Lecture 3:

Parallel Programming Models and their corresponding HW/SW implementations

---

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2014

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: $\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \ldots$ 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);
```

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;
    }
}
```

SPMD programming abstraction:

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

All instances run ISPC code in parallel

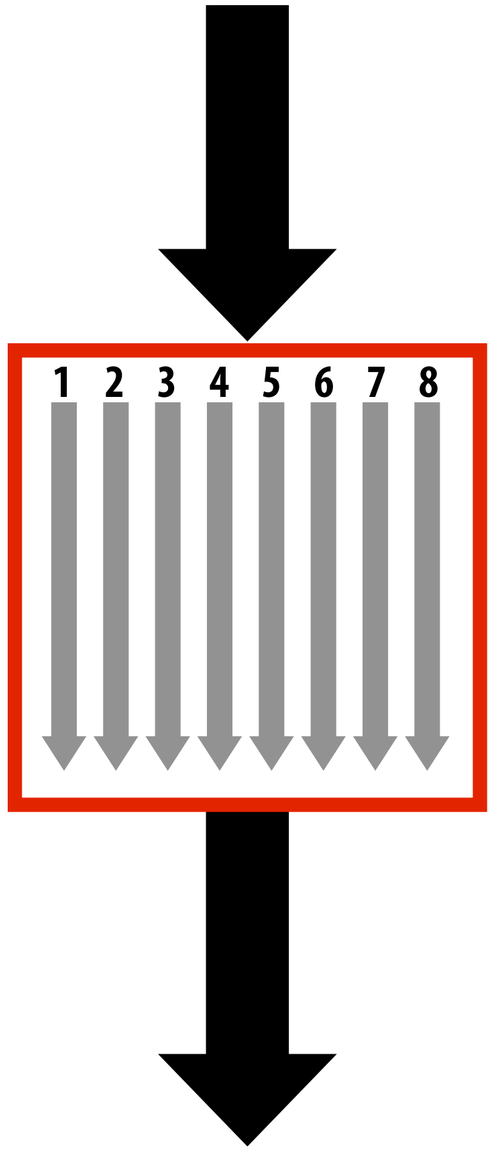
Upon return, all instances have completed

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);
```



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 in parallel

Upon return, all instances have completed

sin(x) in ISPC

Interleaved 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)
{
    // 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;
    }
}
```

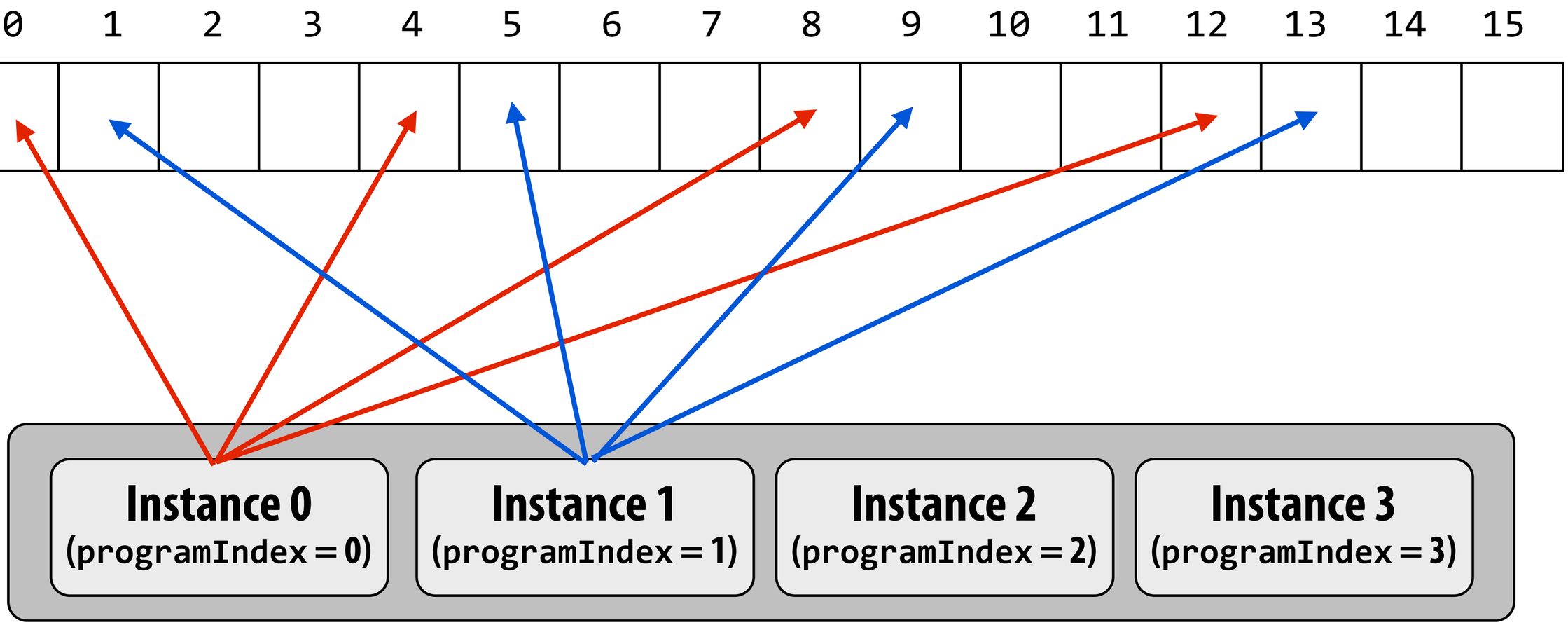
ISPC Keywords:

`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 instances to loop iterations



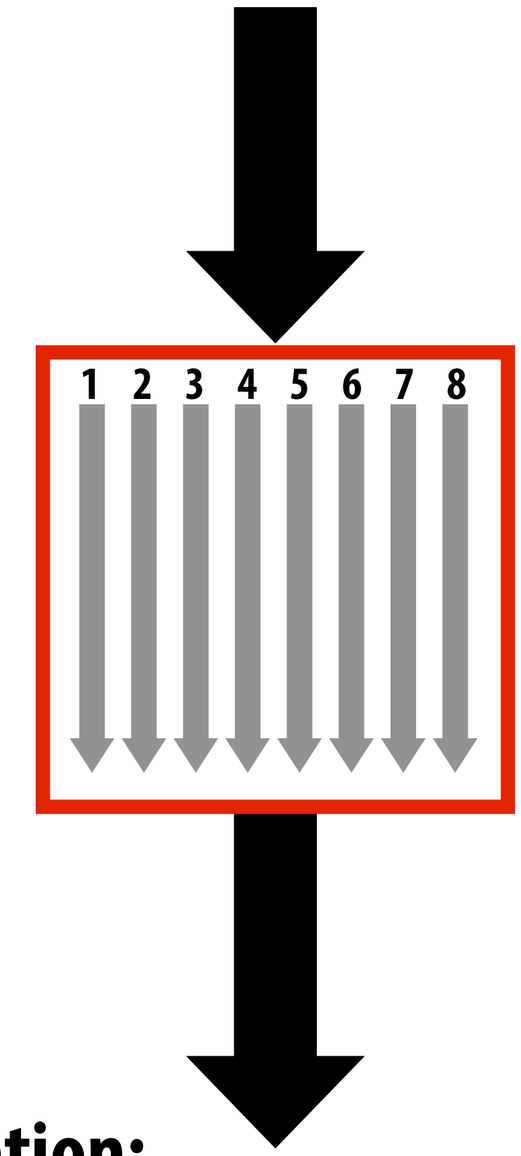
"Gang" of ISPC program instances

Gang contains four instances: `programCount` = 4

ISPC implements it's 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 in parallel
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

Blocked assignment of elements to instances

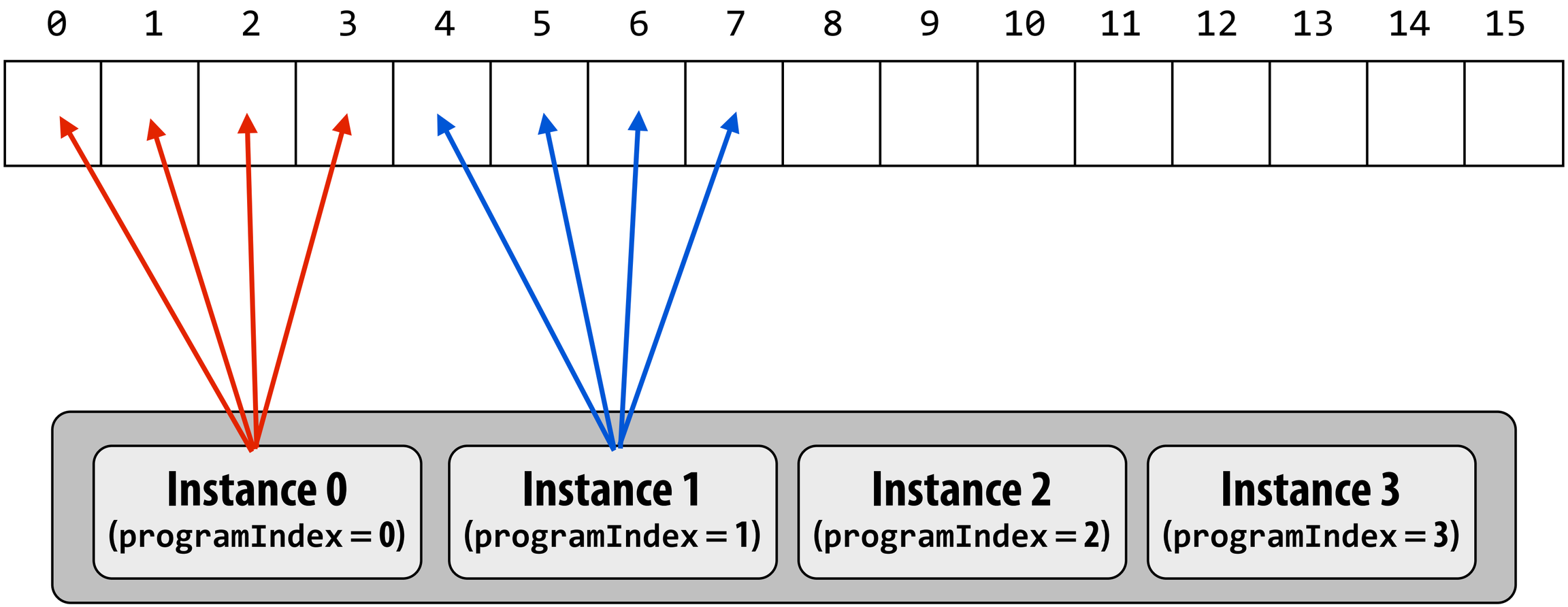
C++ code: `main.cpp`

```
#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 instances to loop iterations



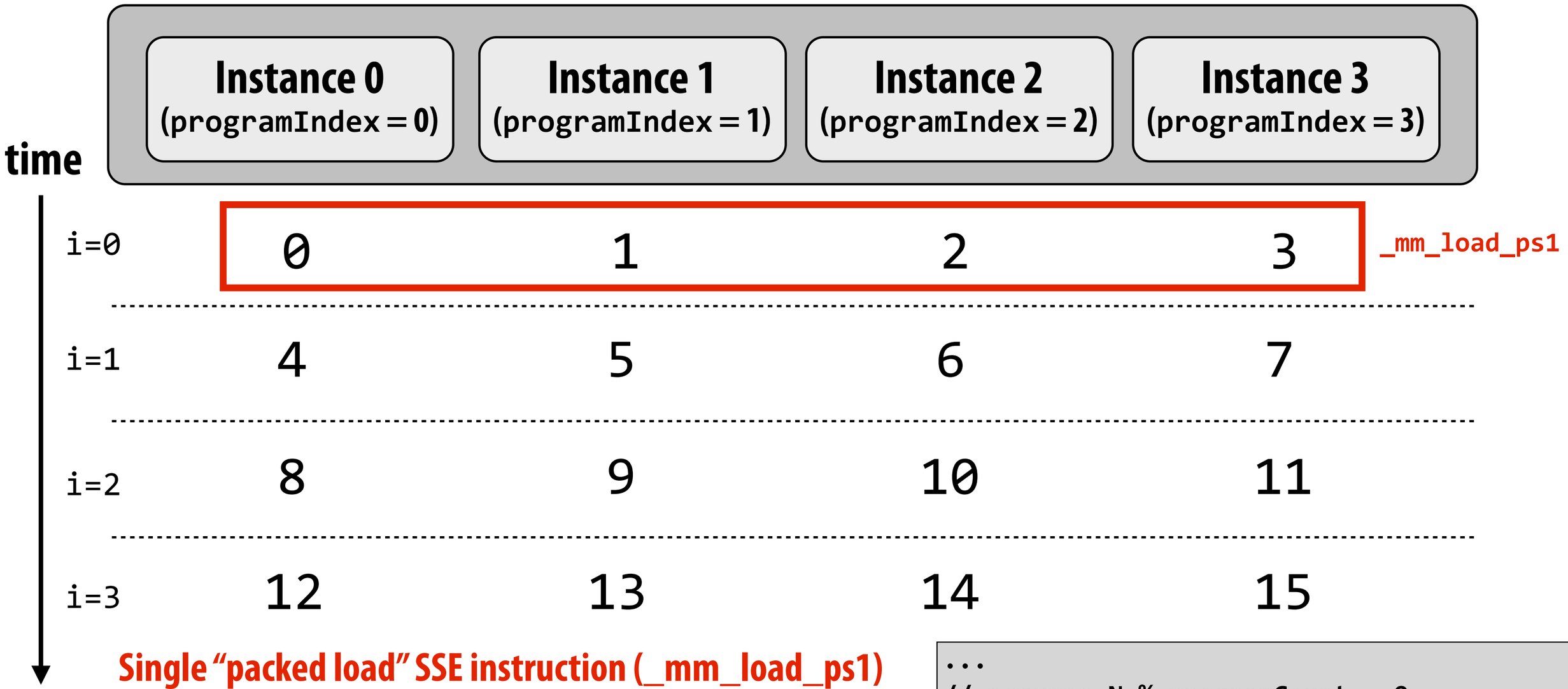
"Gang" of ISPC program instances

Gang contains four instances: `programCount = 4`

Schedule: interleaved assignment

"Gang" of ISPC program instances

Gang contains four instances: `programCount = 4`



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

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

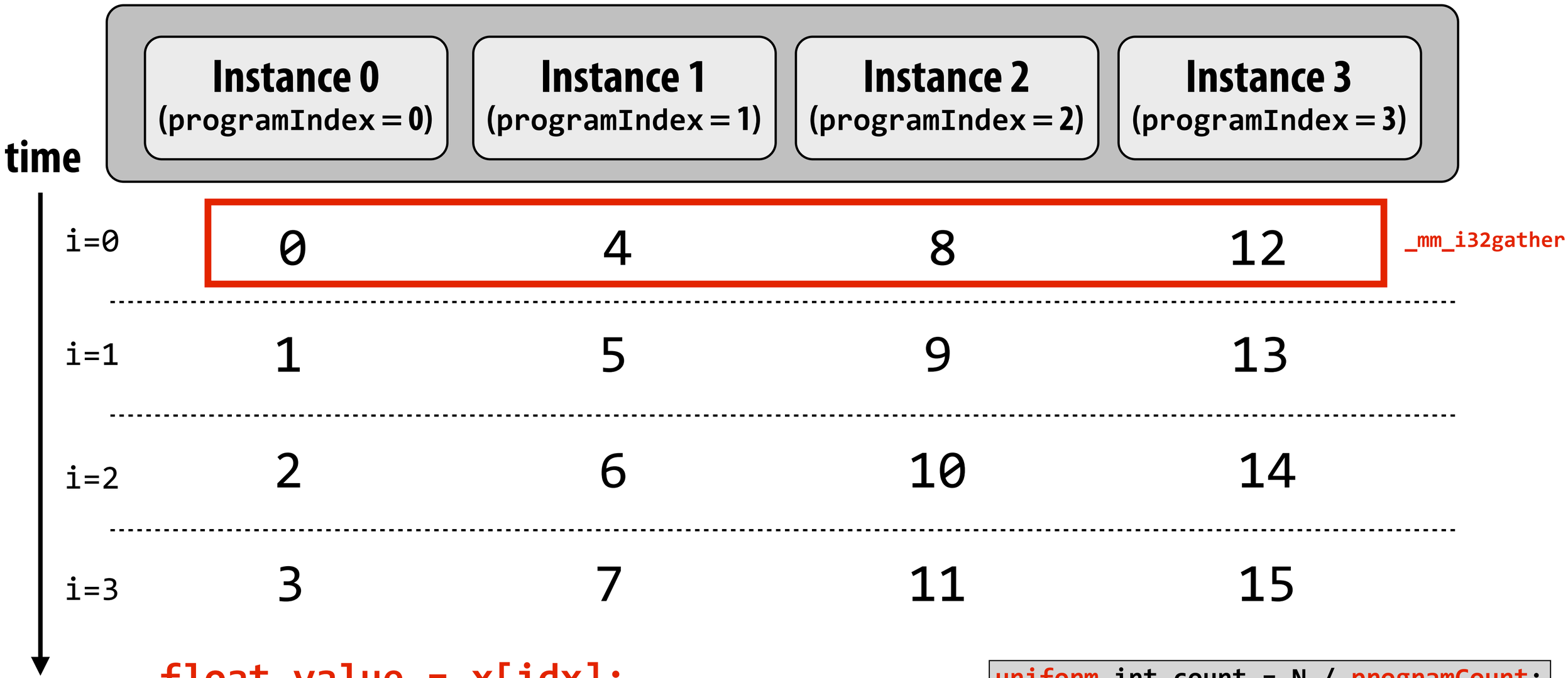
for all program instances, since the four values are contiguous in memory

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

Schedule: interleaved assignment

"Gang" of ISPC program instances

Gang contains four instances: `programCount = 4`




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

now touches four non-contiguous values in memory.

Need "gather" instruction to implement (gather is a far more complex SIMD instruction: available in 2013 on CPUs 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[idx] = value;
    }
}
```

`foreach`: key ISPC language construct

- Used to declare parallel loop iterations
  - Programmer says: these are the iterations the instances in a gang 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**
  - 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)
  - Handles mapping of conditional control flow to vector instructions

- **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 = reduceAdd(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

Each instance accumulates a private partial sum (no communication)

Partial sums are added together using the reduceAdd() cross-instance communication primitive. The result is the same for all instances (uniform)

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

```
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 = reduceAdd(partial);

   return sum;
}
```

```
const int N = 1024;
float* x = new float[N];
__mm256 partial = _mm256_broadcast_ss(0.0f);

// populate x

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

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

*** If you understand why this implementation complies with the semantics of the ISPC gang abstraction, then you've got good command of ISPC.**

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 5205 machines.

- ISPC contains another abstraction: a "task" that is used to achieve multi-core execution. I'll let you read up about that.

Today

- **Three parallel programming models**
  - Abstractions presented to the programmer
  - Influence how programmers think when writing programs

- **Three machine architectures**
  - Abstraction presented by the hardware to low-level software
  - Typically reflect implementation

- **Focus on differences in communication and cooperation**

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

Parallel Applications

Abstractions for describing concurrent, parallel, or independent computation

Abstractions for describing communication

"Programming model" (way of thinking about things)

Language or library primitives/mechanisms

Compiler and/or parallel runtime

OS system call API

Operating system

Hardware Architecture (HW/SW boundary)

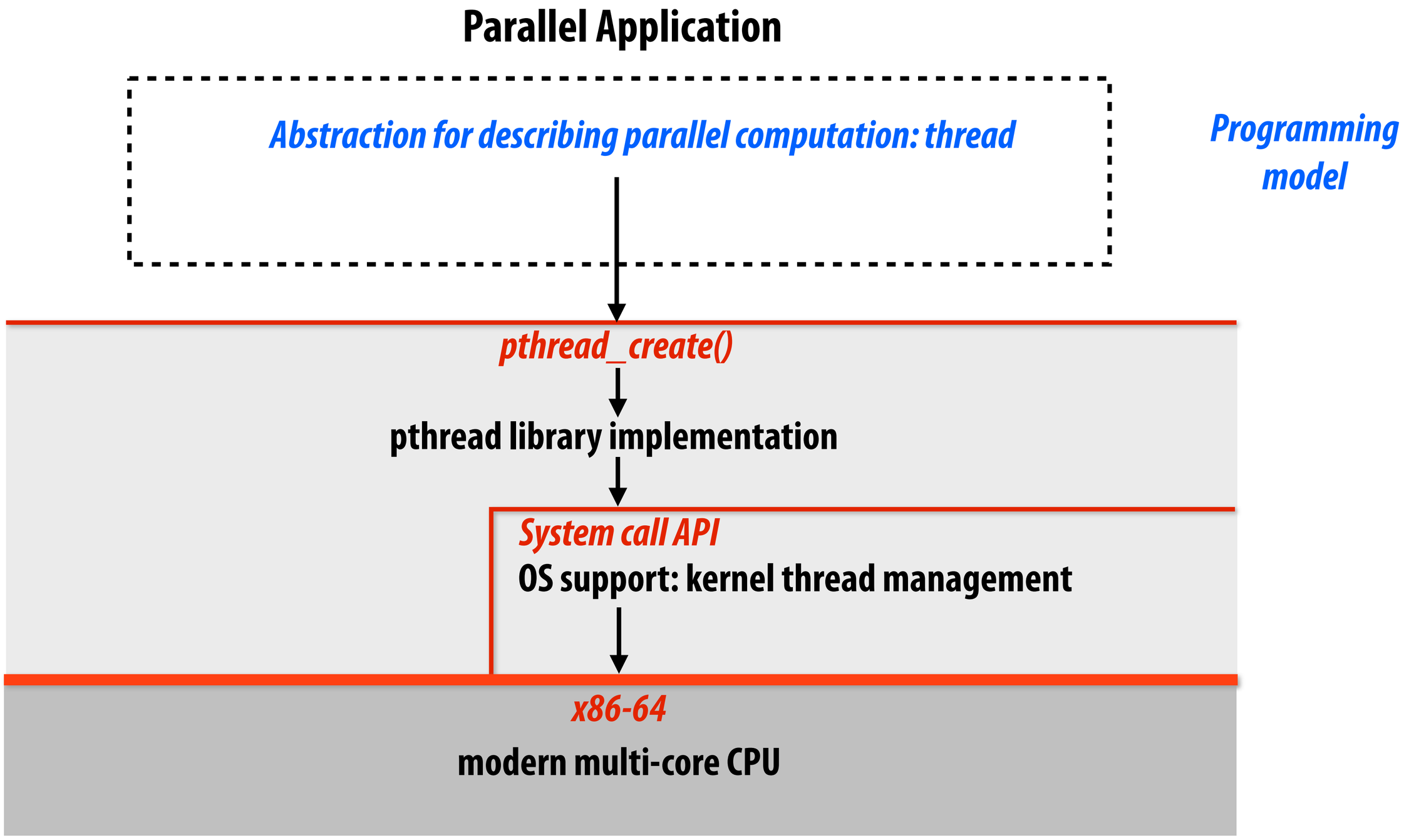
Micro-architecture (hardware implementation)

Blue italic text: abstraction/concept

Red italic text: system interface

Black text: system implementation

Example: expressing parallelism with pthreads

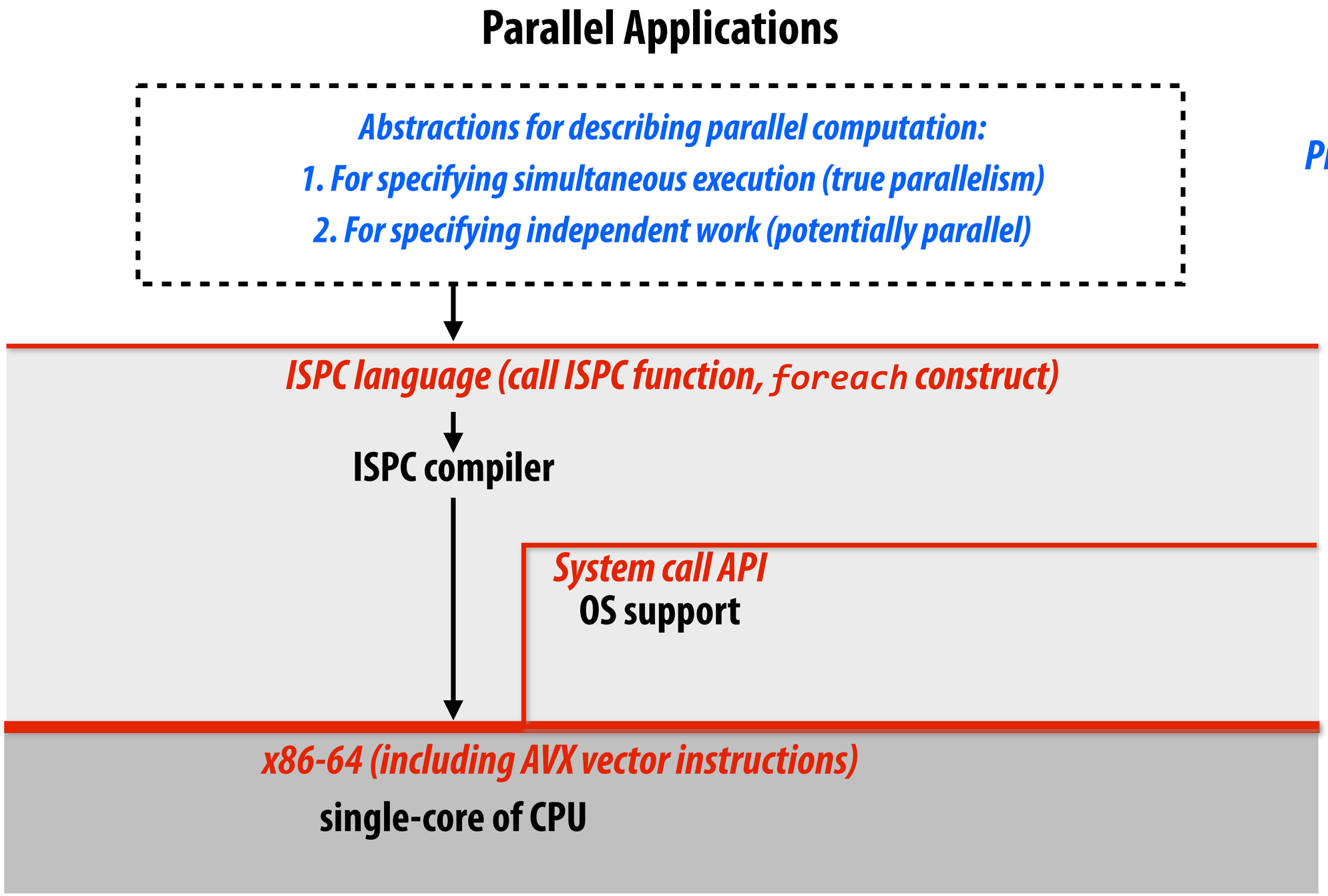


Blue italic text: abstraction/concept

Red italic text: system interface

Black text: system implementation

Example: expressing parallelism (ISPC)



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)

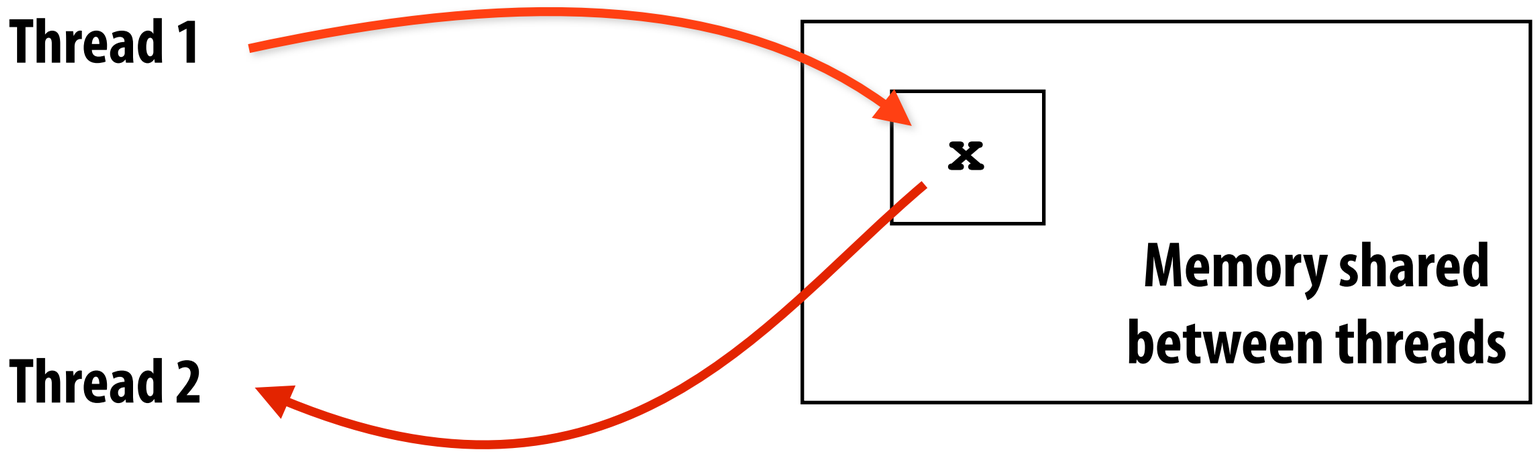
1. Shared address space

2. Message passing

3. Data parallel

Shared address space model (abstraction)

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



Thread 1:

```
int x = 0;
x = 1;
```

Thread 2:

```
int x;
while (x == 0) {}

print x;
```

Shared address space model (abstraction)

- Threads communicate by:
  - Reading/writing to shared variables
    - Interprocessor communication is implicit in memory operations
    - Thread 1 stores to X.
    - Later, thread 2 reads X (observes update)
  - Manipulating synchronization primitives
    - e.g., mutual exclusion using locks

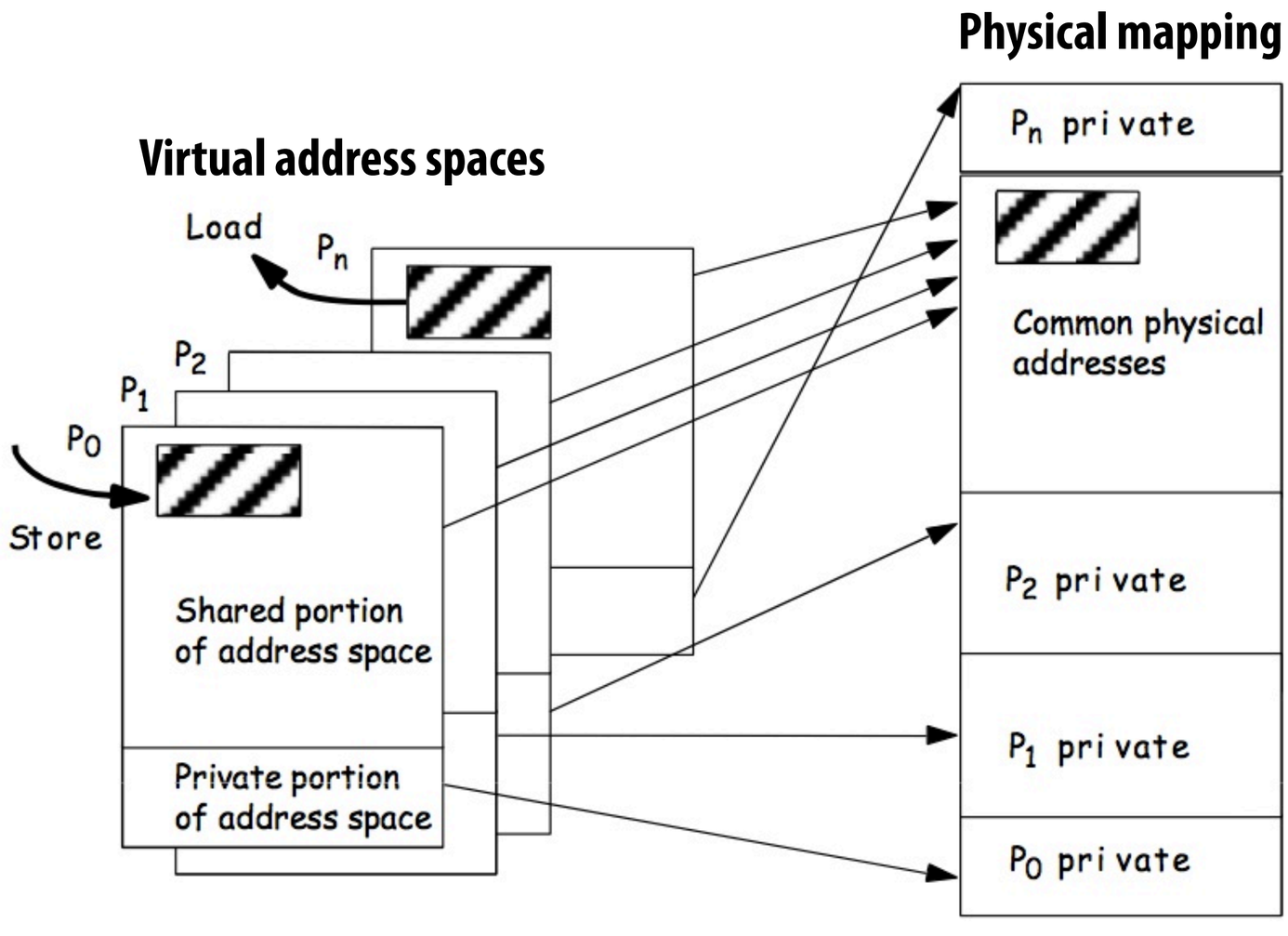
- Natural extension of sequential programming model
  - In fact, all our discussions have assumed a shared address space so far

- Think: shared variables are like a big bulletin board
  - Any thread can read or write

Shared address space (implementation)

- **Implementation option 1: threads share an address space (all data is sharable)**

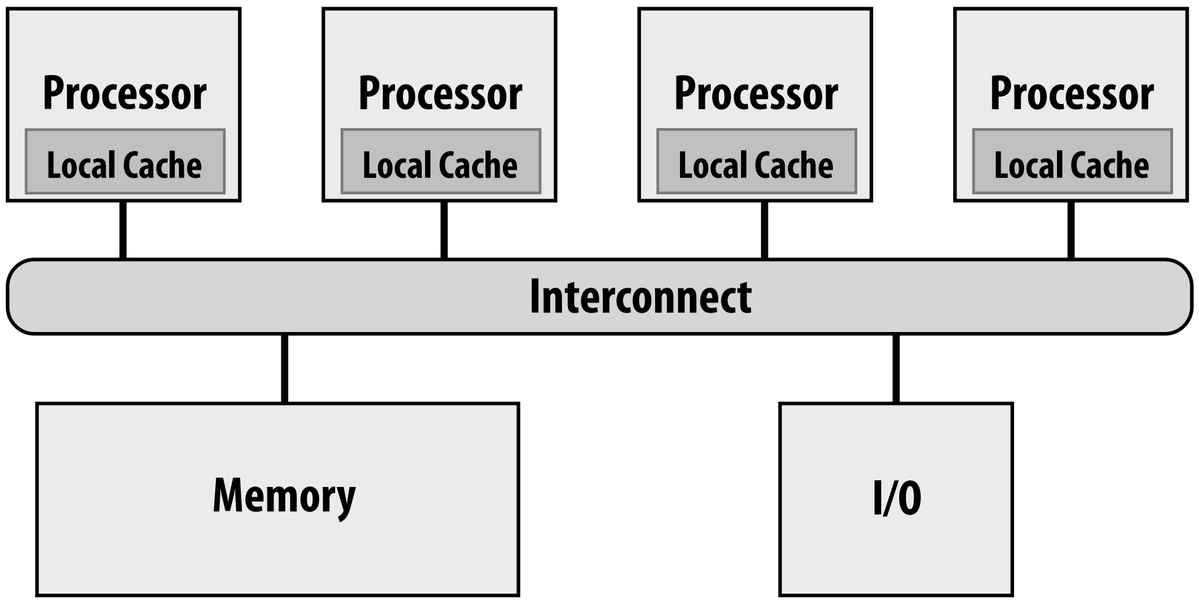
- **Implementation option 2: each thread has its own virtual address space, shared portion of address spaces maps to same physical location**



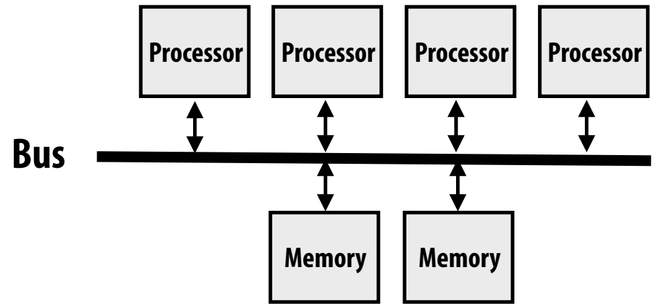
Shared address space HW implementation

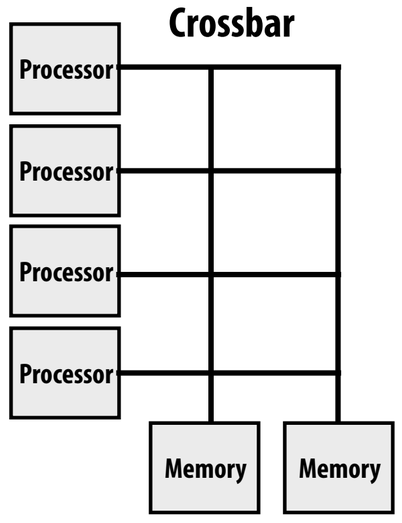
Any processor can directly reference any memory location

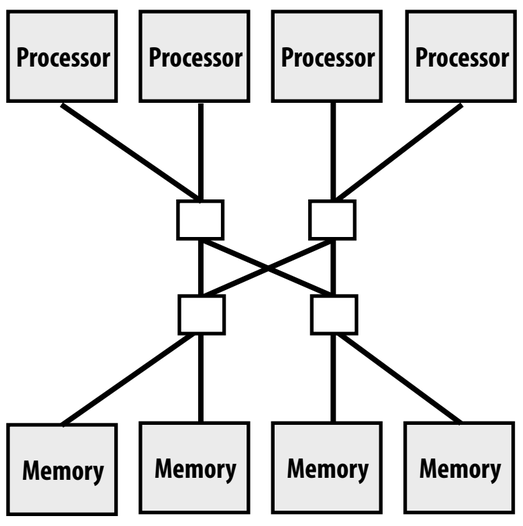
"Dance-hall" organization



Interconnect examples







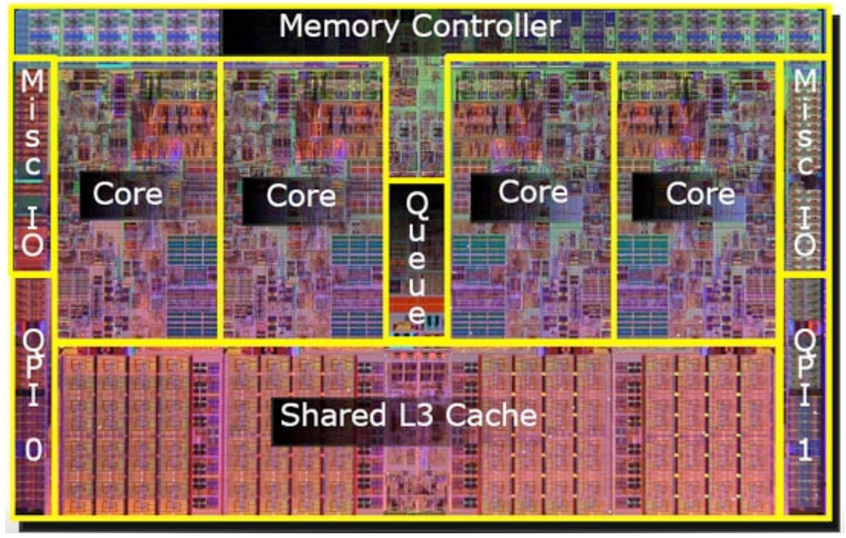
Multi-stage network

- **Symmetric (shared-memory) multi-processor (SMP):**
  - **Uniform memory access time: cost of accessing an uncached* memory address is the same for all processors**

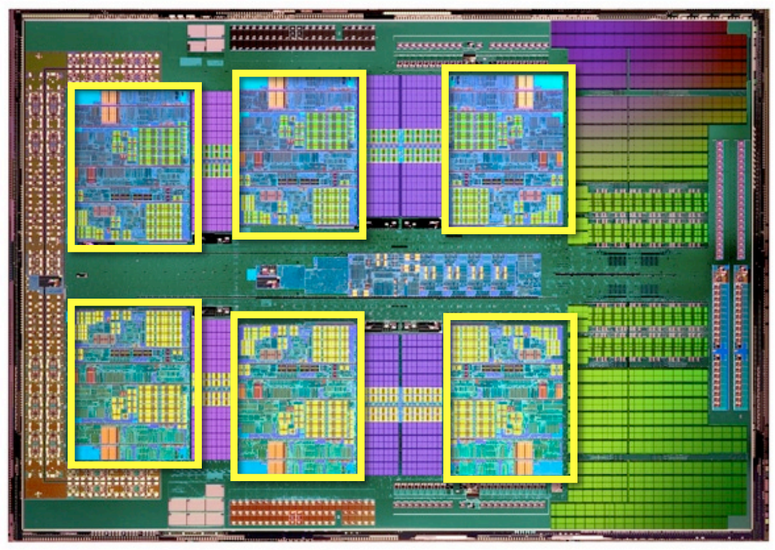
    (* caching introduces non-uniform access times, but we'll talk about that later)

Shared address space architectures

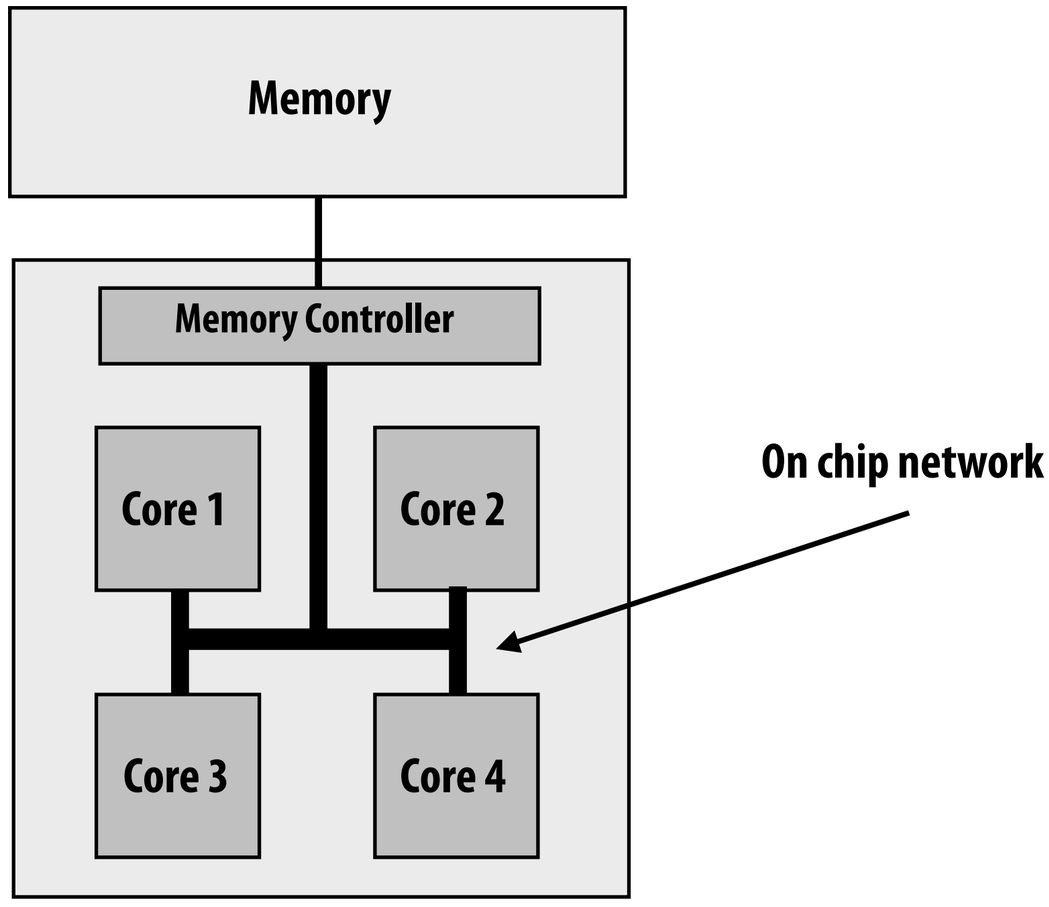
Commodity x86 examples



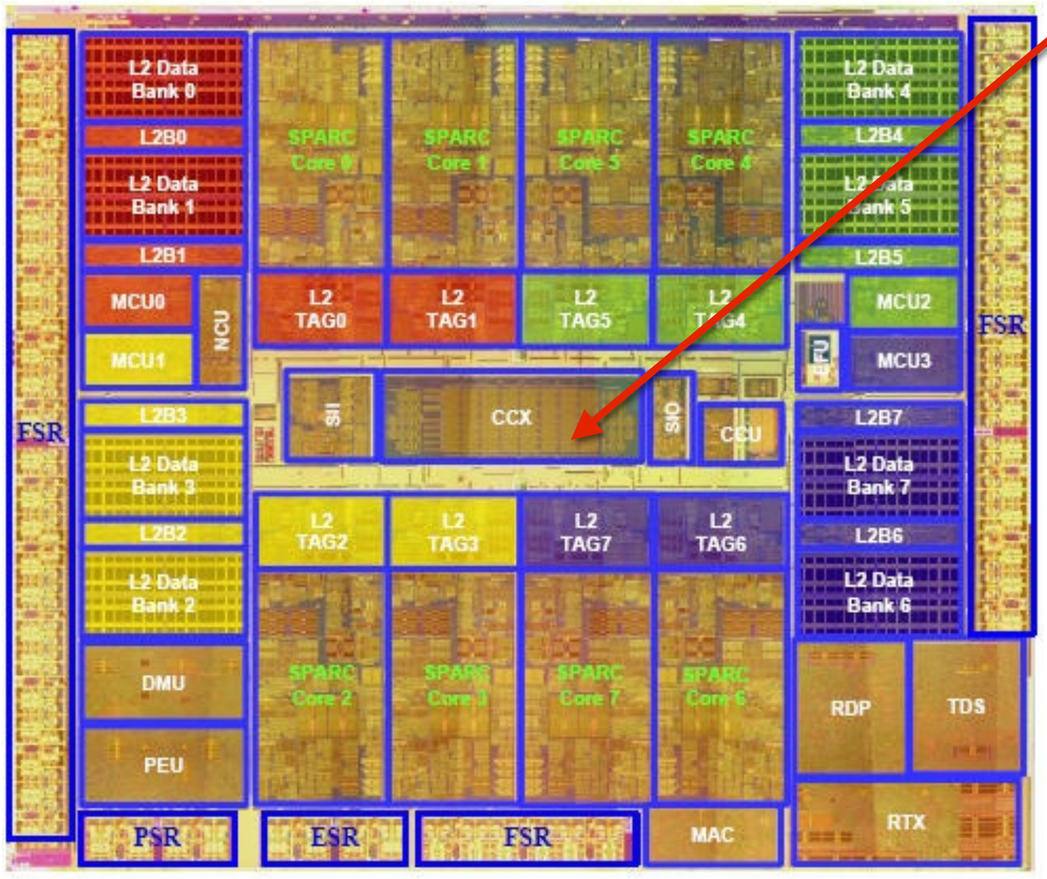
Intel Core i7 (quad core)
(network is a ring)



AMD Phenom II (six core)

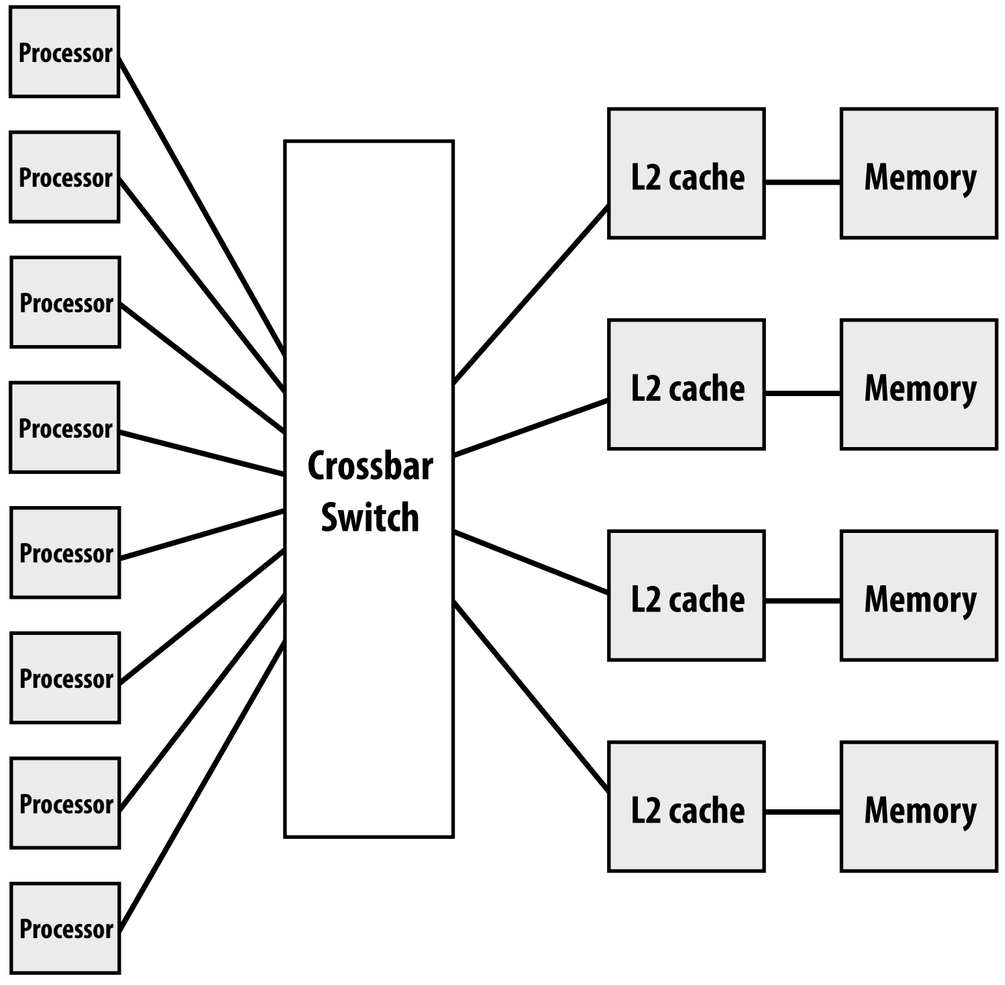


SUN Niagara 2



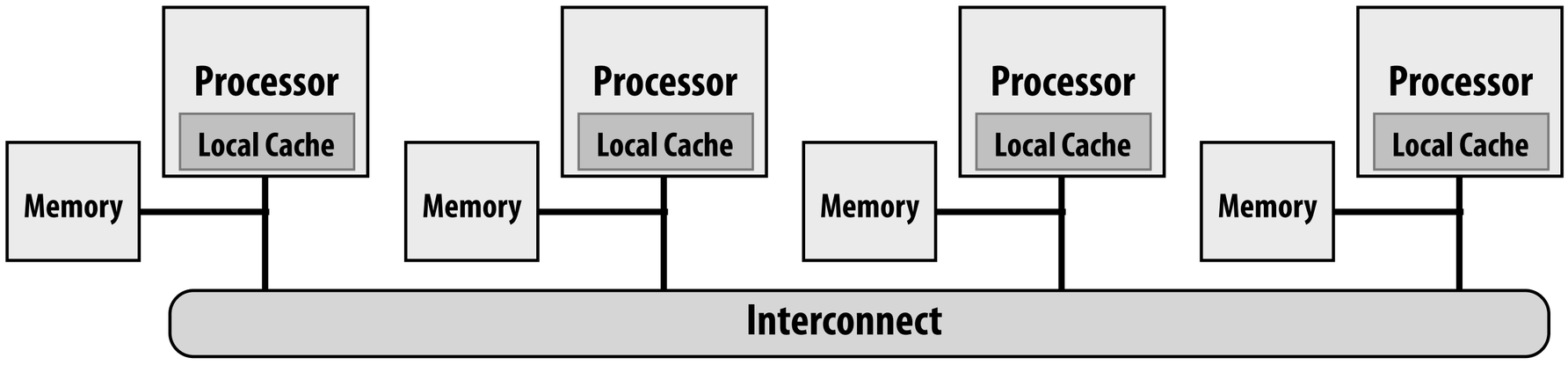
Eight cores

Note size of crossbar: about die area of one core



Non-uniform memory access (NUMA)

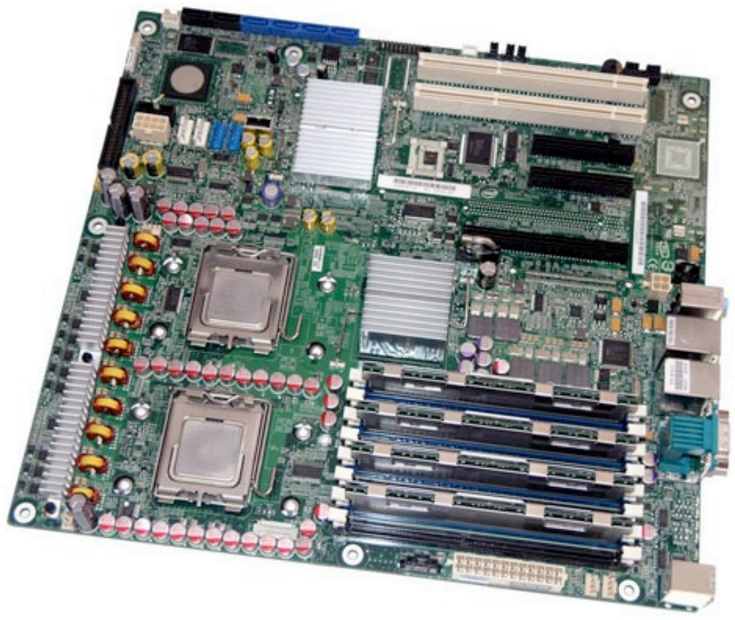
All processors can access any memory location, but... cost of memory access (latency or bandwidth) is different for different processors



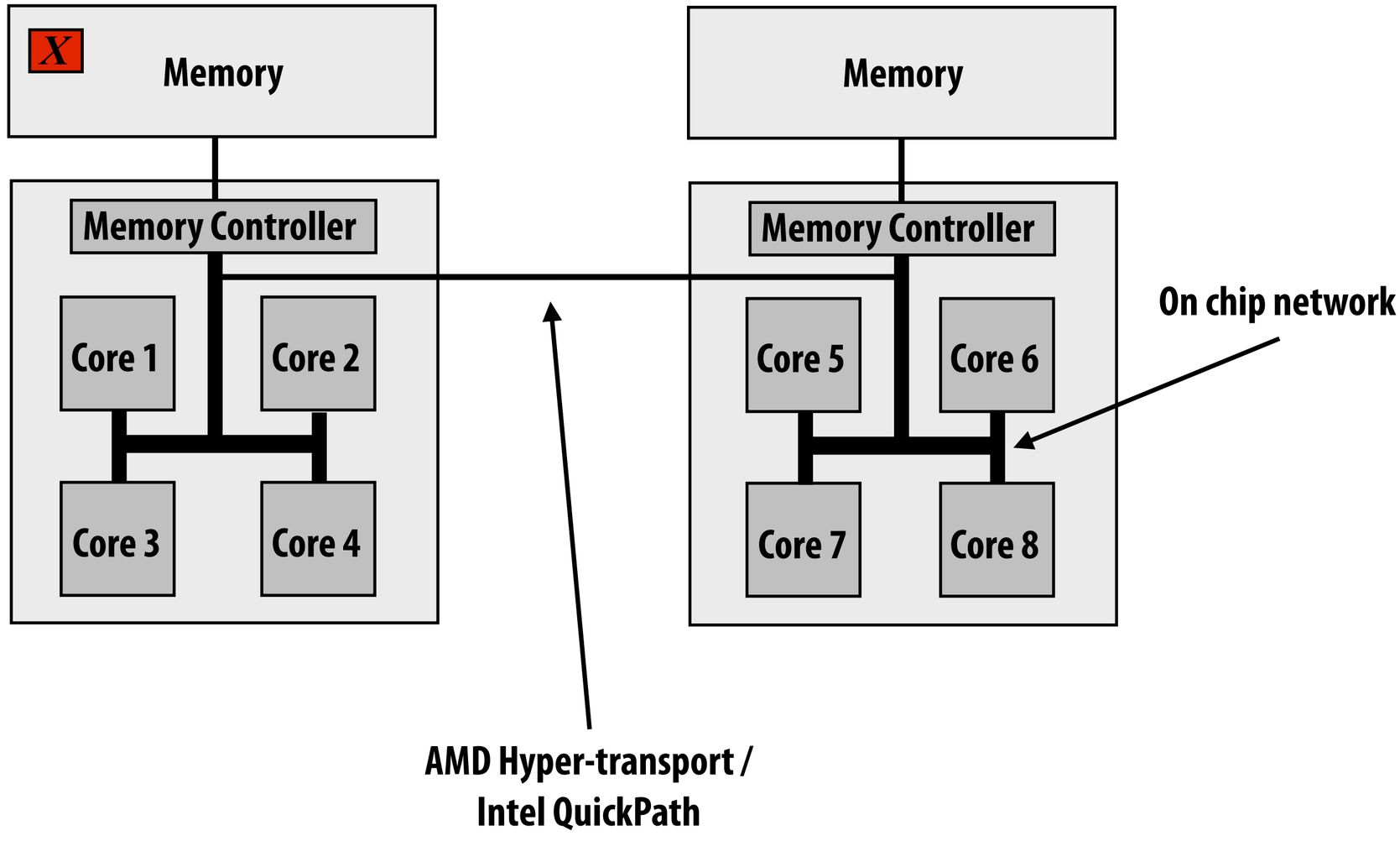
- Problem with preserving uniform access time: scalability
  - GOOD: costs are uniform, BAD: but memory is uniformly far away
- NUMA designs are more scalable
  - High bandwidth to local memory; BW scales with number of nodes if most accesses local
  - Low latency access to local memory
- Increased programmer effort: performance tuning
  - Finding, exploiting locality

Non-uniform memory access (NUMA)

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

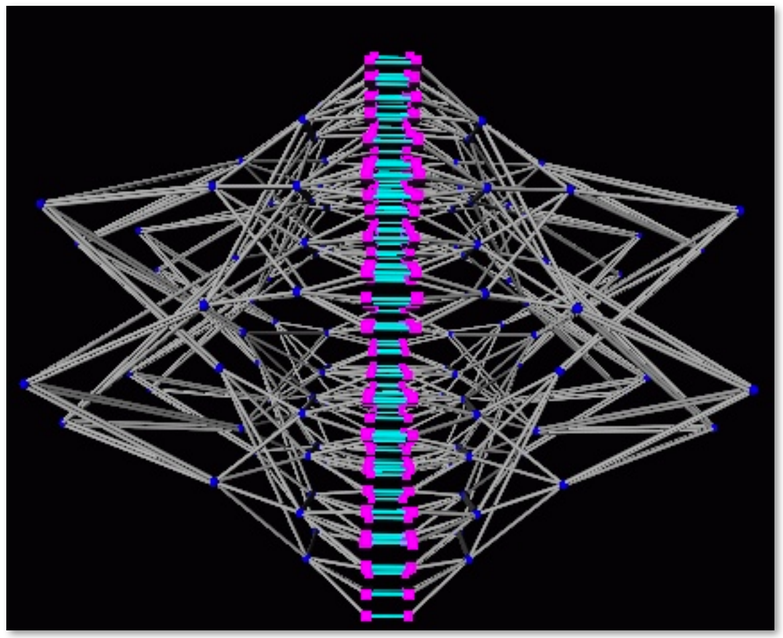


Example: modern dual-socket configuration



SGI Altix UV 1000 (PSC's Blacklight)

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



Fat tree

Image credit: Pittsburgh Supercomputing Center

Shared address space summary

- Communication abstraction
  - Threads read/write shared variables
  - Manipulate synchronization primitives: locks, semaphors, etc.
  - Logical extension of uniprocessor programming
    - But NUMA implementation requires reasoning about locality for perf

- Hardware support to make implementations efficient
  - Any processor can load and store from any address
  - NUMA designs more scalable than uniform memory access
    - Even so, costly to scale (see cost of Blacklight)

Message passing model (abstraction)

- Threads operate within <u>independent</u> address spaces
- Threads communicate by sending/receiving messages
  - Explicit communication via point-to-point messages
  - <u>send</u>: specifies buffer to be transmitted, recipient, optional message "tag"
  - <u>receive</u>: specifies buffer to store data, sender, and (optional) message tag
  - Messages may be synchronous or asynchronous

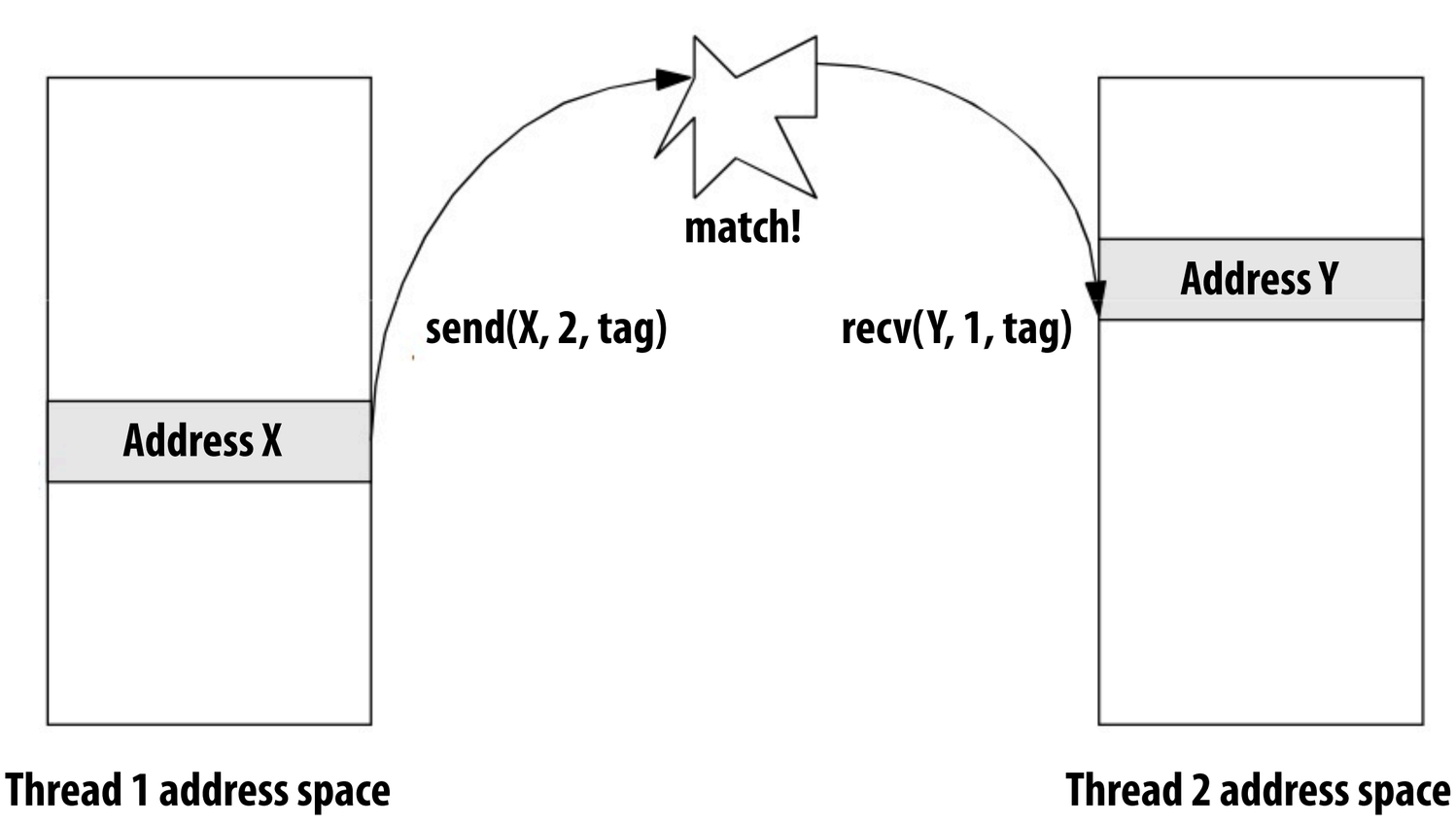
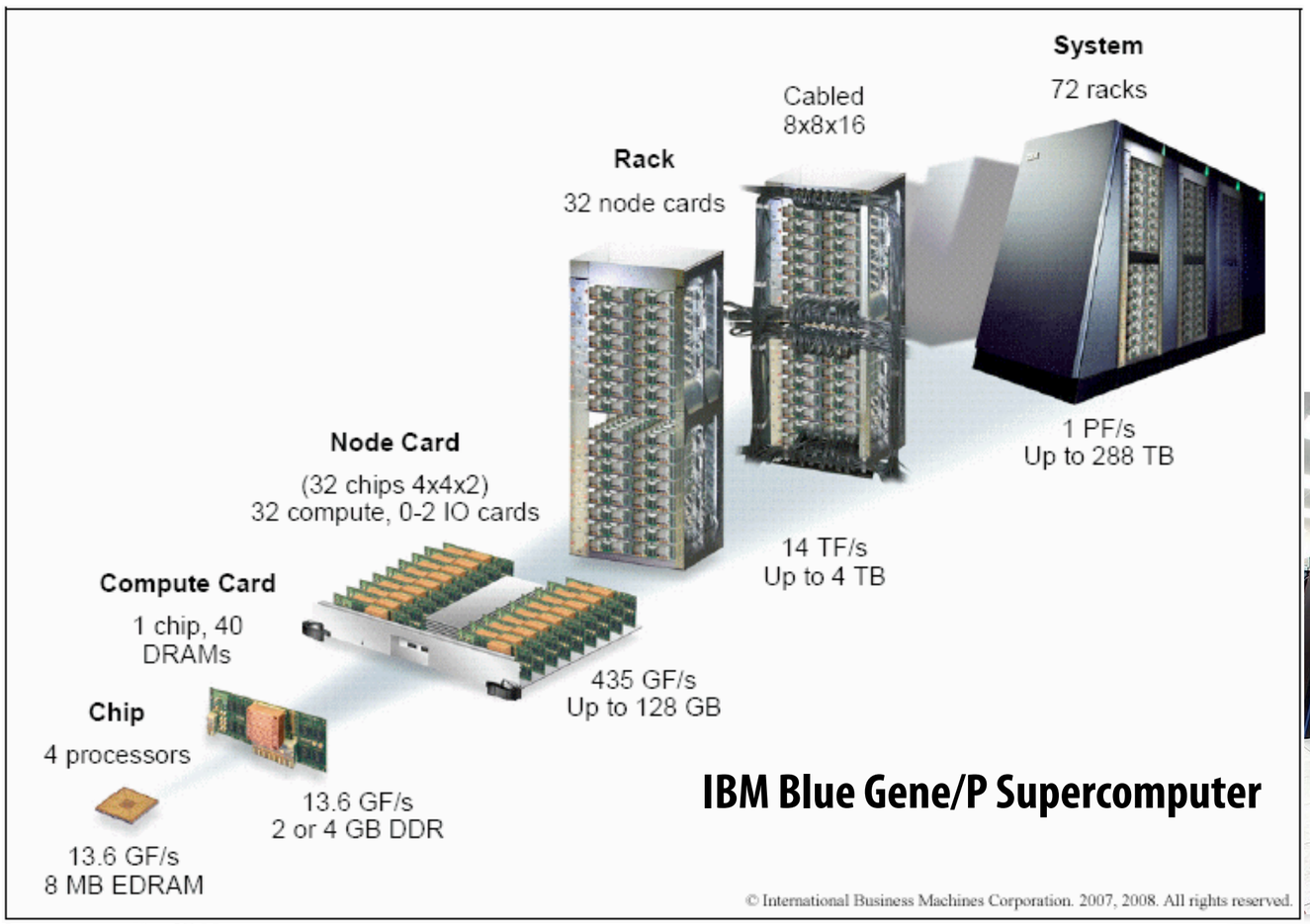


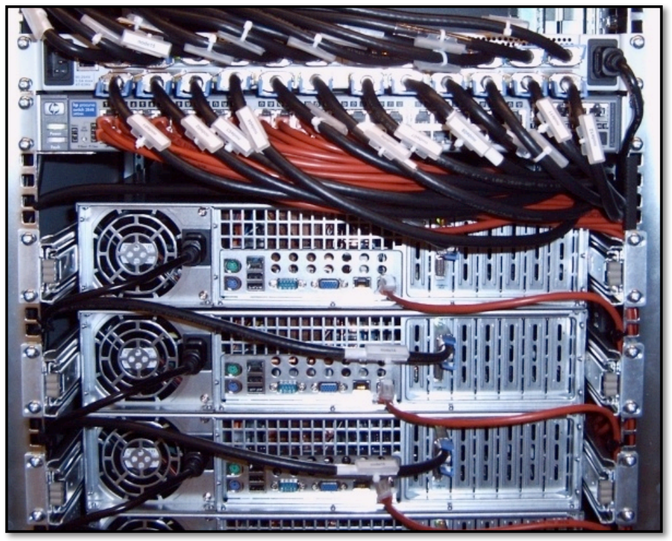
Image credit: Culler, Singh, and Gupta

Message passing (implementation)

- Popular library: MPI (message passing interface)
- Challenges: buffering messages (until application initiates receive), minimizing cost of memory copies
- Hardware need not implement system-wide loads and stores
  - Connect complete (often commodity) systems together
  - Parallel programs for clusters!



IBM Blue Gene/P Supercomputer



Cluster of workstations (Infiniband network)

Image credit: IBM

Correspondence between programming models and machine types is fuzzy

- Common to implement message passing abstractions on machines that support a shared address space in hardware

- 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
  - Page-fault handler issues appropriate network requests

- Keep in mind what is the programming model (abstractions used to specific program) and what is the HW implementation

The data-parallel model

Data-parallel model

- Rigid computation structure
- Historically: same operation on each element of an array
  - Matched capabilities of 80's SIMD supercomputers
    - Connection Machine (CM-1, CM-2): thousands of processors, one instruction
    - And also Cray supercomputer vector processors
      - `Add(A, B, n)` fi this was one instruction on vectors A, B of length n
- Matlab is another good example: A + B
  (A, B are vectors of same length)
- Today: often takes form of SPMD programming
  - `map(function, collection)`
  - Where `function` may be a complicated sequence of logic (e.g., a loop body)
  - Application of `function` to each element of collection is <u>independent</u>
    - In pure form: no communication between iterations of map
  - Synchronization is implicit at the end of the map

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);
```

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

`foreach` construct is a map

Collection code is mapping over is implicitly defined by array indexing logic

```
// 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];
   }
}
```

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];
   }
}
```

Also a valid program!

Takes absolute value of elements of x, repeats them twice in output vector y

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];
    }
}
```

This program is non-deterministic!

Possibility 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)

Data parallelism the more formal way

Note: this is not ISPC syntax

```
// main program:
const int N = 1024;

stream<float> x(N);  // define collection
stream<float> y(N);  // define collection

// initialize N elements of x here

 // map absolute_value onto x, y
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 programing model

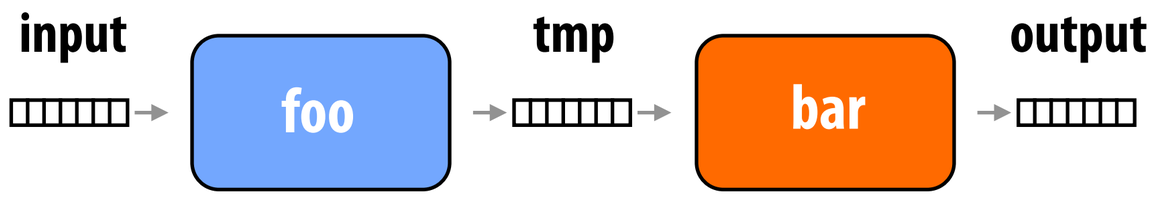
Streams: collections of elements. Elements can be processed independently

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

Think of kernel inputs, outputs, temporaries for each invocation as a private address space

Stream programming benefits

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

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



Functions really are side-effect free!
(cannot write a non-deterministic program)

Program data flow is known:

Predictable data access facilitates prefetching. Inputs and outputs of each invocation are known in advance: prefetching can be employed to hide latency.

Producer-consumer locality. Can structure code so that outputs of first kernel feed immediately into second kernel. Values are stored in on-chip buffers/caches and never written to memory! Save bandwidth!

These optimizations are responsibility of stream program compiler. Requires sophisticated compiler analysis.

Stream programming drawbacks

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

stream_repeat(2, input, tmp);
absolute_value(tmp, output);
```

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

In practice: cross fingers and hope compiler generates code intelligently

Kayvon's experience:

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

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

```
// 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;
   }
}
```

Gather/scatter:

Two key data-parallel communication primitives

Map absolute_value onto stream produced by gather:

```
// main program:
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:

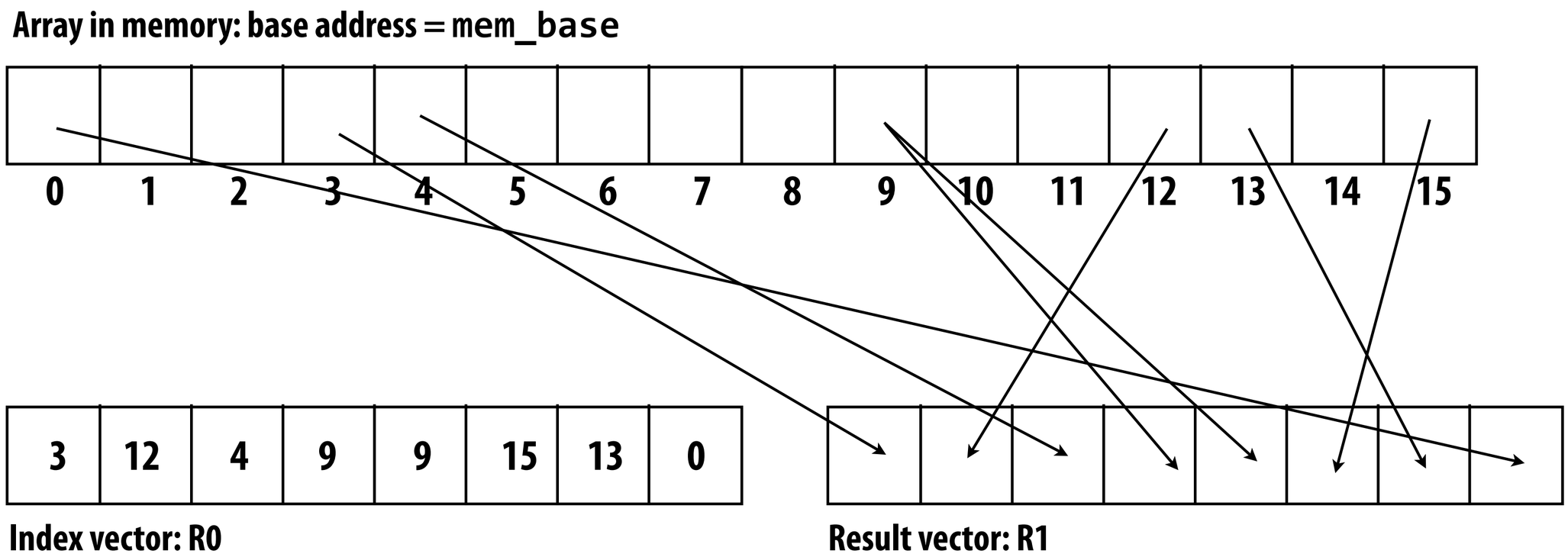
```
// main program:
const int N = 1024;
stream<float> input(N);
stream<int> indices;
stream<float> tmp_output(N);
stream<float> output(N);



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 indices specified by R0."



Gather supported with AVX2 in 2013
But does not directly support SIMD scatter (must implement as scalar loop)

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

Data-parallel model summary

- Data-parallelism is about imposing program structure
- In spirit, map a single program onto a large collection of data
  - Functional: side-effect free execution
  - No communication among invocations
- In practice that's how many programs work
- But... most practical parallel languages do not <u>enforce</u> this
  - OpenCL, CUDA, ISPC, etc.
  - Choose flexibility/familiarity of imperative syntax over safety and complex compiler optimizations required for functional syntax
  - It's been their key to success (and the recent adoption of parallel programming)
  - Hear that PL folks! (sure, functional <u>thinking</u> is great, but structure should enable achieving performance implementations, not hinder hinder it)

Three parallel programming models

- **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 scale
- **Message passing**
  - Structured communication as messages
  - Often harder to get first correct program than shared address space
  - Structure often helpful in getting to first correct, scalable program
- **Data parallel**
  - Structure computation as a big map
  - 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 trend: hybrid programming models

- Shared address space within a multi-core node of a cluster, message passing between nodes
  - Very, very common in practice
  - Use convenience of shared address space where it can be implemented efficiently (within a node)

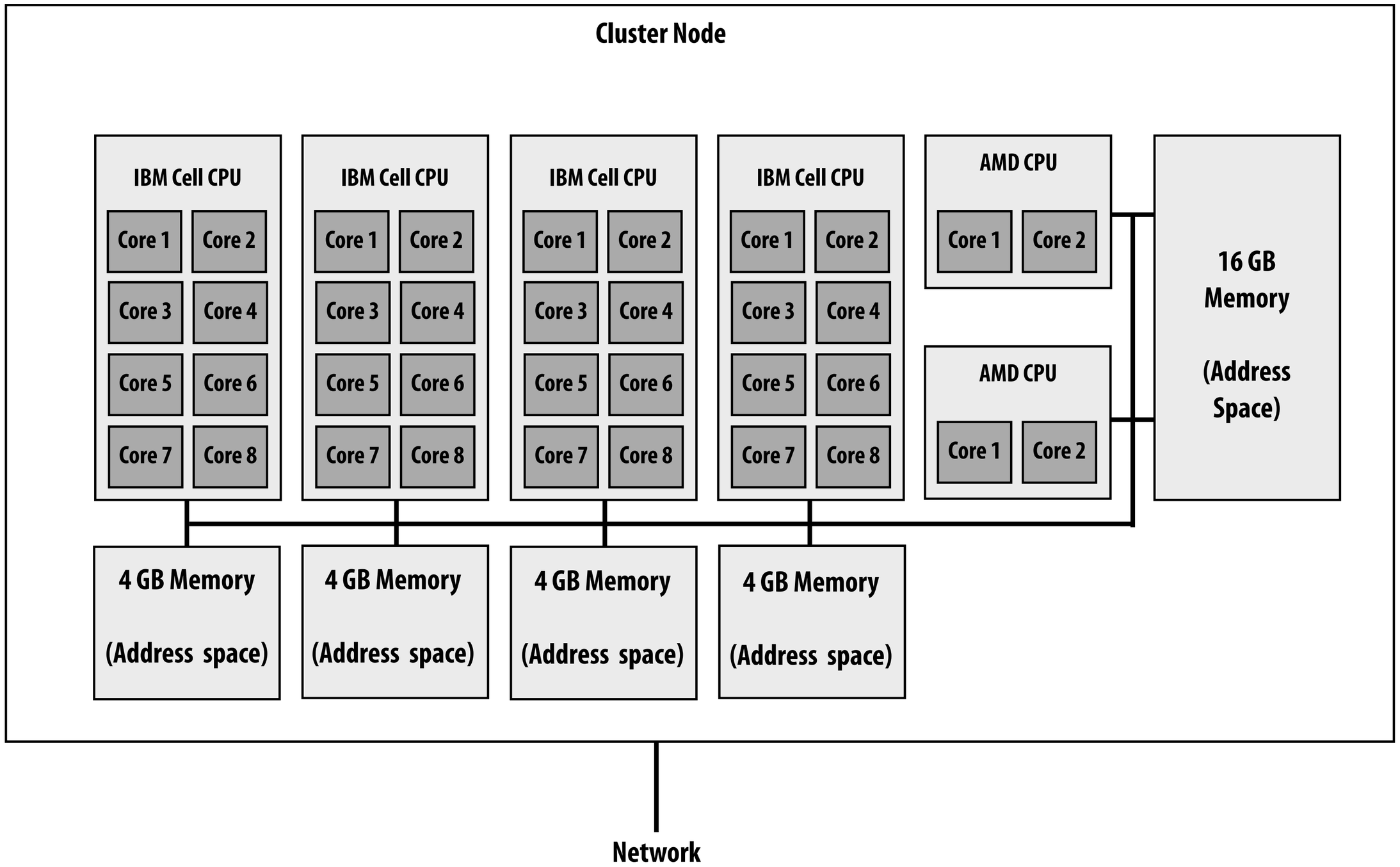
- Data-parallel programming models support synchronization primitives in kernels (CUDA, OpenCL)
  - Permits limited forms of communication

- 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.

Los Alamos National Laboratory: Roadrunner

Fastest computer in the world in 2008 (no longer true)

3,240 node cluster. Heterogeneous nodes.



Summary

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

- But restrictions imposed by abstractions are designed to reflect realities of hardware communication costs
  - Shared address space machines
  - Messaging passing machines
  - It is desirable to keep "abstraction distance" low so programs have predictable performance, but want it high enough for code flexibility/portability

- In practice, you'll need to be able to think in a variety of ways
  - Modern machines provide different types of communication at different scales
  - Different models fit the machine best at the various scales
  - Optimization may require you to think about implementations, not just abstractions