Multi-core Programming

3. Parallel Programming Models

并行程序设计模型

Today's theme

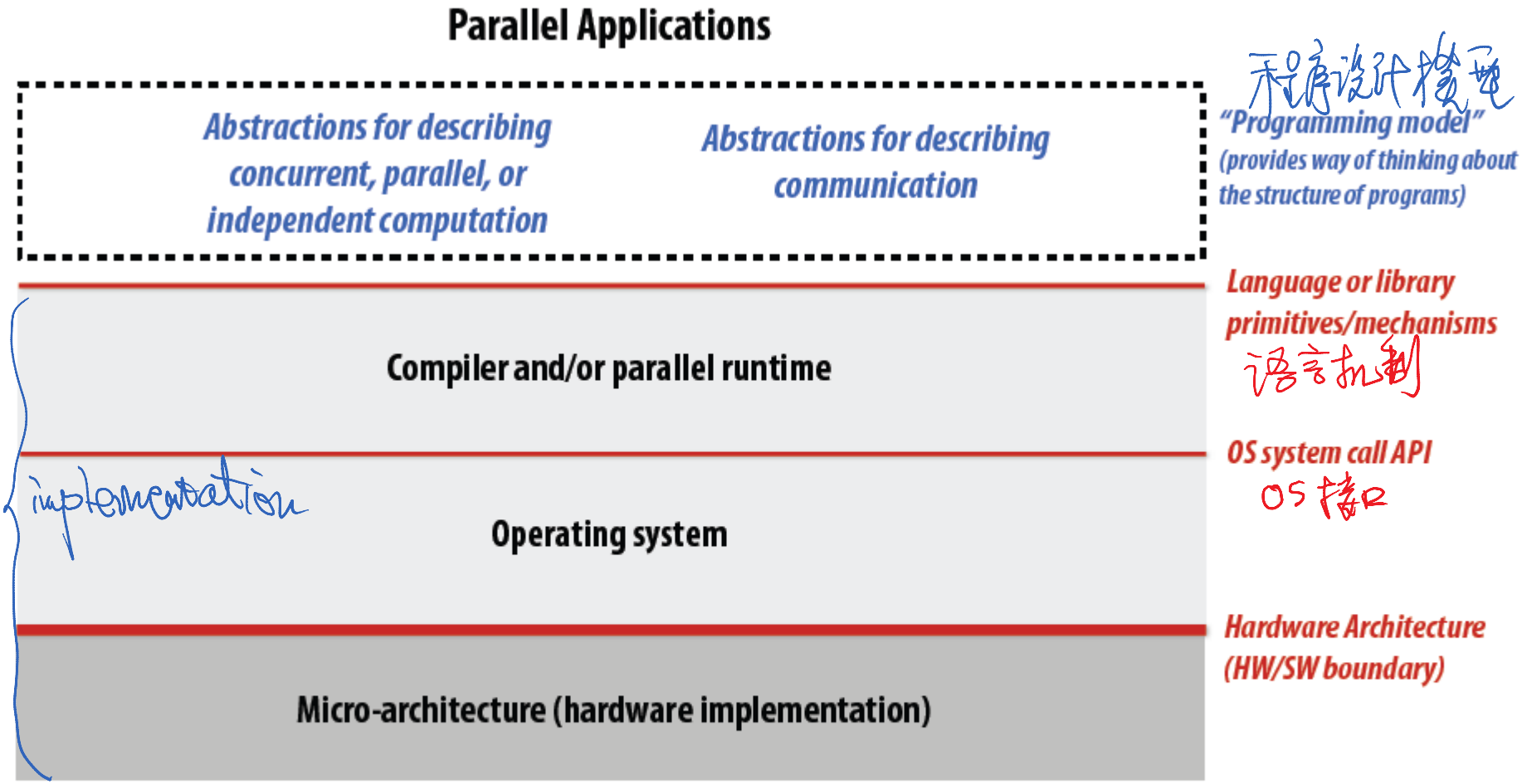
- **Parallel Programming Abstraction vs. implementation**

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

Programming Abstractions and implementations

- **Three parallel programming models** (SAS, MP, DP)
  - **That differ in communication abstractions presented to the programmer**
  - **Programming models influence how programmers think when writing programs**
- **Three machine architectures** (SAS, cluster, SIMD)
  - **Abstraction presented by the hardware to low-level software**
  - **Typically reflect hardware implementation's capabilities**
- **We'll focus on differences in communication and cooperation**

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

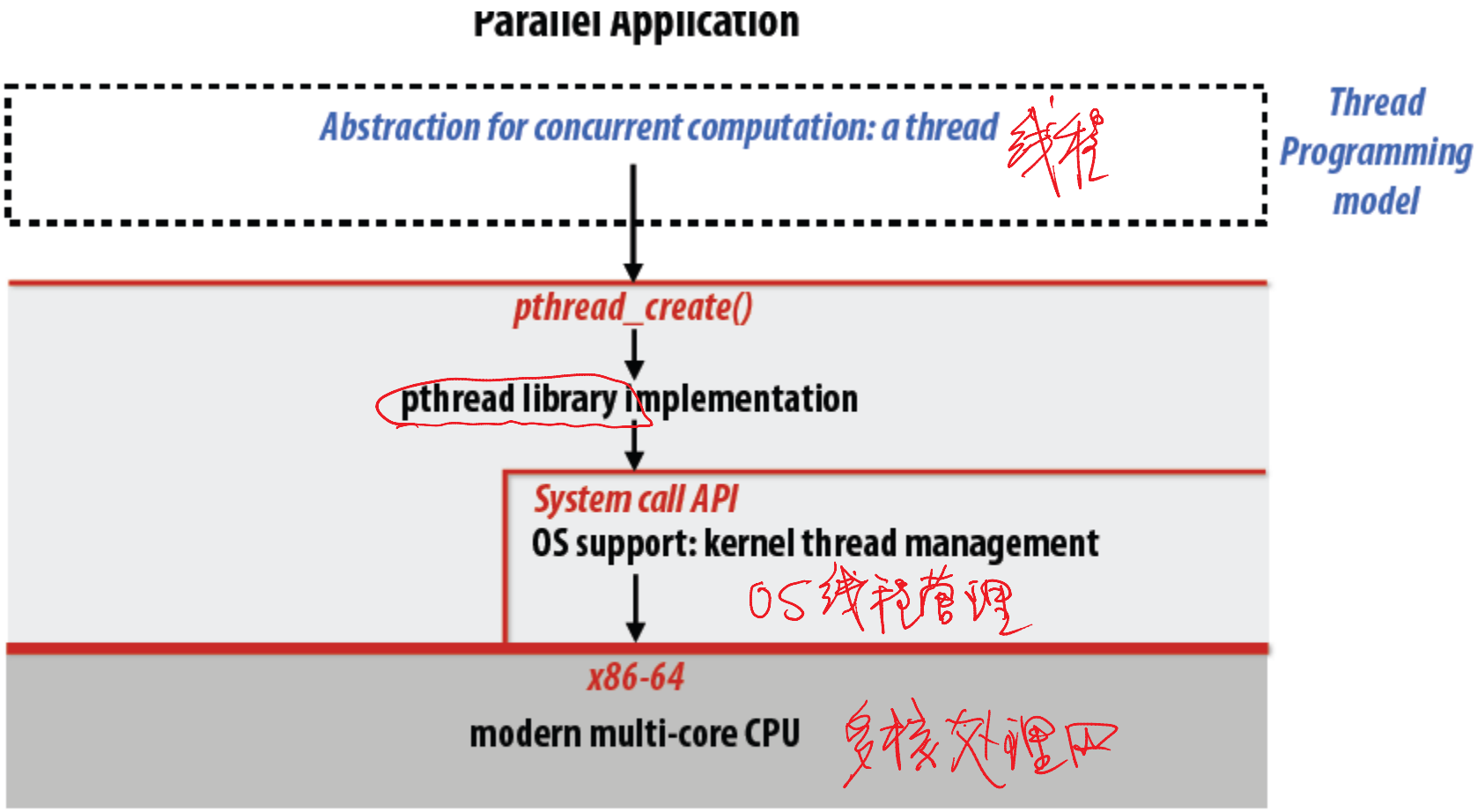


Blue italic text: abstraction/concept

Red italic text: system interface

Black text: system implementation

Example: expressing parallelism with posix threads

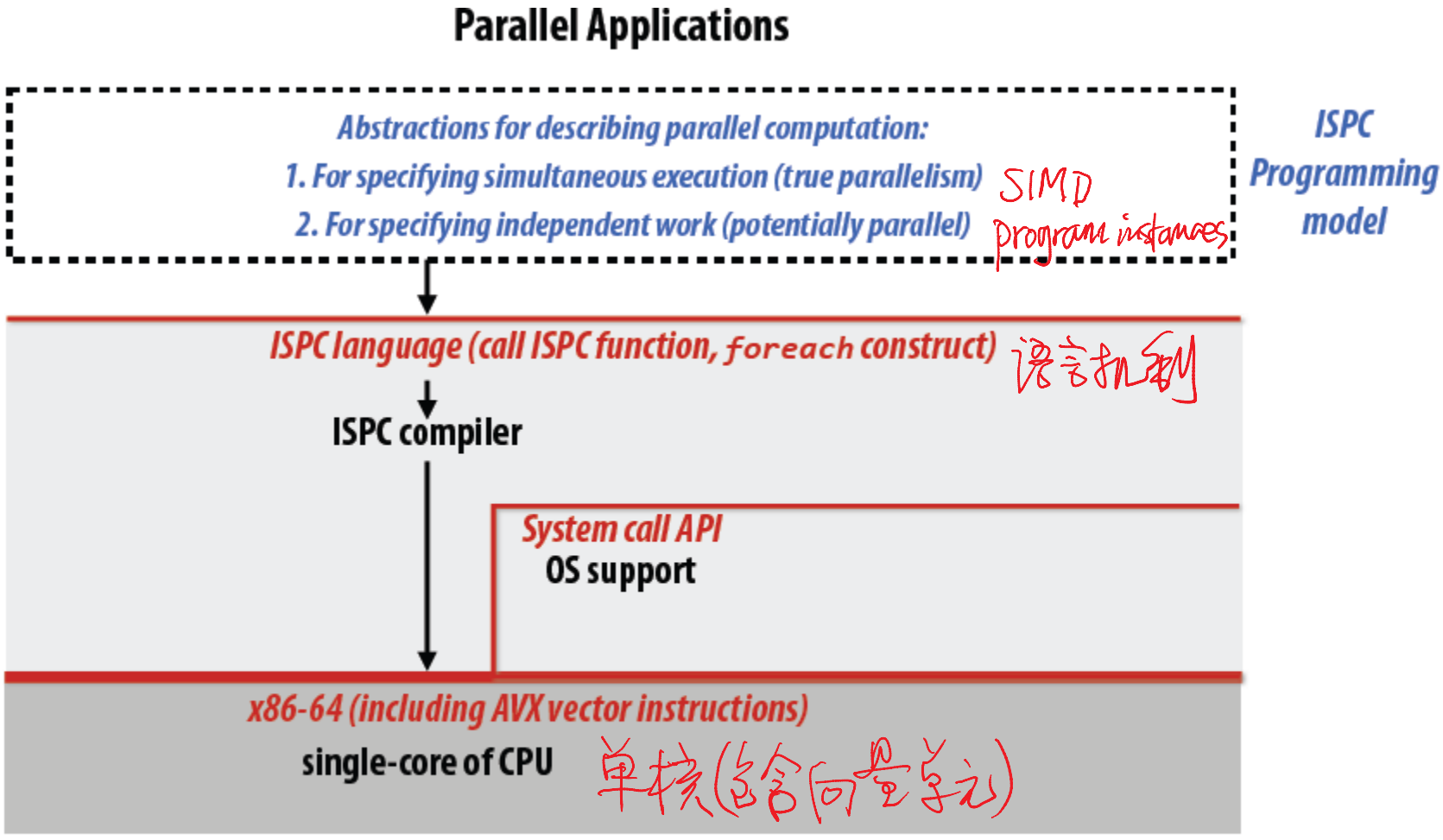


Blue italic text: abstraction/concept

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

共享地址空间

消息传递

数据并行

也是3种程序设计模型.

共享地址空间模型 (单一地址空间)

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

共享存储(变量)

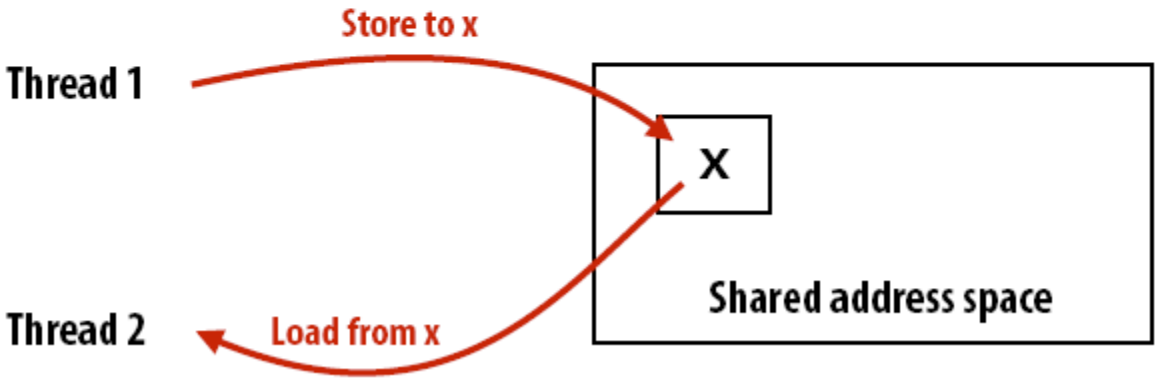
Thread 1:

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

创建foo线程

Thread 2:

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



(Communication operations shown in red)

可以理解为同一个进程中的线程.
共享 Global Data.

Synchronization primitives are also shared variables: e.g., locks 锁

Thread 1:

```
int x = 0;
Lock my_lock;

spawn_thread(foo, &x, &my_lock);


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

mylock is shared variable

如果不上锁，x的值将会如何？

Thread 2:

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

  print x;
}
```

Shared address space model (abstraction)

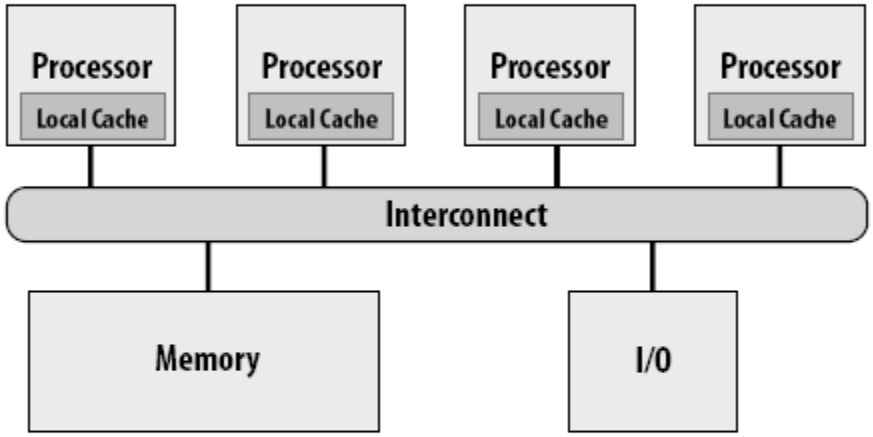
- Threads communicate by:
  - Reading/writing to shared variables
    - Inter-thread communication is implicit in memory operations
    - Thread 1 stores to X
    - Later, thread 2 reads X (and observes update of value by thread 1)
  - Manipulating synchronization primitives
    - e.g., ensuring mutual exclusion via use of locks
- This is a natural extension of sequential programming
  - In fact, all our discussions in class have assumed a shared address space so far!
- Helpful analogy: shared variables are like a big bulletin board
  - Any thread can read or write to shared variables

HW implementation of a shared address space

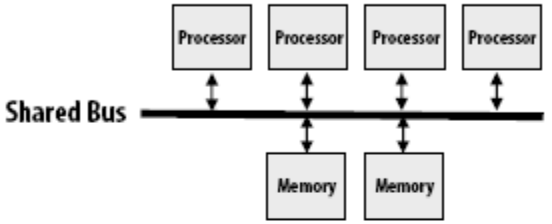
硬件实现(体系结构)

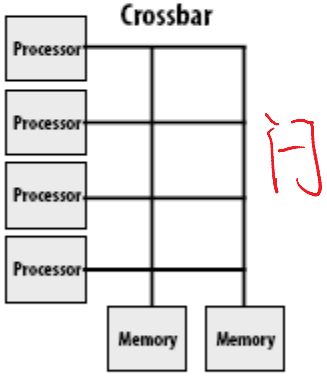
Key idea: any processor can directly reference any memory location

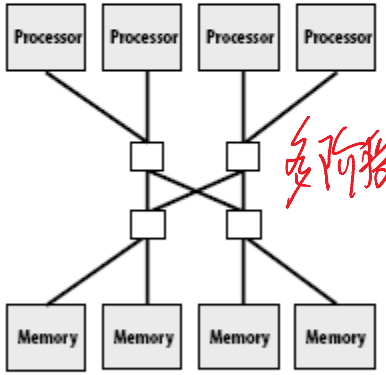
"Dance-hall" organization



Interconnect examples







Multi-stage network

对称多处理机

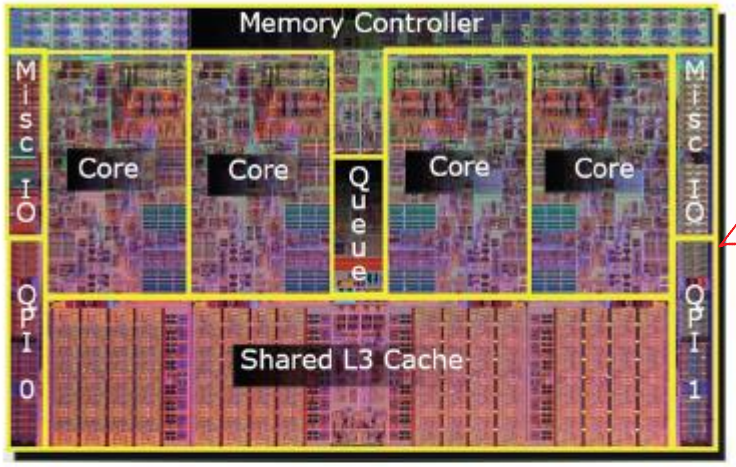
Symmetric (shared-memory) multi-processor (SMP):

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

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

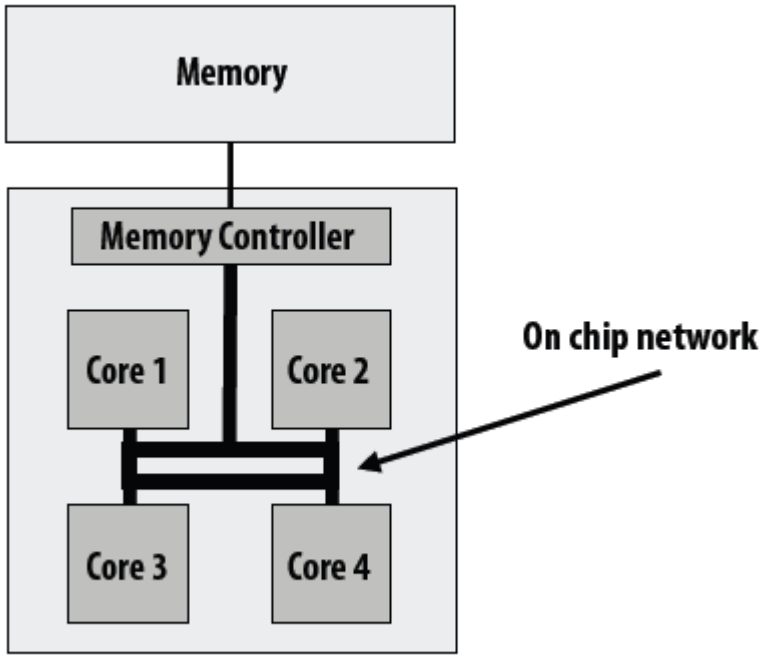
Shared address space HW architectures

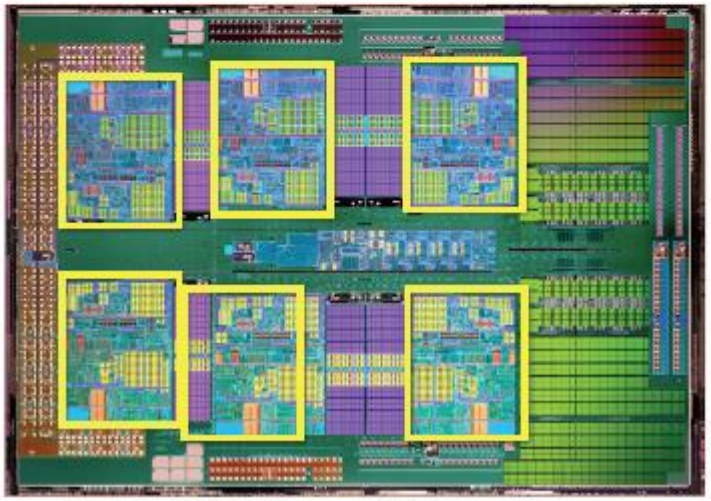
Commodity x86 examples



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

4核





AMD Phenom II (six core)

6核

多核处理四

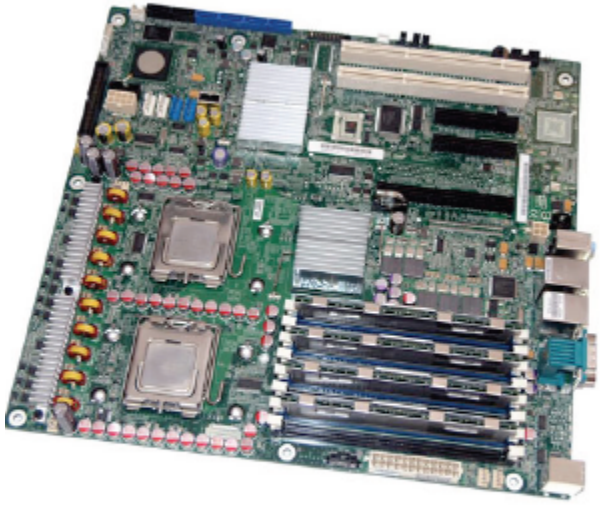
Non-uniform memory access (NUMA)

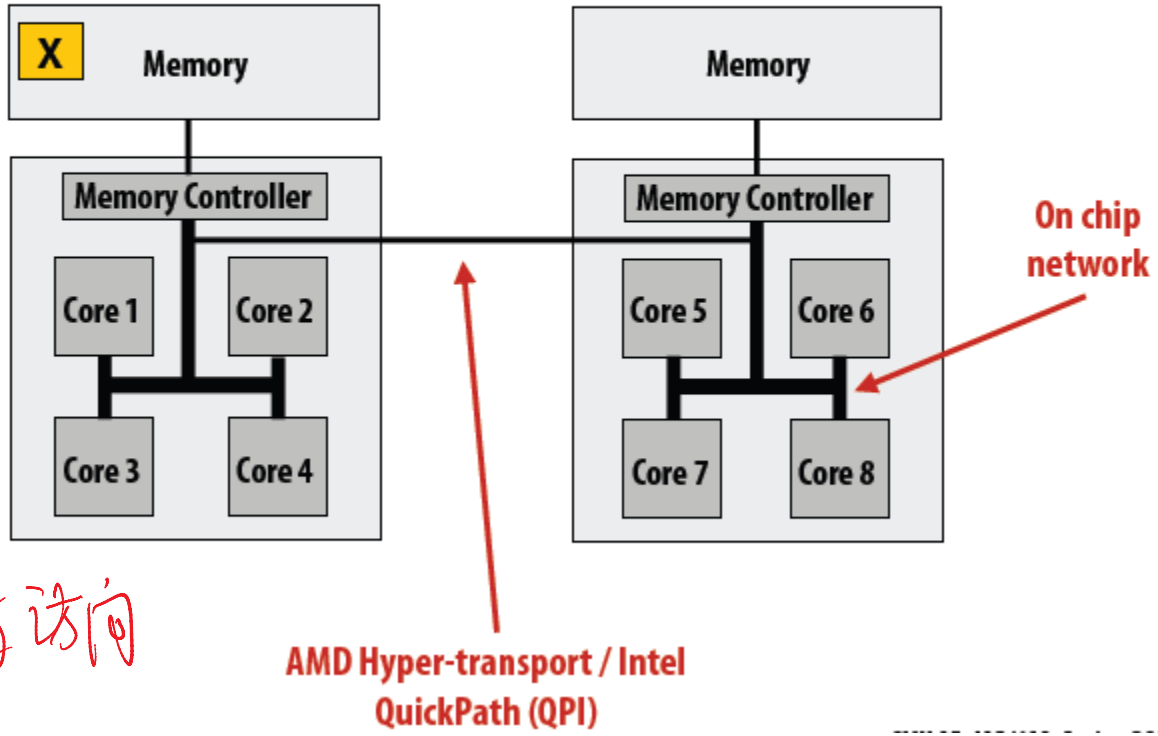
All processors can access any memory location, but... the cost of memory access (latency and/or bandwidth) is different for different processors

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

对x的访问速度不同

Example: modern dual-socket configuration

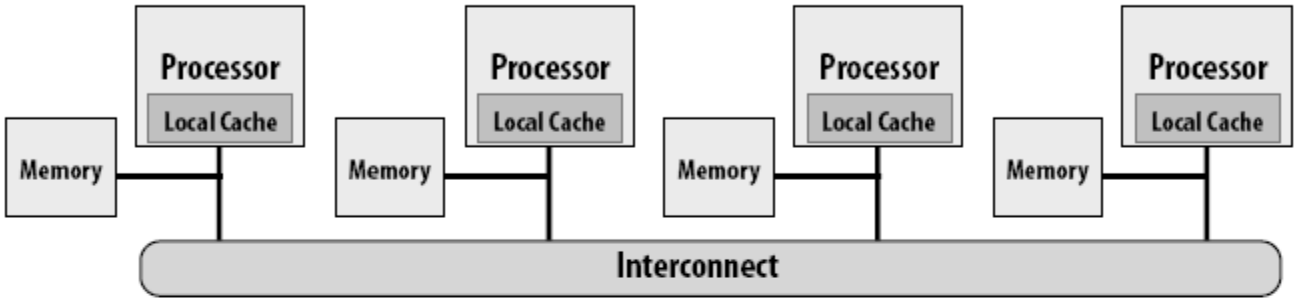




非一致存储访问

Non-uniform memory access (NUMA)

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



- Problem with preserving uniform access time in a system: scalability 伸缩性差
  - GOOD: costs are uniform, BAD: they are uniformly bad (memory is uniformly far away)
- NUMA designs are more scalable 可伸缩性好
  - Low latency access to local memory
  - Provide high bandwidth to local memory
- Cost is increased programmer effort for performance tuning
  - Finding, exploiting locality is important to performance (want most memory accesses to be to local memories) 局部性访问对效率影响大

Summary: shared address space model

- Communication abstraction
  - Threads read/write shared variables
  - Threads manipulate synchronization primitives: locks, semaphors, etc.
  - Logical extension of uniprocessor programming *
- Requires hardware support to implementation efficiently
  - Any processor can load and store from any address (its shared address space!)
  - Even with NUMA, costly to scale (one of the reasons why supercomputers are expensive)

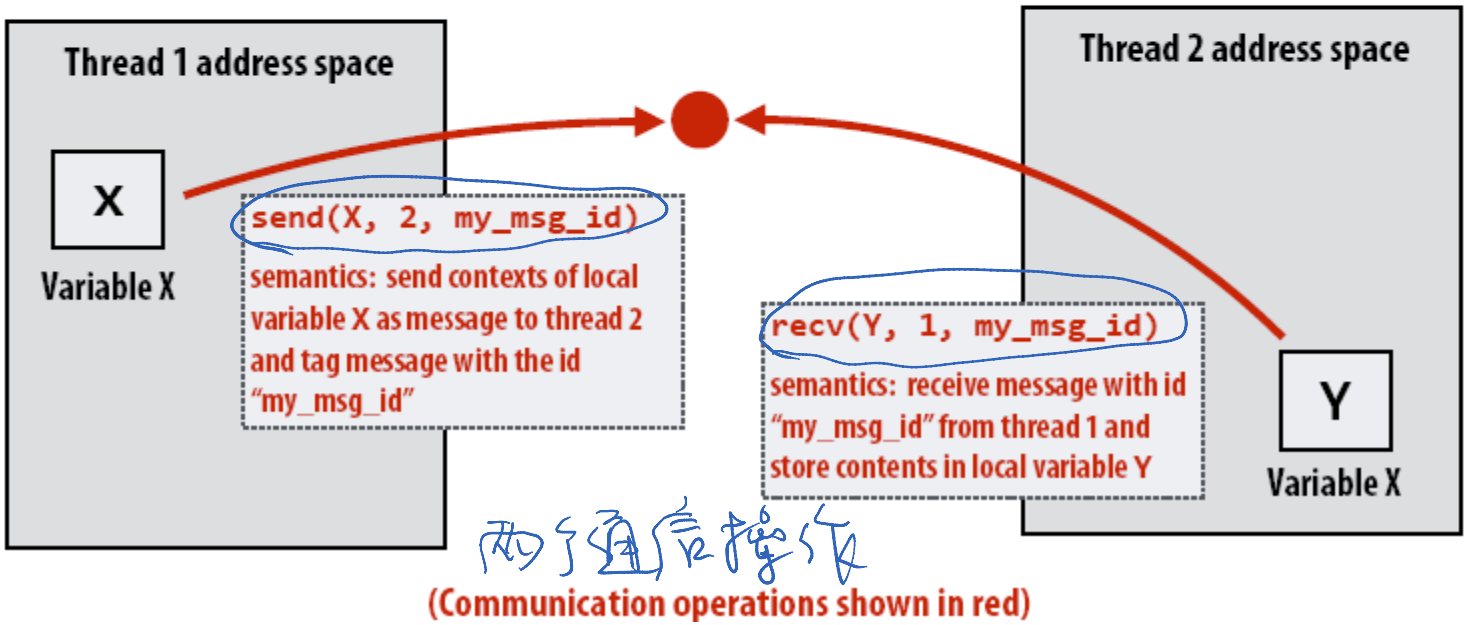
* But NUMA implementation requires reasoning about locality for performance

消息传递模型

MESSAGE PASSING MODEL OF COMMUNICATION

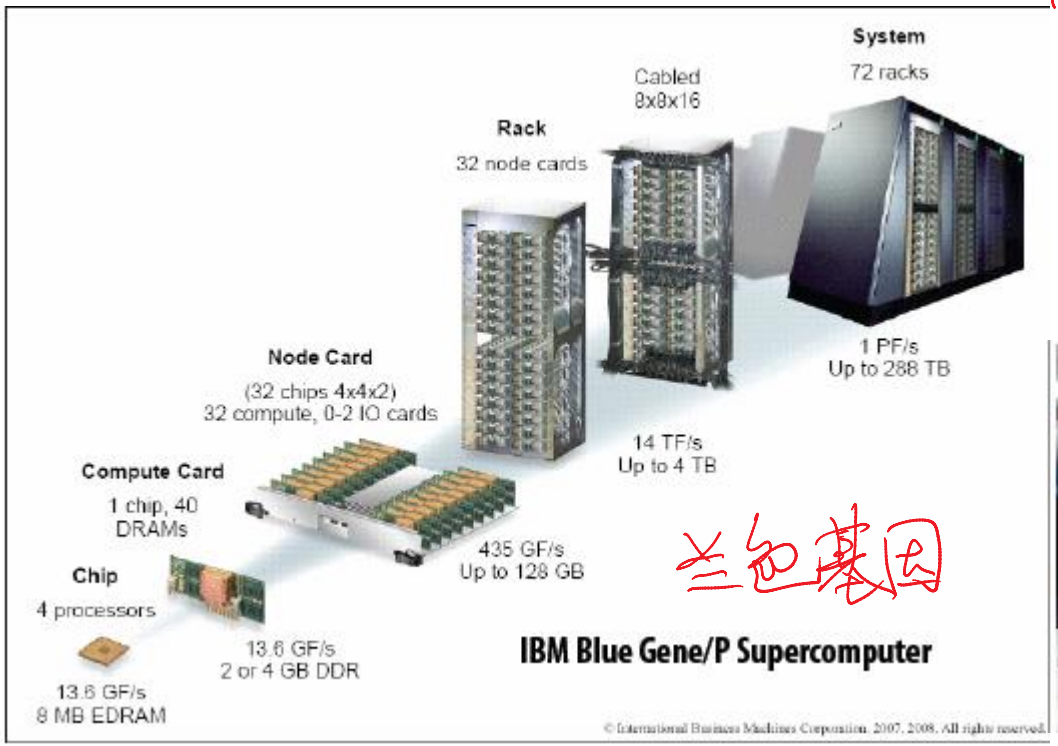
Message passing model (abstraction)

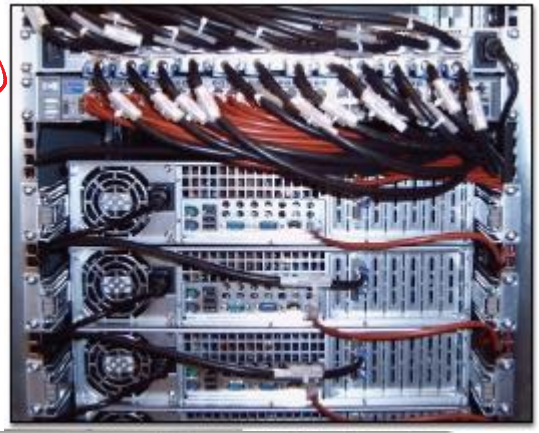
- Threads operate within their own private address spaces
- Threads communicate by sending/receiving messages
  - send: specifies recipient, buffer to be transmitted, and optional message identifier ("tag")
  - receive: sender, specifies buffer to store data, and optional message identifier
  - Sending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



Message passing (implementation)

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





Cluster of workstations
(Infiniband network)

Caveat: 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 solutions)
  - Mark all pages with shared variables as invalid
  - Page-fault handler issues appropriate network requests
- Keep clear in your mind: what is the programming model (abstractions used to specify program)? And what is the HW implementation?

THE DATA-PARALLEL MODEL

数据并行模型

programming models serve to impose structure on programs

- ***Shared address space***: very little structure to communication
  - All threads can read and write to all shared variables
  - Pitfall: due to implementation: not all reads and writes have the same cost
    (and that cost is not apparent in program text)
- ***Message passing***: highly structured communication
  - All communication occurs in the form of messages (can read program and see where the communication is)
- ***Data-parallel***: very rigid computation structure
  - Programs perform same function on different data elements in a collection

Data-parallel model

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

将循环到SIMD.

```
// 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);
```

数据并行

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.

对X,Y数组元素依映射到循环体上.

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

将循环体映射到X和Y的各元素(隐含着)

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.

隐含映射(利用数组的下标)

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

ISPC没有专门的映射语义结构.
而foreach隐含地指明了这种语义

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];
    }
}
```

不同的迭代可能写同一个值，写的顺序是不确定的，这个做法安全总是不确定的.

The output of this program is undefined!

Possible for multiple iterations of the loop body to write to same memory location

Data-parallel model (foreach) provides no specification of order in which iterations occur

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

这种结构尽量避免.

Data parallelism: a more "pure" approach

Note: this is not ISPC syntax

Main program:

```
const int N = 1024;

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

// initialize N elements of x here...

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

"Kernel" definition:

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

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

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

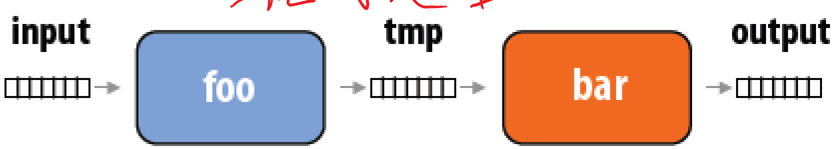
Kernels: side-effect-free functions. Operate element-wise on collections

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

Stream programming benefits

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

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




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

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

Independent processing on elements, kernel functions are side-effect free:

- Optimization: parallelize kernel execution
- Application cannot write a program that is non-deterministic under parallel execution

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

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

Stream programming drawbacks 缺点

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

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

absolute_value(tmp, output);
```

Kernel

复杂数据流的描述需库的支持

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

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

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

Kayvon's experience:

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

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

Gather/scatter: two key data-parallel communication primitives

数据并行的通信元语(两个流操作函数)

Map absolute_value onto stream produced by gather:

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

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

需要重组输入流

Map absolute_value onto stream, scatter results:

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

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

需要分散输出流

ISPC equivalent: 等价的ISPC程序

```
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;
    }
}
```

输入间接引用

ISPC equivalent:

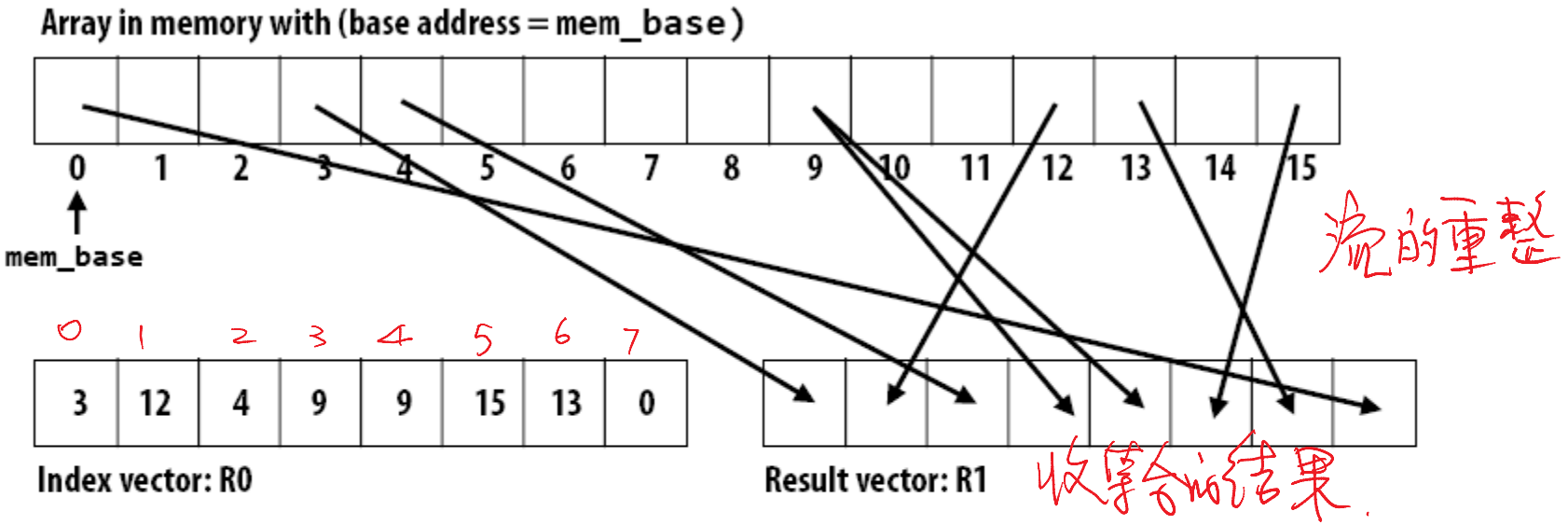
```
export void absolute_value(
    uniform float N,
    uniform float* input,
    uniform float* output,
    uniform int* indices)
{
    foreach (i = 0 ... n)
    {
        if (input[i] < 0)
            output[indices[i]] = -input[i];
        else
            output[indices[i]] = input[i];
    }
}
```

输出间接引用

Gather instruction

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

"Gather from buffer mem_base into R1 according to indices specified by R0."



流的重整

收集合的结果

Gather supported with AVX2 in 2013 AVX2中支持gather指令

But AVX2 does not support SIMD scatter (must implement as scalar loop)

Scatter instruction exists in AVX512 AVX512中支持Scatter指令

Hardware supported gather/scatter does exist on GPUs. (GPU支持，但expensive指令)

(still an expensive operation compared to load/store of contiguous vector)

Summary: data-parallel model

- Data-parallelism is about imposing rigid program structure to facilitate simple programming and advanced optimizations
- Basic structure: map a function onto a large collection of data
  - Functional: side-effect free execution
  - No communication among distinct function invocations (allow invocations to be scheduled in any order, including in parallel)
- In practice that's how many simple programs work
- But... many modern performance-oriented data-parallel languages do not strictly enforce this structure
  - ISPC, OpenCL, CUDA, etc.
  - They choose flexibility/familiarity of imperative C-style syntax over the safety of a more functional form: it's been their key to their adoption
  - Opinion: sure, functional thinking is great, but programming systems sure should impose structure to facilitate achieving high-performance implementations, not hinder them

基本结构

实际上，很多程序按此方法

大多现代的数据并行实现并没有严格受限于基本结构.

Summary: Programming models

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

并行程序的组织结构(方式)

- **They provide abstractions that permit multiple valid implementations.** (且允许多种实现)

Summary: Programming models

Restrictions imposed by these abstractions are designed to:

1. **Reflect realities of parallelization and communication costs to programmer (help a programmer write efficient programs)** 程序设计模型有利于帮助程序员书写有效程序.
   - **Shared address space machines: hardware supports any processor accessing any address**
   - **Messaging passing machines: hardware may accelerate message send/receive/buffering**
   - **Desirable to keep "abstraction distance" low so programs have predictable performance, but want abstractions to be high enough for code flexibility/portability**
2. **Provide useful information to implementors of optimizing compilers/runtimes/hardware to help them efficiently implement programs using these abstractions**
   对于实现者,提供充分的信息进行各种优化.

We discussed 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" over a collection
  - Assumes a shared address space from which to load inputs/store results, but model severely limits communication between iterations of the map (goal: preserve independent processing of iterations)
  - Modern embodiments encourage, but don't enforce, this structure

3 Architectures: SMP (Multi-core), Cluster, SIMD (Vector)

Modern practice: mixed programming models

现在实际情况是混合模型

- Use shared address space programming within a multi-core node of a cluster, use message passing between nodes
  - Very, very common in practice
  - Use convenience of shared address space where it can be implemented efficiently (within a node), require explicit communication elsewhere
- Data-parallel-ish programming models support shared memory style synchronization primitives in kernels
  - Permit limited forms of inter-iteration communication (e.g., CUDA, OpenCL)
- In a future lecture… CUDA/OpenCL use data-parallel model to scale to many cores, but adopt shared-address space model allowing threads running on the same core to communicate.

在集群的多核结点上使用SAS，在结点之间使用MP

数据并行模型中也允许共享存储形式的同步

数据并行用于many core，SAS用于多核.

Questions to consider

(思考题，作为Homework2)

1. **Programming models enforce different forms of structure on programs. What are the benefits of data-parallel structure?**
2. **With respect to the goals of efficiency/performance… what do you think are problems of adopting a very high level of abstraction in a programming system? What about potential benefits?**
3. **Choose a popular parallel programming system (for example MapReduce, Spark, or Cilk) and try and describe its programming model (how are communication and execution expressed?)**