Lecture 3:

Parallel Programming Abstractions

(and their corresponding HW/SW implementations)

Parallel Computing

Stanford CS149, Winter 2019

Today's theme is a critical idea in this course.

And today's theme is:

Abstraction vs. implementation

Conflating abstraction with implementation is a common cause for confusion in this course.

An example:
Programming with ISPC

ISPC

- Intel SPMD Program Compiler (ISPC)
- SPMD: single program multiple data

- http://ispc.github.com/

Recall: example program from last class

Compute $\sin(x)$ **using Taylor expansion:** $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + ...$
for each element of an array of N floating-point numbers

```
void sinx(int N, int terms, float* x, float* result)
{
   for (int i=0; i<N; i++)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      int denom = 6;  // 3!
      int sign = -1;

      for (int j=1; j<=terms; j++)
      {
         value += sign * numer / denom;
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }

      result[i] = value;
   }
}
```

sin(x) in ISPC

Compute $\sin(x)$ **using Taylor expansion:** $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

SPMD programming abstraction:

Call to ISPC function spawns “gang” of ISPC “program instances”

All instances run ISPC code concurrently

Upon return, all instances have completed

ISPC code: `sinx.ispc`

```
export void sinx(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   // assume N % programCount = 0
   for (uniform int i=0; i<N; i+=programCount)
   {
      int idx = i + programIndex;
      float value = x[idx];
      float numer = x[idx] * x[idx] * x[idx];
      uniform int denom = 6;  // 3!
      uniform int sign = -1;

      for (uniform int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[idx] * x[idx];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }
      result[idx] = value;
   }
}
```

sin(x) in ISPC

Compute $\sin(x)$ **using Taylor expansion:** $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

C++ code: `main.cpp`

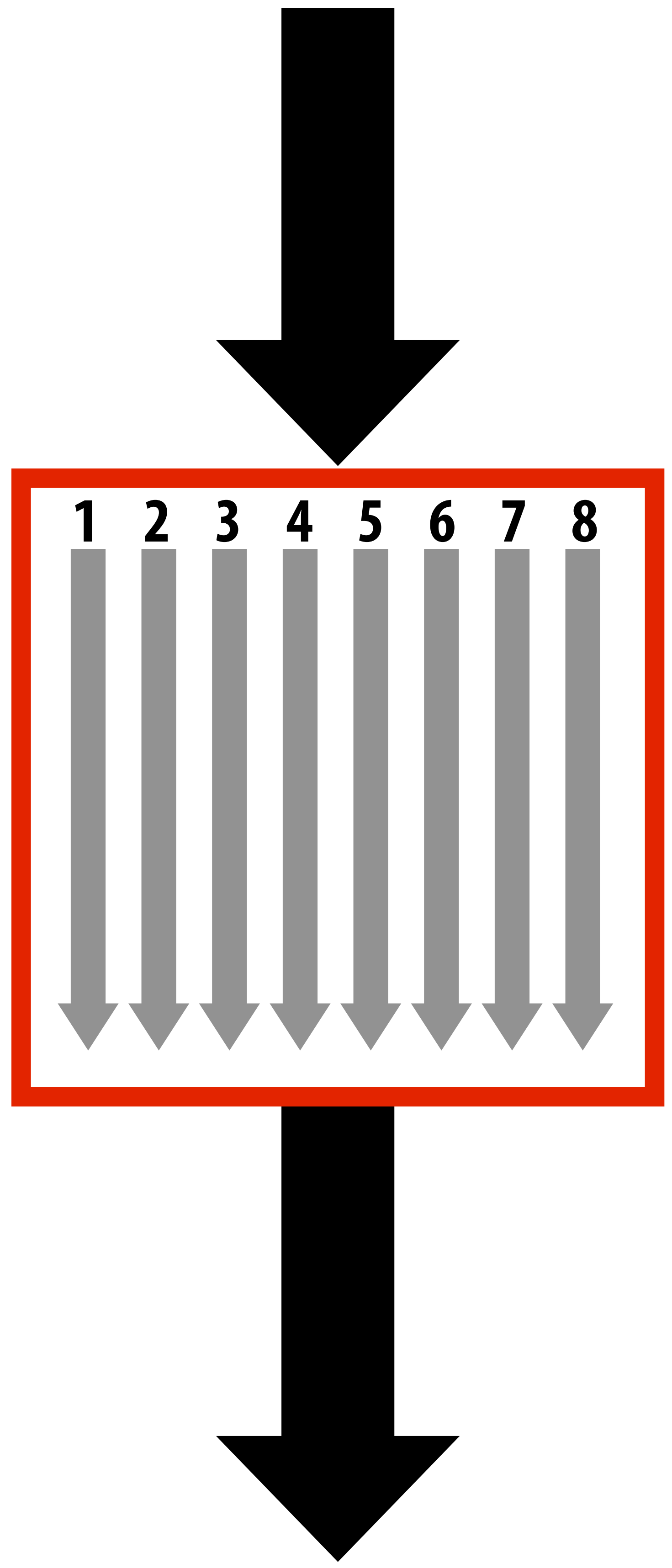
```
#include “sinx_ispc.h”

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

SPMD programming abstraction:

Call to ISPC function spawns “gang” of ISPC “program instances”

All instances run ISPC code concurrently

Upon return, all instances have completed



Sequential execution (C code)

Call to `sinx()`
Begin executing `programCount` instances of `sinx()` (ISPC code)

`sinx()` returns.
Completion of ISPC program instances.
Resume sequential execution

Sequential execution (C code)

In this illustration `programCount` = 8

sin(x) in ISPC

“Interleaved” assignment of array elements to program instances

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: `sinx.ispc`

```
export void sinx(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   // assumes N % programCount = 0
   for (uniform int i=0; i<N; i+=programCount)
   {
      int idx = i + programIndex;
      float value = x[idx];
      float numer = x[idx] * x[idx] * x[idx];
      uniform int denom = 6;  // 3!
      uniform int sign = -1;

      for (uniform int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[idx] * x[idx];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }
      result[idx] = value;
   }
}
```

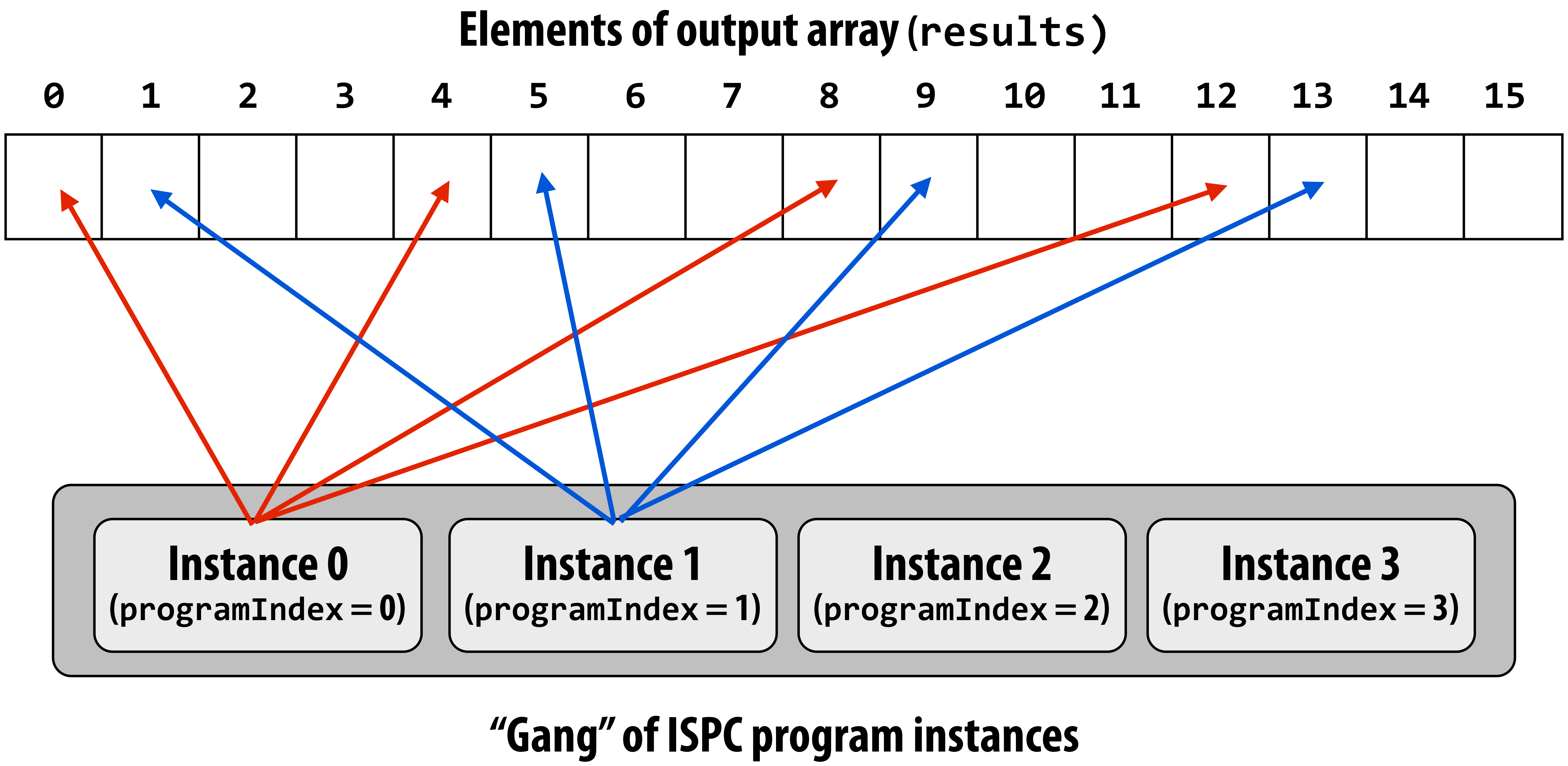
ISPC Keywords:

`programCount`: number of simultaneously executing instances in the gang (uniform value)

`programIndex`: id of the current instance in the gang. (a non-uniform value: “varying”)

`uniform`: A type modifier. All instances have the same value for this variable. Its use is purely an optimization. Not needed for correctness.

Interleaved assignment of program instances to loop iterations

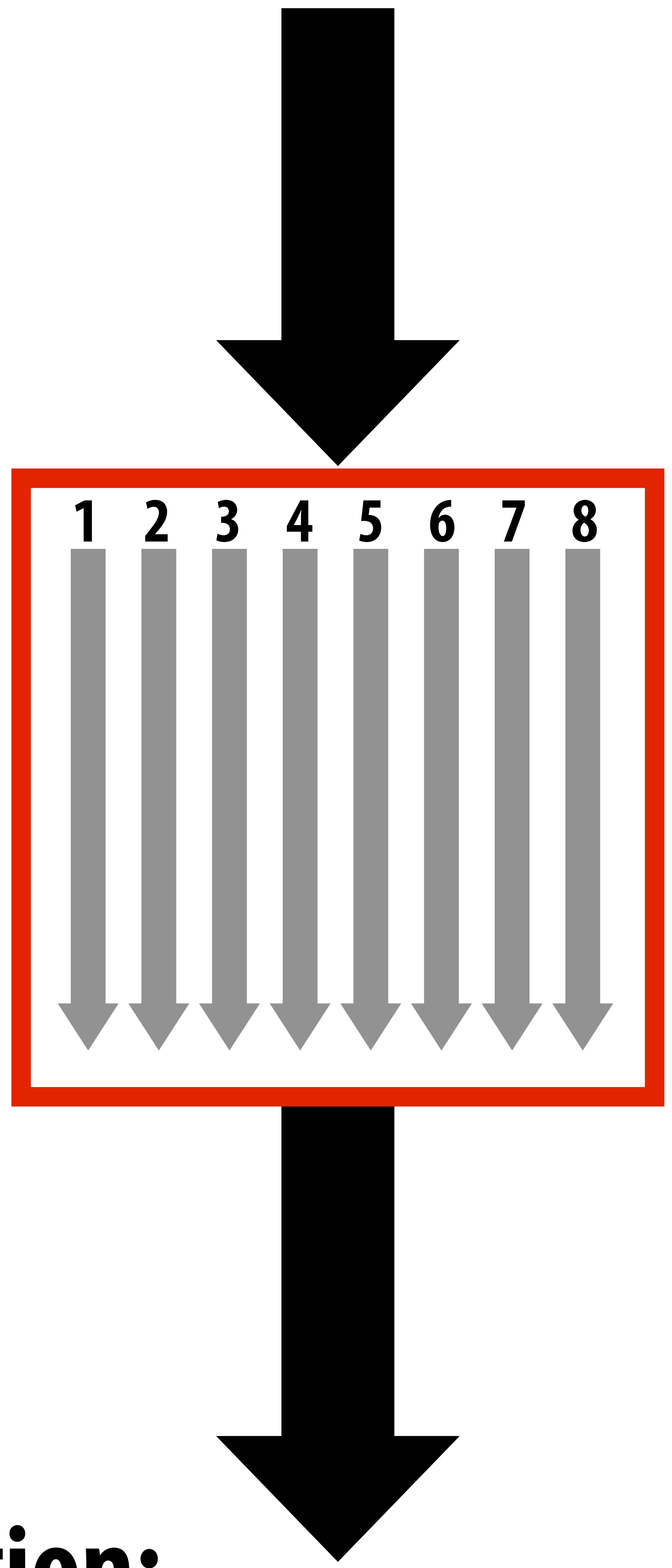


In this illustration: gang contains four instances: `programCount` = 4

ISPC implements the gang abstraction using SIMD instructions

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```



Sequential execution (C code)

Call to `sinx()`
Begin executing `programCount` instances of `sinx()` (ISPC code)

`sinx()` returns.
Completion of ISPC program instances.
Resume sequential execution

Sequential execution (C code)

SPMD programming abstraction:

Call to ISPC function spawns “gang” of ISPC “program instances”

All instances run ISPC code concurrently

Upon return, all instances have completed

ISPC compiler generates SIMD implementation:

Number of instances in a gang is the SIMD width of the hardware (or a small multiple of SIMD width)

ISPC compiler generates binary (.o) with SIMD instructions

C++ code links against object file as usual

sin(x) in ISPC: version 2

"Blocked" assignment of elements to instances

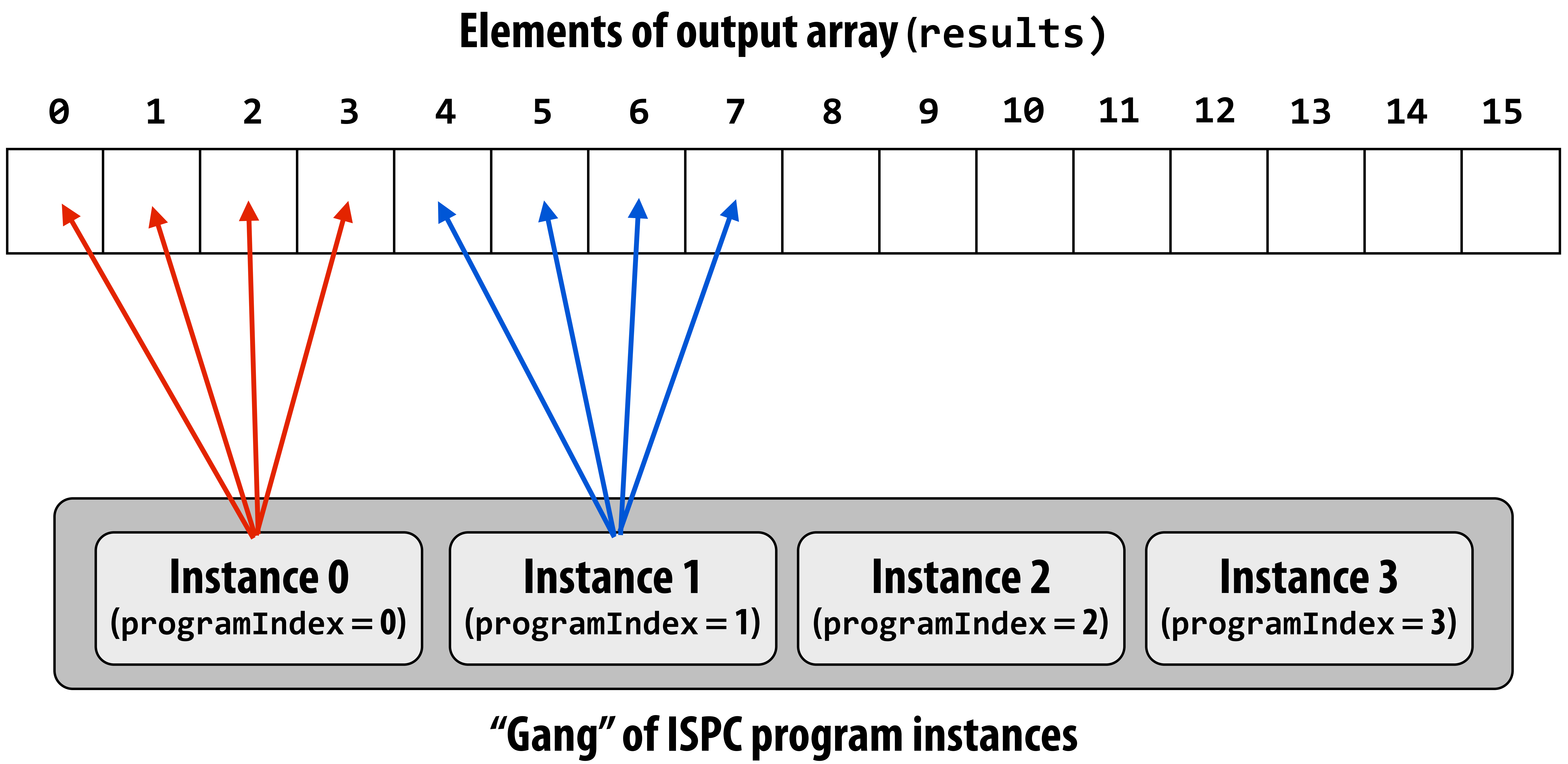
C++ code: `main.cpp`

```
#include "sinx_ispc.h"

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: `sinx.ispc`

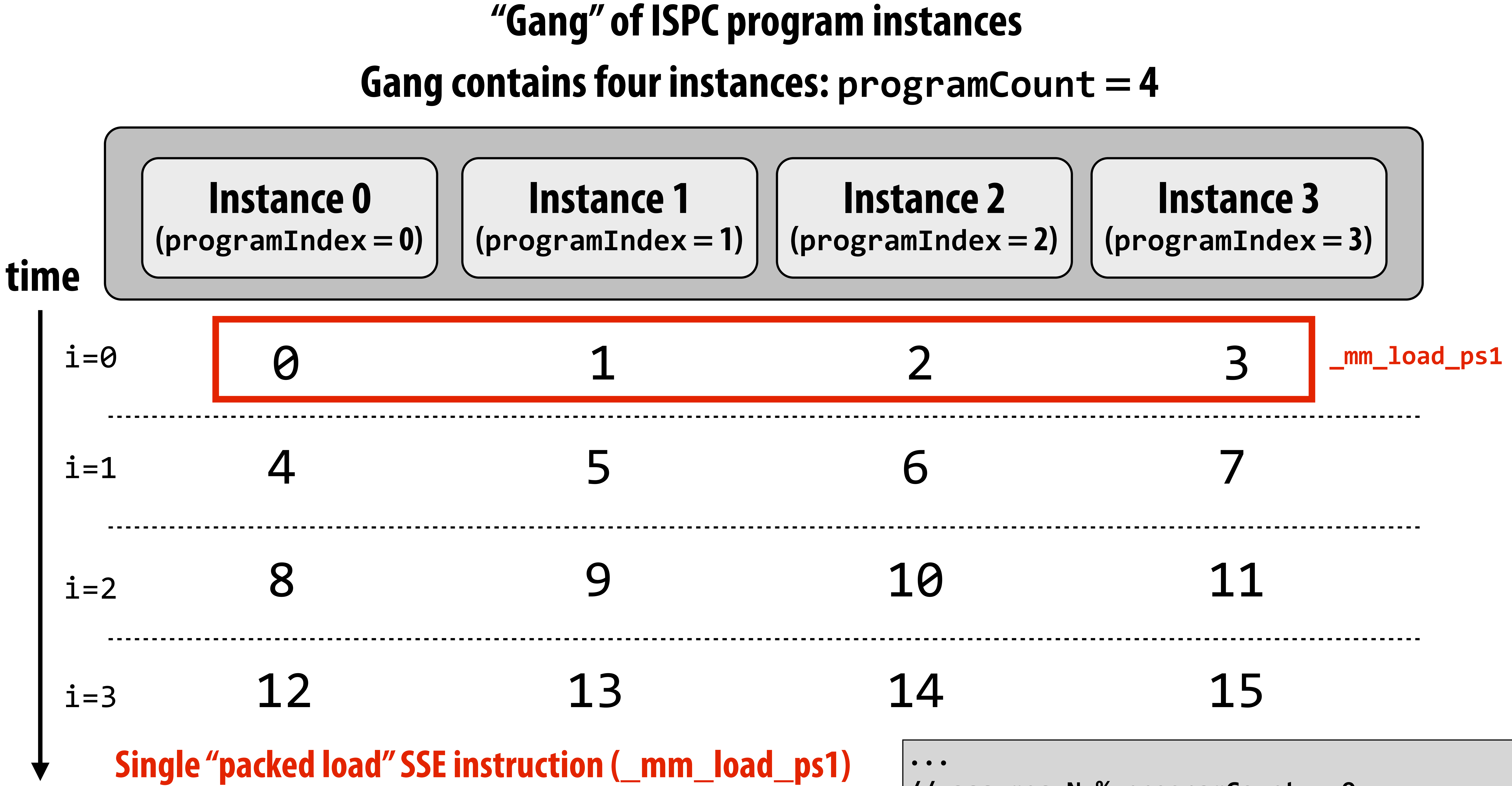
```
export void sinx(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   // assume N % programCount = 0
   uniform int count = N / programCount;
   int start = programIndex * count;
   for (uniform int i=0; i<count; i++)
   {
      int idx = start + i;
      float value = x[idx];
      float numer = x[idx] * x[idx] * x[idx];
      uniform int denom = 6;  // 3!
      uniform int sign = -1;

      for (uniform int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[idx] * x[idx];
         denom *= (j+3) * (j+4);
         sign *= -1;
      }
      result[idx] = value;
   }
}
```

Blocked assignment of program instances to loop iterations



In this illustration: gang contains four instances: `programCount` = 4

Schedule: interleaved assignment



"Gang" of ISPC program instances
Gang contains four instances: programCount = 4
Instance 0 (programIndex = 0)
Instance 1 (programIndex = 1)
Instance 2 (programIndex = 2)
Instance 3 (programIndex = 3)
time
i=0
0 1 2 3
_mm_load_ps1
i=1
4 5 6 7
i=2
8 9 10 11
i=3
12 13 14 15

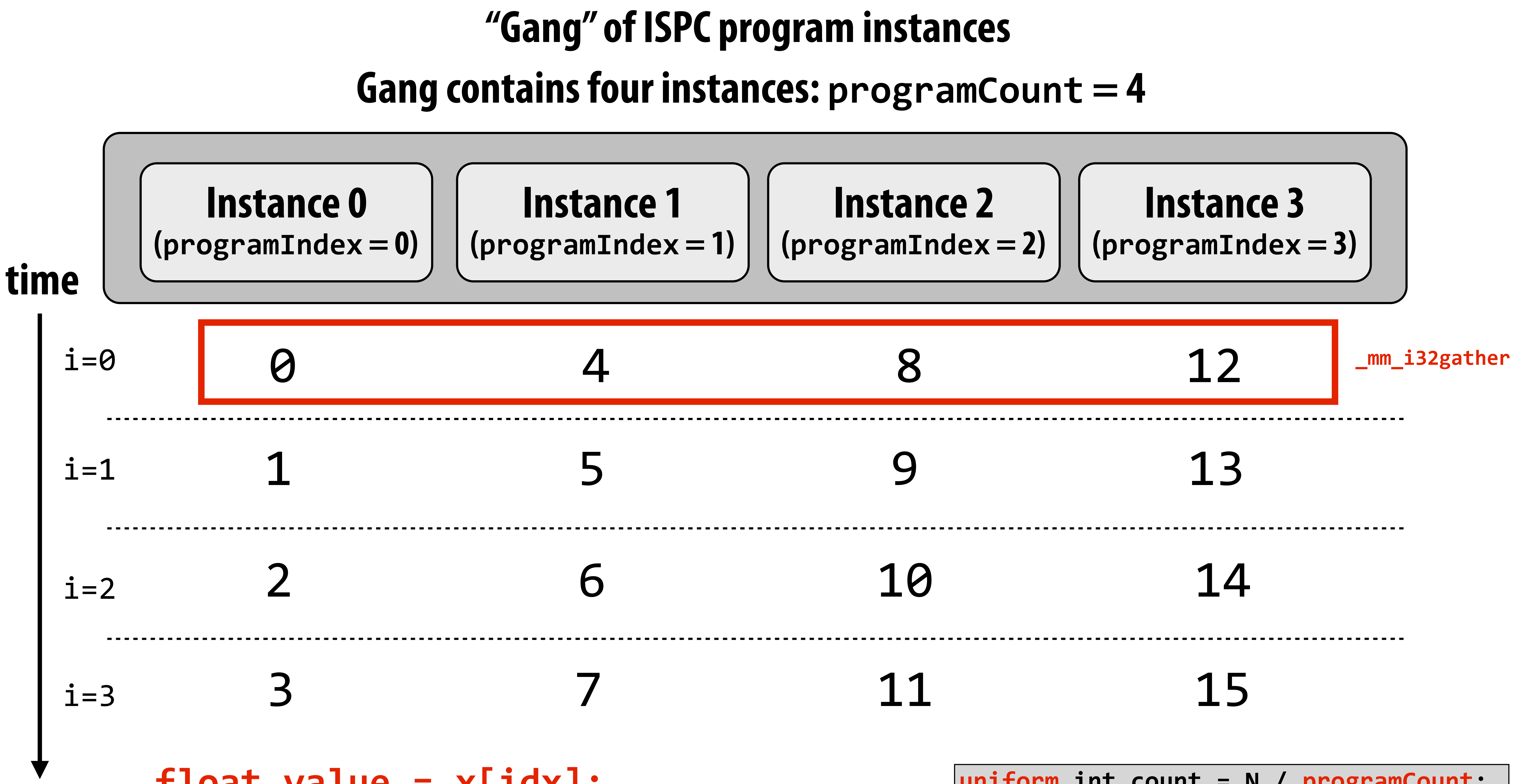
Single "packed load" SSE instruction (_mm_load_ps1) efficiently implements:

`float value = x[idx];`

for all program instances, since the four values are contiguous in memory

```
...
// assumes N % programCount = 0
for (uniform int i=0; i<N; i+=programCount)
   {
      int idx = i + programIndex;
      float value = x[idx];
...
```

Schedule: blocked assignment


"Gang" of ISPC program instances
Gang contains four instances: programCount = 4
Instance 0 (programIndex = 0)
Instance 1 (programIndex = 1)
Instance 2 (programIndex = 2)
Instance 3 (programIndex = 3)
time
i=0
i=1
i=2
i=3
0 4 8 12
1 5 9 13
2 6 10 14
3 7 11 15
_mm_i32gather

```
float value = x[idx];
```

now touches four non-contiguous values in memory.

Need "gather" instruction to implement
(gather is a more complex, and more costly SIMD instruction: only available since 2013 as part of AVX2)

```
uniform int count = N / programCount;
int start = programIndex * count;
for (uniform int i=0; i<count; i++) {
   int idx = start + i;
   float value = x[idx];
...
```

Raising level of abstraction with foreach

Compute $\sin(x)$ **using Taylor expansion:** $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

int N = 1024;
int terms = 5;
float* x = new float[N];
float* result = new float[N];

// initialize x here

// execute ISPC code
sinx(N, terms, x, result);
```

ISPC code: `sinx.ispc`

```
export void sinx(
   uniform int N,
   uniform int terms,
   uniform float* x,
   uniform float* result)
{
   foreach (i = 0 ... N)
   {
      float value = x[i];
      float numer = x[i] * x[i] * x[i];
      uniform int denom = 6;  // 3!
      uniform int sign = -1;

      for (uniform int j=1; j<=terms; j++)
      {
         value += sign * numer / denom
         numer *= x[i] * x[i];
         denom *= (2*j+2) * (2*j+3);
         sign *= -1;
      }
      result[i] = value;
   }
}
```

`foreach`: key ISPC language construct

- `foreach` declares parallel loop iterations
  - Programmer says: these are the iterations the instances in a gang cooperatively must perform
- ISPC implementation assigns iterations to program instances in gang
  - Current ISPC implementation will perform a static interleaved assignment (but the abstraction permits a different assignment)

ISPC: abstraction vs. implementation

- Single program, multiple data (SPMD) programming model
  - Programmer "thinks": running a gang is spawning `programCount` logical instruction streams (each with a different value of `programIndex`)
  - This is the programming <u>abstraction</u>
  - Program is written in terms of this abstraction

- Single instruction, multiple data (SIMD) <u>implementation</u>
  - ISPC compiler emits vector instructions (e.g., AVX2) that carry out the logic performed by a ISPC gang
  - ISPC compiler handles mapping of conditional control flow to vector instructions (by masking vector lanes, etc.)

- Semantics of ISPC can be tricky
  - SPMD abstraction + uniform values
    (allows implementation details to peak through abstraction a bit)

ISPC discussion: sum "reduction"

Compute the sum of all array elements in parallel

```
export uniform float sumall1(
   uniform int N,
   uniform float* x)
{
   uniform float sum = 0.0f;
   foreach (i = 0 ... N)
   {
      sum += x[i];
   }

   return sum;
}
```

```
export uniform float sumall2(
   uniform int N,
   uniform float* x)
{
   uniform float sum;
   float partial = 0.0f;
   foreach (i = 0 ... N)
   {
      partial += x[i];
   }

   // from ISPC math library
   sum = reduce_add(partial);

   return sum;
}
```

Correct ISPC solution

`sum` is of type `uniform float` (one copy of variable for all program instances)

`x[i]` is not a uniform expression (different value for each program instance)

Result: compile-time type error

ISPC discussion: sum “reduction”

Compute the sum of all array elements in parallel

```
export uniform float sumall2(
   uniform int N,
   uniform float* x)
{
   uniform float sum;
   float partial = 0.0f;
   foreach (i = 0 ... N)
   {
      partial += x[i];
   }

   // from ISPC math library
   sum = reduce_add(partial);

   return sum;
}
```

Each instance accumulates a private partial sum (no communication)

Partial sums are added together using the `reduce_add()` cross-instance communication primitive. The result is the same total sum for all program instances (`reduce_add()` returns a uniform float)

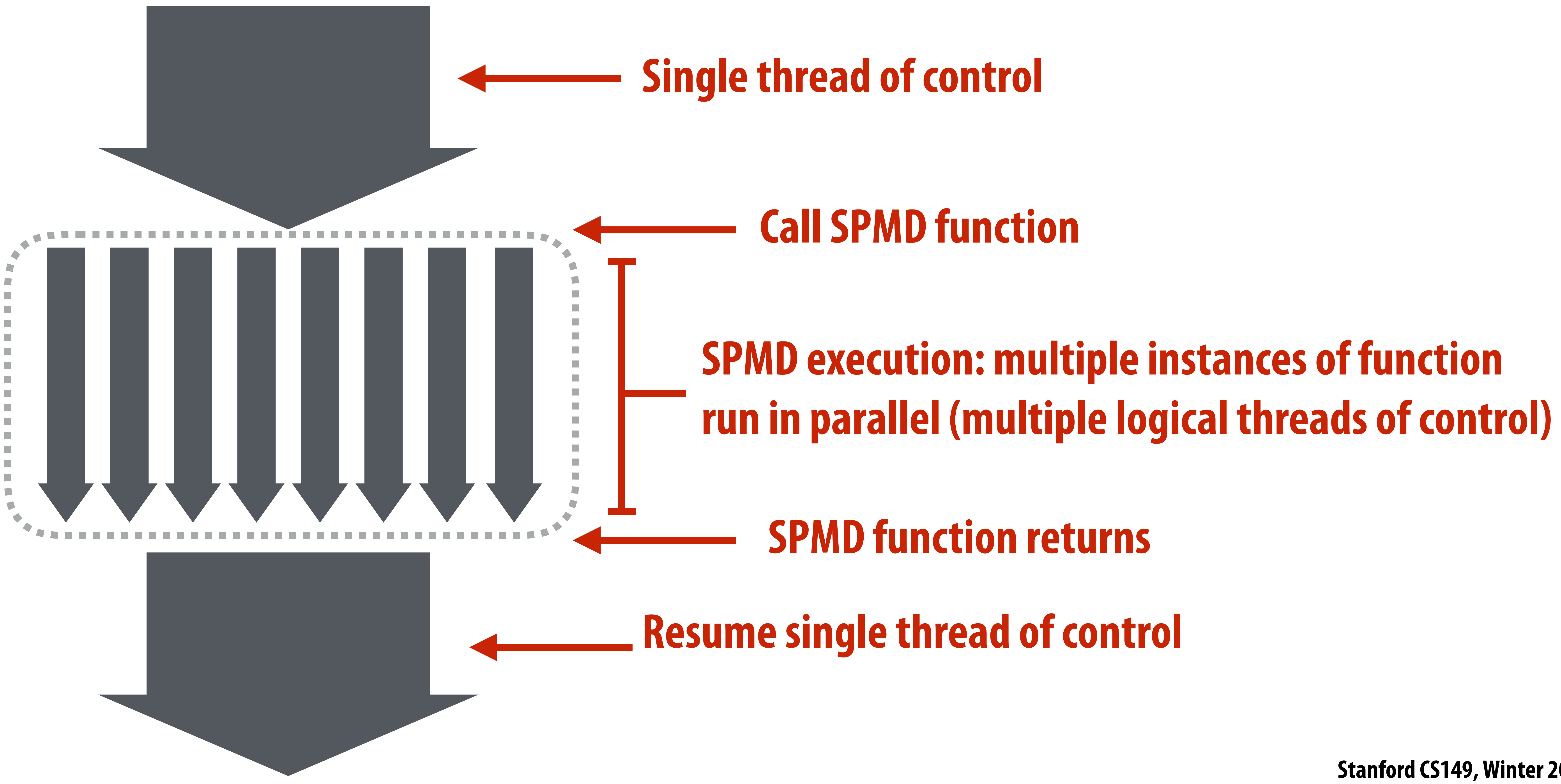
The ISPC code at right will execute in a manner similar to handwritten C + AVX intrinsics implementation below. *

```
float sumall2(int N, float* x) {

  float tmp[8];  // assume 16-byte alignment
  __mm256 partial = _mm256_broadcast_ss(0.0f);

  for (int i=0; i<N; i+=8)
    partial = _mm256_add_ps(partial, _mm256_load_ps(&x[i]));

  _mm256_store_ps(tmp, partial);

  float sum = 0.f;
  for (int i=0; i<8; i++)
    sum += tmp[i];

  return sum;
}
```

*** Self-test: If you understand why this implementation complies with the semantics of the ISPC gang abstraction, then you’ve got a good command of ISPC**

SPMD programming model summary

- SPMD = "single program, multiple data"
- Define one function, run multiple instances of that function in parallel on different input arguments


Single thread of control
Call SPMD function
SPMD execution: multiple instances of function run in parallel (multiple logical threads of control)
SPMD function returns
Resume single thread of control

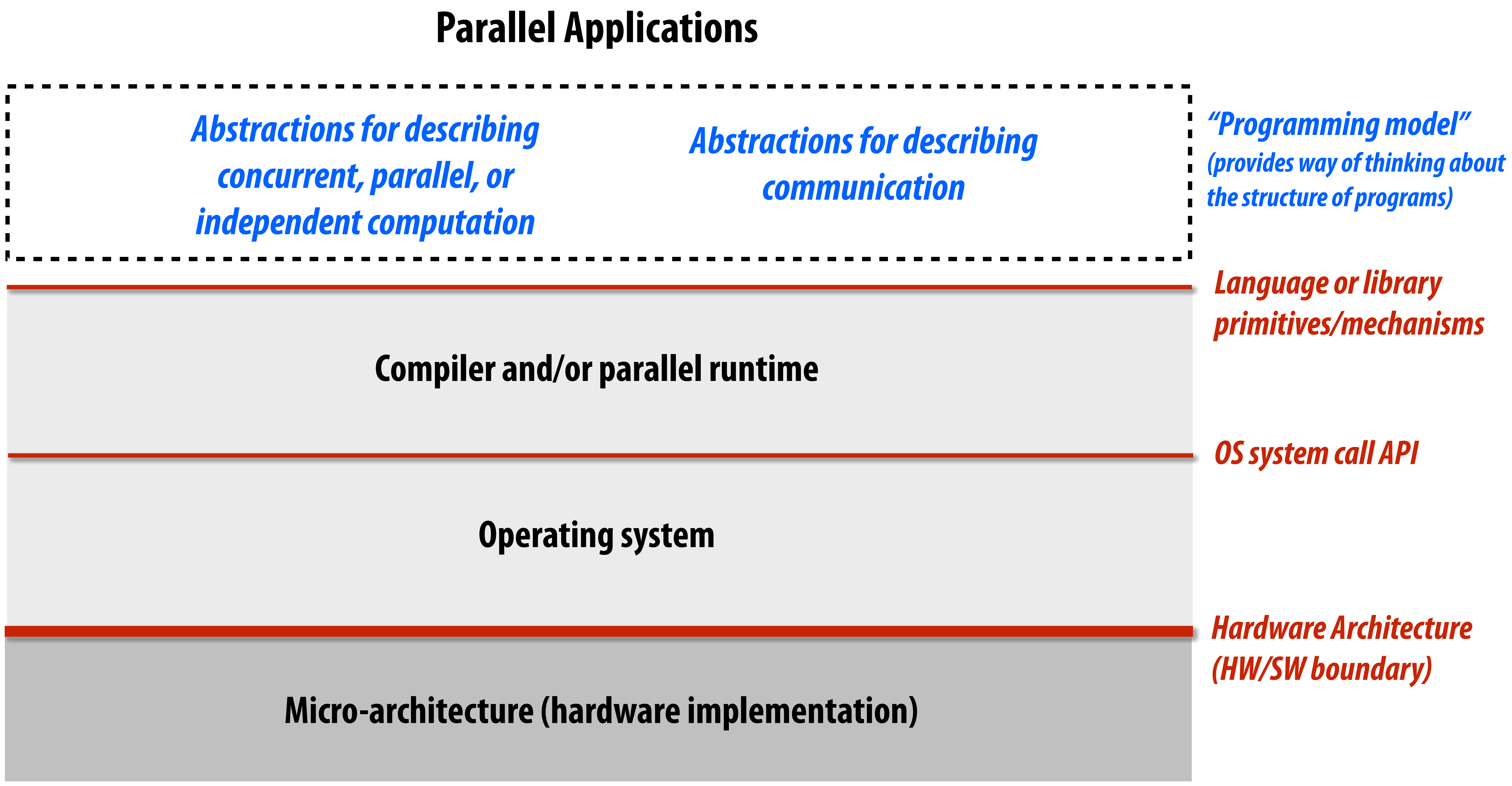
ISPC tasks

- The ISPC gang abstraction is implemented by SIMD instructions on one core.
- So... all the code I've shown you in the previous slides would have executed on only one of the four cores of the GHC machines.
- ISPC contains another abstraction: a "task" that is used to achieve multi-core execution. I'll let you read up about that.

Part 2 of today's lecture

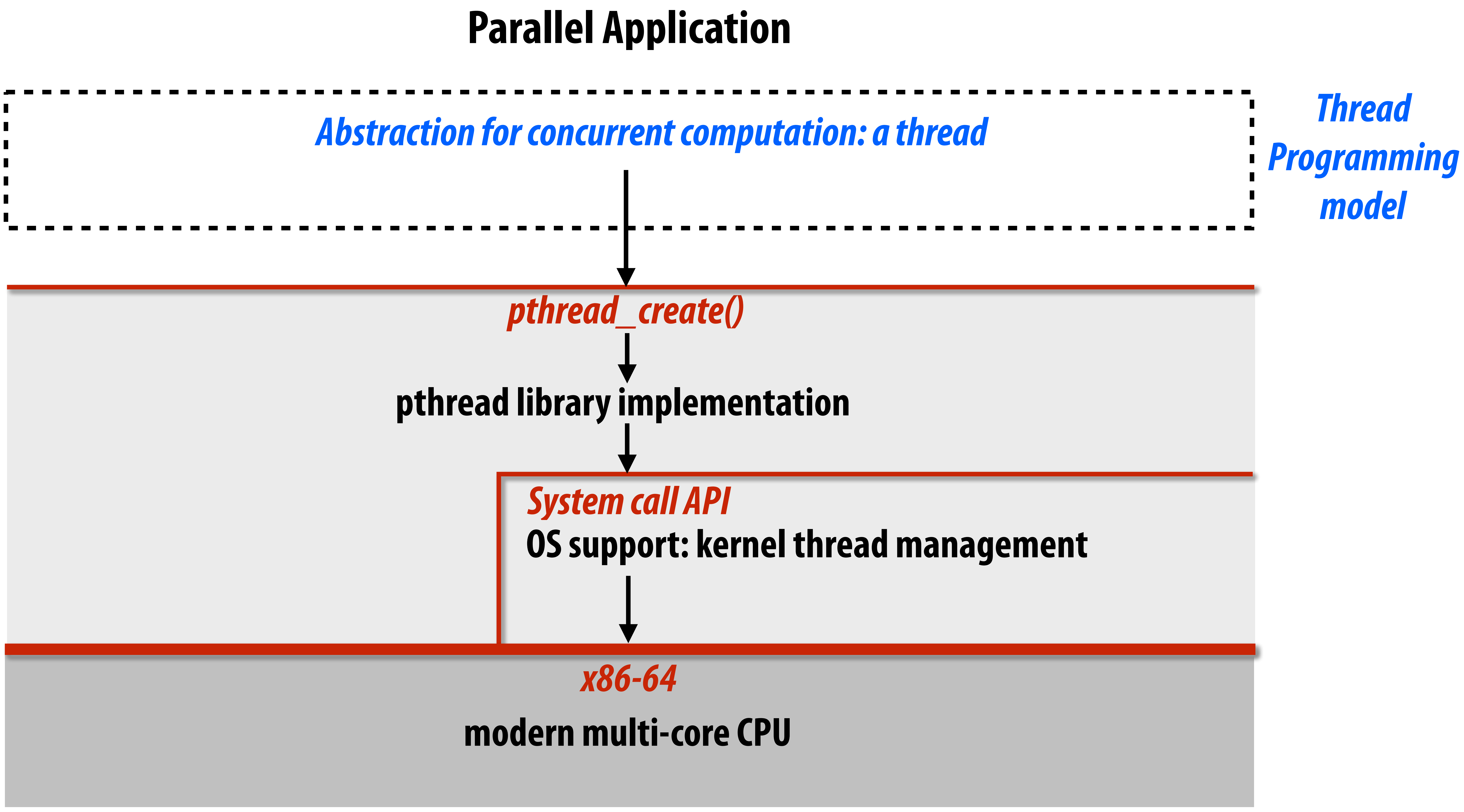
- Three parallel programming models
  - That differ in what communication abstractions they present to the programmer
  - Programming models are important because they (1) influence how programmers think when writing programs and (2) influence the design of parallel hardware platforms designed to execute them
- Corresponding machine architectures
  - Abstraction presented by the hardware to low-level software
- We'll focus on differences in communication and 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)

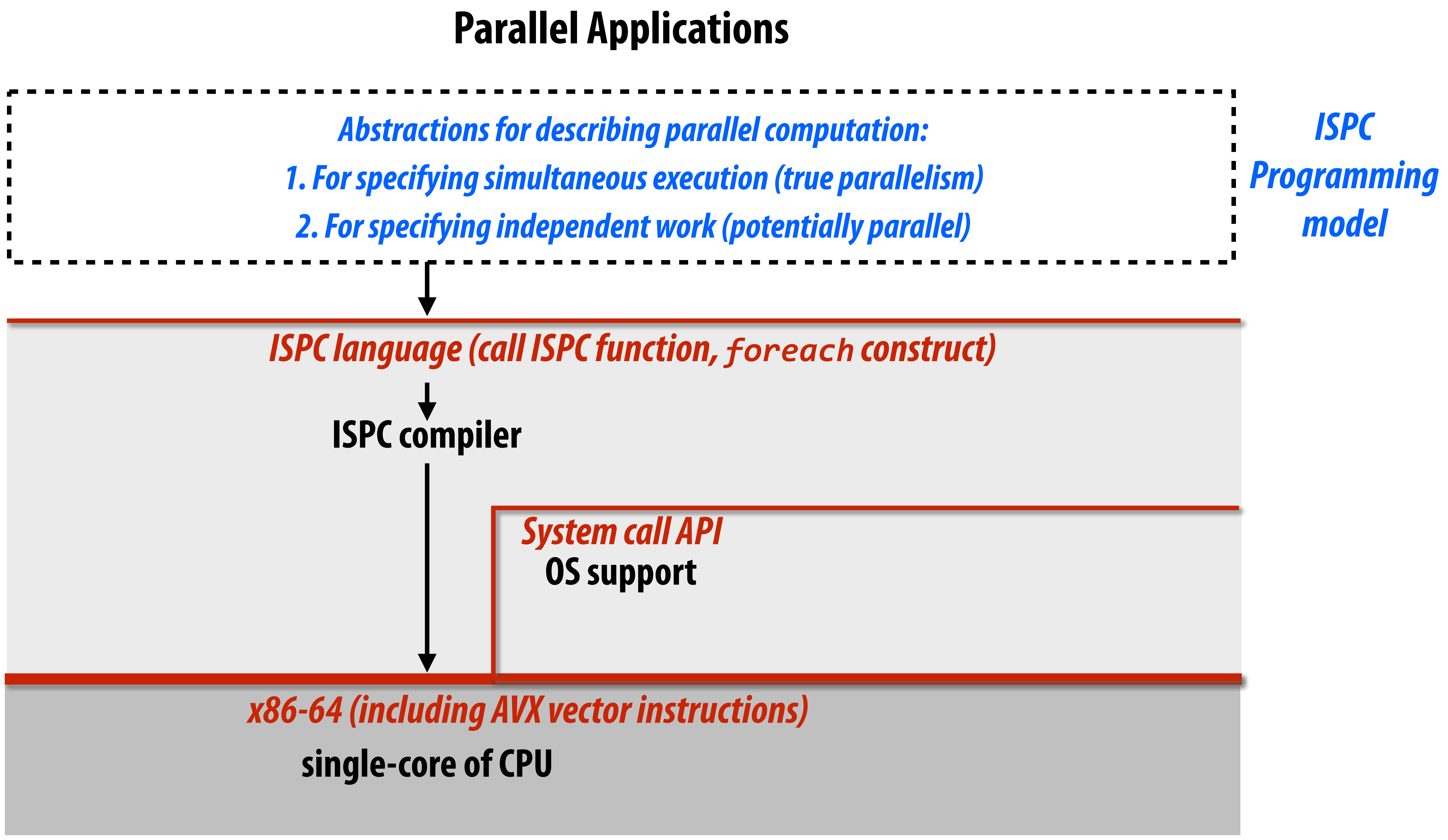
Blue italic text: abstraction/concept
Red italic text: system interface
Black text: system implementation

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

Three models of communication (abstractions)

1. Shared address space
2. Message passing
3. Data parallel

Shared address space model of communication

Shared address space model (abstraction)

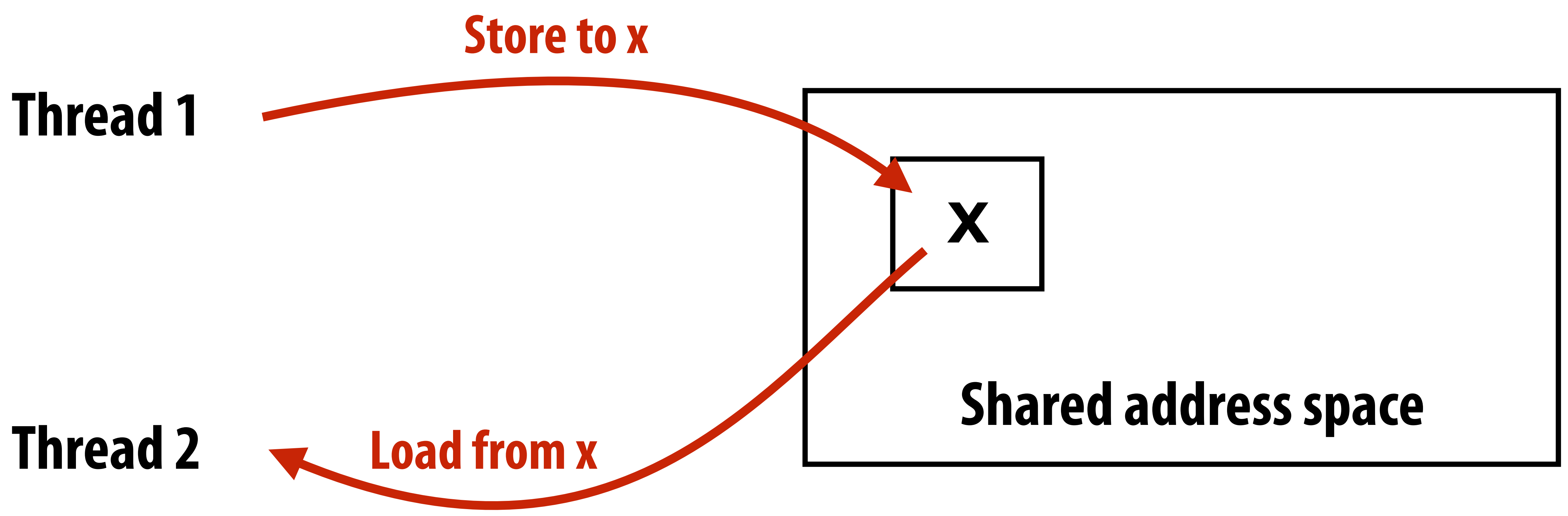
- Threads communicate by reading/writing to shared variables
- Shared variables are like a big bulletin board
    - Any thread can read or write to shared variables

Thread 1:

```
int x = 0;
spawn_thread(foo, &x);
x = 1;
```

Thread 2:

```
void foo(int* 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 (abstraction)

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.)

Review: why do we need mutual exclusion?

- Each thread executes
  - Load the value of diff from shared memory into register r1
  - Add the register r2 to register r1
  - Store the value of register r1 into diff
- One possible interleaving: (let starting value of diff=0, r2=1)

T0	T1	
`r1 ← diff`		`T0 reads value 0`
	`r1 ← diff`	`T1 reads value 0`
`r1 ← r1 + r2`		`T0 sets value of its r1 to 1`
	`r1 ← r1 + r2`	`T1 sets value of its r1 to 1`
`diff ← r1`		`T0 stores 1 to diff`
	`diff ← r1`	`T1 stores 1 to diff`

- This set of three instructions must be "atomic"

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);
```

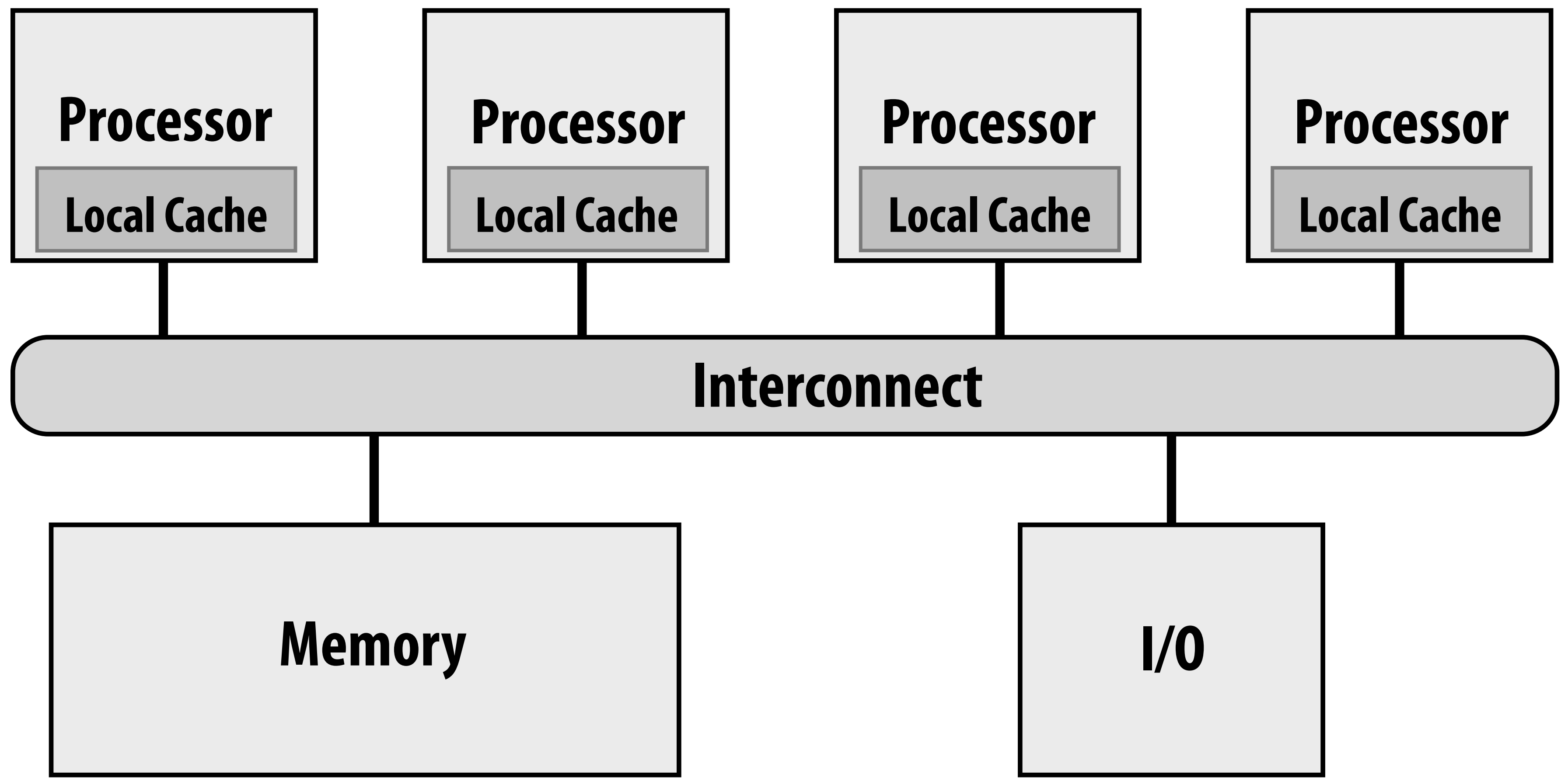
Shared address space model (abstraction)

- **Threads communicate by:**
    - Reading/writing to shared variables
        - Inter-thread communication is implicit in memory operations
        - 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!
- **Helpful analogy: shared variables are like a big bulletin board**
    - Any thread can read or write to shared variables

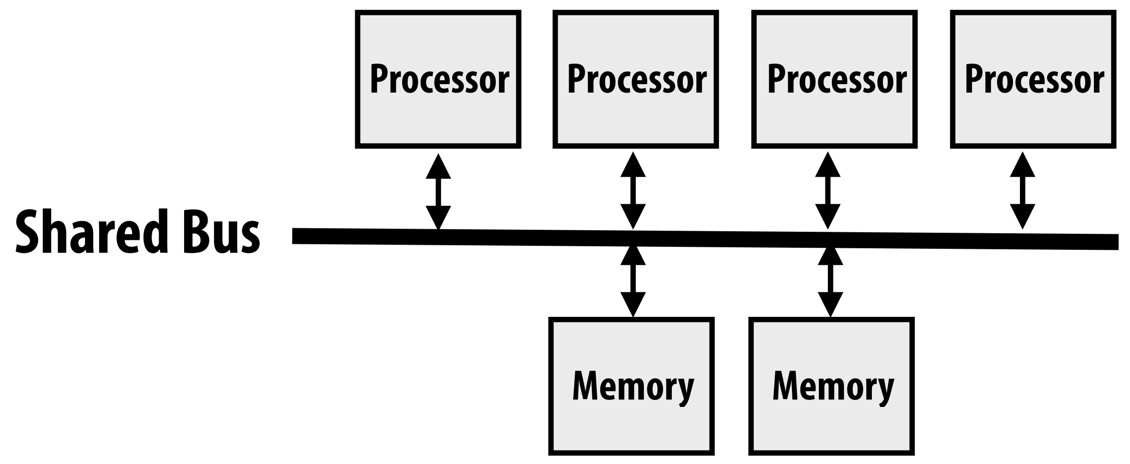
HW implementation of a shared address space

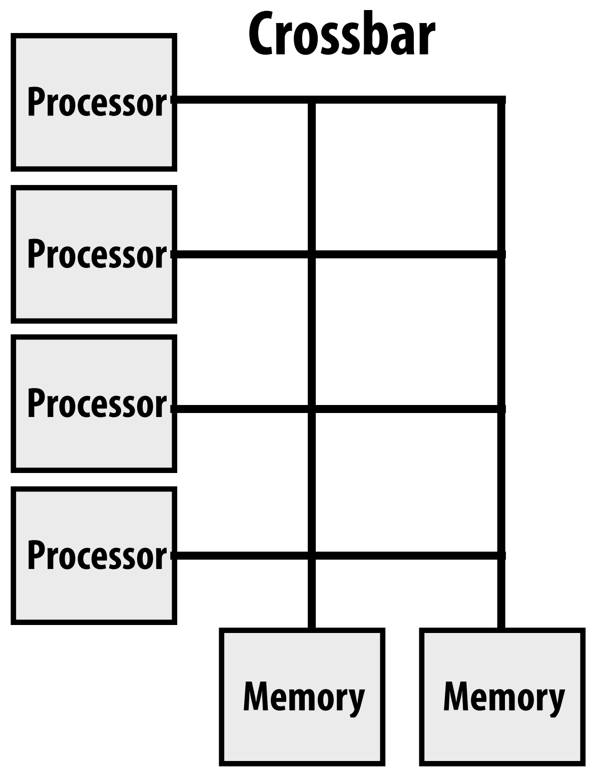
Key idea: any processor can directly reference any memory location

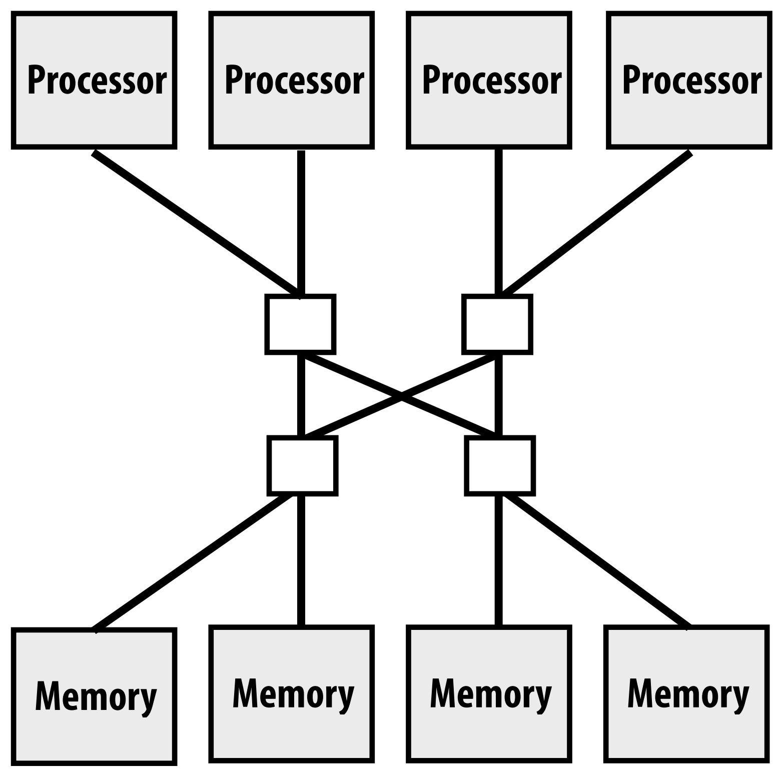
"Dance-hall" organization



Interconnect examples







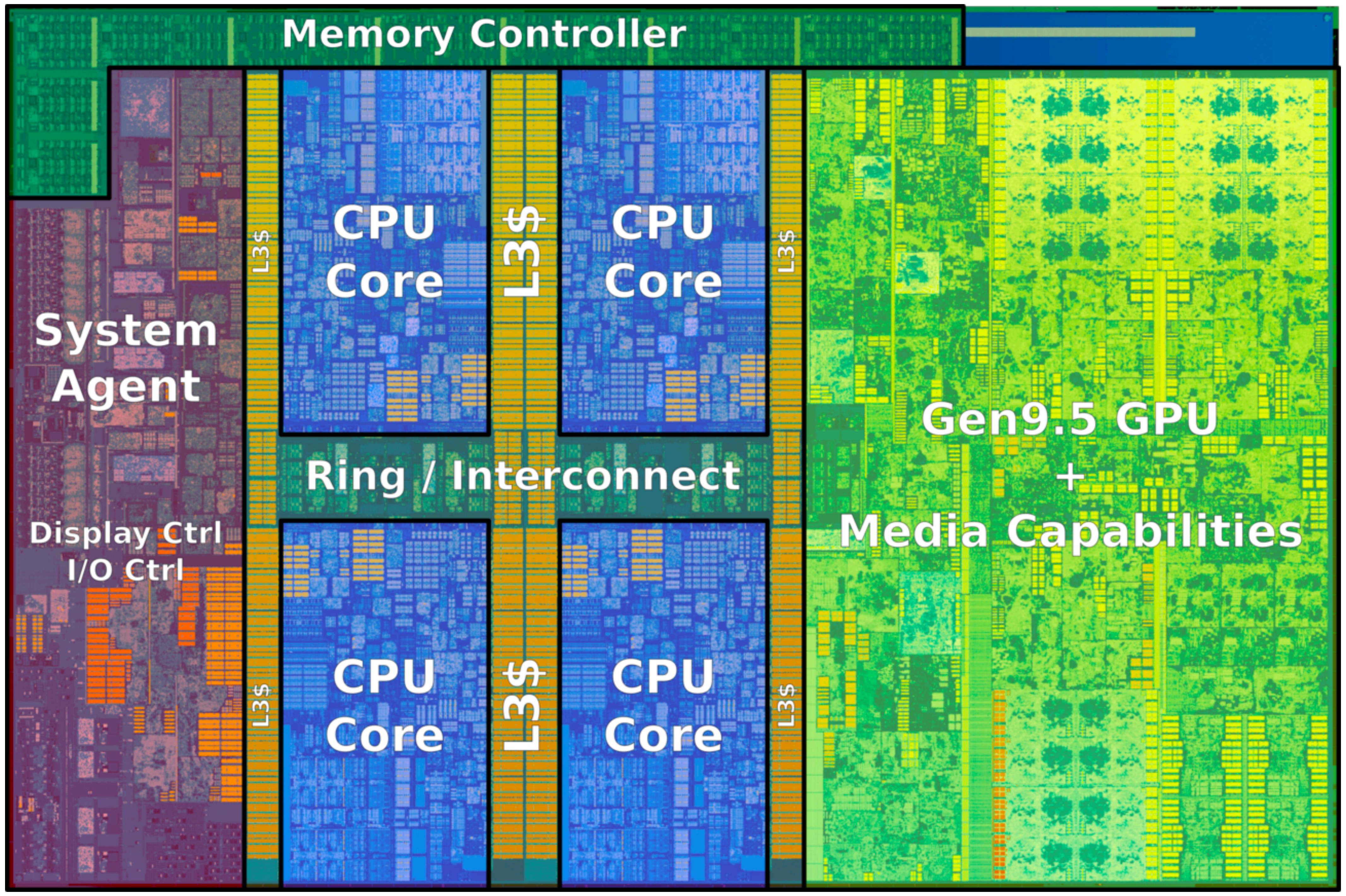
Multi-stage network

Symmetric (shared-memory) multi-processor (SMP):

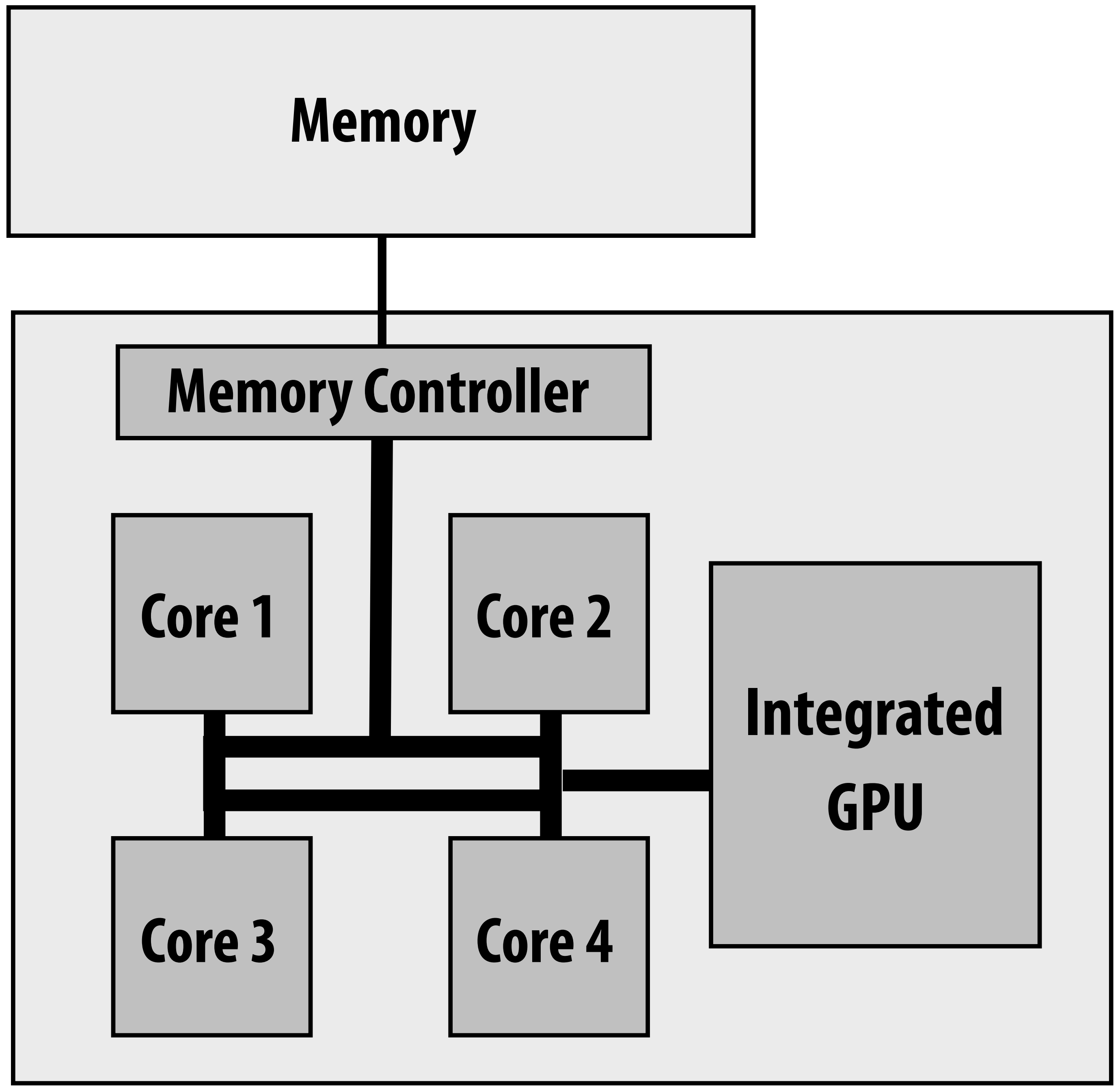
- Uniform memory access time: cost of accessing an uncached * memory address is the same for all processors

* caching introduces non-uniform access times, but we'll talk about that later

Shared address space HW architectures



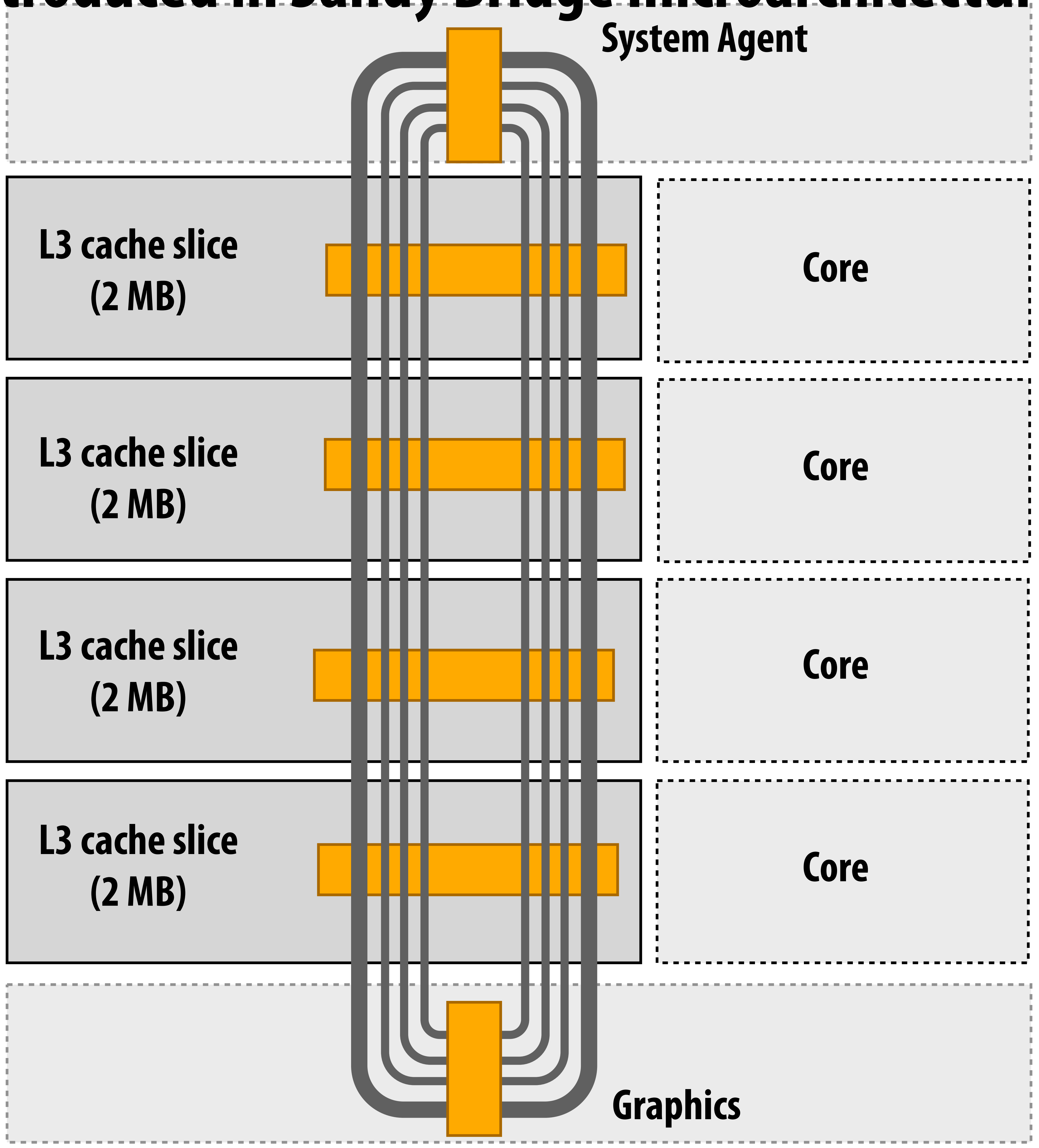
Example: Intel Core i7 processor (Kaby Lake)



Intel Core i7 (quad core)
(interconnect is a ring)

Intel's ring interconnect

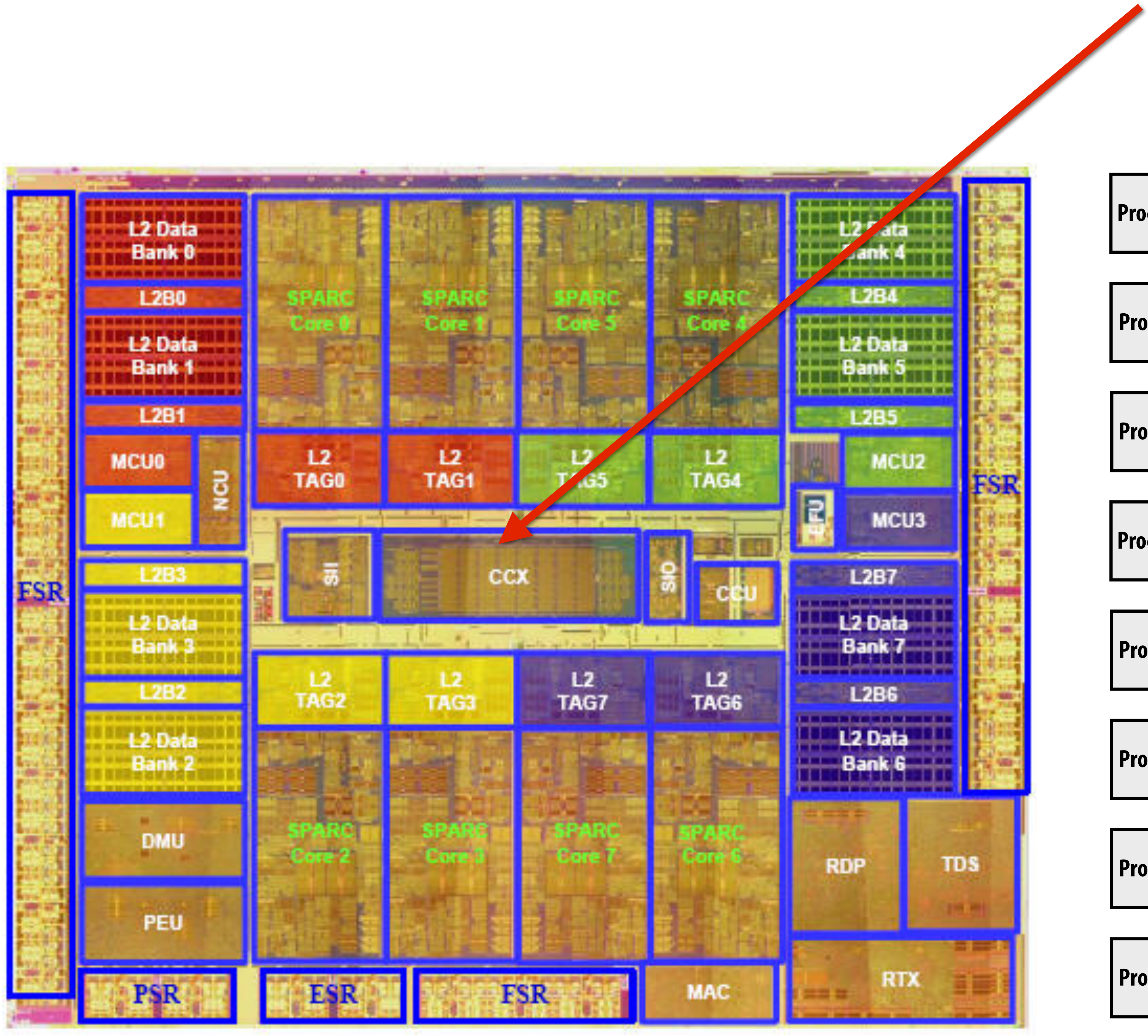
Introduced in Sandy Bridge microarchitecture



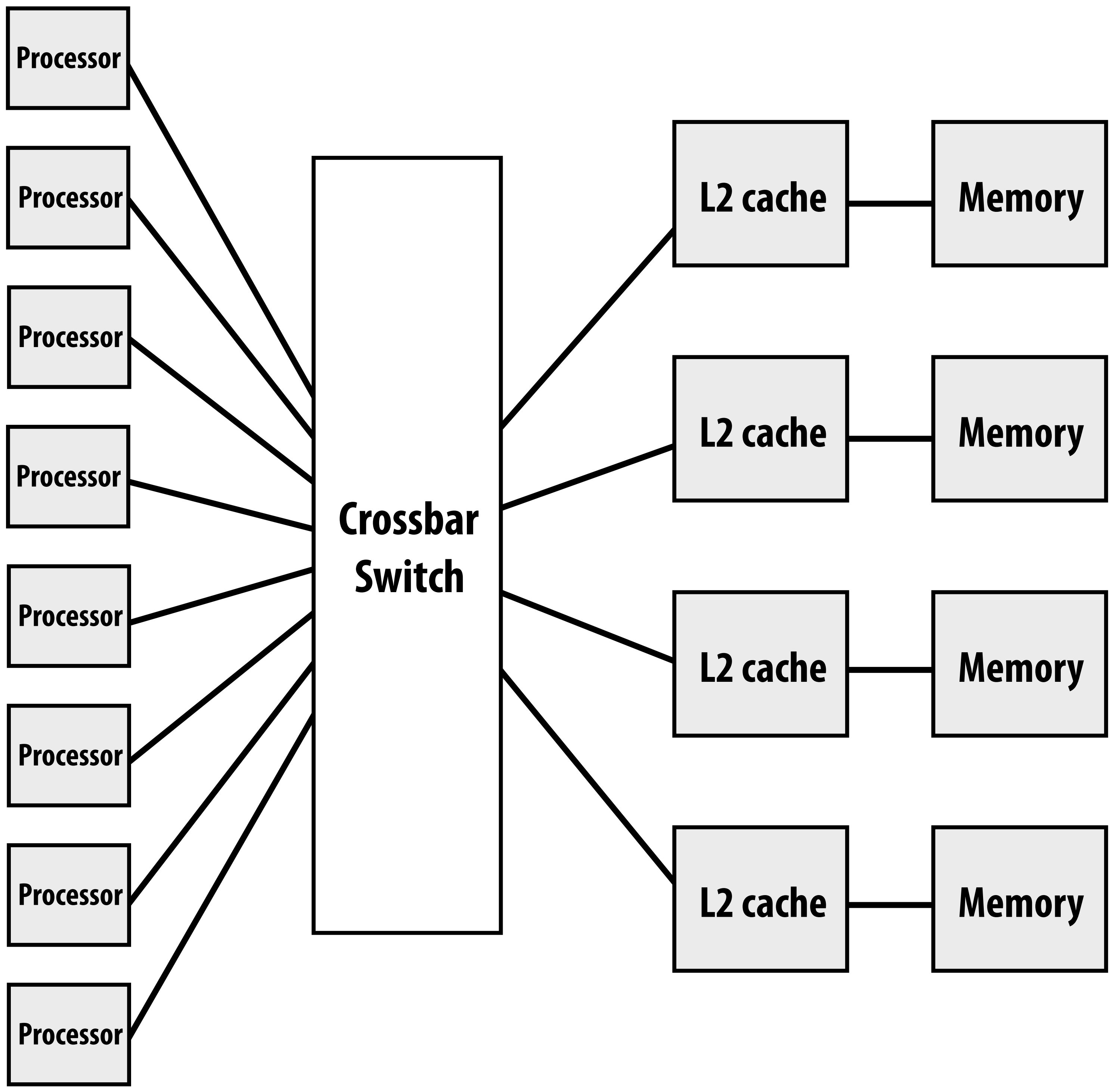
- Four rings
  - request
  - snoop
  - ack
  - data (32 bytes)
- Six interconnect nodes: four "slices" of L3 cache + system agent + graphics
- Each bank of L3 connected to ring bus twice
- Theoretical peak BW from cores to L3 at 3.4 GHz is approx. 435 GB/sec
  - When each core is accessing its local slice

SUN Niagara 2 (UltraSPARC T2)

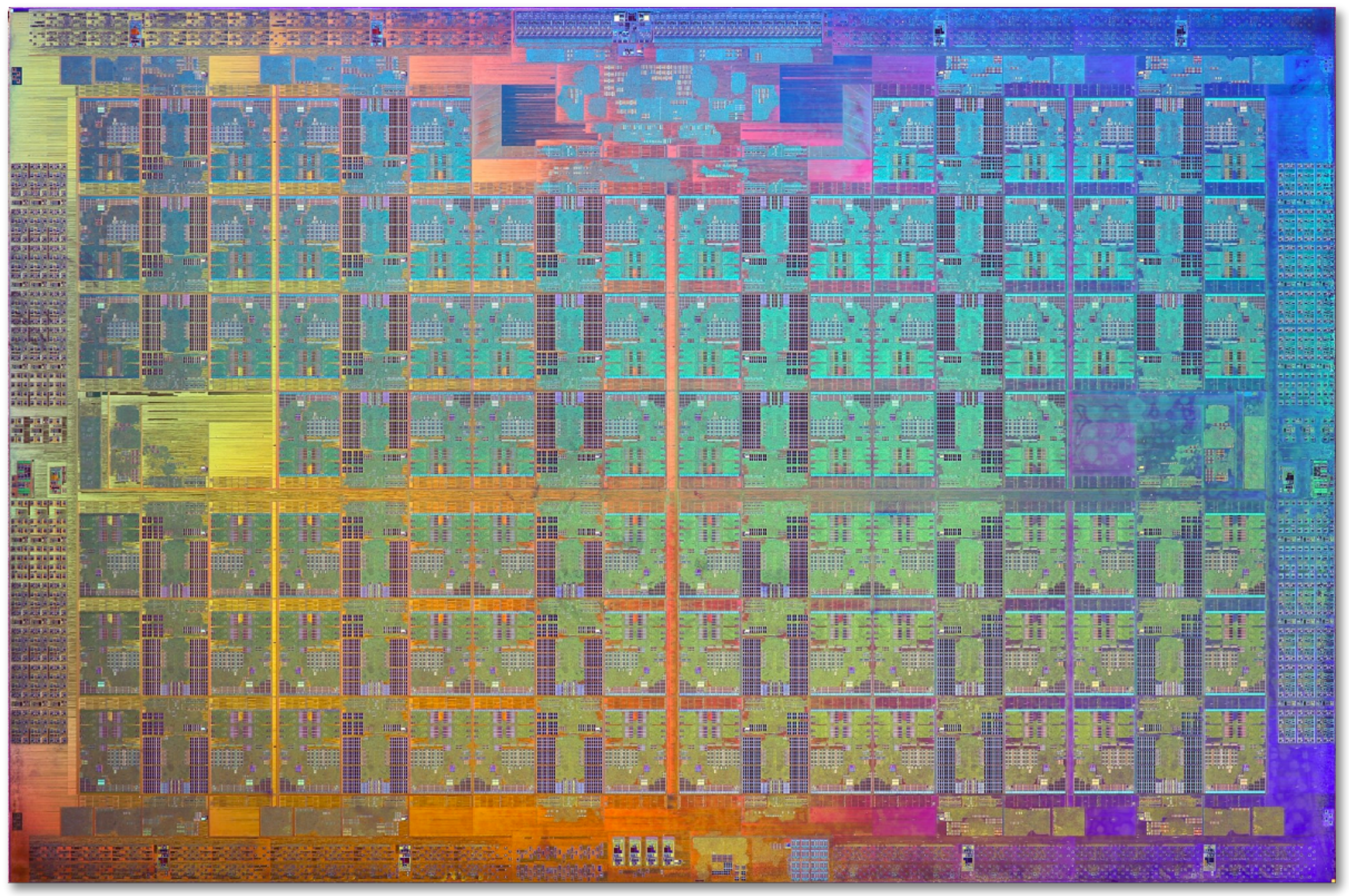
Note area of crossbar (CCX): about same area as one core on chip



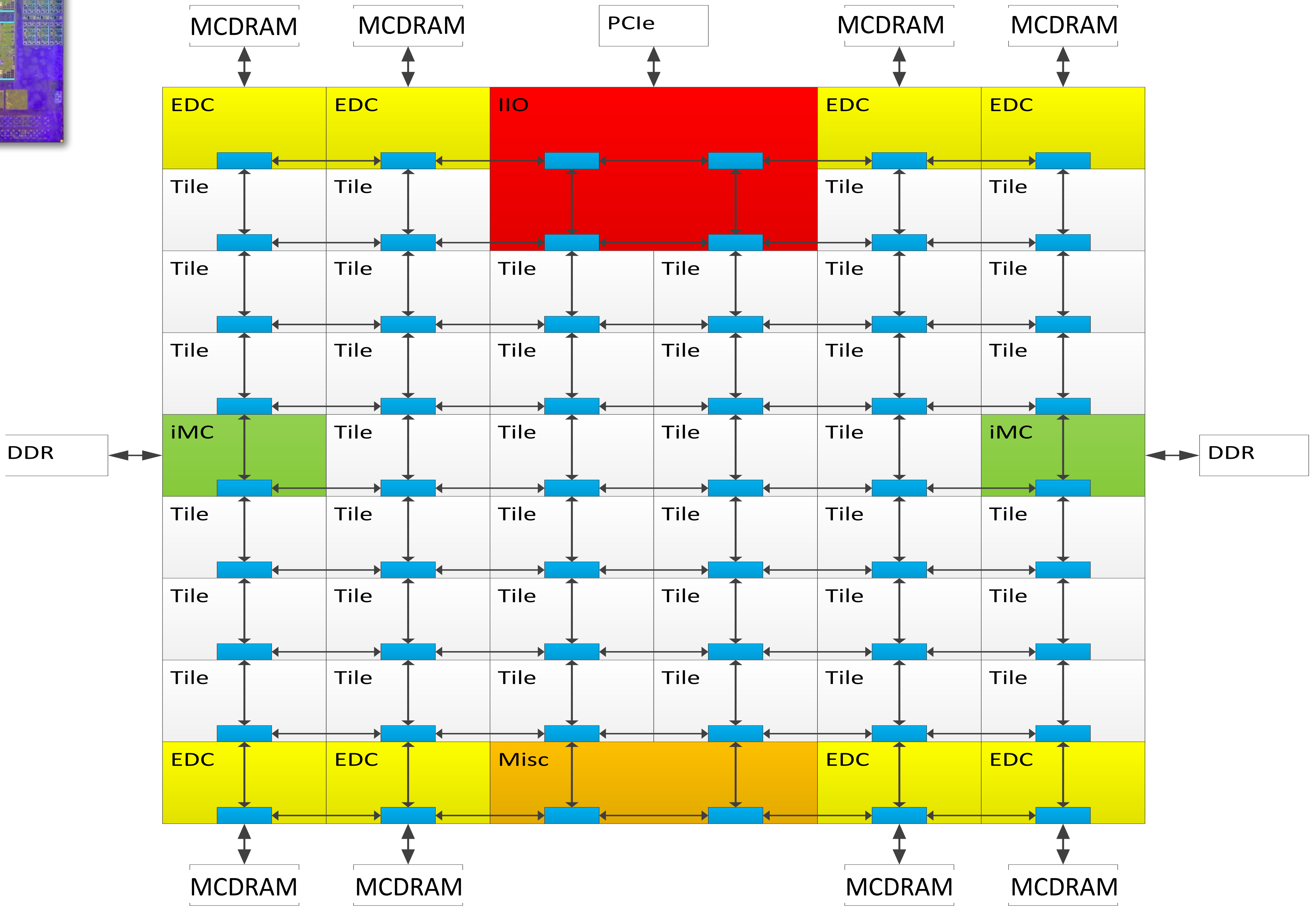
Eight cores



Intel Xeon Phi (Knights Landing)



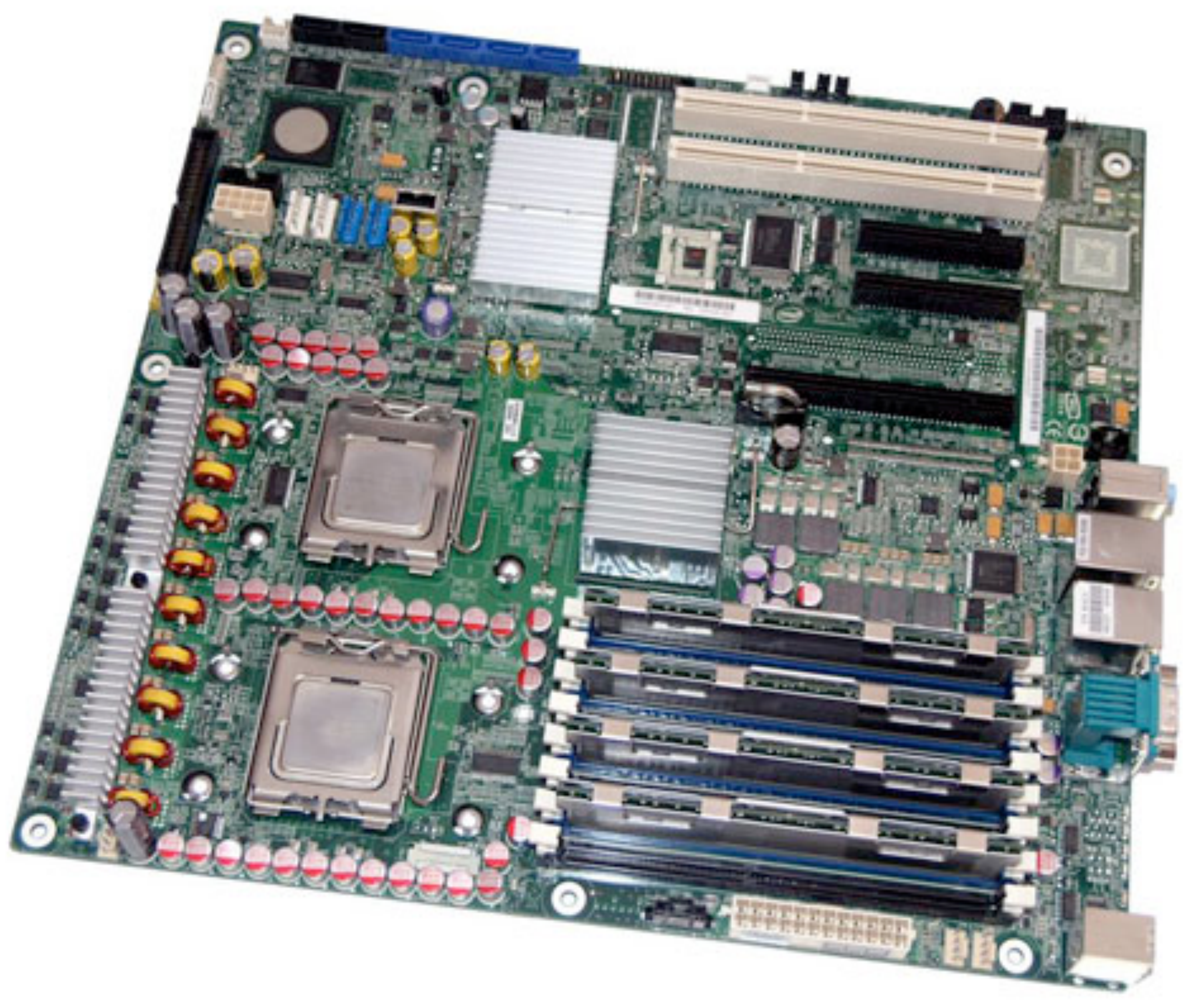
of tiles (2 cores/tile)



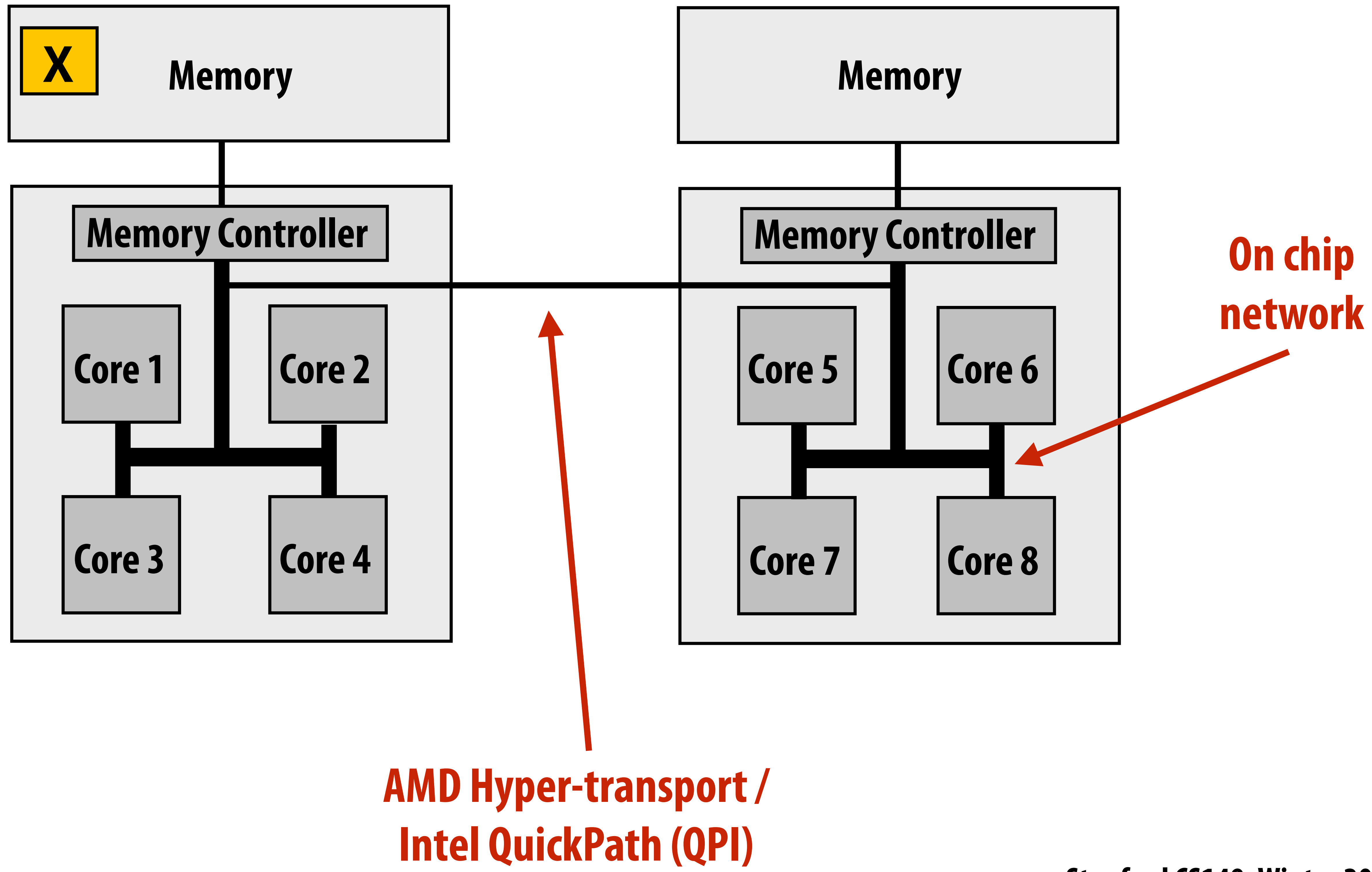
Non-uniform memory access (NUMA)

All processors can access any memory location, but... the cost of memory access (latency and/or bandwidth) is different for different processors

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

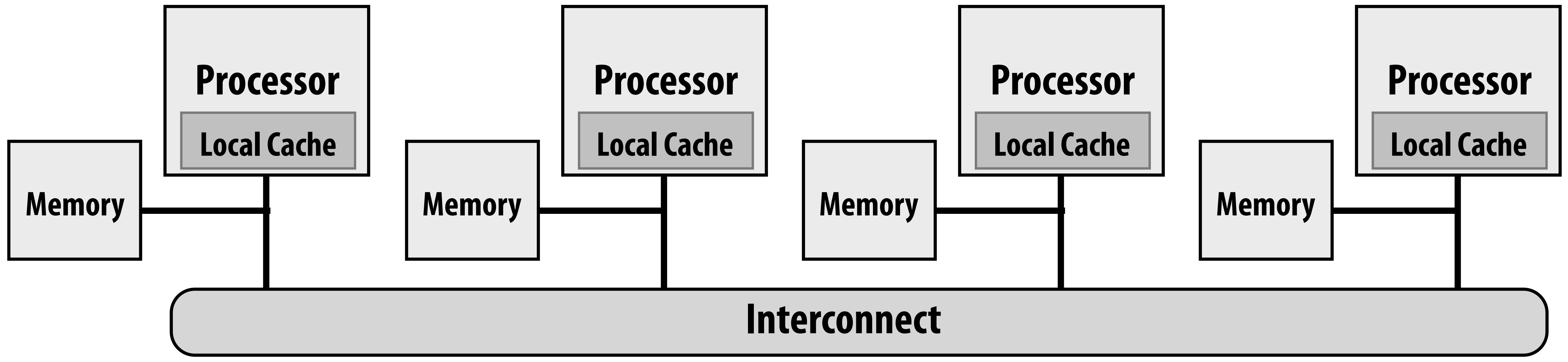


Example: modern dual-socket configuration



Non-uniform memory access (NUMA)

All processors can access any memory location, but... the cost of memory access (latency and/or bandwidth) is different for different processors



- **Problem with preserving uniform access time in a system: scalability**
  - GOOD: costs are uniform, BAD: they are uniformly bad (memory is uniformly far away)
- **NUMA designs are more scalable**
  - Low latency access to local memory
  - Provide high bandwidth to local memory
- **Cost is increased programmer effort for performance tuning**
  - Finding, exploiting locality is important to performance (want most memory accesses to be to local memories)

Summary: shared address space model

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

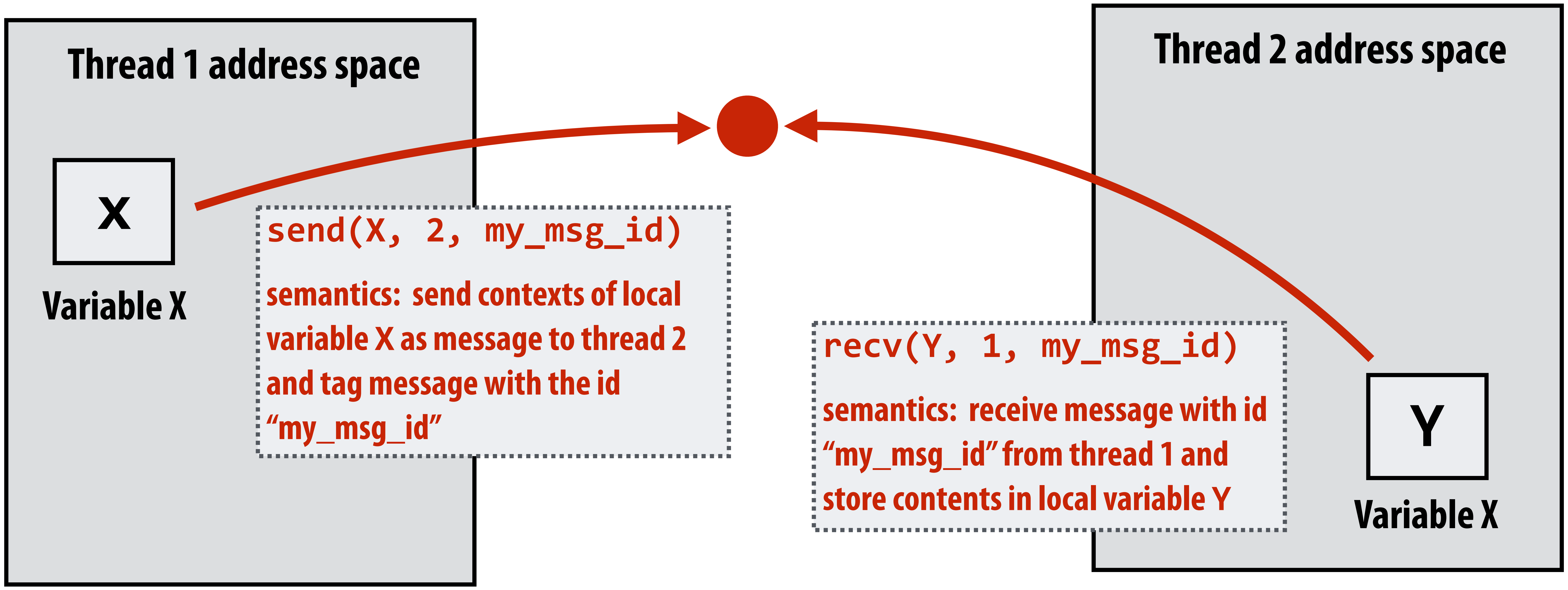
- Requires hardware support to implement efficiently
  - Any processor can load and store from any address (its shared address space!)
  - Even with NUMA, costly to scale
    (one of the reasons why high core count processors are expensive)

* 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
  - send: specifies recipient, buffer to be transmitted, and optional message identifier ("tag")
  - receive: sender, specifies buffer to store data, and optional message identifier
  - Sending messages is the only way to exchange data between threads 1 and 2



```
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)
```

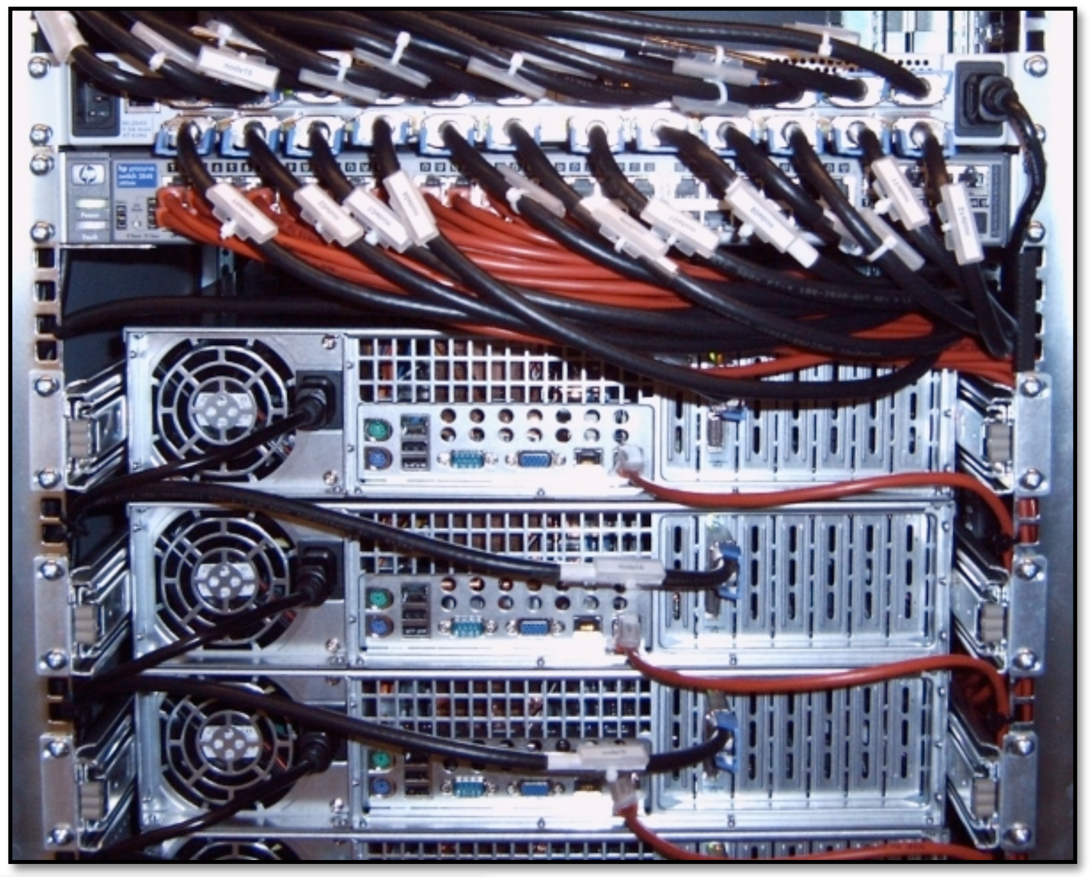
semantics: receive message with id "my_msg_id" from thread 1 and store contents in local variable Y

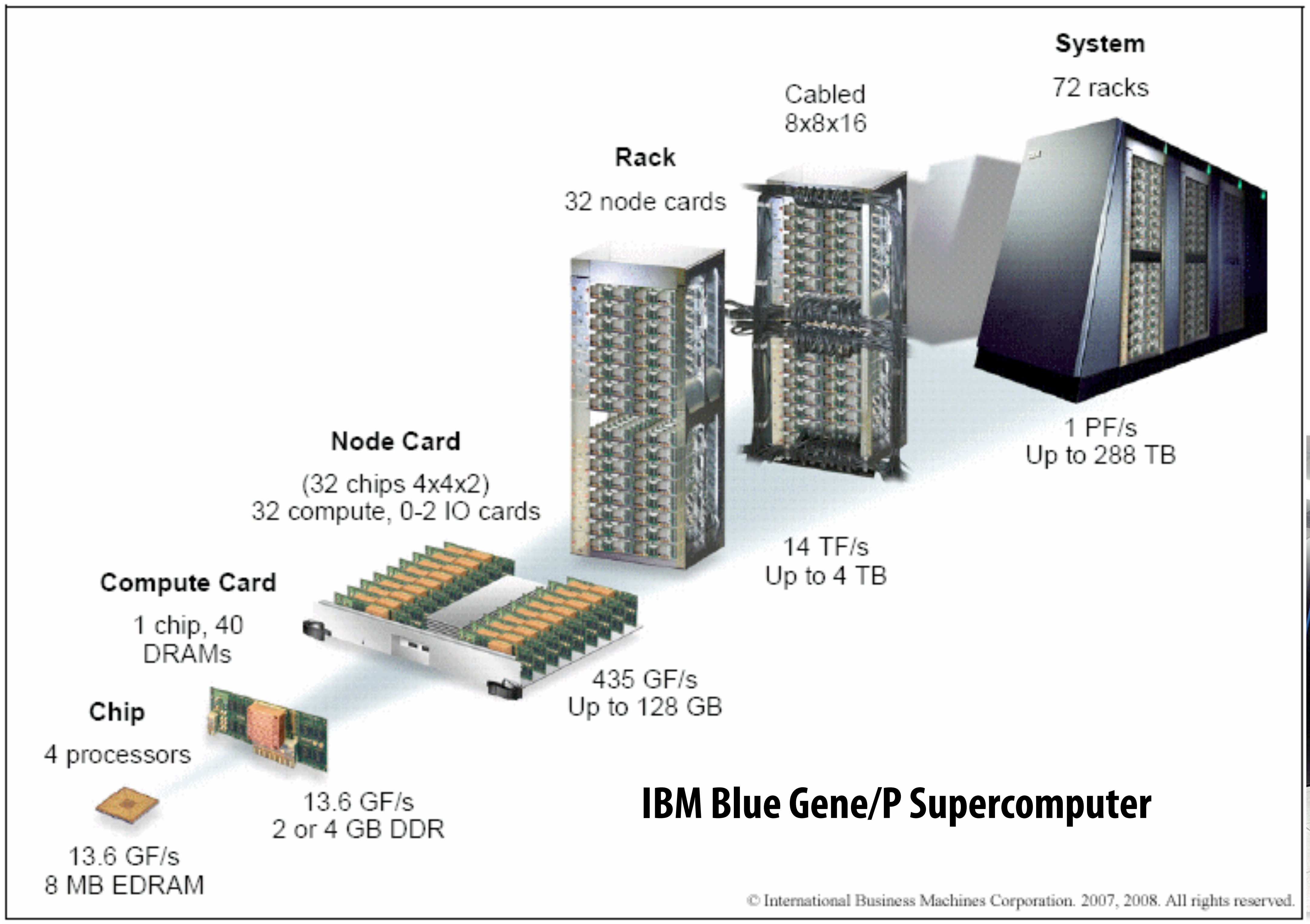
(Communication operations shown in red)

Illustration adopted from Culler, Singh, Gupta

Message passing (implementation)

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


Cluster of workstations (Infiniband network)



IBM Blue Gene/P Supercomputer

Image credit: IBM

Caveat: the correspondence between programming models and machine types is fuzzy

- **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 solutions)**
  - Mark all pages with shared variables as invalid
  - 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?**

The data-parallel model

What are programming models for?

Programming models serve to impose structure on programs!

- **Shared address space: very little structure to communication**
  - All threads can read and write to all shared variables
  - Pitfall: due to implementation: not all reads and writes have the same cost (and that cost is often not apparent in program code)

- **Message passing: highly structured communication**
  - All communication occurs in the form of messages (programmer can read program and see where the communication is—the sends and receives)

- **Data-parallel: very rigid computation structure**
  - Programs perform same function on different data elements in a collection

Data-parallel model

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

- **NumPy is another good example: C = A + B (A, B, and C are vectors of same length)**

- **Today: often takes form of SPMD programming**
  - `map(function, collection)`
  - Where `function` is applied to each element of collection <u>independently</u>
  - `function` may be a complicated sequence of logic (e.g., a loop body)
  - Synchronization is implicit at the end of the map (`map` returns when function has been applied to all elements of `collection`)

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];
   }
}
```

Think of loop body as `function` (from the previous slide)

`foreach` construct is a map

Given this program, it is reasonable to think of the program as mapping the loop body onto each element of the arrays X and Y.

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

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

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 `function`

`foreach` construct is a map

Collection is 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 collections.)

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 `function`

`foreach` construct is a map

Collection is 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

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

Data parallelism: a more "pure" approach

Note: this is not ISPC syntax (more of Kayvon's made up syntax)

Main program:

```
const int N = 1024;

stream<float> x(N);  // sequence (a "stream")
stream<float> y(N);  // sequence (a "stream")

// initialize N elements of x here...

// map function absolute_value onto streams
absolute_value(x, y);
```

"Kernel" definition:

```

void absolute_value(float x, float y)
{
   if (x < 0)
      y = -x;
   else
      y = x;
}
```

Data-parallelism expressed in this functional form is sometimes referred to as the stream programming model

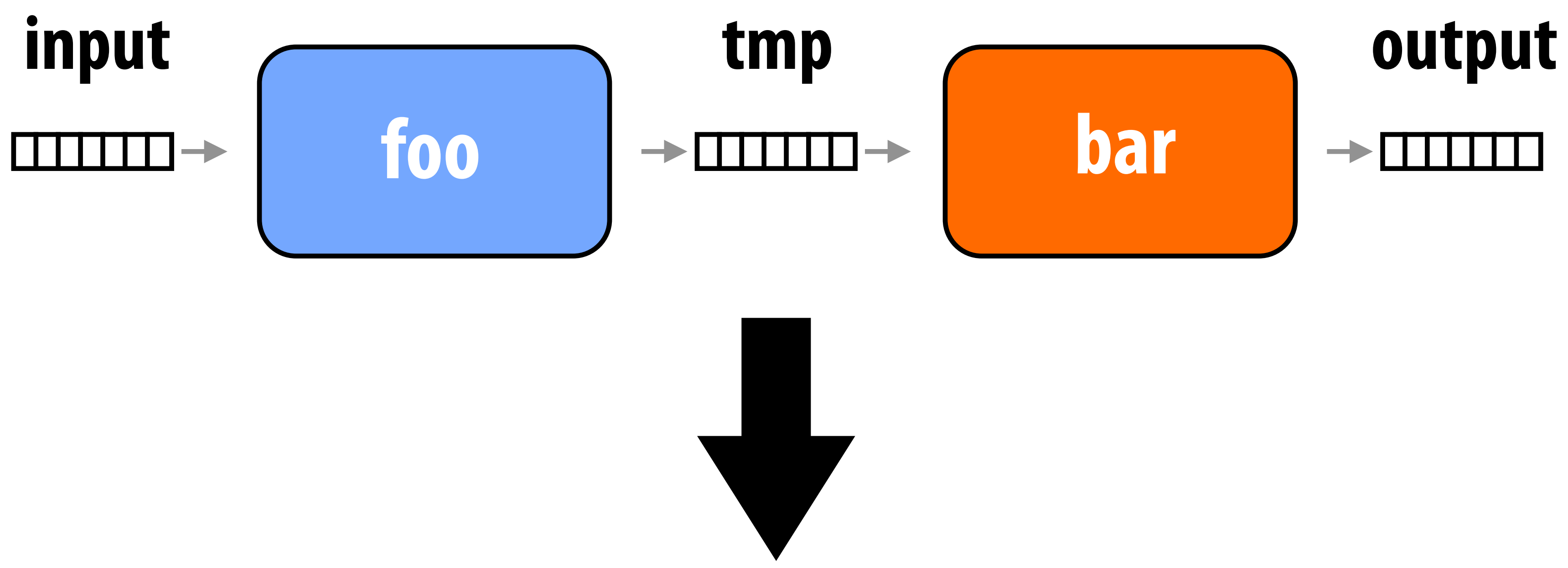
Streams: sequences of elements. Elements in a stream can be processed independently

Kernels: side-effect-free functions. Operate element-wise on collections

Think of the inputs, outputs, and temporaries for each kernel invocation as forming a private per-invocation address space

Stream programming benefits

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

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



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

Global-scale program dependencies are known by compiler (enables compiler to perform aggressive optimizations that require global program analysis):

Independent processing on elements, kernel functions are side-effect free:
- **Optimization: parallelize kernel execution**
- **Application cannot write a program that is non-deterministic under parallel execution**

Inputs/outputs of each invocation known in advance: prefetching can be employed to hide latency.

Producer-consumer dependencies are known in advance: Implementation can be structured so outputs of first kernel are immediately processed by second kernel. (The values are stored in on-chip buffers/caches and never written to memory! Saves bandwidth!)

Stream programming drawbacks

```
const int N = 1024;
stream<float> input(N/2);
stream<float> tmp(N);
stream<float> output(N);

// double length of stream by replicating
// all elements 2x
stream_repeat(2, input, tmp);

absolute_value(tmp, output);
```

Need library of operators to describe complex data flows (see use of `repeat` operator at left to obtain same behavior as indexing code below)

My experience: cross fingers and hope compiler is intelligent enough to generate code below from program at left.

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

Kayvon's experience:

This is the achilles heel of all "proper" data-parallel/stream programming systems.

"If I just had one more operator"...

Gather/scatter: two key data-parallel communication primitives

Map absolute_value onto stream produced by gather:

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

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

Map absolute_value onto stream, scatter results:

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

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

ISPC equivalent:

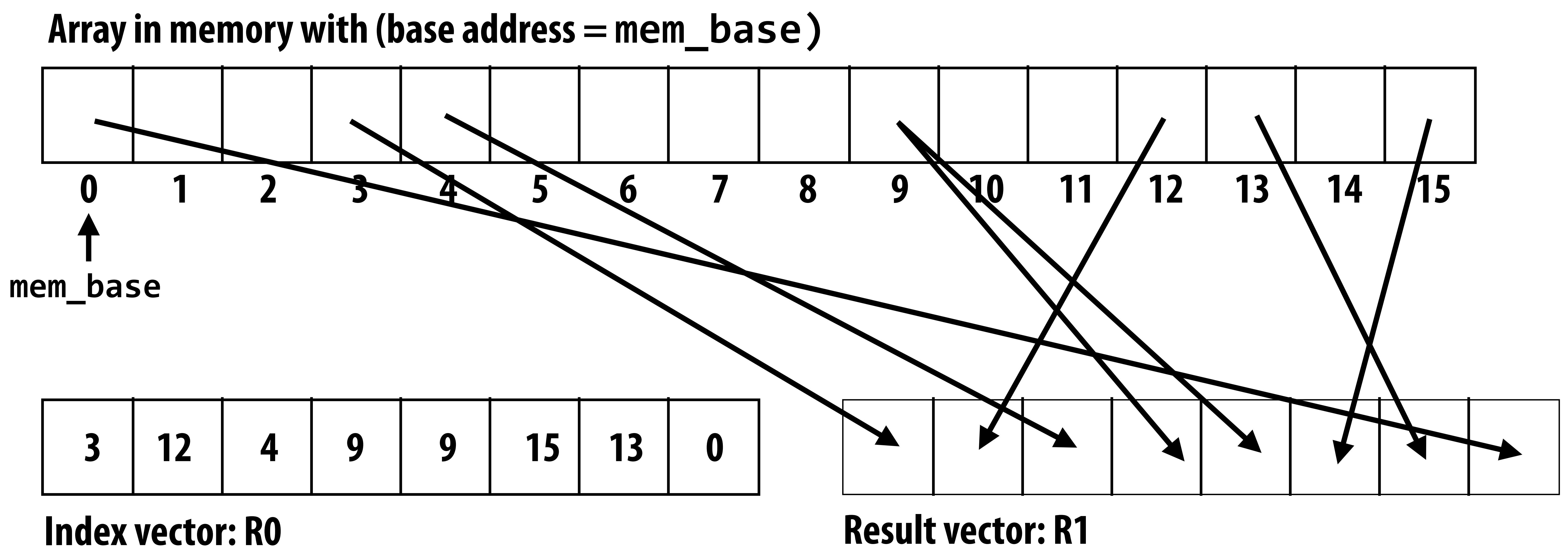
```
export void absolute_value(
   uniform float N,
   uniform float* input,
   uniform float* output,
   uniform int* indices)
{
   foreach (i = 0 ... n)
   {
      float tmp = input[indices[i]];
      if (tmp < 0)
         output[i] = -tmp;
      else
         output[i] = tmp;
   }
}
```

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 structure: 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 strictly <u>enforce</u> this structure**
  - ISPC, OpenCL, CUDA, etc.
  - They choose flexibility/familiarity of imperative C-style syntax over the safety of a more functional form: it's been their key to their adoption
  - Opinion: sure, functional <u>thinking</u> is great, but programming systems sure should impose structure to facilitate achieving high-performance implementations, not hinder them

Summary

Summary

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

- They provide abstractions that permit multiple valid implementations.

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

Summary

Restrictions imposed by these abstractions are designed to:

1. **Reflect realities of parallelization and communication costs to programmer (help a programmer write efficient programs)**
   - Shared address space machines: hardware supports any processor accessing any address
   - Messaging passing machines: hardware may accelerate message send/receive/buffering
   - Desirable to keep "abstraction distance" low so programs have predictable performance, but want abstractions to be high enough for code flexibility/portability

2. **Provide useful information to implementors of optimizing compilers/runtimes/hardware to help them efficiently implement programs using these abstractions**

We discussed three parallel programming models

- **Shared address space**
  - Communication is unstructured, implicit in loads and stores
  - Natural way of programming (extension of single-threaded programming), but programmer can shoot themselves in the foot easily
    - Program might be correct, but not perform well
- **Message passing**
  - Structure all communication as messages
  - Often harder/more tedious to get first correct program than shared address space
  - Structure often helpful in getting to <u>first correct, scalable</u> program
- **Data parallel**
  - Structure computation as a big "map" over a collection
  - Assumes a shared address space from which to load inputs/store results, but severely limits communication between iterations of the map (goal: preserve independent processing of iterations)
  - Modern embodiments encourage, but don't enforce, this structure

Modern practice: mixed programming models

- **Use shared address space programming within a multi-core node of a cluster, use message passing between nodes**
  - Very, very common in practice
  - Use convenience of shared address space where it can be implemented efficiently (within a node), require explicit communication elsewhere

- **Data-parallel-ish programming models support shared-memory style synchronization primitives in kernels**
  - Permit limited forms of inter-iteration communication (e.g., CUDA, OpenCL)

- **In a future lecture… CUDA/OpenCL use data-parallel model to scale to many cores, but adopt shared-address space model allowing threads running on the same core to communicate.**

Questions to consider

- Programming models enforce different forms of structure on programs. What are the benefits of data-parallel structure?

- With respect to the goals of efficiency/performance… what do you think are problems of adopting a very high level of abstraction in a programming system?
  - What about potential benefits?

- Choose a popular parallel programming system (for example Hadoop, Spark, or Cilk) and try and describe its programming model (how are communication and execution expressed?)