并行计算
Parallel Computing

主讲 孙经纬

2024年 春季学期

概要

- 第三篇 并行编程
  - 第十三章 并行程序设计基础
  - 第十四章 共享存储系统并行编程
  - 第十五章 分布存储系统并行编程
  - 补充章节1 GPU并行编程
  - **补充章节2 关于并行编程的更多话题**

关于并行编程的更多话题

- 数据依赖与自动并行
- MapReduce
- 并行程序性能优化
- 案例：图与矩阵算法的GPU优化

并行程序性能优化

- **任务分配与调度**
- 锁
- 一致性

本章节综合参考了教材第13.5章节，Stanford CS419，以及《并行算法设计与性能优化》（刘文志）

线程的开销

线程的开销主要来自两个方面

1. 线程的创建和终止
   - 几微秒到几毫秒之间，不同条件下有所不同

```
1  for (int iter = 0; iter < num; iter++)
2  #pragma omp parallel
3  {
4      doSomething();
5  }
6
7
8  // 优化后
9  #pragma omp parallel
10 for (int iter = 0; iter < num; iter++)
11 {
12     doSomething();
13     #pragma omp barrier
14 }
```

代码实现影响线程创建和终止的频率

线程的开销

2. 线程调度引起的额外开销
   - 线程切换（时间片轮转）
   - 并行（parallel）与并发（concurrent）

- 线程切换的主要开销来源
  - 寄存器切换
  - Cache切换
  - 内存切换

线程的开销

- 寄存器切换
  - 每个线程都有自己的寄存器状态，包括通用寄存器、程序计数器、堆栈指针等。
  - 在进行线程切换时，当前线程的寄存器状态需要保存到它的线程控制块（TCB）中，以便线程再次被调度时可以恢复这些状态。切换到新线程时，需要加载该线程之前保存的寄存器状态。
  - 由于寄存器的存取速度非常快，一般情况下恢复上下文的时间会比时间片小。

线程的开销

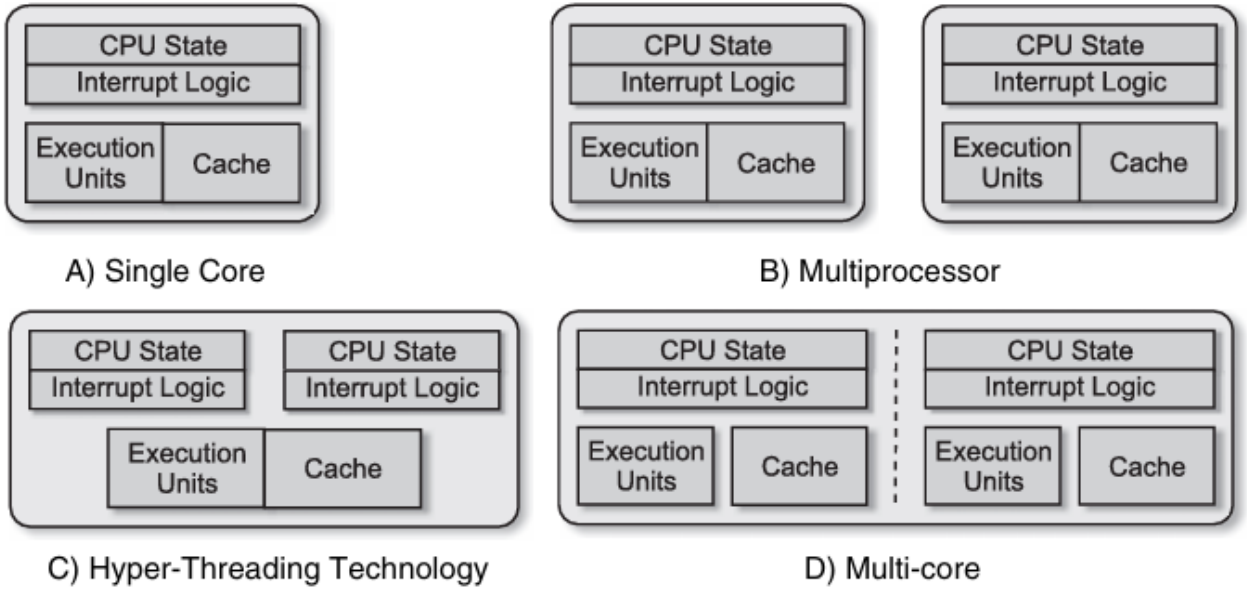
- Cache切换
  - 当某线程处于执行状态时，系统需要将它常用的数据缓存到Cache中，以提高读写数据的速度。但如果线程过多，Cache很容易被占满，这时需要淘汰Cache中最近使用最少的数据，而这些数据很可能就是其他线程的。
  - 线程切换可能导致新的线程使用不同的数据集，从而引发缓存失效，因为缓存中的数据属于之前的线程。这种缓存失效会导致新线程在开始执行时需要从较慢的主存中重新加载数据，增加了延迟。
  - 这样，线程调度的同时伴随着线程对Cache的争夺，导致Cache的内容频繁地被更新而损失性能。

线程的开销

- 内存切换
  - 与Cache切换开销相似的还有争夺内存的开销。多数系统使用虚拟存储器的机制来提高内存使用率，将频繁使用的数据保存在实际的内存中，而将其他数据存放在访问速度慢得多的磁盘虚拟空间上。
  - 与争夺Cache类似，如果线程过多，内存很容易被占满，当前正在执行的线程就需要淘汰内存中最近最少访间的数据，并将其所需的数据从磁盘虚拟空间装入到内存中，而这一过程需要很大的开销。

超线程

- 也称为“硬件线程”或Hyper-Threading（Intel）
- 允许单个物理CPU核心模拟多个逻辑核心



https://www.es.ele.tue.nl/~heco/courses/ACA/Multi-CoreProgramming-Ch1_Akhter_Roberts.pdf

超线程

有利的场景

- 混合类型的负载：

如果系统同时运行多种类型的应用程序，例如同时运行CPU密集型和I/O密集型任务，超线程可以帮助更平衡地分配资源，提高整体的系统响应性和吞吐量。

- 上下文切换频繁的环境：

在高负载的多任务环境中，超线程可以减少因进程或线程上下文切换而产生的开销。通过一个逻辑核心接管另一个逻辑核心的任务，可以减少等待时间和切换成本。

超线程

不利的场景

- CPU密集型任务：

对于严重依赖CPU计算并且已经高效利用CPU资源的应用程序（如某些类型的科学计算或图形处理），超线程可能不会带来明显的性能提升。这是因为所有的CPU资源（如算术逻辑单元ALU）已经被充分使用，额外的线程无法获得更多的CPU时间。

- 资源冲突：

当多个线程试图访问相同的资源（如缓存、内存带宽等）时，超线程可能导致性能下降。在这种情况下，线程之间的资源争夺可能会增加延迟和降低效率。

线程数的设置

- 线程调度开销的出现是因为有限的资源被太多的线程共享，同时线程调度使得程序的数据局部性只能保持很短的时间。
- 可运行线程是指处于就绪状态或执行状态的线程，系统对它的调度会引起额外的开销。若一个线程因等待诸如鼠标点击或磁盘I/O请求这样的外部事件时进入阻塞状态，则系统会将其从轮转调度表中删除，从而不会引起调度开销。
- 当系统中可运行线程的个数超过处理器数时，处理器的利用率不会明显地提高，同时系统又会因为调度线程而产生较大的额外开销。
- 因此应当限制可运行线程的个数。典型的设置是使其不超过处理器数。

线程数的设置

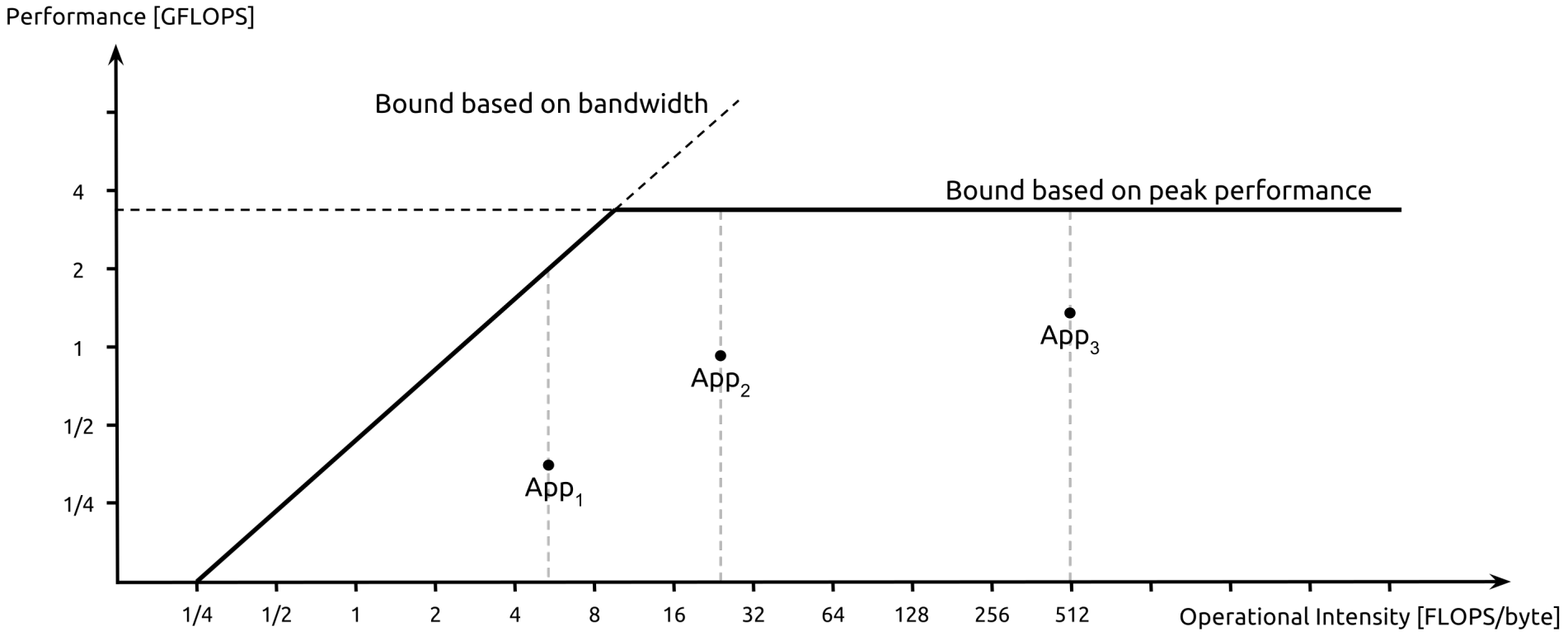
- 如果并行程序在执行过程中多数线程处于阻塞状态，则在该程序拥有的线程数量比处理器数多的情况下，它仍然能够高效地运行，即使不用超线程技术。
- 例如，在设计并行程序时，可将计算线程和I/O线程分开。计算线程用来执行并行程序中的计算任务，在多数时间内都是可运行线程。而I/O线程用来处理并行程序中各种I/O操作，在多数时间内都处于阻塞状态。这样，只需使计算线程的个数与处理器数目相匹配，即可有效避免调度线程产生的额外开销，最大限度地提升并行程序的性能。

性能瓶颈分析

- 上述讨论表明，当线程数超过处理器数时，可以尝试通过混合不同资源偏好的工作负载来提升性能。
- 如何判断工作负载是计算密集、访存密集、IO密集、还是通信密集？
- 更准确地来说，如何判断工作负载是计算受限、访存受限、IO受限、还是通信受限？
- “受限”（bound）意味着该资源已经满载，混入更多的使用该资源的工作负载无法进一步提升性能。

Roofline model

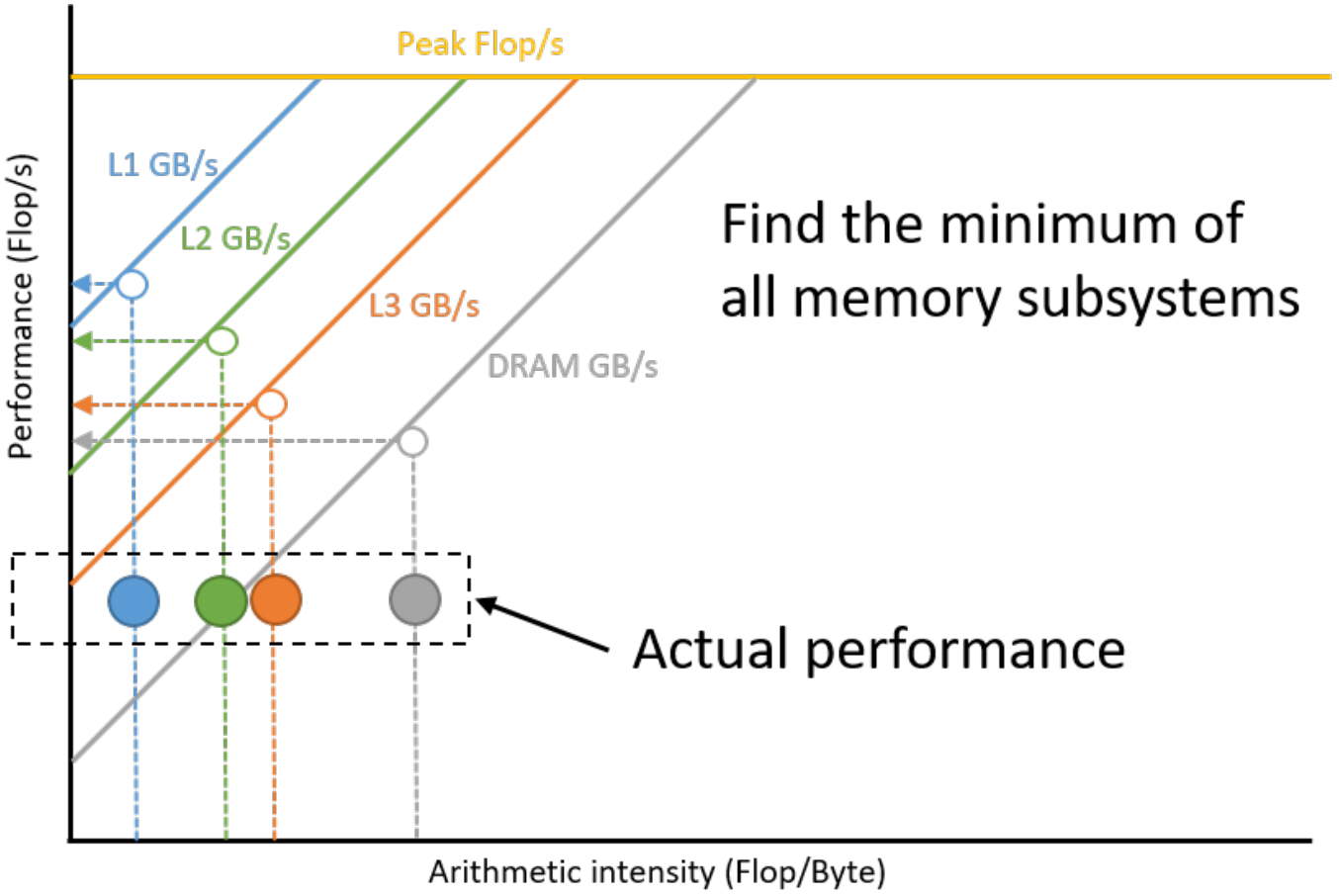
- 屋顶模型是一种简单的可视化性能瓶颈评估方法



示例：屋顶模型分析程序是浮点计算受限还是访存受限

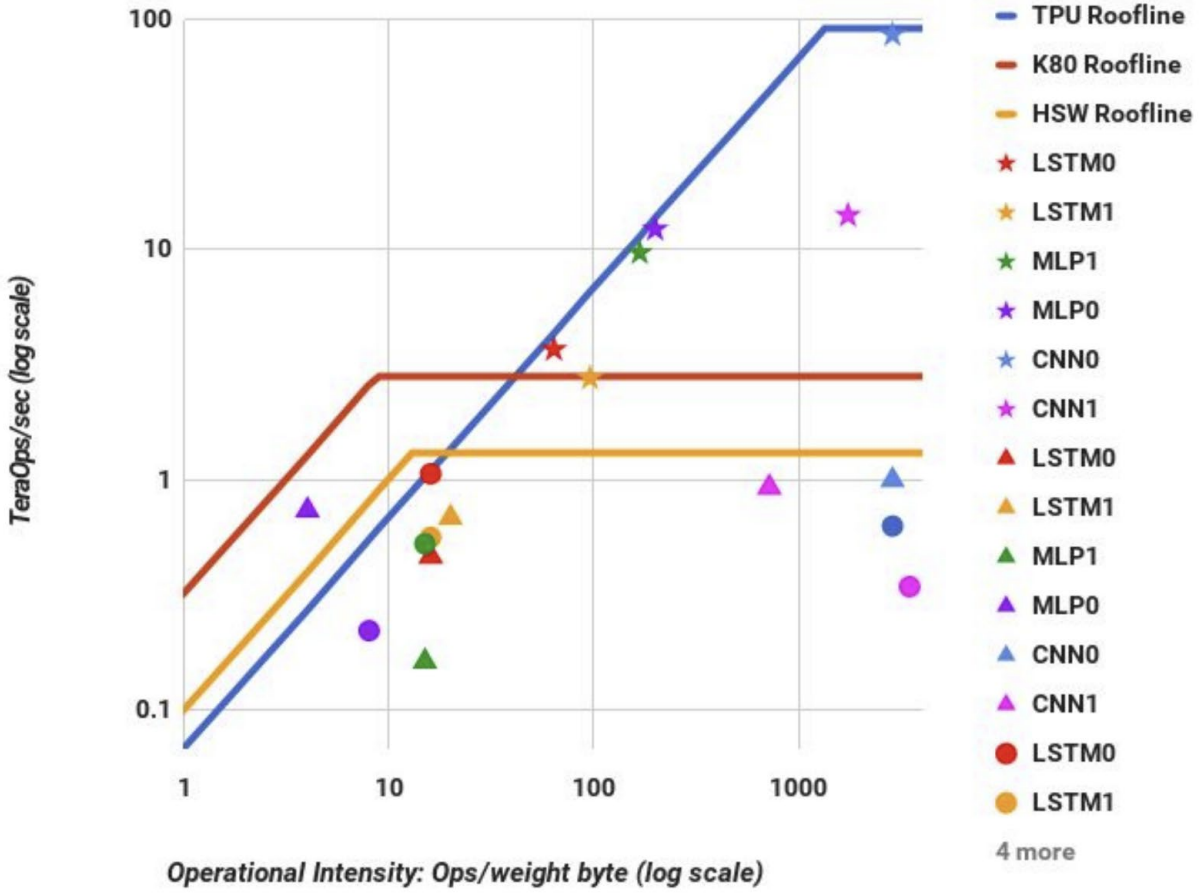
Roofline model

- 可以通过不同的屋顶，分析更具体的受限原因



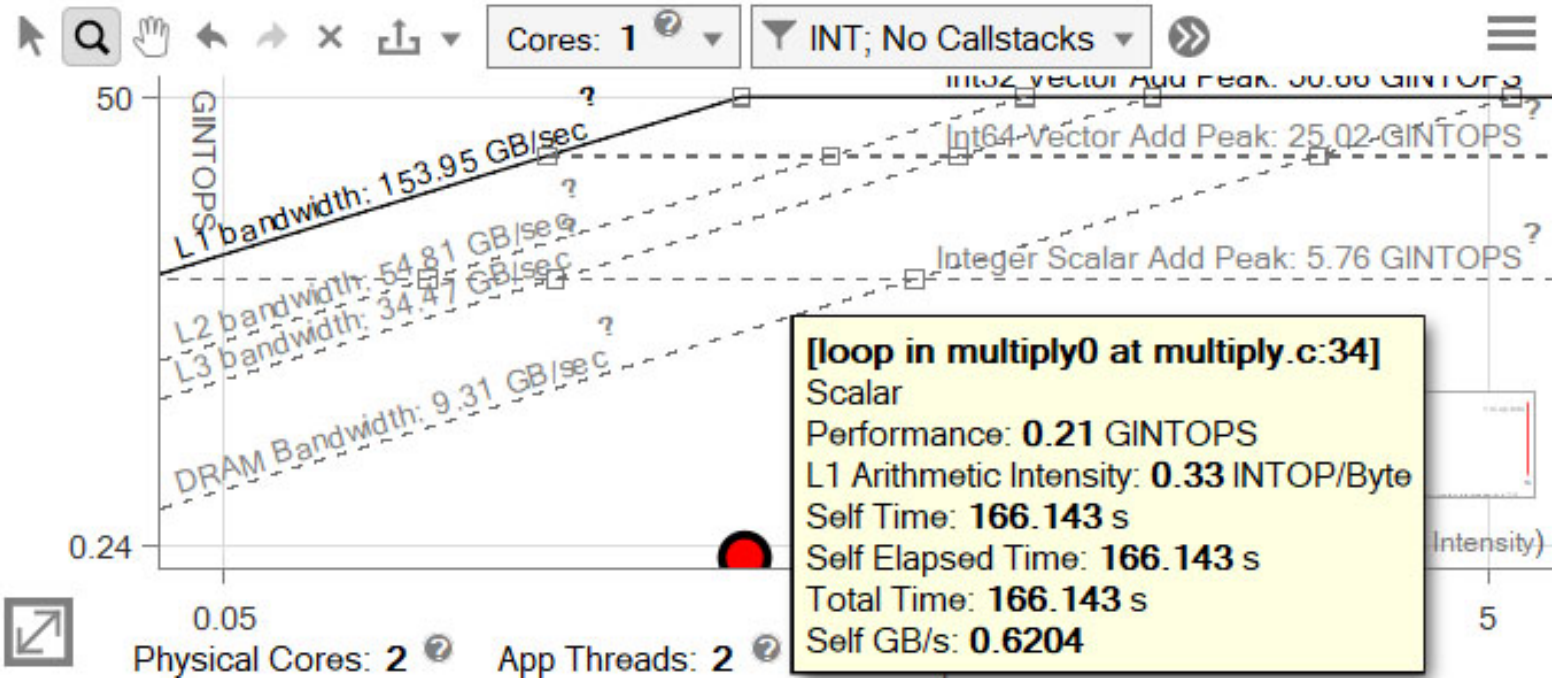
Roofline model

- 可以通过不同的屋顶，分析优化是否有用



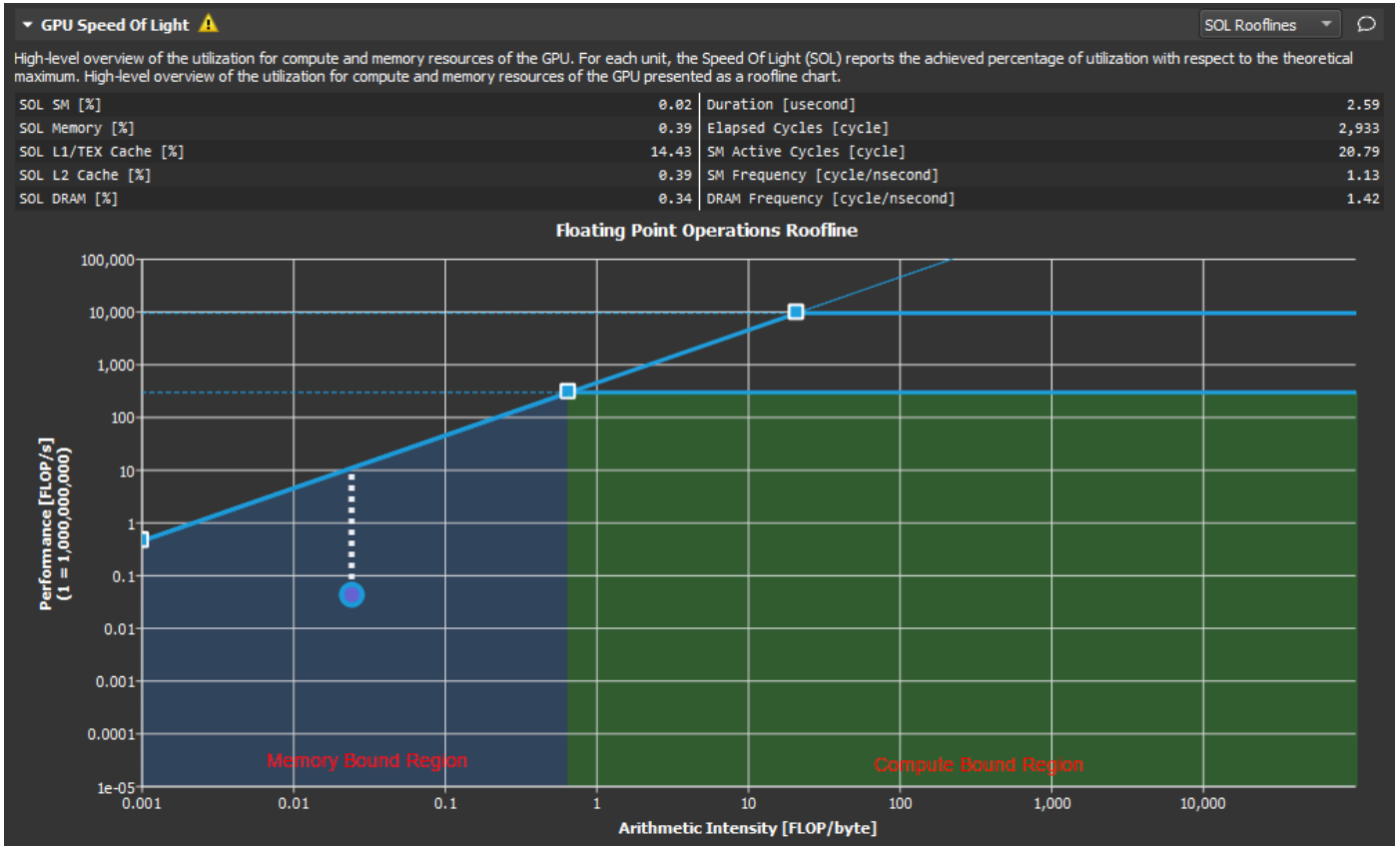
Roofline model

- 许多性能分析工具提供roofline model分析和可视化
  - Intel Advisor



Roofline model

- 许多性能分析工具提供roofline model分析和可视化
  - NVIDIA nsight compute



任务分配

- 为每个线程分配合理数量的工作负载
- 优化目标:
  - 负载均衡
  - 额外开销

- 静态分配
- 动态分配

（回看OpenMP相关内容）

任务分配

- 试除法判断质数的并行程序示例（动态分配）

Sequential program
(independent loop iterations)

```
int N = 1024;
int* x = new int[N];
bool* prime = new bool[N];

// assume elements of x initialized here

for (int i=0; i<N; i++)
{
    // unknown execution time
    is_prime[i] = test_primality(x[i]);
}
```

Parallel program
(SPMD execution by multiple threads)

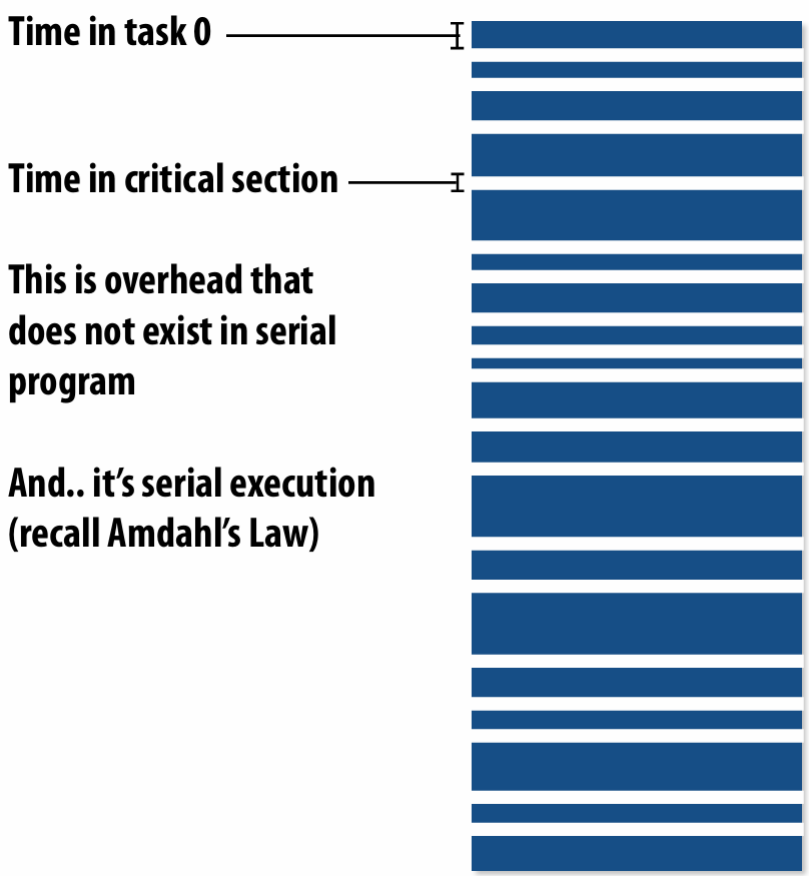
```
int N = 1024;

// assume allocations are only executed by 1 thread
int* x = new int[N];
bool* is_prime = new bool[N];

// assume elements of x are initialized here

LOCK counter_lock;
int counter = 0;    // shared variable

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  is_prime[i] = test_primality(x[i]);
}
```

atomic_incr(counter);

任务分配

- 动态分配容易负载均衡，但引入较多额外开销

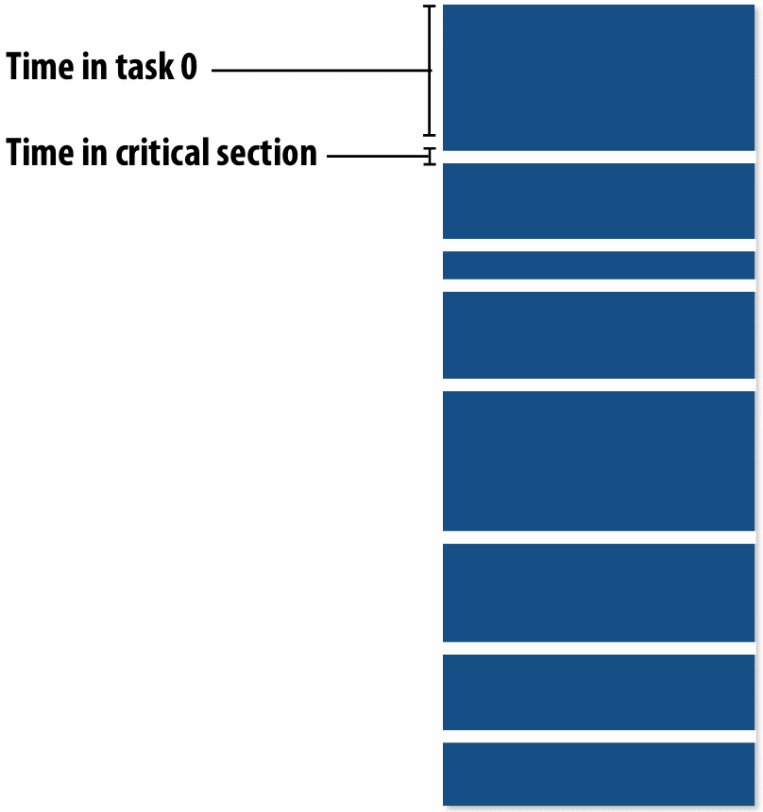
```
const int N = 1024;

// assume allocations are only executed by 1 thread
float* x = new float[N];
bool* prime = new bool[N];

// assume elements of x are initialized here

LOCK counter_lock;
int counter = 0;

while (1) {
  int i;
  lock(counter_lock);
  i = counter++;
  unlock(counter_lock);
  if (i >= N)
     break;
  is_prime[i] = test_primality(x[i]);
}
```



任务分配

- 增加任务粒度，降低额外开销的比例

```
const int N = 1024;
const int GRANULARITY = 10;

// assume allocations are only executed by 1 thread
float* x = new float[N];
bool* prime = new bool[N];

// assume elements of x are initialized here

LOCK counter_lock;
int counter = 0;

while (1) {
  int i;
  lock(counter_lock);
  i = counter;
  counter += GRANULARITY;
  unlock(counter_lock);
  if (i >= N)
     break;
  int end = min(i + GRANULARITY, N);
  for (int j=i; j<end; j++)
     is_prime[i] = test_primality(x[i]);
}
```

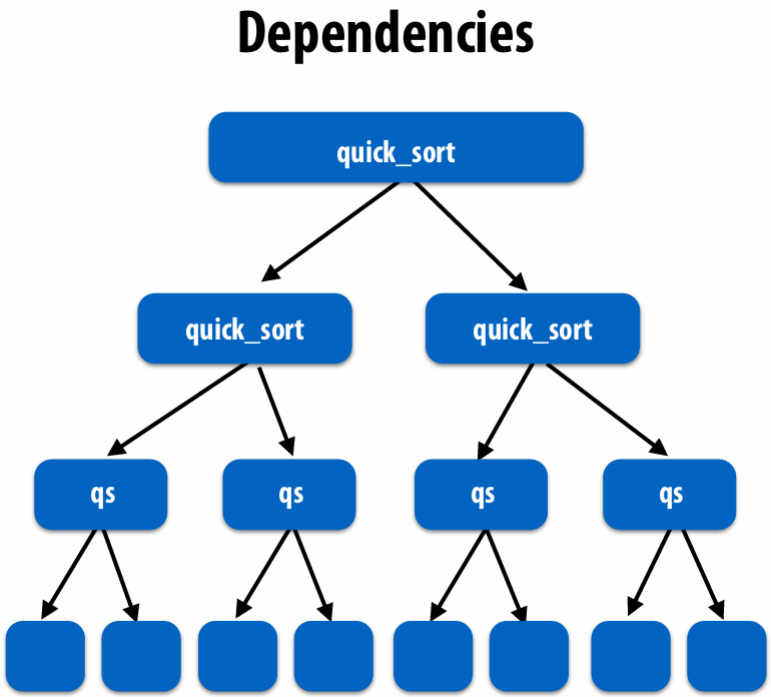


Work stealing

- 回想一下分治并行的快速排序
- 用OpenMP实现，比较麻烦

```
// sort elements from 'begin' up to (but not including) 'end'
void quick_sort(int* begin, int* end) {

  if (begin >= end-1)
    return;

  else {

    // choose partition key and partition elements
    // by key, return position of key as `middle`
    int* middle = partition(begin, end);

    quick_sort(begin, middle);

    quick_sort(middle+1, last);

  }

}
```

independent work!



Work stealing

- 和OpenMP类似，Cilk是一种编程语言扩展
- 提供了Work stealing机制的编程和运行时支持
  - 一种基于Fork-Join的特殊动态线程调度
- 包括Cilk++, Cilk Plus, OpenCilk等版本
  - 目前只有OpenCilk还在维护

Work stealing

- Cilk这个名字的意思大致可以这么解读：
  - 一种基于C语言的高级线程（丝绸）
  - "nice threads" (silk) based on C programming language
  - Cilk和silk谐音，以及thread本意是“线”

Work stealing

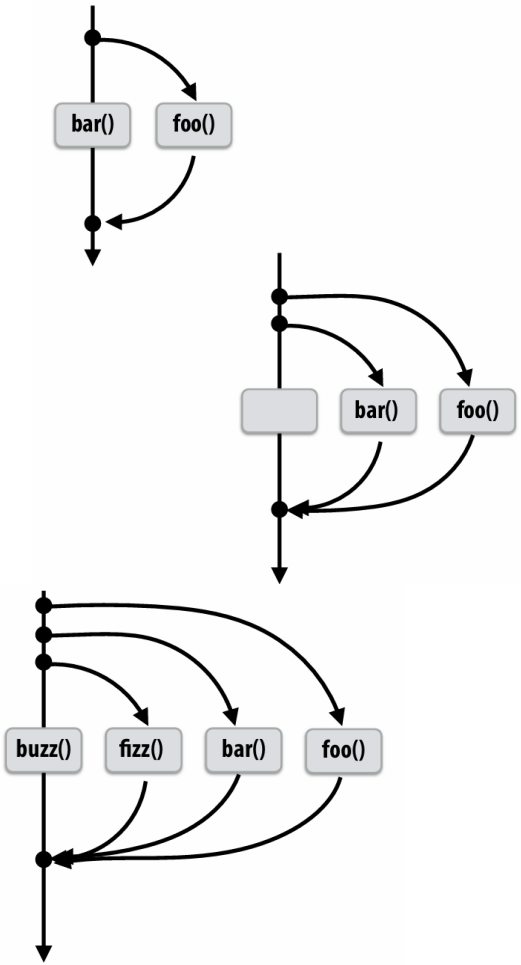
- Cilk的两个基本操作是spawn和sync
  - 也就是fork和join

```
// foo() and bar() may run in parallel
cilk_spawn foo();
bar();
cilk_sync;
```

```
// foo() and bar() may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_sync;
```

Same amount of independent work first example, but potentially higher runtime overhead (due to two spawns vs. one)

```
// foo, bar, fizz, buzz, may run in parallel
cilk_spawn foo();
cilk_spawn bar();
cilk_spawn fizz();
buzz();
cilk_sync;
```

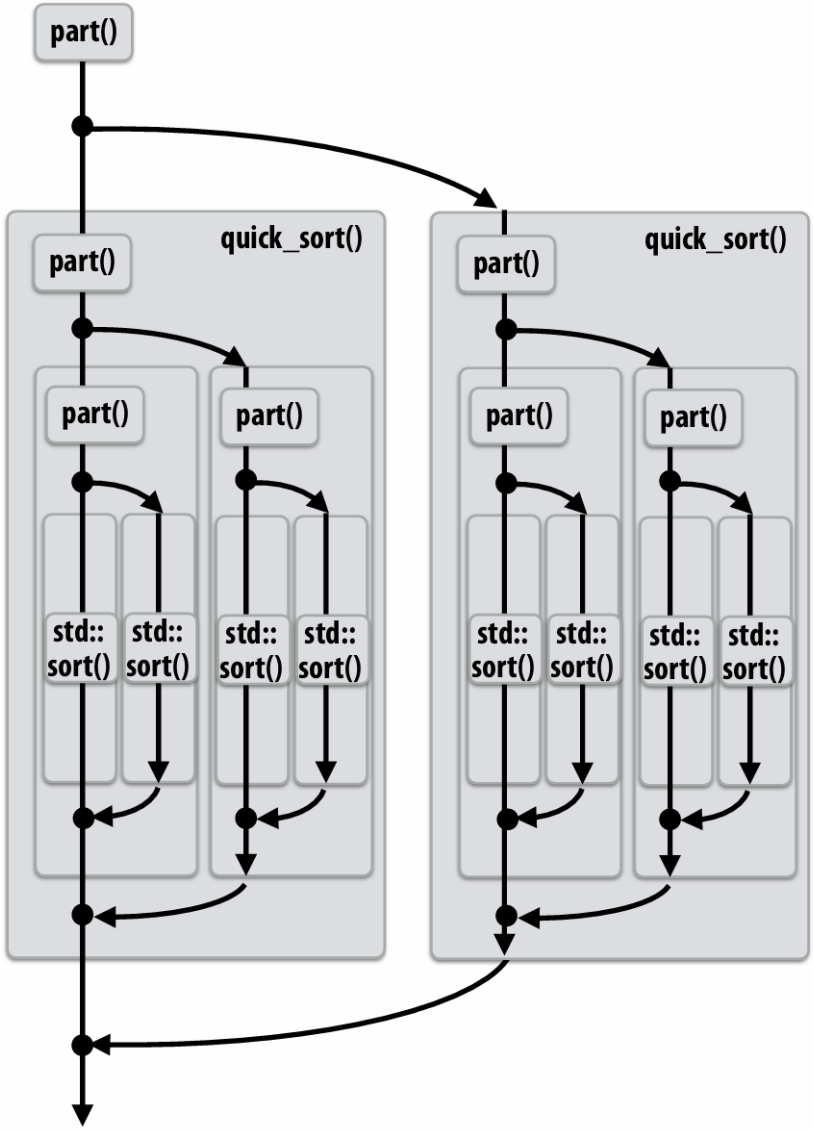


Work stealing

- 基于Cilk的并行快速排序

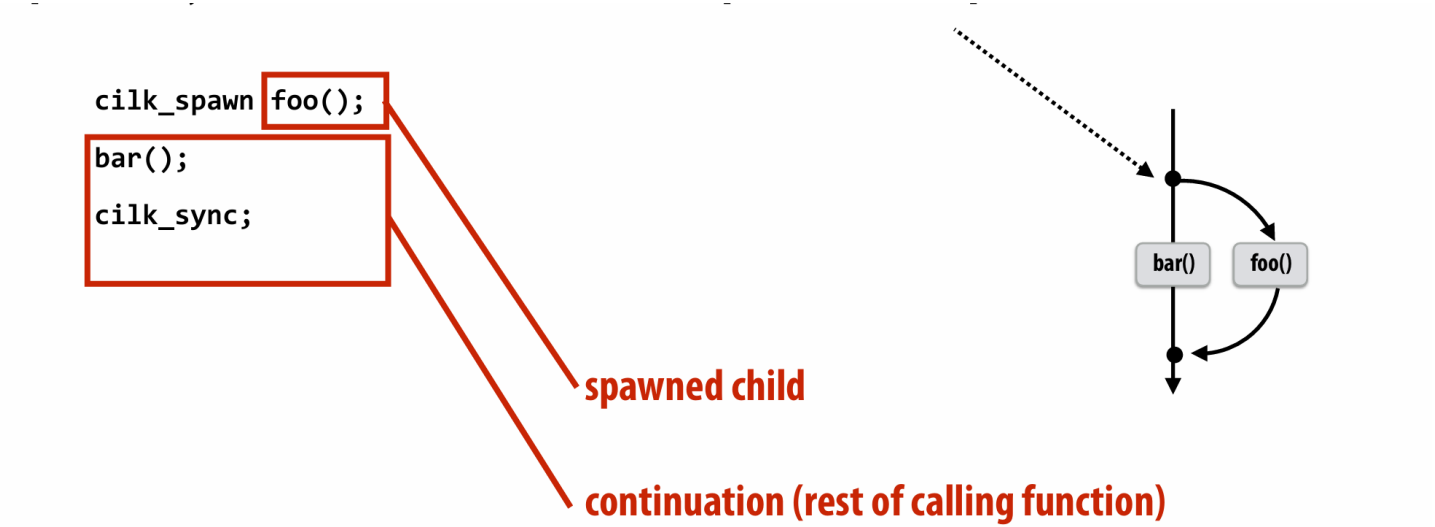
```
void quick_sort(int* begin, int* end) {
  if (begin >= end - PARALLEL_CUTOFF)
    std::sort(begin, end);
  else {
     int* middle = partition(begin, end);
     cilk_spawn quick_sort(begin, middle);
     quick_sort(middle+1, last);
  }
}
```

Sort sequentially if problem size is sufficiently small (overhead of spawn trumps benefits of potential parallelization)

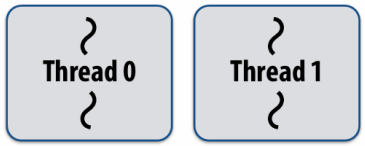


Work stealing

- Cilk运行时维护一个线程池，每个线程有一个任务队列。
  需要决定每个任务实际由哪一线程执行或插入到哪个队列。



Assignment question: what threads should foo() and bar() be executed by?



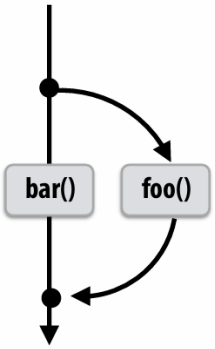
Work stealing

- Cilk运行时需要决定任务执行顺序

```
cilk_spawn foo();
bar();
cilk_sync;
```

spawned child

continuation (rest of calling function)



Run continuation first: queue child for later execution

- Child is made available for stealing by other threads ("child stealing")

当前线程先运行后续任务，将孩子任务加入队列

Run child first: enqueue continuation for later execution

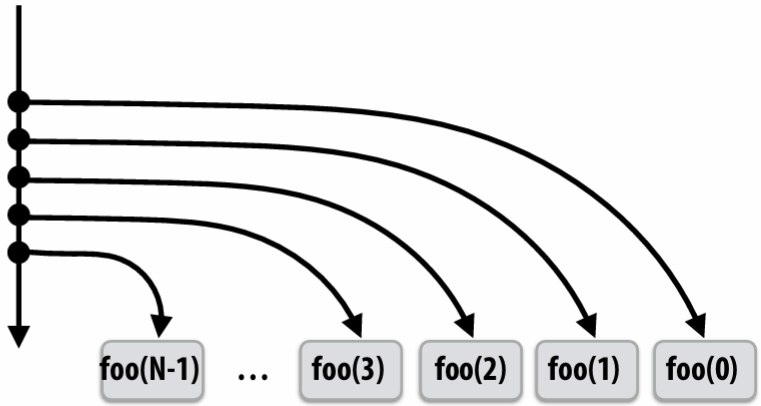
- Continuation is made available for stealing by other threads ("continuation stealing")

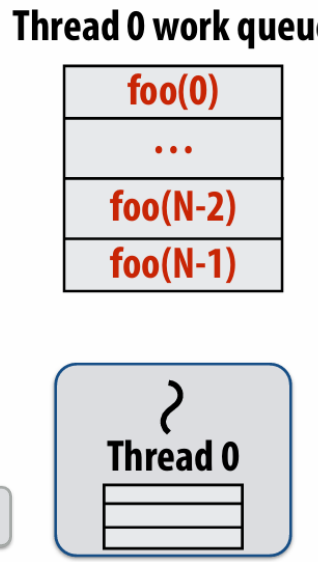
反之

Work stealing

- Run continuation first (“child stealing”)
- 新增的任务先加入队列
- 相当于宽度优先遍历任务依赖图

```
for (int i=0; i<N; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
```

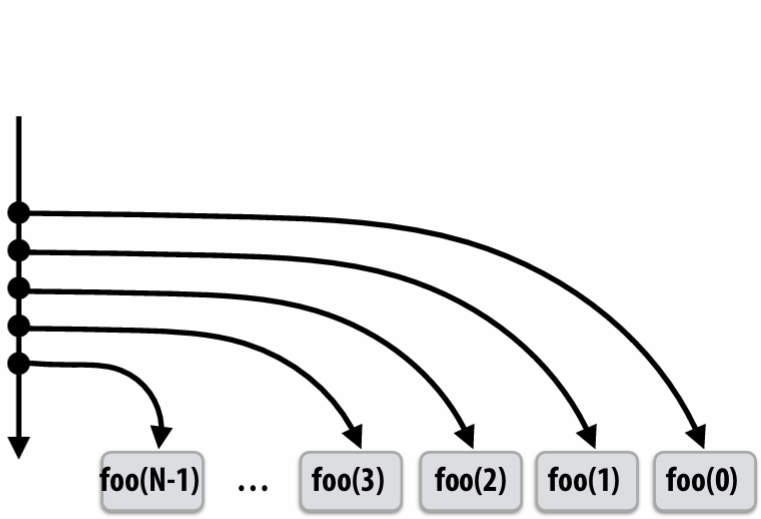


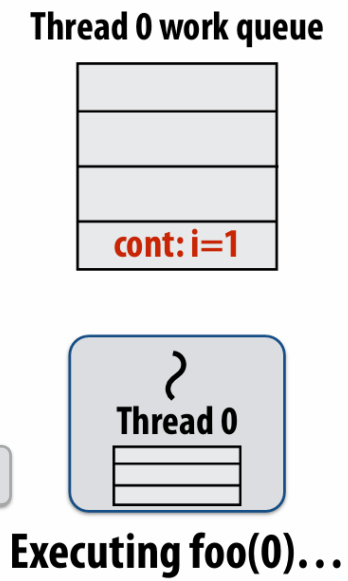


Work stealing

- Run child first (“continuation stealing”)
- 先运行新增的任务
- 相当于深度优先遍历任务依赖图

```
for (int i=0; i<N; i++) {
  cilk_spawn foo(i);
}
cilk_sync;
```

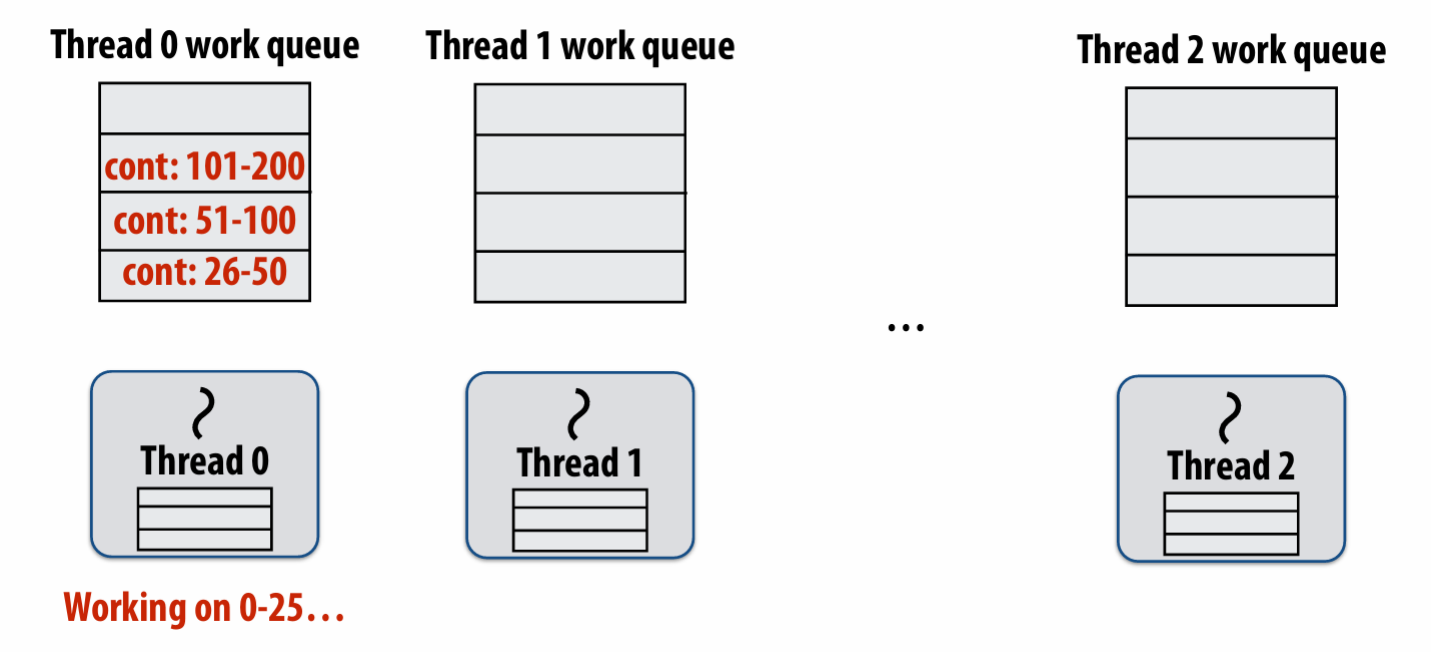




Work stealing

- 假设要排序200个元素
- 线程0已产生一些任务

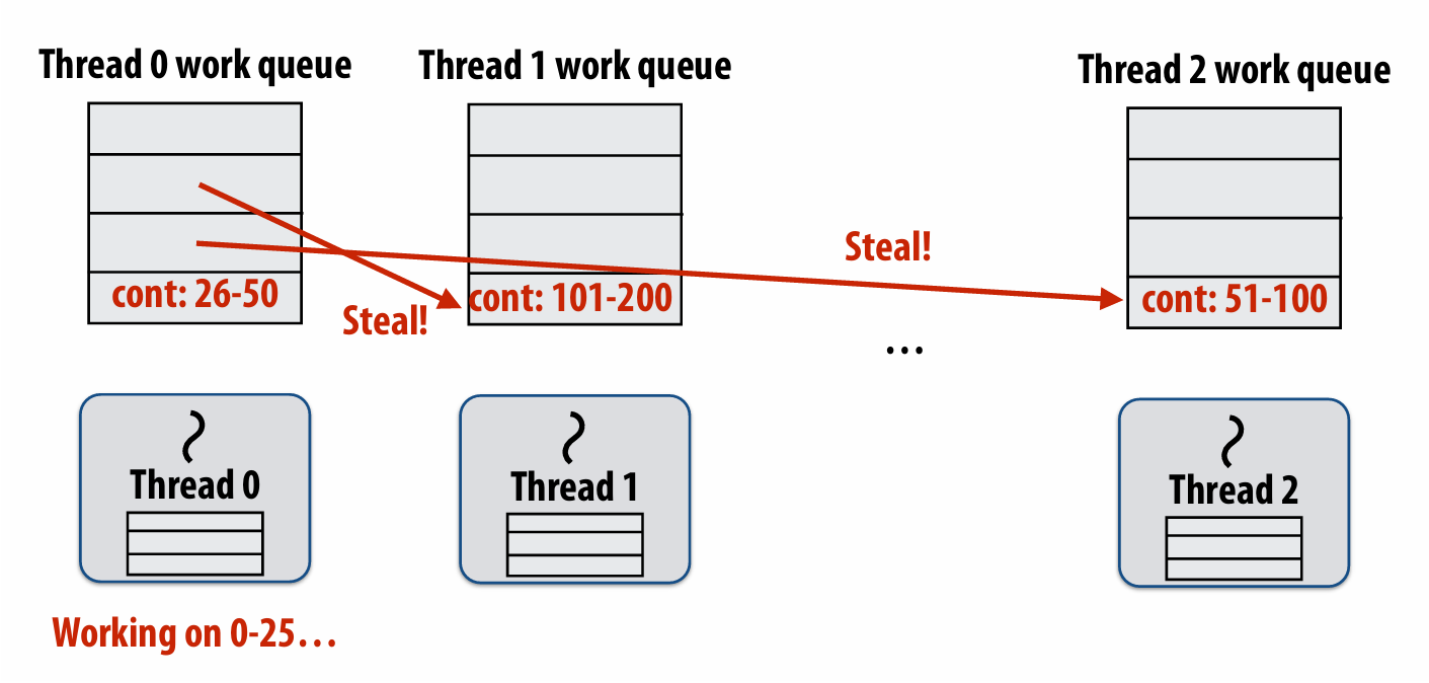
```
void quick_sort(int* begin, int* end) {
  if (begin >= end - PARALLEL_CUTOFF)
    std::sort(begin, end);
  else {
     int* middle = partition(begin, end);
     cilk_spawn quick_sort(begin, middle);
     quick_sort(middle+1, last);
  }
}
```



Work stealing

- 其他线程偷取任务

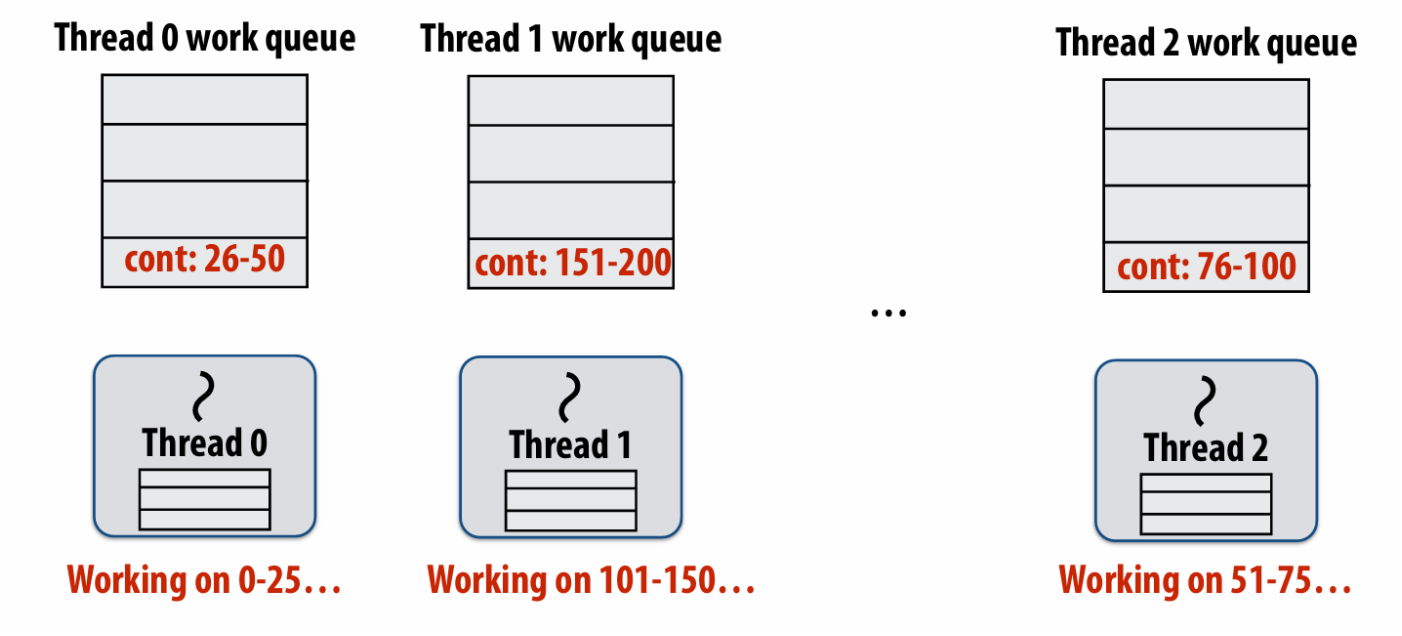
```
void quick_sort(int* begin, int* end) {
  if (begin >= end - PARALLEL_CUTOFF)
    std::sort(begin, end);
  else {
     int* middle = partition(begin, end);
     cilk_spawn quick_sort(begin, middle);
     quick_sort(middle+1, last);
  }
}
```



Work stealing

- 每个线程执行任务

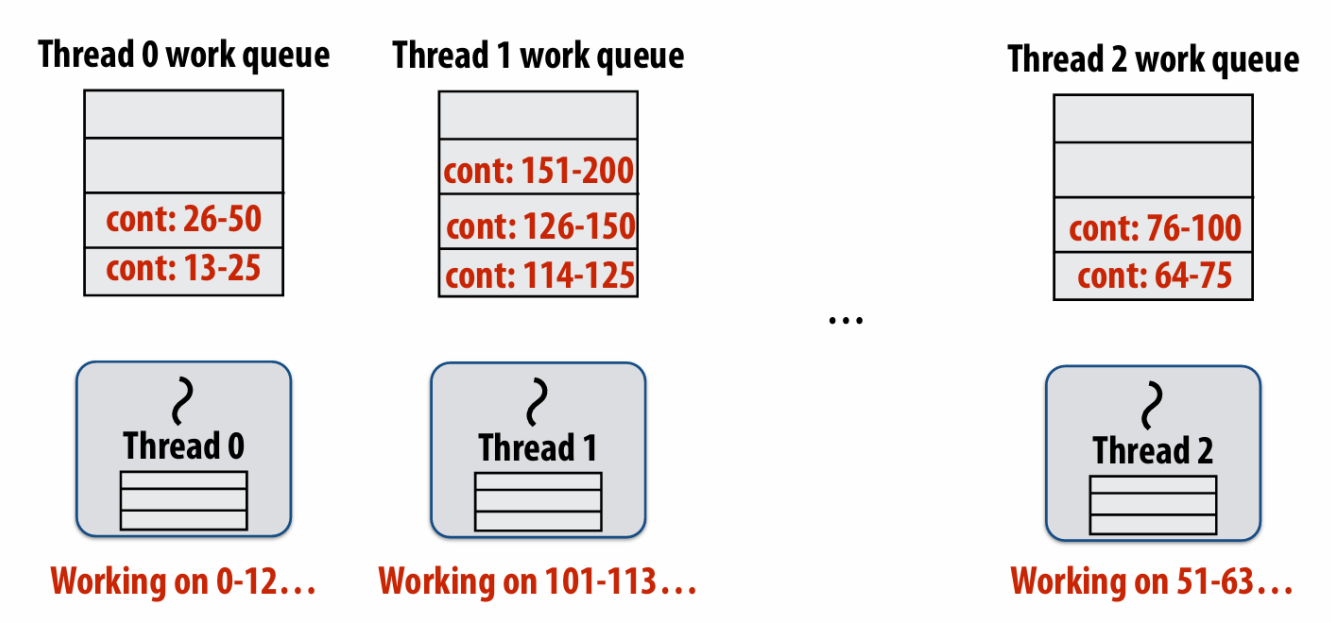
```
void quick_sort(int* begin, int* end) {
  if (begin >= end - PARALLEL_CUTOFF)
    std::sort(begin, end);
  else {
     int* middle = partition(begin, end);
     cilk_spawn quick_sort(begin, middle);
     quick_sort(middle+1, last);
  }
}
```



Work stealing

- 每个线程产生新任务

```
void quick_sort(int* begin, int* end) {
  if (begin >= end - PARALLEL_CUTOFF)
    std::sort(begin, end);
  else {
     int* middle = partition(begin, end);
     cilk_spawn quick_sort(begin, middle);
     quick_sort(middle+1, last);
  }
}
```



并行程序性能优化

- 任务分配与调度
- **锁**
- 一致性

数据竞争

- 当并行程序在共享变量的多处理机系统上执行时，多个线程通常需要对共享变量执行诸如“x++”这样的读写操作，但这样的读写操作并不具有原子性，在现代机器上通常实现为多条指令。

线程 1	线程 2	变量 x 的值
LOAD A,(x address)		0
	LOAD B,(x address)	0
ADD A, 1		0
	ADD B,1	0
STORE A,(x address)		1
	STORE B,(x address)	1

数据竞争

- 多个线程对共享数据的非同步访问会引起数据竞争(Data Race)，得到的结果将依赖于这些线程对共享数据读操作和写操作之间的相对执行顺序，具有不确定性。
- 锁（lock）可以保证线程对共享变量的读写具有原子性，避免数据竞争引起的读写错误。
- 锁通常是基于一些原子操作指令实现的，例如Test-and-set，Compare-and-swap (CAS)

锁

- 全局锁
- 细粒度锁
- 无锁
- 死锁
- 活锁

案例 —— 有序链表

```
struct Node {
   int value;
   Node* next;
};
```

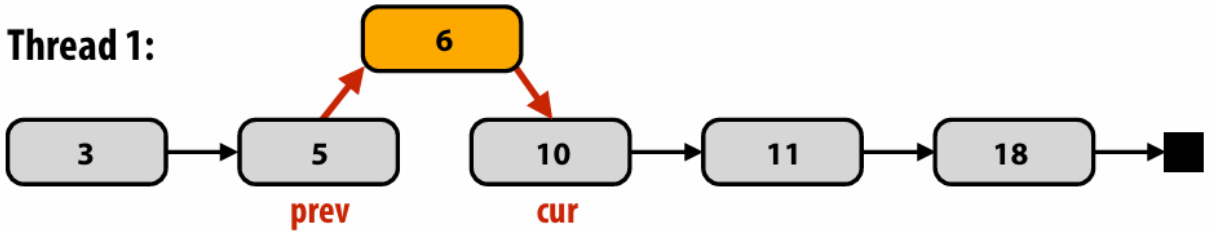
```
struct List {
  Node* head;
};
```

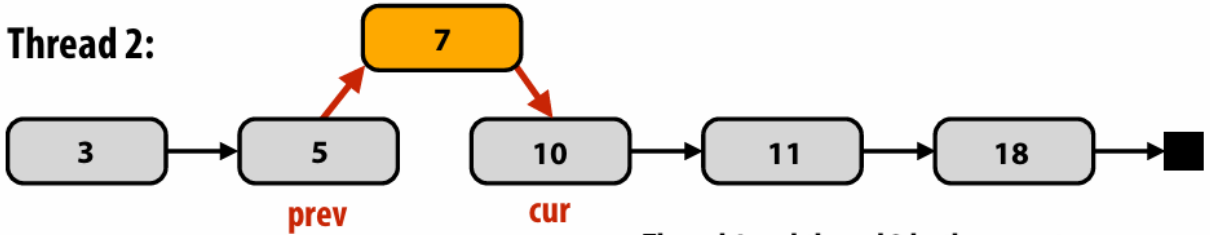
```
void insert(List* list, int value) {

   Node* n = new Node;
   n->value = value;

   // assume case of inserting before head of
   // of list is handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value > value)
       break;

     prev = cur;
     cur = cur->next;
   }

   n->next = cur;
   prev->next = n;
}
```

```
void delete(List* list, int value) {

   // assume case of deleting first node in list
   // is handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value == value) {
       prev->next = cur->next;
       delete cur;
       return;
     }

     prev = cur;
     cur = cur->next;
   }
}
```

并发有序链表

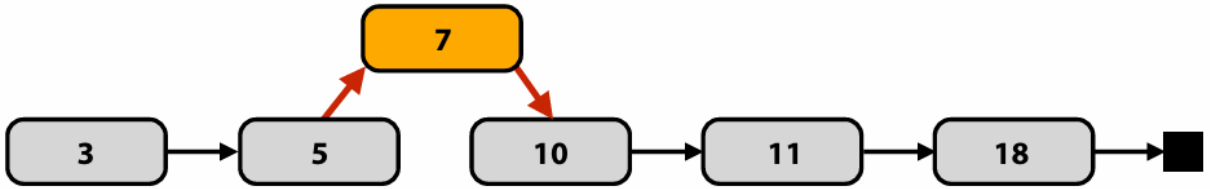
Thread 1 attempts to insert 6
Thread 2 attempts to insert 7





Thread 1 and thread 2 both compute same prev and cur.
Result: one of the insertions gets lost!

Result: (assuming thread 1 updates `prev->next` before thread 2)



并发有序链表（全局锁）

```
struct Node {
   int value;
   Node* next;
};
```

```
struct List {
  Node* head;
  Lock  lock;
};
```

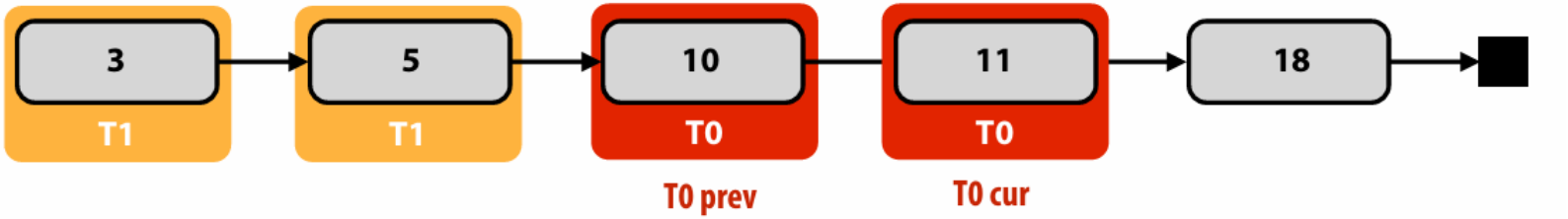
Per-list lock

```
void insert(List* list, int value) {

   Node* n = new Node;
   n->value = value;

   lock(list->lock);

   // assume case of inserting before head of
   // of list is handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value > value)
       break;

     prev = cur;
     cur = cur->next;
   }
   n->next = cur;
   prev->next = n;
   unlock(list->lock);
}
```

```
void delete(List* list, int value) {

   lock(list->lock);

   // assume case of deleting first element is
   // handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value == value) {
       prev->next = cur->next;
       delete cur;
       unlock(list->lock);
       return;
     }

     prev = cur;
     cur = cur->next;
   }
   unlock(list->lock);
}
```

并发有序链表（细粒度锁）

- Thread 0: delete(11)
- Thread 1: delete(3)
- Thread 2: delete(10)



无锁

- 锁的底层实现是原子操作指令，但通常也涉及很多软件操作以及操作系统的管理，带来很多额外开销。
- 如果跳过这些软件部分，直接使用原子操作指令，能不能提升性能？
- 这就是无锁数据结构（Lock-free data structures）的基本思想。

无锁

无锁编程常用指令

- compare-and-swap (CAS)
- load-link/store-conditional (LL/SC)
- fetch-and-op

具体编程实现可以使用C11 #include <stdatomic.h>

无锁

- compare-and-swap (CAS)

当目标变量为一个指定的旧值时，更新为新值

```
bool compare_and_swap(int* ptr, int old_val, int new_val) {
    if (*ptr == old_val) {
        *ptr = new_val;
        return true;
    }
    return false;
}
```

无锁

- load-link/store-conditional (LL/SC)

LL返回变量的当前值，SC尝试将新值写回该变量如果自LL以来没有新值写入

```
int load_link(int* ptr) {
    return *ptr;  // 模拟加载链接操作
}

bool store_conditional(int* ptr, int val) {
    // 假设自load_link()调用以来没有其他线程写入*ptr
    // 这是一个简化 - 实际上，硬件确保没有中间写入
    *ptr = val;
    return true;  // 模拟存储成功
}
```

无锁

- fetch-and-op

对变量执行某种操作，并返回旧值

```
int fetch_and_add(int* ptr, int add_val) {
    int old_val = *ptr;
    *ptr += add_val;
    return old_val;
}
```

并发有序链表（无锁）

```
struct Node {
   int value;
   Node* next;
};

struct List {
  Node* head;
};

// insert new node after specified node
void insert_after(List* list, Node* after, int value) {

   Node* n = new Node;
   n->value = value;

   // assume case of insert into empty list handled
   // here (keep code on slide simple for class discussion)

   Node* prev = list->head;

   while (prev->next) {
     if (prev == after) {
       while (1) {
         Node* old_next = prev->next;
         n->next = old_next;
         if (compare_and_swap(&prev->next, old_next, n) == old_next)
             return;
       }
     }

     prev = prev->next;
   }
}
```

死锁（Deadlock）

- 死锁的发生的条件:
    1. 资源访问互斥;
    2. 请求对方拥有的资源;
    3. 资源持有不可剥夺;
    4. 循环等待。

- 通信死锁

（回看MPI相关内容）

活锁（Livelock）

- 活锁通常发生在系统设计中存在过度的响应机制时
- 线程为了避免冲突而不断改变状态，但这些改变反而形成循环，导致无法完成任何实际的工作。

并行程序性能优化

- 任务分配与调度
- 锁
- **一致性**

一致性

- 缓存一致性（Cache Coherence）
- 内存一致性（Memory Consistency）

缓存一致性

- 缓存一致性主要是指在多处理器系统中，当多个处理器的缓存中存储了同一内存地址的副本时，保证这些副本的一致性。
- 当其中一个处理器修改了数据后，其他处理器的缓存副本也需要相应更新，以避免数据不一致的问题。

- 缓存一致性的保证机制不是本课程内容，略

伪共享

- Cache行(Cache Line)是Cache与内存或Cache与Cache之间交换数据的最小存储单元，是内存中一块连续存储单元的镜像。
- 两个独立的Cache可从内存中读取具有相同地址空间的Cache行，从而共享该Cache行。
- 如果在其中一个Cache中，该Cache行被写入，而在另一个Cache中，该Cache行被读取，那么即使它们读写的地址不相交，为了保持Cache一致性，也需要在这两个Cache之间移动该Cache行。

伪共享

- 多个处理器需要读写的不同数据共享一个Cache行称为Cache伪共享(False Sharing)。当两个处理器读写同一Cache行的不同部分时，会导致该Cache行在两个Cache之间来回移动，也叫Cache行乒乓现象。

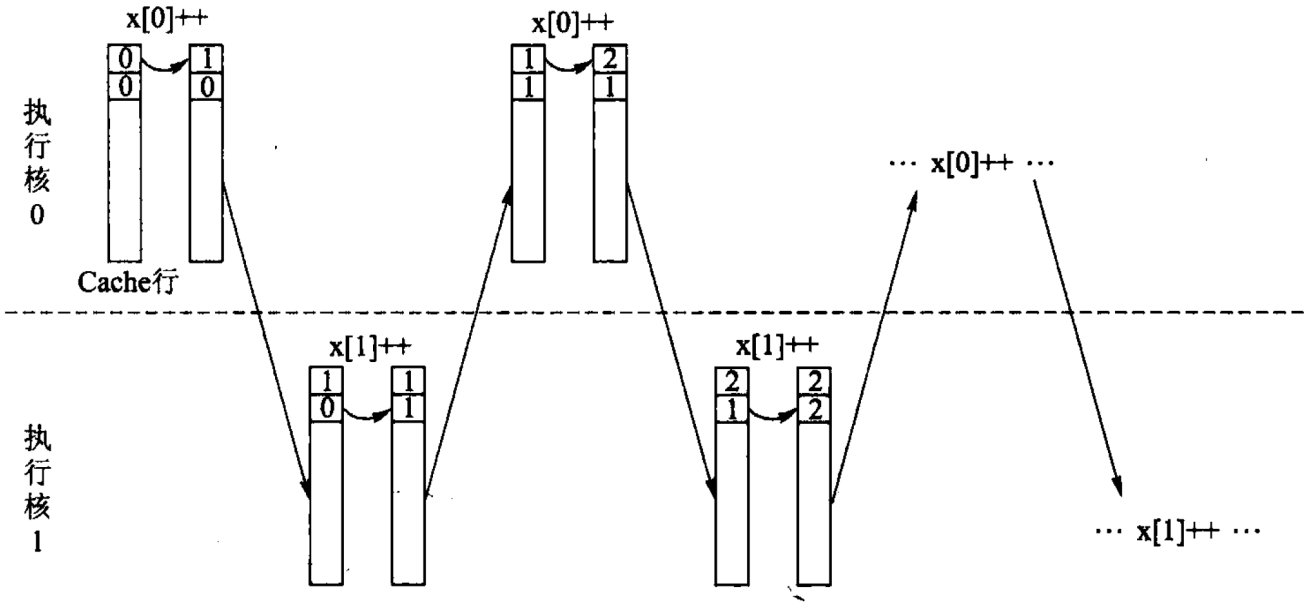


图 13.8　伪共享引起的 Cache 行乒乓现象

伪共享

- 解决方法

```
struct Data {
    int a;
    int b;
};

struct Data data;

void thread1() {
    for (int i = 0; i < 1000000; ++i) {
        data.a++;
    }
}

void thread2() {
    for (int i = 0; i < 1000000; ++i) {
        data.b++;
    }
}
```

1. 数据填充（Padding）：
在变量之间插入足够的填充数据，确保它们不在同一个缓存行中。

```
struct Data {
    int a;
    char padding[64]; // 假设缓存行大小为64字节
    int b;
};
```

2. 对齐：
使用编译器提供的对齐指令，确保变量在不同的缓存行中。

```
struct Data {
    int a __attribute__((aligned(64)));
    int b __attribute__((aligned(64)));
};
```

内存一致性

- 内存一致性是指在多处理器系统中，多个处理器对共享内存的访问顺序和结果的一致性。
- 内存一致性模型（Memory Consistency Model）是一种抽象，用于描述在多处理器系统中，处理器如何看到内存操作的顺序。这些模型定义了读写操作的顺序规则和可见性，影响程序的行为和正确性。

- 两个重要的内存一致性模型
  - 顺序一致性 Sequential consistency
  - 松弛一致性 Relaxed consistency (实际上包括了一类模型)

内存一致性

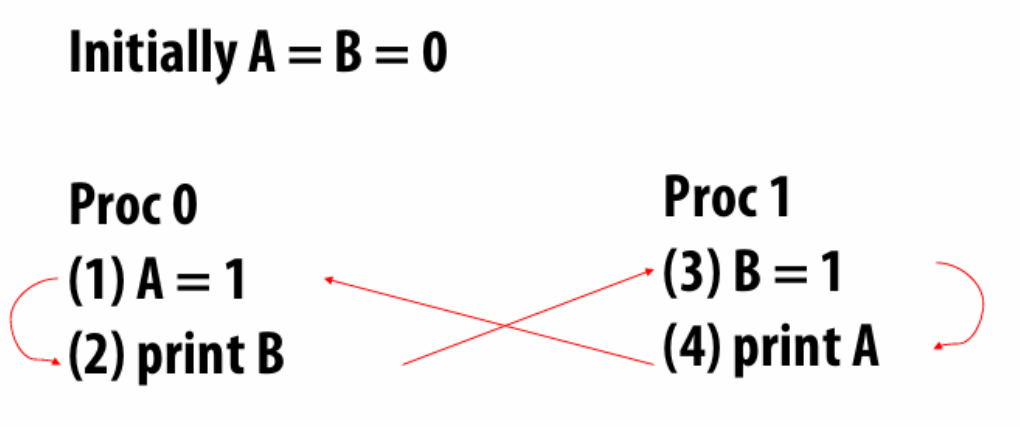
Initially A = B = 0

Proc 0	Proc 1
(1) A = 1	(3) B = 1
(2) print B	(4) print A

What can be printed?

- "01"?
- "10"?
- "11"?
- "00"?

内存一致性



- The program should not print “00” or “10”
- A “happens-before” graph shows the order in which events must execute to get a desired outcome
- If there’s a cycle in the graph, an outcome is impossible—an event must happen before itself!

内存一致性

- 考虑对变量X, Y的读和写，顺序一致性要求必须保证四种顺序：

指的是代码词法顺序

Four types of memory operation orderings

- $W_X \rightarrow R_Y$: write to X must commit before subsequent read from Y *
- $R_X \rightarrow R_Y$: read from X must commit before subsequent read from Y
- $R_X \rightarrow W_Y$: read to X must commit before subsequent write to Y
- $W_X \rightarrow W_Y$: write to X must commit before subsequent write to Y

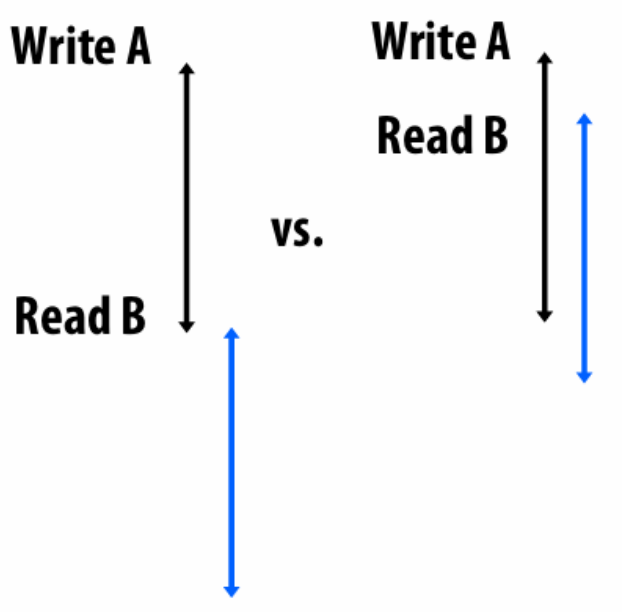
Lamport, Leslie. "How to make a multiprocessor computer that correctly executes multiprocess programs." *IEEE Transactions on Computers* 100.9 (1979): 690-691.

内存一致性

- 顺序一致性要求过于严格，损失了一些可并行性
- 松弛一致性允许部分违反顺序一致性

Four types of memory operation orderings

- ~~$W_X \rightarrow R_Y$: write must complete before subsequent read~~
- $R_X \rightarrow R_Y$: read must complete before subsequent read
- $R_X \rightarrow W_Y$: read must complete before subsequent write
- $W_X \rightarrow W_Y$: write must complete before subsequent write



概念区分

- The cache coherency problem exists because hardware implements the optimization of duplicating data in multiple processor caches. The copies of the data must be kept coherent. 缓存一致性问题的存在是因为硬件实现了将数据复制到多个处理器缓存中的优化。数据的副本必须保持一致。
- Relaxed memory consistency issues arise from the optimization of reordering memory operations. Consistency is unrelated to whether or not caches exist in the system. 松弛内存一致性问题源于内存操作重排序的优化。一致性问题与系统中是否存在缓存无关。

并行程序性能优化小结

- 任务分配与调度
  - 并行的额外开销、任务粒度
  - 线程数的设置（进程数也类似）
  - 性能瓶颈分析
  - 负载均衡、静态与动态分配
- 锁
  - 细粒度锁与无锁编程
- 一致性
  - 伪共享