System and Architectural Models

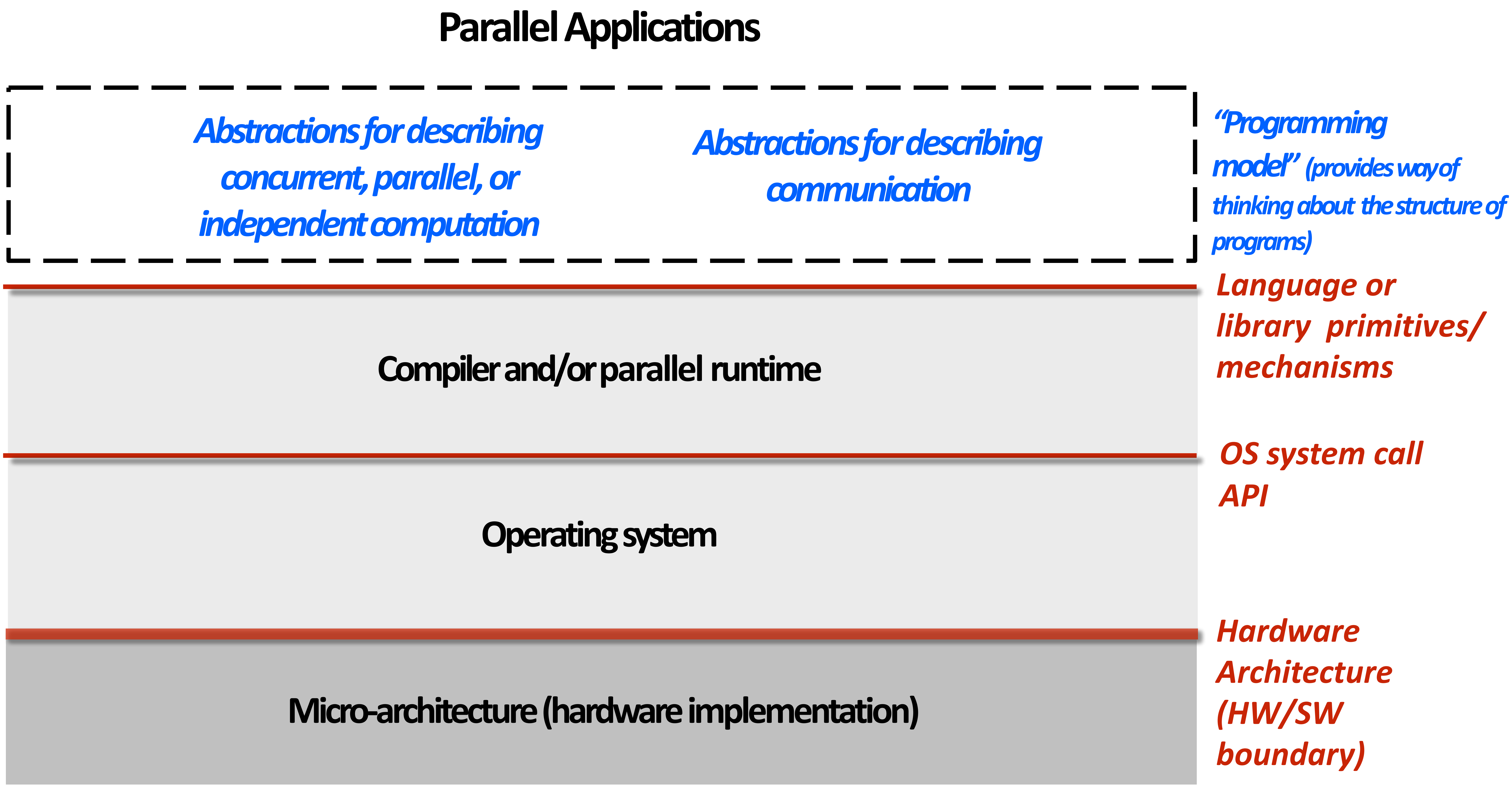
Architectural Models

- **Three parallel programming models**
  - That differ in what communication abstractions they present to the programmer
  - Programming models are important because they (1) influence how programmers think when writing programs and (2) influence the design of parallel hardware platforms designed to execute them

- **Corresponding machine architectures**
  - Abstraction presented by the hardware to low-level software

- **We'll focus on differences in communication/synchronization**

System layers: interface, implementation, interface, ...



Parallel Applications

Abstractions for describing concurrent, parallel, or independent computation

Abstractions for describing communication

"Programming model" (provides way of thinking about the structure of programs)

Language or library primitives/ mechanisms

Compiler and/or parallel runtime

OS system call API

Operating system

Hardware Architecture (HW/SW boundary)

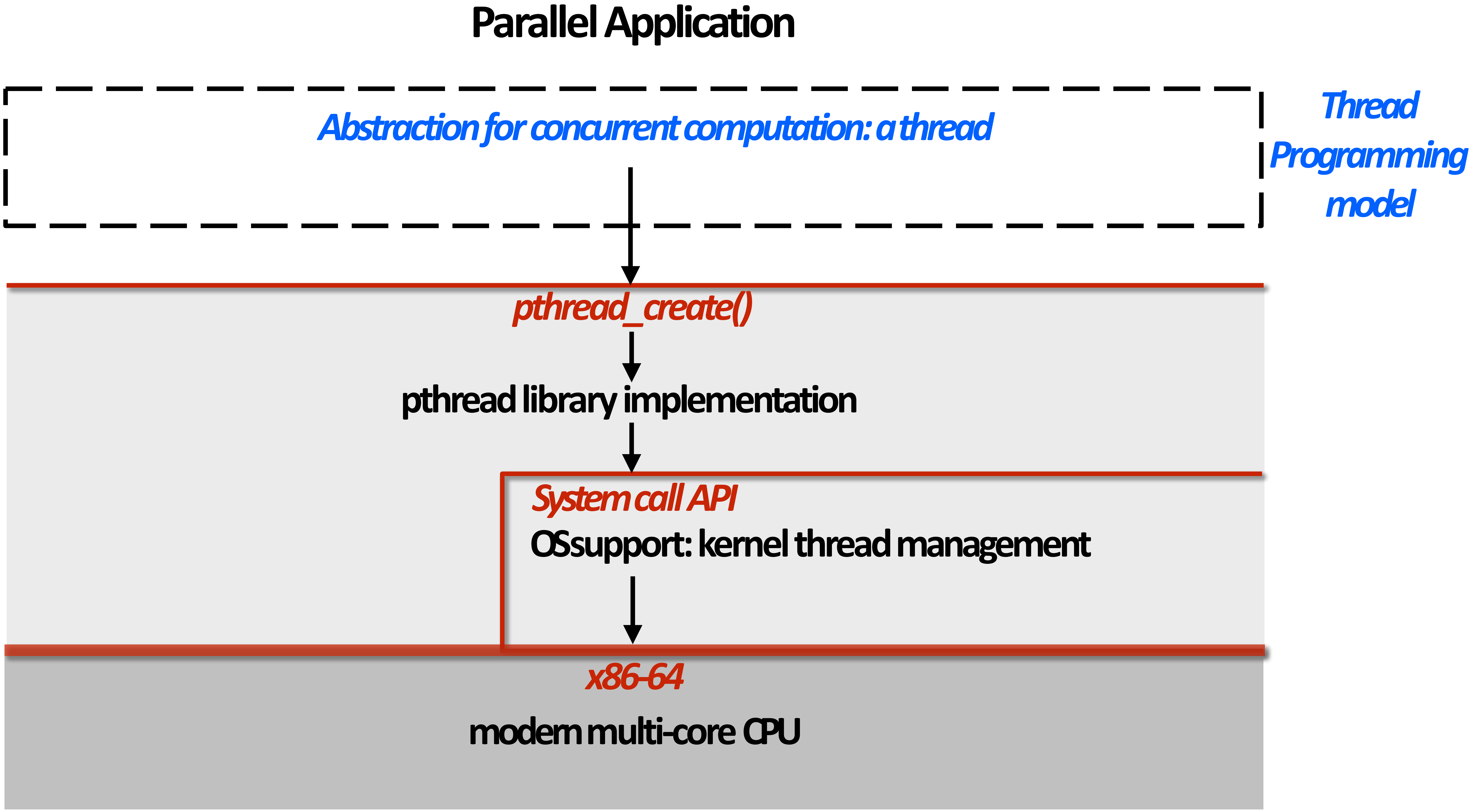
Micro-architecture (hardware implementation)

Blue italic text: abstraction/concept

Red italic text: system interface

Black text: system implementation

Example: expressing parallelism with pthreads



Parallel Application

Abstraction for concurrent computation: a thread

Thread Programming model

pthread_create()

pthread library implementation

System call API

OS support: kernel thread management

x86-64

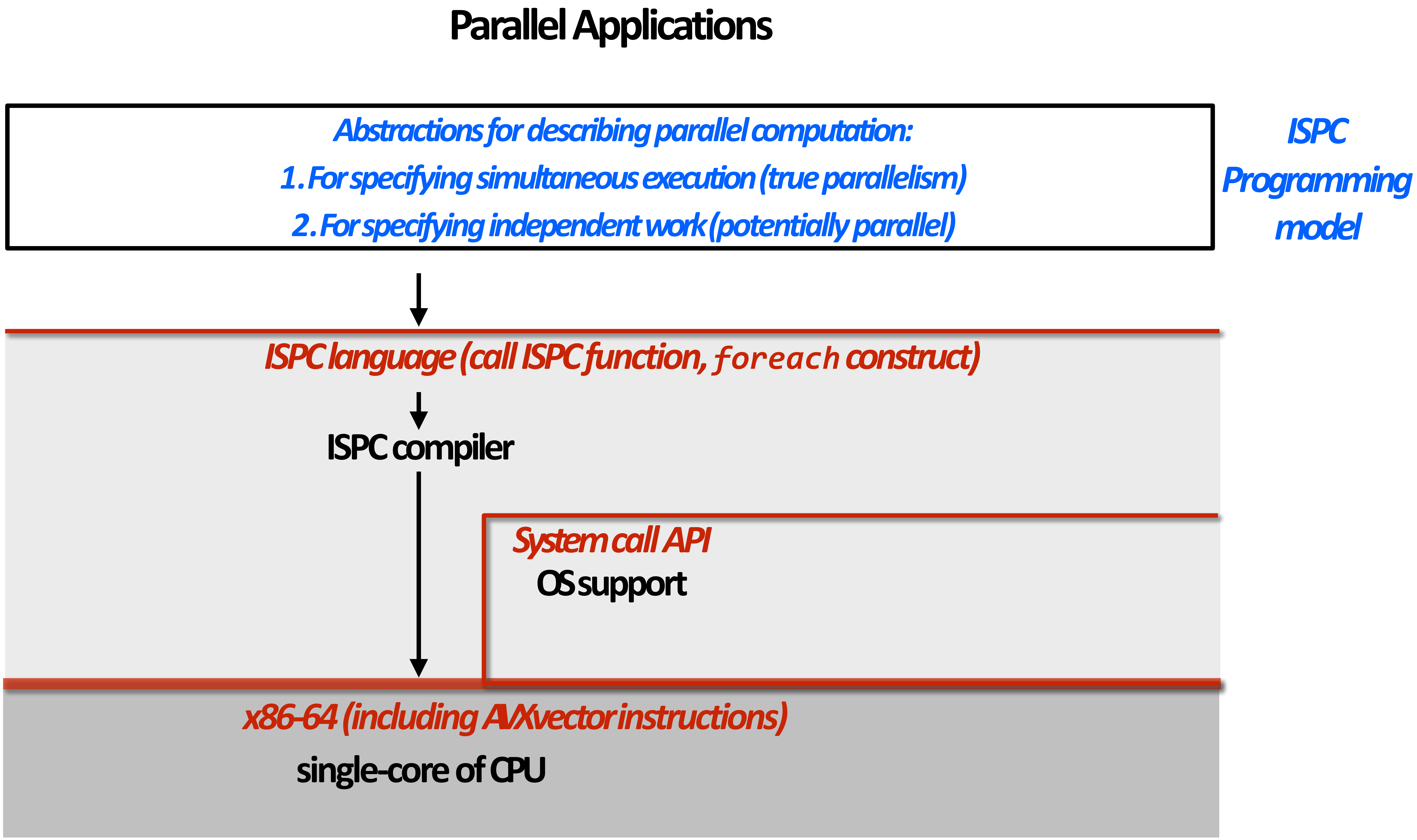
modern multi-core CPU

Blue italic text: abstraction/concept

Red italic text: system interface

Black text: system implementation

Example: expressing parallelism with ISPC



Parallel Applications

Abstractions for describing parallel computation:
1. For specifying simultaneous execution (true parallelism)
2. For specifying independent work (potentially parallel)

ISPC Programming model

ISPC language (call ISPC function, `foreach` construct)

ISPC compiler

System call API

OS support

x86-64 (including AVX vector instructions)

single-core of CPU

Note: This diagram is specific to the ISPC gang abstraction. ISPC also has the "task" language primitive for multi-core execution. I don't describe it here but it would be interesting to think about how that diagram would look

Parallel Programming Models

- Programming model is made up of the languages and libraries that create an abstract view of the machine
- Control
  - How is parallelism created?
  - What orderings exist between operations?
- Data
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- Synchronization
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?
- Cost
  - How do we account for the cost of each of the above?

Three programming models (abstractions)

1. Shared address space

2. Message passing

3. Data parallel

Shared address space model

What is memory?

- On the first day of class, we described a program as a sequence of instructions.

- Some of those instructions read and write from memory.

- But what is memory?
  - To be precise, what I’m really asking is: what is the logical abstraction of memory presented to a program

A program’s memory address space

- A computer’s memory is organized as a array of bytes

- Each byte is identified by its “address” in memory (its position in this array)

“The byte stored at address 0x8 has the value 32.”

“The byte stored at address 0x10 (16) has the value 128.”

In the illustration on the right, the program’s memory address space is 32 bytes in size
(so valid addresses range from 0x0 to 0x1F)

Address	Value
0x0	16
0x1	255
0x2	14
0x3	0
0x4	0
0x5	0
0x6	6
0x7	0
0x8	32
0x9	48
0xA	255
0xB	255
0xC	255
0xD	0
0xE	0
0xF	0
0x10	128
…	…
0x1F	0

Shared address space model (abstraction)

- Threads communicate by reading/writing to shared variables

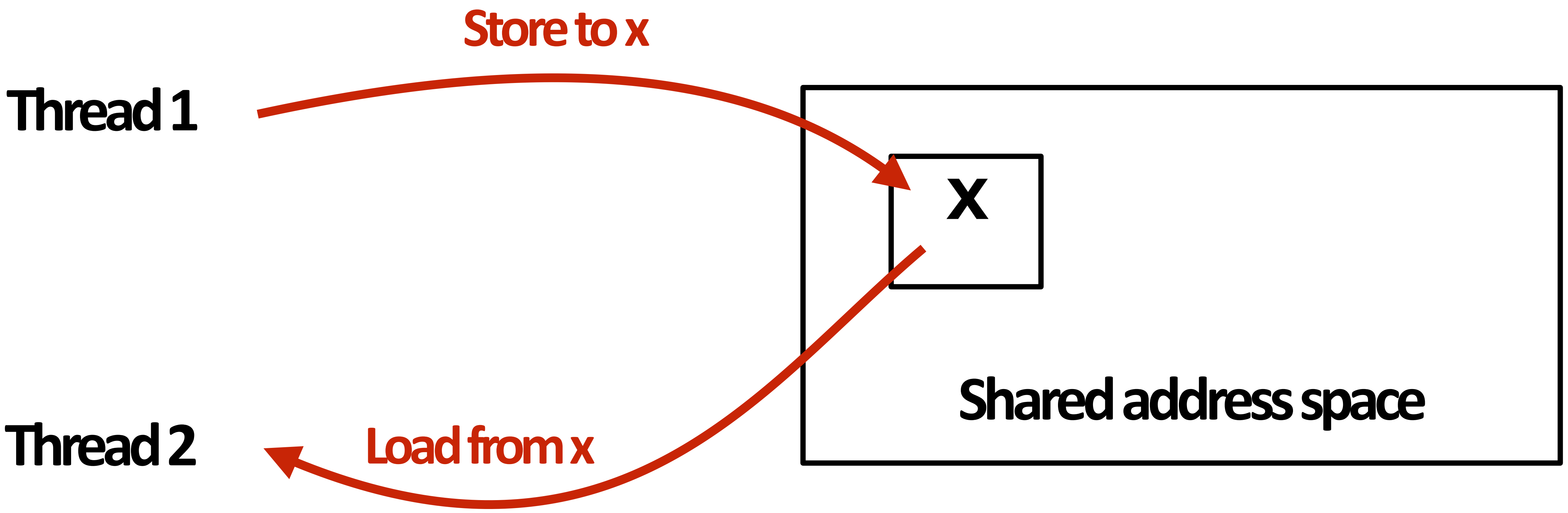
Thread 1:

```
int x = 0;
spawn_thread(foo, &x);

// write to address holding
// contents of variable x
x = 1;
```

Thread 2:

```
void foo(int* x) {
  // read from addr storing
  // contents of variable x
  while (x == 0) {}
  print x;
}
```



(Communication operations shown in red)

(Pseudocode provided in a fake C-like language for brevity.)

Shared address space model

Synchronization primitives are also shared variables: e.g., locks

Thread 1:

```
int x = 0;
Lock my_lock;

spawn_thread(foo, &x, &my_lock);


mylock.lock();
x++;
mylock.unlock();
```

Thread 2:

```
void foo(int* x, lock* my_lock)
{
  my_lock->lock();
  x++;
  my_lock->unlock();

  print x;
}
```

(Pseudocode provided in a fake C-like language for brevity.)

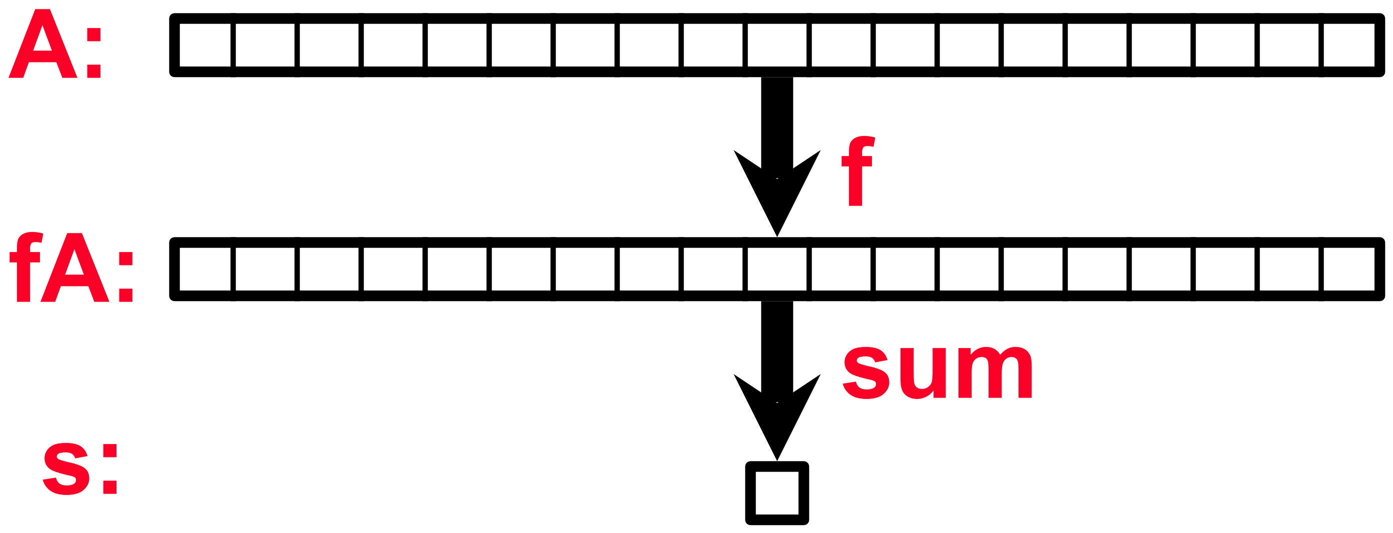
Simple Example

- Consider applying a function f to the elements of an array A and then computing its sum:

$$\sum_{i=0}^{n-1} f(A[i])$$

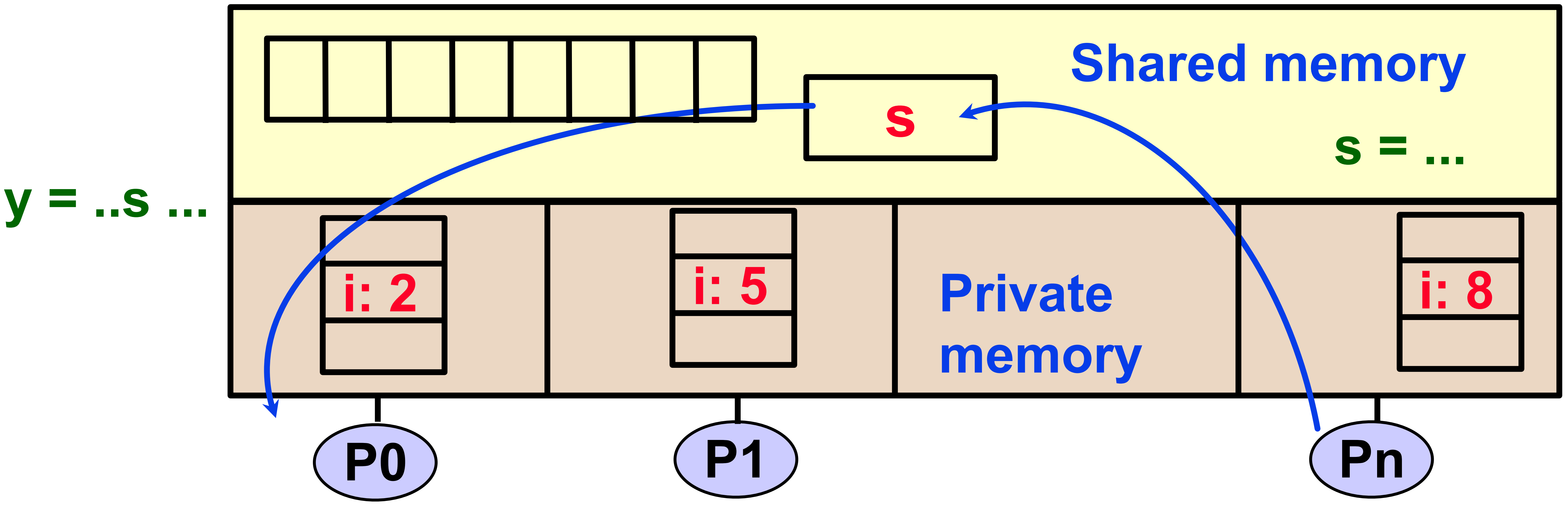
- Questions:
  - Where does A live? All in single memory? Partitioned?
  - What work will be done by each processors?
  - They need to coordinate to get a single result, how?

A = array of all data
fA = f(A)
s = sum(fA)



Programming Model 1: Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

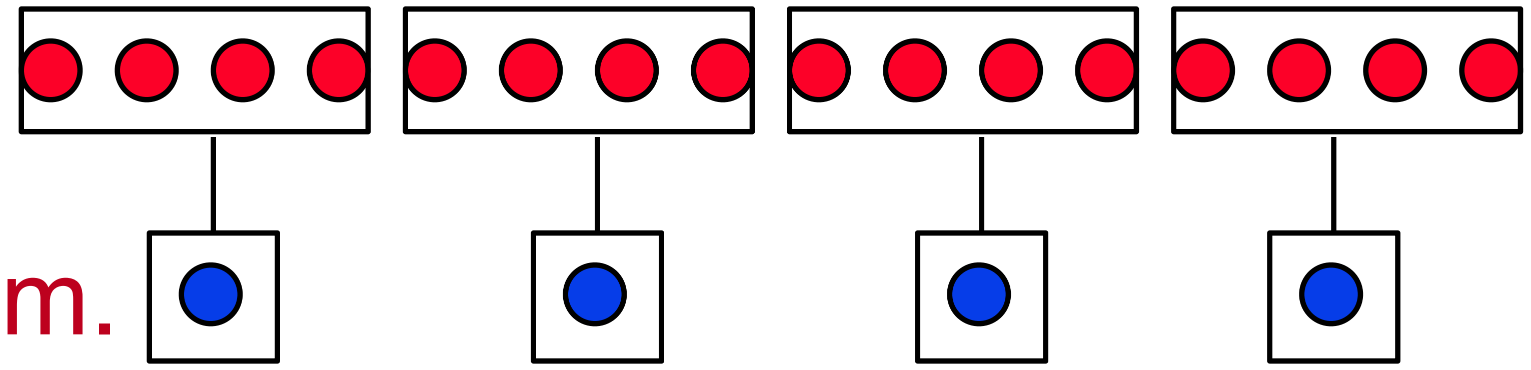


Simple Example

$$\sum_{i=0}^{n-1} f(A[i])$$

- Shared memory strategy:
  - small number p << n=size(A) processors
  - attached to single memory
- Parallel Decomposition:
  - Each evaluation and each partial sum is a task.
- Assign n/p numbers to each of p procs
  - Each computes independent “private” results and partial sum.
  - Collect the p partial sums and compute a global sum.

Two Classes of Data:



- Logically Shared
  - The original n numbers, the global sum.
- Logically Private
  - The individual function evaluations.
  - What about the individual partial sums?

Shared Memory “Code” for Computing a Sum

```
fork(sum,a[0:n/2-1]);
sum(a[n/2,n-1]);
```

```
static int s = 0;
```

```
Thread 1

  for i = 0, n/2-1
      s = s + f(A[i])
```

```
Thread 2

  for i = n/2, n-1
      s = s + f(A[i])
```

- What is the problem with this program?

- A race condition or data race occurs when:
  - Two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Shared Memory “Code” for Computing a Sum

A=

3	5

$f(x) = x^2$

```
static int s = 0;
```

```
Thread 1
 ....
  compute f([A[i]) and put in reg0     9
  reg1 = s                             0
  reg1 = reg1 + reg0                   9
  s = reg1                             9
 ...
```

```
Thread 2
 ...
  compute f([A[i]) and put in reg0   25
   reg1 = s                          0
   reg1 = reg1 + reg0                25
   s = reg1                          25
  ...
```

- Assume A = [3,5], f(x) = x^2, and s=0 initially
- For this program to work, s should be $3^2 + 5^2$ = 34 at the end
  - but it may be 34,9, or 25
- The *atomic* operations are reads and writes
  - Never see ½ of one number, but += operation is *not* atomic
  - All computations happen in (private) registers

Improved Code for Computing a Sum

```
static int s = 0;
static lock lk;
```

Why not do lock
Inside loop?

```
Thread 1

   local_s1= 0
   for i = 0, n/2-1
      local_s1 = local_s1 + f(A[i])
   lock(lk);
   s = s + local_s1
   unlock(lk);
```

```
Thread 2

   local_s2 = 0
   for i = n/2, n-1
      local_s2= local_s2 + f(A[i])
   lock(lk);
   s = s +local_s2
   unlock(lk);
```

- Since addition is associative, it’s OK to rearrange order
- Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
  - The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)

Mechanisms for preserving atomicity

- Lock/unlock mutex around a critical section

```
LOCK(mylock);
// critical section
UNLOCK(mylock);
```

- Some languages have first-class support for atomicity of code blocks

```
atomic {
  // critical section
}
```

- Intrinsics for hardware-supported atomic read-modify-write operations

```
atomicAdd(x, 10);
```

Review: shared address space model

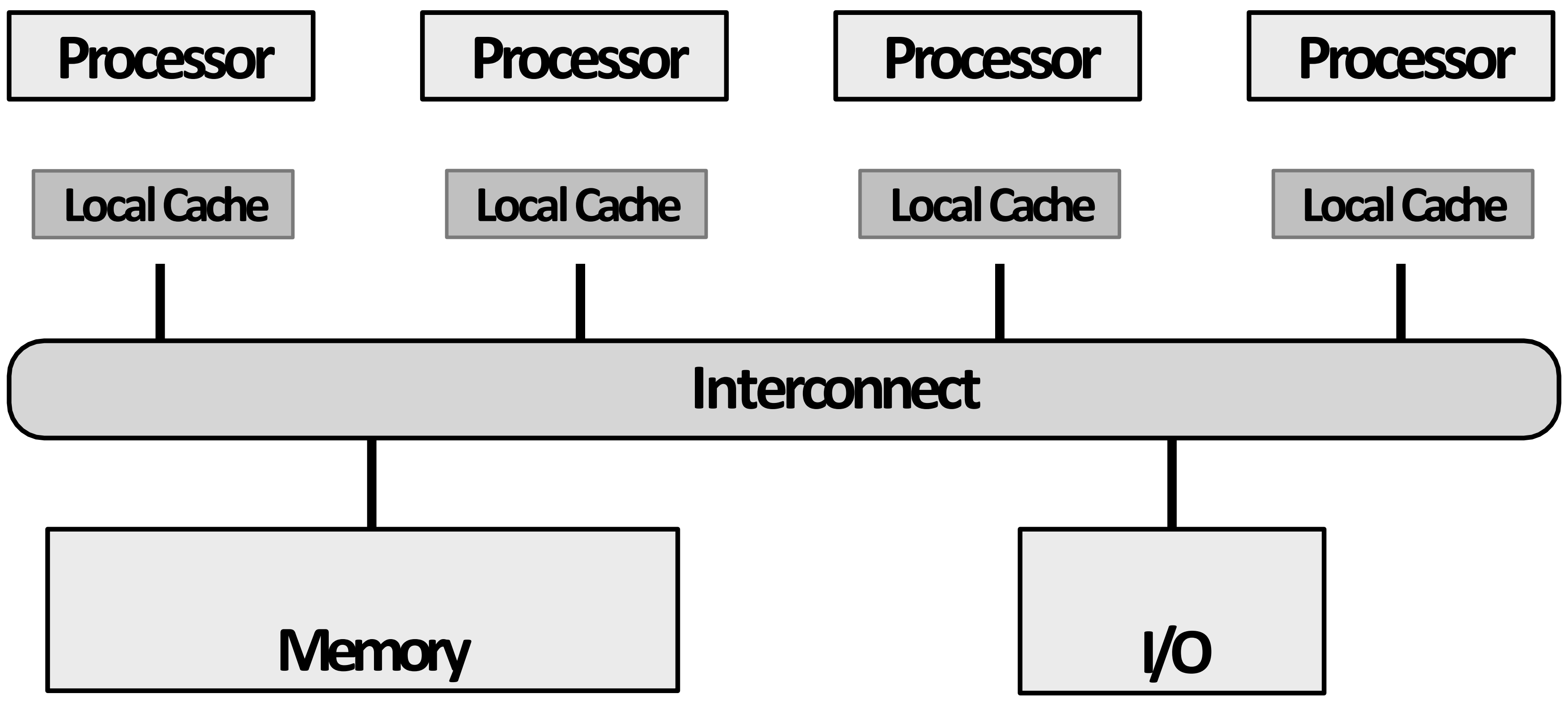
- Threads communicate by:
  - Reading/writing to shared variables in a shared address space
    - Inter-thread communication is implicit in memory loads/stores
    - Thread 1 stores to X
    - Later, thread 2 reads X(and observes update of value by thread 1)
  - Manipulating synchronization primitives
    - e.g., ensuring mutual exclusion via use of locks

- This is a natural extension of sequential programming
  - In fact, all our discussions in class have assumed a shared address space so far!

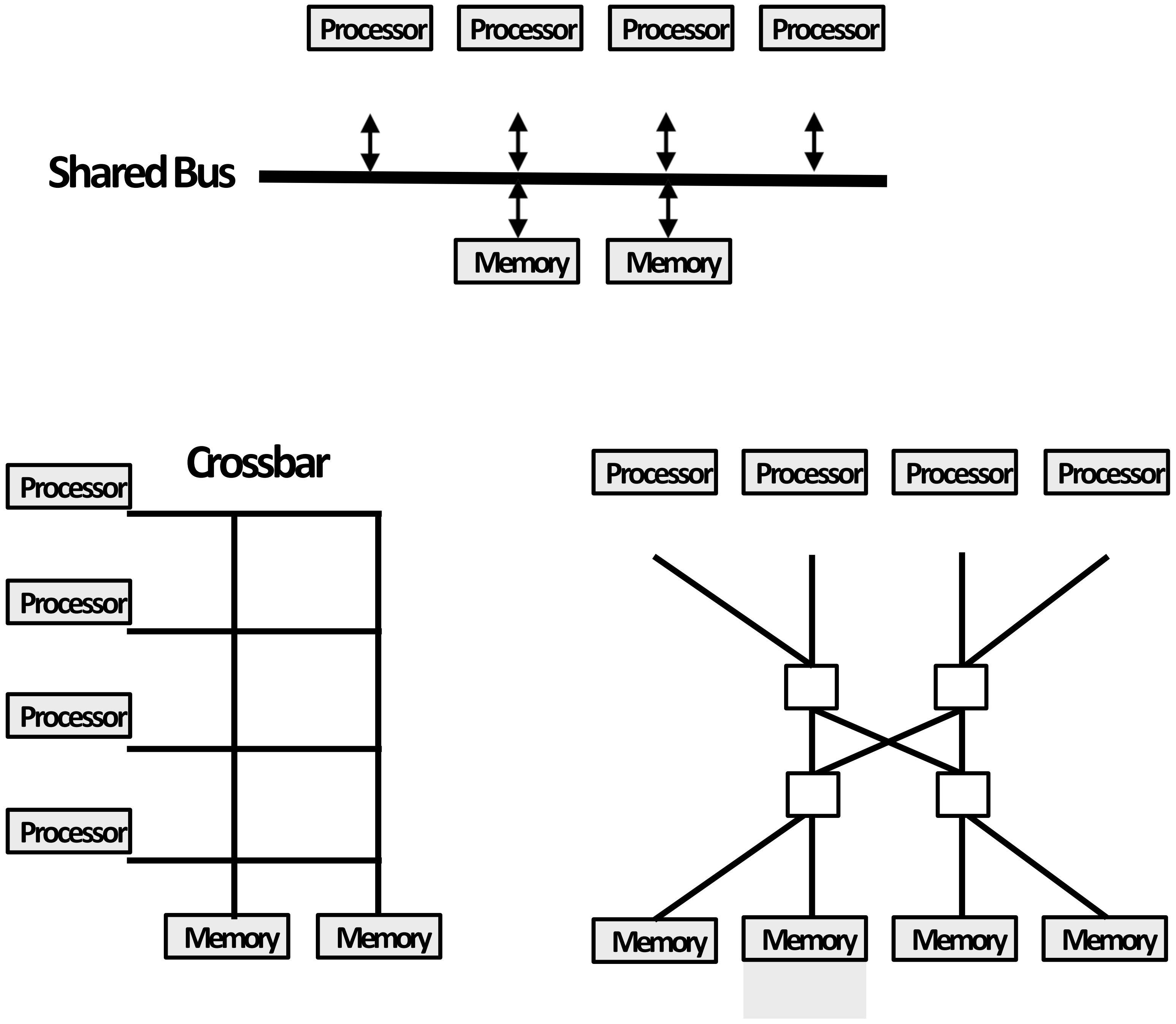
HW implementation of a shared address space

Key idea: any processor can <u>directly</u> reference contents of any memory location

"Dance-hall" organization



Interconnect examples



Multi-stage network

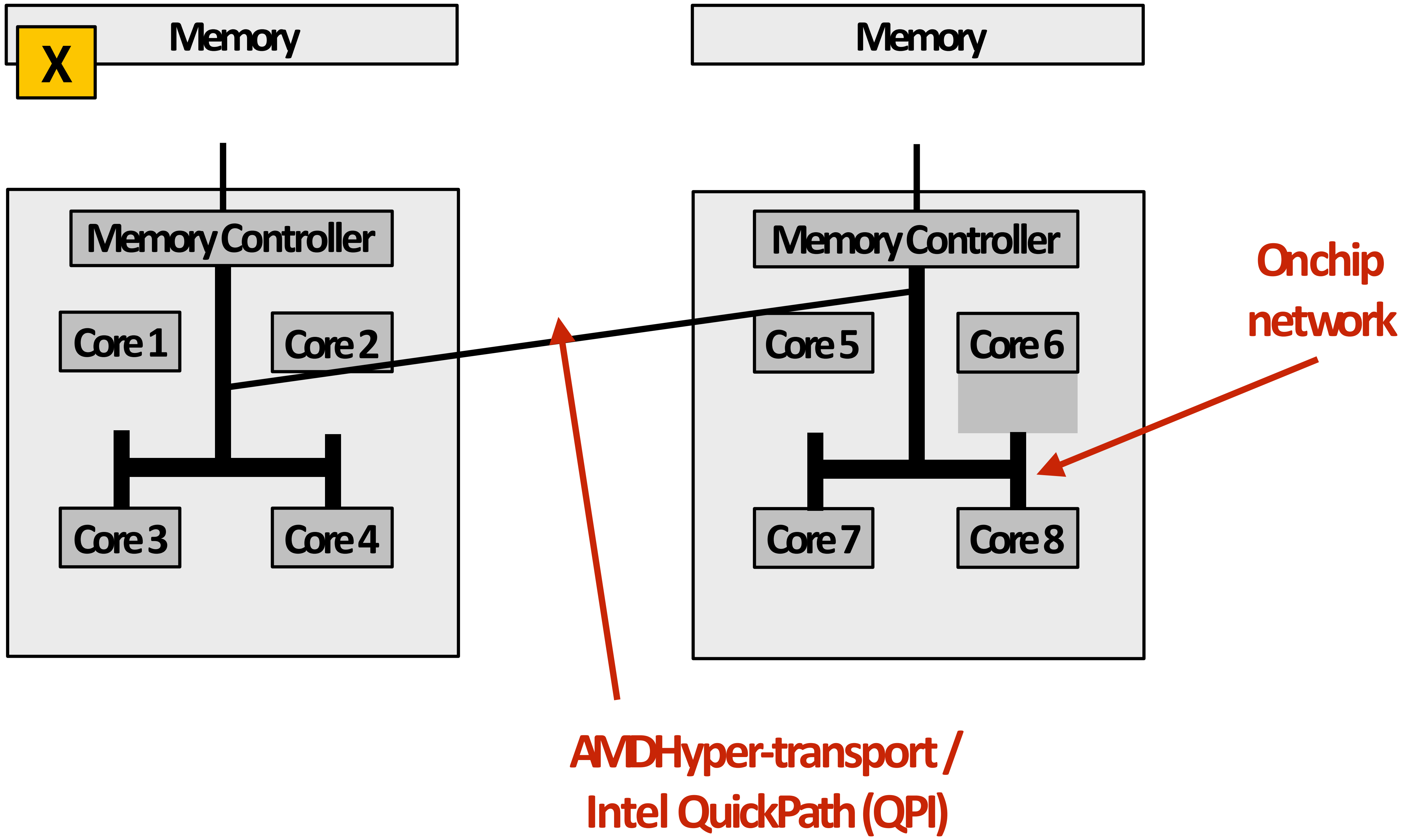
*Caches (not shown) are another implementation of a shared address space (more on this in a later lecture)

Non-uniform memory access (NUMA)

The latency * of accessing a memory location may be different from different processing cores in the system

Example: latency to access address x is higher from cores 5-8 than cores 1-4

Example: modern dual-socket configuration



* Bandwidth from any one location may also be different to different CPU cores

Summary: shared address space model

- **Communication abstraction**
  - Threads read/write variables in shared address space
  - Threads manipulate synchronization primitives: locks, atomic ops, etc.
  - Logical extension of uniprocessor programming *

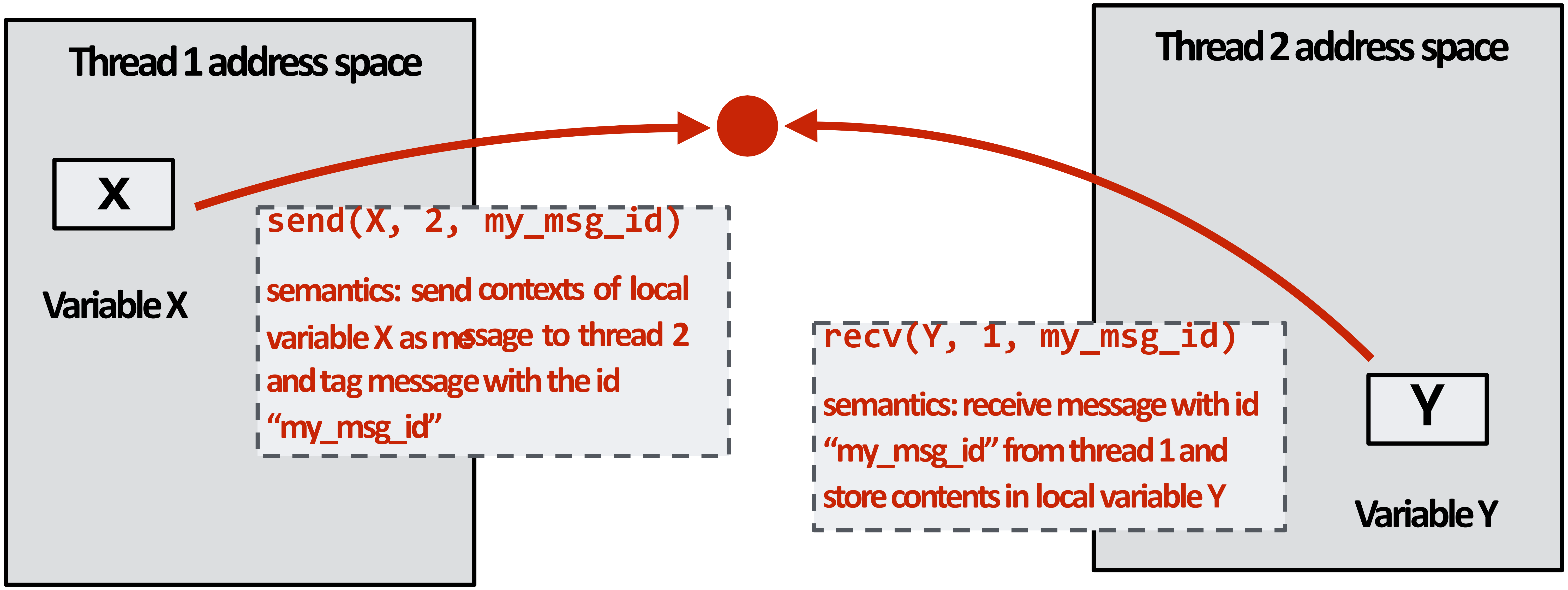
- **Requires hardware support to implement efficiently**
  - Any processor can load and store from any address (its shared address space!)
  - Can be costly to scale to large numbers of processors (one of the reasons why high-core count processors are expensive)

* But NUMA implementation requires reasoning about locality for performance

Message passing model of communication

Message passing model (abstraction)

- Threads operate within their own private address spaces
- Threads communicate by sending/receiving messages
  - <u>send</u>: specifies recipient, buffer to be transmitted, and optional message identifier ("tag")
  - <u>receive</u>: sender, specifies buffer to store data, and optional message identifier
  - Sending messages is <u>the only way</u> to exchange data between threads 1 and 2
    - Why?



```
send(X, 2, my_msg_id)
```

semantics: send contexts of local variable X as message to thread 2 and tag message with the id "my_msg_id"

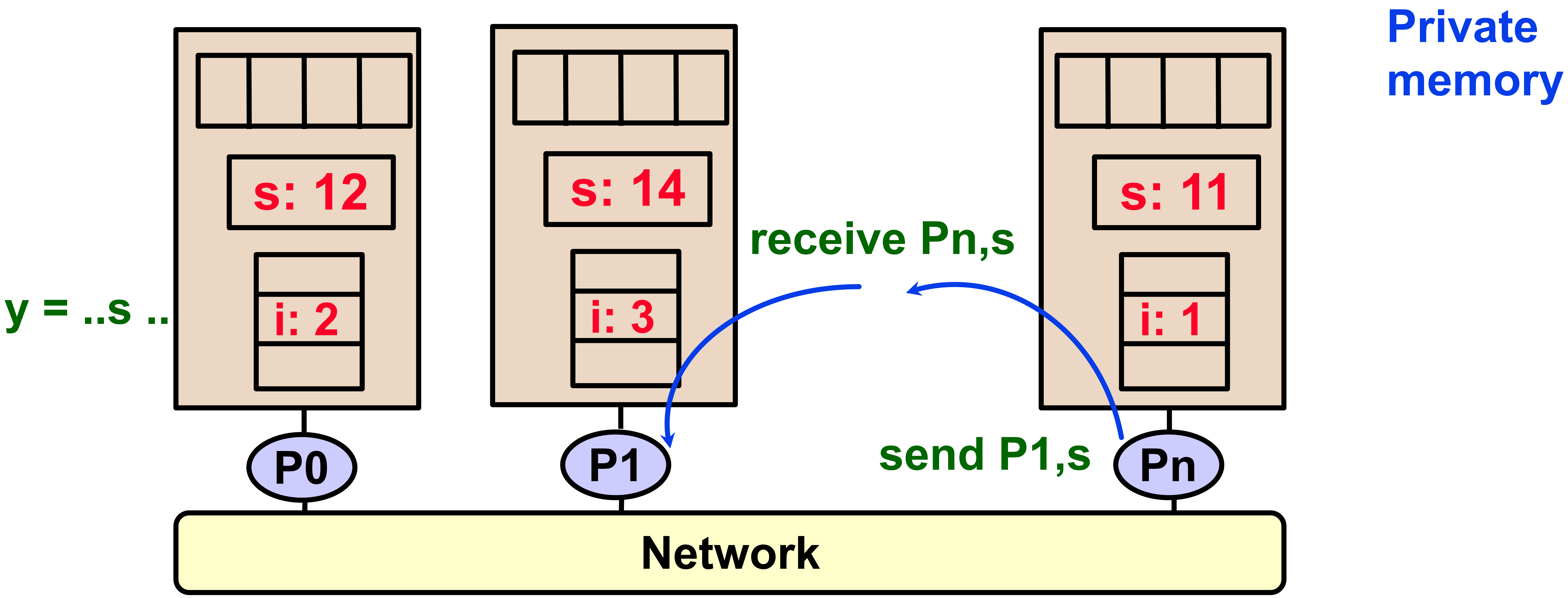
```
recv(Y, 1, my_msg_id)
```

semantics: receive message with id "my_msg_id" from thread 1 and store contents in local variable Y

(Communication operations shown in red)

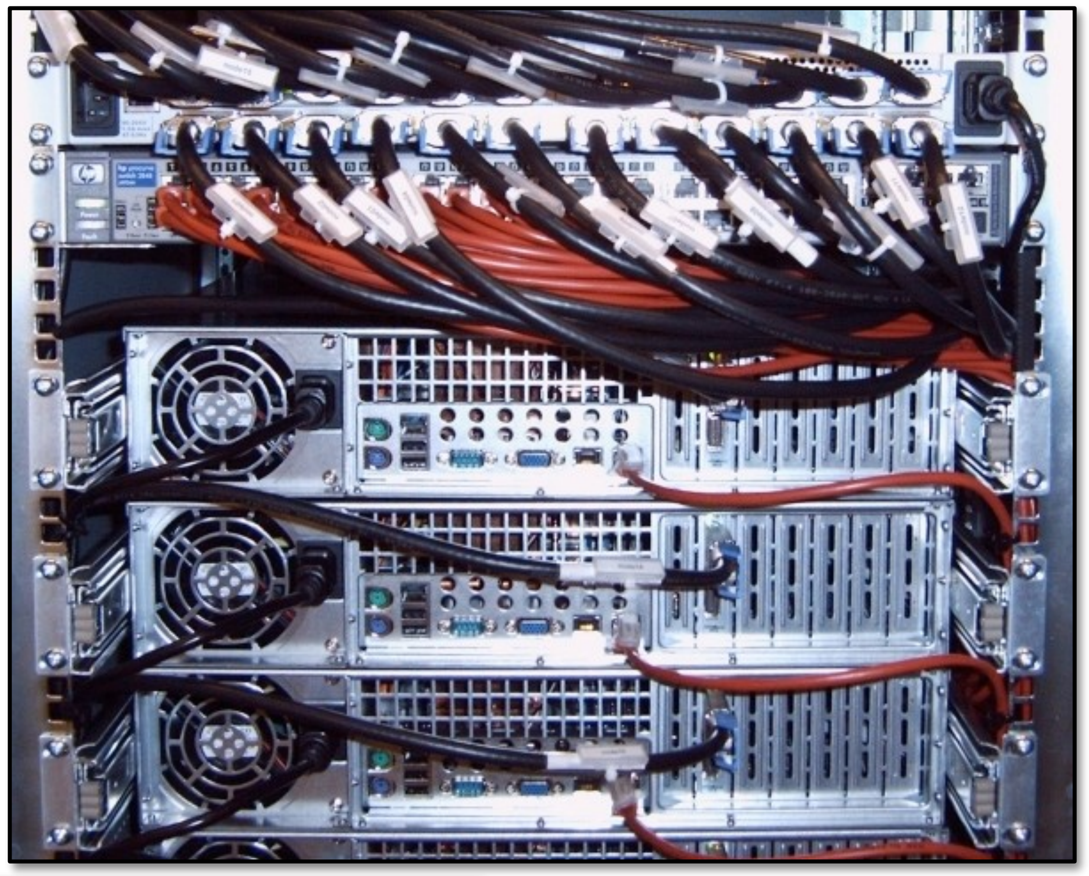
Message Passing

- Program consists of a collection of named processes.
  - **Usually fixed at program startup time**
  - **Thread of control plus local address space -- NO shared data.**
  - **Logically shared data is partitioned over local processes.**
- Processes communicate by explicit send/receive pairs
  - **Coordination is implicit in every communication event.**
  - **MPI (Message Passing Interface) is the most commonly used SW**

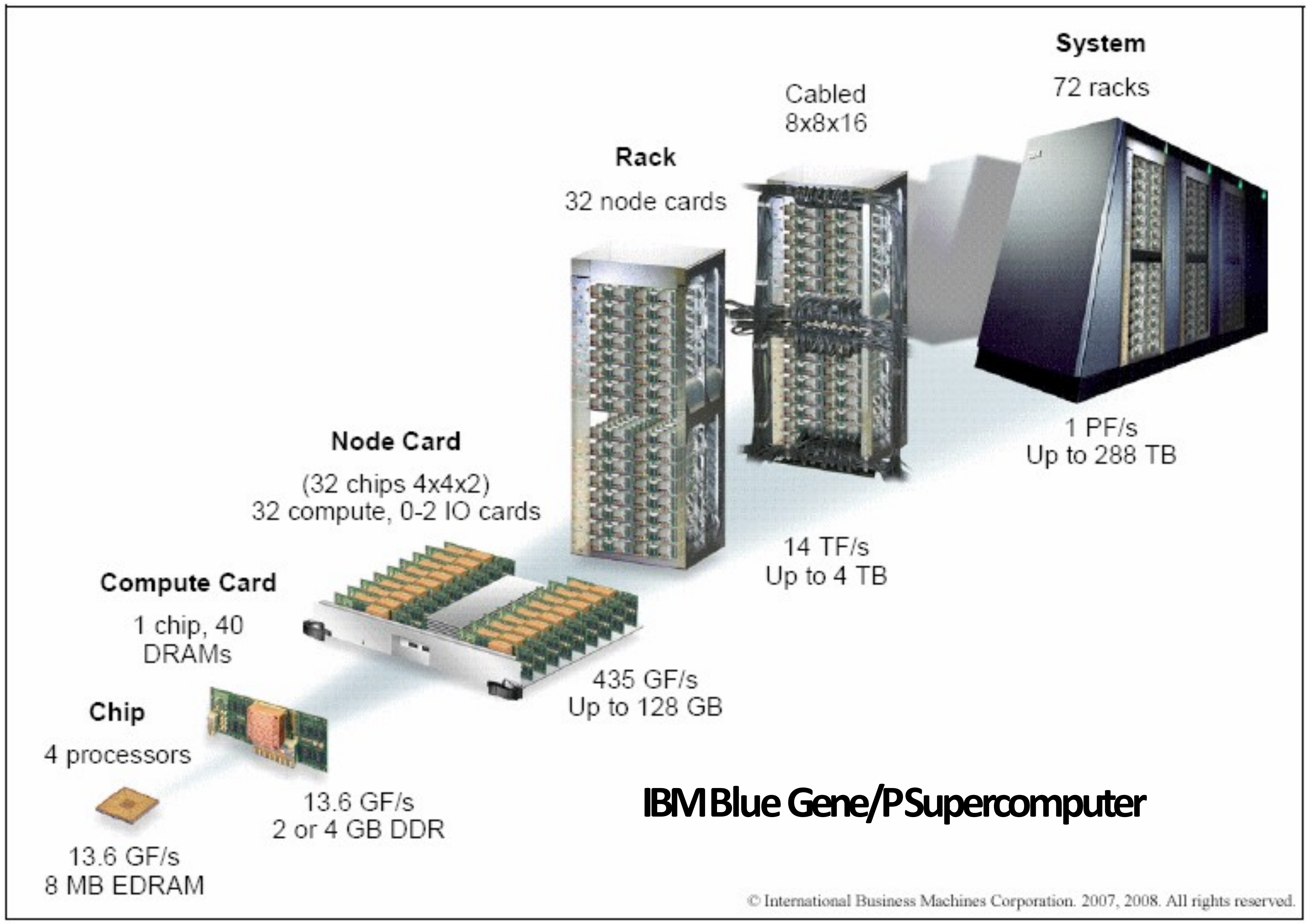


Message passing (implementation)

- **Hardware need not implement system-wide loads and stores to execute message passing programs (to need only communicate messages between nodes)**
  - **Can connect commodity systems together to form large parallel machine (message passing is a programming model for clusters and supercomputers)**



Cluster of workstations (Infiniband network)



IBM Blue Gene/P Supercomputer

Image credit: IBM

Programming model vs. implementation of communication

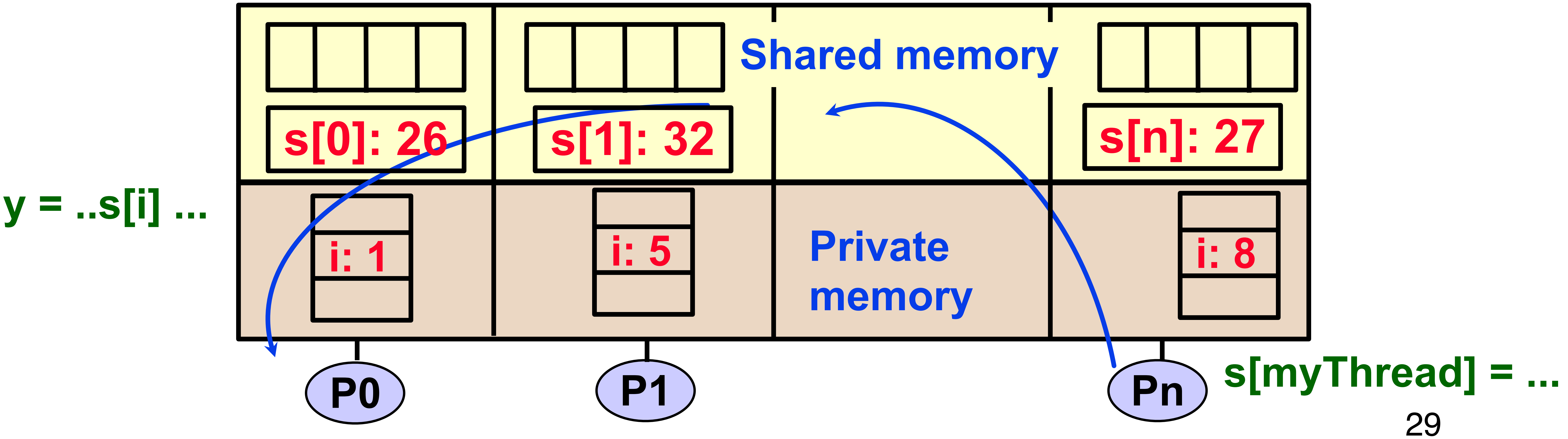
- **Common to implement message passing <u>abstractions</u> on machines that implement a shared address space in hardware**
  - **"Sending message" = copying memory from message library buffers**
  - **"Receiving message" = copy data from message library buffers**

- **Can implement shared address space abstraction on machines that do not support it in HW (via less efficient SW implementations)**
  - **OS marks all pages with shared variables as invalid**
  - **OS page-fault handler issues appropriate network requests**

- **Keep clear in your mind: what is the programming model (abstractions used to specify program)? And what is the HW implementation?**

Programming Model 2a: Global Address Space

- Program consists of a collection of named threads.
  - **Usually fixed at program startup time**
  - **Local and shared data, as in shared memory model**
  - **But, shared data is partitioned over local processes**
  - **Cost models says remote data is expensive**
- Examples: UPC, Titanium, Co-Array Fortran
- Global Address Space programming is an intermediate point between message passing and shared memory



The data-parallel model

Programming models provide a way to think about the organization of parallel programs

- **Shared address space: very little structure to communication**
  - All threads can read and write to all shared variables
  - Challenge: due to implementation details: not all reads and writes are same cost (cost is often not apparent when reading source code!)

- **Message passing: structured communication in the form of messages**
  - All communication occurs in the form of messages (communication is explicit in source code—the sends and receives)

- **Data parallel: rigid structure to computation**
  - Perform same function on elements of large collections

Data-parallel model

- **Organize computation as operations on sequences of elements**
  - **e.g., perform same function on all elements of a sequence**

- **Historically: same operation on each element of vector**
  - **Matched capabilities SIMD supercomputers of 80’s**
  - **Connection Machine (CM-1, CM-2): thousands of processors, one instruction decode unit**
  - **Early Cray supercomputers were vector processors**
  - **`add(A, B, n)` ← this was one instruction on vectors A, B of length n**

Key data type: sequences

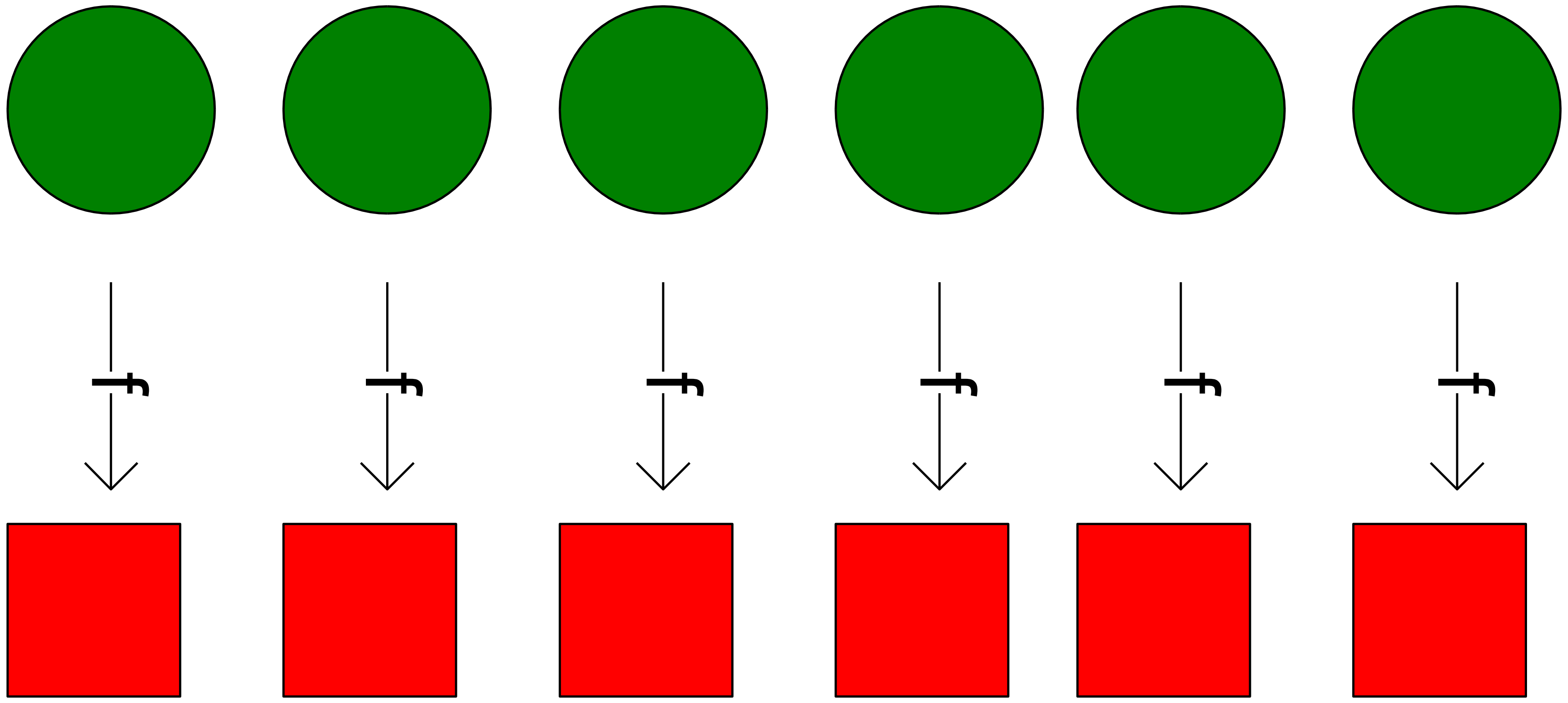
- Ordered collection of elements
- For example, in a C++ like language: Sequence<T>
- e.g., Scala lists: List[T]
- In a functional language (like Haskell): seq T

- Can only access elements of sequence through specific operations

Map

- Higher order function (function that takes a function as an argument)
- Applies side-effect free unary function `f :: a -> b` to all elements of input sequence, to produce output sequence of the same length
- In a functional language (e.g., Haskell)
  - `map :: (a -> b) -> seq a -> seq b`
- In C++: transform

```
template<class InputIt, class OutputIt, class UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation unary_op);
```



Parallelizing map

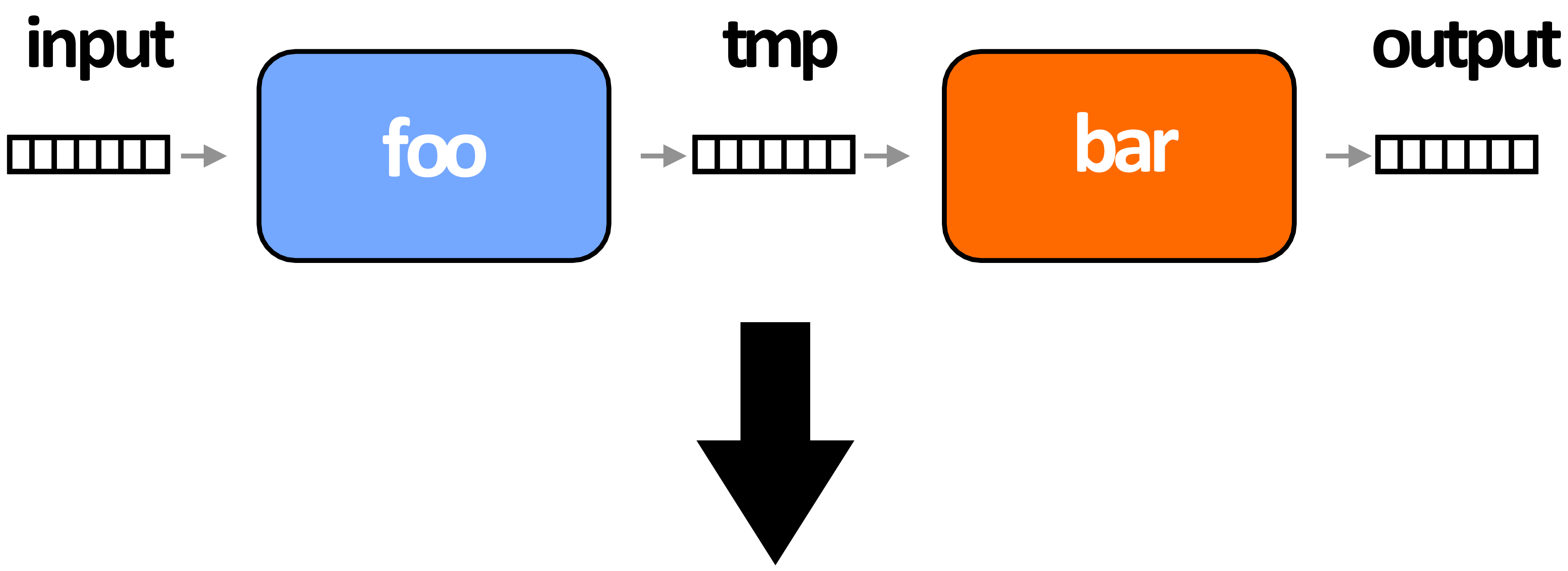
- Since `f :: a -> b` is a function (side-effect free), then applying `f` to all elements of the sequence can be done in any order without changing the output of the program

- The implementation of map has flexibility to reorder/ parallelize processing of elements of sequence however it sees fit

Optimizing data movement in map

```
const int N = 1024;
Sequence<float> input(N);
Sequence<float> tmp(N);
Sequence<float> output(N);

map(foo, input, tmp);
map(bar, tmp, output);
```



```
parallel_for(int i=0; i<N; i++)
{
   output[i] = bar(foo(input[i]));
}
```

- Consider code that performs two back-to-back maps (like that to left)
- Optimizing compiler or runtime can reorganize code (bottom-left) to eliminate memory loads and stores ("map fusion")
- Additional optimizations: highly optimized implementations of map can also perform optimizations like prefetching next element of input sequence (to hide memory latency)
- Why are these complex optimizations possible?

Data parallelism in ISPC

```
// main C++ code:
const int N = 1024;
float* x = new float[N];
float* y = new float[N];

// initialize N elements of x here

absolute_value(N, x, y);
```

```
// ISPC code:
export void absolute_value(
   uniform int N,
   uniform float* x,
   uniform float* y)
{
   foreach (i = 0 ... N)
   {
       if (x[i] < 0)
          y[i] = -x[i];
       else
          y[i] = x[i];
   }
}
```

`foreach` construct

Think of loop body as a function

Given this program, it is reasonable to think of the program as using `foreach` to "map the loop body onto each element" of the arrays X and Y.

But if we want to be more precise: a sequence is not a first-class ISPC concept. It is implicitly defined by how the program has implemented array indexing logic in the `foreach` loop.

(There is no operation in ISPC with the semantic: "map this code over all elements of this sequence")

Data parallelism in ISPC

```
// main C++ code:
const int N = 1024;
float* x = new float[N/2];
float* y = new float[N];
// initialize N/2 elements of x here

absolute_repeat(N/2, x, y);
```

Think of loop body as a function

The input/output sequences being mapped over are implicitly defined by array indexing logic

```
// ISPC code:
export void absolute_repeat(
   uniform int N,
   uniform float* x,
   uniform float* y)
{
   foreach (i = 0 ... N)
   {
       if (x[i] < 0)
          y[2*i] = -x[i];
       else
          y[2*i] = x[i];
       y[2*i+1] = y[2*i];
   }
}
```

This is also a valid ISPC program!

It takes the absolute value of elements of x, then repeats it twice in the output array y

(Less obvious how to think of this code as mapping the loop body onto existing sequences.)

Data parallelism in ISPC

```
// main C++ code:
const int N = 1024;
float* x = new float[N];
float* y = new float[N];
// initialize N elements of x

shift_negative(N, x, y);
```

Think of loop body as a function

The input/output sequences being mapped over are implicitly defined by array indexing logic

```
// ISPC code:
export void shift_negative(
   uniform int N,
   uniform float* x,
   uniform float* y)
{
   foreach (i = 0 ... N)
   {
       if (i >= 1 && x[i] < 0)
         y[i-1] = x[i];
       else
         y[i] = x[i];
   }
}
```

The output of this program is undefined!

Possible for multiple iterations of the loop body to write to same memory location

Data-parallel model (foreach) provides no specification of order in which iterations occur

But model provides no primitives for fine-grained mutual exclusion/synchronization). It is not intended to help programmers write programs with that structure

Gather/scatter: two key data-parallel communication primitives

Map absolute_value onto stream produced by gather:

```
const int N = 1024;
Sequence<float> input(N);
Sequence<int> indices;
Sequence<float> tmp_input(N);
Sequence<float> output(N);

stream_gather(input, indices, tmp_input);
absolute_value(tmp_input, output);
```

ISPC equivalent:

```
export void absolute_value(
   uniform float N,
   uniform float* input,
   uniform float* output,
   uniform int* indices)
{
   foreach (i = 0 ... n)
   {
      float tmp = input[indices[i]];
      if (tmp < 0)
         output[i] = -tmp;
      else
         output[i] = tmp;
   }
}
```

Map absolute_value onto stream, scatter results:

```
const int N = 1024;
Sequence<float> input(N);
Sequence<int> indices;
Sequence<float> tmp_output(N);
Sequence<float> output(N);

absolute_value(input, tmp_output);
stream_scatter(tmp_output, indices, output);
```

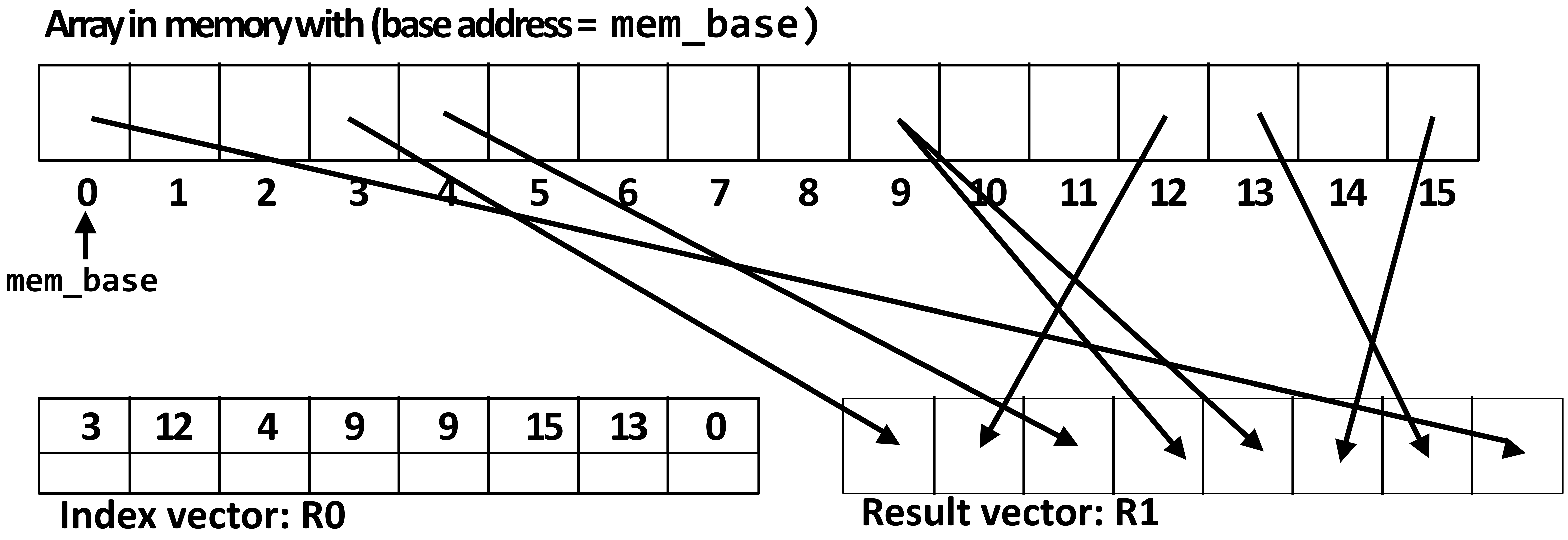
ISPC equivalent:

```
export void absolute_value(
   uniform float N,
   uniform float* input,
   uniform float* output,
   uniform int* indices)
{
   foreach (i = 0 ... n)
   {
      if (input[i] < 0)
         output[indices[i]] = -input[i];
      else
         output[indices[i]] = input[i];
   }
}
```

Gather instruction

`gather(R1, R0, mem_base);`

"Gather from buffer `mem_base` into R1 according to indices specified by R0."



Gather supported with AVX2 in 2013
But AVX2 does not support SIMD scatter (must implement as scalar loop)
Scatter instruction exists in AVX512

Hardware supported gather/scatter does exist on GPUs.
(still an expensive operation compared to load/store of contiguous vector)

Summary: data-parallel model

- **Data-parallelism is about imposing rigid program** structure to facilitate simple programming and advanced optimizations

- **Basic idea: map a function onto a large collection of data**
  - Functional: side-effect free execution
  - No communication among distinct function invocations (allow invocations to be scheduled in any order, including in parallel)

- In practice that's how many simple programs work

- But... many modern performance-oriented data-parallel languages do not <u>enforce</u> this structure in the language
  - ISPC, OpenCL, CUDA, etc.
  - They choose flexibility/familiarity of imperative C-style syntax over the safety of a more functional form

Summary

- **Programming models provide a way to think about the organization of parallel programs.**

- **They provide <u>abstractions</u> that permit multiple valid <u>implementations</u>.**

- ***I want you to always be thinking about abstraction vs. implementation for the remainder of this course.***