

---

# 并行处理

## L02-02: 并行处理过程的**抽象** **(Abstractions)**

及其相应的硬件/软件协同**实现** **(implementations)**

叶笑春

中国科学院计算技术研究所

---

# Abstraction抽象 vs. Implementation实现

是本课程的关键

**讨论并行时，抽象与实现是非常容易混淆的**

---

# An example: Programming with ISPC

## ■ Intel SPMD Program Compiler (ISPC)

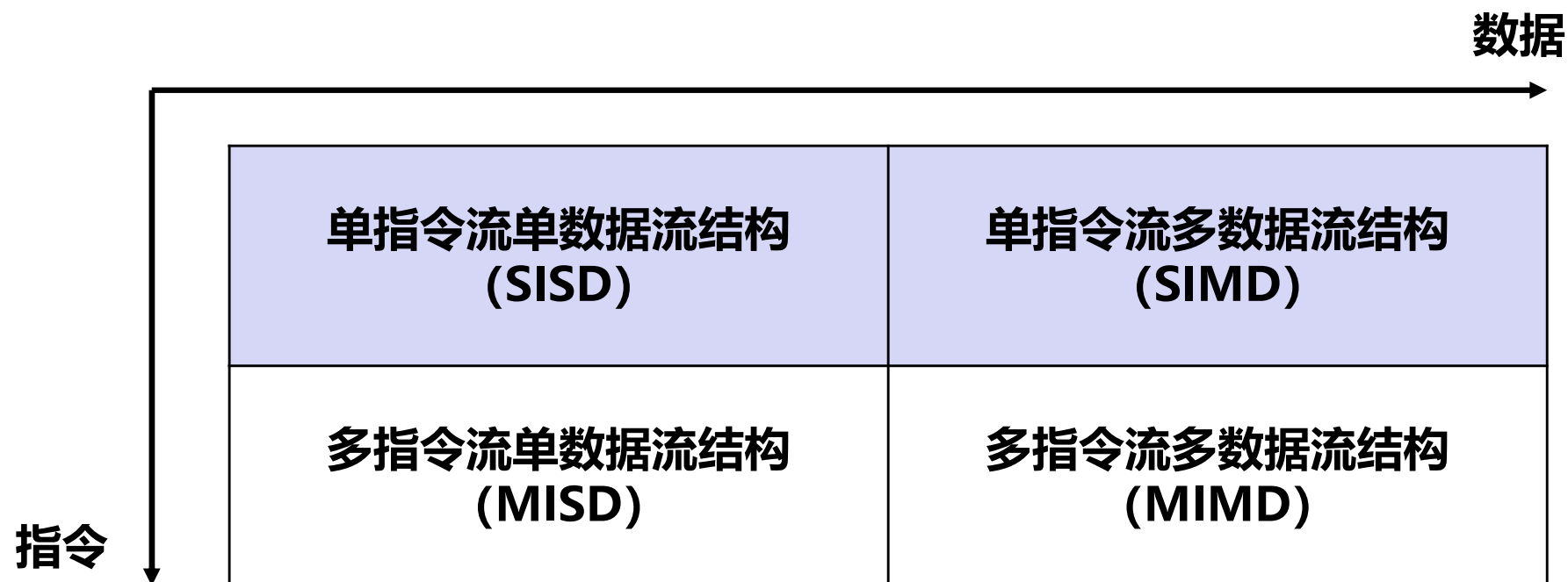
- An open-source compiler for high-performance SIMD programming on the CPU and GPU

## ■ 单程序多数据 (SPMD) : single program multiple data

<https://ispc.github.io>

# 并行分类

## ■ Flynn分类法



**SPMD: 单程序多数据** single program multiple data

# SPMD 与SIMD的区别

---

## ■ 单程序多数据single program multiple data(SPMD):

- 任务的粒度更粗（处理数据的任务单位是整个程序），
- SPMD是从整个程序级上看的，并行粒度更粗，意味着处理的多数据**不一定是执行相同的指令操作**，因为程序里面可以有分支等，即执行路径可以是多条。

## ■ 单指令多数据single instruction multiple data(SIMD):

- 任务的粒度更细（处理数据的任务单位是单个指令），
- SIMD是从指令级上看的，这意味着SIMD处理的多数据**是执行相同的指令操作**，比如都执行加法指令。

**思考：GPU的SIMT（单指令多线程，single instruction multiple threads）与之又有什么区别？**

# 示例：以计算 $\sin(x)$ 为例

计算 $\sin(x)$  的Taylor 展开表达式:

$$\sin(x) = x - x^3/3! + x^5/5! - x^7/7! + \dots$$

泰勒循环展开terms位, 对于 N 个浮点数数组x[N]中的每个元素, 求解 $\sin(x[i])$ , 把N个结果对应存储到result[N]中

```
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]; //分子
        int denom = 6; // 3! 分母
        int sign = -1; //符号

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

## C++ code

```
#include "sinx_ispc.h"

int N = 1024; %共有1024个x要算
int terms = 5;%泰勒循环展开5项
float* x = new float[N];
float* result = new float[N];

// initialize x here

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

## ISPC code

```
export void sinx(
    uniform int N,
    uniform int terms,
    uniform float* x,
    uniform float* result)
{
    // assume N % programCount = 0
    for (uniform int i=0; i<N; i+=programCount)
    {
        int idx = i + programIndex;
        float value = x[idx];
        float number = 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 * number / denom;
            number *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```

## SPMD 程序抽象:

- 调用 ISPC 函数会产生一“组(gang)”的 ISPC 程序实例 (program instances)
- 所有实例并发运行 ISPC 代码
- 返回时, 所有实例都已完成

# sin(x) in ISPC

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

## ISPC Keywords:

要算的N个sin(x[N])中的一个  
sin(x(i))程序，是一个程序实例

- **programCount**: 程序组 (gang) 中同时执行的实例数 (uniform value), N个程序实例并发执行
- **programIndex**: id of the current instance in the gang. (a non-uniform value: "varying" ) 当前执行的程序实例的ID序号
- **uniform**: 类型修饰符。 对于此变量, 所有实例都具有相同的值。

## ISPC code: main.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 number = 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 * number / denom;
            number *= x[idx] * x[idx];
            denom *= (2*j+2) * (2*j+3);
            sign *= -1;
        }
        result[idx] = value;
    }
}
```



# sin(x) in ISPC

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

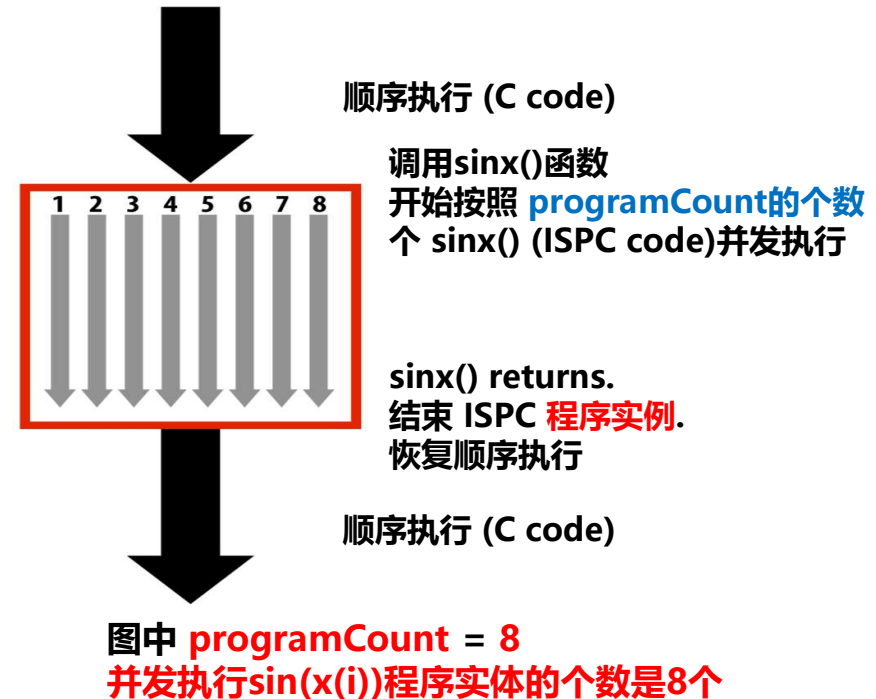
## C++ code: main.cpp

```
#include "sinx_ispc.h"

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

// initialize x here

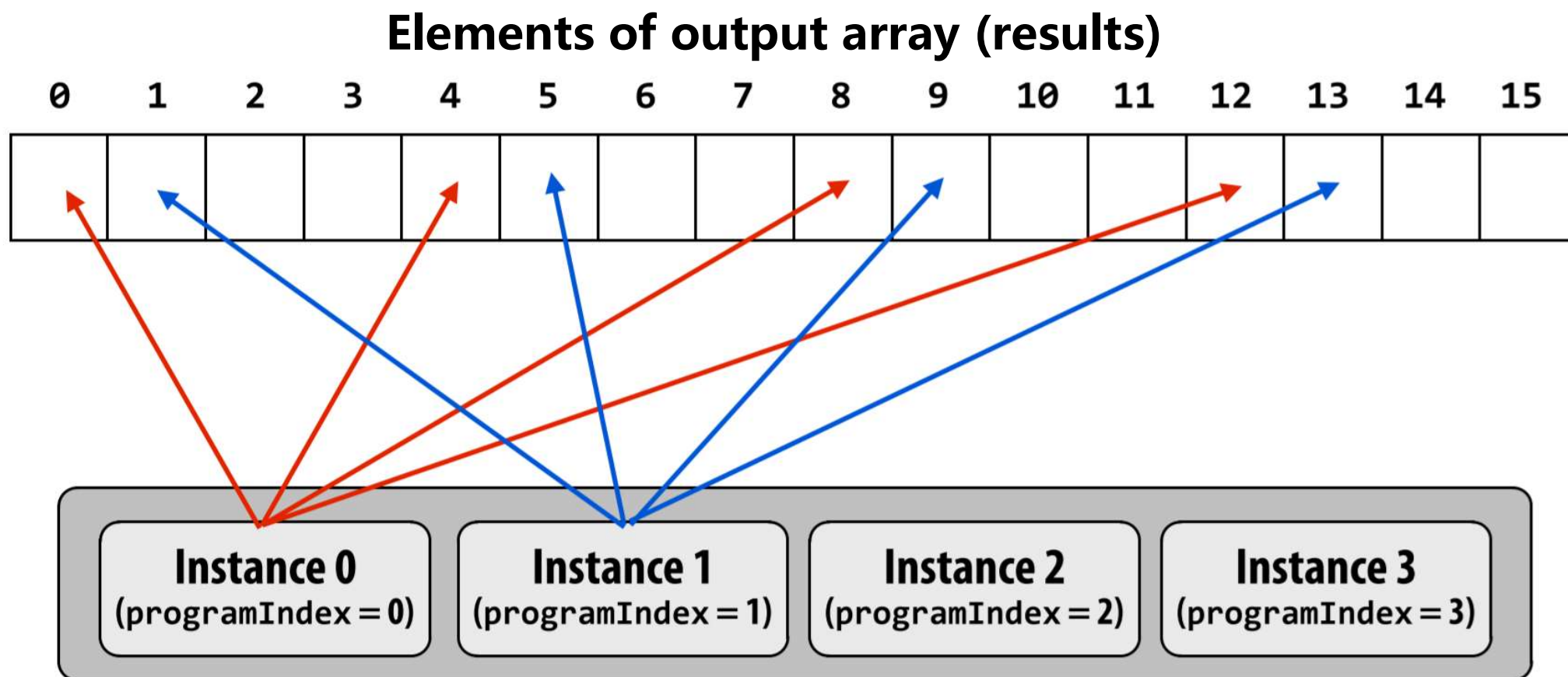
// execute ISPC code
sinx(N, terms, x, result);
```



## ISPC 编译器生成 SIMD的对应实现 (implementation)

- gang 中的**实例数 (Number of instances)** 是硬件的 **SIMD 宽度** (或 SIMD 宽度的倍数)  
ISPC 编译器生成带有 SIMD 指令的二进制文件 (.o)
- C++ 代码像往常一样链接到目标文件

# 分配方式一：交错分配 Interleaved assignment



In this illustration: gang contains four instances: **programCount=4**  
并发执行 $\sin(x(i))$ 程序实体的个数是4个

泰勒循环展开terms位，对于 16 个浮点数数组 $x[16]$ 中的每个元素，求解 $\sin(x[i])$ ，把16个结果对应存储到 $result[16]$ 中。一共需要执行4组，每组并发执行的程序实例数是4个，按照**交错分配方式**认领程序实例。

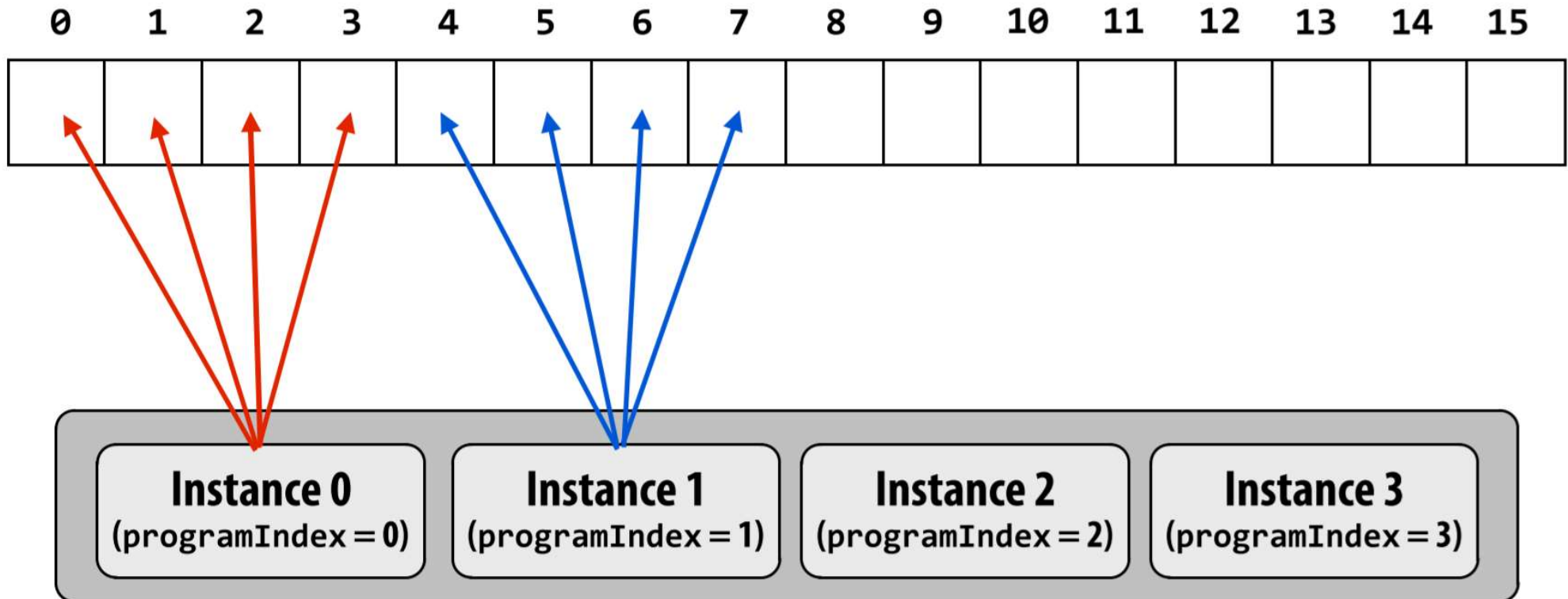
# sin(x) in ISPC: version 2

## 元素到实例的阻塞分配(Blocked assignment)

```
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 *= (2*j+2) * (2*j+3);  
            sign *= -1;  
        }  
        result[idx] = value;  
    }  
}
```

# 分配方式二：阻塞分配(Blocked assignment)

Elements of output array (results)



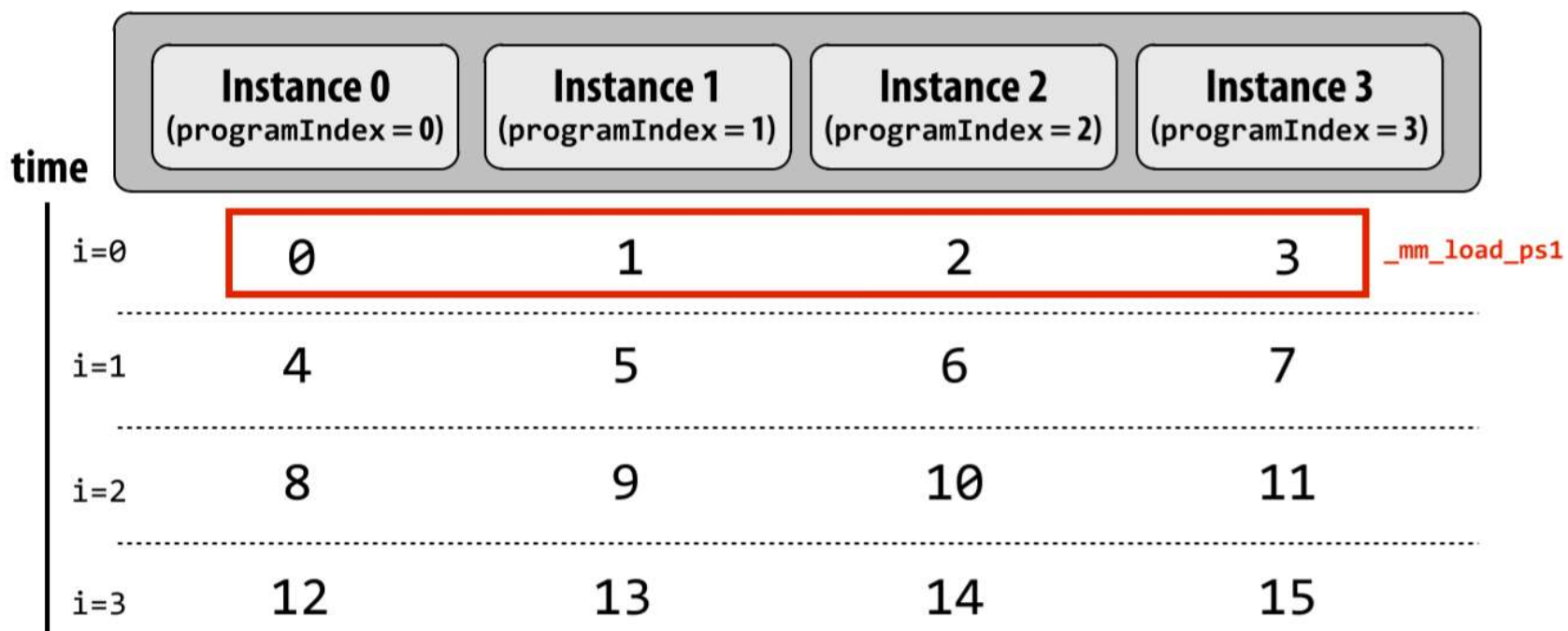
“Gang” of ISPC program instances

In this illustration: gang contains four instances: **programCount = 4**  
并发执行 $\sin(x(i))$ 程序实体的个数是4个

# Schedule: 交错分配

## “Gang” of ISPC program instances

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

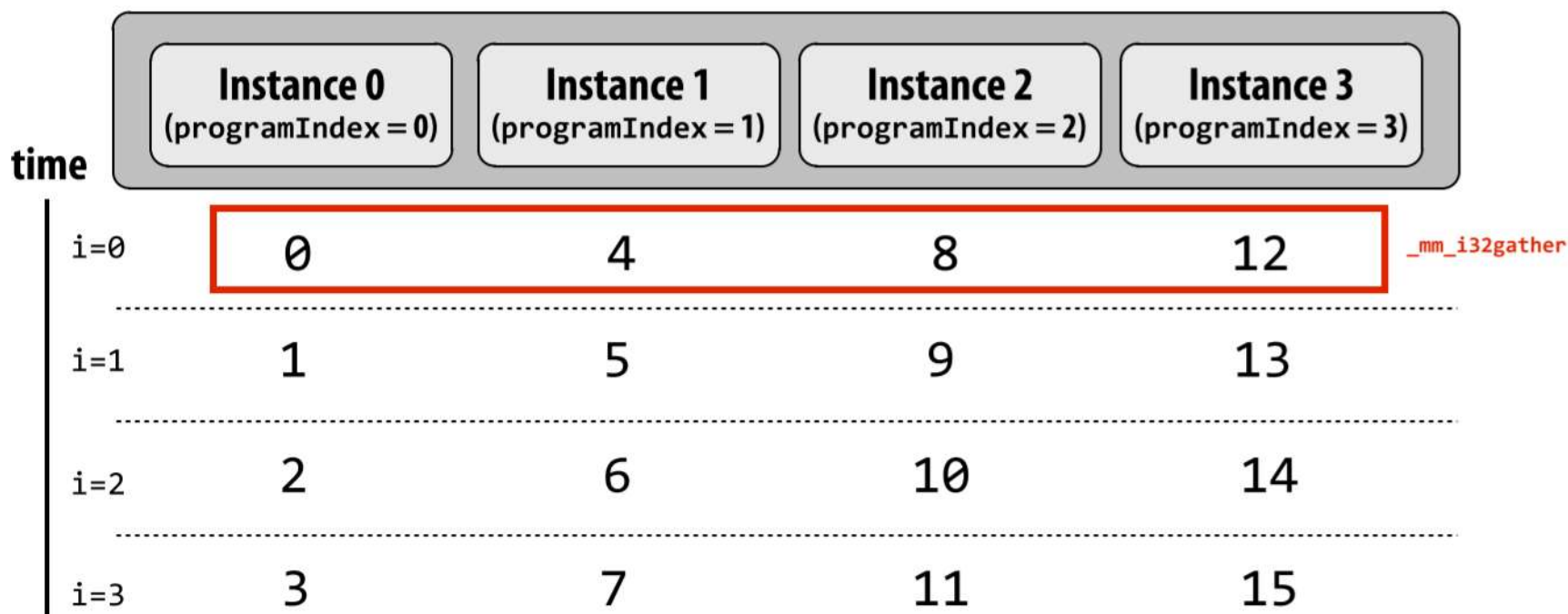


单个“打包加载” ( “**packed load**” ) SSE 指令 (`_mm_load_ps1`) 高效实现:  
`float value = x[idx];`  
对于所有程序实例, 因为这四个值在内存中是连续的

# Schedule: 阻塞分配

## “Gang” of ISPC program instances

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



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

现在触及内存中的四个非连续值。

需要“收集”（“gather”）指令才能实施，实现效率会显著降低

（gather 是一个更复杂、成本更高的 SIMD 指令：自 2013 年后才成为 AVX2 的一部分）

# 使用 foreach 提高抽象级别

## ISPC code: main.ispc

```
export void sinx(  
    uniform int N,  
    uniform int term,  
    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[i] = value;  
    }  
}
```

# 使用 foreach 提高抽象级别

---

## foreach: 关键 ISPC 语言结构

### ■ 用来声明并行循环迭代

程序员说：这些是程序组（gang）中的程序实例执行迭代

### ■ ISPC 的实现（implementation） 将各个迭代分别分配给 程序组（gang）中的各个程序实例

抽象（abstraction）本身允许不同的分配方式，但具体实现（implementation）可以只执行交错分配或者阻塞分配中的一种



# ISPC: abstraction vs. implementation

---

## Single program, multiple data (SPMD) : 编程模型 (抽象)

- 程序员“认为”：运行一个程序组，包含 programCount 个逻辑指令流（每个流都有不同的 programIndex 值）
- 这是编程抽象
- 程序是根据这种抽象来编写的（程序员的工作范围）

## Single instruction, multiple data (SIMD) : 实现

- ISPC 编译器发出矢量指令（SSE4 或 AVX），执行由 ISPC 程序组内的程序逻辑
- ISPC 编译器处理条件控制流到向量指令的映射（通过屏蔽向量通道等方式完成）

**抽象：主要面向开发人员，设法隐藏硬件细节，提高编程效率**

**实现：主要面向硬件，通过高效的硬件还原抽象，提高执行效率**

# ISPC tasks

---

- ISPC 程序组的抽象是通过多条 SIMD 指令在一个计算核心上实现完成的。
- 所以.....前面幻灯片中显示的所有代码只会在处理器的一个计算核心（core）上执行。
- ISPC 包含另一个抽象——任务（“task”）：用于完成对多个计算核心的程序执行（multi-core execution）。

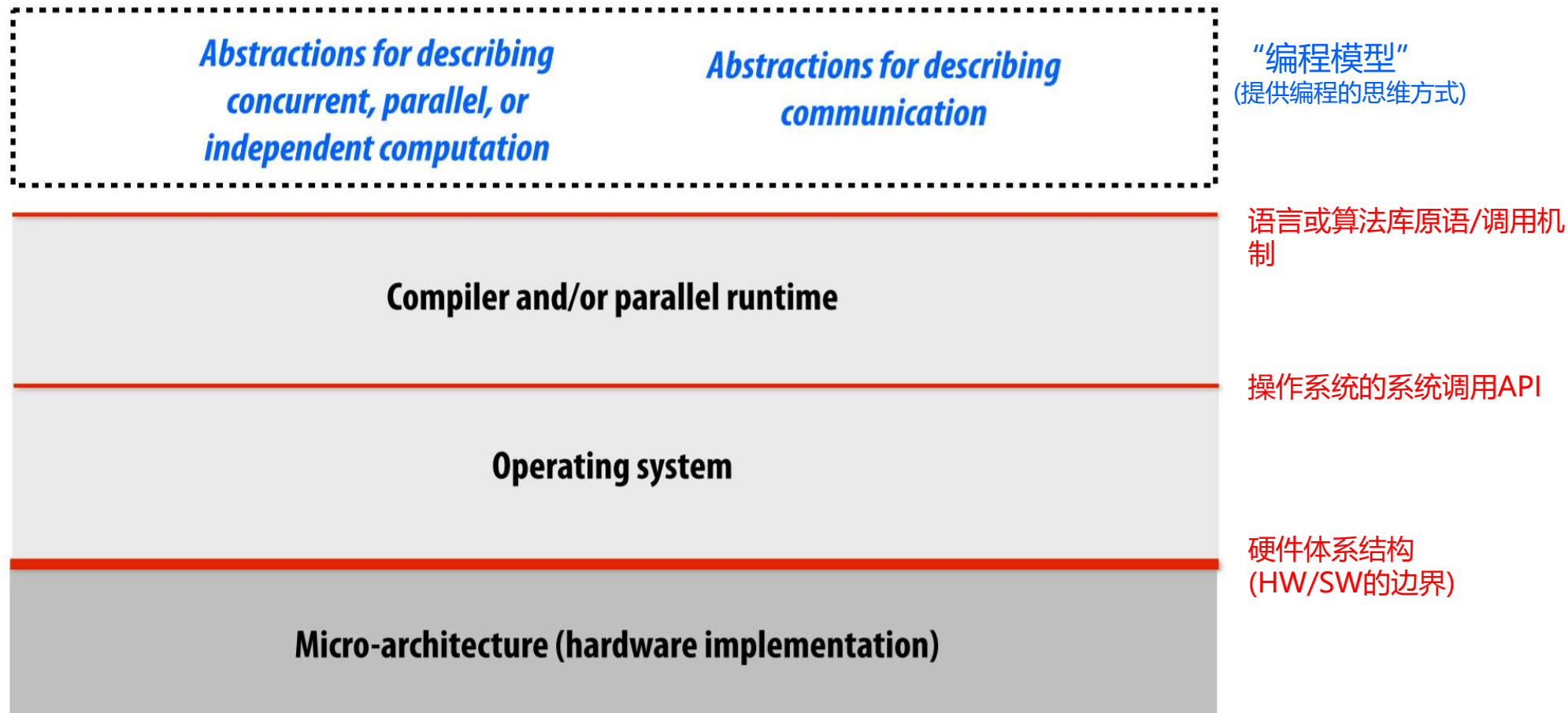
# 这节课的后半部分

---

- **从通信（communication）的角度来看三种并行编程模型**
  - 共享地址空间
  - 消息传递
  - 数据并行
- 呈现给程序员的通信抽象不同
- 编程模型影响程序员在编写程序时的思维方式

# 系统层：接口，实现，接口，...

## 并行应用

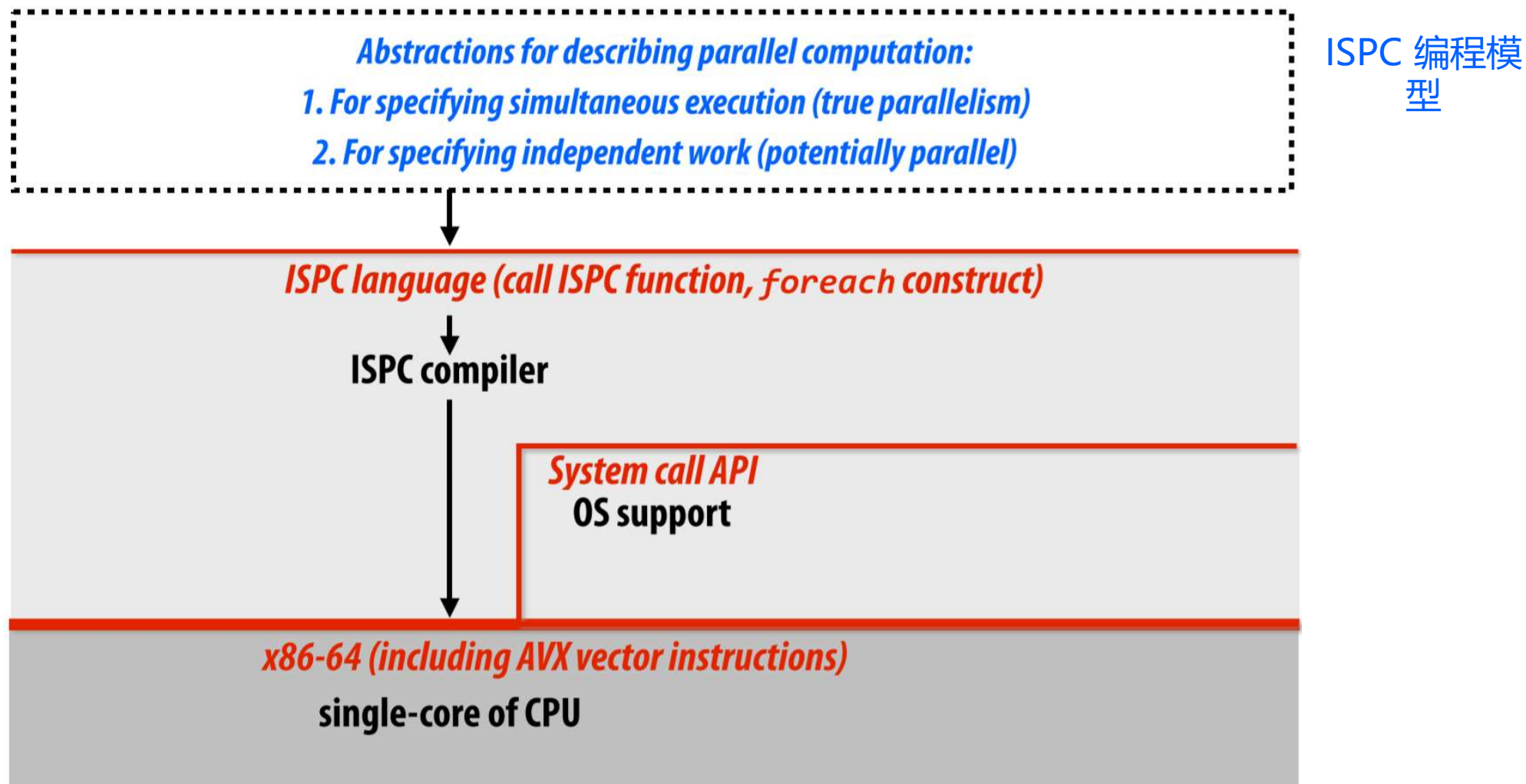


蓝色文字：抽象/概念

红色文字：系统的接口

黑色文字：系统的实现

# 示例：ISPC的抽象与实现



注意：此图特定于 ISPC 程序组的抽象。ISPC 还具有用于多核执行的“任务 (task)”编程原语。

# 三种通信模式

---

(抽象)

1. 共享地址空间(Shared address space)
2. 消息传递(Message passing)
3. 数据并行(Data parallel)

# 共享地址空间模型（抽象）

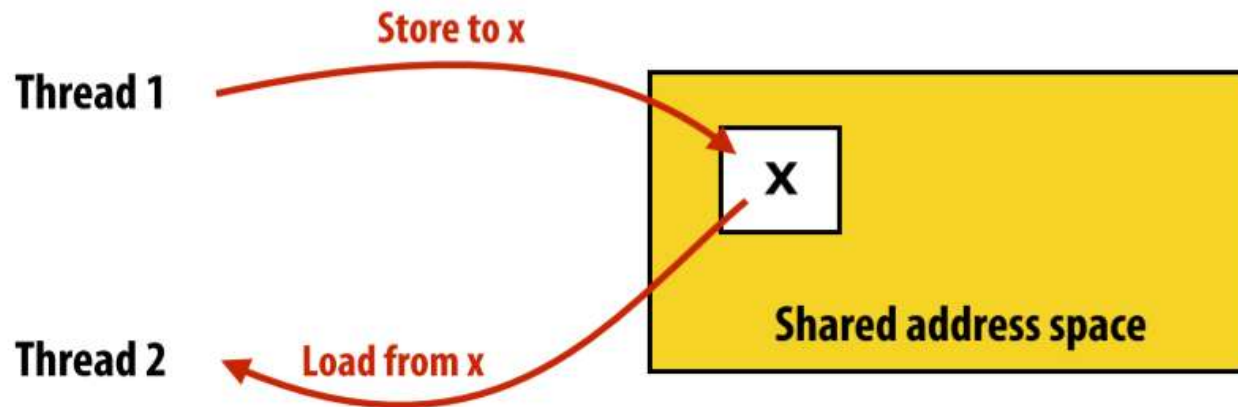
- 线程通过读/写共享变量进行通信
- 共享变量是线程均可见的
  - 任何线程都可以读取或写入共享变量

## Thread 1

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

## Thread 2

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



(Communication operations shown in red)

# 共享地址空间模型 (抽象)

同步原语也是**共享变量(shared variables)**: 例如, **锁(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;
}
```



# 共享地址空间模型（抽象）

---

## ■ 线程通过以下方式通信：

- 读/写共享变量
  - 线程间通信隐含在内存操作中
  - 线程 1 存储到 变量X中
  - 随后，线程 2 读取 变量X（并观察线程 1 对值的更新）
- 处理同步原语
  - 例如，通过使用锁确保互斥

## ■ 这是串行编程（sequential programming）的自然延伸

- 到目前为止，我们在课堂上的所有讨论都采用共享地址空间！

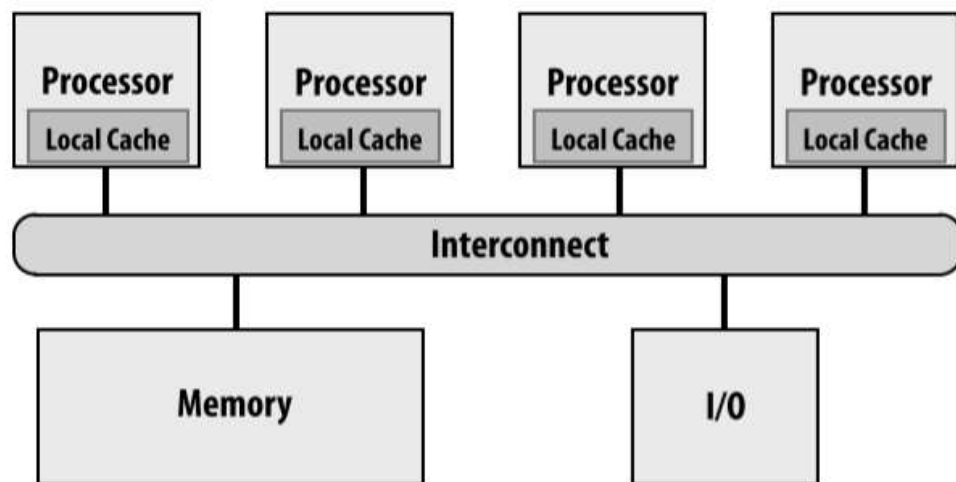
## ■ 简单的类比：共享变量就像一个大布告栏

- 任何线程都可以读取或写入共享变量

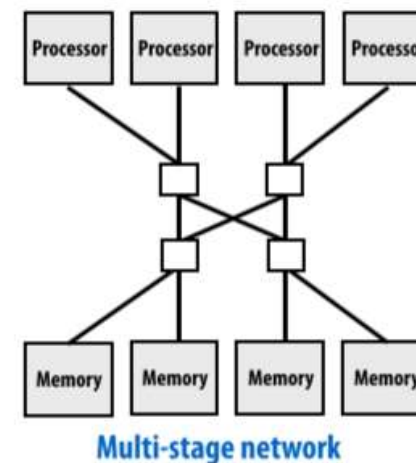
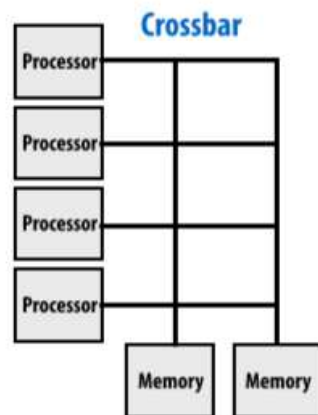
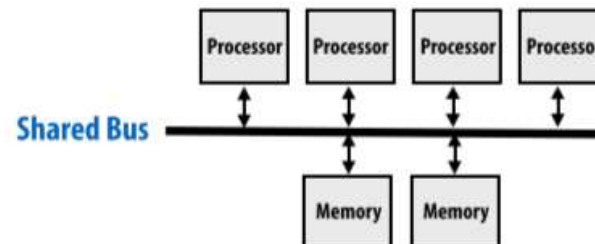
# 共享地址空间的硬件实现(HW implementation)

Key idea: 任何处理器都可以**直接访问**任何内存位置

“Dance-hall” organization



Interconnect examples



对称（共享内存）多处理器

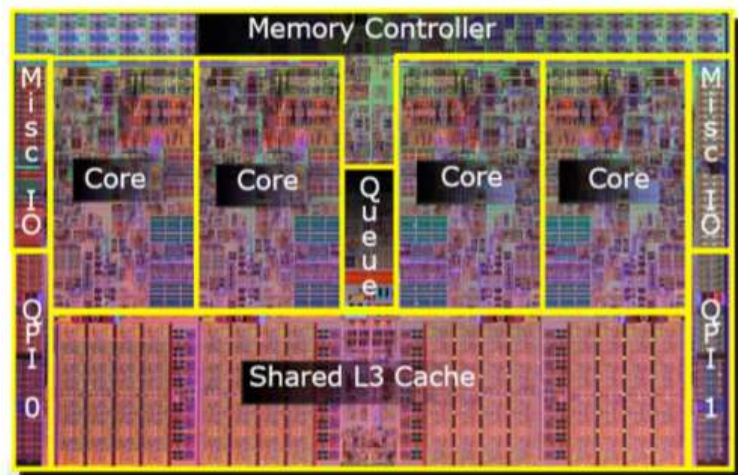
Symmetric (shared-memory) multi-processor (**SMP**):

- **内存访问的时间一致性**: 访问未缓存 (uncached) 内存地址的成本对所有处理器都是相同的

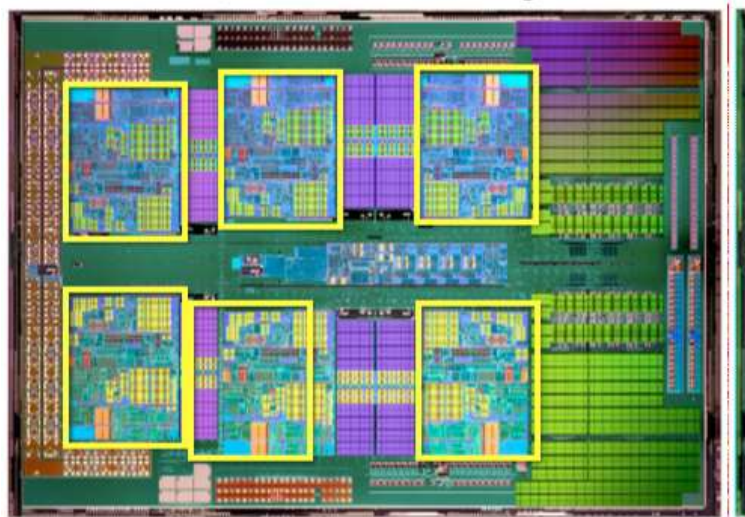
\*缓存(Caching)引入了非统一的访问时间，但我们后续会讨论

# 共享地址空间的硬件体系结构(NoC)

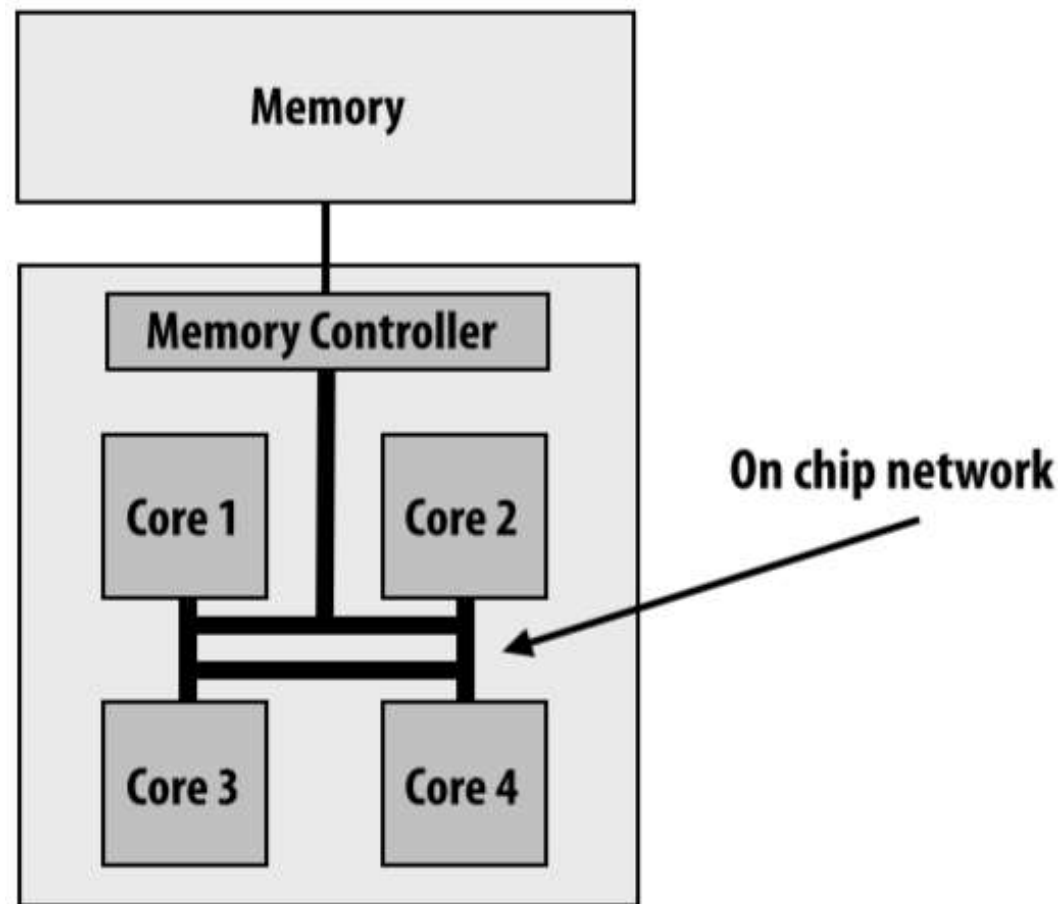
商用x86处理作为示例:NoC (network on chip)



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



AMD Phenom II (six core)

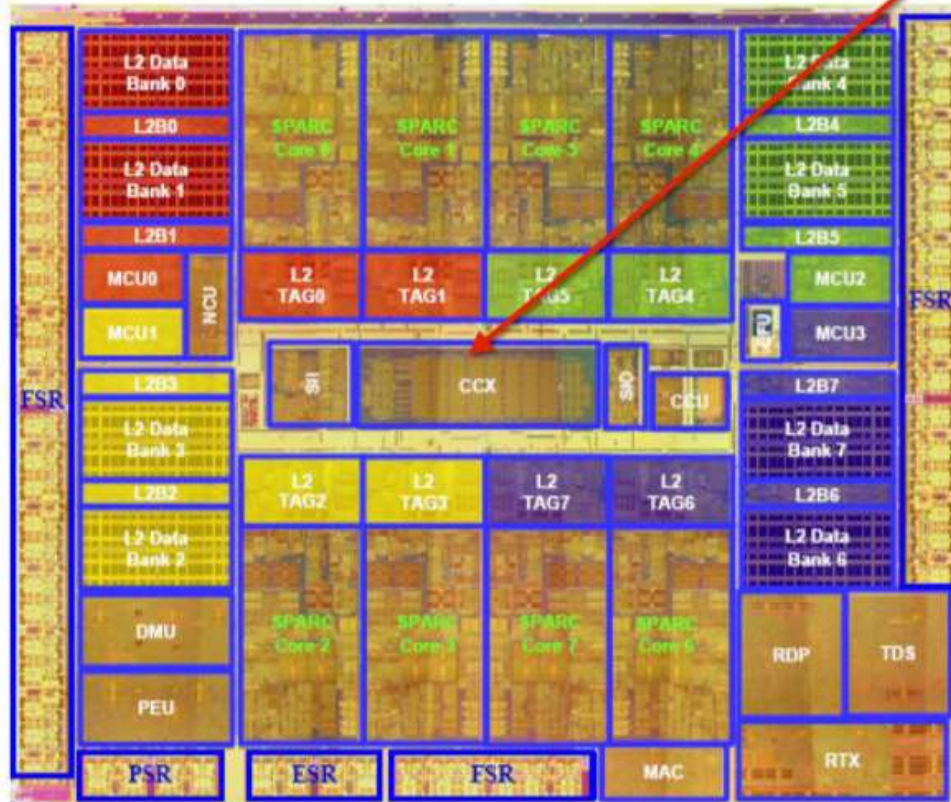


# 共享地址空间的硬件体系结构(Crossbar)

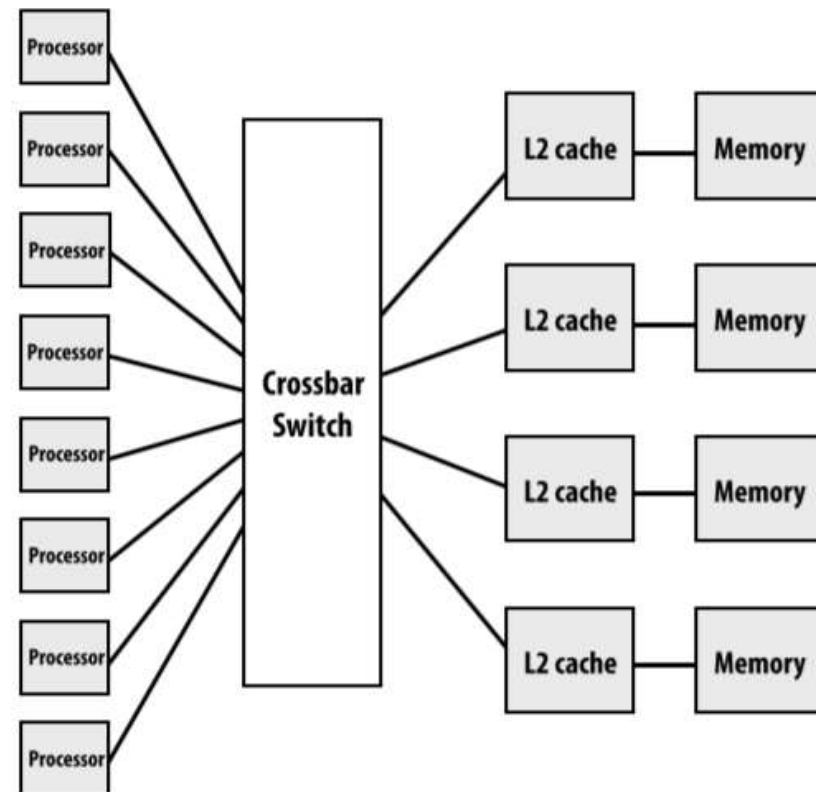
SUN Niagara 2 (UltraSPARC T2)

交叉开关: Crossbar

Note area of crossbar: about die area of one core



Eight cores



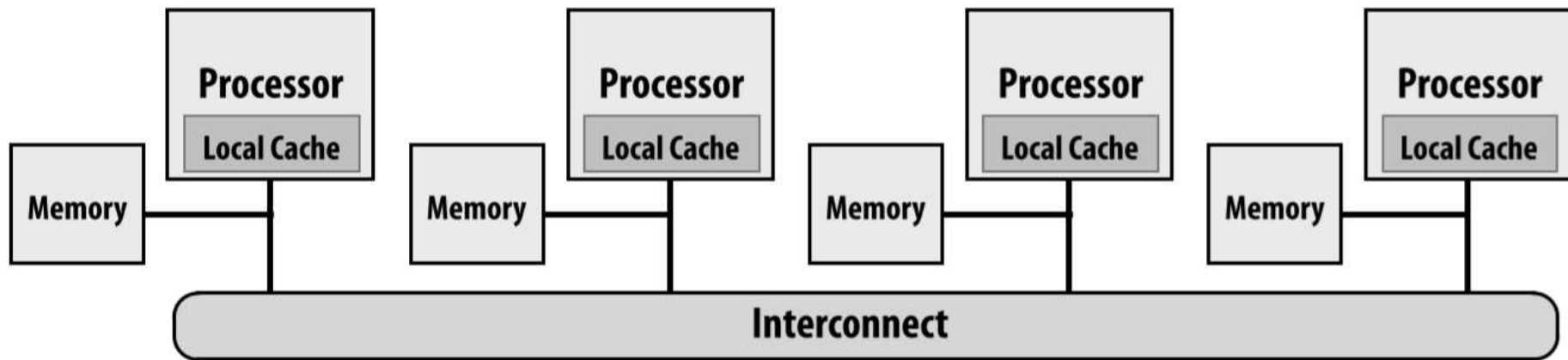


# 非统一内存访问的体系结构(NUMA)

非统一内存访问:

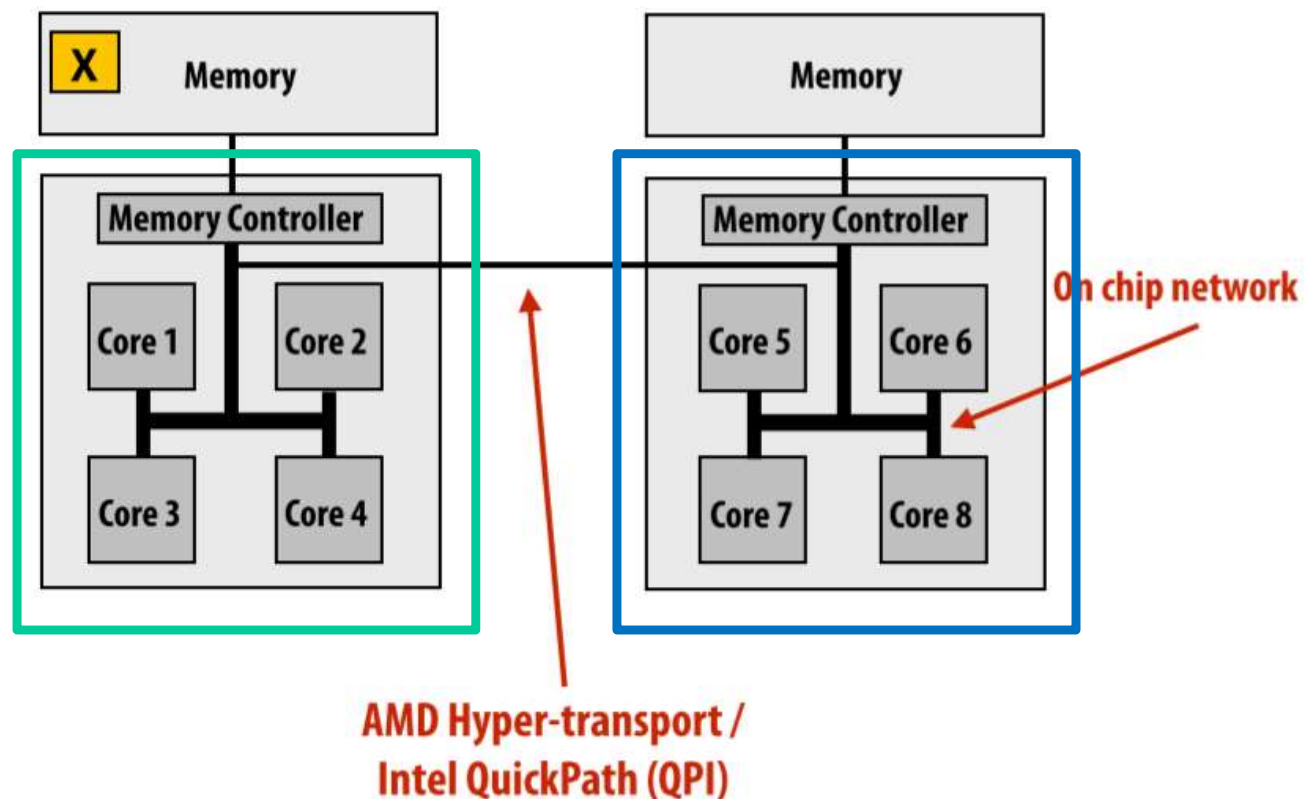
Non-uniform memory access (NUMA)

所有处理器都可以访问任何内存位置，但是.....内存访问的成本（**延迟和/或带宽**）对于不同的处理器是不同的



# 非统一内存访问的体系结构(NUMA)

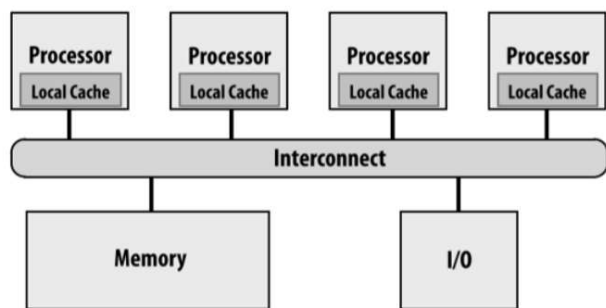
Example: 访问地址 **X** 的访存延迟性能方面，计算核心 **5-8** 比计算核心**1-4**的访存延迟明显要高



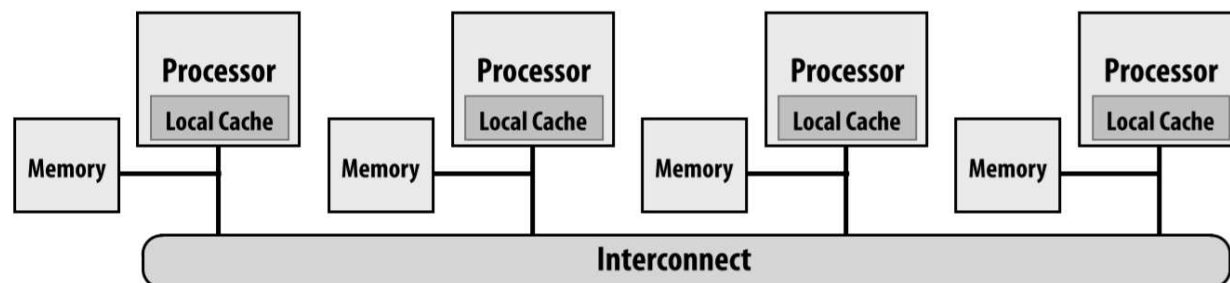
Modern dual-socket configuration

# SMP vs NUMA

- **SMP在系统中保持统一访问时间：**
  - **GOOD:** 成本是一样的
  - **BAD:** 访存开销一样都很高（内存一样远）、扩展性问题
- **NUMA 设计更具可扩展性(*scalability*)**
  - 本地内存 (local memory) 的低延迟和高带宽
- **NUMA系统的成本是增加了程序员的性能优化工作**
  - 发现、利用局部性(locality)对性能很重要
  - 希望大多数内存访问是本地内存 (local memory)



**SMP:对称（共享内存）多处理器体系结构**



**NUMA:非统一内存访问体系结构**

# 总结：共享地址空间模型

---

## ■ 通信抽象

- 线程读/写共享变量
- 线程操作同步原语：锁、信号量等。
- 对单处理器程序设计的逻辑扩展以支持多处理器的程序设计\*

## ■ 需要硬件支持才能高效实现

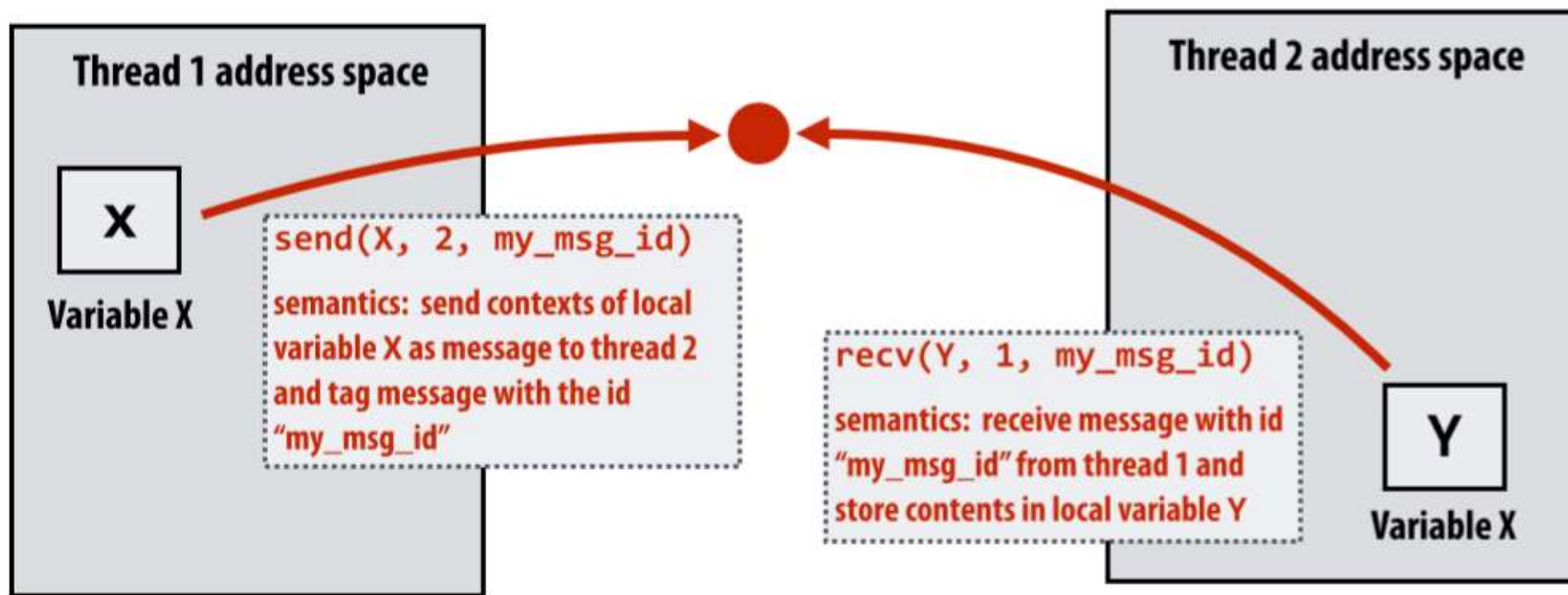
- 任何处理器都可以从任何地址加载和存储  
(具有共享地址空间！)
- 即使使用 NUMA，扩展成本也很高  
(超级计算机价格昂贵的原因之一)

\* 但是 NUMA 实现需要对局部性进行优化以提高性能



# 消息传递模型（抽象）

线程在自己的私有地址空间内运行 (private address spaces)



(Communication operations shown in red)

# 消息传递模型（抽象）

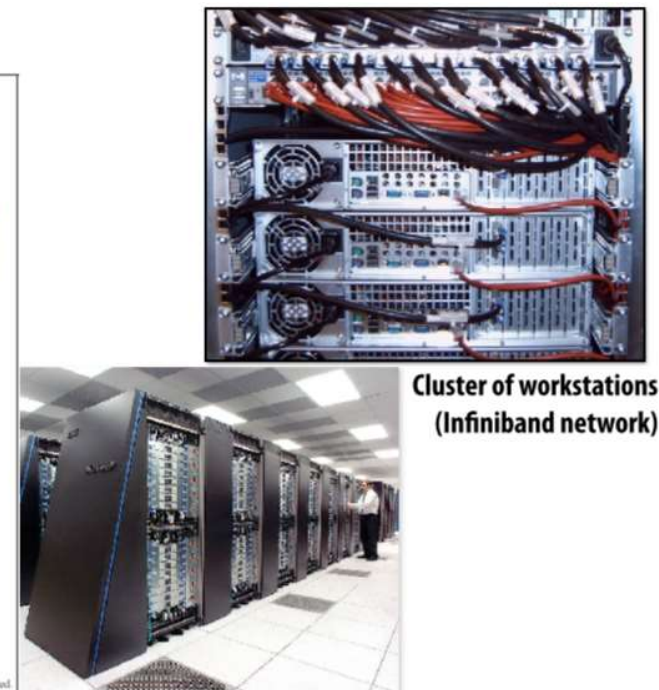
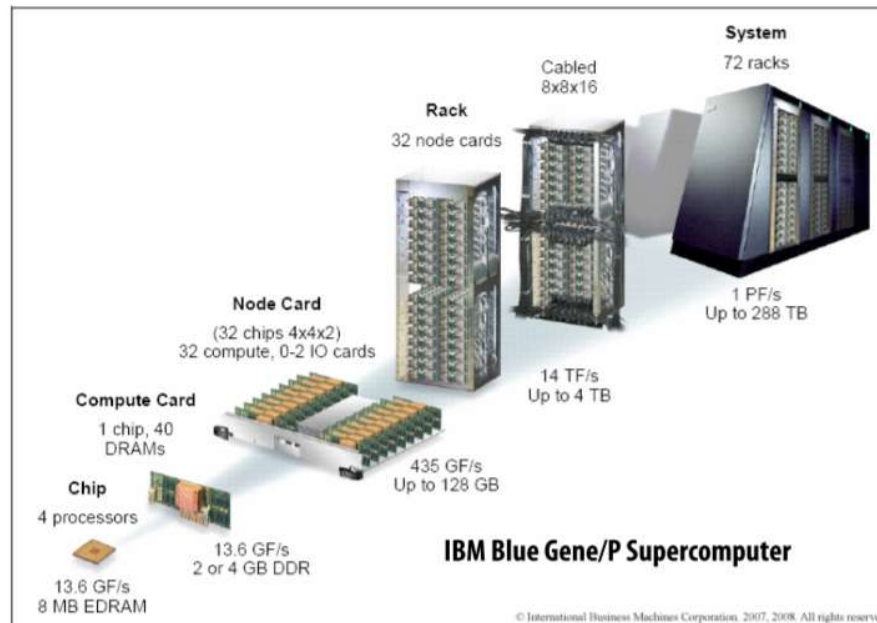
---

## 线程通过**发送/接收消息**（**sending/receiving messages**） 进行通信

- 发送者的消息：指定接收者、要传输的缓冲区和可选的消息标识符（“标签tag”）
- 接收者的消息：发送者、指定存放数据的缓冲区、以及可选的消息标识符
- 发送消息是线程之间交换数据的唯一方式

# 消息传递（实现）

- 流行的通信库软件：**MPI** (message passing interface)通信库
- 硬件不需要实现全系统范围内的loads and stores来执行消息传递程序(只需要实现传递消息过程即可)
  - 可以将多台商业服务器进行互连，形成大型计算系统——超级计算机（例如：机群clusters）
  - 消息传递是机群的一种主要编程模型



# 编程模型与机器类型

---

**注意：编程模型和机器类型之间的对应关系其实是模糊的，不确定的**

- **硬件可以通过共享地址空间实现消息传递抽象**
  - “发送消息” = 发送方将消息缓冲区数据，写入共享内存
  - “接收消息” = 接收方从共享内存复制数据，写入消息库缓冲区
- **也可以在不支持共享内存的硬件上实现共享通信(通过效率较低的软件解决方案)**
  - 将所有包含共享变量的页面标记为无效
  - 页面错误处理程序发出适当的网络请求

# 数据并行编程模型

---

回顾前面的内容：编程模型将“结构化”赋予给程序  
程序的结构化高低：取决于是否能够从程序代码看到影响性能的因素。可以理解为程序员对代码的掌控力。

- **共享地址空间 (Shared address space) : 结构化程度低**
  - 所有线程都可以读写所有共享变量
  - 问题：由于实现的差异，并非所有读取和写入都具有相同的访存延迟开销  
(并且访存延迟开销在程序代码中无法体现)
- **消息传递 (Message passing) : 高度结构化的通信**
  - 所有通信都以消息的形式发生
  - 可以读取程序并查看通信的位置
- **数据并行 (Data-parallel) : 非常严格的计算结构**
  - 程序对集合中的不同数据元素执行相同的计算程序

# 数据并行编程模型

---

## ■ 过去: 对数组的每个元素执行相同的操作

- 在80年代, 用于计算功能匹配类应用的 SIMD 超级计算机
- Thinking推出的连接机Connection Machine (CM-1, CM-2): 一个指令译码, 调用数千个处理器
- Cray 超级计算机: 向量处理器
  - $\text{add}(A, B, n) \leftarrow$  一条指令对 A和 B的两个向量的每个元素进行对位求和, 每个向量长度为8

## ■ 今天: 经常采用 SPMD 编程的形式

- **map(function, collection)**
- **function**函数程序**独立地**处理于**collection**集合中的每个元素
- **function**函数程序可能是一个复杂的逻辑序列 (例如, 循环体)
- 同步 (Synchronization) 在**map**的末尾是隐式执行的, 无需程序员在程序代码中编写同步的编程原语 (当函数完成集合的所有元素的计算后, 自动执行同步, 并返回结果)

# ISPC中的数据并行编程模型

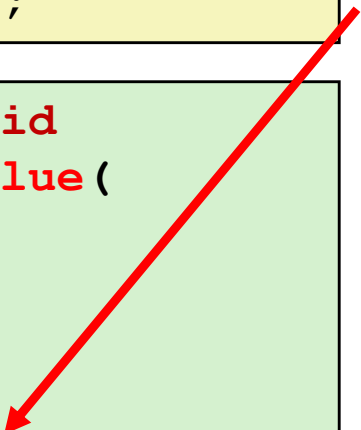
```
// main C++ code:
const int N = 1024;
float* x = new float[N];
float* y = new float[N];
// initialize N elements of x here

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

将循环体 (**loop body**) 视为函数  
(**function**)

**foreach**构造过程就是 **map**映射

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



鉴于此程序，可以很容易地将各个程序要执行的函数程序，映射到数组 X 和 Y 的各个元素上。



# 数据并行编程模型：流编程

```
const int N = 1024;
stream<float> x(N); // define collection
stream<float> y(N); // define collection

// initialize N elements of x here

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

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

以这种形式表示的数据并行，通常被称为**流编程模型 (stream programming model)**

**Streams:** 元素归集. 各个元素可以被独立地处理

**Kernels:** 无副作用的函数 (side-effect-free functions).  
对集合进行逐元素操作

将每次调用的**Kernels**的输入、输出和临时地址视为私有地址空间

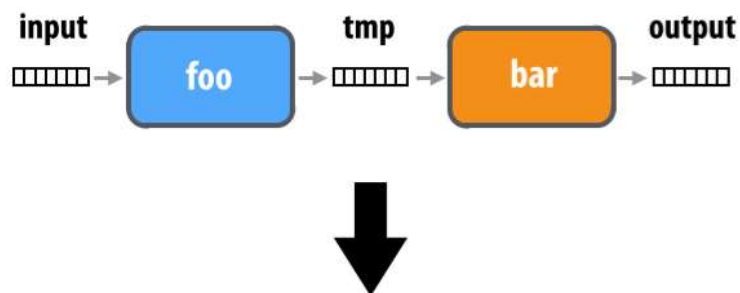
注意：这不是 ISPC 标准语法（可以看成是编造的语法）



# 流编程模型的好处

```
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]));  
}
```

## Functions 没有副作用！（不适合非确定性程序）

- 编译器能够明确程序的数据流
- 每个调用的输入和输出都是预先知道的：可以使用预取来隐藏延迟。
- 生产者-消费者位置是事先已知的：实现可以被结构化，因此第一个内核的输出立即由第二个内核处理。（这些值存储在片上缓冲区/缓存中，永远不会写入内存！节省带宽！）
- 这些优化是流程序编译器的自动实现的。需要对整个程序进行全局程序分析。

# 收集 (Gather) /分散 (Scatter) 通信原语

将 absolute\_value 映射到 **gather** 生成的流上

```
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;
    }
}
```

将 absolute\_value 映射到 **scatter** 生成的流上

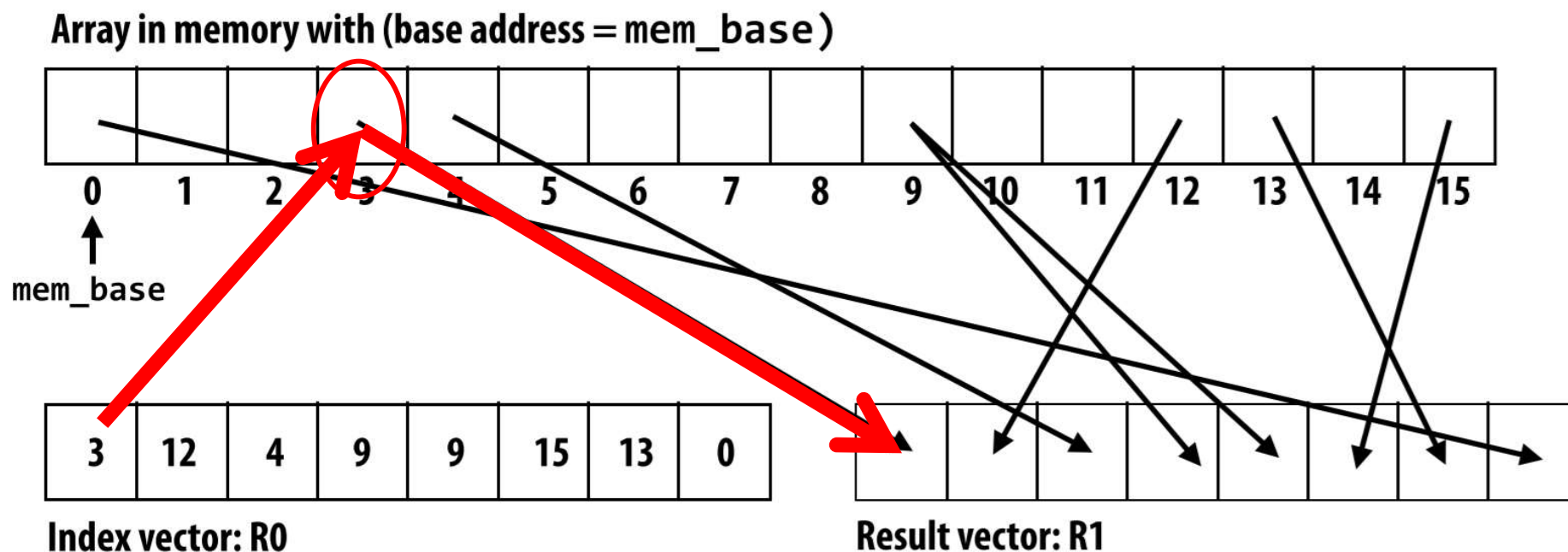
```
const int N = 1024;
stream<float> input(N);
stream<int> indices;
stream<float> tmp_output(N);
stream<float> output(N);

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

```
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 指令（非连续访存寻址指令）

**gather**(R1, R0, mem\_base); “根据 R0 指定的索引值，从缓冲区 mem\_base 中收集对应的数据，并收集到 R1 向量的对应元素中。”



Gather 在 2013 年支持 AVX2指令  
但 AVX2 不支持 SIMD scatter（必须作为标量循环实现）  
AVX512指令集中有专门的scatter 分散指令

现代GPU 上通常也存在硬件支持的**gather/scatter**。  
(但与连续向量的加载/存储相比仍然是一项非常昂贵的操作)

# 总结：数据并行模型

---

- 数据并行是通过**强加程序的结构化设计**，来简化编程和优化性能
- 基本结构化做法是：将一个函数映射到一个大的数据集上
  - Functional:无副作用执行 side-effect free execution
  - 不同函数调用之间没有通信  
(每个函数允许以任何顺序被调度，包括并行)
- 实际上，这就是许多简单程序的工作原理
- 但是.....许多现代面向性能的数据并行语言**并没有严格执行**这种结构化
  - ISPC, OpenCL, CUDA, etc.
  - 他们倾向于选择更加灵活/程序员熟悉的命令式 C语言风格的语法，而不是使用过于结构化的形式

# 三种并行编程模型的总结

---

## ■ 共享地址空间 (Shared address space)

- 通信是非结构化的，隐含在 loads and stores之中
- 十分自然的编程方式，但很容易搬起石头砸自己的脚
  - 程序容易实现正确，但可能性能不佳

## ■ 消息传递 (Message passing)

- 将所有通信构造为消息 (messages)
- 通常比共享地址空间更难实现程序的正确性
- 结构化通常有助于获得一个正确的、可扩展的程序

## ■ 数据并行 (Data parallel)

- 将整个计算任务视为一个任务集合， a big “map”
- 假定满足共享地址空间，包含load inputs/store results，但要严格限制了各个迭代计算之间的通信量（尽量不通信）  
(目标：保留每轮迭代计算都能独立执行)
- 现代实际实施方案中对此都是鼓励，但通常不强制执行这种结构

# 现代实践：混合编程模型

---

- 在集群的多核节点内使用共享地址空间 (shared address space) 编程；而在节点间使用的消息传递 (message passing)
  - 在实践中非常非常普遍
  - 使用共享地址空间的便利性，可以有效地（在节点内）实现多核并行程序
  - 而在节点间的并行，需要显式地实现通信
  
- 数据并行编程模型支持，在一个核函数内的共享内存式同步原语
  - 允许有限形式的迭代间通信  
(e.g., CUDA, OpenCL)
  
- 在未来的课程中，我们会看到CUDA/OpenCL 这种使用数据并行模型扩展到多个计算核心的并行编程，同时也会通过采用共享地址空间，来允许运行在同一计算核心上的不同线程间进行通信。

# 总结

---

- 编程模型提供了一种思考并行程序组织的方法。 它们通过提供了**抽象 (abstractions)** 来满足不同的**实现 (implementations)** 。
- 这些抽象所施加的限制旨在反映并行和通信的延迟开销成本的现实情况
  - 共享地址空间的计算系统 (Shared address space machines) : 硬件支持任何处理器访问任何地址
  - 消息传递的计算系统 (Messaging passing machines) : 通过硬件实现来加速消息发送send/接收receive/缓冲buffering
  - Tradeoff 1: 低层次的抽象有益于保证程序代码性能的可预测性
  - Tradeoff 2: 高层次的抽象有益于保证代码实现的灵活性/可移植性
- 在实践中, **应该以多种编程模型和多个层次来思考**
  - 现代计算系统, 针对以不同的计算系统规模, 提供不同类型的通信模式
  - 不同类型的通信模式, 适合于不同规模下的不同类型计算系统
  - 性能优化不能单纯通过编程模型的抽象来保障, 更需要通过独特的底层实现来达到