Lecture 3:

Parallel Programming Abstractions

(and their corresponding HW/SW

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Fall 2025

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

An example:
Programming with ISPC

ISPC

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

- http://ispc.github.com/

Recall: example program from

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

```
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: $\sin(x) = x - x^3/3! + x^5/5! -$

C++ code:

```
#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:

```
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;
   }
```

SPMD programming abstraction:

Call to ISPC function spawns "gang" of ISPC
"program instances"

sin(x) in ISPC

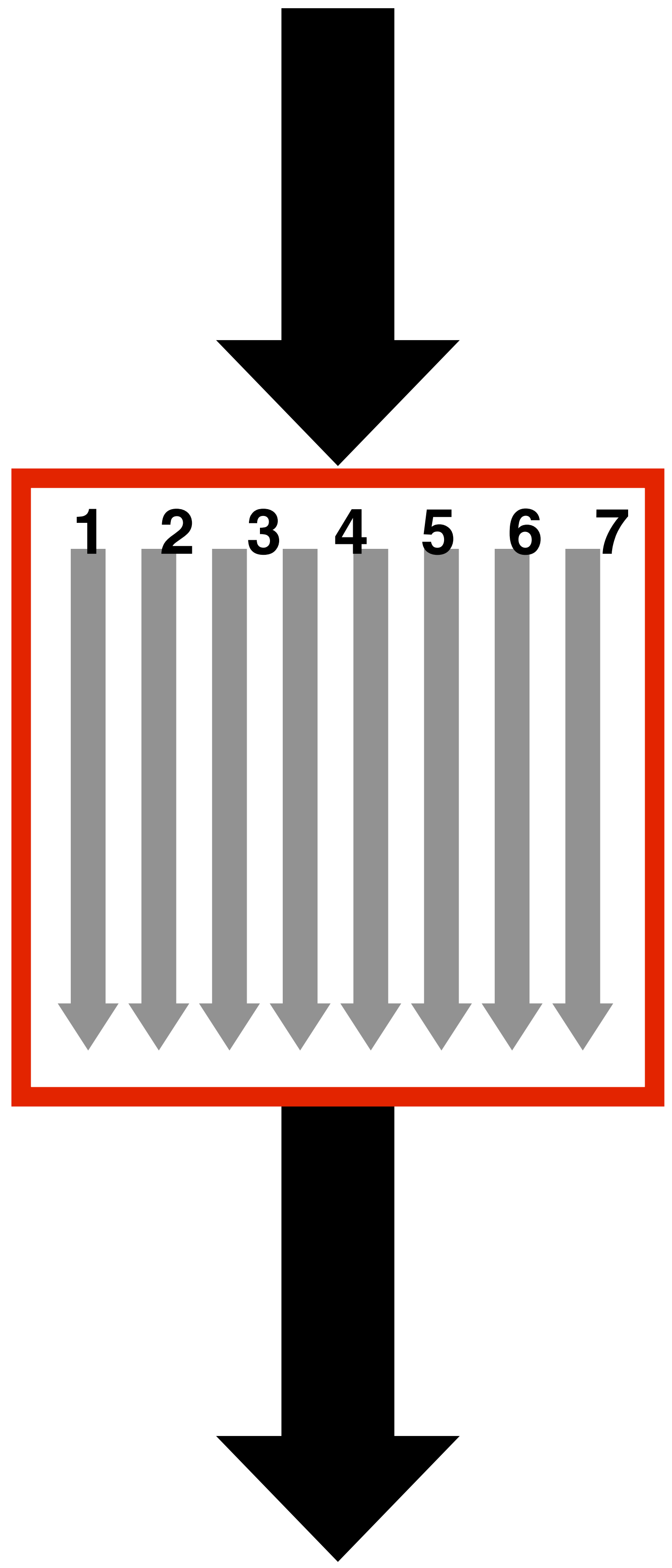
Compute $\sin(x)$ using Taylor expansion: $\sin(x) = x - x^3/3! + x^5/5! -$

C++ code:

```
#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"



Sequential execution

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

**sinx() returns.
Completion of ISPC program instances.**

Sequential execution (C code)

In this illustration programCount = 8

sin(x) in ISPC

"Interleaved" assignment of array elements to program instances

C++ code:

```
#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:

```
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;
   }
```

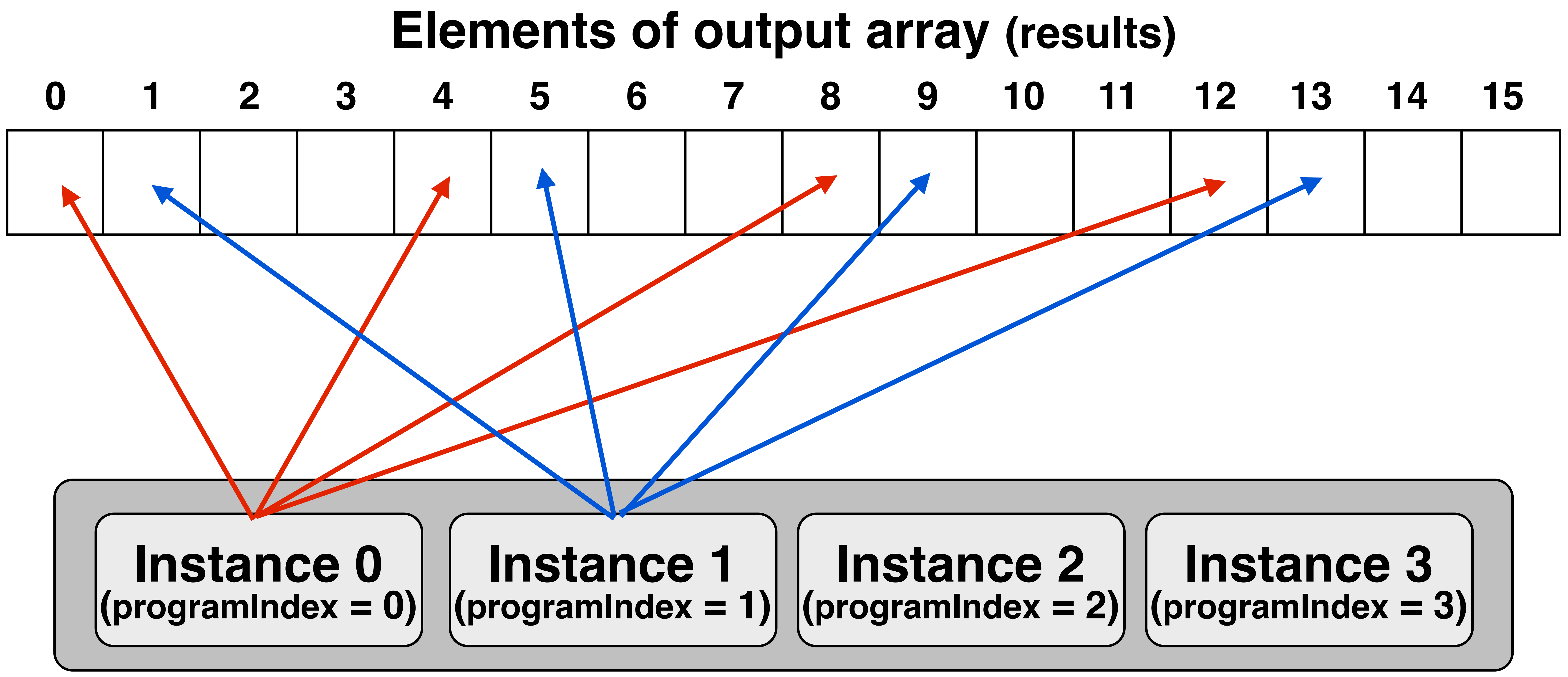
ISPC Keywords:

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

Interleaved assignment of program instances to loop


Elements of output array (results)
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Instance 0 (programIndex = 0)
Instance 1 (programIndex = 1)
Instance 2 (programIndex = 2)
Instance 3 (programIndex = 3)

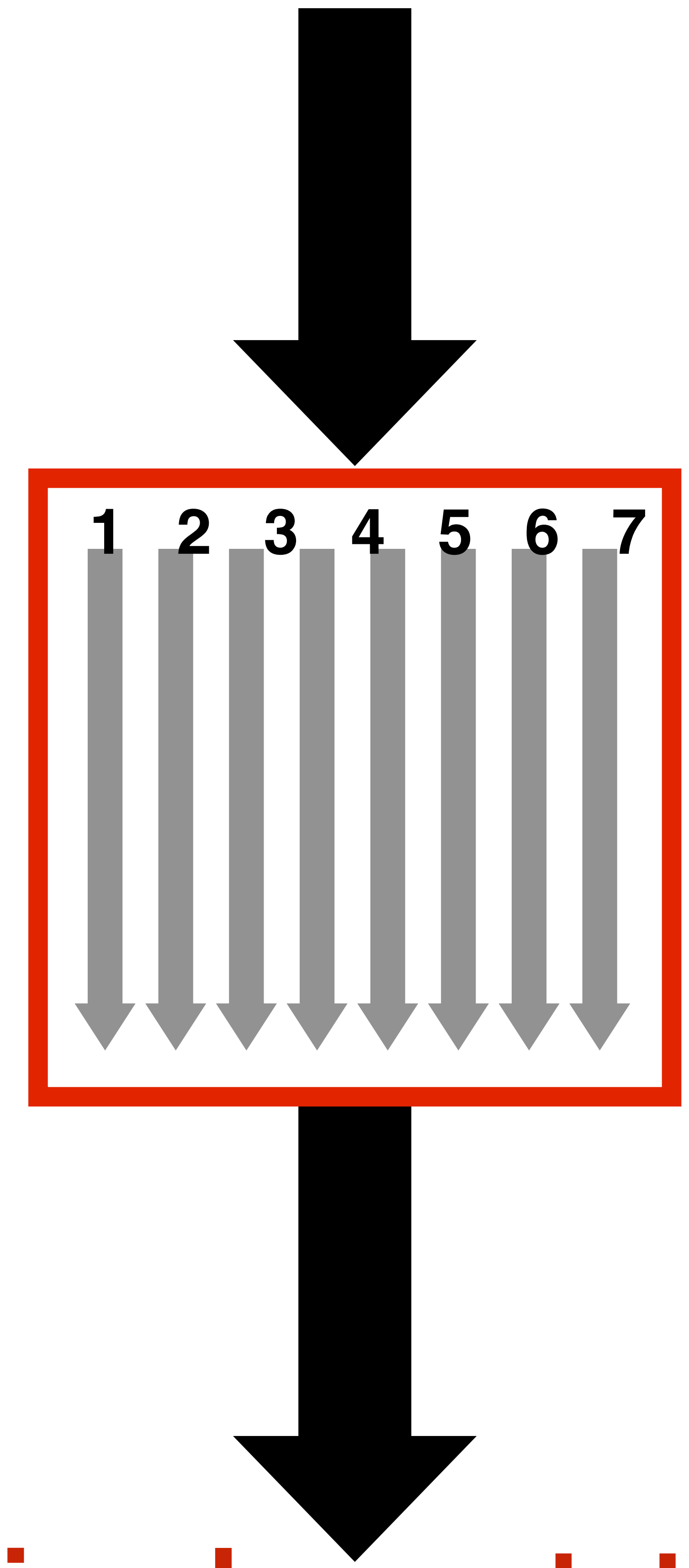
“Gang” of ISPC program instances

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

ISPC implements the gang abstraction using SIMD

C++ code:

```
#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

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

sinx() returns.
Completion of ISPC program instances.

Sequential execution
(C code)

SPMD programming abstraction:

Call to ISPC function spawns "gang" of ISPC "program instances"

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

sin(x) in ISPC: version 2

"Blocked" assignment of elements to

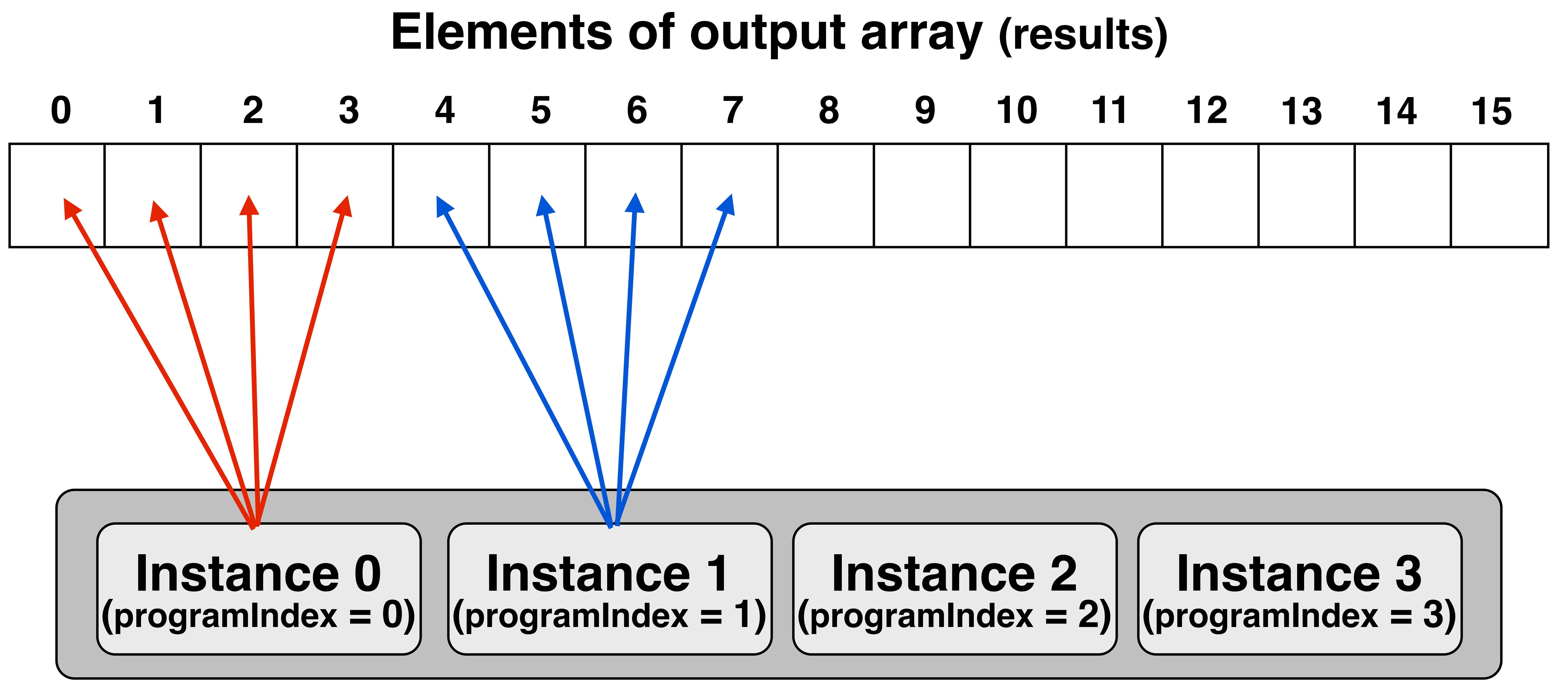
C++ code:

```
#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:

```
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;
  }
```

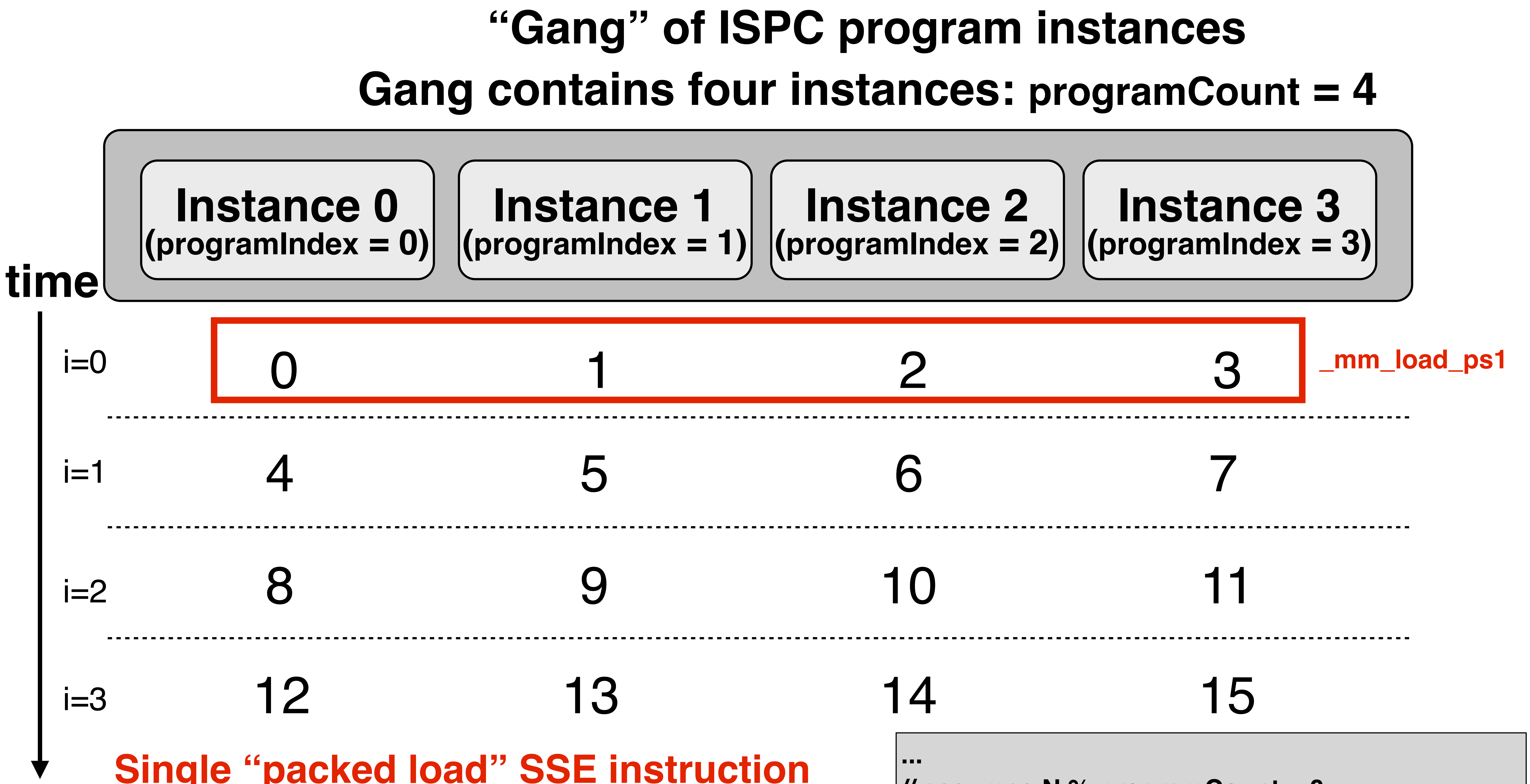
Blocked assignment of program instances to loop iterations



“Gang” of ISPC program instances

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

Schedule: interleaved



"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

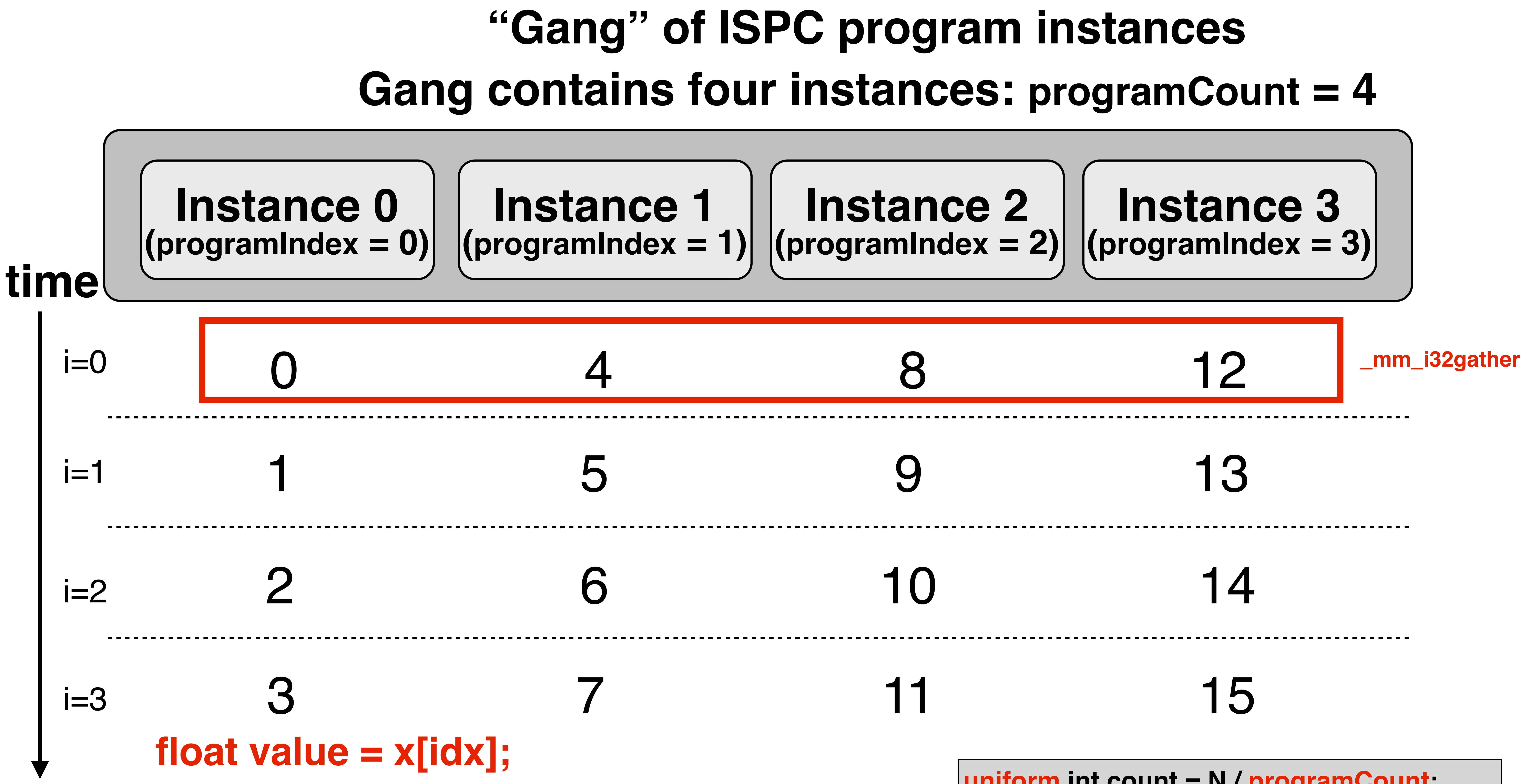
Single "packed load" SSE instruction (_mm_load_ps1) efficiently implements:

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
0
4
8
12
_mm_i32gather
i=1
1
5
9
13
i=2
2
6
10
14
i=3
3
7
11
15

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

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

C++ code:

```
#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:

```
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[idx] = value;
  }
```

foreach: key ISPC language construct

- foreach declares parallel loop iterations
  - Programmer says: these are the iterations the instances in a gang must cooperatively perform
- ISPC implementation assigns iterations to program instances in

ISPC: abstraction vs.

- **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 abstraction**
  - **Program is written in terms of this abstraction**

- **Single instruction, multiple data (SIMD) implementation**
  - **ISPC compiler emits vector instructions (SSE4 or AVX) 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.)**

ISPC discussion: sum

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)

ISPC discussion: sum

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)

```
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 good**

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.

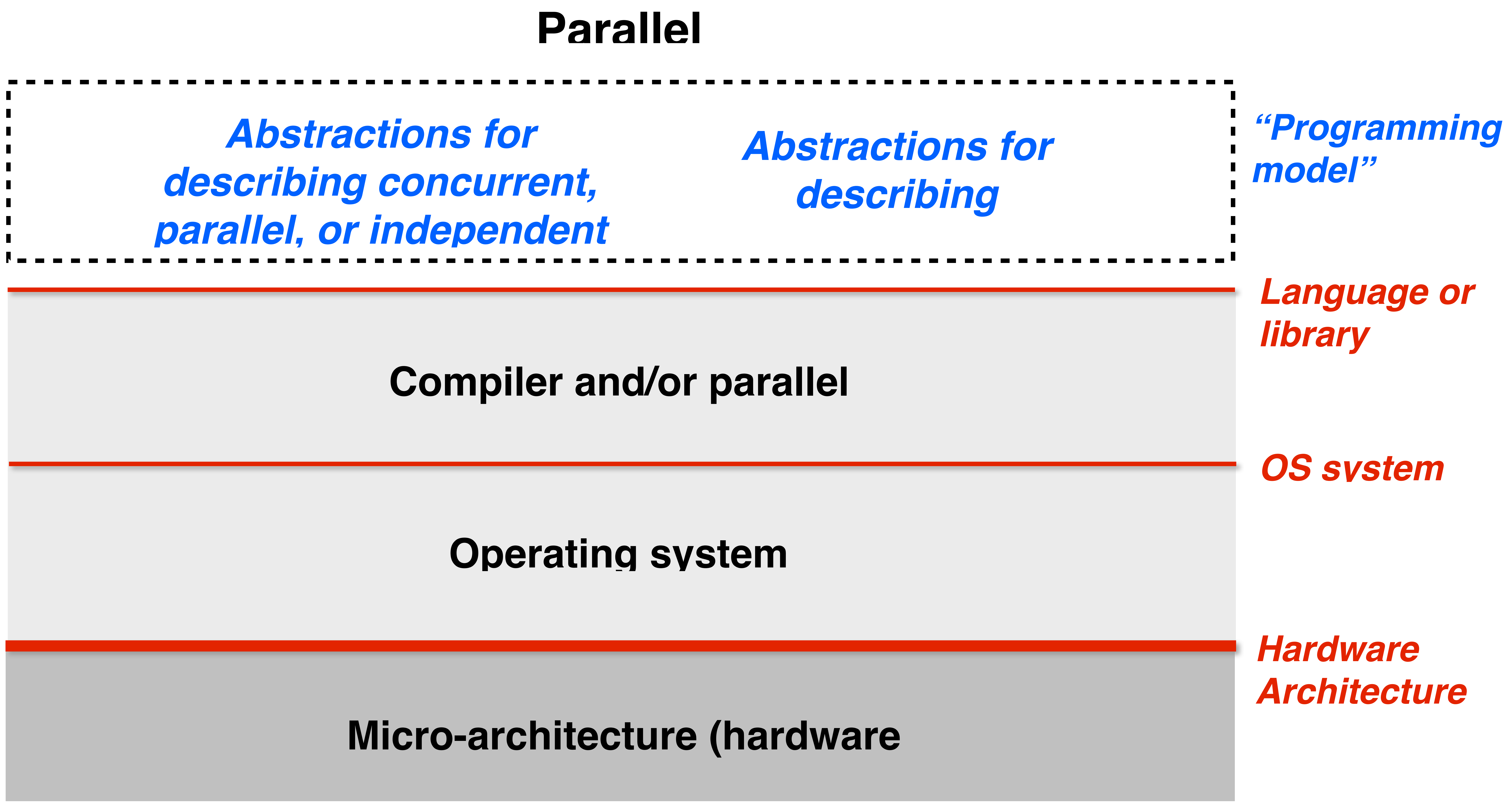
The second half of today's

- **Four parallel programming models**
  - That differ in communication abstractions presented to the programmer
  - Programming models influence how programmers think when writing programs

- **Four machine architectures**
  - Abstraction presented by the hardware to low-level software
  - Typically reflect hardware implementation's capabilities

- **We'll focus on differences in communication and cooperation**

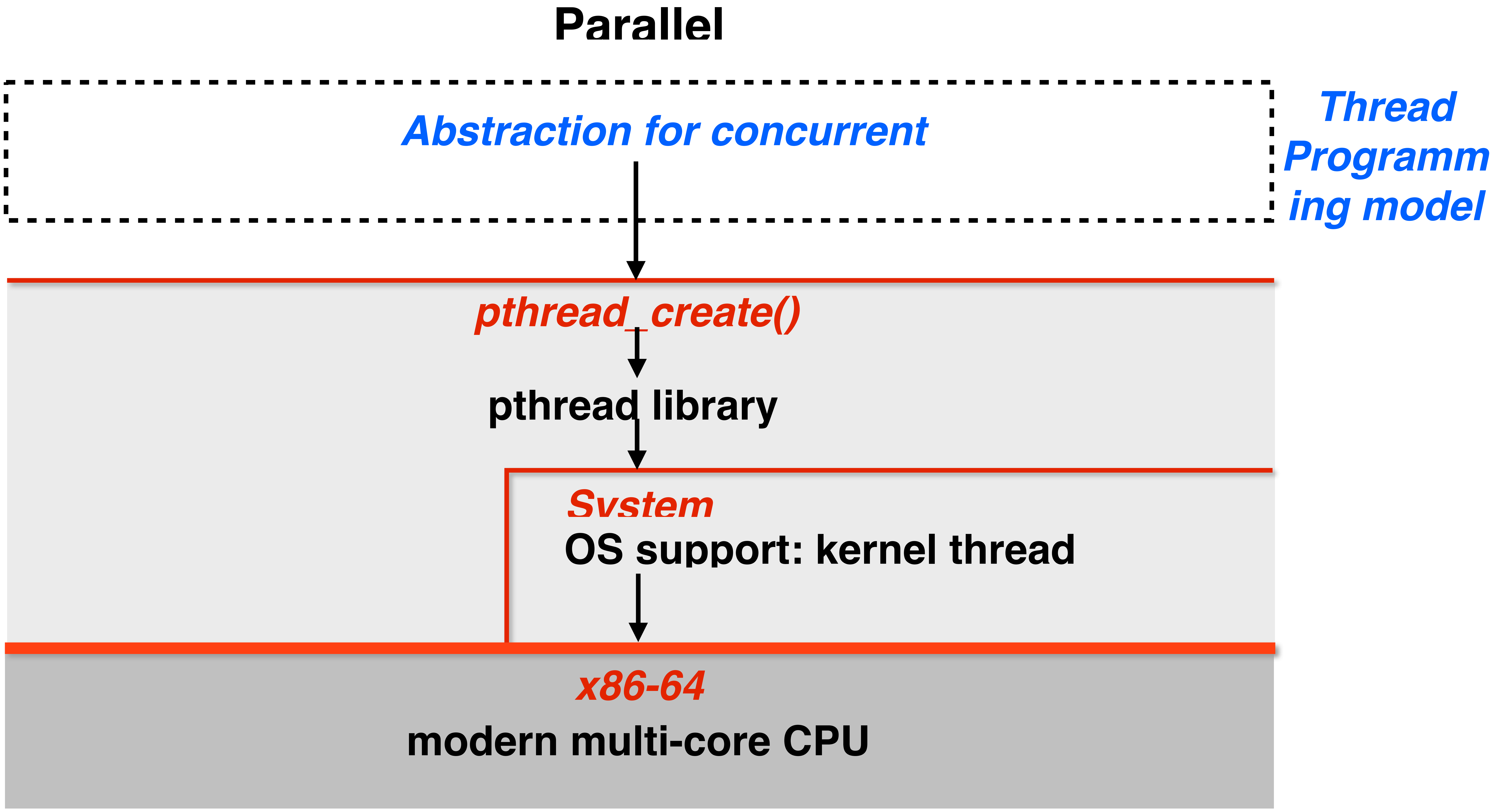
System layers: interface,


Parallel
Abstractions for describing concurrent, parallel, or independent
Abstractions for describing
"Programming model"
Language or library
Compiler and/or parallel
OS system
Operating system
Hardware Architecture
Micro-architecture (hardware

Blue italic text: abstraction/ concept

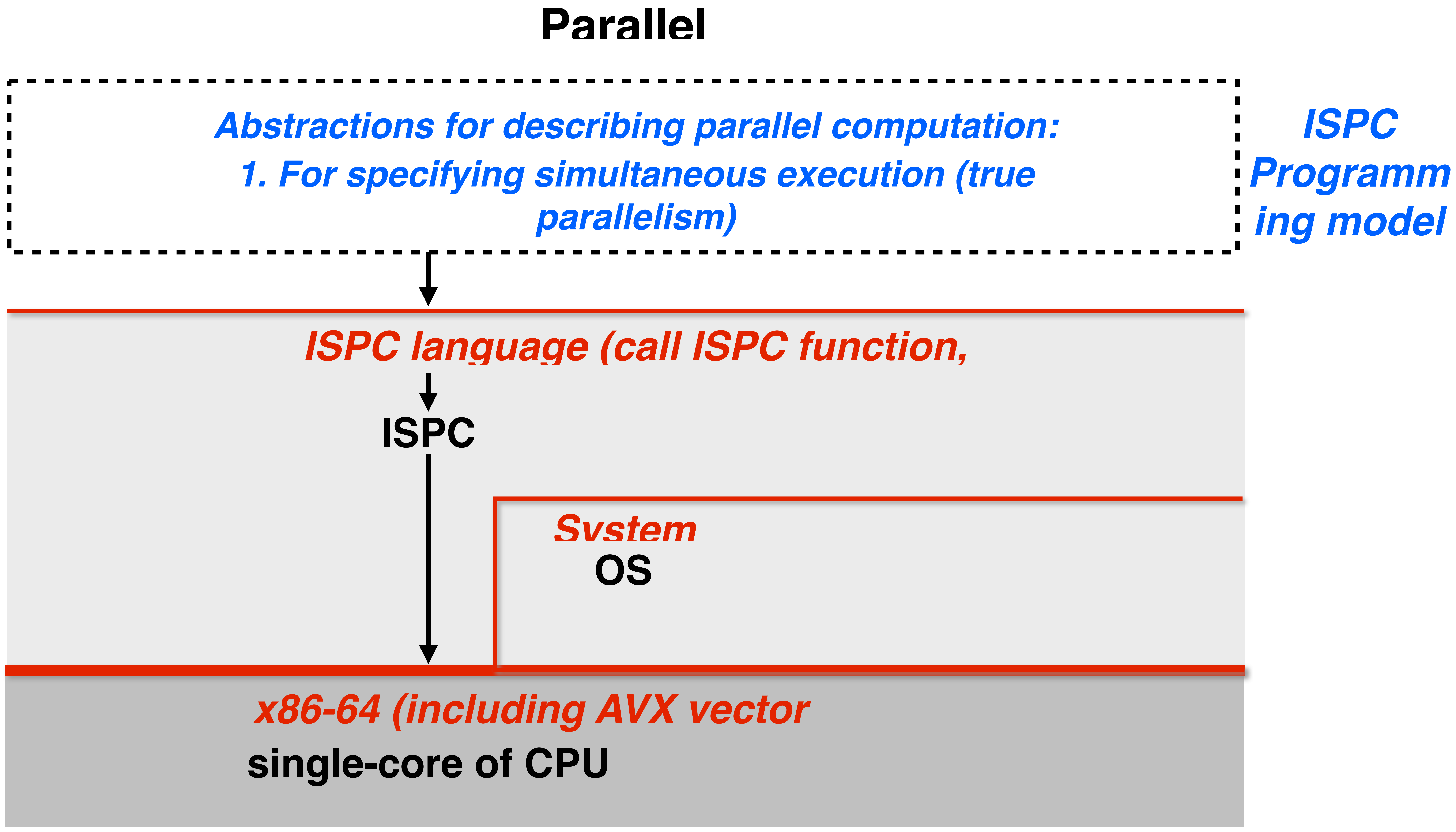
Black text: system

Example: expressing parallelism



Blue italic text: abstraction/ concept
Black text: system

Example: expressing parallelism


Parallel
Abstractions for describing parallel computation:
1. For specifying simultaneous execution (true parallelism)
ISPC Programming model
ISPC language (call ISPC function,
ISPC
System
OS
x86-64 (including AVX vector
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.

Four models of communication (abstractions)

1. Shared address space
2. Message passing
3. Data parallel
4. Systolic arrays

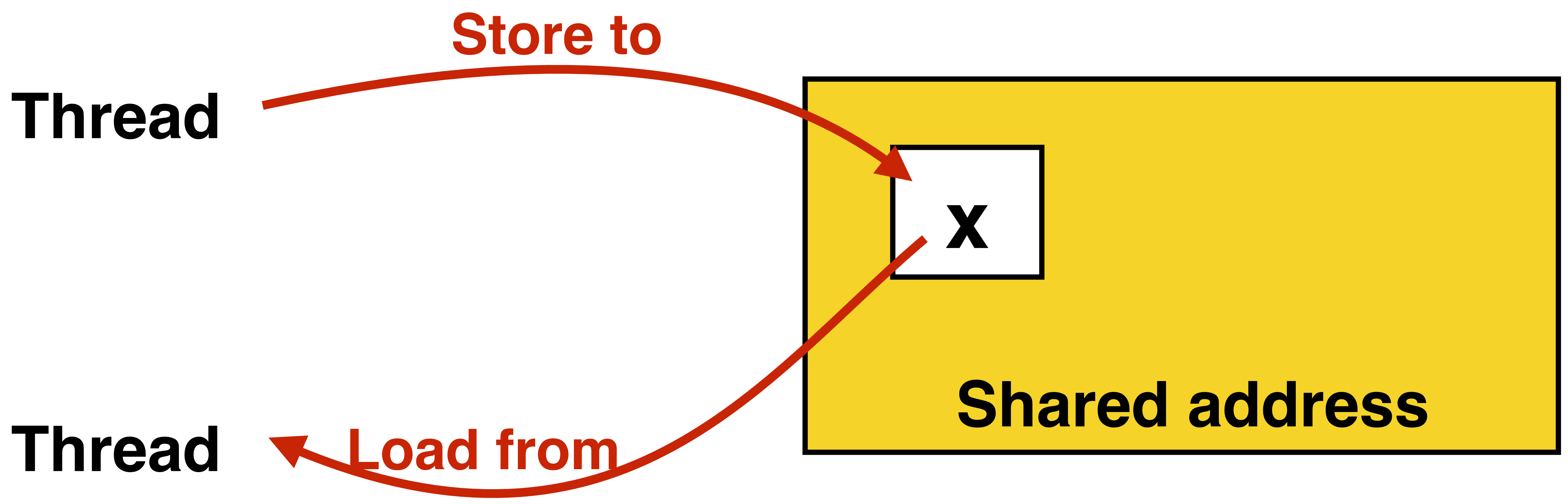
Shared address space model of

Shared address space model

- 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

(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

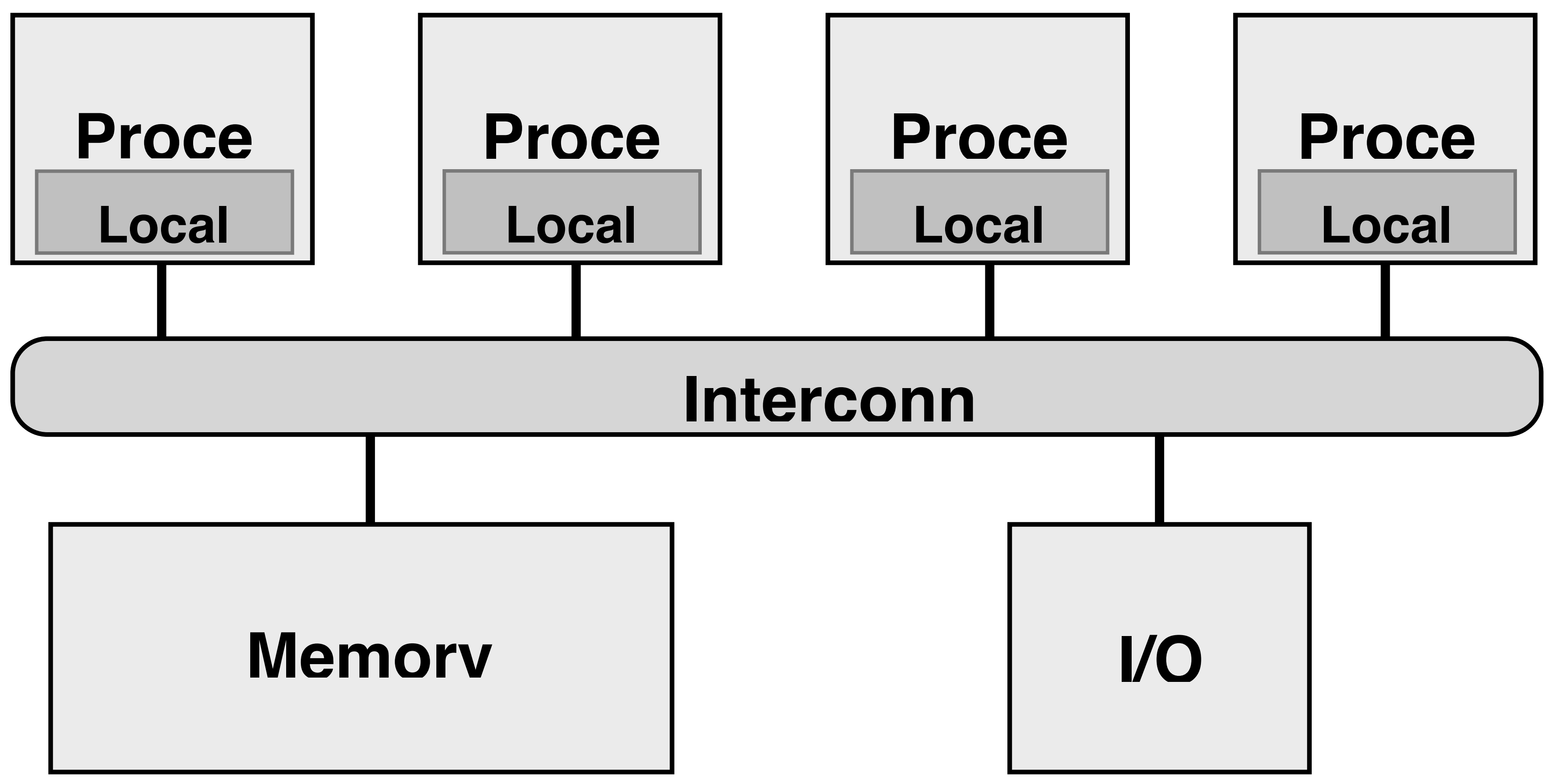
Shared address space model

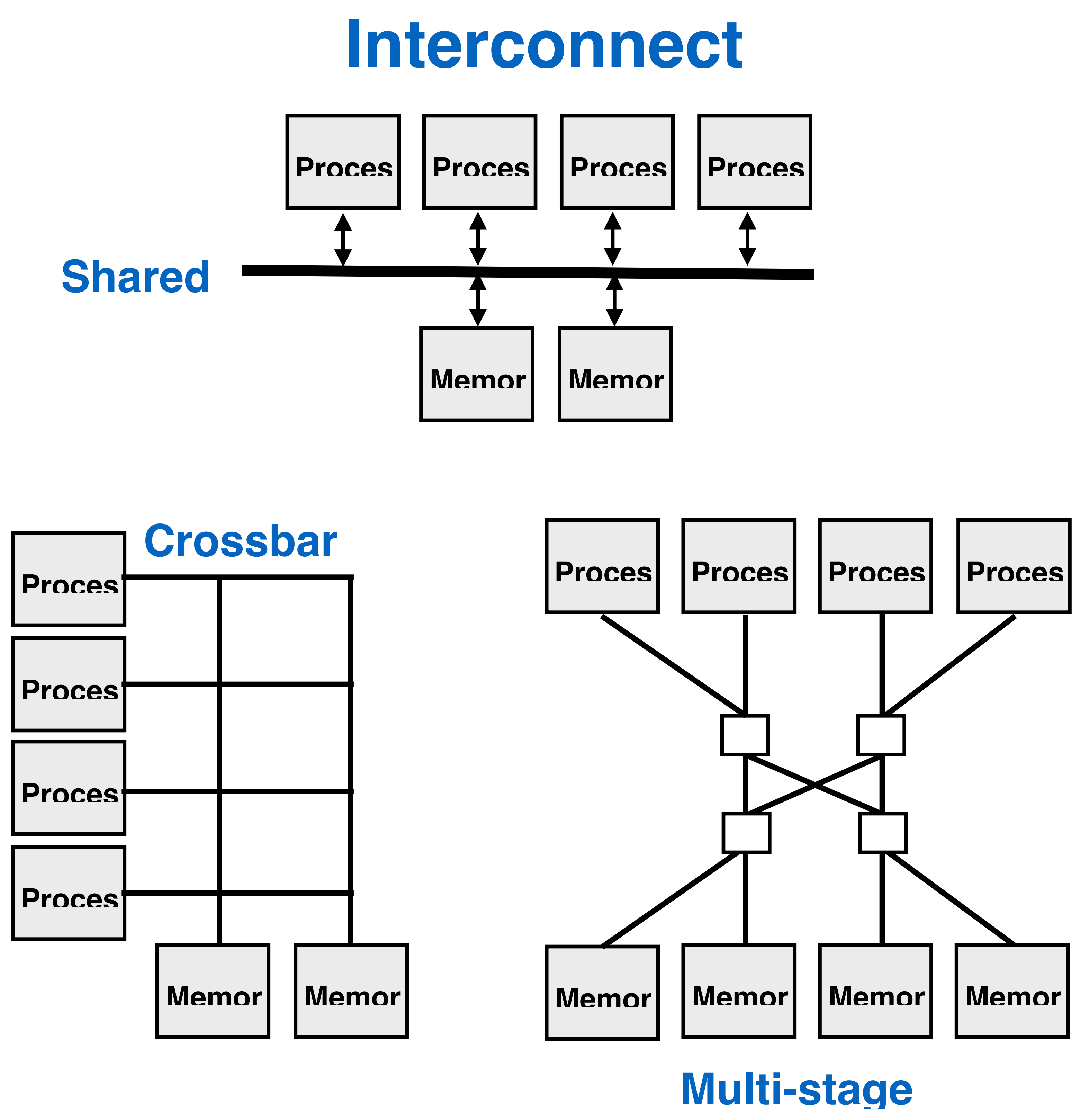
- Threads communicate by:
  - Reading/writing to shared variables
    - Inter-thread communication is implicit in memory operations
    - Thread 1 stores to X
    - Later, thread 2 reads X (and observes update of value by thread 1)
  - Manipulating synchronization primitives
    - e.g., ensuring mutual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

HW implementation of a shared

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



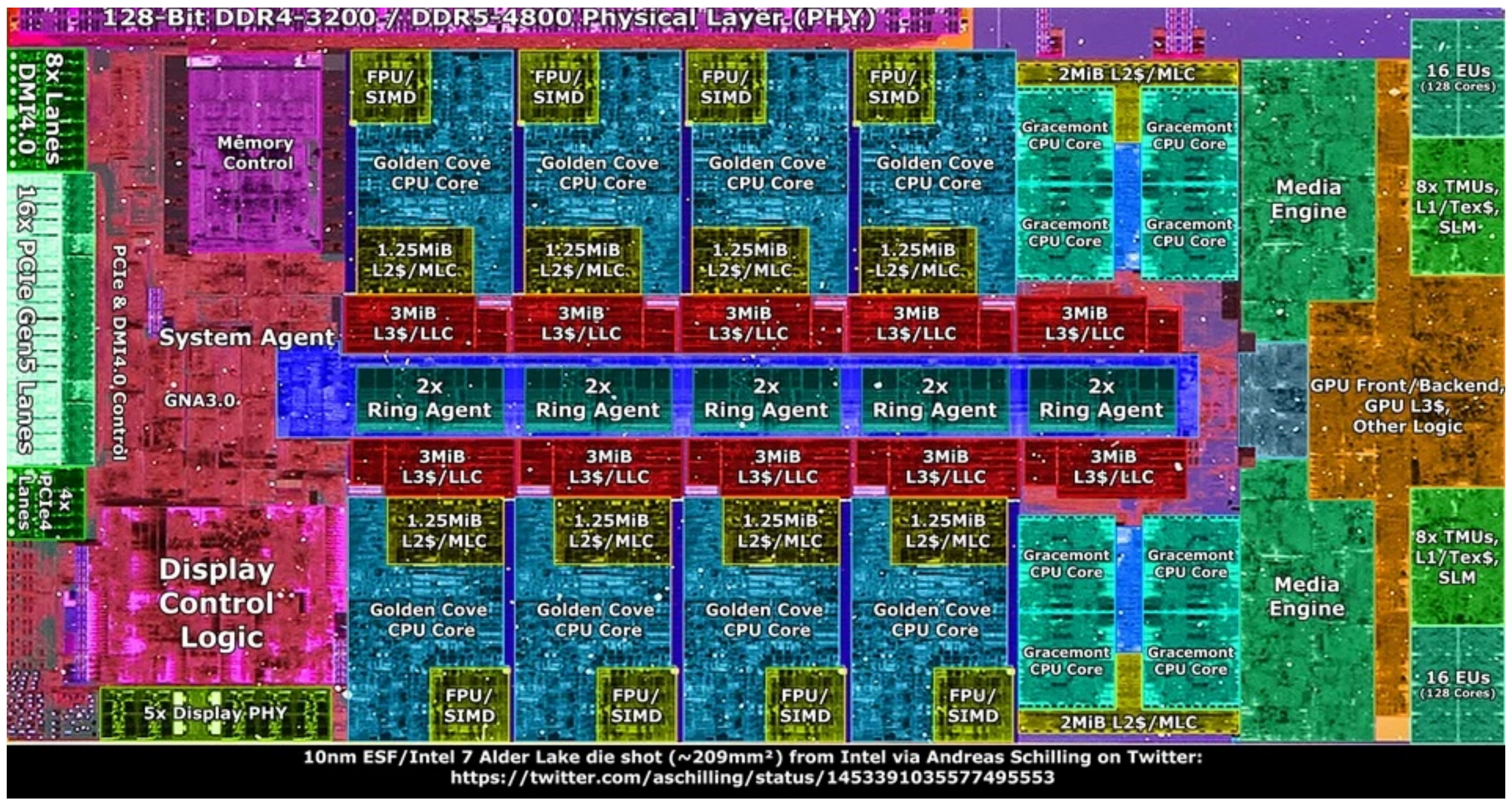


Symmetric (shared-memory) multi-processor (SMP):

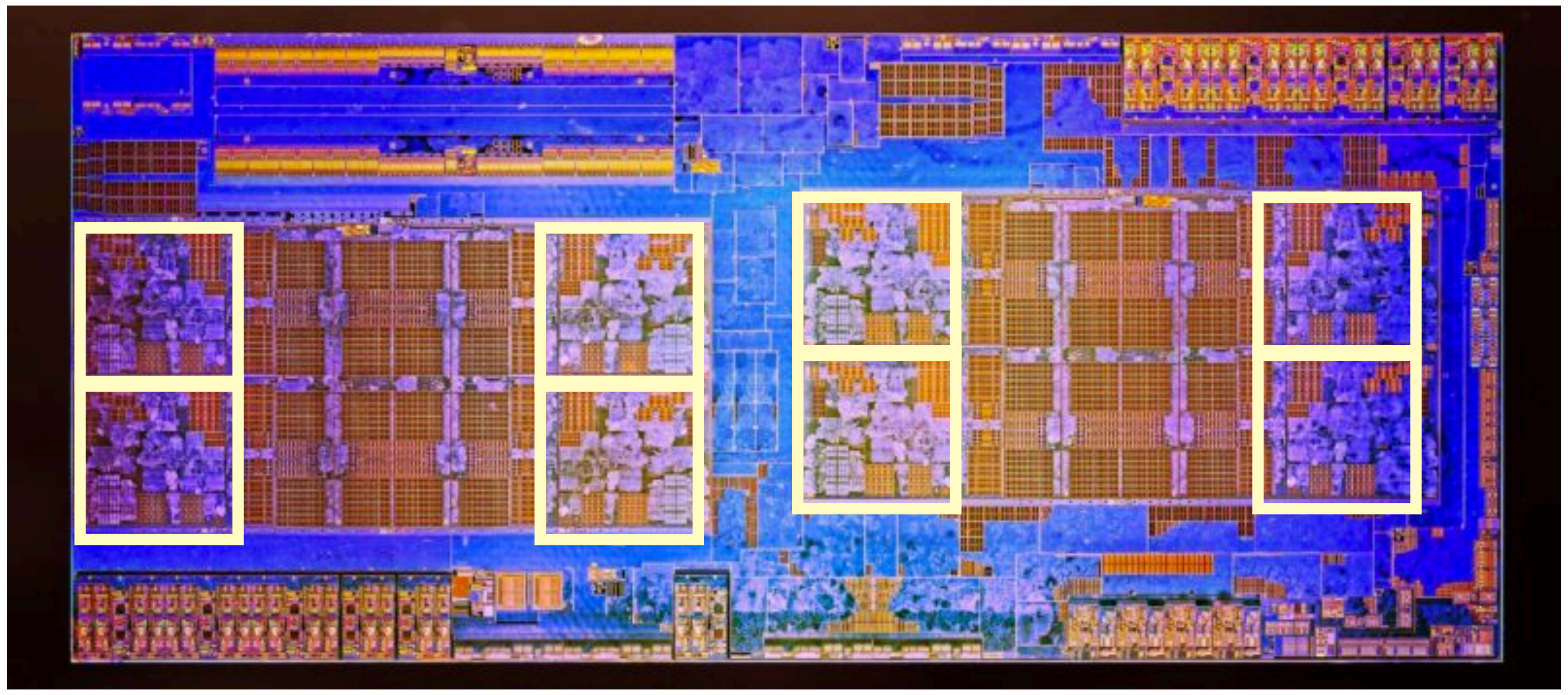
* caching introduces non-uniform access times, but we'll talk about that later

Shared address space HW

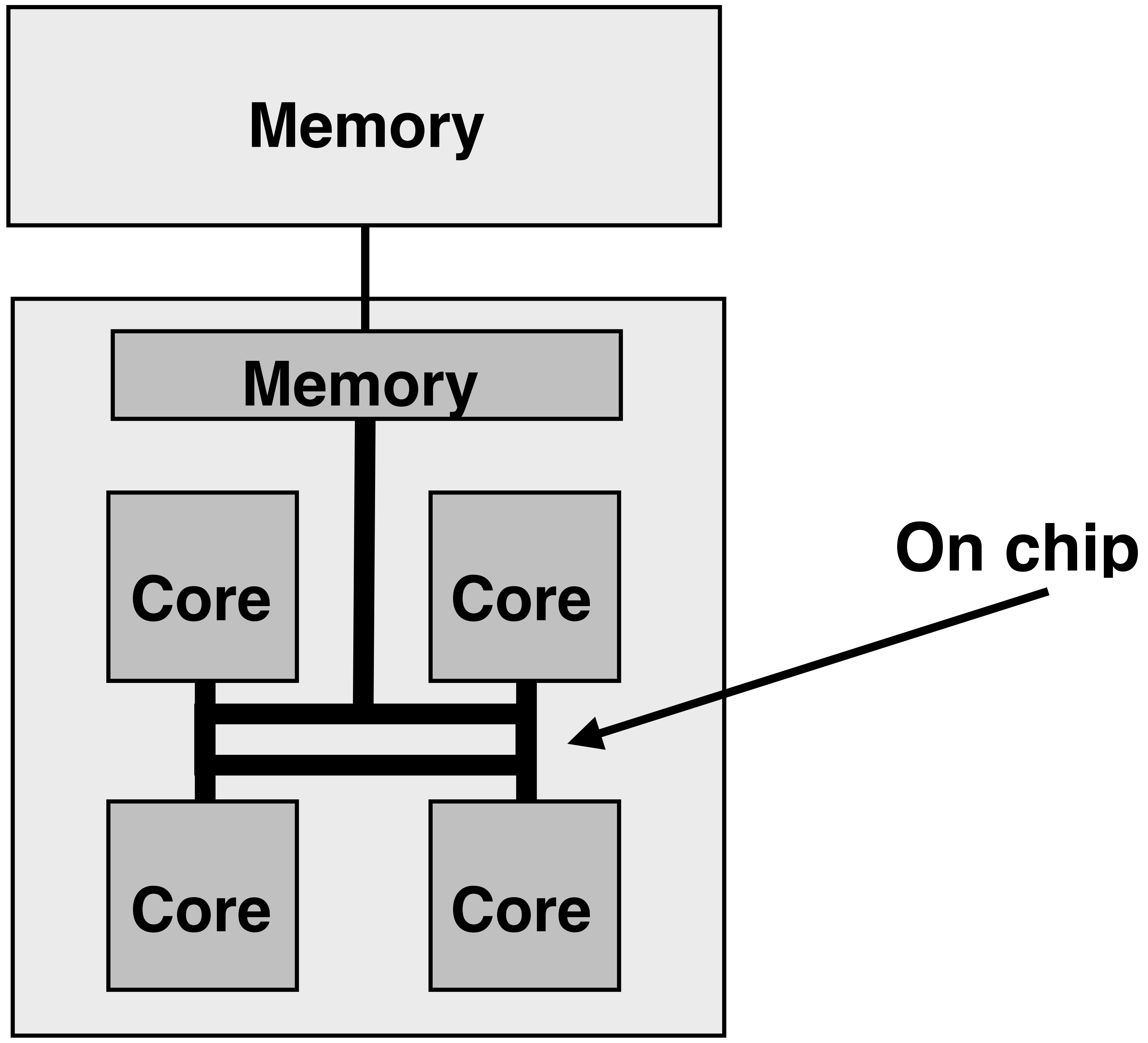
Commodity x86 examples



Intel Alder Lake-S, 16 cores (8+8)
(interconnect is a ring)



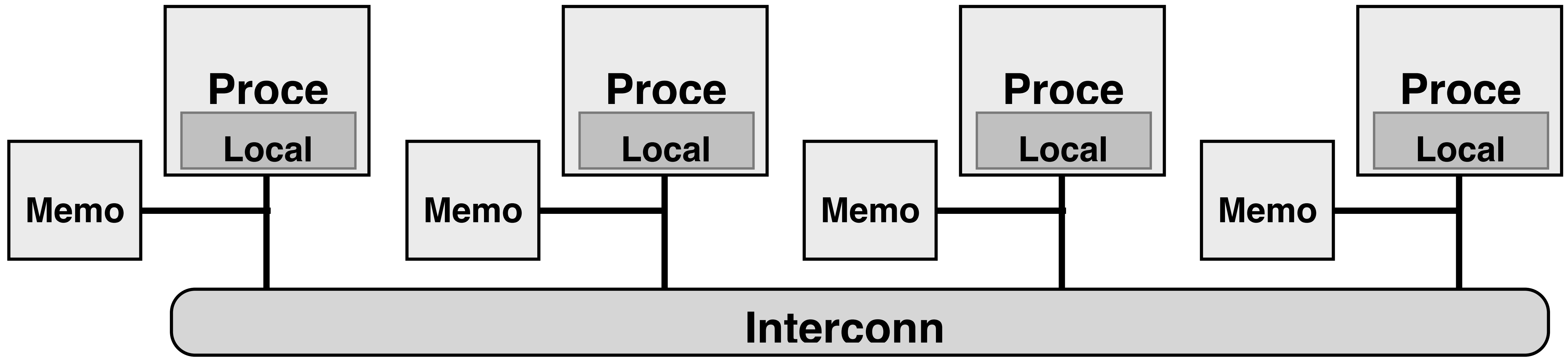
AMD Ryzen (8



AMD Ryzen Photo credit: https://arstechnica.com/information-technology/2017/02/amd-ryzen-arrives-march-2-8-cores-16-threads-from-just-329/

Non-uniform memory access

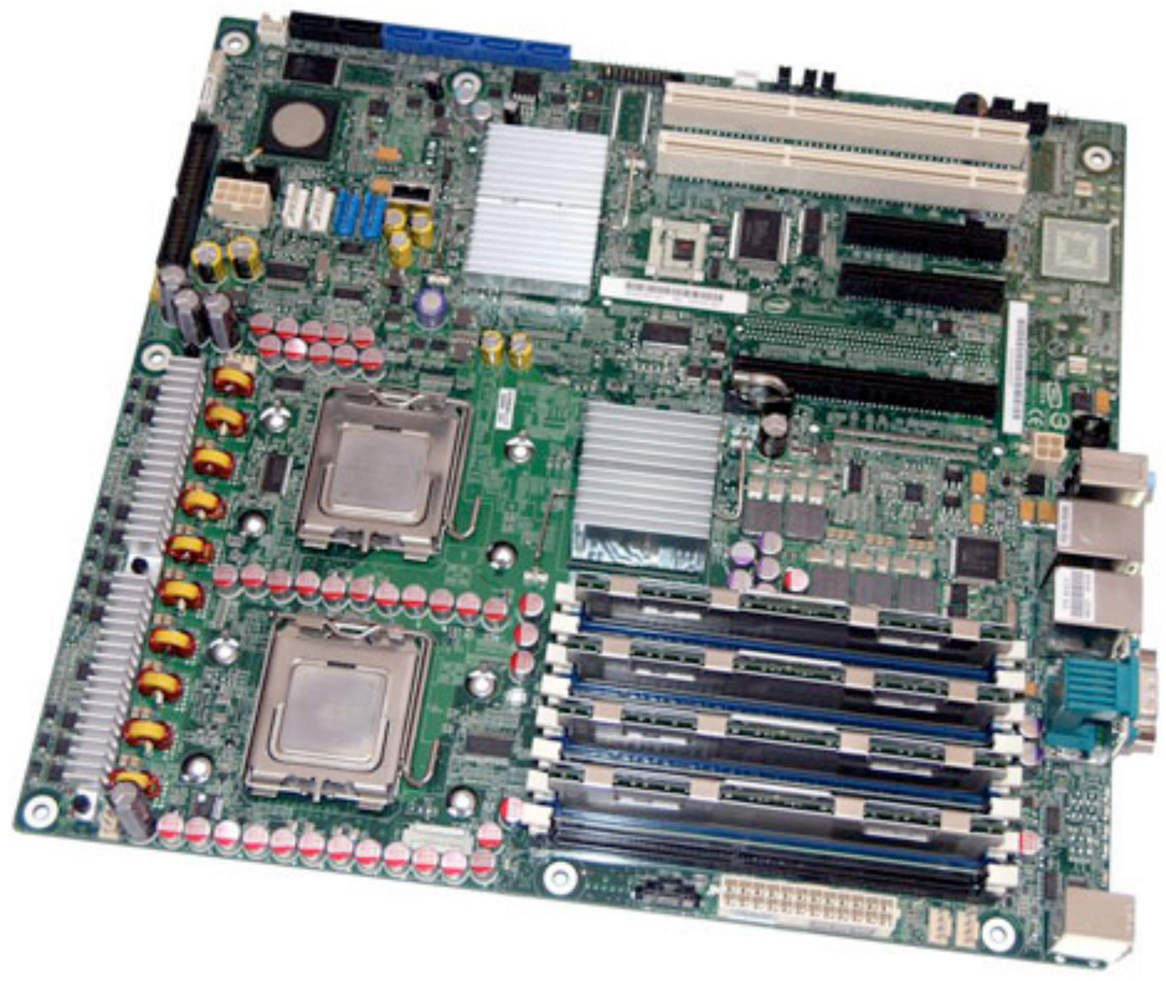
All processors can access any memory location, but... the cost of memory access (latency and/or bandwidth) is



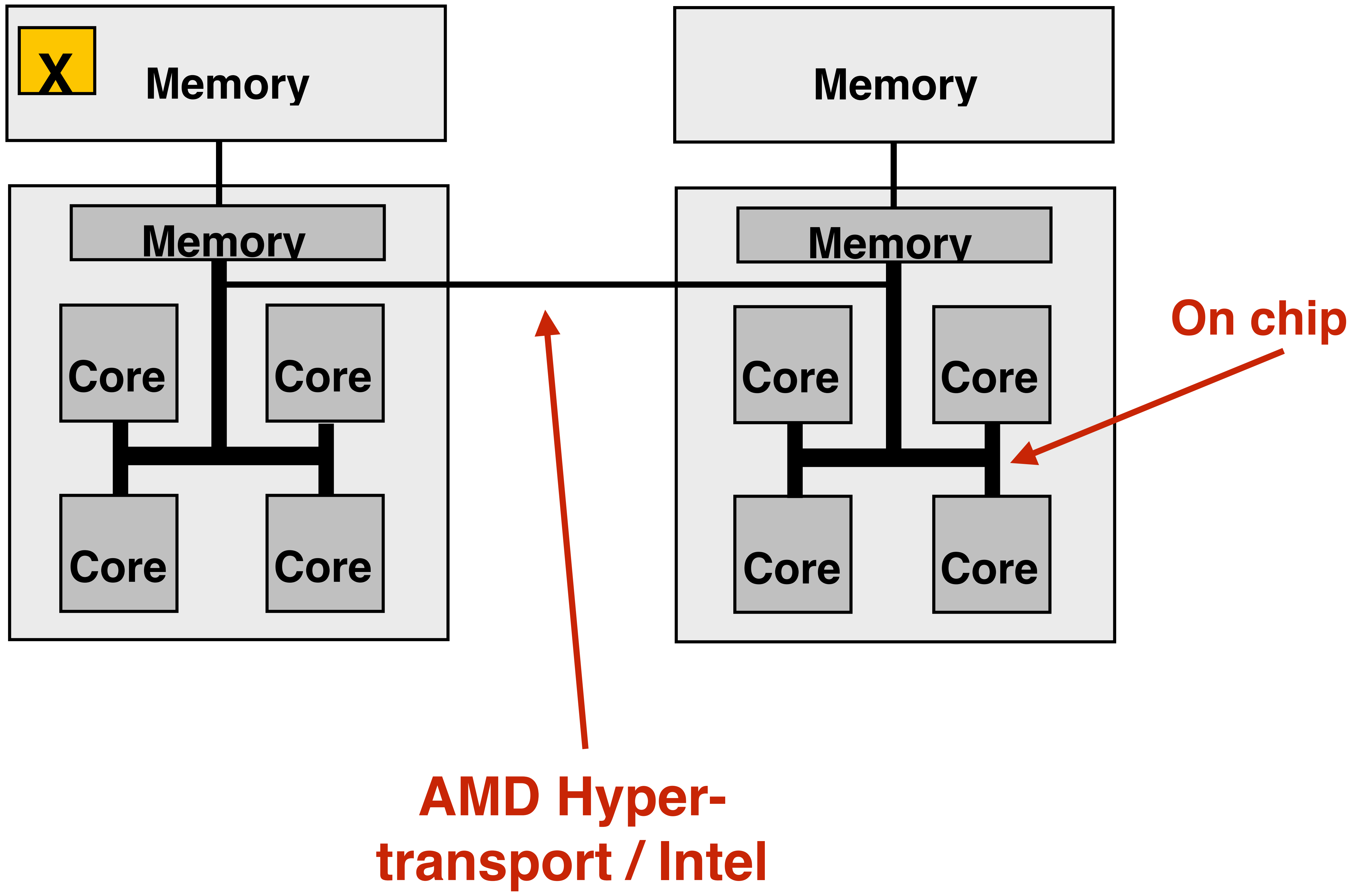
- **Problem with preserving uniform access time in a system: scalability**
  - **GOOD: costs are uniform, BAD: they are uniformly bad (memory is uniformly far away)**
- **NUMA designs are more scalable**
  - **Low latency and high bandwidth to local memory**
- **Cost is increased programmer effort for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

Non-uniform memorv access

Example: latency to access address **x** is higher from



Example: modern dual-socket



SGI Altix UV 1000

- 256 blades, 2 CPUs per blade, 8 cores per CPU = 4096 cores
- Single shared address space
- Interconnect: fat tree

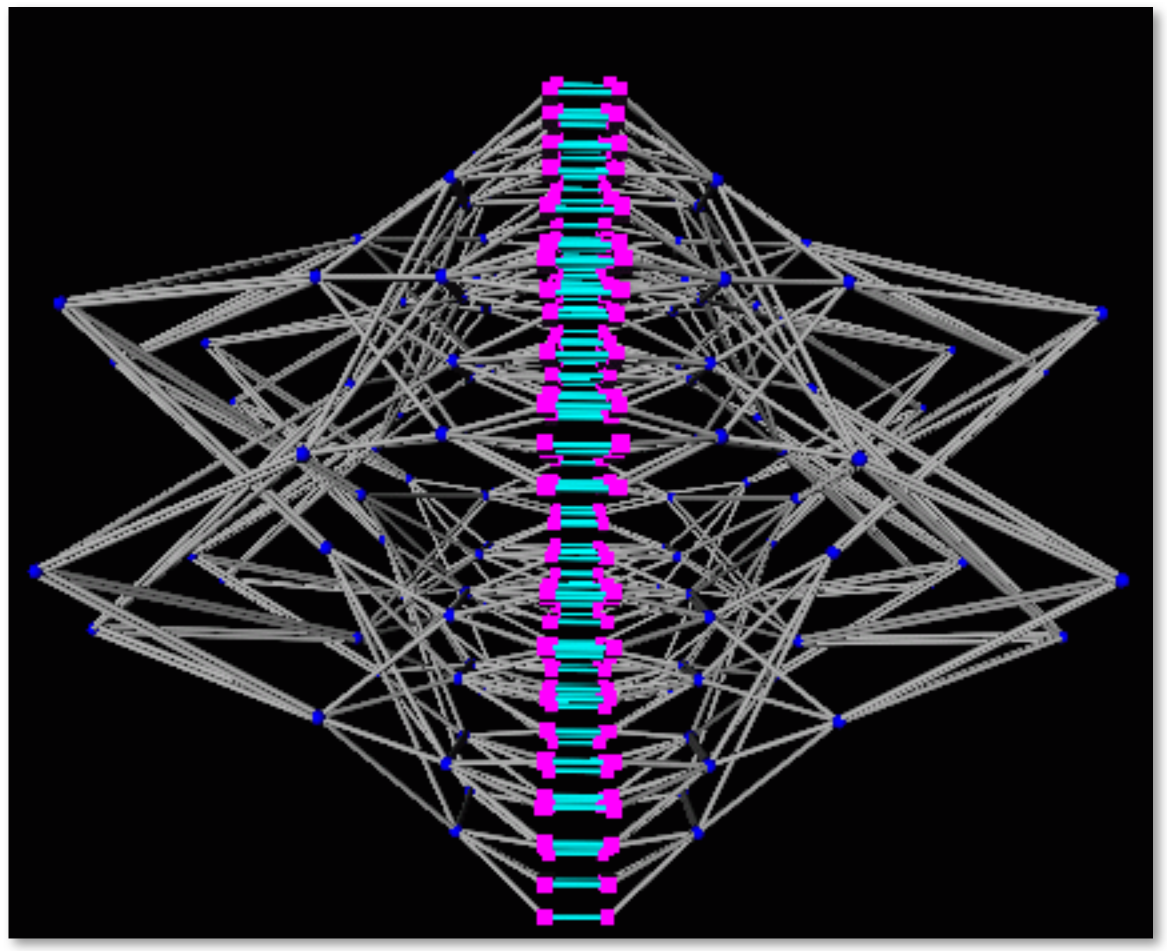

Fat tree topology

Image credit: Pittsburgh Supercomputing Center

Summary: shared address

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

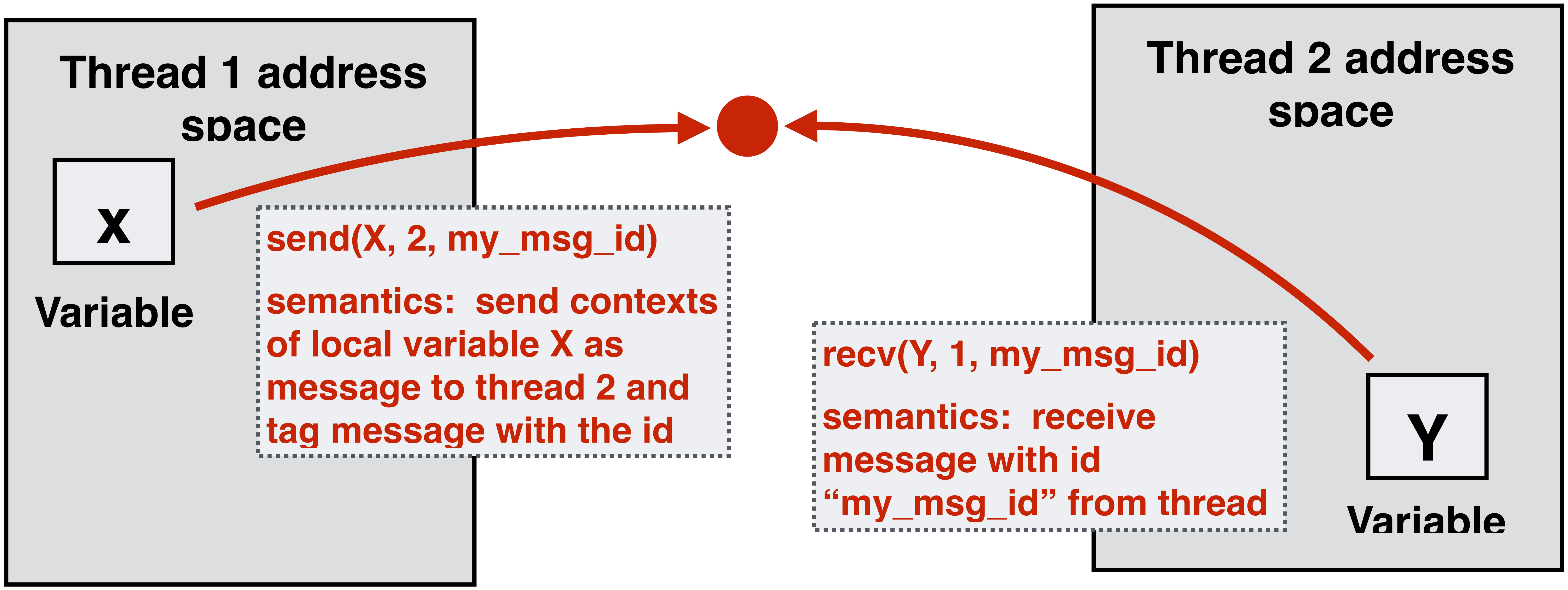
- Requires hardware support to implementation efficiently
  - Any processor can load and store from any address (its shared address space!)

* But NUMA implementation requires reasoning about locality for performance

Message passing model of

Message passing model

- 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



send(X, 2, my_msg_id)

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

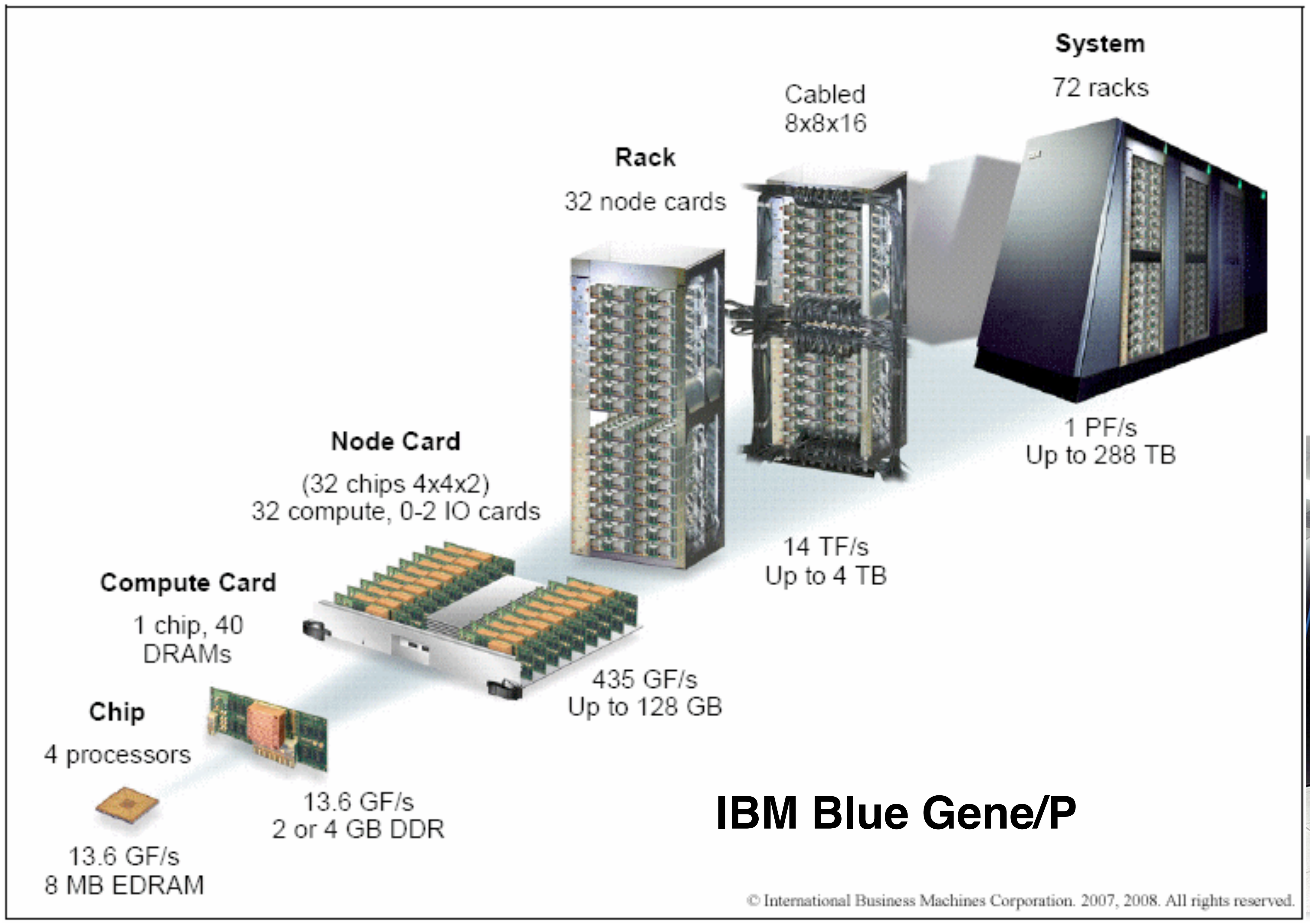
recv(Y, 1, my_msg_id)

semantics: receive message with id "my_msg_id" from thread

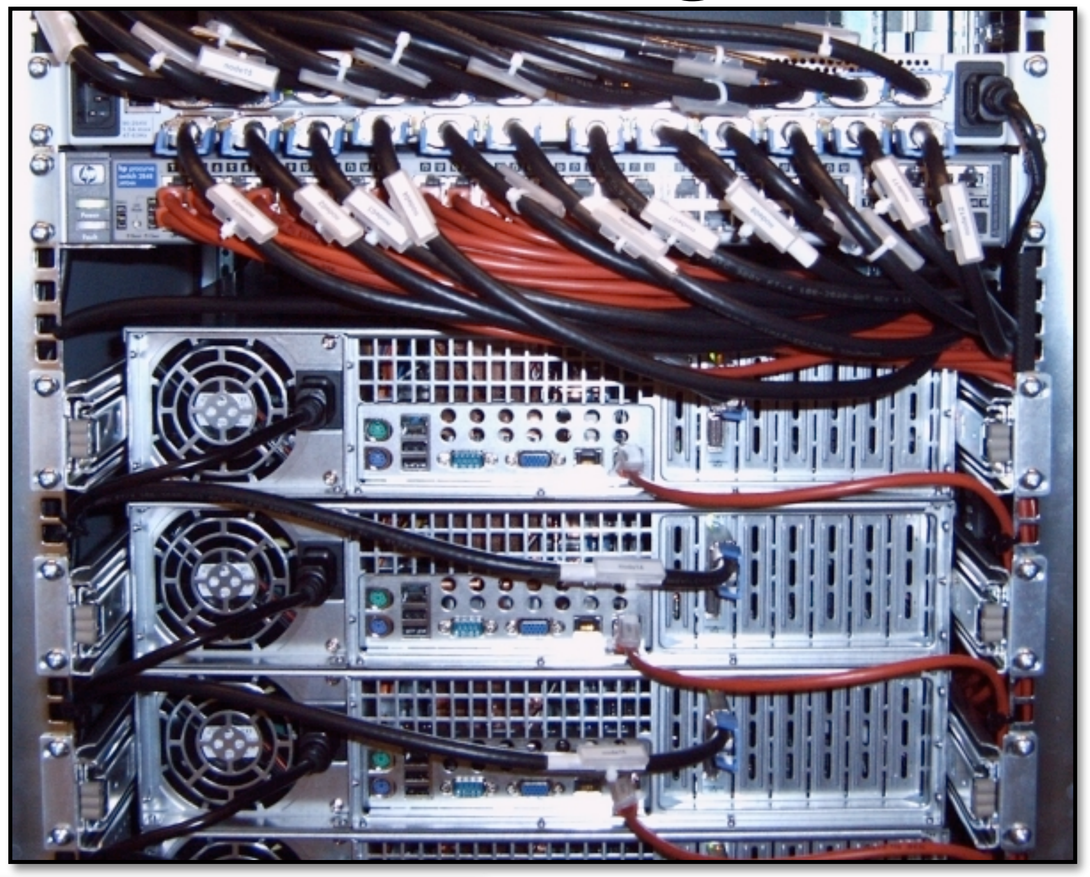
(Communication operations shown

Message passing

- Popular software library: MPI (message passing interface)
- Hardware need not implement system-wide loads and stores to execute message passing programs (need only be able to communicate messages)
  - Can connect commodity systems together to form large



IBM Blue Gene/P



Cluster of workstations

Image credit: IBM

15-418/618 “latedavs” cluster

- 15 node cluster *
  - 1 head node, 14 worker nodes accessible via a batch job queue
- Each node contains:
  - Two, six-core Xeon e5-2620 v3 processors (844 GFLOPS peak)
    - 2.4 GHz, 15MB L3 cache, hyper-threading, AVX2 (“8-wide”) instruction support
  - 16 GB RAM (60 GB/sec bandwidth)
  - NVIDIA K40 GPU (4.2 TFLOPS peak)
  - One Xeon Phi 5110p co-processor board (2 TFLOPS peak)
    - 60 “simple” 1 GHz x86 cores
    - 4-threads per core, AVX512 ("16-wide") instruction support
    - 8 GB RAM (320 GB/sec bandwidth)

* There are also three other nodes that have Titan X GPUs instead of Phis and K40s (so add

The correspondence between programming models and

- Common to implement message passing abstractions 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 solution)
  - Mark all pages with shared variables as invalid

The data-parallel model

Recall: programming models impose structure on programs

- **Shared address space: very little structure**
  - 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 not apparent in program text)
- **Message passing: highly structured communication**
  - All communication occurs in the form of messages (can read program and see where the communication is)
- **Data-parallel: very rigid computation**

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

- Matlab 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>

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);
```

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:

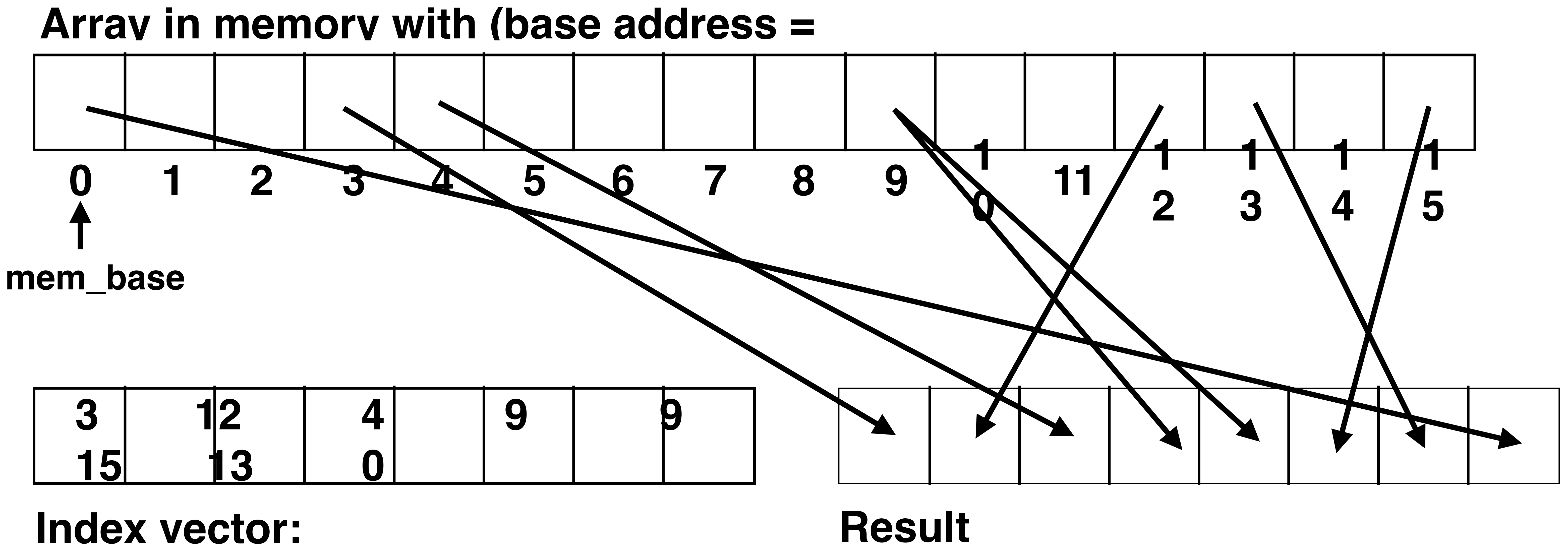
```
const int N = 1024;
stream<float> input(N);
stream<int> indices;
stream<float> tmp_output(N);
stream<float> output(N);

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)
   {
      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



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.

Summary: data-parallel model

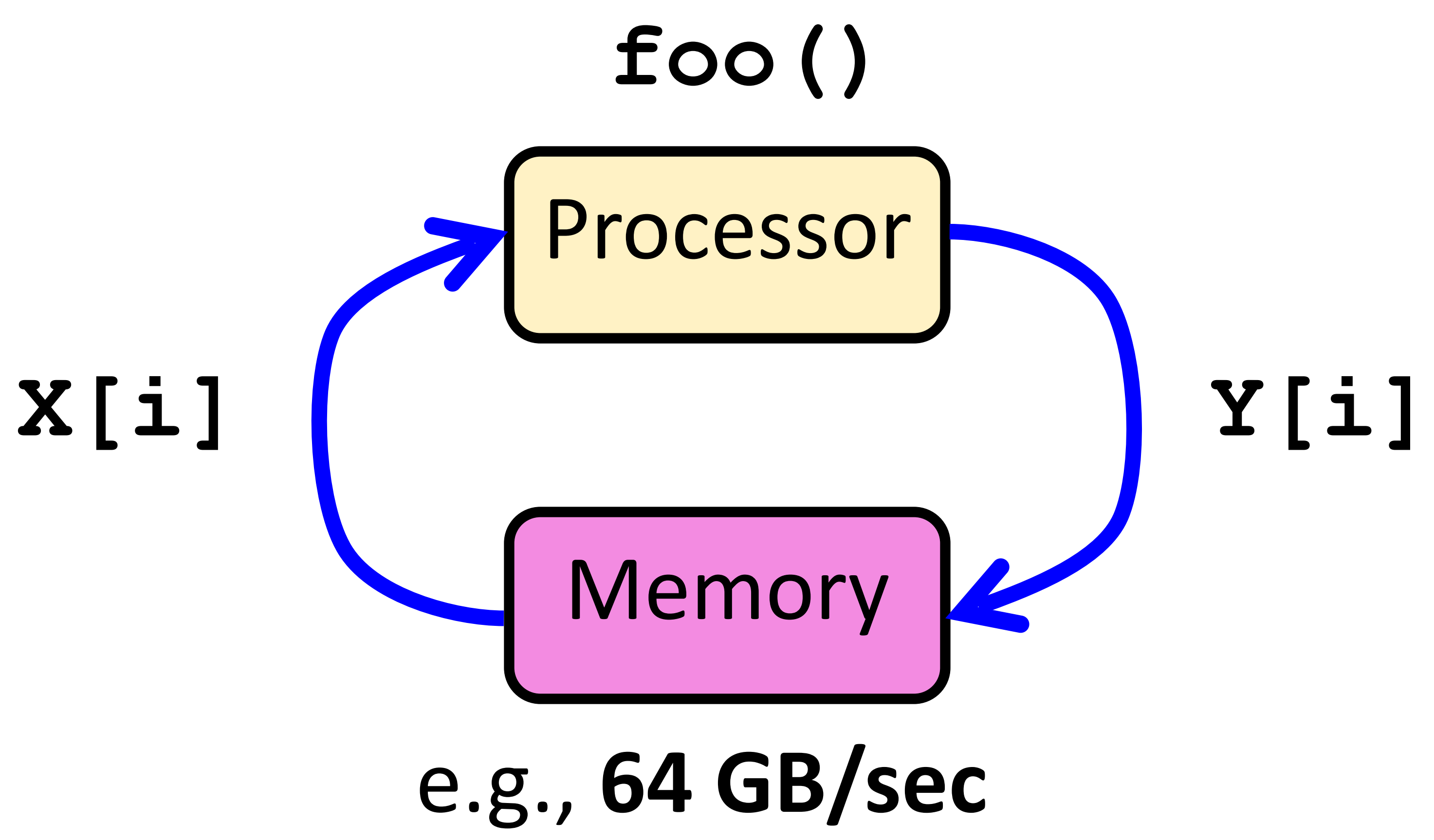
- Data-parallelism is about imposing rigid program structure to facilitate simple programming and advanced optimizations
- Basic structure: map a function onto a large collection of data
  - Functional: side-effect free execution
  - No communication among distinct function invocations (allow invocations 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 enforce this structure

The systolic arrays model

Motivation for Systolic Arrays

- <u>Target domain</u>:
  - data-intensive applications where memory bandwidth is the bottleneck
- Simple bandwidth limited example:

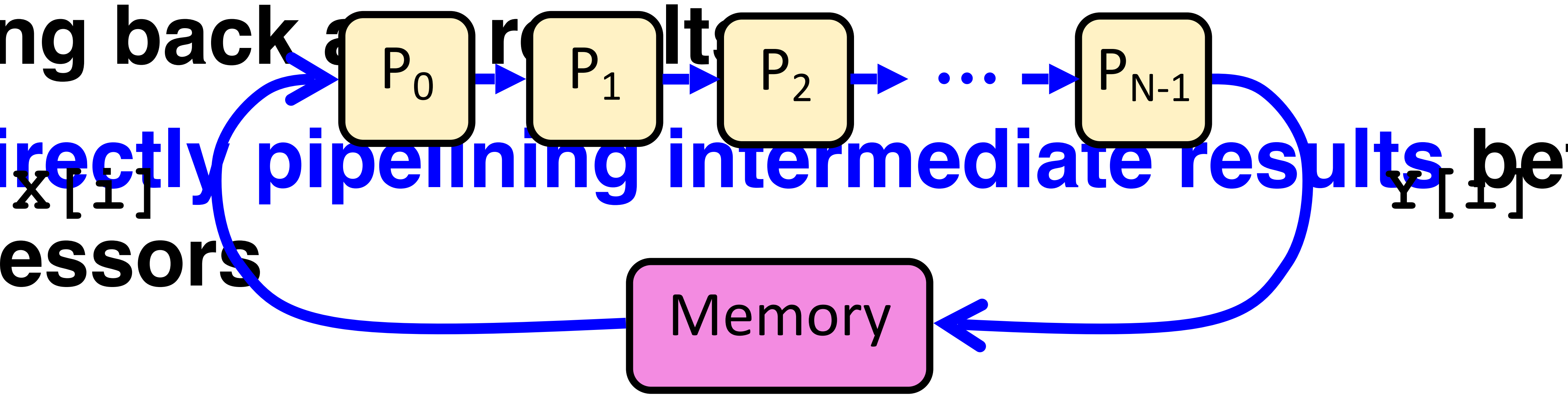
```
for i = 0 to N-1
   Y[i] = foo(X[i]);
```



- **Due to lack of temporal locality, caches do not solve t problem**

Systolic Arrays: Key Insights

- Because memory bandwidth is the critical resource:
  - we really want to **avoid accessing memory unnecessarily**
- Hence whenever we read data from memory:
  - **perform all necessary computation** on it before writing back all results
  - by **directly pipelining intermediate results** between processors

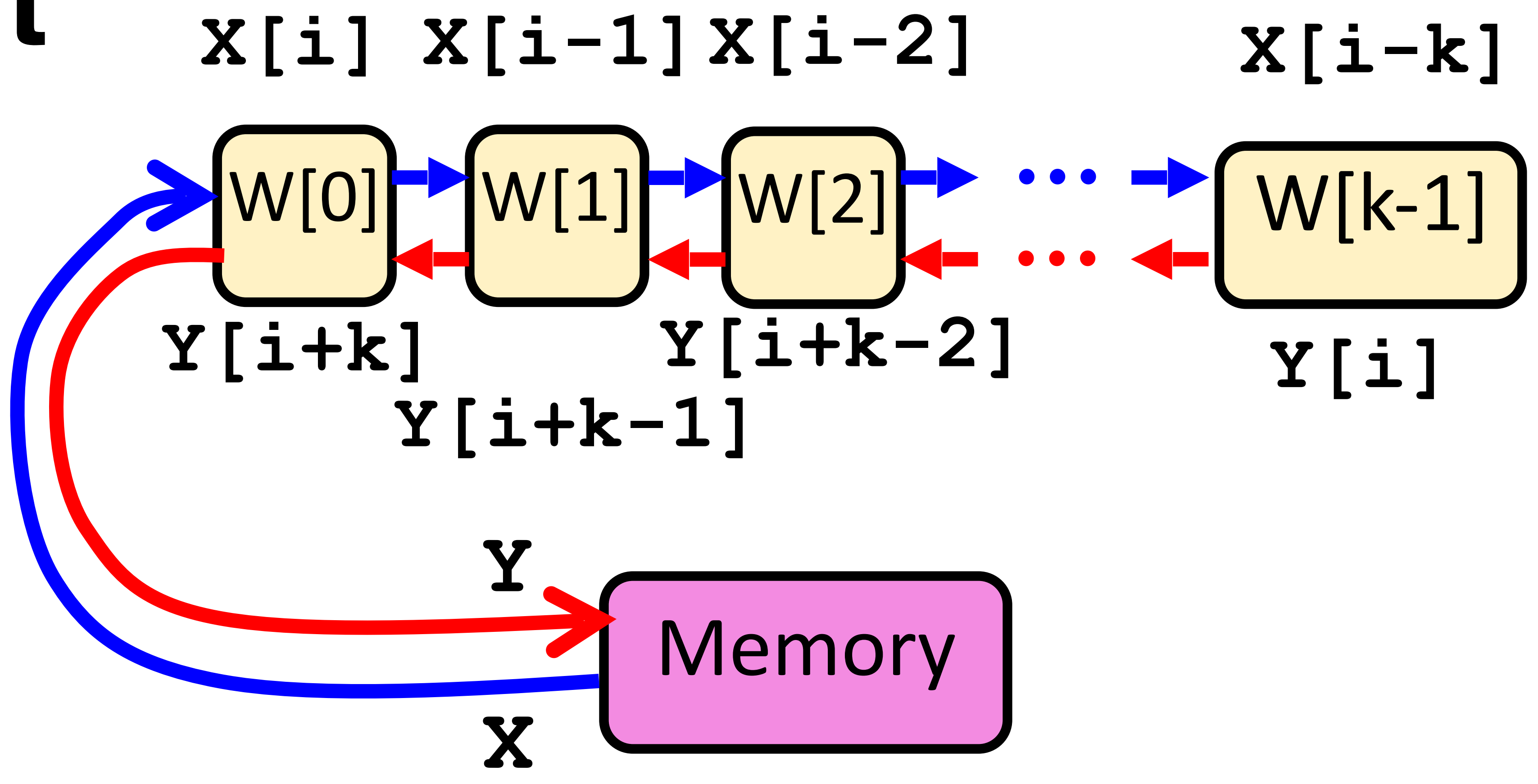


e.g., **64 GB/sec**

Systolic Arrays Example: 1D

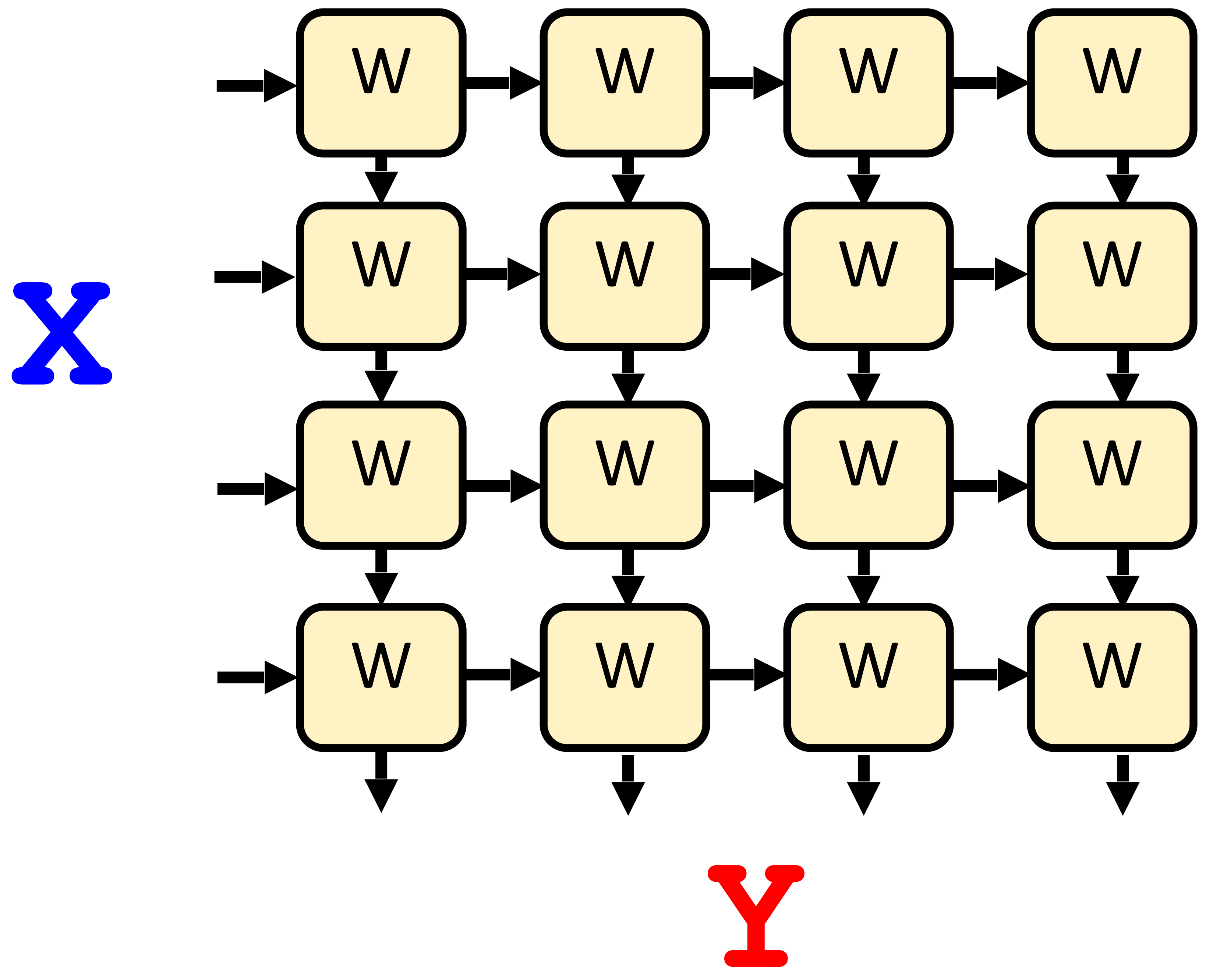
$$Y[i] = \sum_{k=0}^{K-1} W[k] \cdot X[i-k]$$

- **pre-load the weights (W[i]) onto each processor**
- **input data (X[i]) flows from left to right**
- **partial results (eventually forming Y[i]) flow from right to left**



Systolic Arrays Today: ML

$$Y_{i,j} = \sum_{k=1}^{n} X_{i,k} W_{k,j}$$



- Basic idea behind AWS Trainium, Google

Four parallel programming

- **Shared address space**
  - Communication is unstructured, implicit in loads and stores
  - Natural way of programming, but can shoot yourself in the foot easily
    - Program might be correct, but not perform well
- **Message passing**
  - Structure all communication as messages
  - Often harder 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

Modern practice: mixed

- **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-**

Summary

- Programming models provide a way to think about the organization of parallel programs. They provide abstractions that admit many possible implementations.

- Restrictions imposed by these abstractions are designed to reflect realities of parallelization and communication costs
  - Shared address space machines: hardware supports any processor accessing any address
  - Messaging passing machines: may have hardware to accelerate message send/receive/buffering
  - It is desirable to keep "abstraction distance" low so programs have predictable performance, but want it high enough for code flexibility/portability