Lecture 3:
Communication Architectures and Parallel Programming models:

CMU 15-418: Parallel Computer Architecture and Programming (Spring 2012)

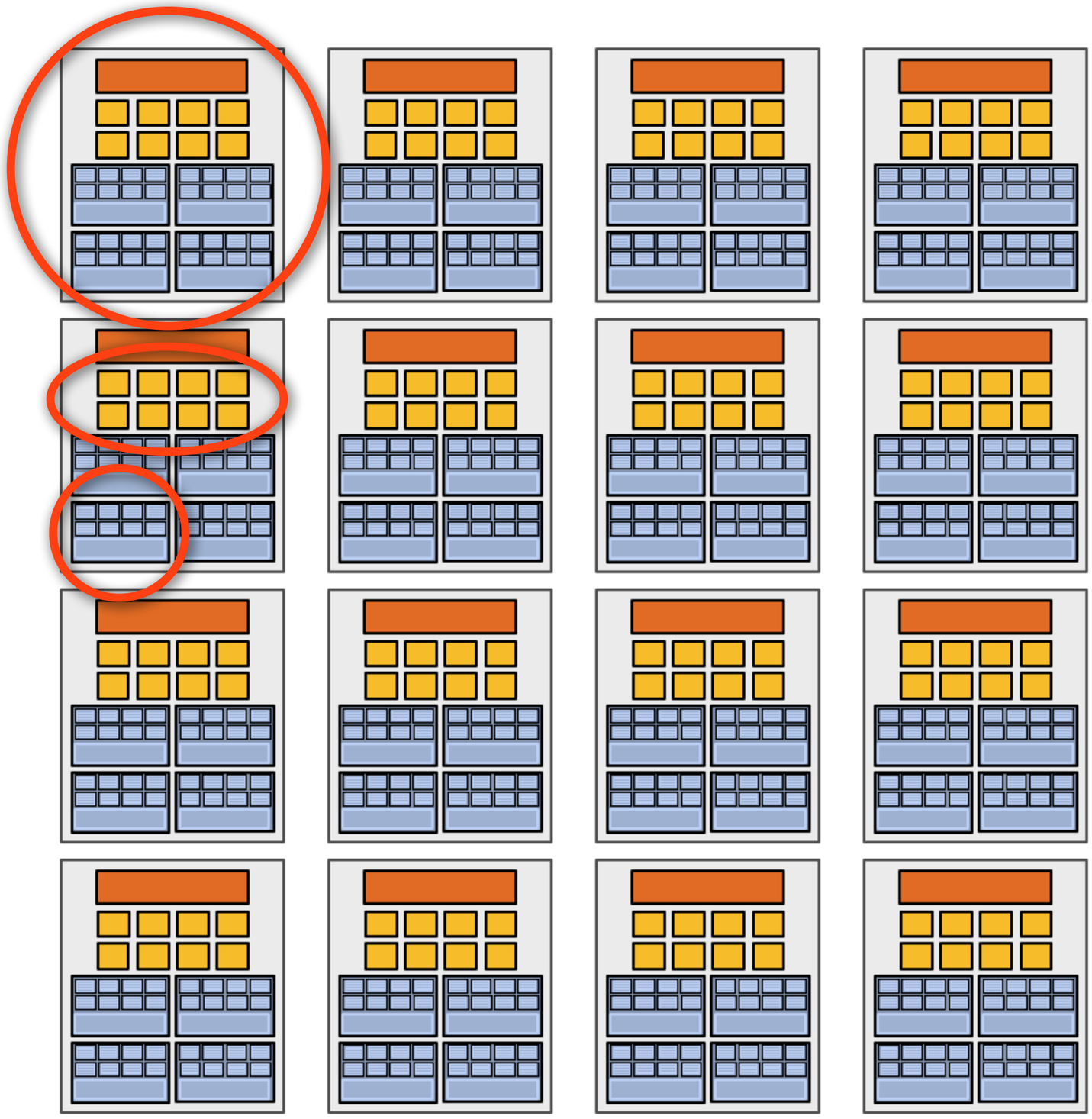
Announcements

- **Office hours**
  - Fatahalian: Tue/Thurs 1:30-2:30PM, GHC 7005
  - Papamichael: Wed 1-2PM, HH A-312
  - Mu: Mon 4-5PM, West Wing cluster

- **Assignment 1 out today!**
  - Due: Jan 31st
  - No late handin (1 week is more than enough time)

- **Course discussions, message boards hosted on piazza.com**
  - I've heard good things
  - Please go to piazza.com and enroll in 15-418

Review: Kayvon's fictitious multi-core chip



We talked about forms of parallel/concurrent execution.

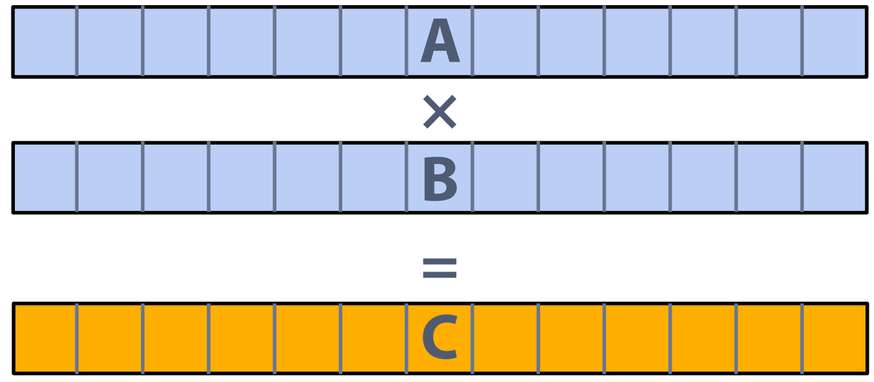
Finishing up from last time

Review: thought experiment

Task: element-wise multiplication of two vectors A and B

Assume vectors contain millions of elements

1. **Load input A[i]**
2. **Load input B[i]**
3. **Compute A[i] × B[i]**
4. **Store result into C[i]**



Three memory operations (12 bytes) for every MUL

NVIDIA GTX 480 GPU can do 480 MULs per clock (1.4 GHz)

Need ~8.4 TB/sec of bandwidth to keep functional units busy

~ 2% efficiency… but 8x faster than CPU!

(3 GHz Core i7 quad-core CPU: similar efficiency on this computation)

Bandwidth limited!

If processors request data at too high a rate, the memory system cannot keep up.

No amount of latency hiding helps this.

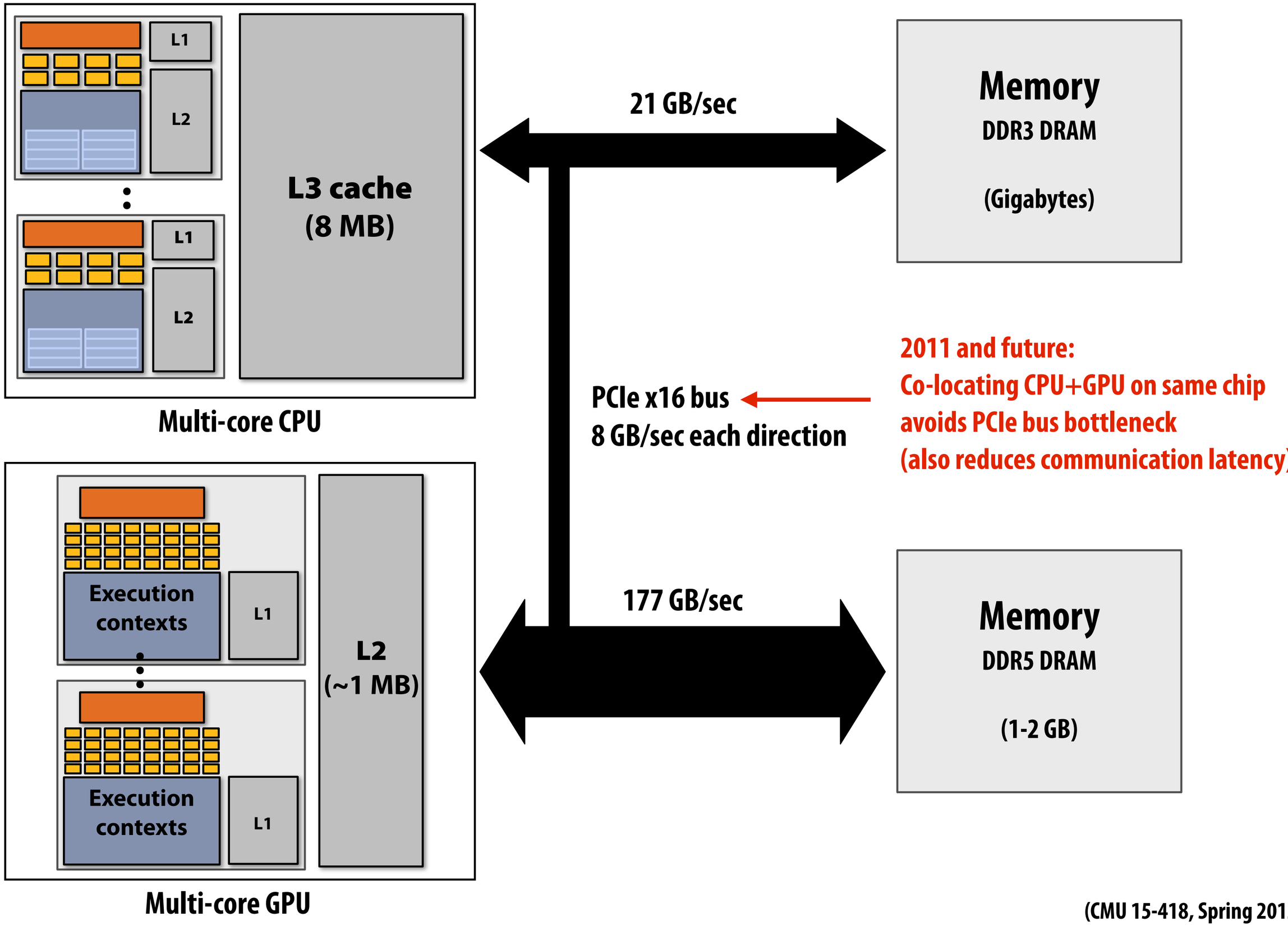
Overcoming bandwidth limits are a common challenge for application developers on throughput-optimized systems.

Bandwidth is a critical resource

Performant parallel programs will:

- Fetch data from memory less often
  - Reuse data in a thread (traditional locality optimizations)
  - Share data across threads (cooperation)
- Request data less often (instead, do more math: it is "free")
  - Term: "arithmetic intensity" - ratio of math to data access

Entire system view: CPU + discrete GPU



Programming with ISPC
(quick tutorial for assignment 1)

ISPC

- Intel SPMD Program Compiler (ISPC)
- SPMD: single *program* multiple data
- http://ispc.github.com/

sin(x) in ISPC

Compute sin(x) using Tailor 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 *= (j+3) * (j+4);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

SPMD abstraction:

Spawn “gang” of ISPC program instances

All instances run ISPC code in parallel

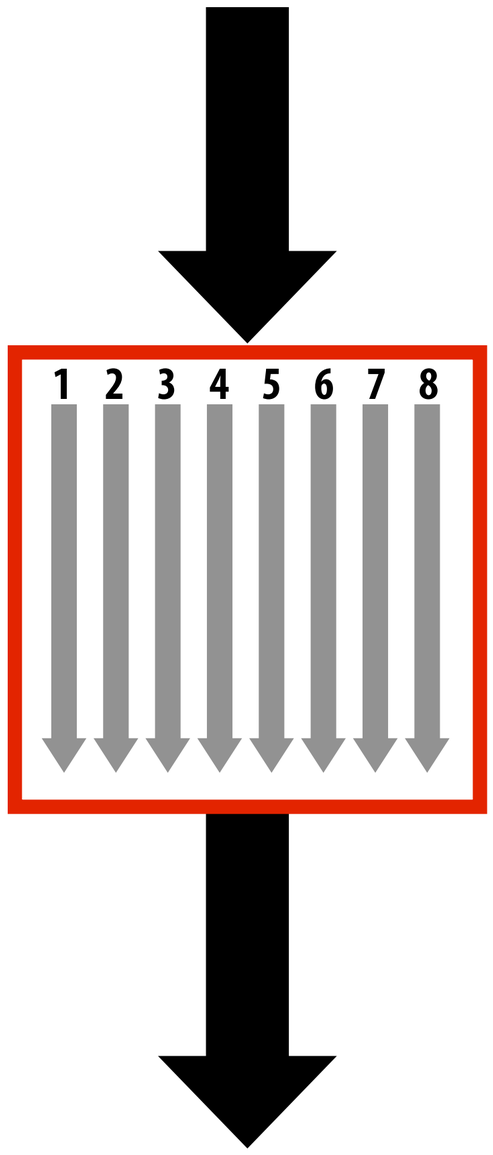
Upon return, all instances have completed

sin(x) in ISPC

Compute sin(x) using Tailor 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 abstraction:

Spawn “gang” of ISPC program instances
All instances run ISPC code in parallel
Upon return, all instances have completed

SIMD implementation:

Number of instances in a gang is the SIMD width (or a small multiple of)
ISPC compiler generates binary (.o) with SIMD instructions
C++ code links against object as usual

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 *= (j+3) * (j+4);
            sign *= -1;
        }
        result[i] = 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.

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[i] = value;
    }
}
```

sin(x) in ISPC

Compute sin(x) using tailor 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 *= (j+3) * (j+4);
            sign *= -1;
        }
        result[i] = value;
    }
}
```

`foreach`: key language construct

Declares parallel loop iterations

ISPC assigns iterations to program instances in gang

(ISPC implementation will perform static assignment, but nothing in the abstraction prevents a dynamic assignment)

ISPC: abstraction vs. implementation

- **Single program, multiple data (SPMD) programming model**
  - This is the programming abstraction

- **Single instruction, multiple data (SIMD) implementation**
  - Compiler emits vector instructions
  - 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)
  - What does `sumall1` do?
  - What does `sumall2` do?
  - Which one is correct?

```
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;
}
```

Today

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

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

- **Focus on communication and cooperation**

- **Reading: Textbook section 1.2**

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

Parallel Applications

Abstractions for describing parallel computation

Abstractions for describing communication

"Programming model" (way of thinking about things)

Language or API primitives/mechanisms

Compiler and/or runtime

System call API

Operating system support

Hardware Architecture (HW/SW boundary)

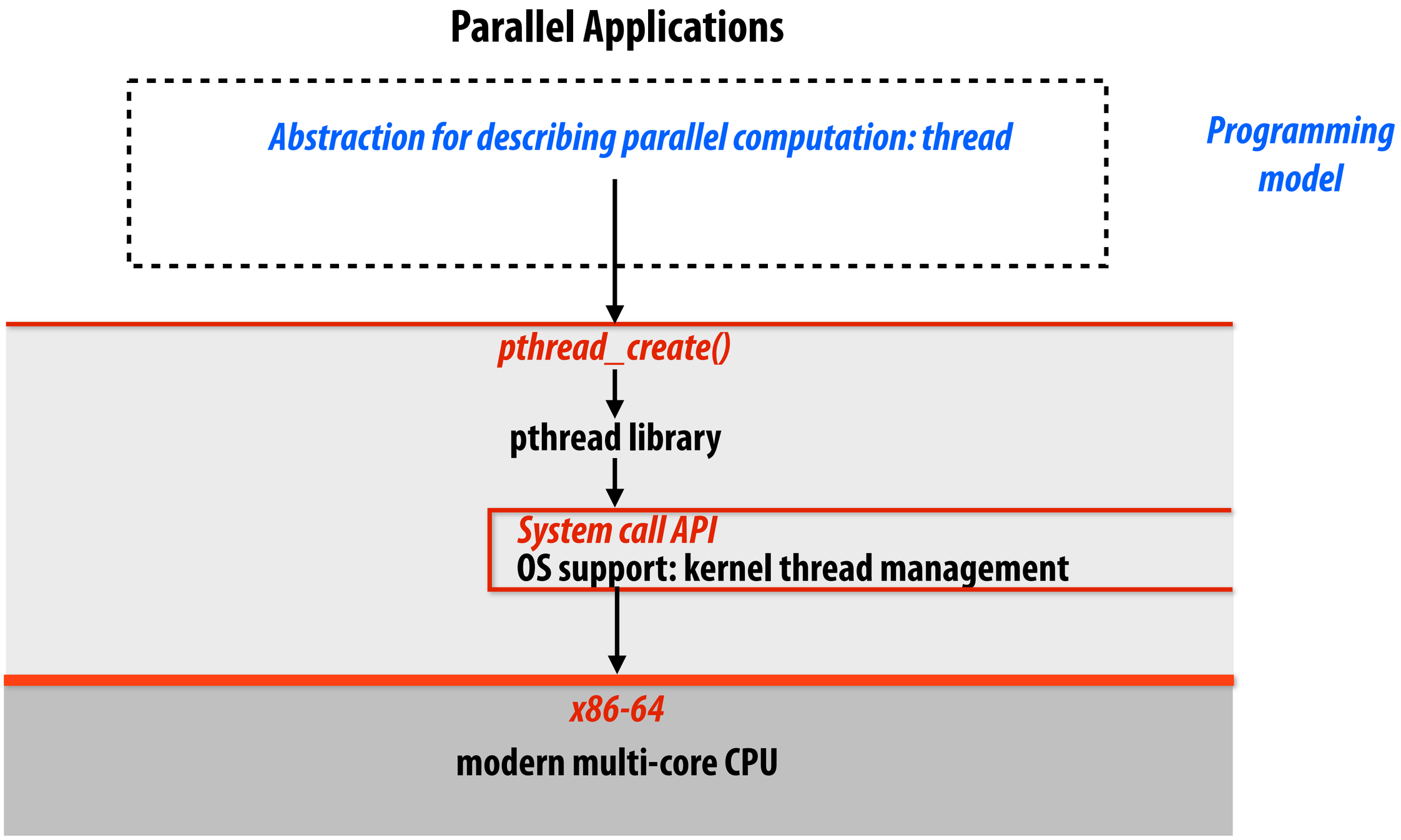
Micro-architecture (HW implementation)

Blue italic text: concept

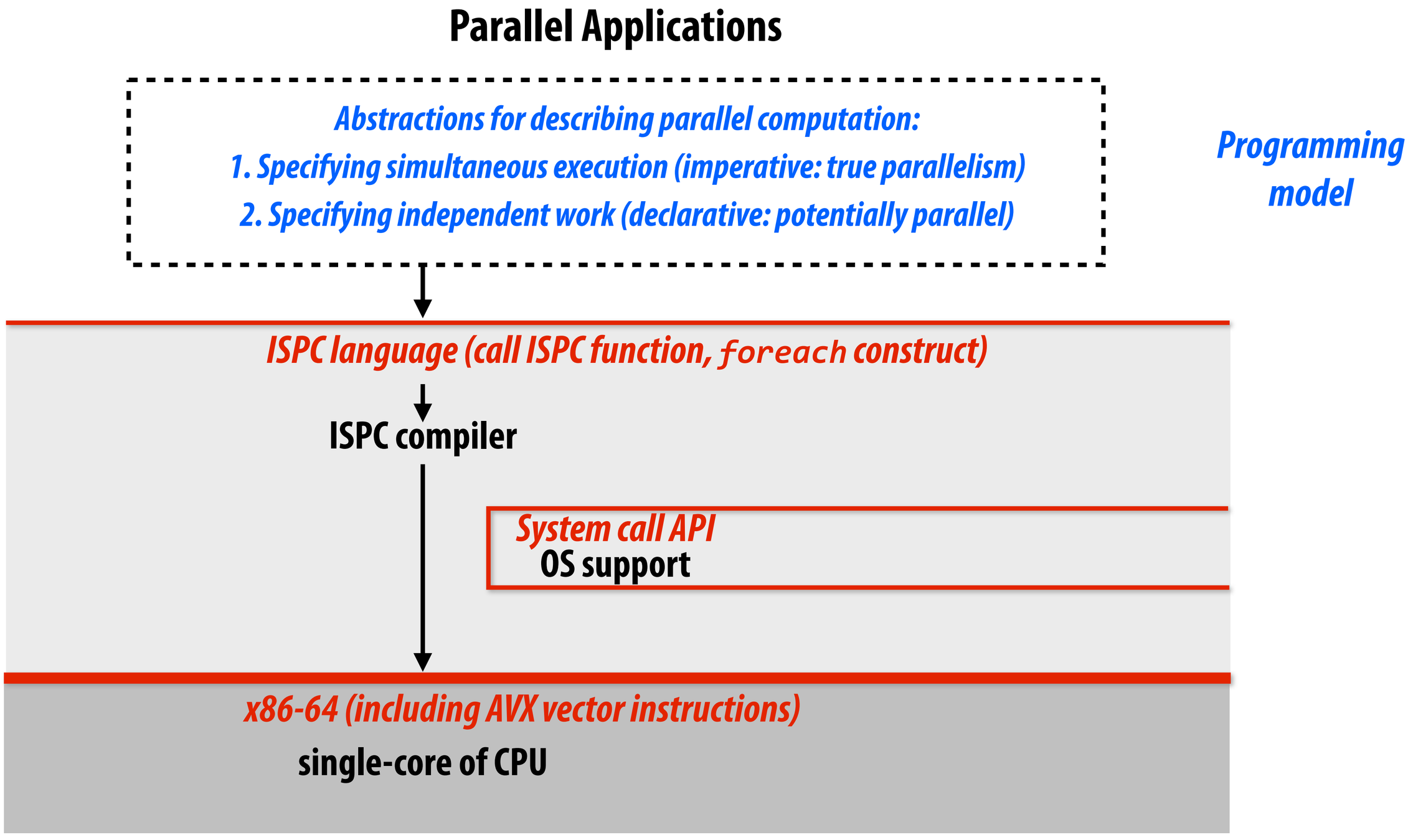
Red italic text: system abstraction

Black text: system implementation

Example: expressing parallelism (pthreads)



Example: expressing parallelism (ISPC)



Note: ISPC has additional language primitives for multi-core execution (not discussed here)

Three models of communication

1. Shared address space

2. Message passing

3. Data parallel

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. Thread 2 reads X (observes update)
  - Manipulating synchronization primitives
    - e.g., mutual exclusion using locks

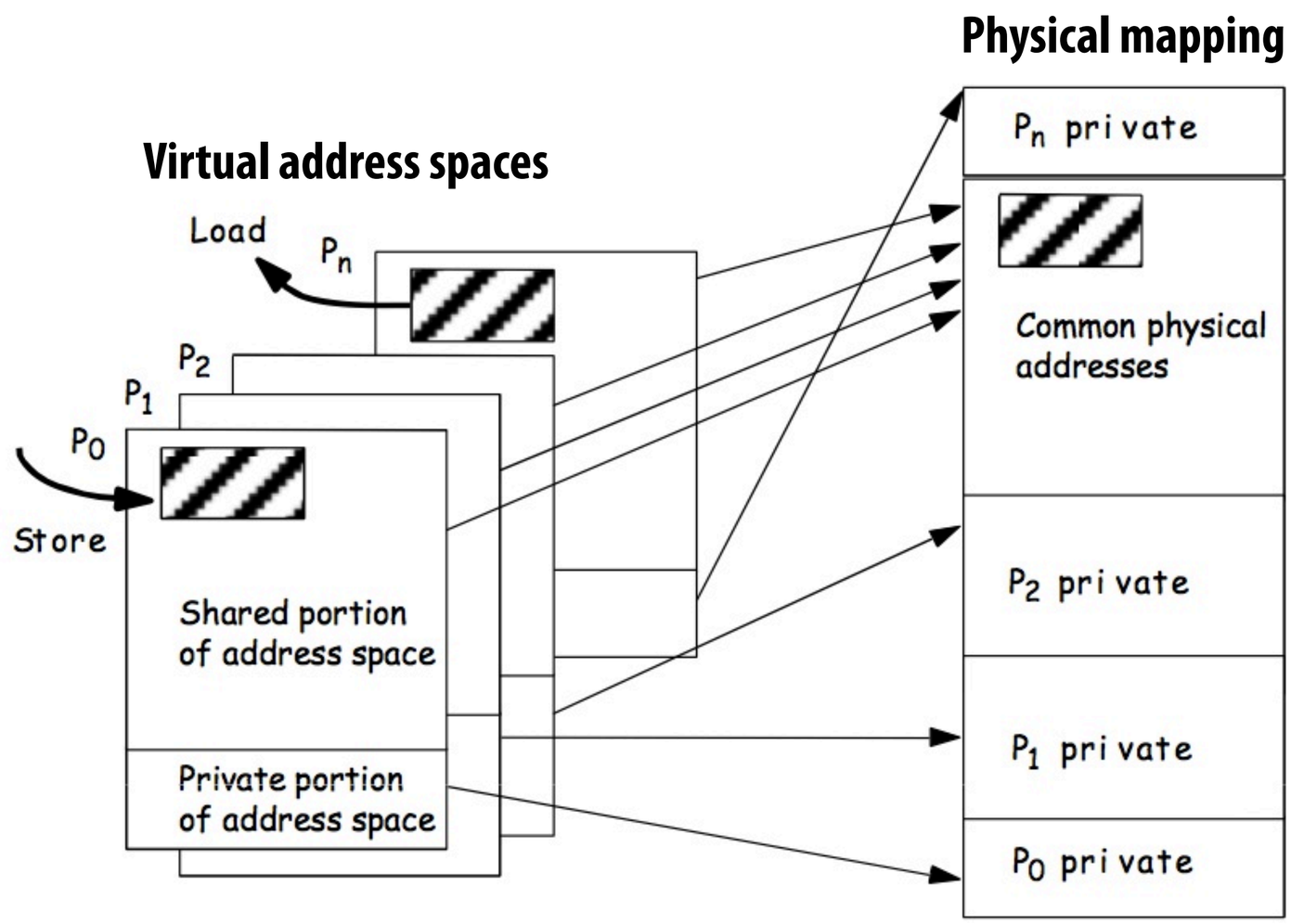
- Natural extension of sequential programming model
  - In fact, all our discussions have assumed a shared address space so far

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

Shared address space (implementation)

- Option 1: threads share an address space (all data is sharable)

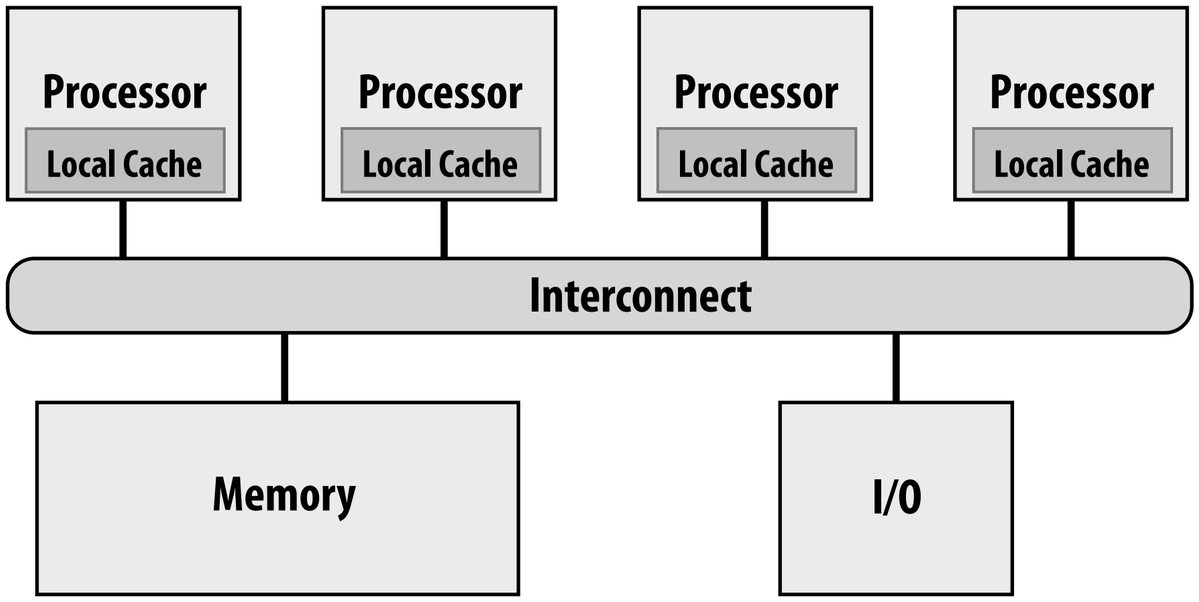
- Option 2: each thread has its own virtual address space, shared portion of address spaces maps to same physical location
(like processes, described this way in book)



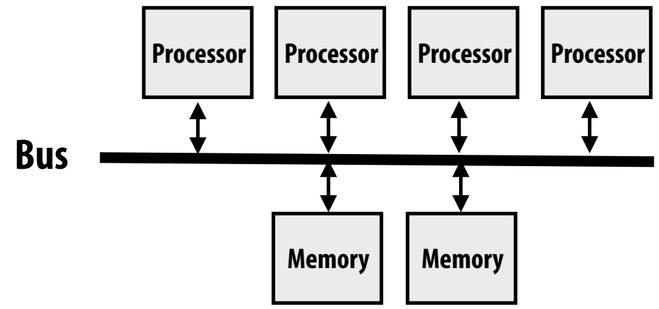
Shared address space HW implementation

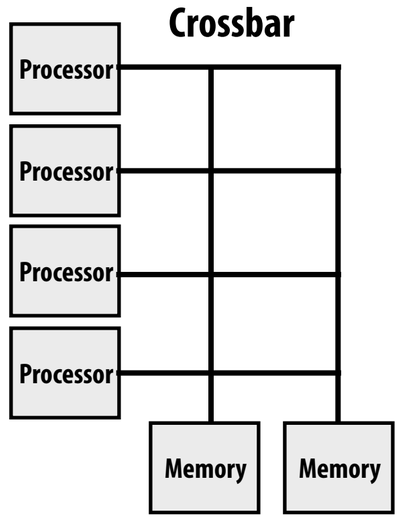
Any processor can directly reference any memory location

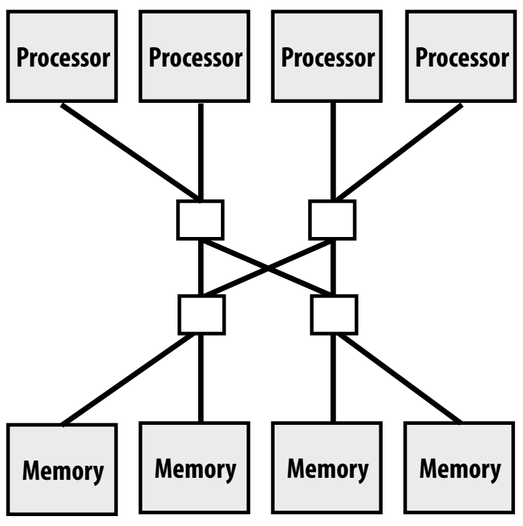
"Dance-hall" organization



Interconnect examples







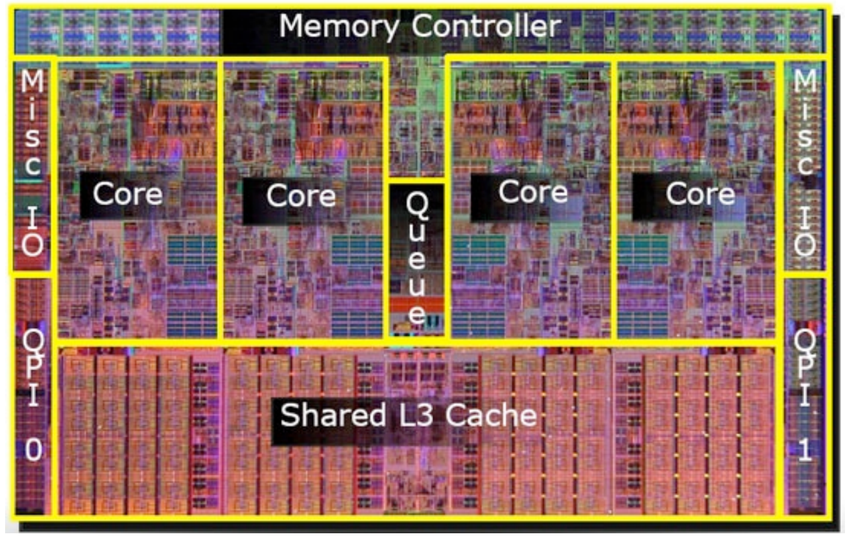
Multi-stage network

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

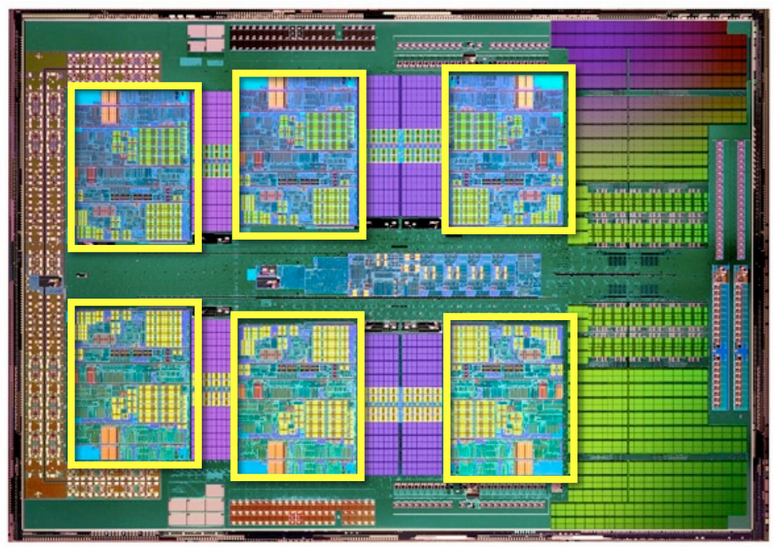
(* caching introduces non-uniform access times, but we'll talk about that later)

Shared address space architectures

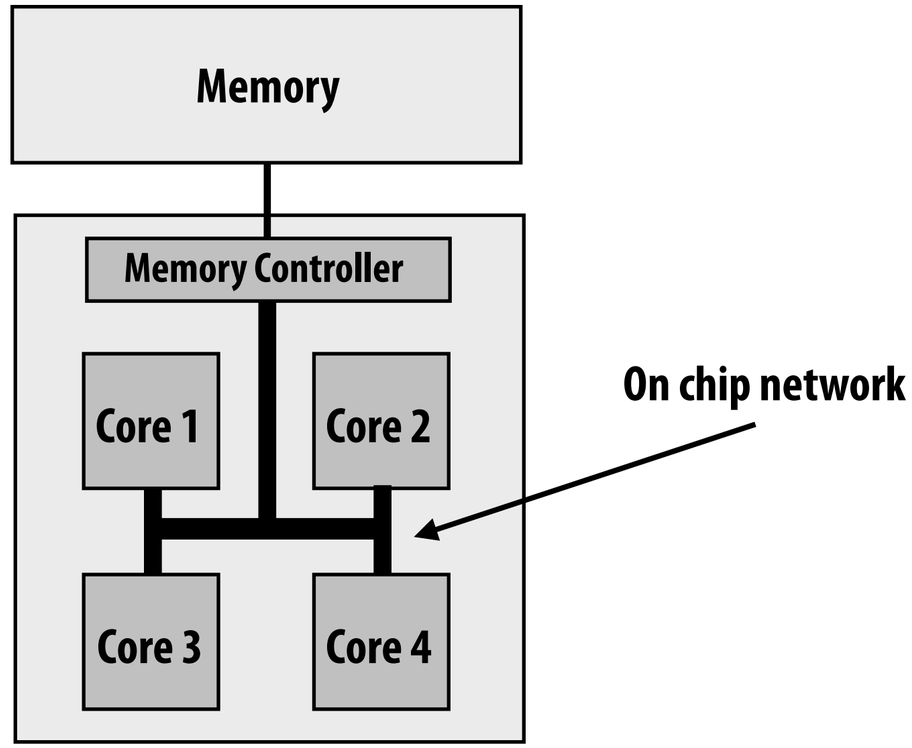
Commodity x86 examples



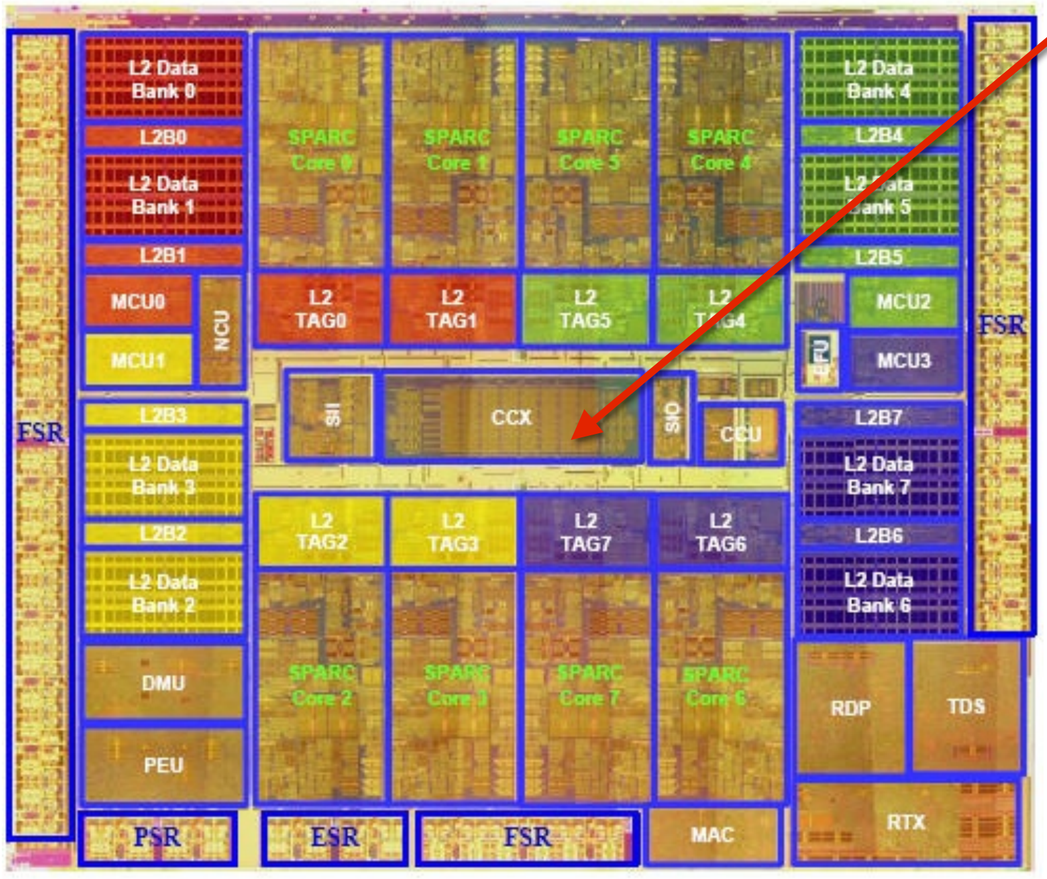
Intel Core i7 (quad core)
(network is a ring)



AMD Phenom II (six core)

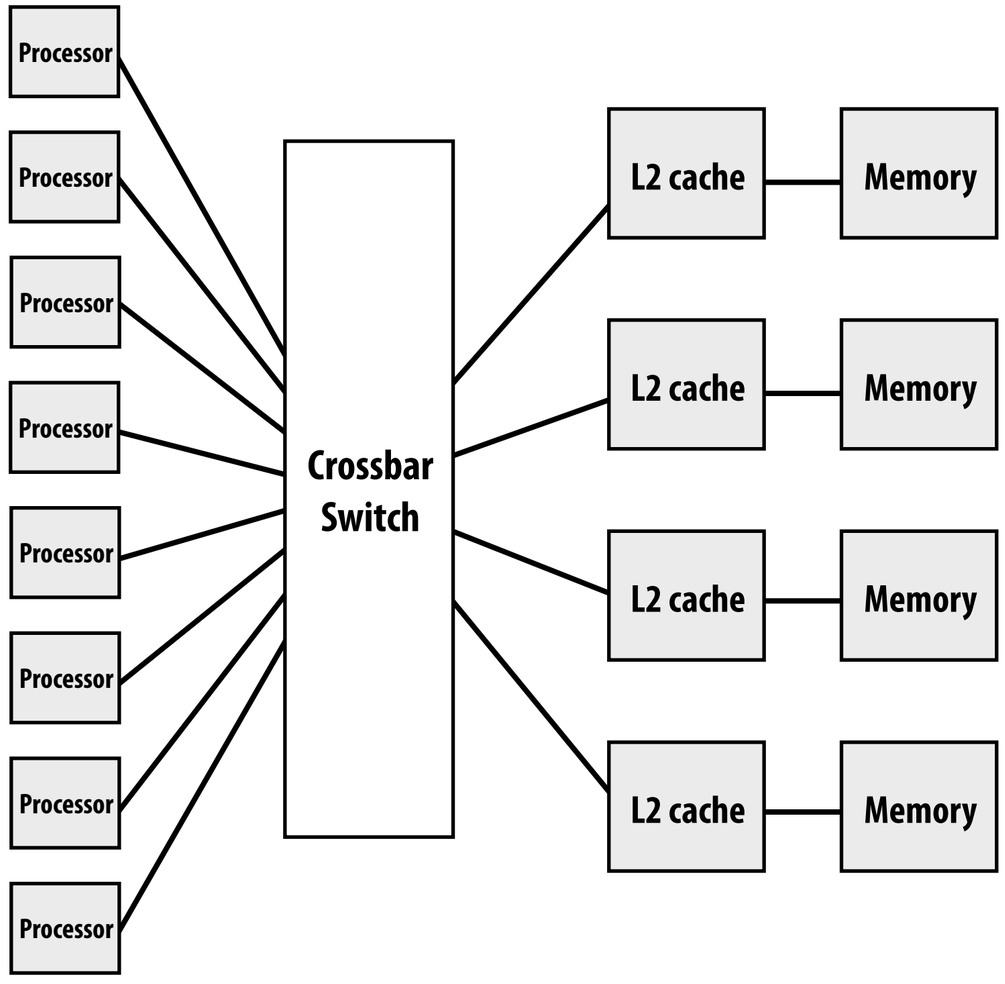


SUN Niagara 2



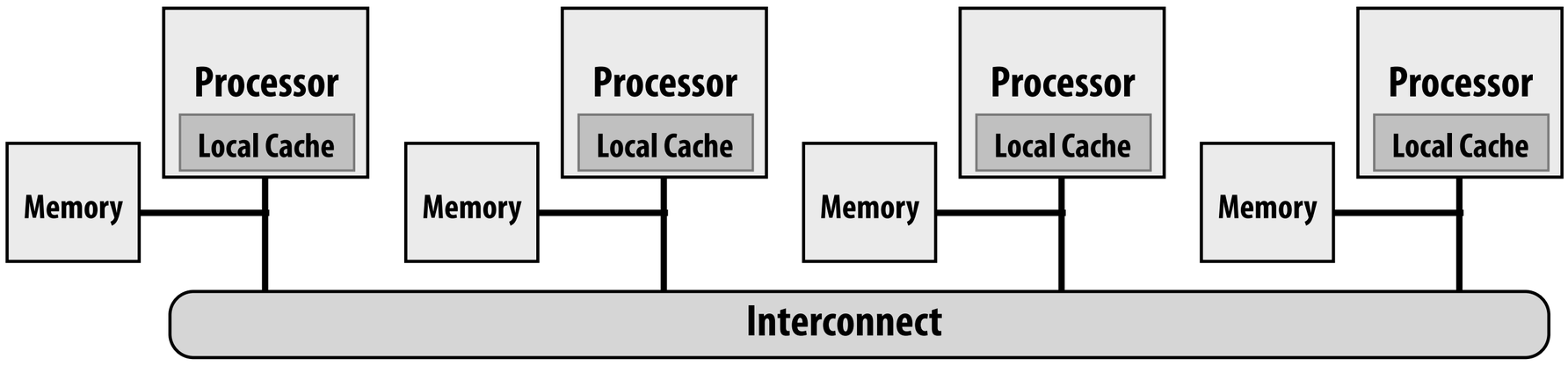
Eight cores

Note size of crossbar: about die area of one core



Non-uniform memory access (NUMA)

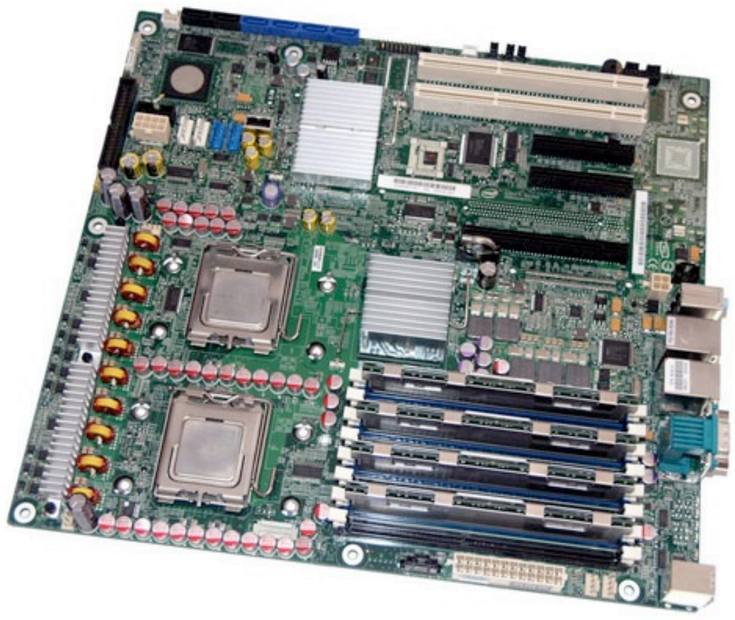
All processors can access any memory location, but cost of memory access is different for different processors



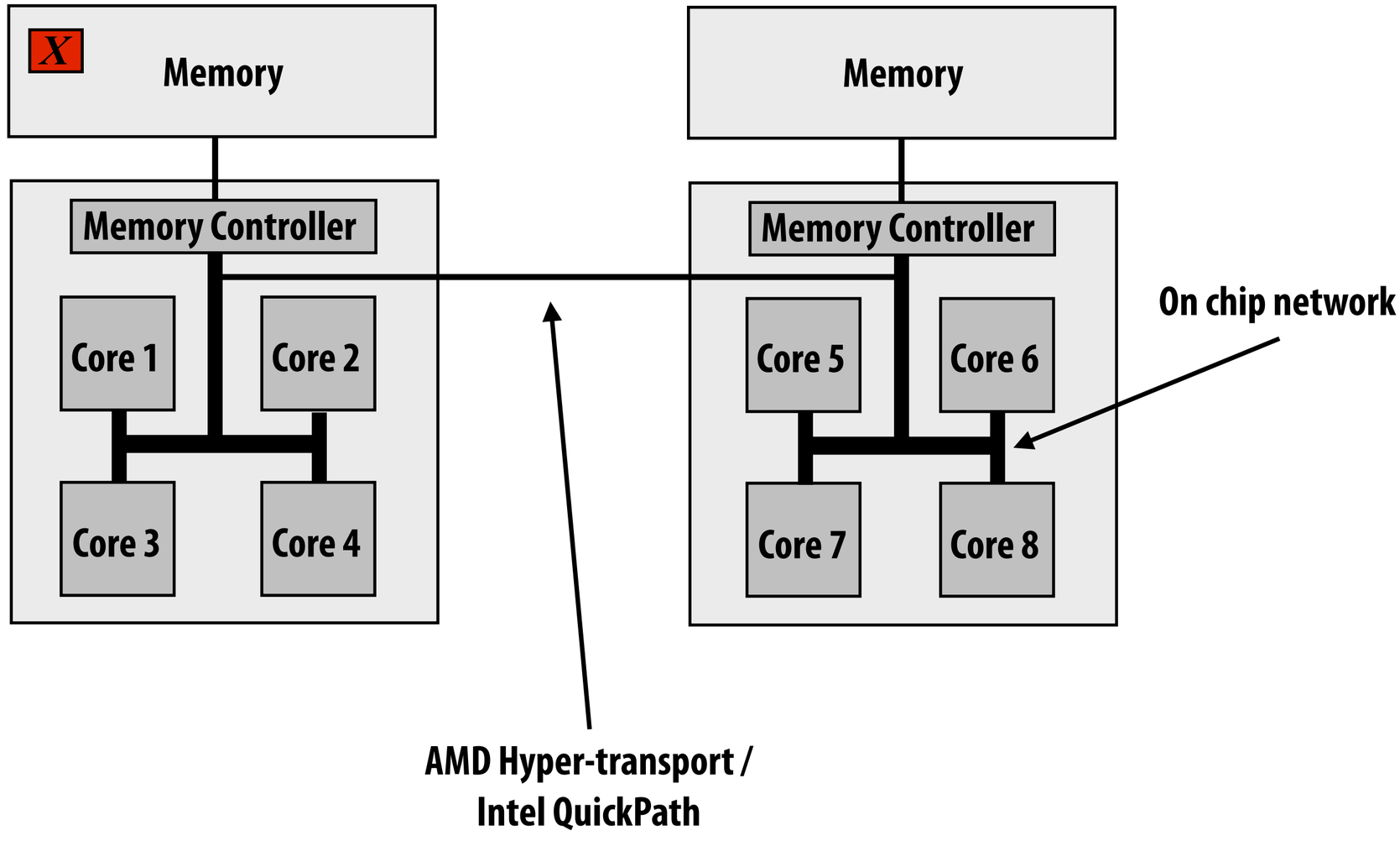
- Problem with preserving uniform access time: scalability
  - Costs are uniform, but memory is uniformly far away
- NUMA designs are more scalable
  - High bandwidth to local memory; BW scales with number of nodes if most accesses local
  - Low latency access to local emory
- Increased programmer effort: 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

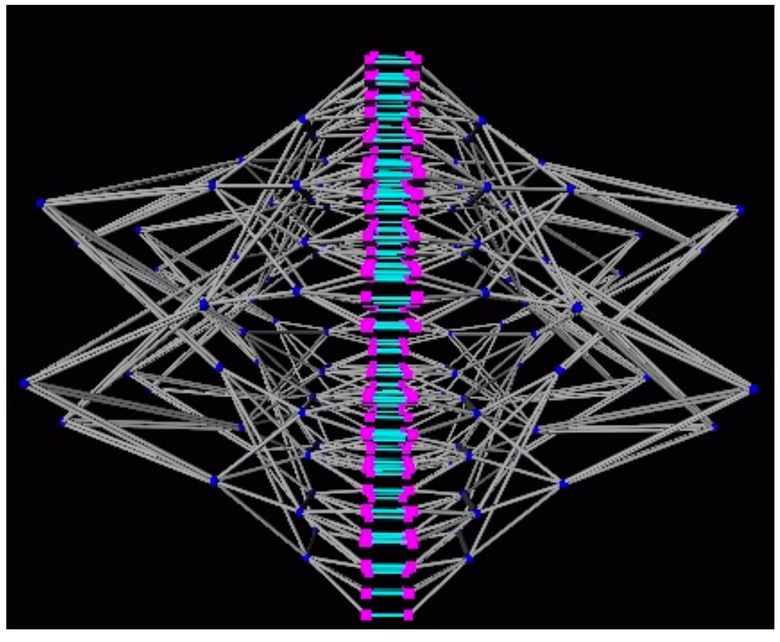


Example: modern dual-socket configuration

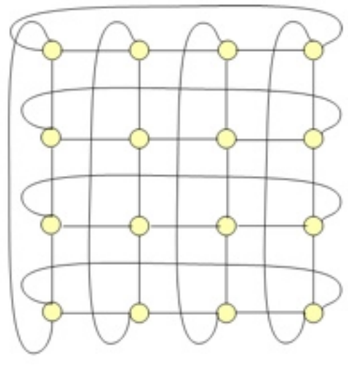


SGI Altix UV 1000 (PSC Blacklight)

- 256 blades, 2 CPUs per blade, 8 cores per CPU = 4K cores
- Single shared address space
- Interconnect
  - Fat tree of 256 CPUs (15 GB/sec links)
  - 2D torus to scale up another factor of 2



Fat tree



2D torus

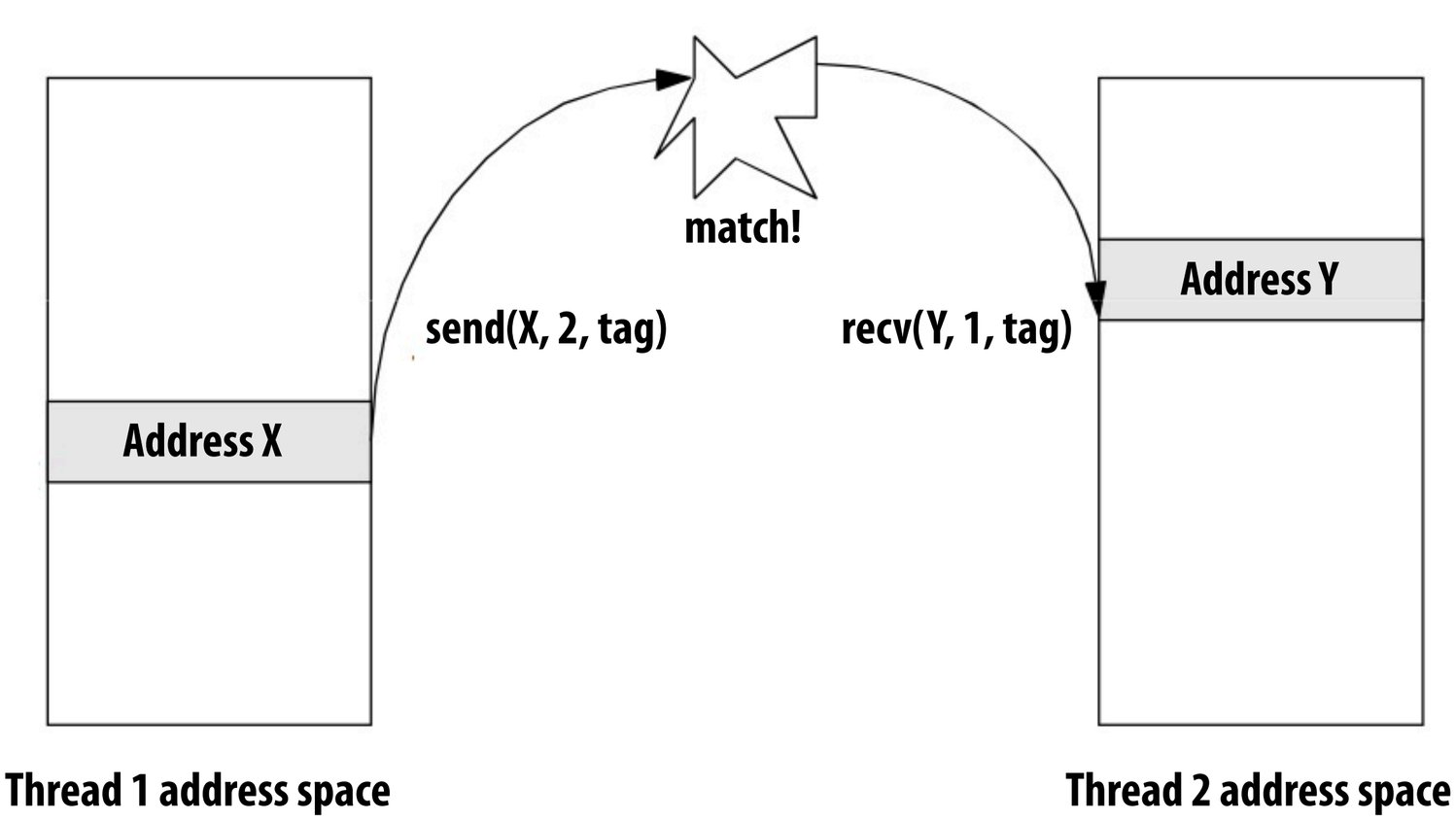
Shared address space summary

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

- **Hardware support**
  - 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)

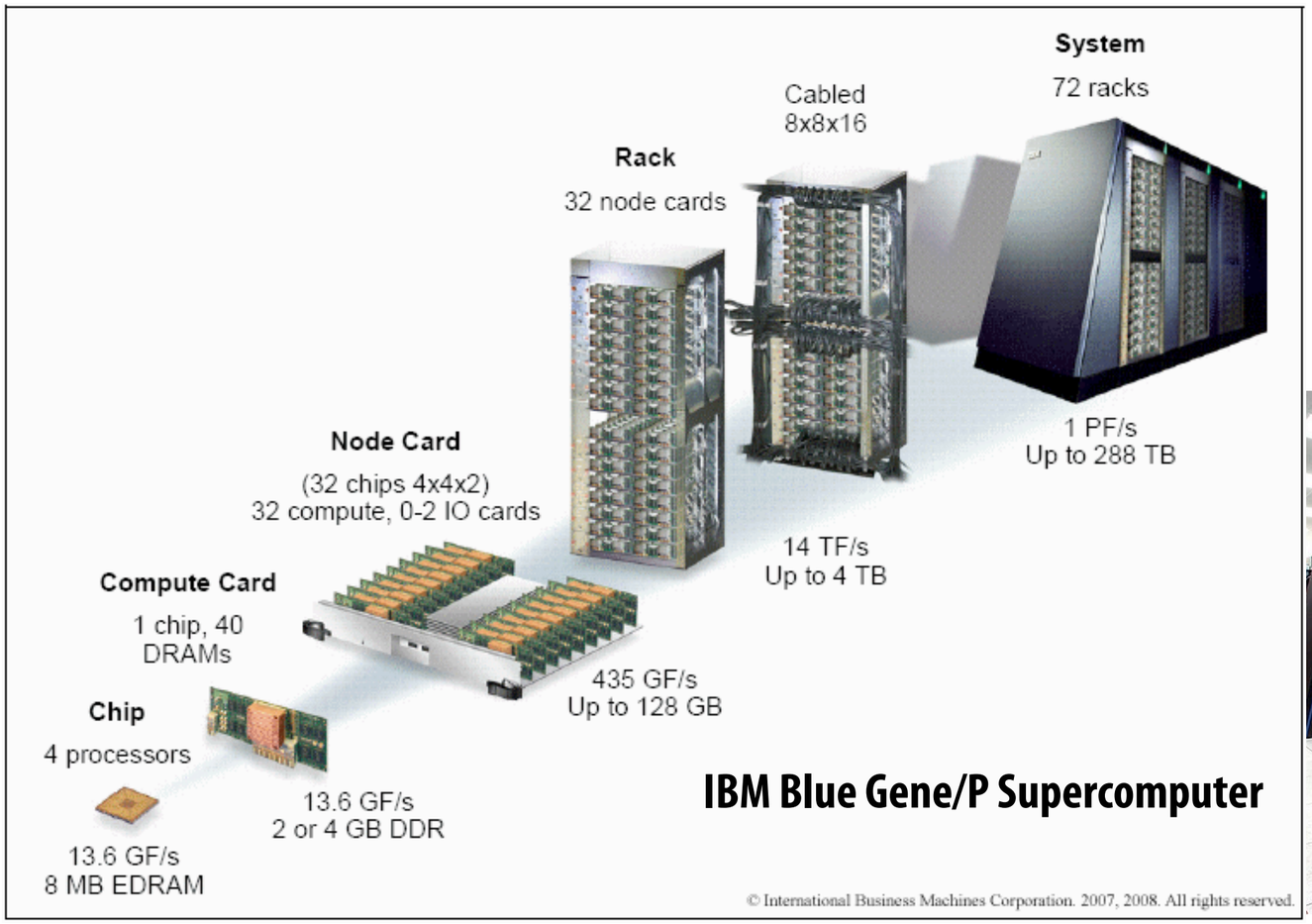
Message passing model (abstraction)

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

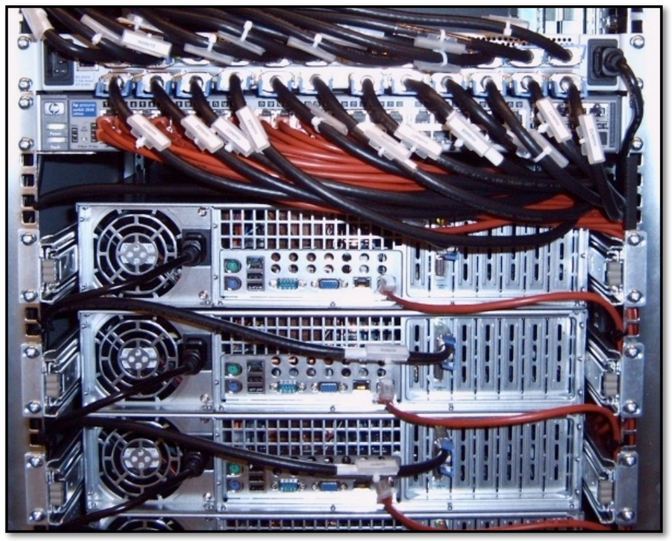


Message passing (implementation)

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



IBM Blue Gene/P Supercomputer



Cluster of workstations (Infiniband network)

Correspondence between models and machines 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 (less efficient SW solution)
  - Mark all pages with shared variables as invalid
  - Page-fault handler issues appropriate network requests

- Reminder: keep in mind what is the programming model (abstraction) and what is the HW implementation

Data-parallel model

- Rigid computation structure
- Historically: same operation on each element of an array
  - Operation ≈ instruction
  - 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)` ⟵ this was an instruction on vectors A, B of length n
  - Matlab 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 during map
  - Synchronization is implicit at the end of the map

Data parallelism example 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`

`foreach` construct is a map

Collection 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 example 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 example 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

(as described, data parallel 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 this way is sometimes referred to as stream programing model

Streams: collections of elements. Elements can be processed independently

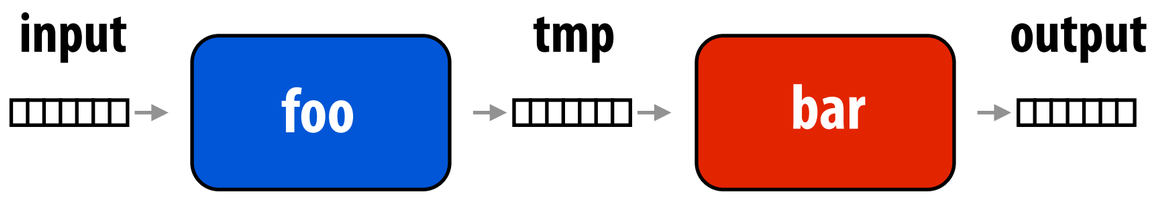
Kernels: side-effect-free functions. Operate element-wise on elements of collections

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

If you’ve ever written OpenGL shader code (e.g., 15-462), you’ve coded in the stream programming system

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:

Allows 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 function feed immediately into second function, never written to memory.

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

Cross fingers and hope compiler generates code intelligently

Kayvon's experience:

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

"I just need one more operator"...

```
// ISPC code:
export void absolute_value(
   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];
   }
}
```

Gather/scatter:
Two key data-parallel communication primitives

```
// 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);
```

Gather: (ISPC equivalent)

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

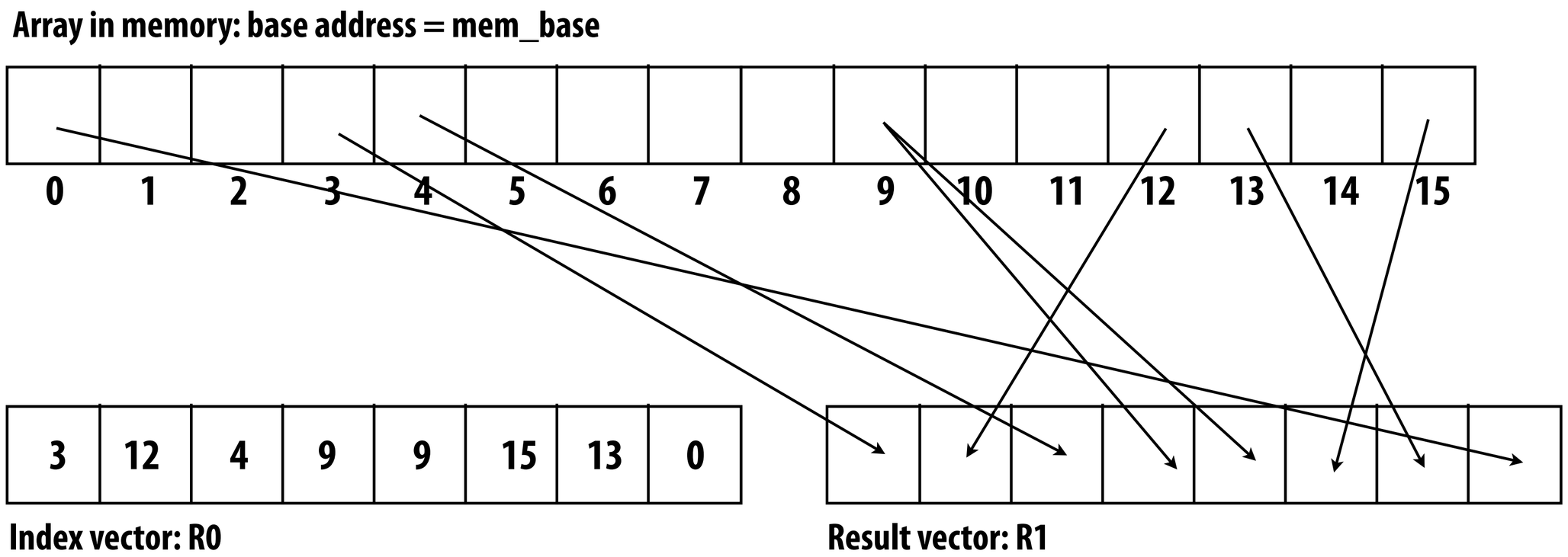
```
// main program:
const int N = 1024;
stream<float> input(N);
stream<int> indices;
stream<float> tmp_output(N);
stream<float> output(N);



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

Scatter: (ISPC equivalent)

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

Gather operation:

```
gather(R1, R0, mem_base);
```



SSE, AVX instruction sets do not directly support SIMD gather/scatter (must implement as scalar loop)

Hardware supported gather/scatter does exist on GPUs. (still an expensive operation)

Data-parallel model summary

- Data-parallelism is about program structure
- In spirit, map a single program onto a large collection of data
  - Functional: side-effect free executions
  - No communication among invocations
- In practice that's how most programs work
- But... most popular 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! (lighten up!)

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 within the map (preserve independent processing)
  - Modern embodiments encourage, don't enforce, this structure

Trend: hybrid programming models

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

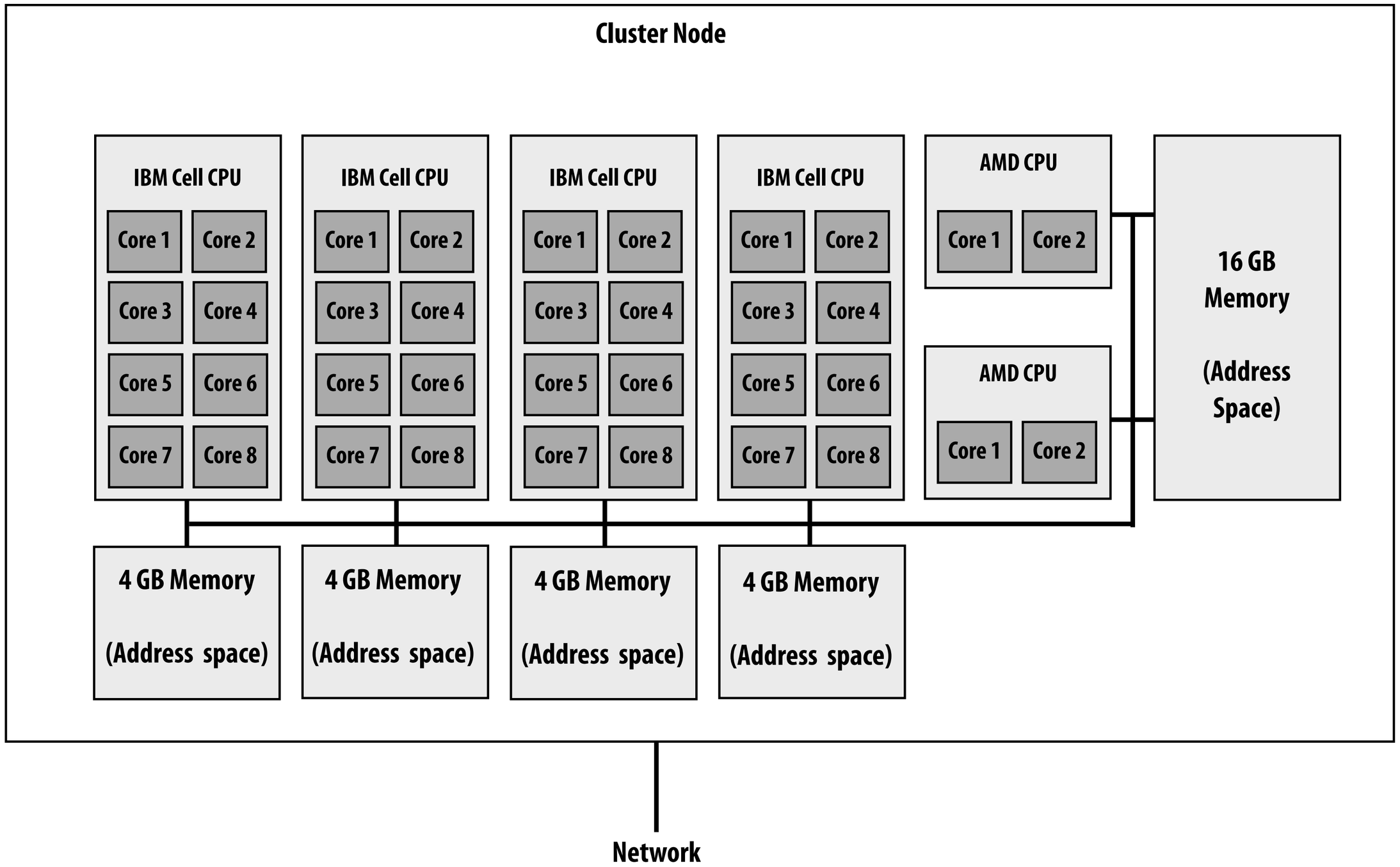
- Data-parallel programming models support synchronization primitives in kernels (CUDA, OpenCL)
  - Permits limited forms of communication

- CUDA/OpenCL use data parallel model to scale to many cores, but adopt shared-address space model for threads in a single core.

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 are abstractions that admit many possible implementations.
- But restrictions imposed my models reflect realities of hardware costs of communication
  - Shared address space machines
  - Messaging passing machines
  - Usually wise to keep abstraction distance low (performance predictability). But want it high enough for 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