf("x is %d\n", x);
}
```

Creates local (uninitialized) copy of the specified variables for each thread

Thread num at runtime

```
Thread number: 0     x: 0
Thread number: 0     x: 1
Thread number: 0     x: 2
Thread number: 3     x: 9
Thread number: 3     x: 10
Thread number: 2     x: 6
Thread number: 2     x: 7
Thread number: 2     x: 8
Thread number: 1     x: 3
Thread number: 1     x: 4
Thread number: 1     x: 5
x is 44
```

https://michaellindon.github.io/lindonslog/programming/openmp/openmp-tutorial-firstprivate-and-lastprivate/

OMP Variables Scope - lastprivate

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

int main(void){
  int i;
  int x;
  x=44;
  #pragma omp parallel for lastprivate(x)
  for(i=0;i<=10;i++){
    x=i;
    printf("Thread number: %d      x:
    %d\n”,omp_get_thread_num(),x);
  }
  printf("x is %d\n", x);
}
```

Special case of private: allows us to keep value of x at the **last iteration** after the parallel region

```
Thread number: 3     x: 9
Thread number: 3     x: 10
Thread number: 1     x: 3
Thread number: 1     x: 4
Thread number: 1     x: 5
Thread number: 0     x: 0
Thread number: 0     x: 1
Thread number: 0     x: 2
Thread number: 2     x: 6
Thread number: 2     x: 7
Thread number: 2     x: 8
x is 10
```

https://michaellindon.github.io/lindonslog/programming/openmp/openmp-tutorial-firstprivate-and-lastprivate/

OMP Variables Scope - firstprivate

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

int main(void){
  int i;
  int x;
  x=44;
  #pragma omp parallel for firstprivate(x)
  for(i=0;i<=10;i++){
    x=i;
    printf("Thread number: %d     x:
    %d\n”,omp_get_thread_num(),x);
  }
  printf("x is %d\n", x);
}
```

Special case of private: **initializes** each thread's private copy of the variable with the value it had before the parallel construct

```
Thread number: 3     x: 9
Thread number: 3     x: 10
Thread number: 1     x: 3
Thread number: 1     x: 4
Thread number: 1     x: 5
Thread number: 0     x: 0
Thread number: 0     x: 1
Thread number: 0     x: 2
Thread number: 2     x: 6
Thread number: 2     x: 7
Thread number: 2     x: 8
x is 44
```

https://michaellindon.github.io/lindonslog/programming/openmp/openmp-tutorial-firstprivate-and-lastprivate/

OMP Reduction Example

```
int main()
{
  int i, n;
  n = 10000;
  float a[n], b[n];
  double result, sequential_result;
  /* Some initializations */
  result = 0.0;
  for (i = 0; i < n; i++) {
    a[i] = i * 1.0; b[i] = i * 2.0;
  }

  #pragma omp parallel for default(shared)
  private(i) schedule(static)
  reduction(+ : result)
  for (i = 0; i < n; i++)
    result = result + (a[i] * b[i]);
  printf("Final result= %f\n", result);
  return 0;
}
```

- The reduction clause allows **accumulative operations** on the value of variables.
- Syntax:
  - `reduction (operator:variable list)`
- Operators:
  - Arithmetic (i.e., +, *, -)
  - Bitwise (i.e., &, |, ^)
  - Logical (i.e., &&, ||)

Determinacy Races and Mutual Exclusion

Determinacy Races

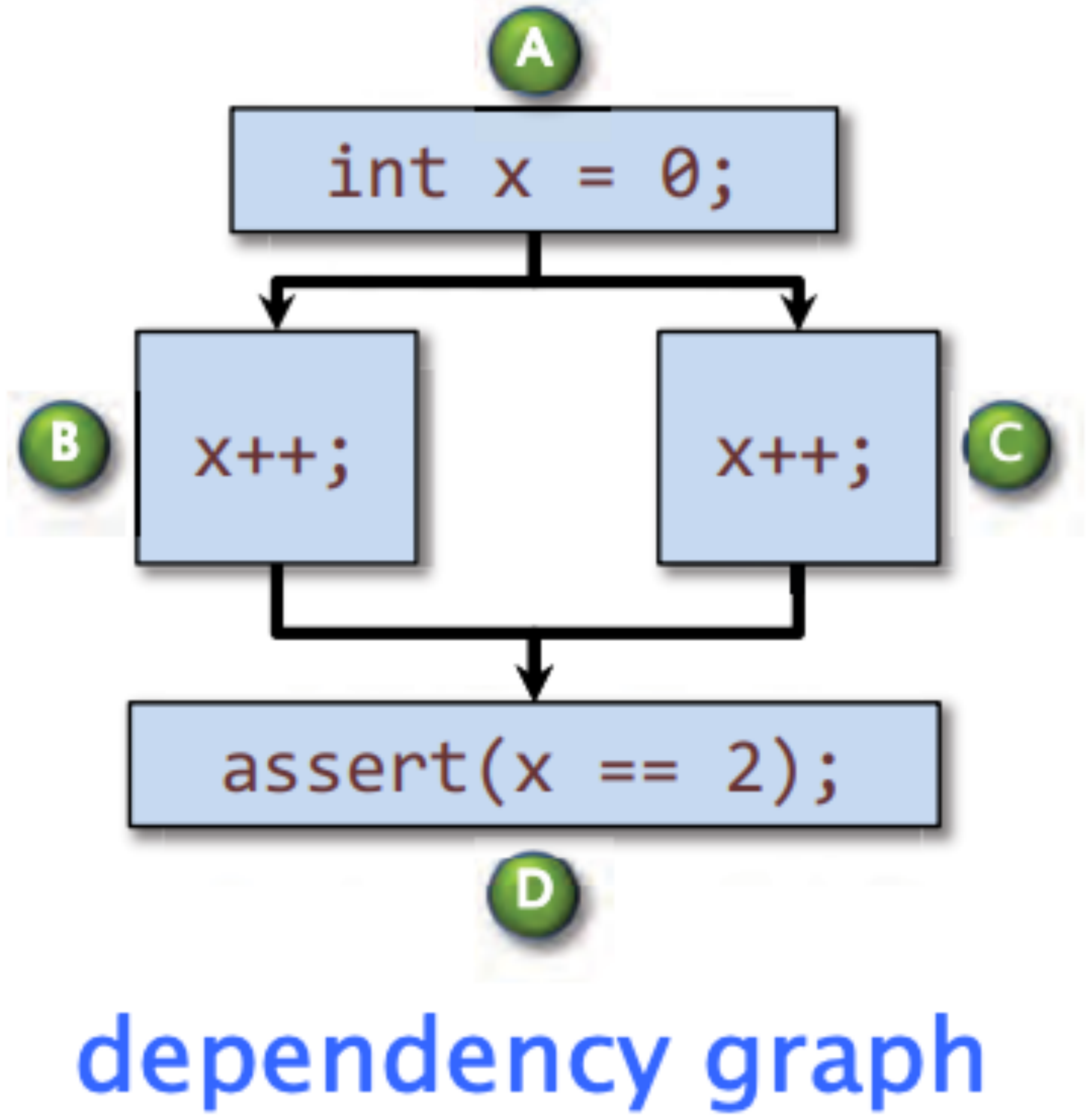
A **determinacy race** occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

A

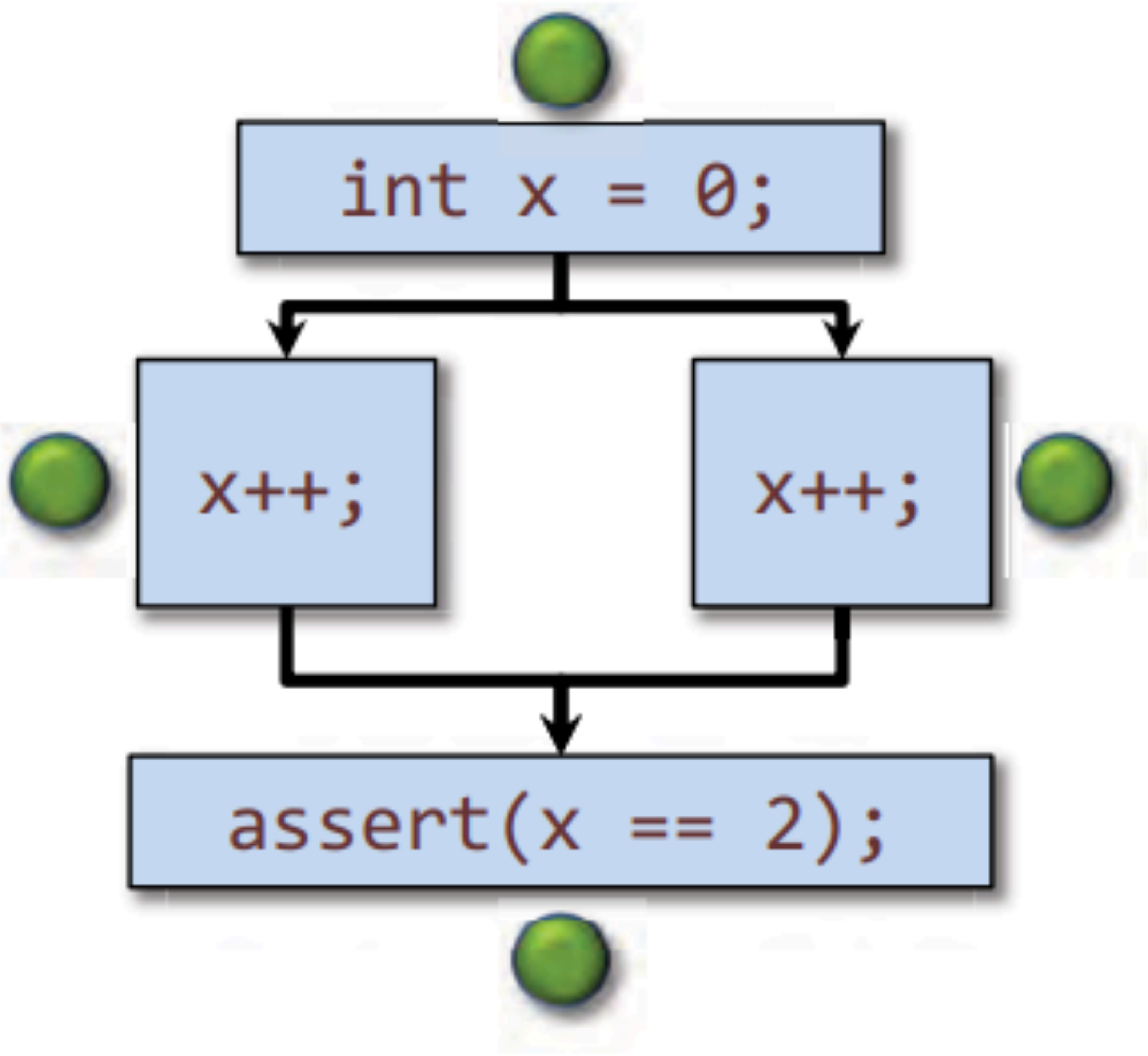
B, C

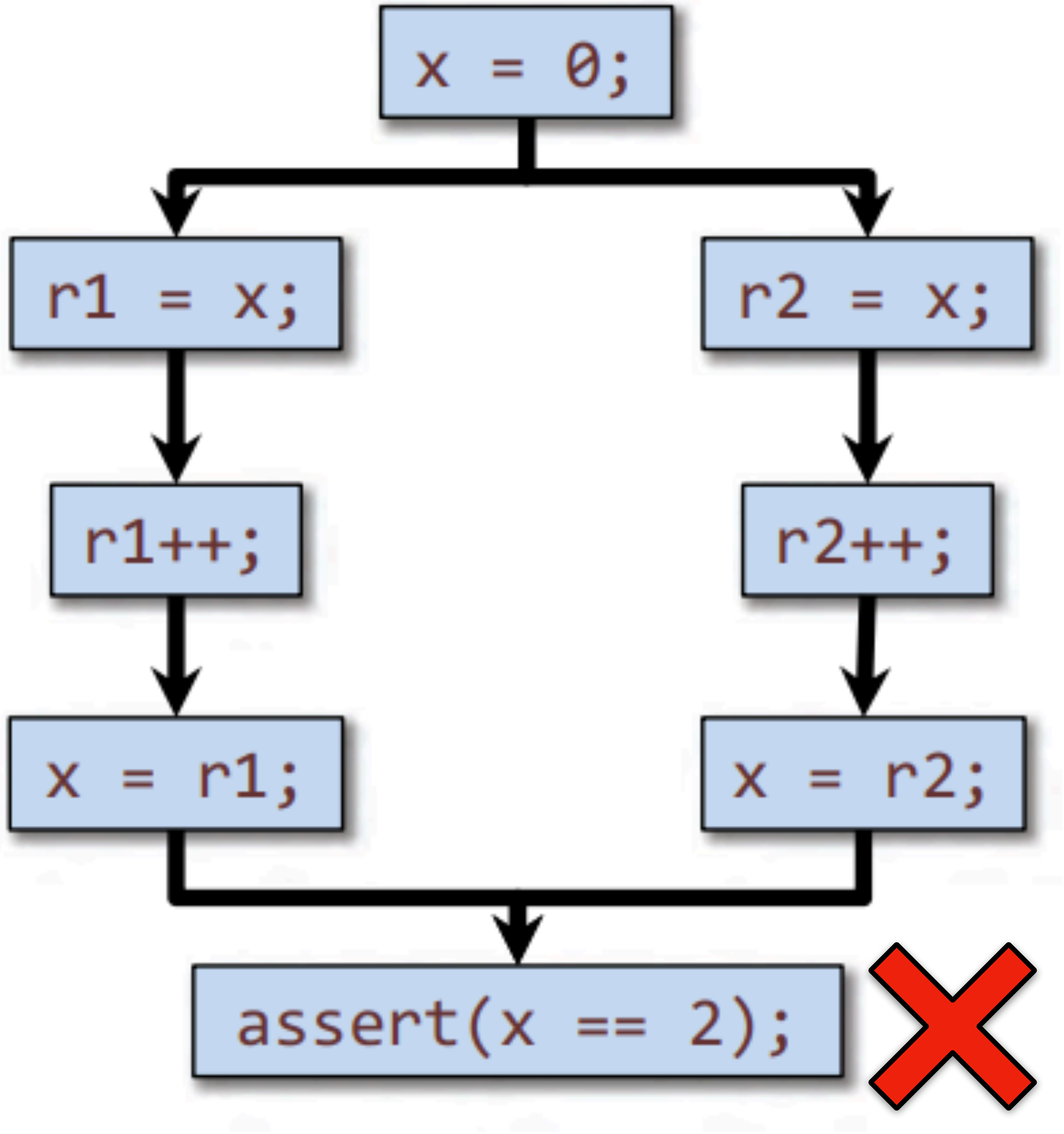
D

```
int x = 0;

#pragma omp parallel for
for (i = 0; i < 2; i++) {
  x++;
}
assert(x == 2);
```



A Closer Look





From MIT OCW 6.172

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).

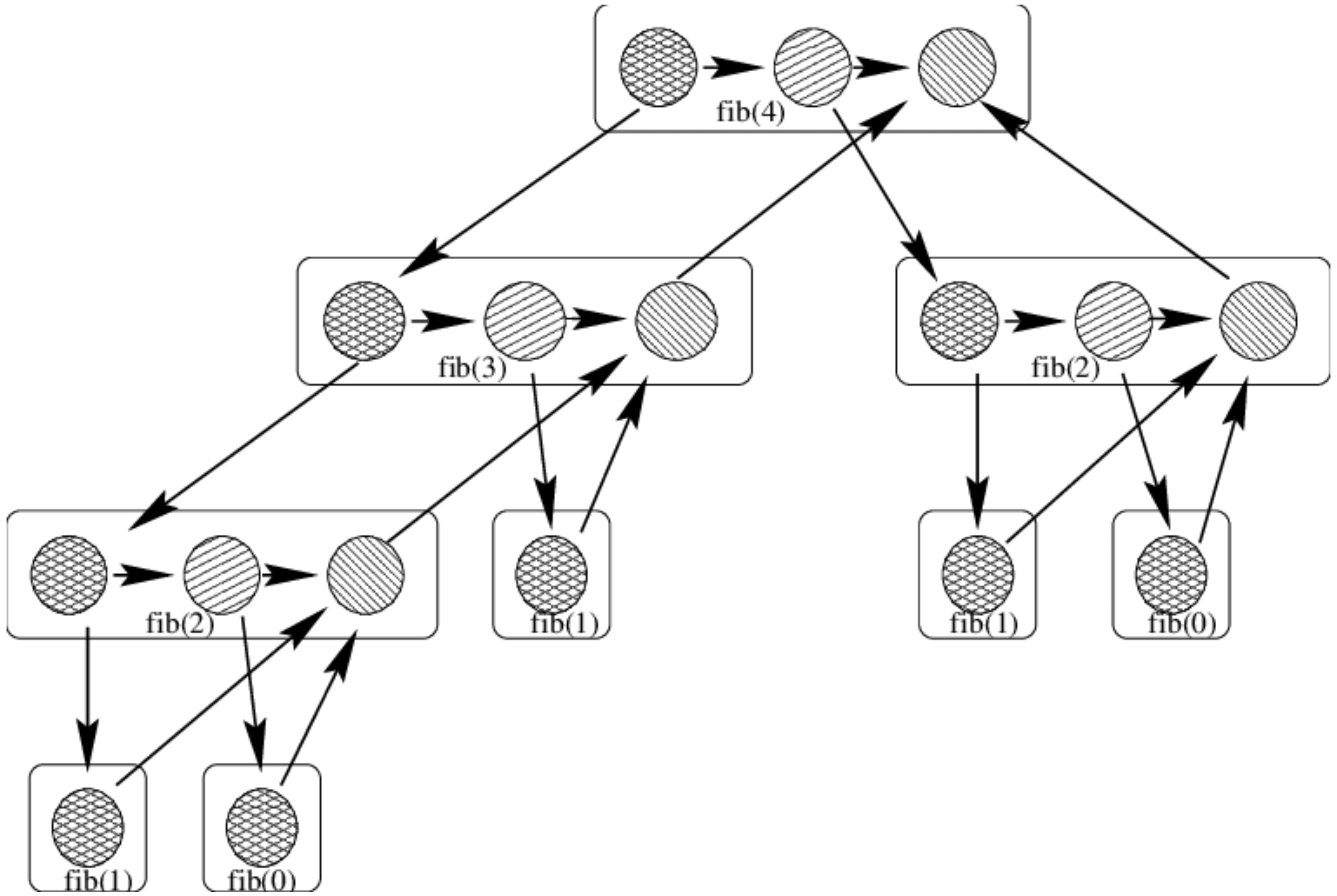
A	B	Race Type
read	read	none
read	write	read race
write	read	read race
write	write	write race

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

Avoiding Races

- Iterations of **parallel for** should ideally be independent.

- Between parallel tasks and the corresponding taskwait, the code of the spawned task should be **independent of the code of the parent**, including code executed by additional spawned tasks.



https://www.researchgate.net/figure/A-dag-representing-the-multithreaded-computation-of-Fib4-Threads-are-shown-as-circles_fig1_2817427

Locks and Mutual Exclusion

A thread lock is a form of **mutual exclusion**. If you must access the same data in parallel, use locks to protect it.

A lock can be held by at most **one thread at a time**.

```
lock_t lock;

parallel_for ( i = 0; i < n; i++ ) {
  set_lock(&lock);

  // only one thread at a time can access this part

  unset_lock(&lock);
}
```

Note: Locks are not cheap (performance-wise)

OMP Locks

The syntax for locks in OpenMP (`omp_lock_t`) is as follows:

The four operations are:

- `omp_init_lock(omp_lock_t *)` – initialize a lock
- `omp_set_lock(omp_lock_t*)` – wait until the lock is available, then set it. No other thread can set the lock until it is released
- `omp_unset_lock(omp_lock_t*)` – unset (release) the lock
- `omp_destroy_lock(omp_lock_t*)` – The reverse of omp_init_lock

https://wgropp.cs.illinois.edu/courses/cs598-s15/lectures/lecture19.pdf

https://www.openmp.org/spec-html/5.0/openmpse31.html

OMP Lock Example

Declare lock

Initialize lock

Set the lock so only this thread has access

Unset the lock when the thread is done to allow others to take it

```
omp_lock_t writelock;

omp_init_lock(&writelock);

#pragma omp parallel for
for ( i = 0; i < n; i++ ) {
  omp_set_lock(&writelock);

  // only one thread at a time can access this part

  omp_unset_lock(&writelock);
}

omp_destroy_lock(&writelock);
```

(next lecture will be all about nondeterminism and locks)

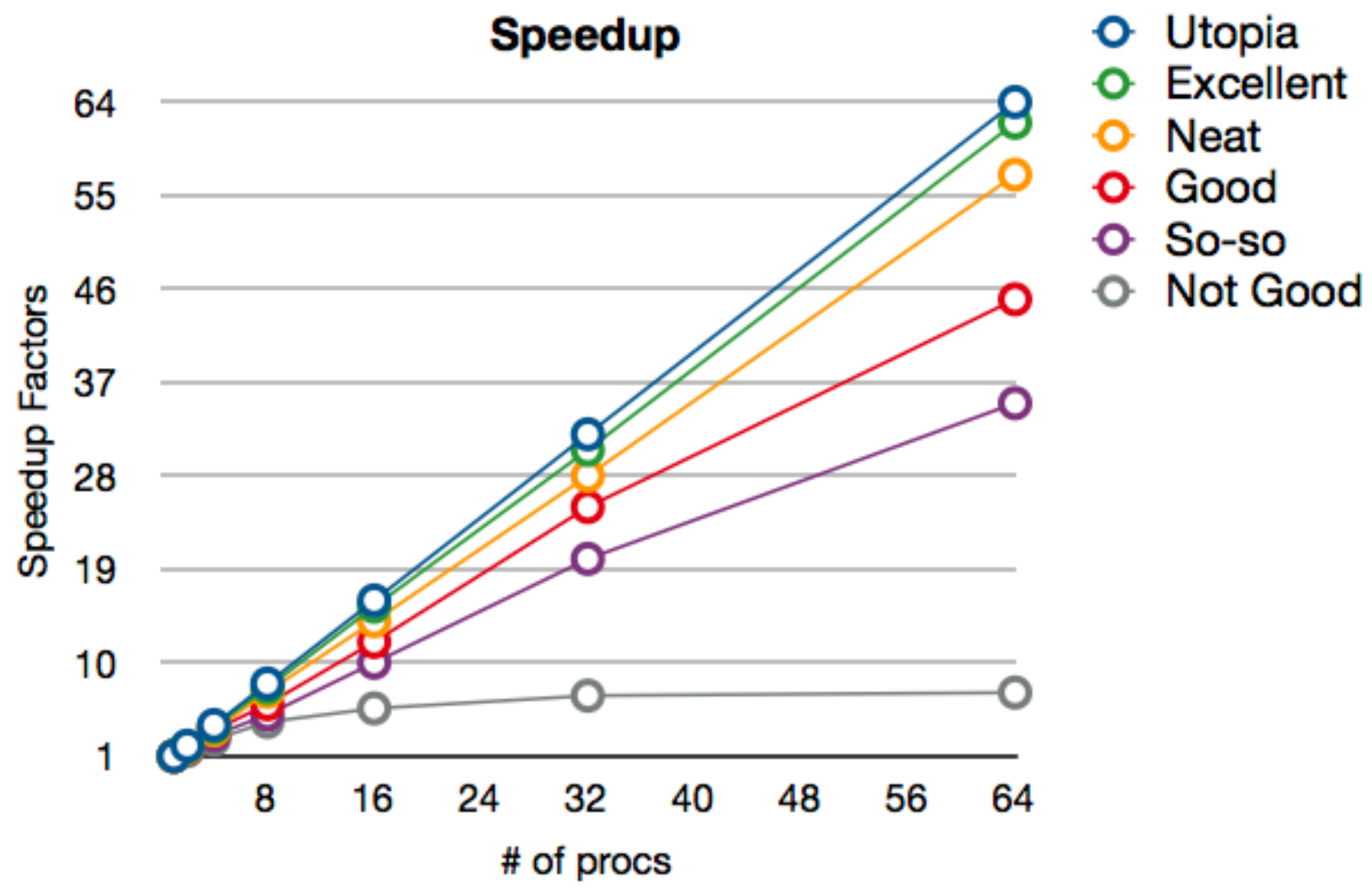
https://www.openmp.org/spec-html/5.0/openmpse31.html

Measuring Parallel Programs

Parallel Scalability

Scalability is the ability of hardware and software to deliver greater computational power **when the amount of resources is increased**.

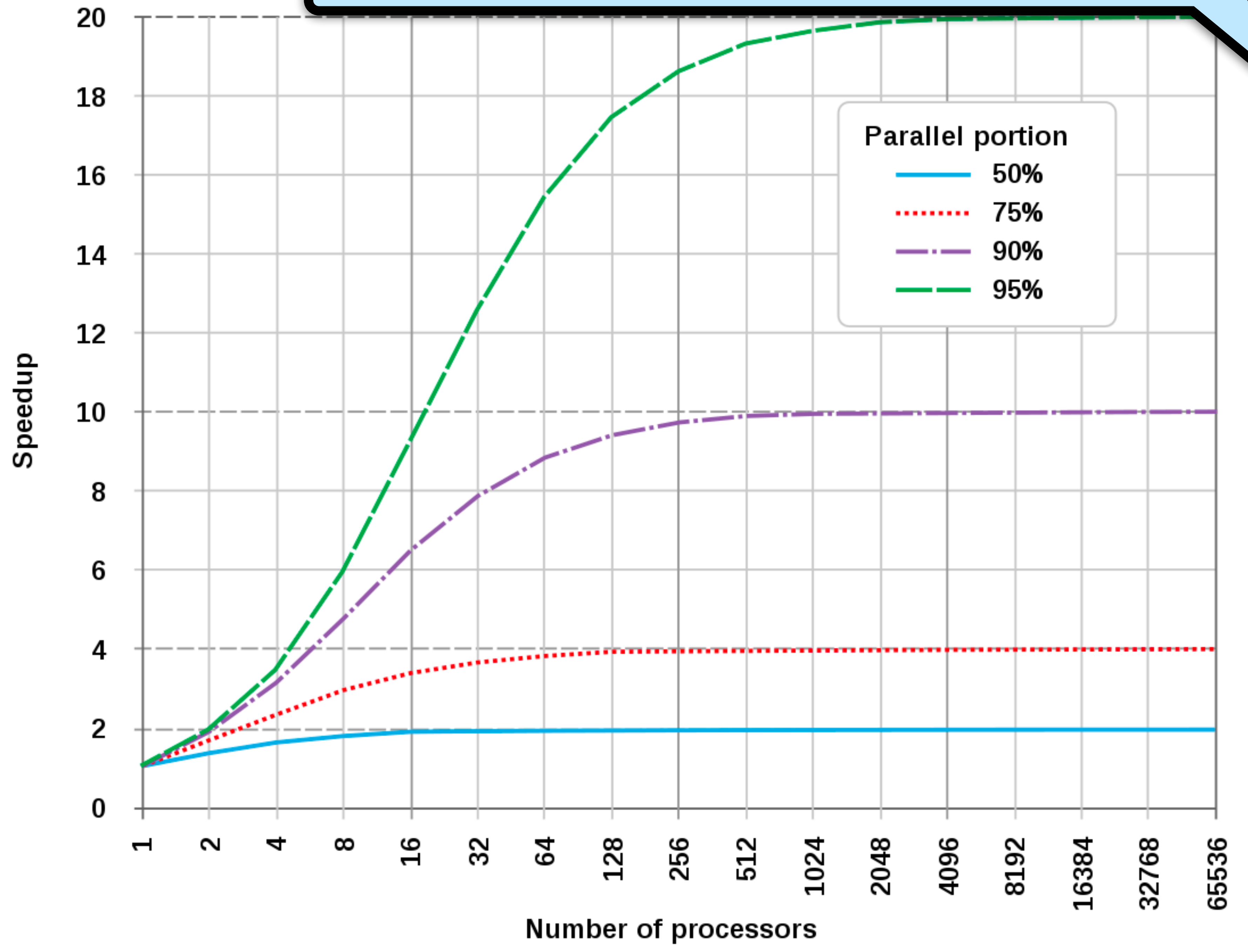
Speedup in parallel computing is defined as T_1/T_p, where T_1 is the serial time and T_p is the time to run on p processors.



https://web.eecs.utk.edu/~huangj/hpc/hpc_intro.php
https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

Amdahl's Law and Strong Scaling

Work done by the **best sequential algorithm**



Amdahl's law bounds **strong scaling**, or the speedup of a fixed problem given more parallel resources.

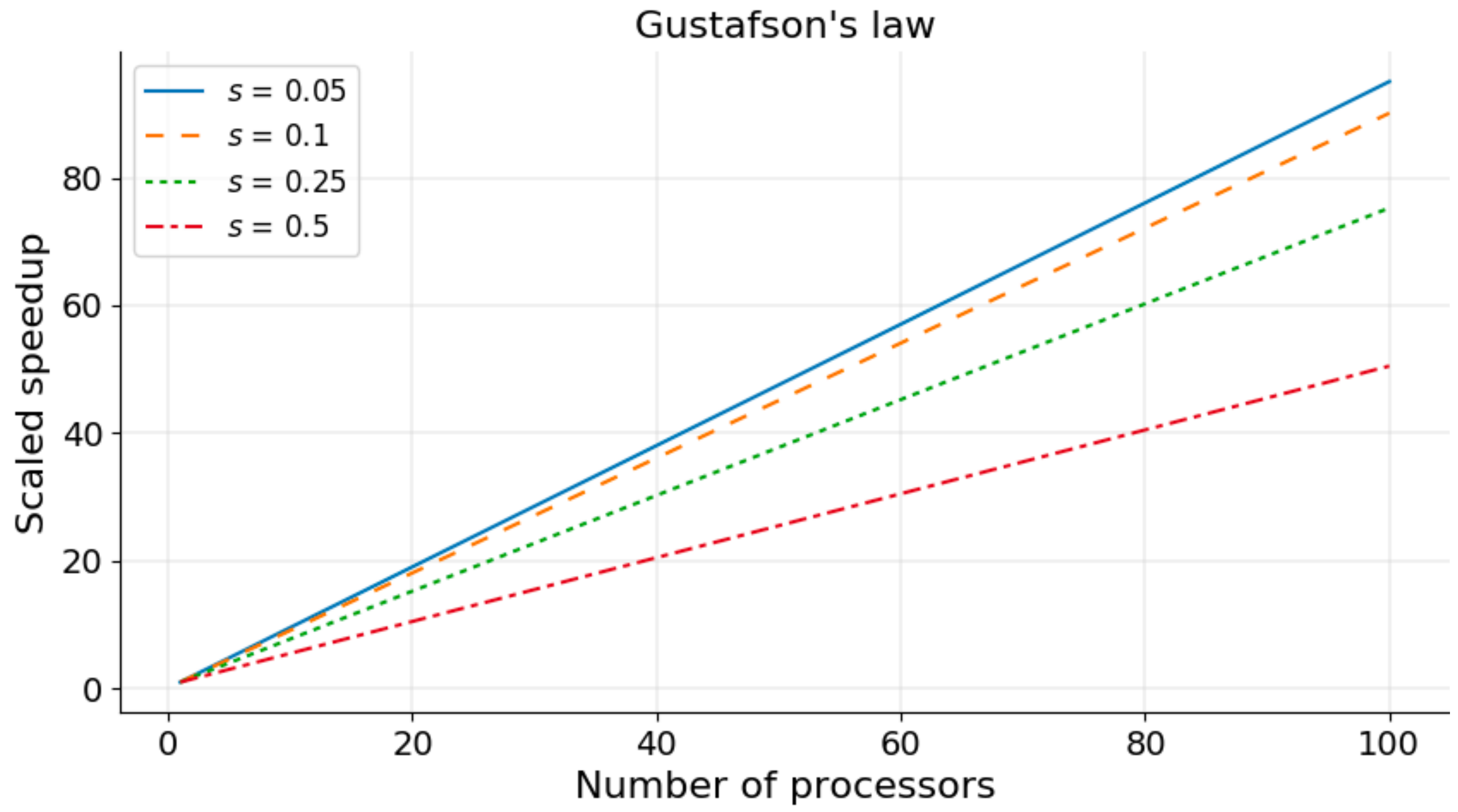
s = fraction of work done seqentially (Amdahl fraction)
1-s = parallelizable fraction
P = number of processors

Speedup (P) = Time(1) / Time(P)
<= 1/(s + (1-s)/P)
<= 1/s

For a fixed problem, the upper limit of speedup is **determined by the serial fraction.**

From Ramki Kannan

Gustafson's Law and Weak Scaling



- In practice, the **problem sizes often scale** with the amount of available resources.

- Gustafson's law - based on approximations that **the parallel part scales, but the serial part does not increase** wrt the size of the problem

  - Scaled speedup = s + (1-s) * P

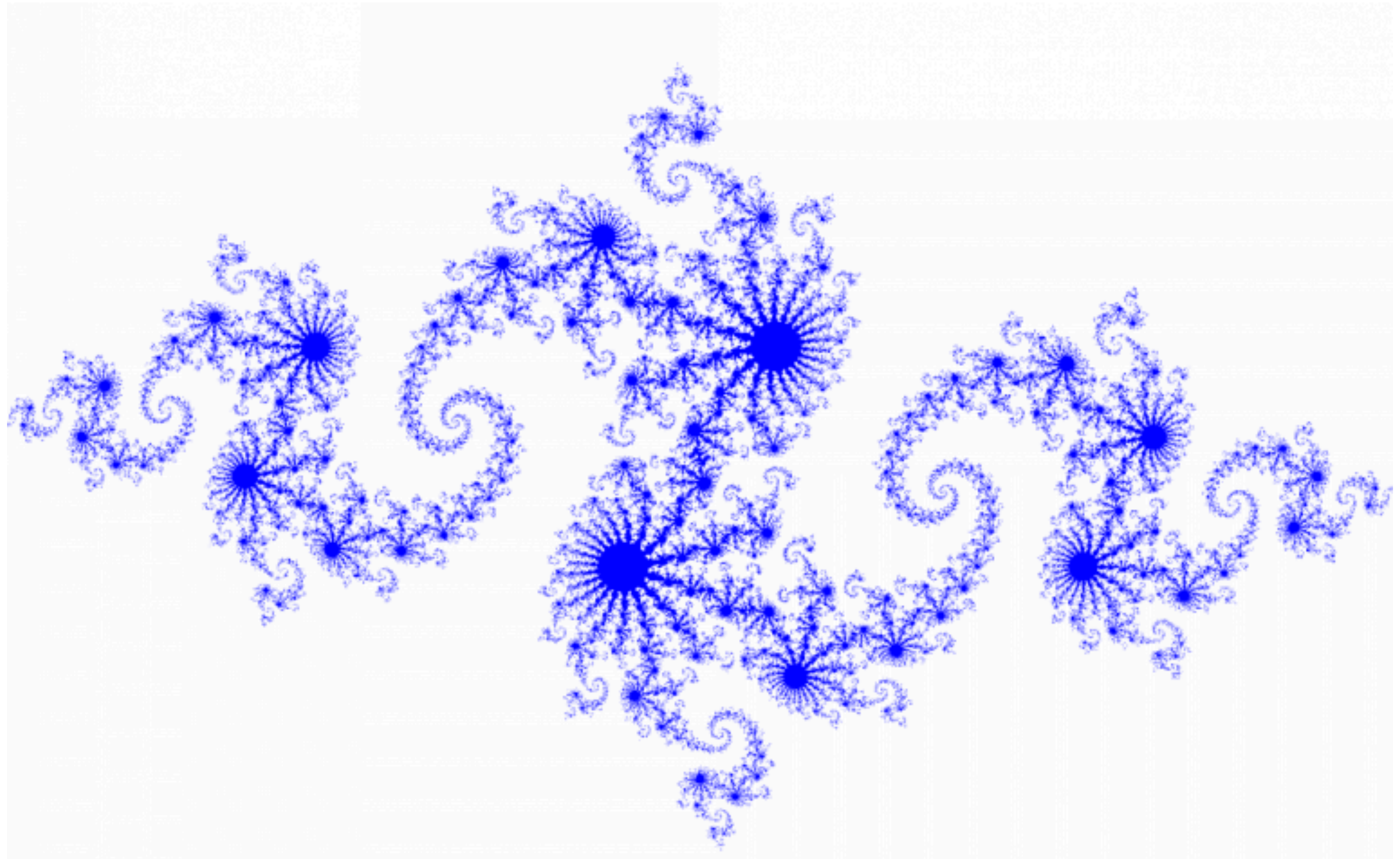
- Weak scaling: scaled speedup is calculated based on the amount of work done for a **scaled problem size**.

https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

Example: Julia Set

A Julia set is associated with a complex function and can be converted to an image by mapping each pixel onto the complex plane.

The total size of the image in this example is parametrized by integers h and w.



https://en.wikipedia.org/wiki/Julia_set

https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

Example: Julia Set Code

```
# pragma omp parallel for schedule(dynamic) \
  shared ( h, w, xl, xr, yb, yt ) \
  private ( i, j, k, juliaValue )
  for ( j = 0; j < h; j++ ) {
    for ( i = 0; i < w; i++) {
         // some O(1) calculation
    }
  }
```

https://people.sc.fsu.edu/~jburkardt/c_src/julia_set_openmp/julia_set_openmp.c

https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

Measuring Strong Scaling

Strong scaling is measured by **varying the number of threads** while **keeping the problem size** (in this case, the width and height) **constant**.

Table 1: Strong scaling for Julia set generator code

height	width	threads	time
10000	2000	1	3.932 sec
10000	2000	2	2.006 sec
10000	2000	4	1.088 sec
10000	2000	8	0.613 sec
10000	2000	12	0.441 sec
10000	2000	16	0.352 sec
10000	2000	24	0.262 sec

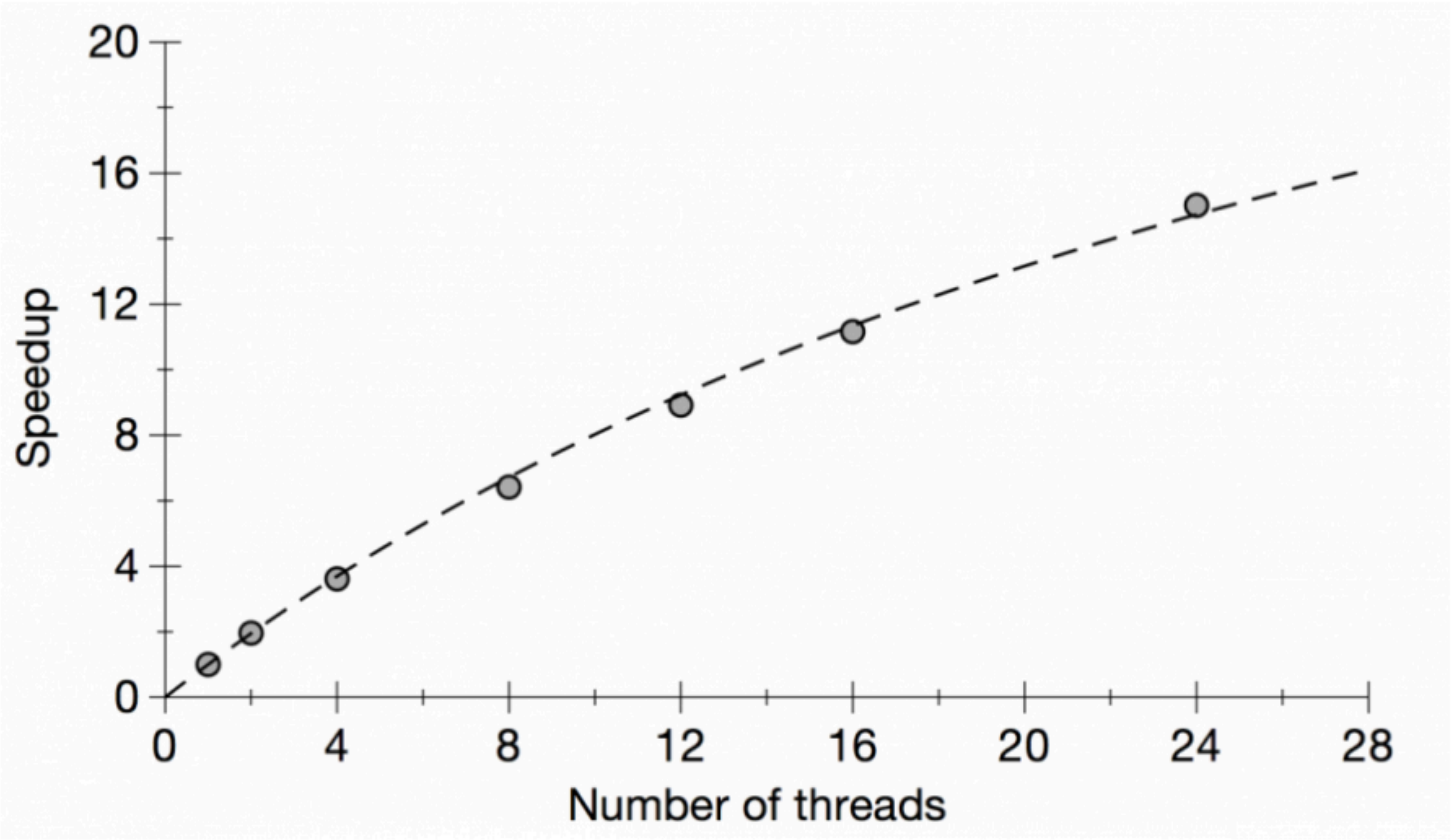


Figure 1: Plot of strong scaling for Julia set generator code
The dashed line shows the fitted curve based on Amdahl's law.

https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

Measuring Weak Scaling

Weak scaling is measured by **varying the number of threads and the problem size** proportionally with the thread count. In this example, the height is scaled and the width is kept constant.

Given p threads, scaled speedup is defined as efficiency * p, where efficiency is T_1/T_p.

Table 2: Weak scaling for Julia set generator code

height	width	threads	time
10000	2000	1	3.940 sec
20000	2000	2	3.874 sec
40000	2000	4	3.977 sec
80000	2000	8	4.258 sec
120000	2000	12	4.335 sec
160000	2000	16	4.324 sec
240000	2000	24	4.378 sec

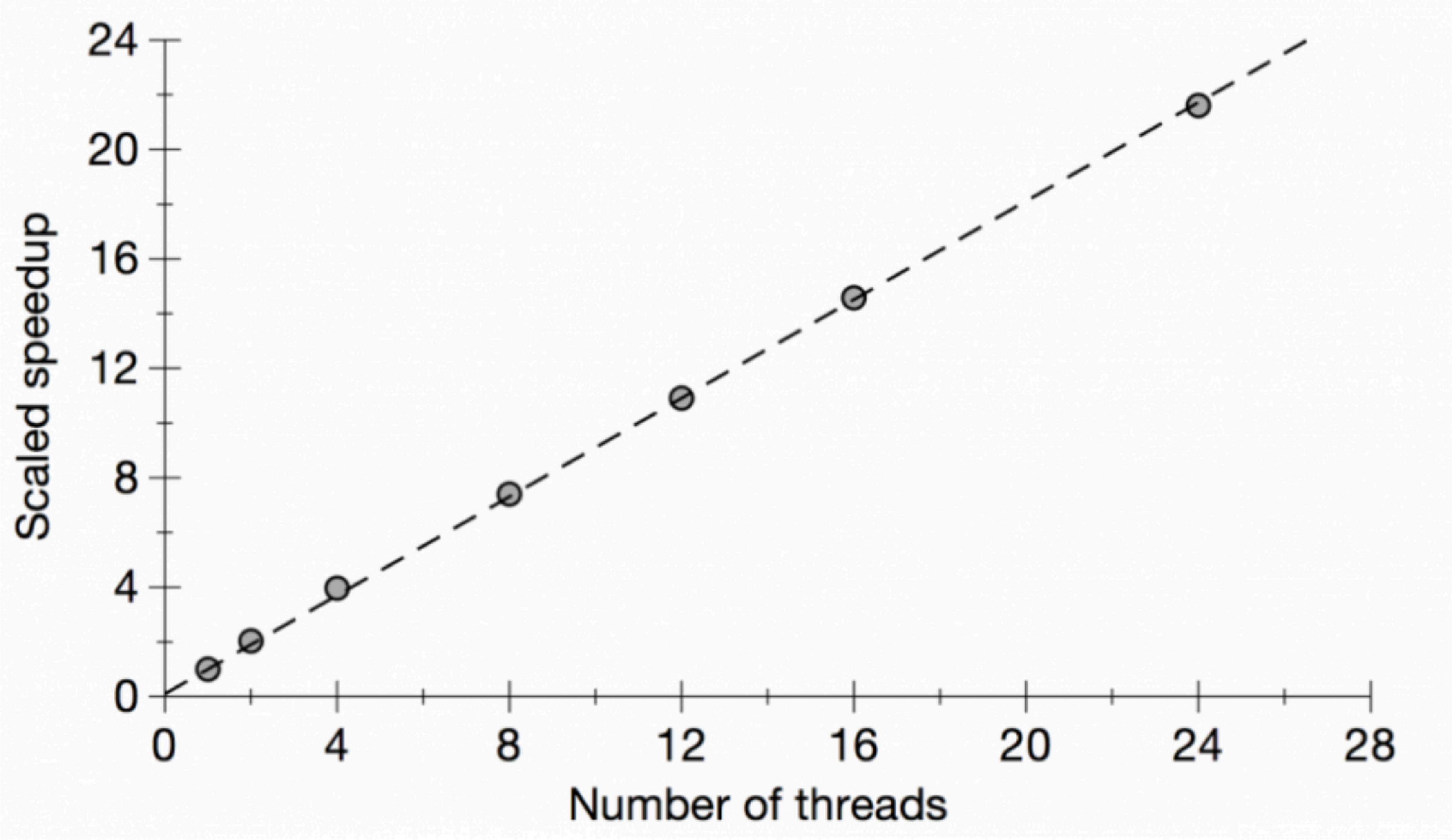


Figure 2: Plot of weak scaling for Julia set generator code
The dashed line shows the fitted curve based on Gustafson's law.

https://www.kth.se/blogs/pdc/2018/11/scalability-strong-and-weak-scaling/

Summary

- Processors today have multiple cores, and obtaining high performance requires **parallel programming**.
- Programming directly on processor cores is painful and error-prone.
- **OpenMP** abstracts processor cores, handles synchronization and communication protocols, and implements load-balancing methods.
- Scalability is important for efficient parallel computing.
- Homework 2: Particle simulation in OpenMP