Lecture 3:

Parallel Programming Models and their corresponding HW/SW implementations

---

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Tunes

N.W.A.

"Express yourself"

(Straight Outta Compton)

"… and choose the appropriate programming model to do it."

- Ice Cube

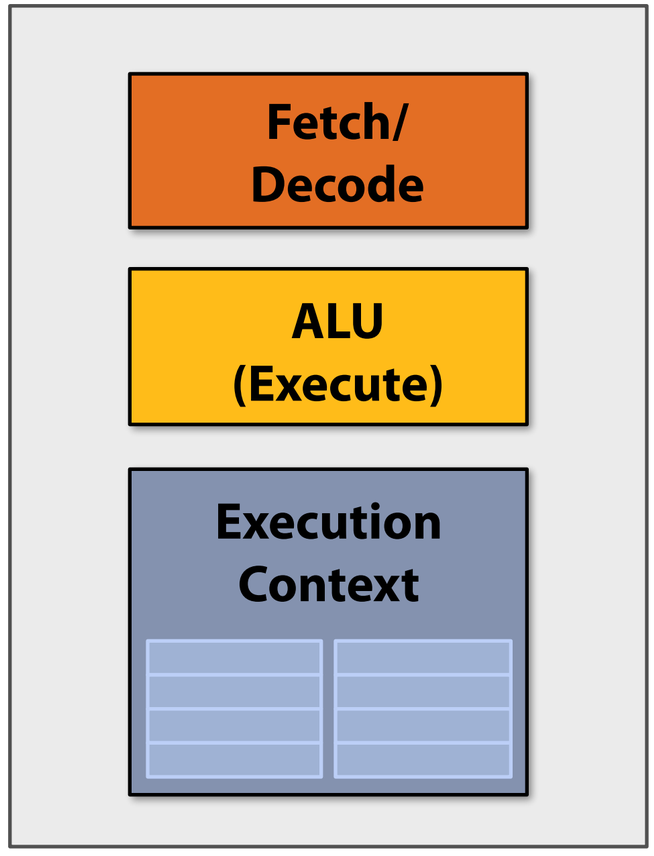
Review from last class:

Running code on a simple processor

My very simple program: compute $\sin(x)$ using Taylor expansion

```
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;
   }
}
```

My very simple processor: completes one instruction per clock



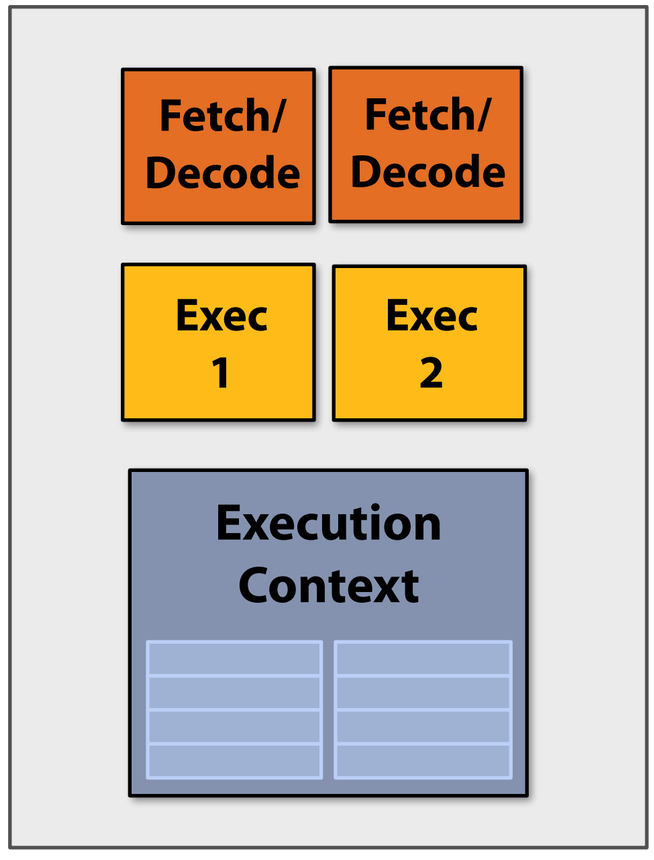
Review: superscalar execution

Unmodified program:

```
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;
   }
}
```

Independent operations in instruction stream (can be executed in parallel by processor on execution units 1 and 2)

My single core, superscalar processor: executes up to two instructions per clock from a single instruction stream.



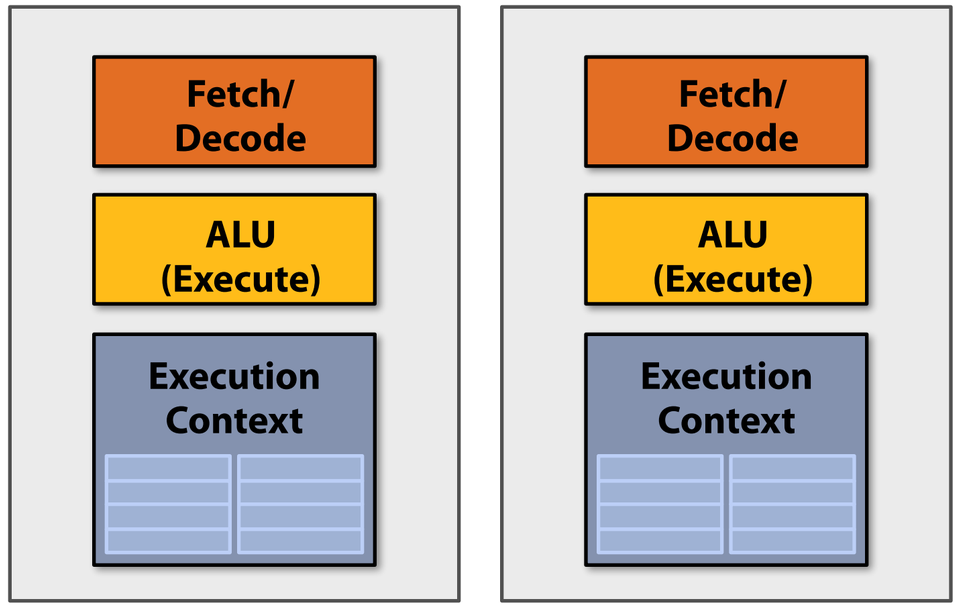
Review: multi-core execution (two cores)

Modify program to create two threads of control (two instruction streams)

```
typedef struct {
   int N;
   int terms;
   float* x;
   float* result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result)
{
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg)
{
   my_args* thread_args = (my_args*)thread_arg;
   sinx(args->N, args->terms, args->x, args->result); // do work
}
```

My dual-core processor:

executes one instruction per clock from an instruction stream on each core.

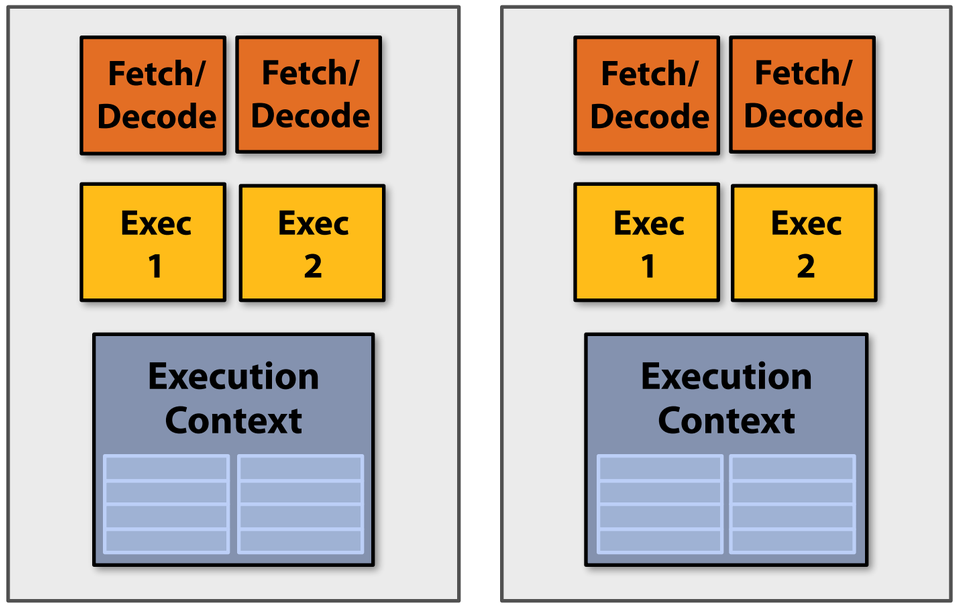


Review: multi-core + superscalar execution

Modify program to create two threads of control (two instruction streams)

```
typedef struct {
   int N;
   int terms;
   float* x;
   float* result;
} my_args;

void parallel_sinx(int N, int terms, float* x, float* result)
{
    pthread_t thread_id;
    my_args args;

    args.N = N/2;
    args.terms = terms;
    args.x = x;
    args.result = result;

    pthread_create(&thread_id, NULL, my_thread_start, &args); // launch thread
    sinx(N - args.N, terms, x + args.N, result + args.N); // do work
    pthread_join(thread_id, NULL);
}

void my_thread_start(void* thread_arg)
{
   my_args* thread_args = (my_args*)thread_arg;
   sinx(args->N, args->terms, args->x, args->result); // do work
}
```

My superscalar dual-core processor: executes up to two instructions per clock from an instruction stream on each core.



Review: multi-core (four cores)

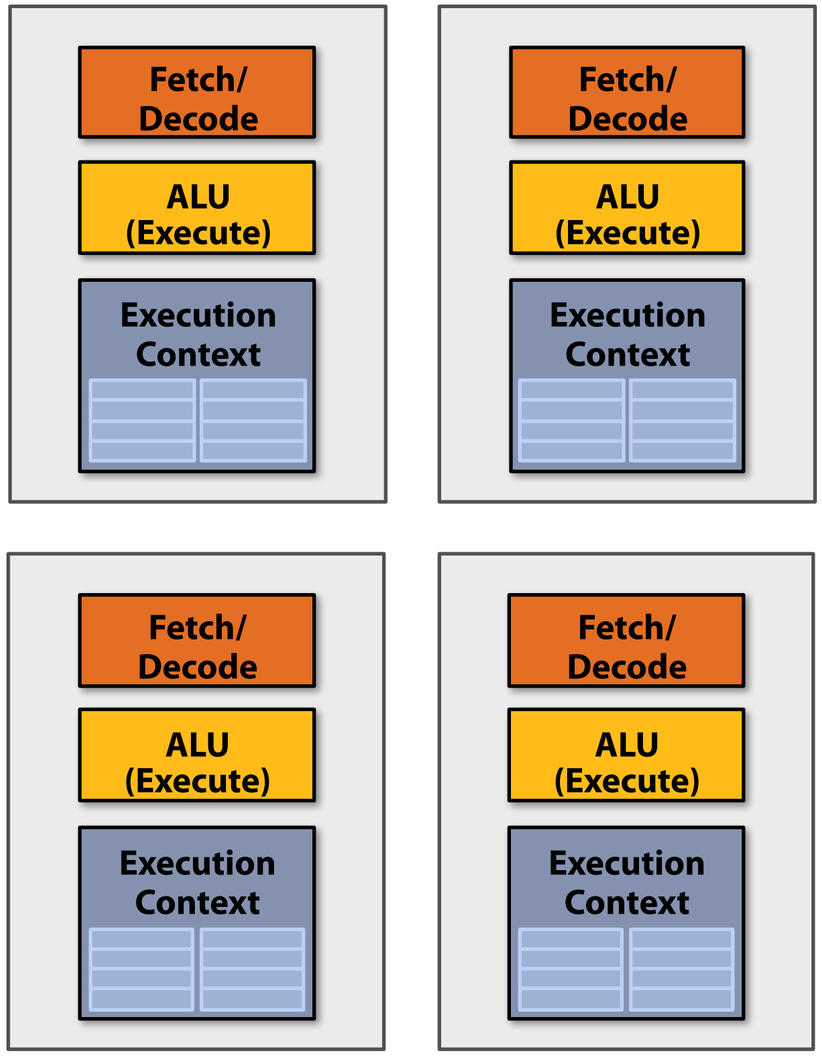
Modify program to create many threads of control: recall Kayvon's fictitious language

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      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;
   }
}
```

My quad-core processor:

executes one instruction per clock

from an instruction stream on <u>each</u> core.



Review: four, 8-wide SIMD cores

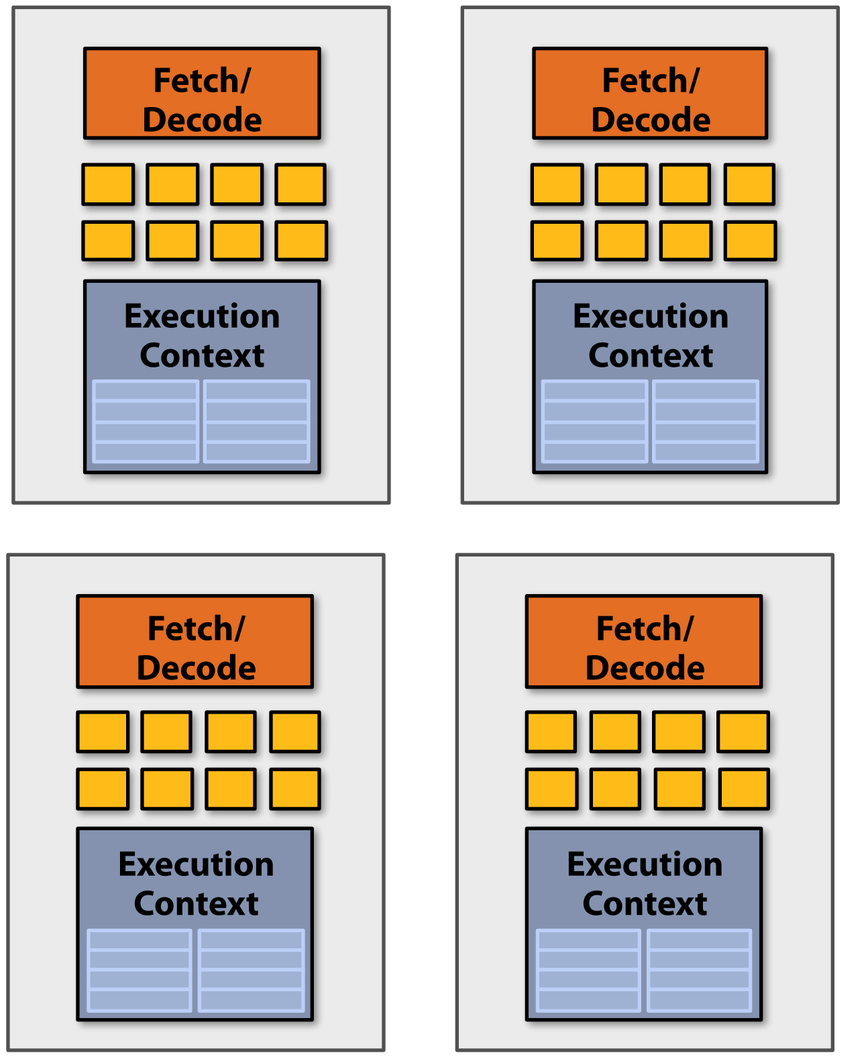
Observation: must perform many iterations of the same loop body.

Optimization: share instruction stream across execution of multiple iterations (single instruction multiple data = SIMD)

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      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;
   }
}
```

My SIMD quad-core processor:

executes one 8-wide SIMD instruction per clock from an instruction stream on each core.



Review: four SIMD, multi-threaded cores

Observation: memory operations have very long latency

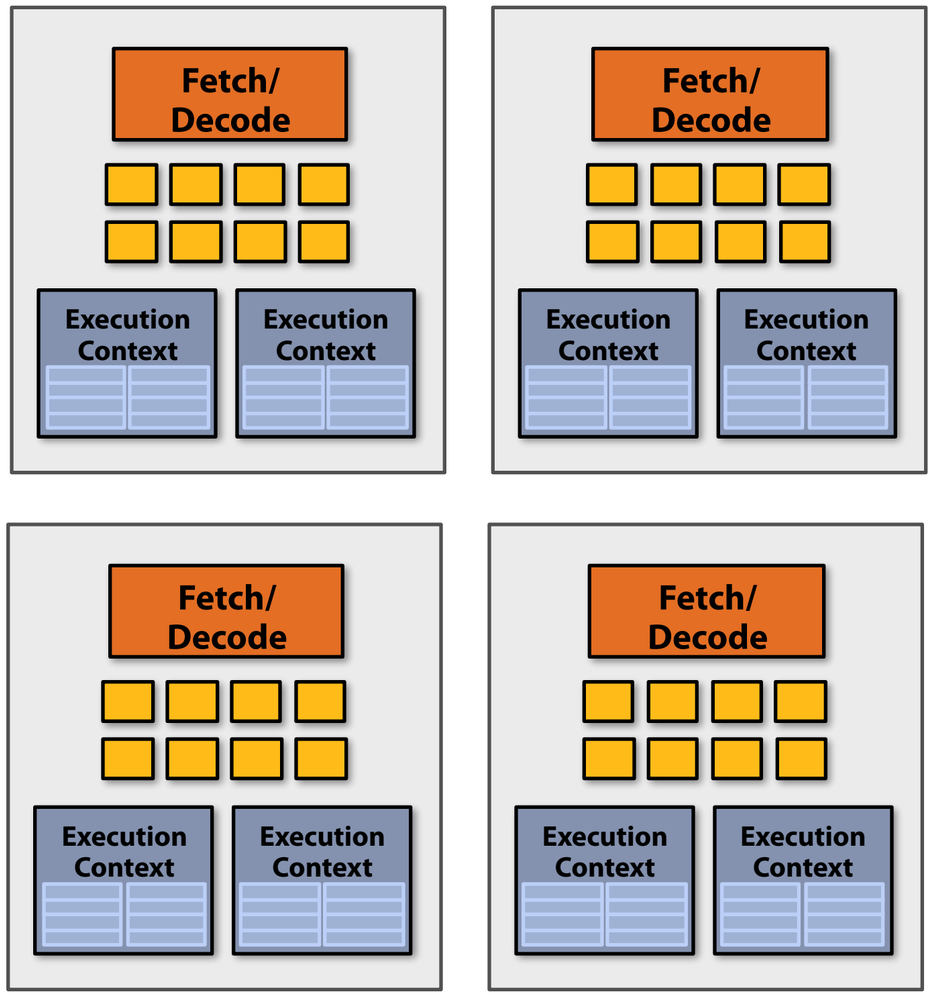
Solution: hide latency of loading data for one iteration by executing arithmetic instructions from other iterations

```
void sinx(int N, int terms, float* x, float* result)
{
   // declare independent loop iterations
   forall (int i from 0 to N-1)
   {
      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;
   }
}
```

Memory load

Memory store

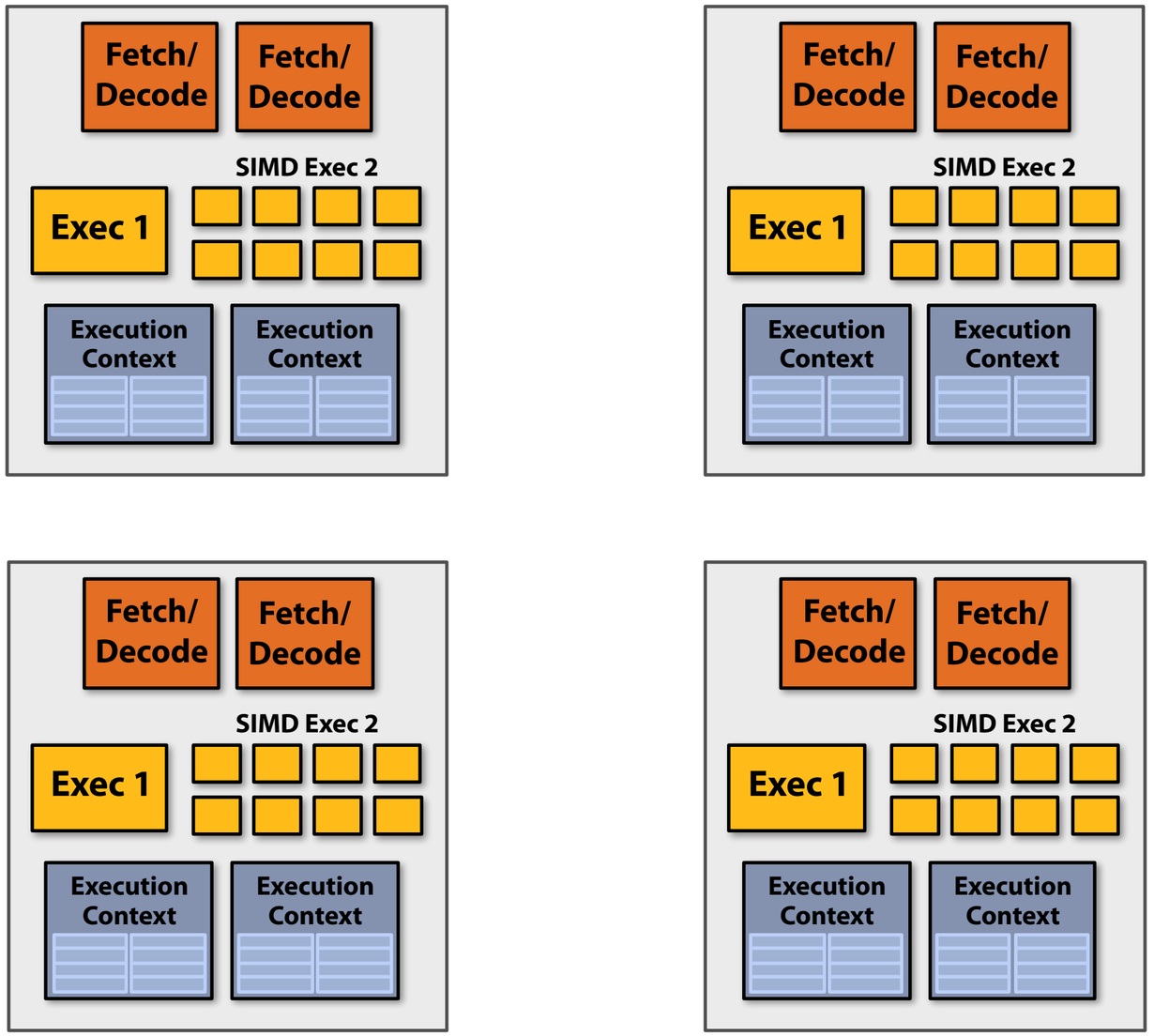
My multi-threaded, SIMD quad-core processor: executes one SIMD instruction per clock from one instruction stream on each core. But can switch to other instruction stream when faced with stall.



Review: four superscalar, SIMD, multi-threaded cores

My multi-threaded, superscalar, SIMD quad-core processor:

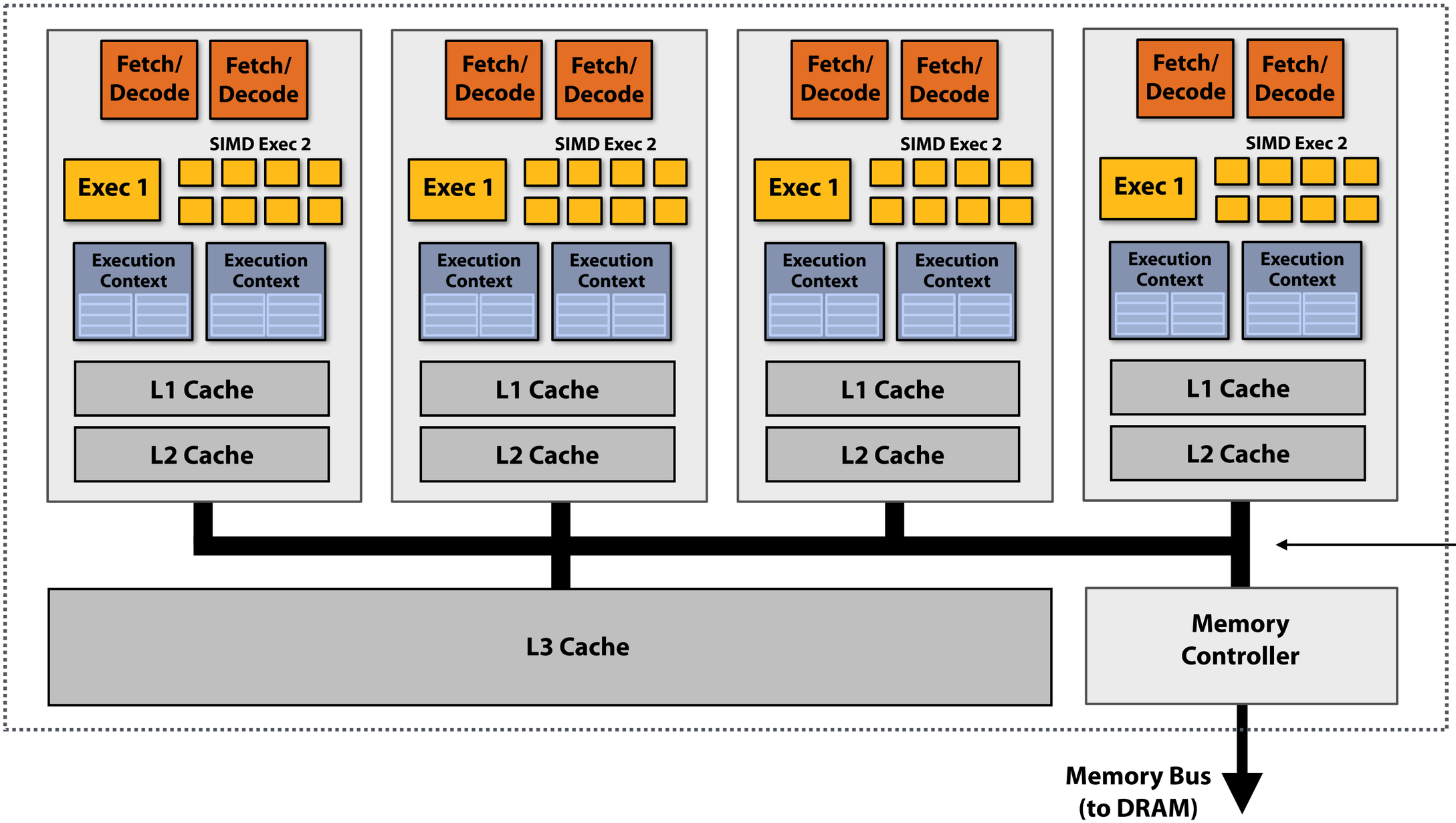
executes up to two instructions per clock from one instruction stream on each core (one SIMD instruction + one scalar instruction). But can switch to other instruction stream when faced with stall.



Connecting it all together

Kayvon's simple quad-core processor.

Four cores, two-way multi-threading per core (max eight threads active on chip at once), up to two instructions per clock per core (one of those instructions is 8-wide SIMD)

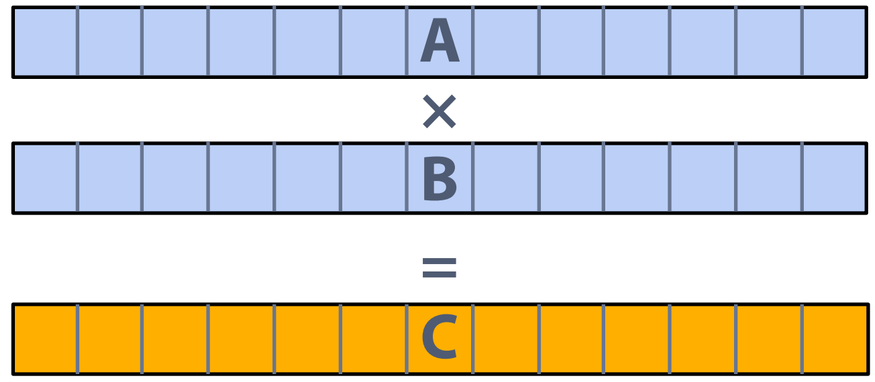


Review: thought experiment

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

Assume vectors contain millions of elements

- **Load input A[i]**
- **Load input B[i]**
- **Compute A[i] × B[i]**
- **Store result into C[i]**



Three memory operations (12 bytes) for every MUL

NVIDIA GTX 480 GPU can do 480 MULs per clock (@ 1.2 GHz)

Need ~6.4 TB/sec of bandwidth to keep functional units busy (only have 177 GB/sec)

~ 3% efficiency… but 7x faster than quad-core CPU!

(2.6 GHz Core i7 Gen 4 quad-core CPU connected to 25 GB/sec memory bus will exhibit 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:

- Organize computation to fetch data from memory less often
  - Reuse data while processing instruction stream for a single thread (traditional temporal/spatial locality optimizations)
  - Share data across threads (inter-thread cooperation)

- Request data less often (instead, do more arithmetic: it's "free")
  - Term: "arithmetic intensity" — ratio of math operations to data access operations in an instruction stream
  - Main point: programs must have high arithmetic intensity to utilize modern processors efficiently

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 concurrently

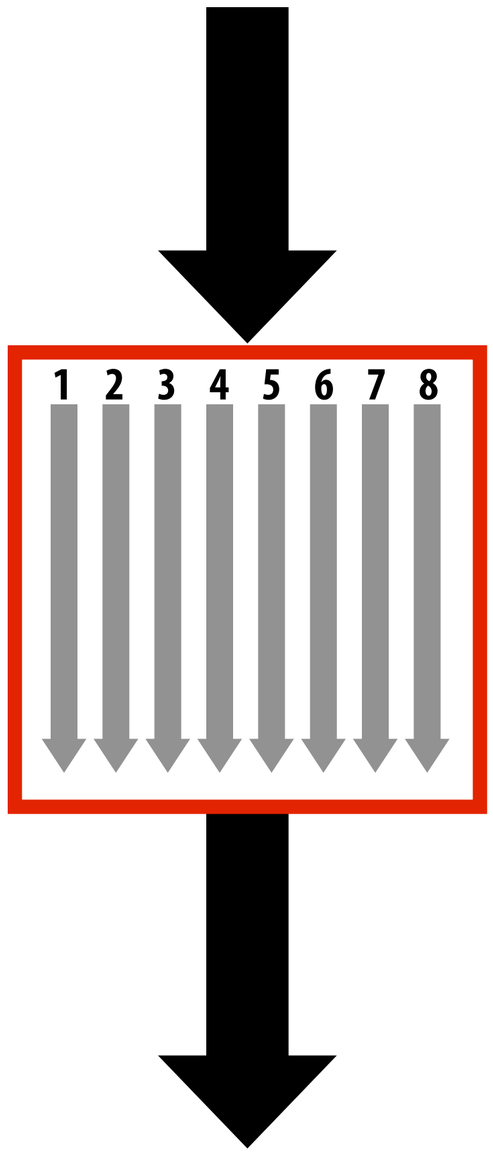
Upon return, all instances have completed

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 concurrently

Upon return, all instances have completed

In this illustration `programCount` = 8

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;
    }
}
```

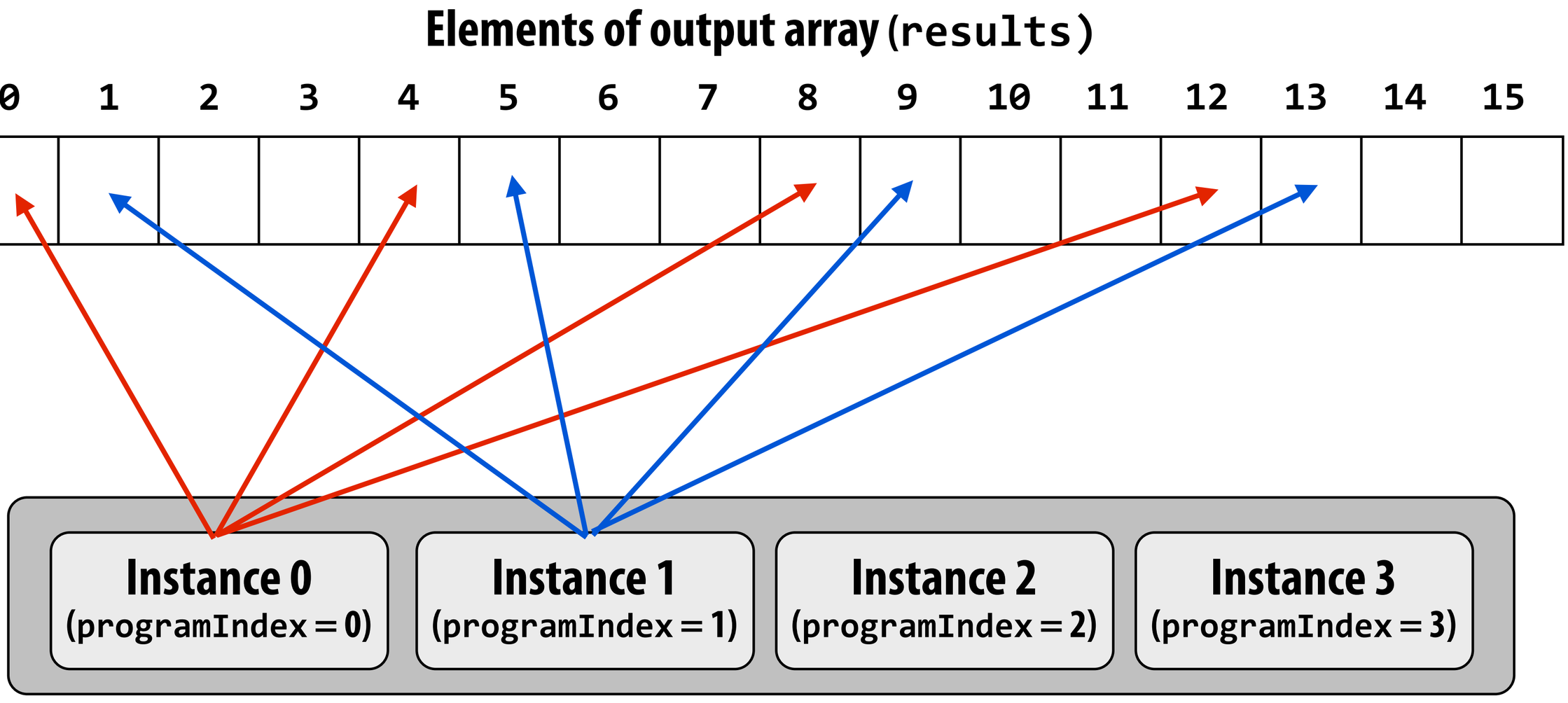
ISPC Keywords:

`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 program instances to loop iterations



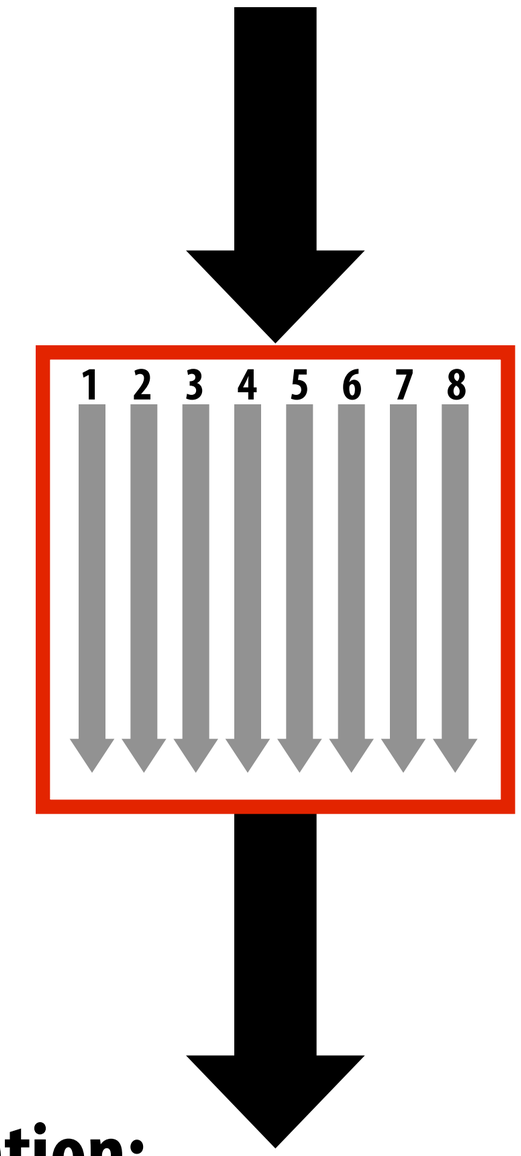
"Gang" of ISPC program instances

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

ISPC implements the gang abstraction using SIMD instructions

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

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

// initialize x here

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



Sequential execution (C code)

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

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

Sequential execution (C code)

SPMD programming abstraction:

Call to ISPC function spawns “gang” of ISPC “program instances”
All instances run ISPC code concurrently
Upon return, all instances have completed

ISPC compiler generates SIMD implementation:

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

sin(x) in ISPC: version 2

Blocked assignment of elements to instances

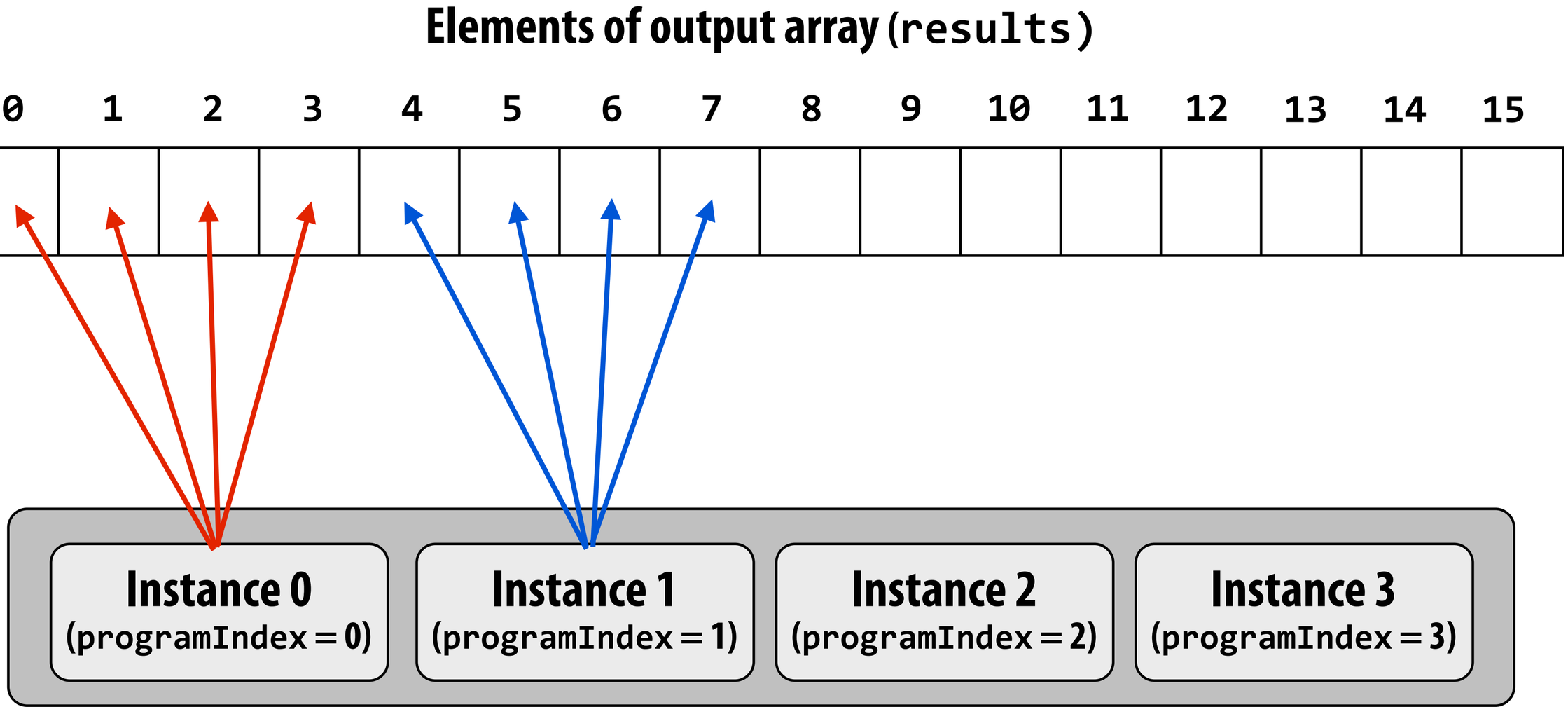
C++ code: `main.cpp`

```
#include “sinx_ispc.h”

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

// initialize x here

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

ISPC code: `sinx.ispc`

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

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

Blocked assignment of program instances to loop iterations



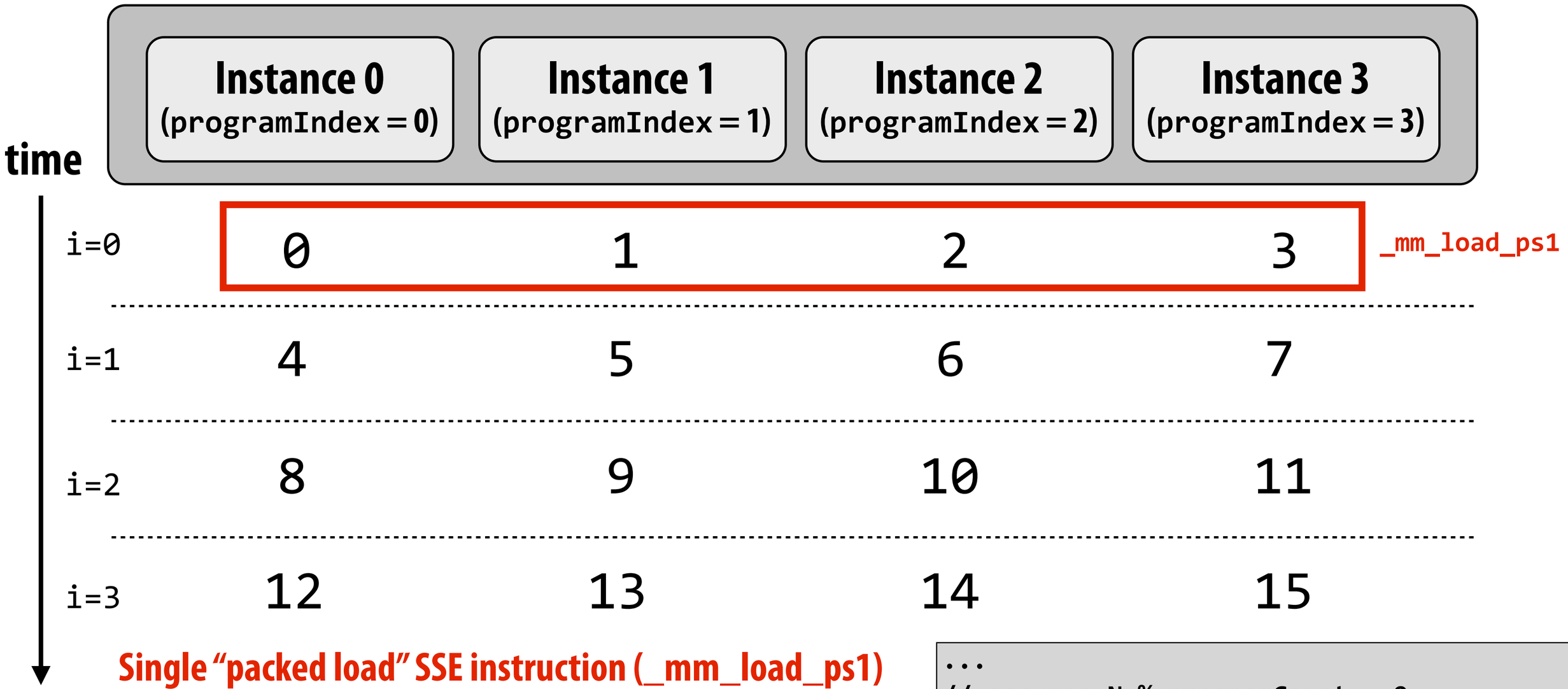
"Gang" of ISPC program instances

In this illustration: 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];
```

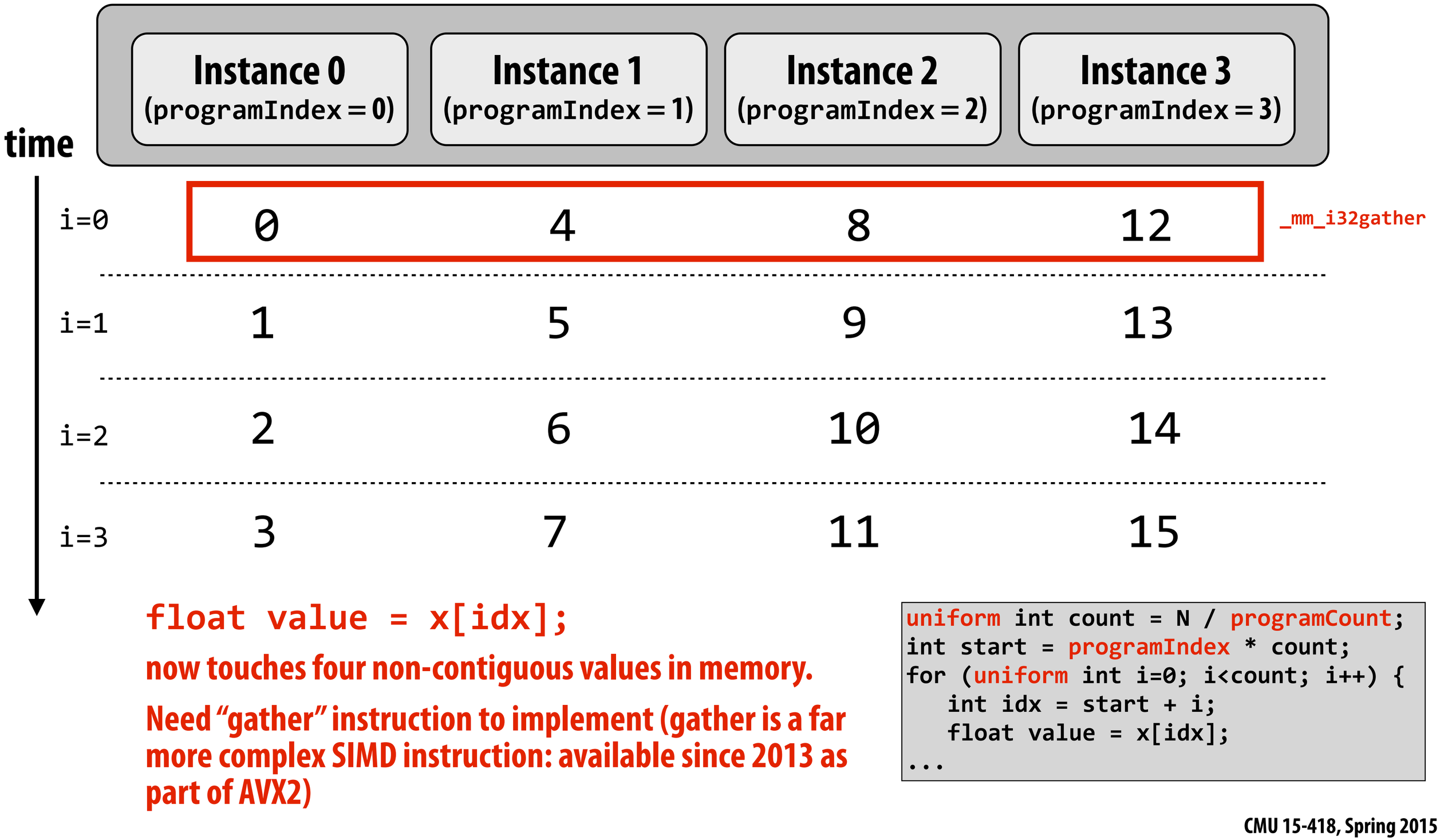
for all program instances, since the four values are contiguous in memory

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

Schedule: blocked 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 since 2013 as part of AVX2)

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

Raising level of abstraction with foreach

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

C++ code: `main.cpp`

```
#include “sinx_ispc.h”

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

// initialize x here

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

ISPC code: `sinx.ispc`

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

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

   return sum;
}
```

Correct ISPC solution

`sum` is of type `uniform float` (one copy of variable for all program instances)
`x[i]` is not a uniform expression (different value for each program instance)
Result: compile-time type error

ISPC discussion: sum "reduction"

Compute the sum of all array elements in parallel

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 for all instances (uniform)

The 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 = reduce_add(partial);

   return sum;
}
```

```
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 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 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" (provides way of thinking about the structure of programs)

Language or library primitives/mechanisms

Compiler and/or parallel runtime

OS system call API

Operating system

Hardware Architecture (HW/SW boundary)

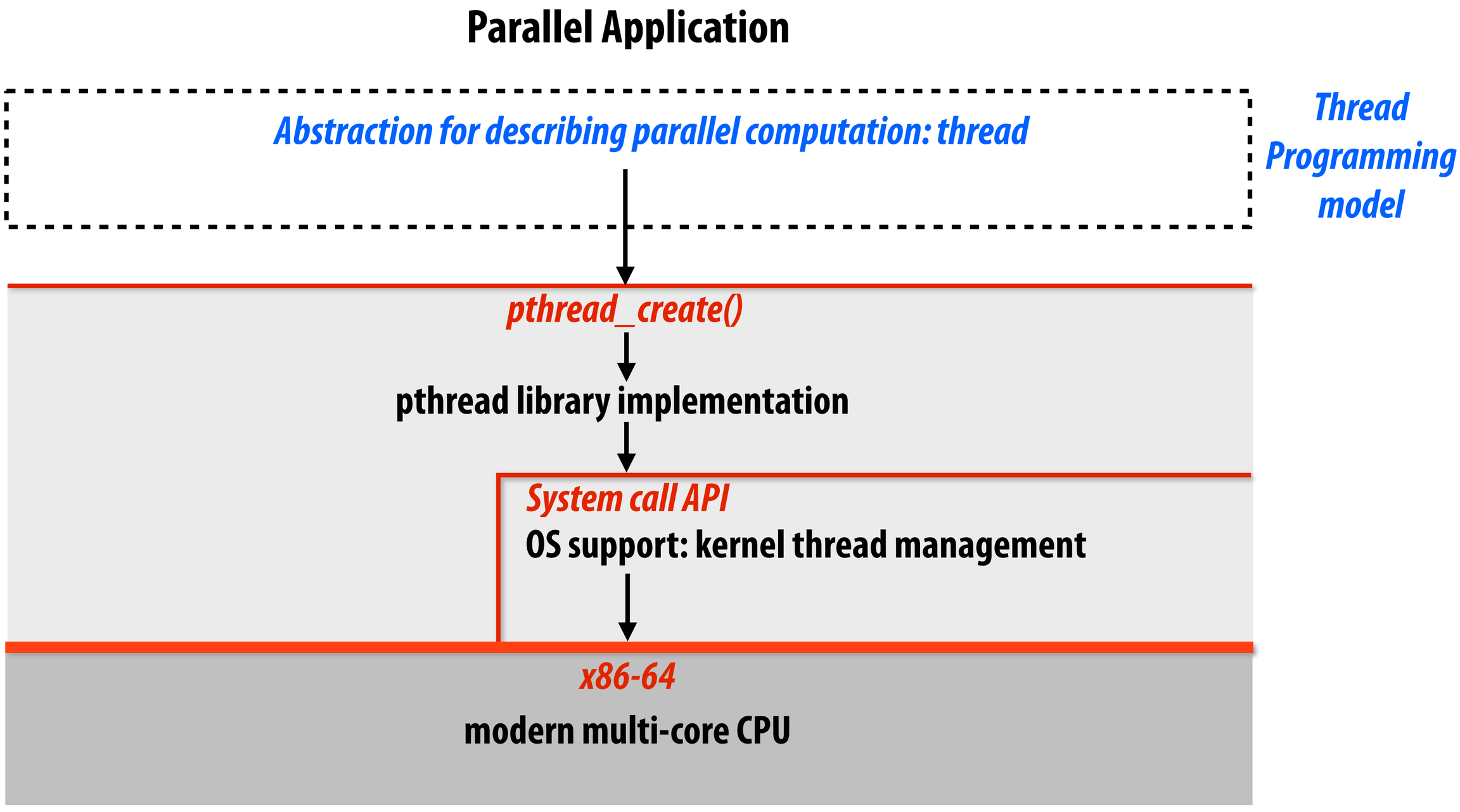
Micro-architecture (hardware implementation)

Blue italic text: abstraction/concept

Red italic text: system interface

Black text: system implementation

Example: expressing parallelism with pthreads

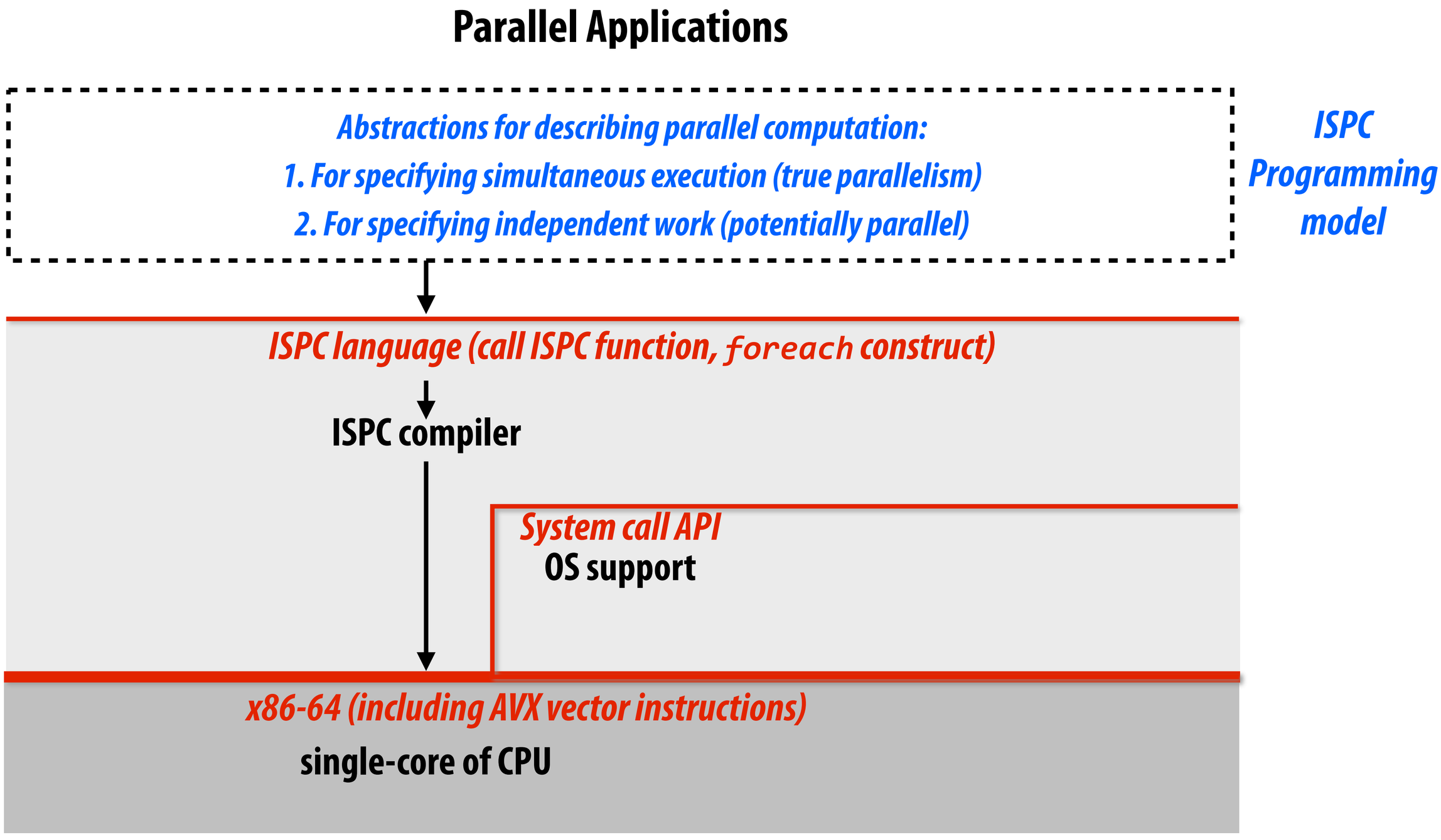


Blue italic text: abstraction/concept

Red italic text: system interface

Black text: system implementation

Example: expressing parallelism with 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

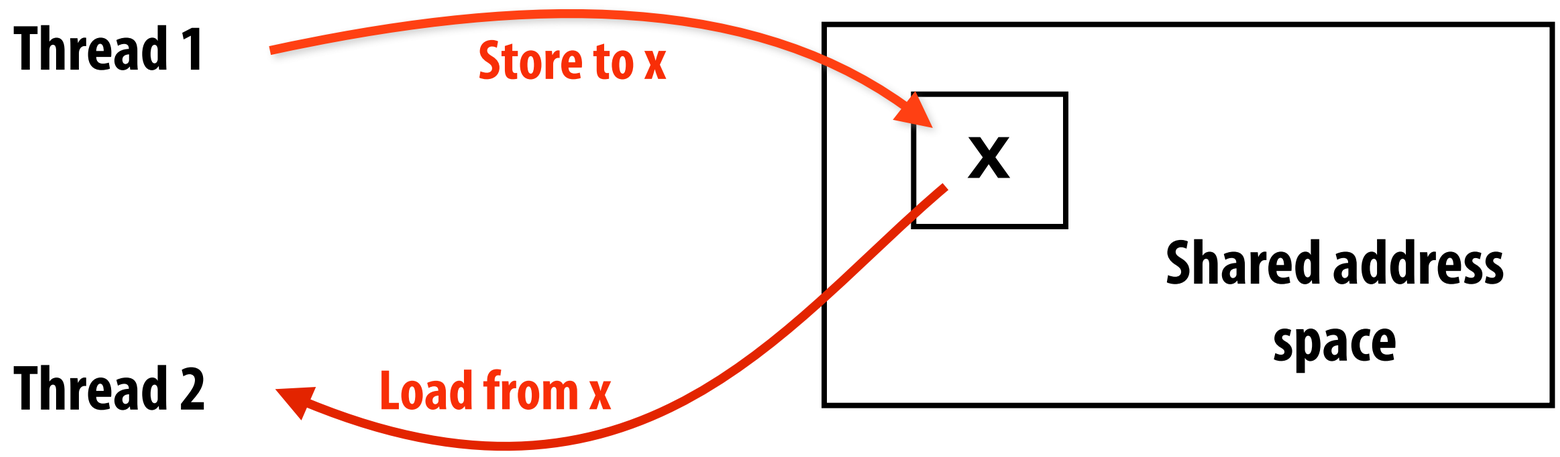
3. Data parallel

Shared address space model of communication

Shared address space model (abstraction)

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

(Communication operations shown in red)



Thread 1:

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

Thread 2:

```
void foo(int* x)
{
  while (x == 0) {}
  print x;
}
```

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

Shared address space model (abstraction)

- Synchronization primitives are also shared variables
  - Example: locks

Thread 1:

```
int x = 0;
Lock my_lock;

spawn_thread(foo, &x, &my_lock);


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

Thread 2:

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

  print x;
}
```

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

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 of value by thread 1)
  - Manipulating synchronization primitives
    - e.g., mutual exclusion using locks

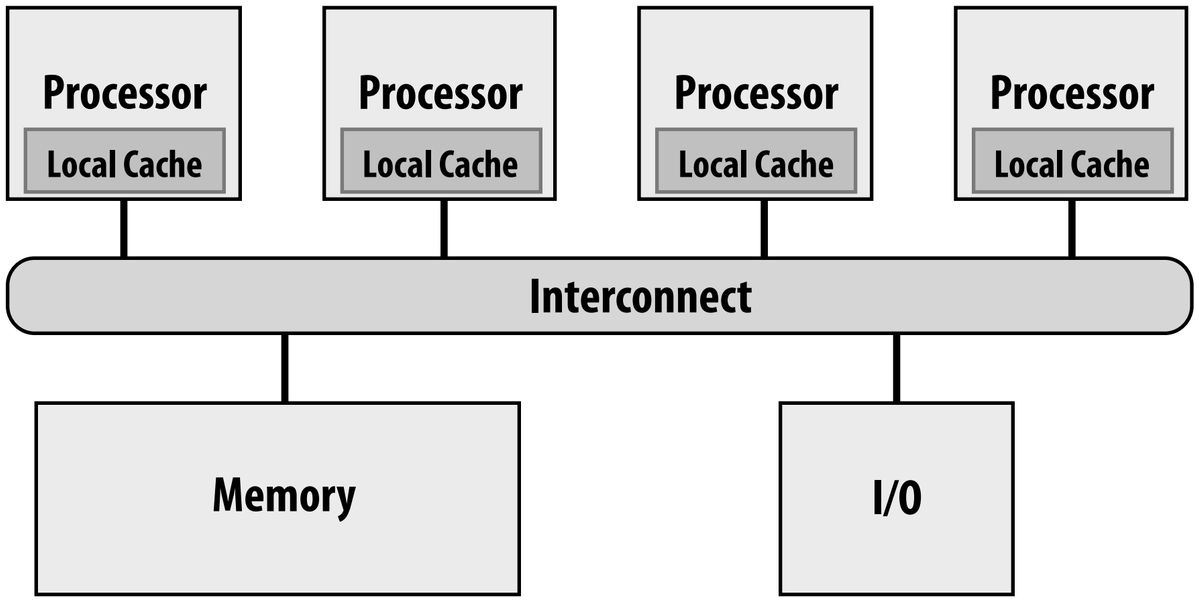
- Natural extension of sequential programming model
  - In fact, all our discussions have assumed a shared address space so far

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

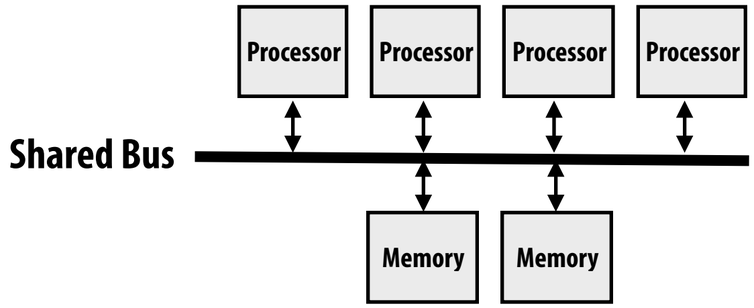
Shared address space HW implementation

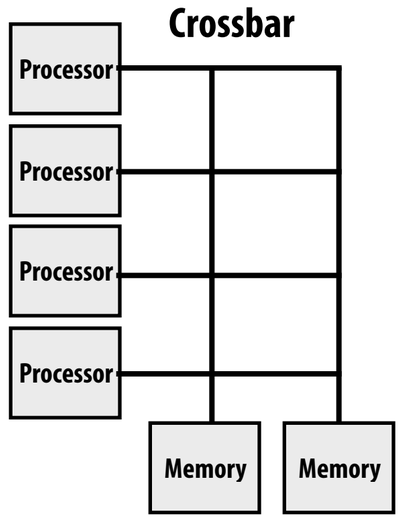
Any processor can directly reference any memory location

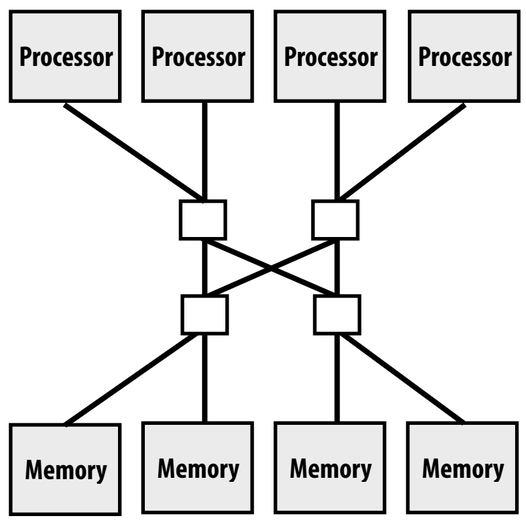
"Dance-hall" organization



Interconnect examples







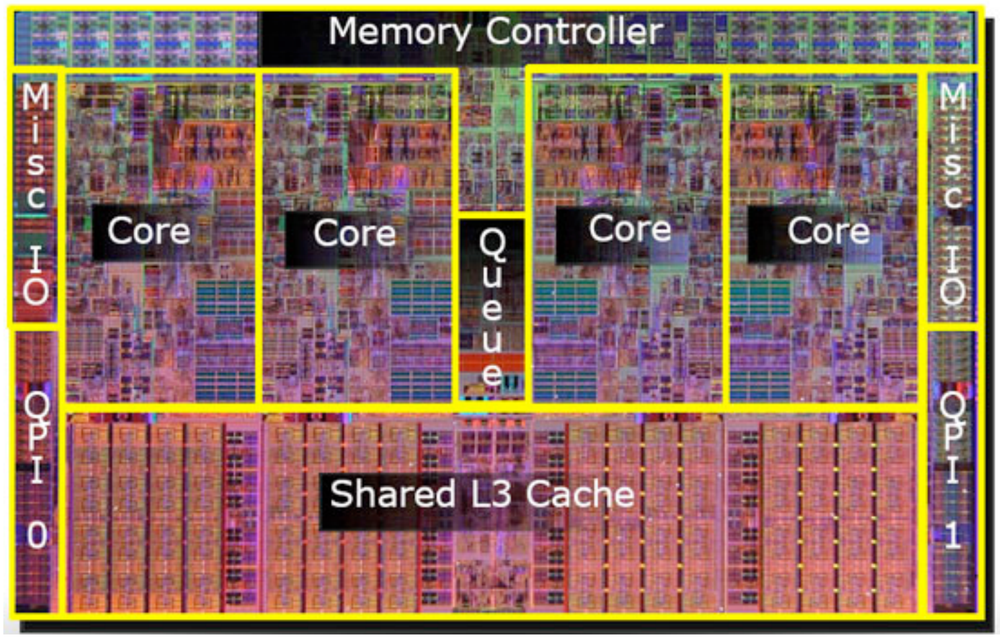
Multi-stage network

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

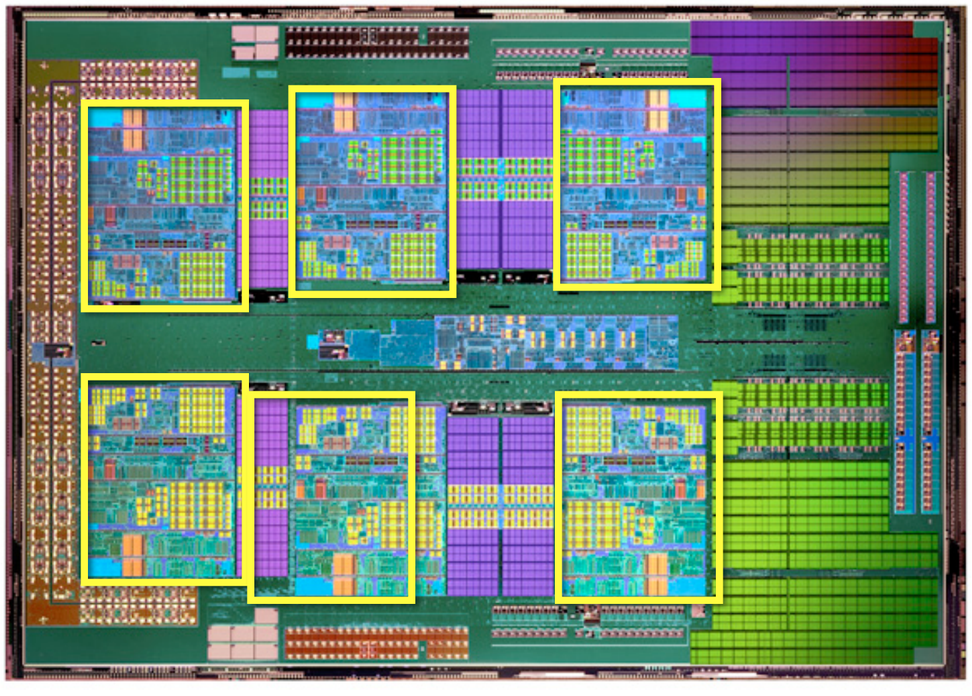
    (* caching introduces non-uniform access times, but we'll talk about that later)

Shared address space architectures

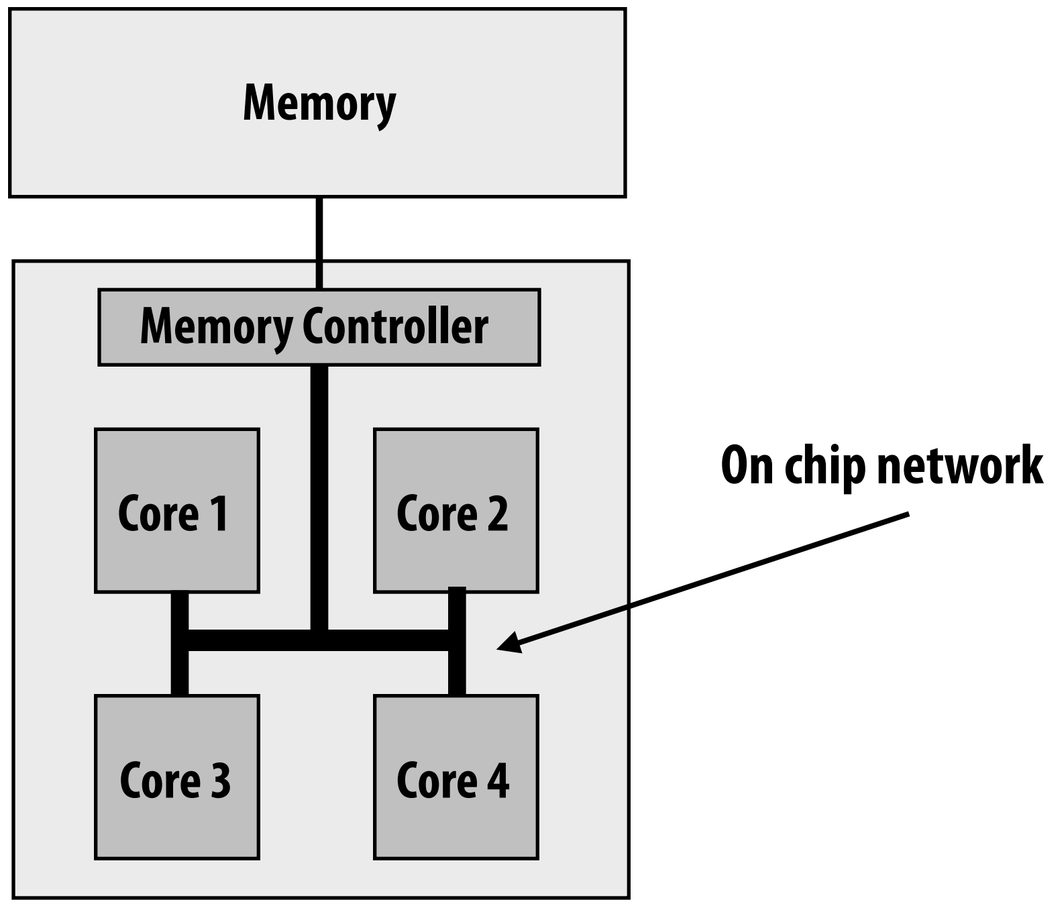
Commodity x86 examples



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

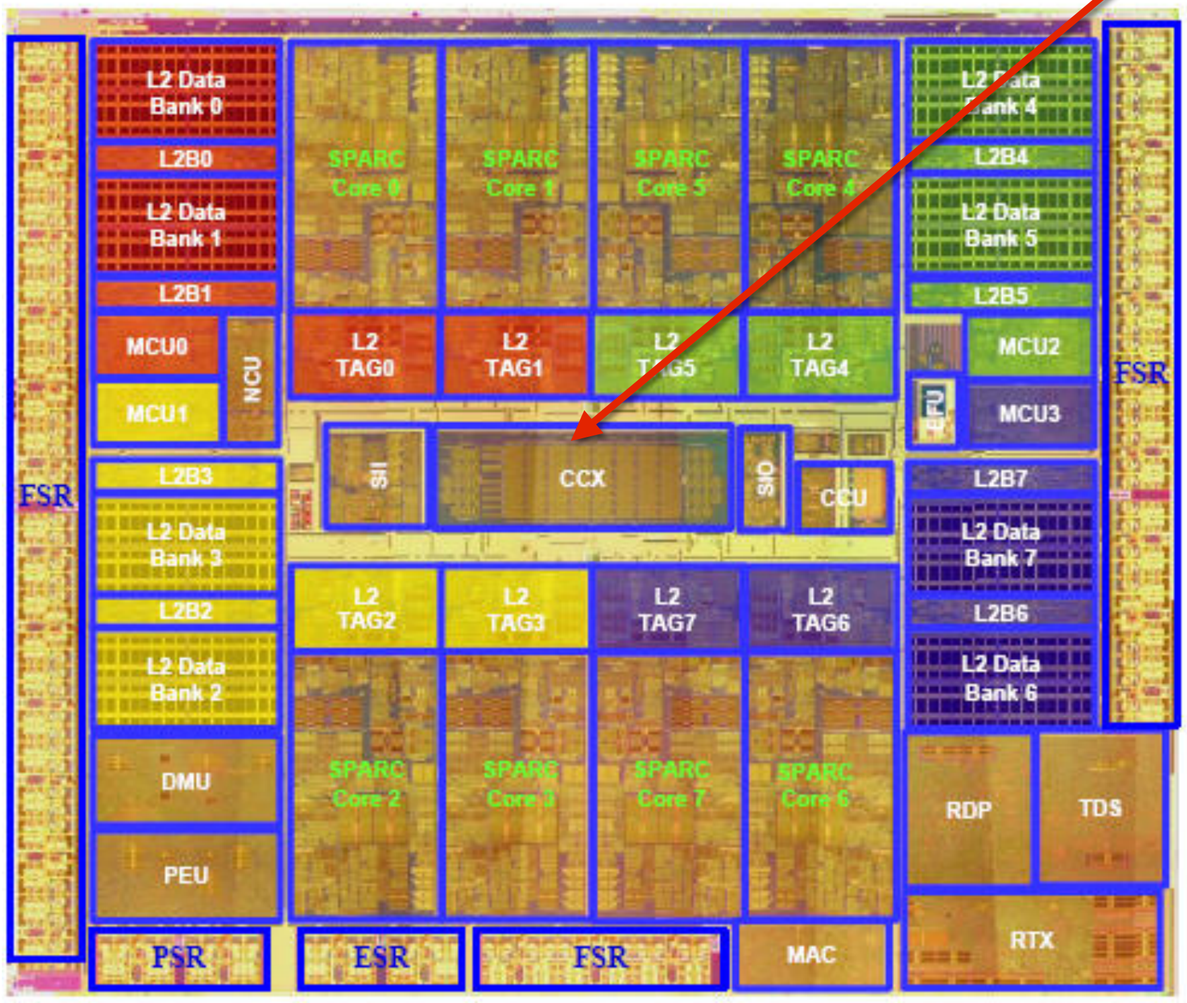


AMD Phenom II (six core)

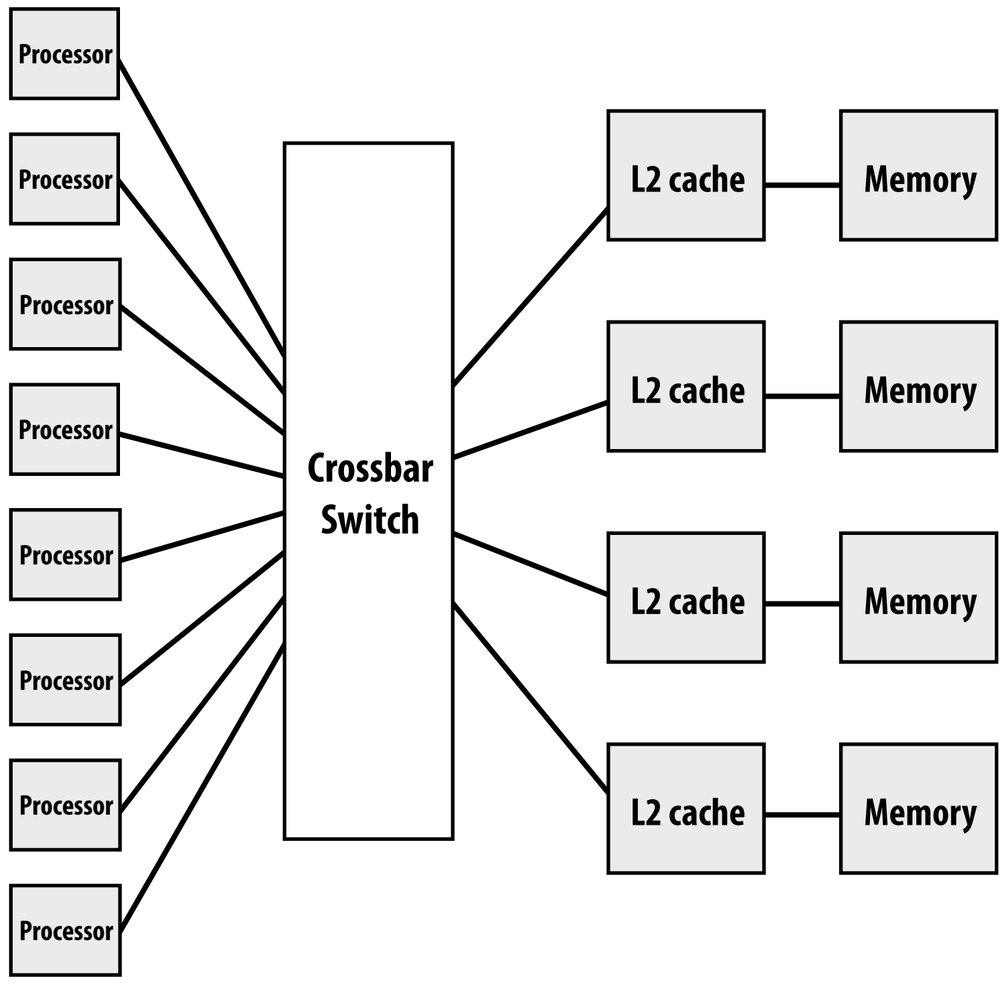


SUN Niagara 2 (UltraSPARC T2)

Note area of crossbar: about die area of one core

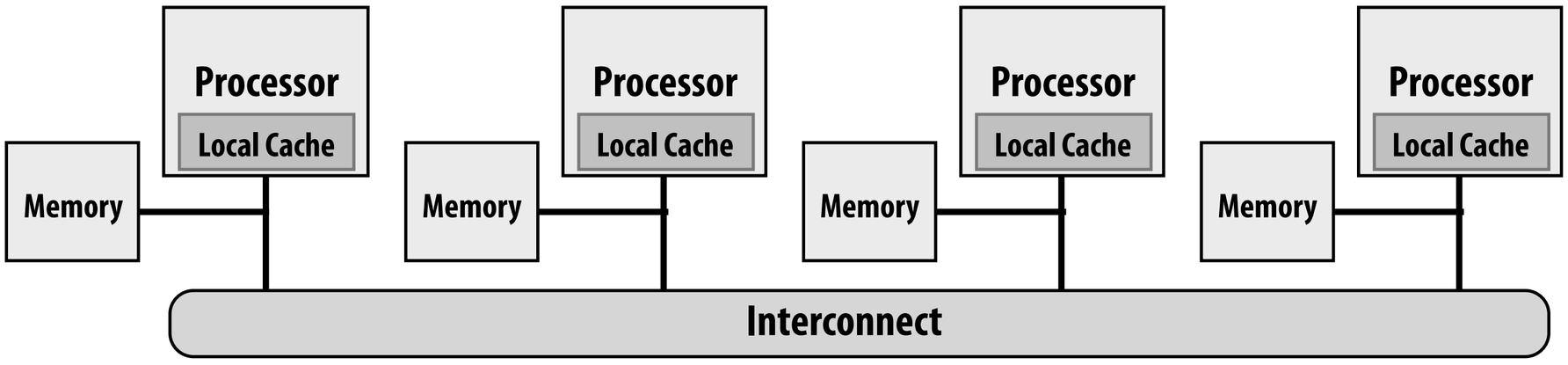


Eight cores



Non-uniform memory access (NUMA)

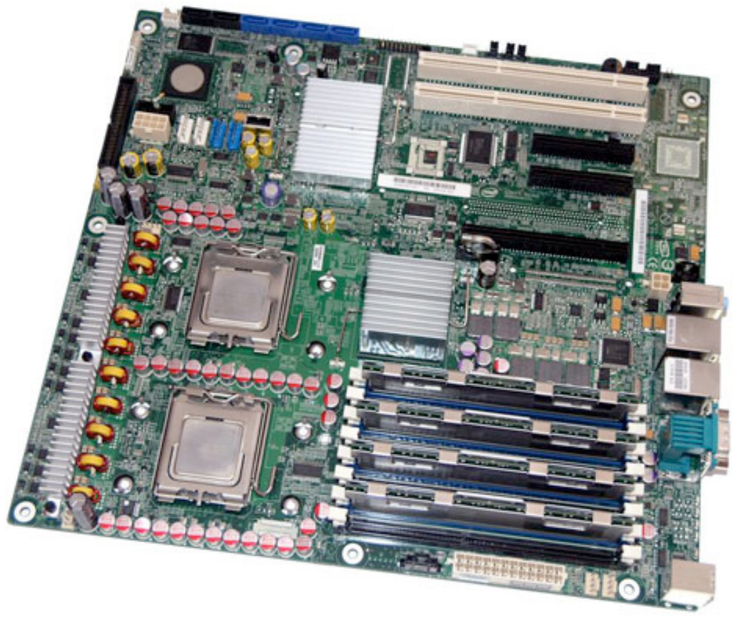
All processors can access any memory location, but... cost of memory access (latency and/or bandwidth) is different for different processors



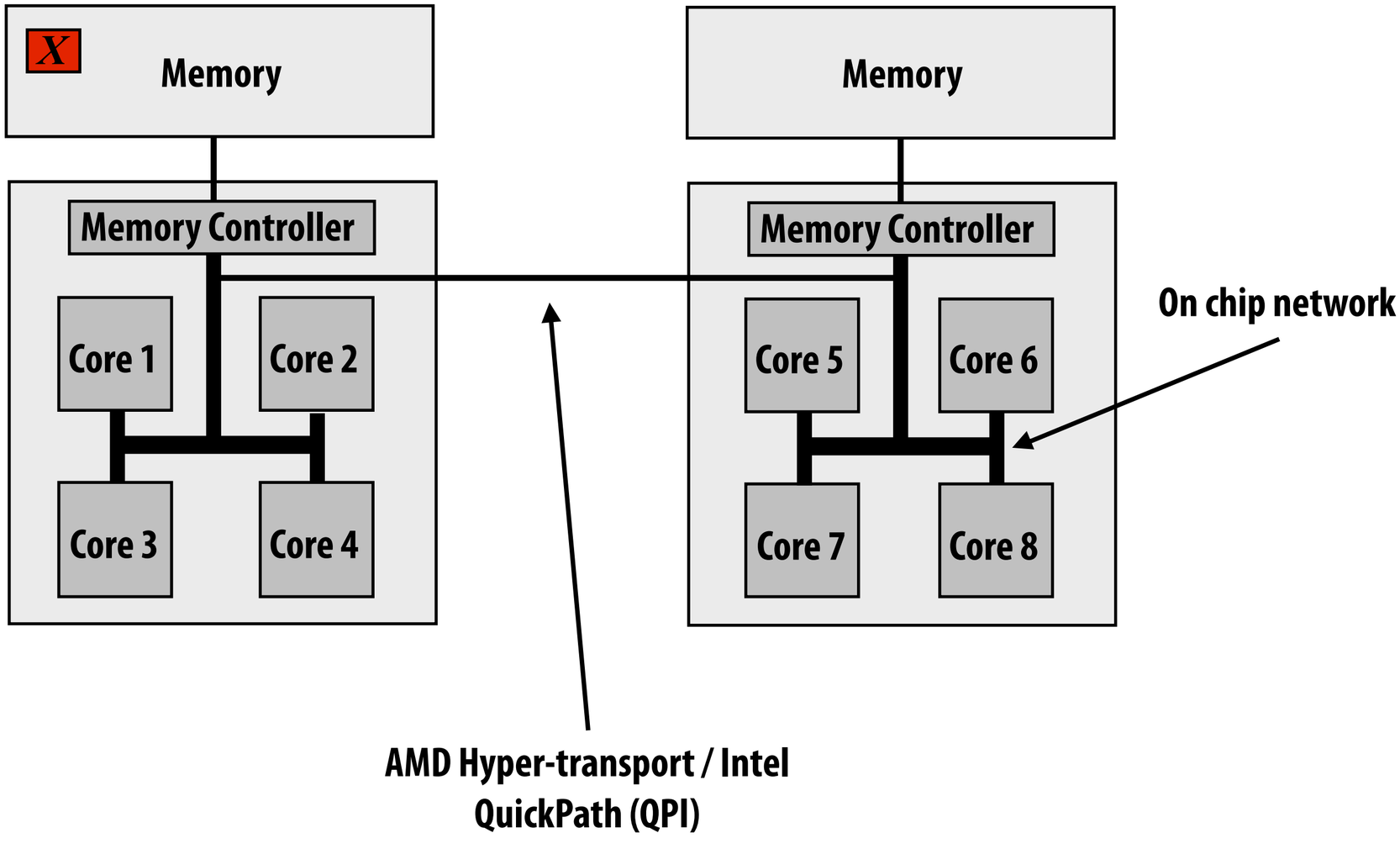
- Problem with preserving uniform access time in a system: scalability
  - GOOD: costs are uniform, BAD: but memory is uniformly far away
- NUMA designs are more scalable
  - High bandwidth to local memory; BW scales with number of nodes if most accesses are local
  - Low latency access to local memory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

Non-uniform memory access (NUMA)

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

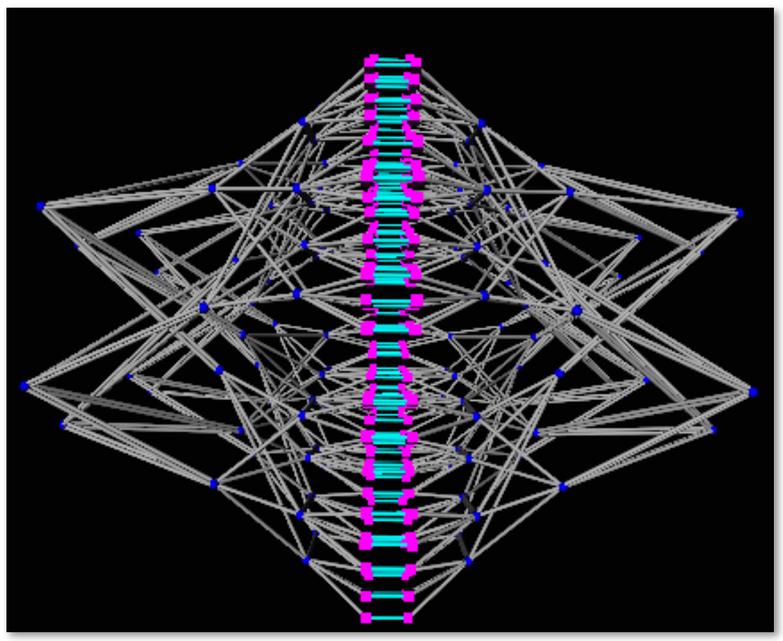


Example: modern dual-socket configuration



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 topology

Image credit: Pittsburgh Supercomputing Center

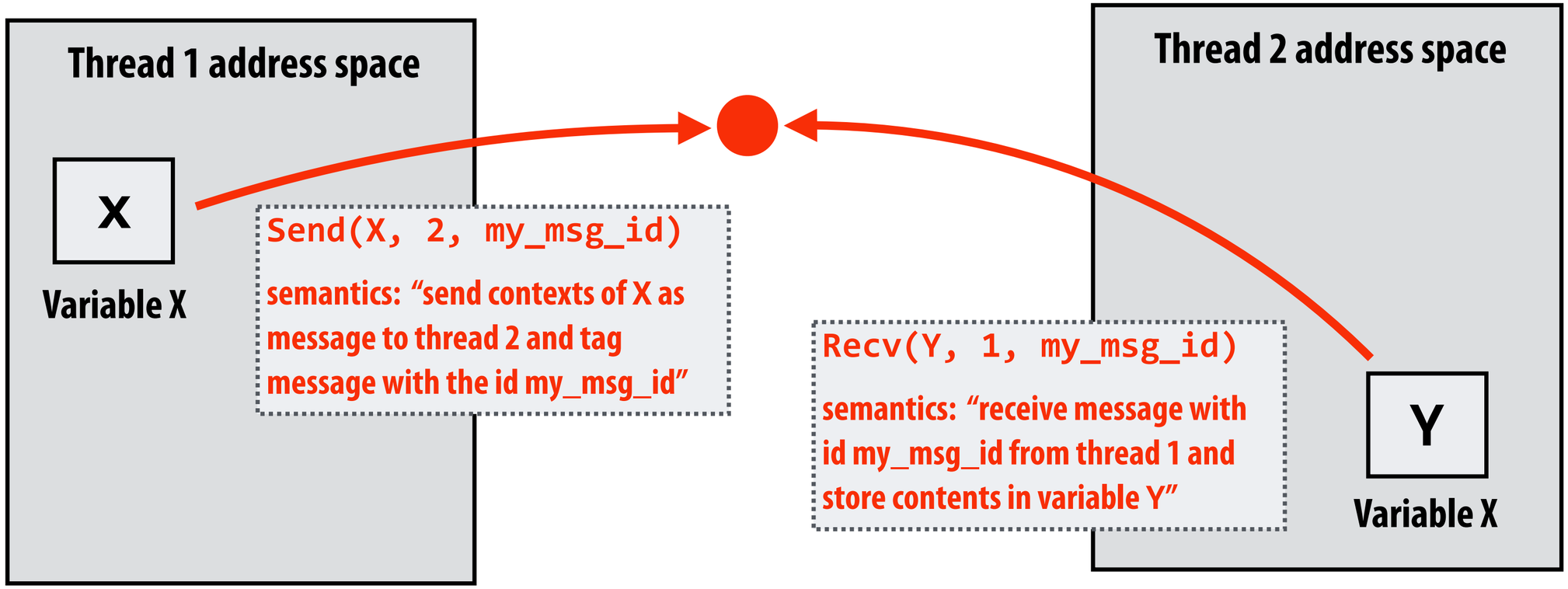
Summary: shared address space model

- **Communication abstraction**
  - Threads read/write shared variables
  - Threads manipulate synchronization primitives: locks, semaphors, etc.
  - Logical extension of uniprocessor programming
    - But NUMA implementation requires reasoning about locality for performance
- **Requires hardware support to make implementations efficient**
  - Any processor can load and store from any address (share address space)
  - NUMA designs more scalable than uniform memory access
    - Even so, costly to scale (one of the reasons why supercomputers are expensive)

Message passing model of communication

Message passing model (abstraction)

- Threads operate within their own private address spaces
- Threads communicate by sending/receiving messages
  - Explicit communication via point-to-point messages
  - send: specifies buffer to be transmitted, recipient, optional message "tag"
  - receive: specifies buffer to store data, sender, and (optional) message tag
  - Sending messages is the only way to exchange data between thread 1 and 2

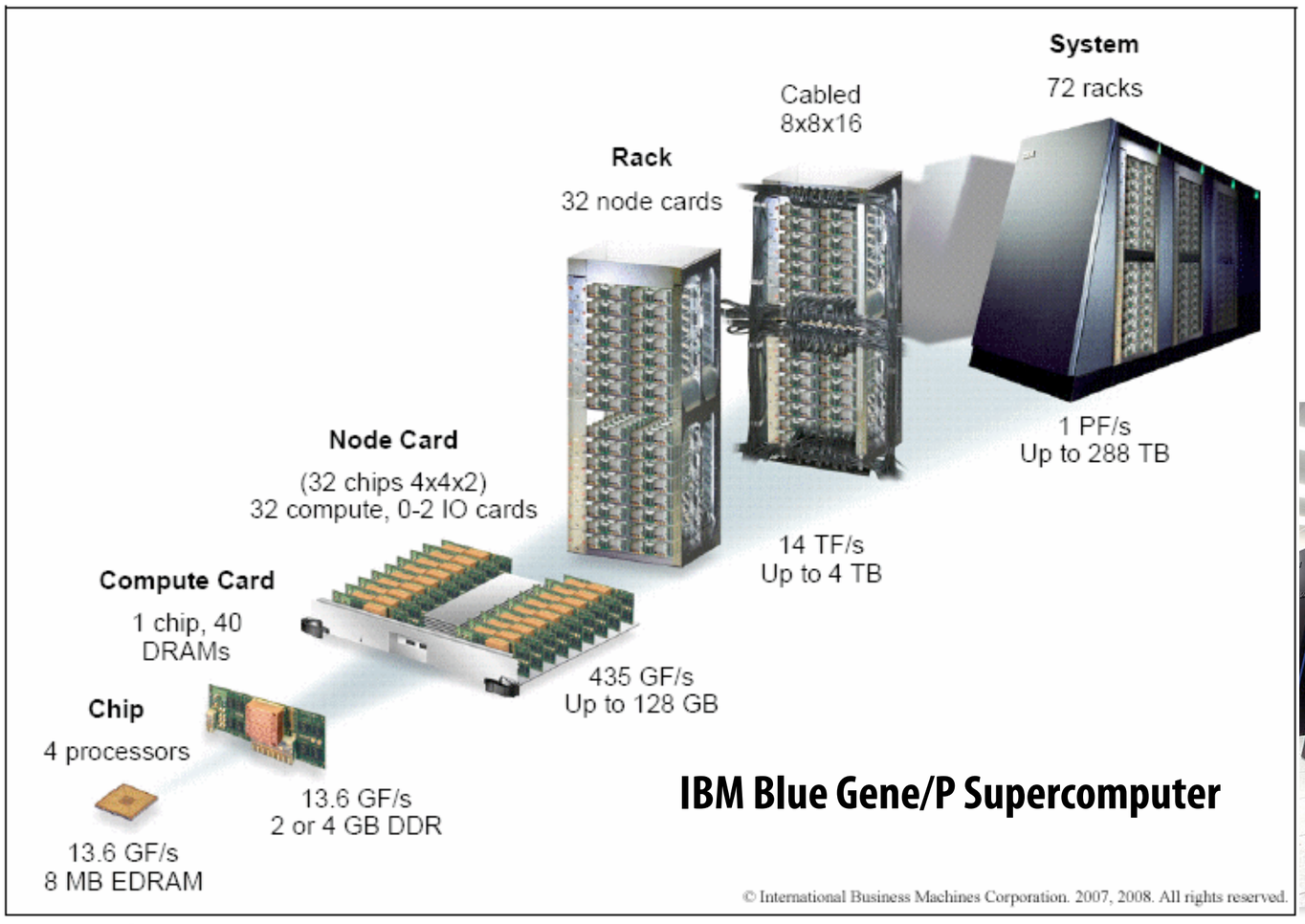


(Communication operations shown in red)

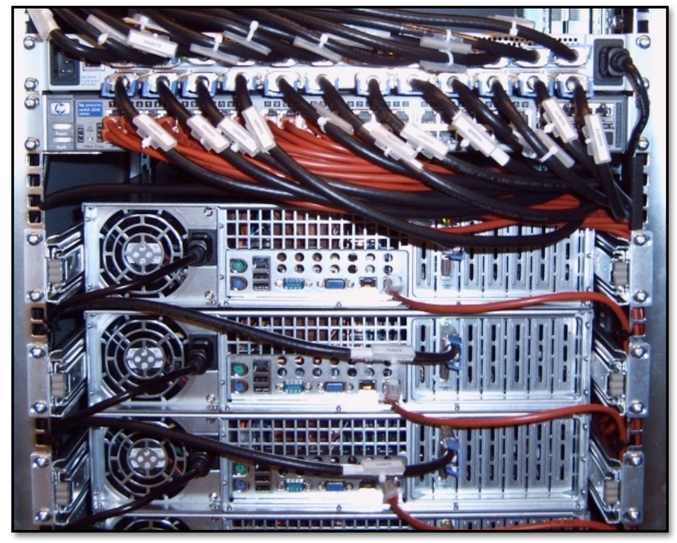
Illustration adopted from Culler, Singh, Gupta

Message passing (implementation)

- Popular software library: MPI (message passing interface)
- Hardware need not implement system-wide loads and stores to execution message passing programs
  - Connect commodity systems together to form large parallel machine
  - Parallel programming for clusters



IBM Blue Gene/P Supercomputer



Cluster of workstations (Infiniband network)

Image credit: IBM

The correspondence between programming models and machine types is fuzzy

- 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
  - Page-fault handler issues appropriate network requests

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

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 structure
  - Programs structured as performing same function on different data elements of a collection

Data-parallel model

- **Historically: same operation on each element of an array**
  - Matched capabilities SIMD supercomputers of 80's
  - Connection Machine (CM-1, CM-2): thousands of processors, one instruction decode
  - 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 independently
  - `function` may be a complicated sequence of logic (e.g., a loop body)
  - Synchronization is implicit at the end of the map (`map` returns when function has been applied to all elements of `collection`)

Data parallelism in ISPC

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

// initialize N elements of x here

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

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

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

`foreach` construct is a map

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

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

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

Data parallelism in ISPC

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

// initialize N/2 elements of x here

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

Think of loop body as `function`

`foreach` construct is a map

Collection is implicitly defined by array indexing logic

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

This is also a valid ISPC program!

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

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

Data parallelism in ISPC

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

// initialize N elements of x

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

Think of loop body as `function`

`foreach` construct is a map

Collection is implicitly defined by array indexing logic

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

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). It is not intended to help programmers write programs with that structure.

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

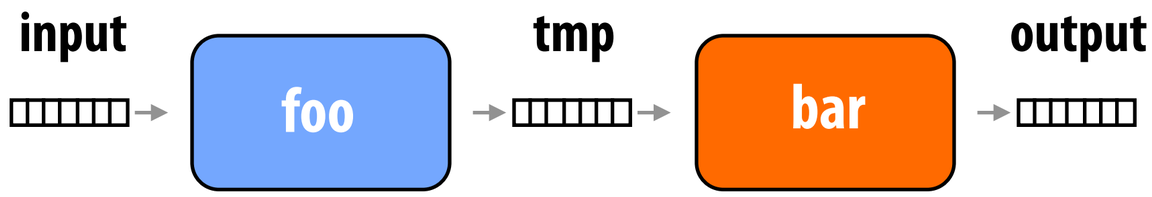
Streams: collections of elements. Elements can be processed independently

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 by compiler:

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

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

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

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

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:

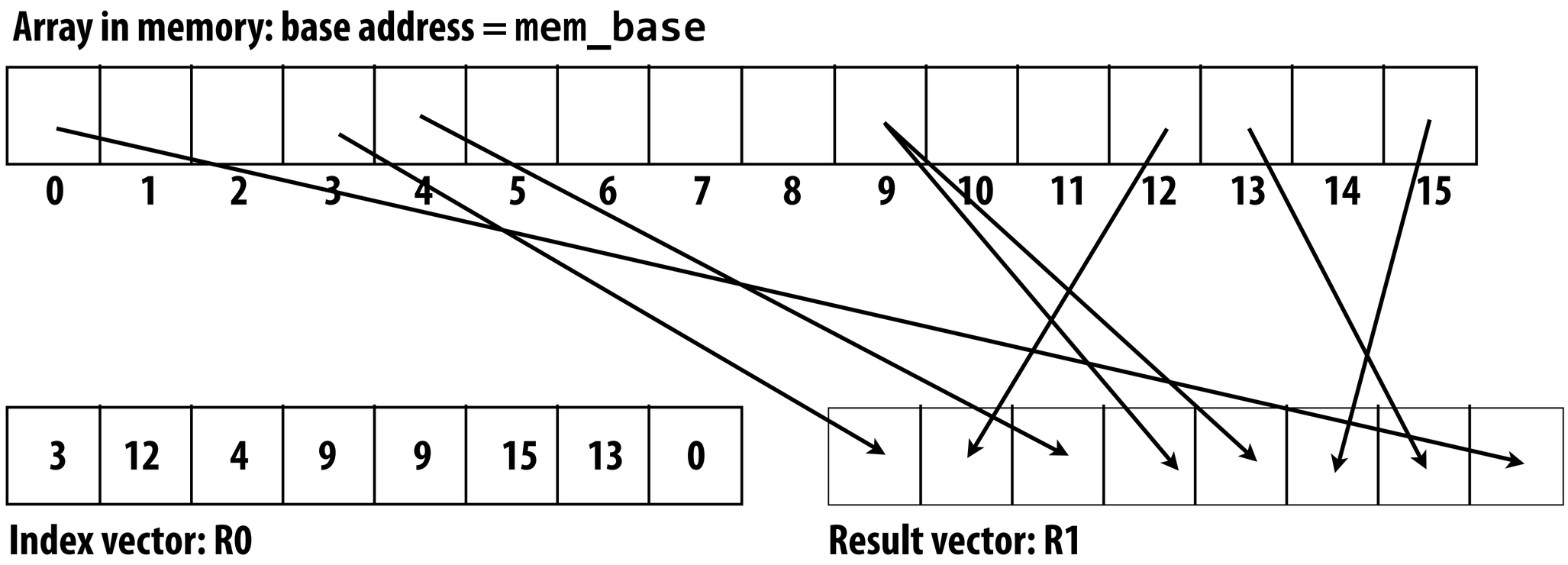
```
// main program:
const int N = 1024;
stream<float> input(N);
stream<int> indices;
stream<float> tmp_output(N);
stream<float> output(N);



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 AVX2 does not 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)

Summary: data-parallel model

- Data-parallelism is about imposing rigid program structure
- In spirit, map a single function onto a large collection of data
  - Functional: side-effect free execution
  - No communication among invocations (allow invocations to be scheduled in any order, including in parallel)
- In practice that's how many simple programs work
- But... most practical parallel languages do not enforce this structure
  - ISPC, OpenCL, CUDA, etc.
  - They choose flexibility/familiarity of imperative syntax over safety of (but complex compiler optimizations required by) systems with functional syntax
  - It's been their key to success (and the recent adoption of parallel programming)
  - Sure, functional thinking is great, but programming languages sure should impose structure to facilitate achieving high-performance implementations, not hinder hinder them

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 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 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 practice: mixed programming models

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

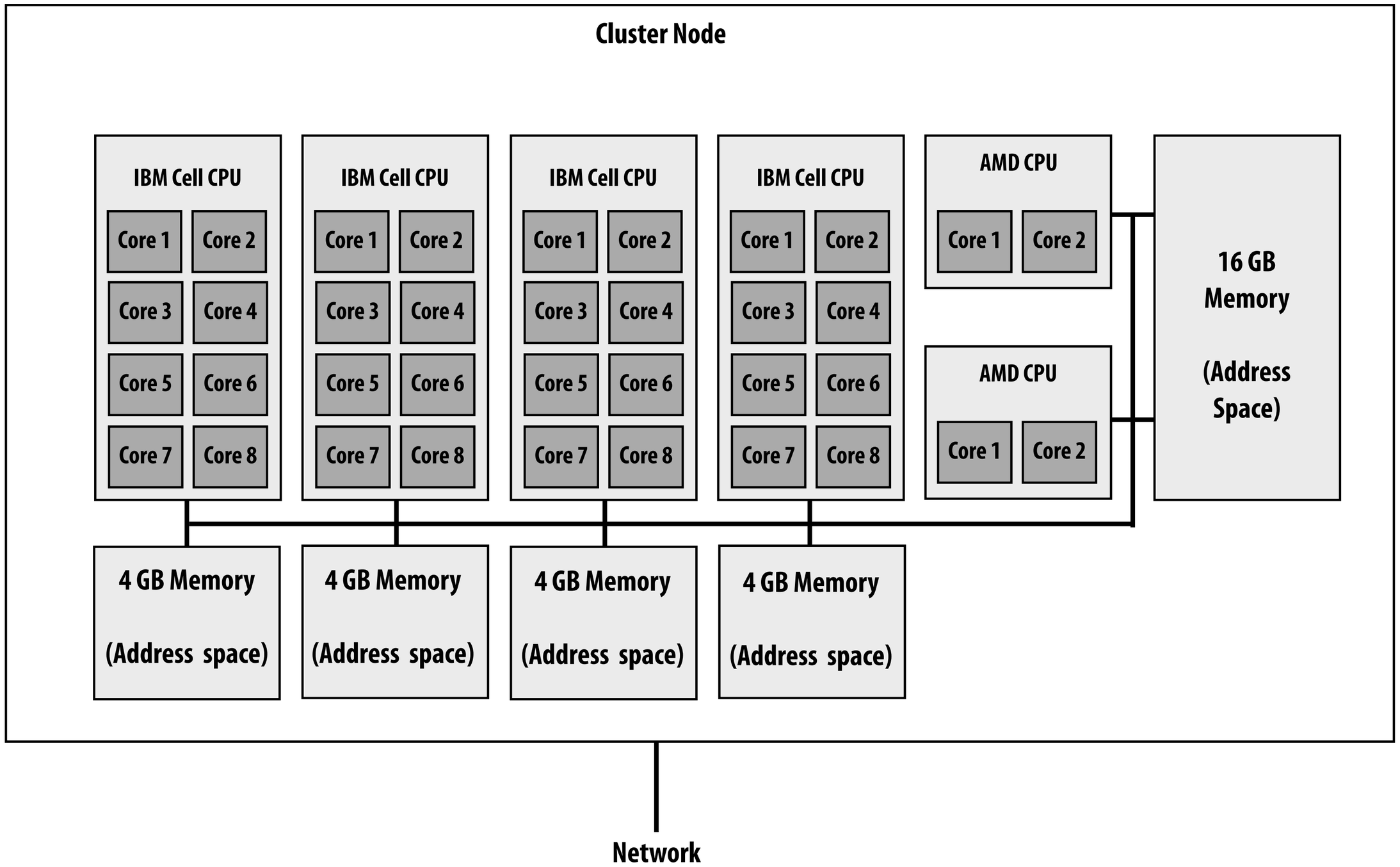
- Data-parallel-ish programming models (CUDA, OpenCL) support shared-memory style synchronization primitives in kernels
  - Permit limited forms of inter-iteration communication

- 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: 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

- 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