



第12章 并发编程

线程级并行 Thread-Level Parallelism

100076202: 计算机系统导论

任课教师:

宿红毅 张艳 黎有琦 李秀星

原作者:

Randal E. Bryant and David R. O'Hallaron



Carnegie
Mellon
University

议题 Today



■ 并行计算硬件 **Parallel Computing Hardware**

- 多核 Multicore
 - 单个芯片上有多个独立处理器 Multiple separate processors on single chip
- 超线程化 Hyperthreading
 - 在单核上高效执行多个线程 Efficient execution of multiple threads on single core

■ 一致性模型 **Consistency Models**

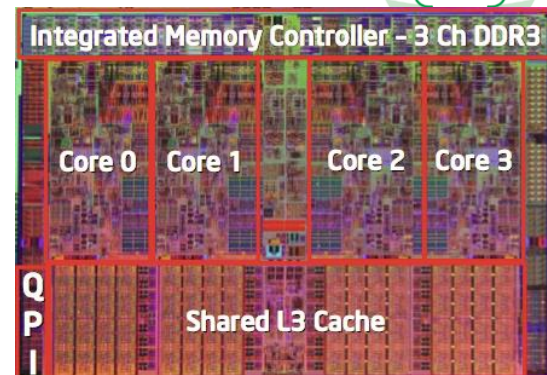
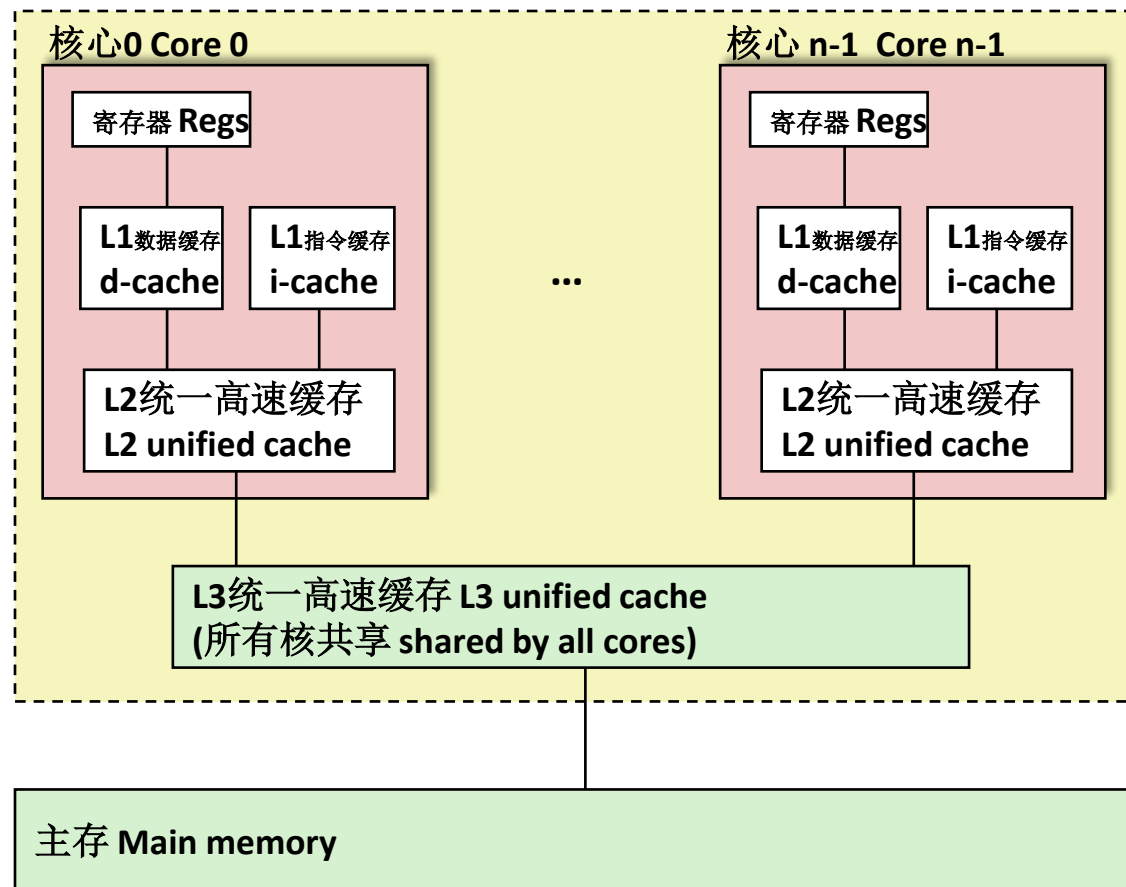
- 当多个线程读取和写入共享状态时会发生什么 What happens when multiple threads are reading & writing shared state

■ 线程级并行 **Thread-Level Parallelism**

- 将程序拆分为独立任务 Splitting program into independent tasks
 - 示例：并行求和 Example: Parallel summation
 - 检查一些性能工件 Examine some performance artifacts
- 分而治之 Divide-and conquer parallelism
 - 示例：并行快速排序 Example: Parallel quicksort

典型的多核处理器

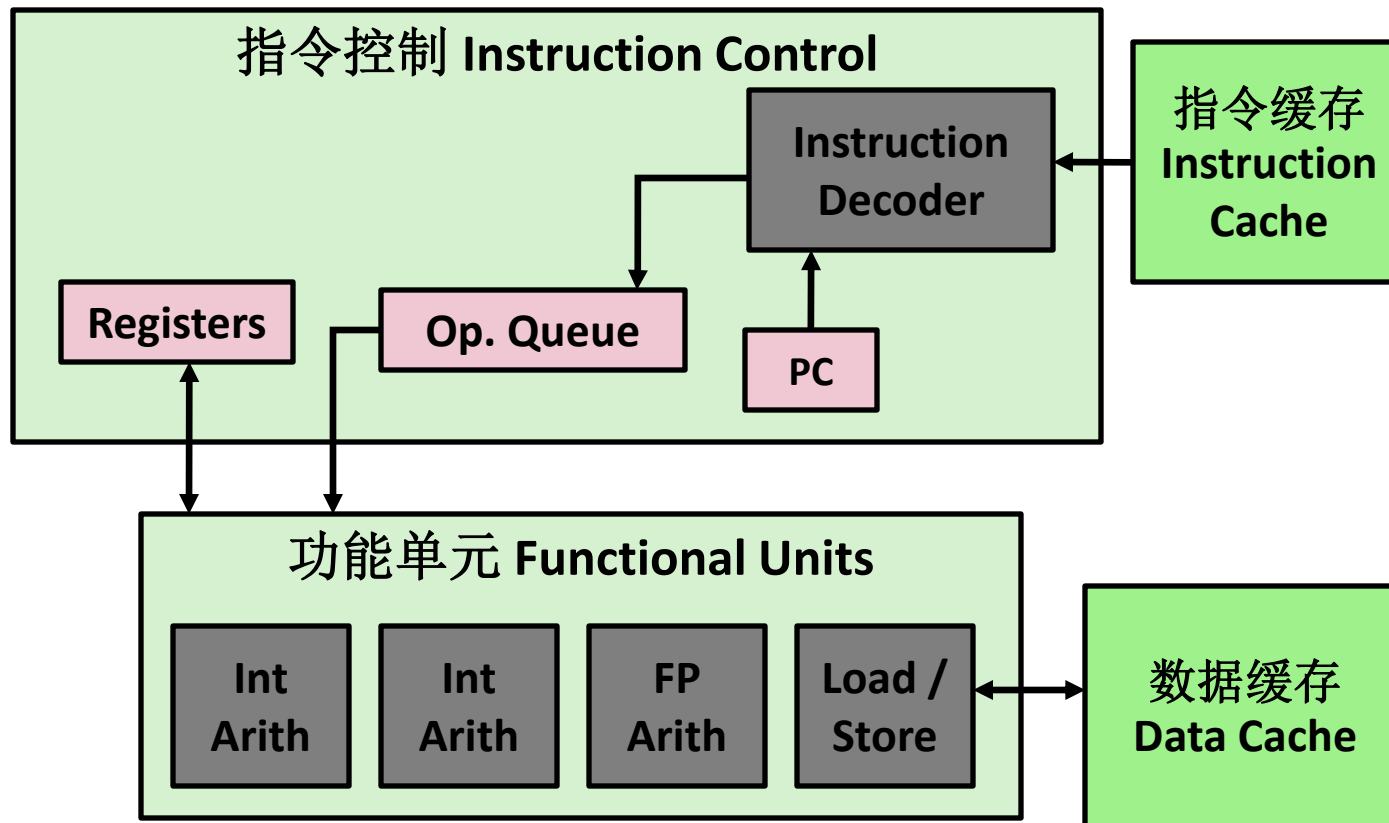
Typical Multicore Processor



- 多个处理器以一致的内存视图运行 Multiple processors operating with coherent view of memory

乱序处理器结构

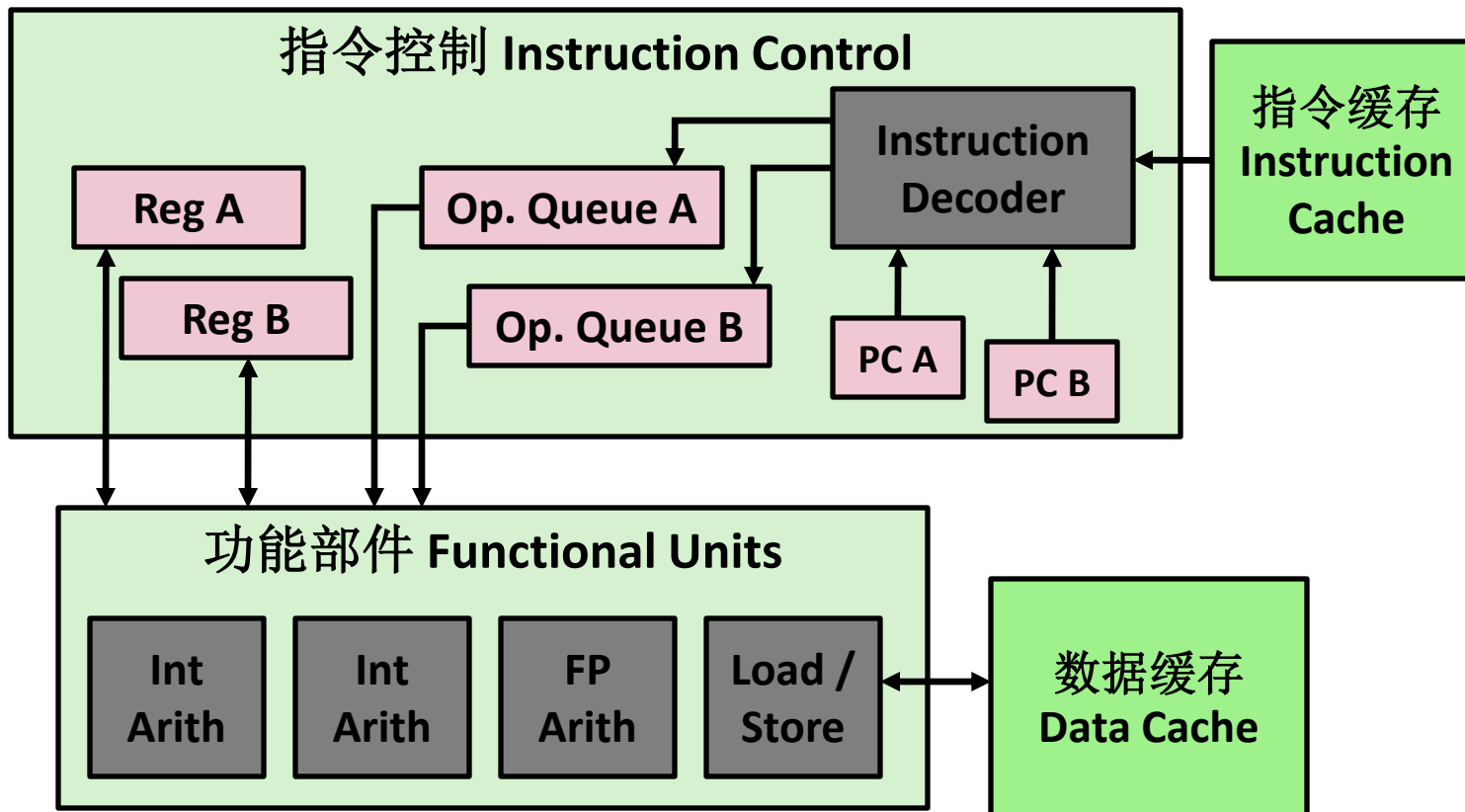
Out-of-Order Processor Structure



- 指令控制将程序动态转换为操作流 **Instruction control dynamically converts program into stream of operations**
- 操作映射到功能单元以并行方式执行 **Operations mapped onto functional units to execute in parallel**

超线程实现

Hyperthreading Implementation



- 复制指令控制以处理K个指令流 Replicate instruction control to process K instruction streams
- 所有寄存器有K份拷贝 K copies of all registers
- 共享功能单元 Share functional units



基准测试机 Benchmark Machine

- 从/`proc/cpuinfo`获取有关计算机的数据 Get data about machine from `/proc/cpuinfo`
- Shark机器 Shark Machines
 - Intel Xeon E5520 @ 2.27 GHz
 - Nehalem, ca. 2010
 - 8核 8 Cores
 - 每个核心可以执行2倍超线程 Each can do 2x hyperthreading

利用并行执行 Exploiting parallel execution



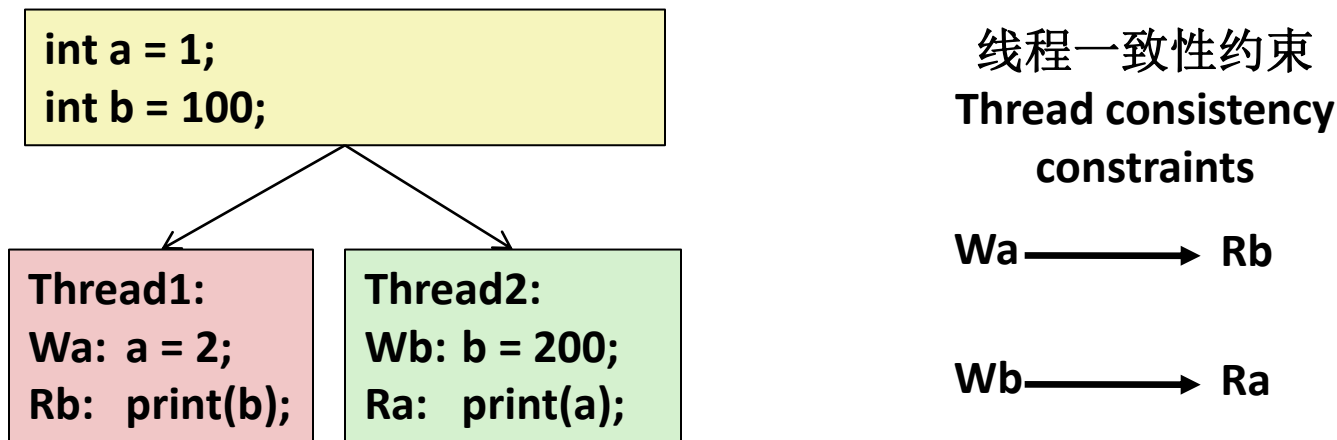
- 到目前为止，我们已经使用线程来处理I/O延迟 So far, we've used threads to deal with I/O delays
 - 例如每个客户端一个线程，以防止一个线程延迟另一个线程 e.g., one thread per client to prevent one from delaying another
- 多核CPU提供了另一个机会 Multi-core CPUs offer another opportunity
 - 在N个核心上并行执行的线程上扩展工作 Spread work over threads executing in parallel on N cores
 - 如果有许多独立任务，则自动发生 Happens automatically, if many independent tasks
 - 例如，运行许多应用程序或为许多客户端提供服务 e.g., running many applications or serving many clients
 - 还可以编写代码以加快一项大型任务的执行速度 Can also write code to make one big task go faster
 - 通过将其组织为多个并行子任务 by organizing it as multiple parallel sub-tasks

利用并行执行 Exploiting parallel execution



- **Shark机器可以同时执行16个线程 Shark machines can execute 16 threads at once**
 - 8核心, 每个带2路超线程 8 cores, each with 2-way hyperthreading
 - 理论上16倍加速比 Theoretical speedup of 16X
 - 在我们的基准测试中从未达到 never achieved in our benchmarks

内存一致性 Memory Consistency



■ 打印的可能值是什么？ What are the possible values printed?

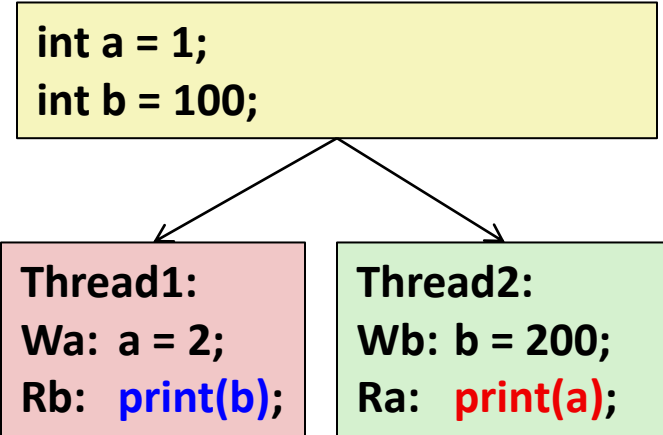
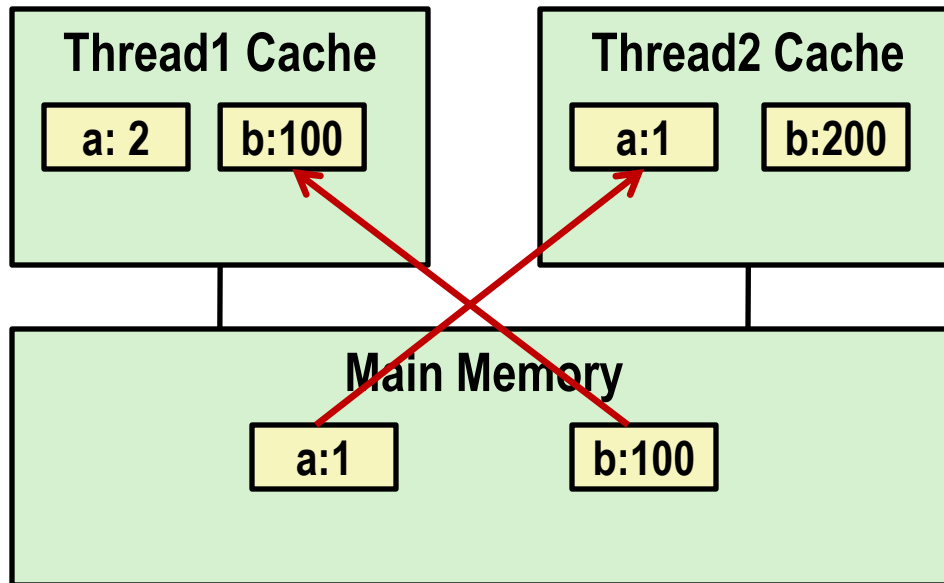
- 取决于内存一致性模型 Depends on memory consistency model
- 硬件如何处理并发访问的抽象模型 Abstract model of how hardware handles concurrent accesses



非一致性高速缓存方案

Non-Coherent Cache Scenario

- 写回高速缓存，线程间没有协作 Write-back caches, without coordination between them



print 1

print 100

稍后，`a:2`和`b:200`被写回主存储器
At later points, `a:2` and `b:200` are written back to main memory

Snoopy缓存

Snoopy Caches



■ 用状态标记每个缓存块 Tag each cache block with state

无效 Invalid

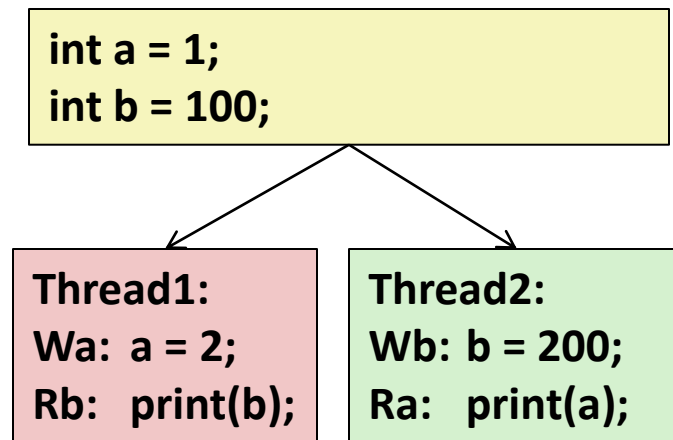
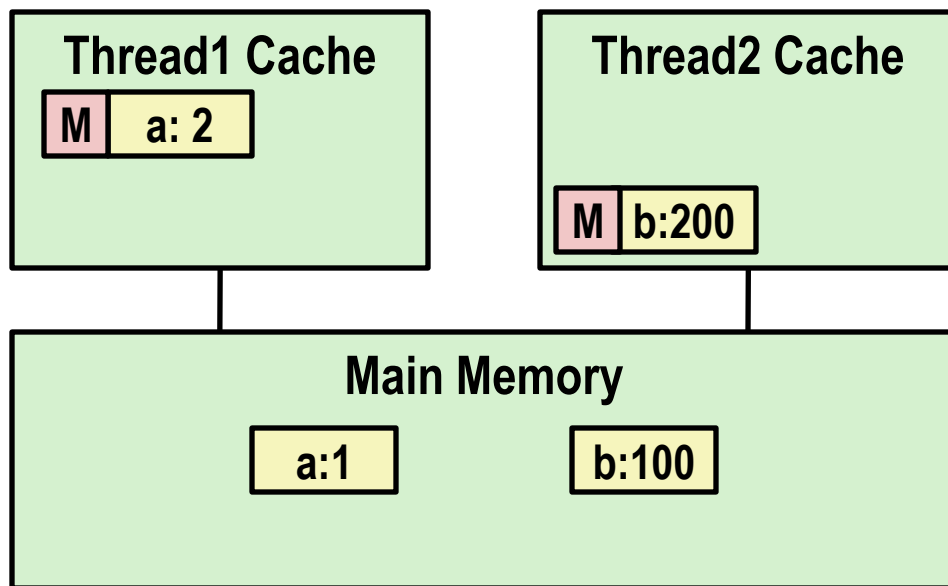
不能使用其值 Cannot use value

共享 Shared

可读拷贝 Readable copy

修改 Modified

可写拷贝 Writeable copy



Snoopy缓存

Snoopy Caches

- 用状态标记每个缓存块
- Tag each cache block with state

无效 Invalid

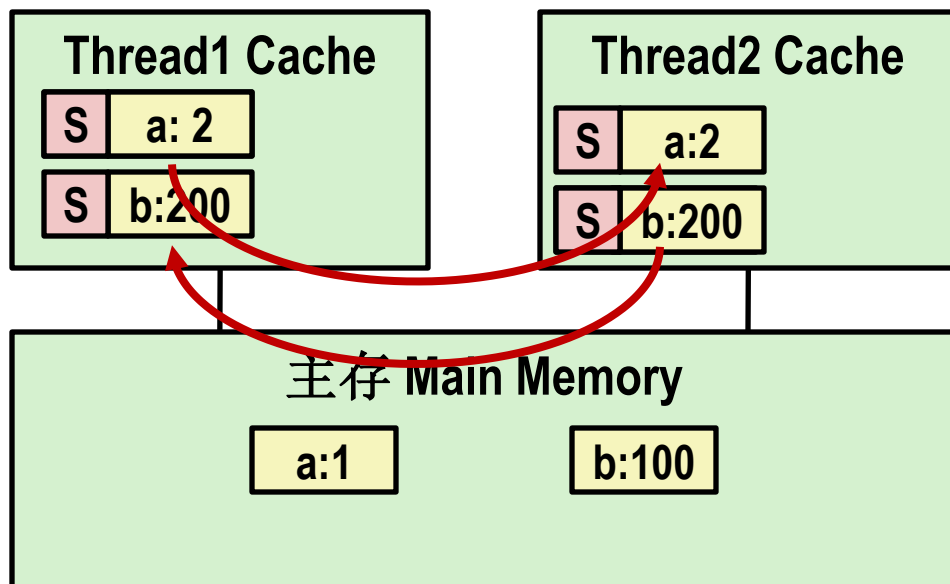
共享 Shared

修改 Modified

不能使用其值 Cannot use value

可读拷贝 Readable copy

可写拷贝 Writeable copy



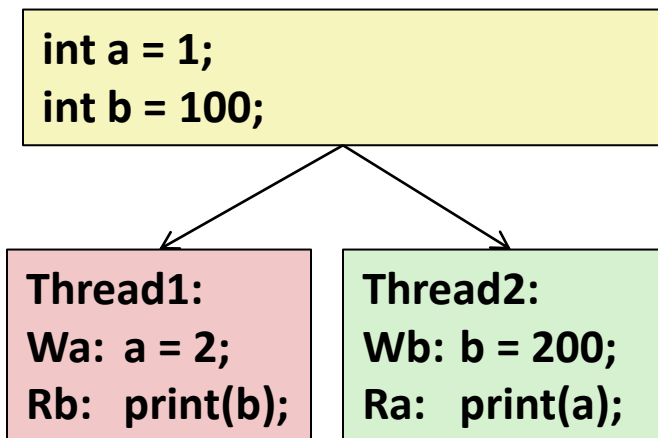
print 2

print 200

当缓存看到对其M标记块之一的请求时 When cache sees request for one of its M-tagged blocks

- 从缓存提供值（注意：内存中的值可能已过时） Supply value from cache (Note: value in memory may be stale)
- 将标记设置为S Set tag to S

内存一致性 Memory Consistency



线程一致性约束
Thread consistency
constraints

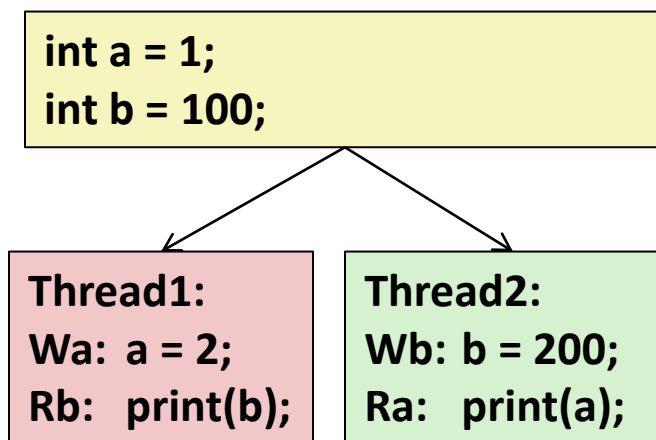
Wa → Rb

Wb → Ra

■ 打印的可能值是什么？ What are the possible values printed?

- 取决于内存一致性模型 Depends on memory consistency model
- 硬件如何处理并发访问的抽象模型 Abstract model of how hardware handles concurrent accesses

内存一致性 Memory Consistency



线程一致性约束
Thread consistency
constraints

Wa → Rb

Wb → Ra

■ 打印的可能值是什么？ What are the possible values printed?

- 取决于内存一致性模型 Depends on memory consistency model
- 硬件如何处理并发访问的抽象模型 Abstract model of how hardware handles concurrent accesses

■ 顺序一致性 Sequential consistency

- 就好像一次只有一个操作一样，其顺序与每个线程内的操作顺序一致
As if only one operation at a time, in an order consistent with the order of operations within each thread
- 因此，总体效果与每个单独的线程一致，但允许任意交错 Thus, overall effect consistent with each individual thread but otherwise allows an arbitrary interleaving

顺序一致性示例

Sequential Consistency Example



线程一致性约束 Thread consistency constraints

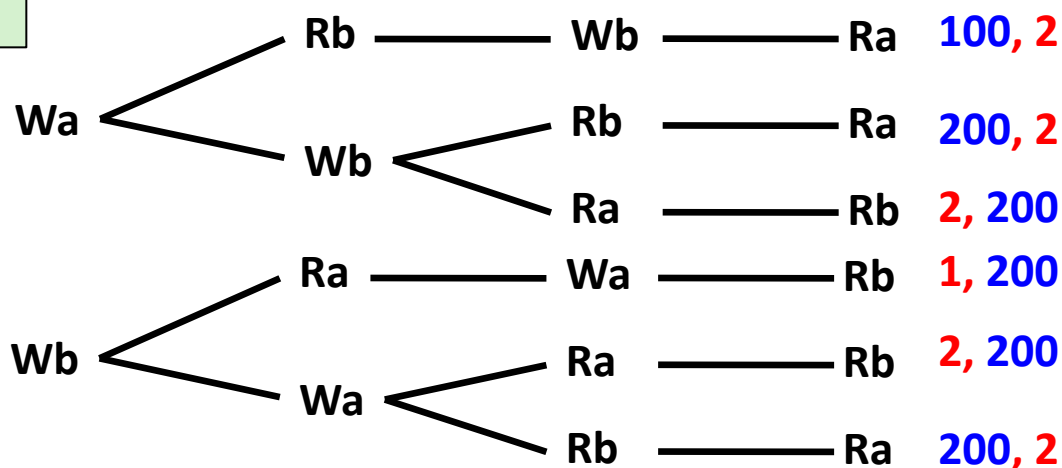
Wa ————— Rb

Wb ————— Ra

```
int a = 1;
int b = 100;
```

Thread1:
Wa: a = 2;
Rb: **print(b);**

Thread2:
Wb: b = 200;
Ra: **print(a);**



■ 不可能输出 Impossible outputs

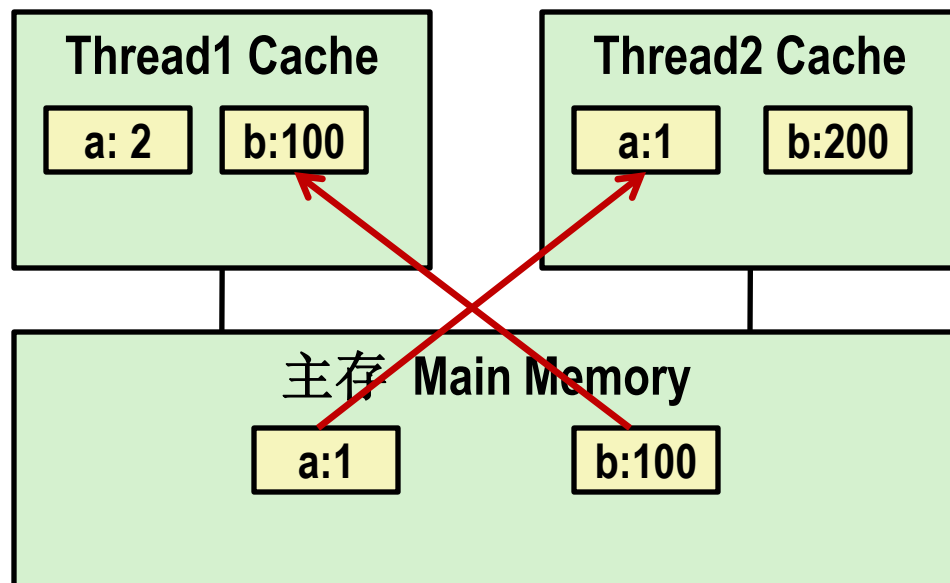
- **100, 1** and **1, 100**
- 需要在Wa或Wb之前达到Ra和Rb Would require reaching *both* Ra and Rb before *either* Wa or Wb

非一致性缓存方案

Non-Coherent Cache Scenario



- 写回缓存，线程间没有协作
Write-back caches, without coordination between them



```
int a = 1;  
int b = 100;
```

Thread1:
Wa: a = 2;
Rb: **print(b);**

Thread2:
Wb: b = 200;
Ra: **print(a);**

print 1

print 100

顺序一致性?

Sequentially consistent?

否!

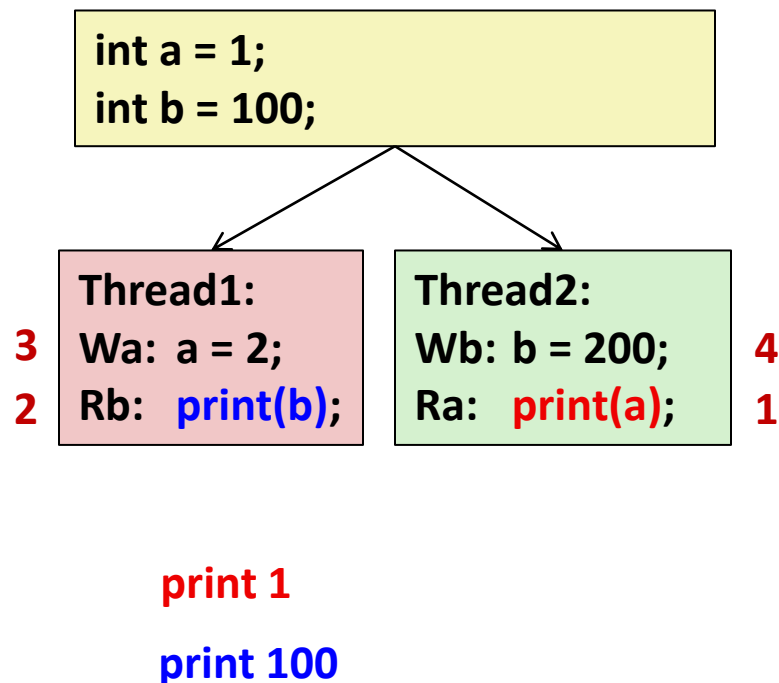
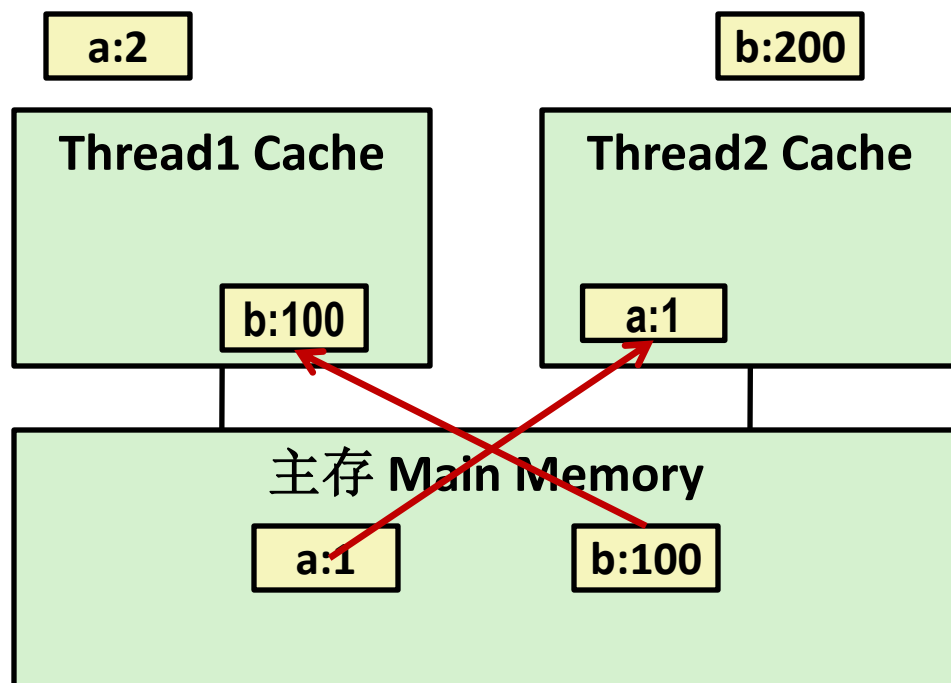
No!

非顺序一致性方案



Non-Sequentially Consistent Scenario

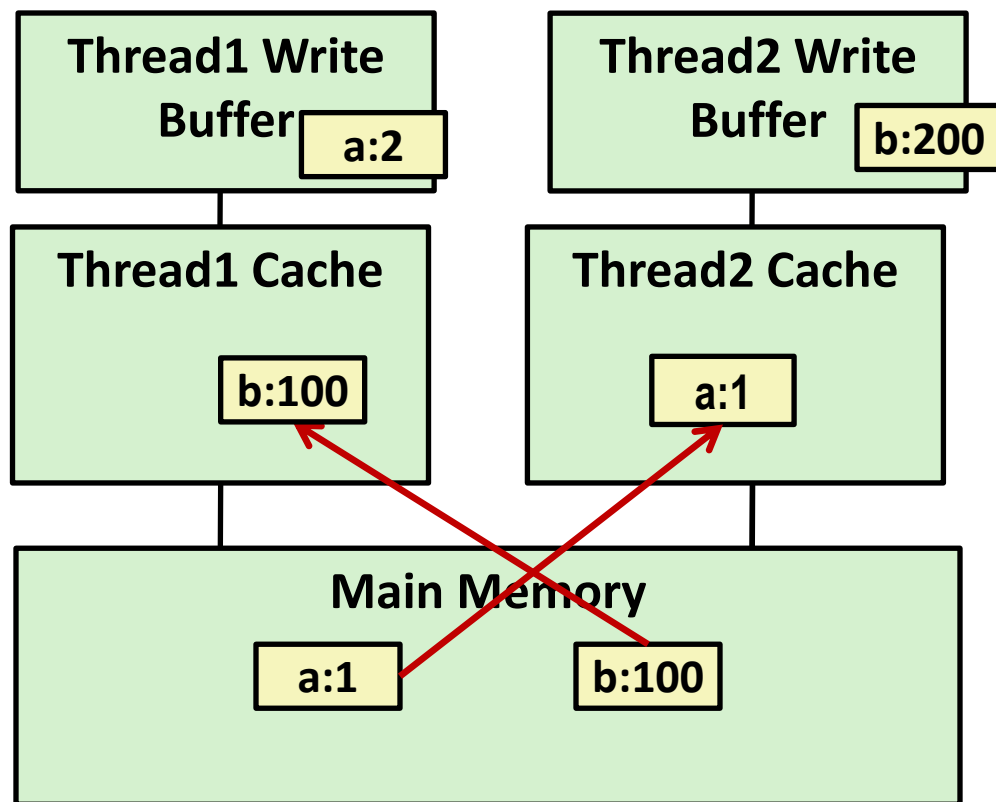
- 一致性缓存，但由于操作重新排序而违反了线程一致性约束 Coherent caches, but thread consistency constraints violated due to *operation reordering*



- 体系结构允许读取在写入之前完成，因为单个线程访问不同的内存位置 Architecture lets reads finish before writes because single thread accesses different memory locations

非顺序一致性方案

Non-Sequentially Consistent Scenario



```
int a = 1;  
int b = 100;
```

3
2
Thread1:
Wa: a = 2;
Rb: `print(b)`;

4
1
Thread2:
Wb: b = 200;
Ra: `print(a)`;

- 为什么重新排序？写入需要很长时间。缓冲区写入，让读取继续。指令级并行性 Why Reordered? Writes take long time. Buffer write, let read go ahead. *Instruction-level parallelism*

- 修复：在Wa&Rb和Wb&Ra之间添加SFENCE指令 Fix: Add SFENCE instructions between Wa & Rb and Wb & Ra



内存模型 Memory Models

- **顺序一致: Sequentially Consistent:**
 - 每个线程以正确的顺序执行, 任意交错 Each thread executes in proper order, any interleaving
- **为了确保, 需要 To ensure, requires**
 - 正确的缓存/内存行为 Proper cache/memory behavior
 - 适当的线程内排序约束 Proper intra-thread ordering constraints
- **线程排序约束 Thread ordering constraints**
 - 使用同步确保程序没有数据竞争 Use synchronization to ensure the program is free of data races

议题 Today



- 并行计算硬件 **Parallel Computing Hardware**
 - 多核 Multicore
 - 单芯片上有多个独立的处理器 Multiple separate processors on single chip
 - 超线程化 Hyperthreading
 - 在单核上高效执行多个线程 Efficient execution of multiple threads on single core
- 一致性模型 **Consistency Models**
 - 当多个线程在读/写共享状态时会发生什么情况 What happens when multiple threads are reading & writing shared state
- 线程级并行 **Thread-Level Parallelism**
 - 将程序分成独立的任务 Splitting program into independent tasks
 - 例如：并行求和 Example: Parallel summation
 - 检查一些性能小工件 Examine some performance artifacts
 - 分而治之 Divide-and conquer parallelism
 - 例如：并行快速排序 Example: Parallel quicksort



求和示例 Summation Example

- 求数字0, ..., N-1的和 **Sum numbers 0, ..., N-1**
 - 应该加起来得到 $(N-1)*N/2$ Should add up to $(N-1)*N/2$
- 分区成K个区域 **Partition into K ranges**
 - 每个区域有 $\lfloor N/K \rfloor$ 个值 $\lfloor N/K \rfloor$ values each
 - t个线程每个处理一个区域 Each of the t threads processes 1 range
 - 连续累加剩余值 Accumulate leftover values serially
- 方法#1: 所有线程更新单个全局变量 **Method #1: All threads update single global variable**
 - 1A: 无同步 1A: No synchronization
 - 1B: 用pthread信号量同步 1B: Synchronize with pthread semaphore
 - 1C: 用pthread互斥锁同步 1C: Synchronize with pthread mutex
 - “二元”信号量, 仅取值0和1 “Binary” semaphore. Only values 0 & 1

累积在单个全局变量中：声明

Accumulating in Single Global Variable: Declarations



```
typedef unsigned long data_t;  
/* Single accumulator */  
volatile data_t global_sum;
```

累积在单个全局变量中：声明

Accumulating in Single Global Variable: Declarations



```
typedef unsigned long data_t;  
/* Single accumulator */  
volatile data_t global_sum;  
  
/* Mutex & semaphore for global sum */  
sem_t semaphore;  
pthread_mutex_t mutex;
```

累积在单个全局变量中：声明

Accumulating in Single Global Variable: Declarations



```
typedef unsigned long data_t;
/* Single accumulator */
volatile data_t global_sum;

/* Mutex & semaphore for global sum */
sem_t semaphore;
pthread_mutex_t mutex;

/* Number of elements summed by each thread */
size_t nelems_per_thread;

/* Keep track of thread IDs */
pthread_t tid[MAXTHREADS];

/* Identify each thread */
int myid[MAXTHREADS];
```


累积在单个全局变量中：操作

Accumulating in Single Global Variable: Operation



```
nelems_per_thread = nelems / nthreads;
```

```
/* Set global value */
```

```
global_sum = 0;
```

```
/* Create threads and wait for them to finish */
```

```
for (i = 0; i < nthreads; i++) {
```

```
    myid[i] = i;
```

```
    Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
```

```
}
```

```
for (i = 0; i < nthreads; i++)
```

```
    Pthread_join(tid[i], NULL);
```

```
result = global_sum;
```

```
/* Add leftover elements */
```

```
for (e = nthreads * nelems_per_thread; e < nelems; e++)
```

```
    result += e;
```

线程ID Thread ID

线程例程
Thread routine

线程参数 Thread argum
(void *p)



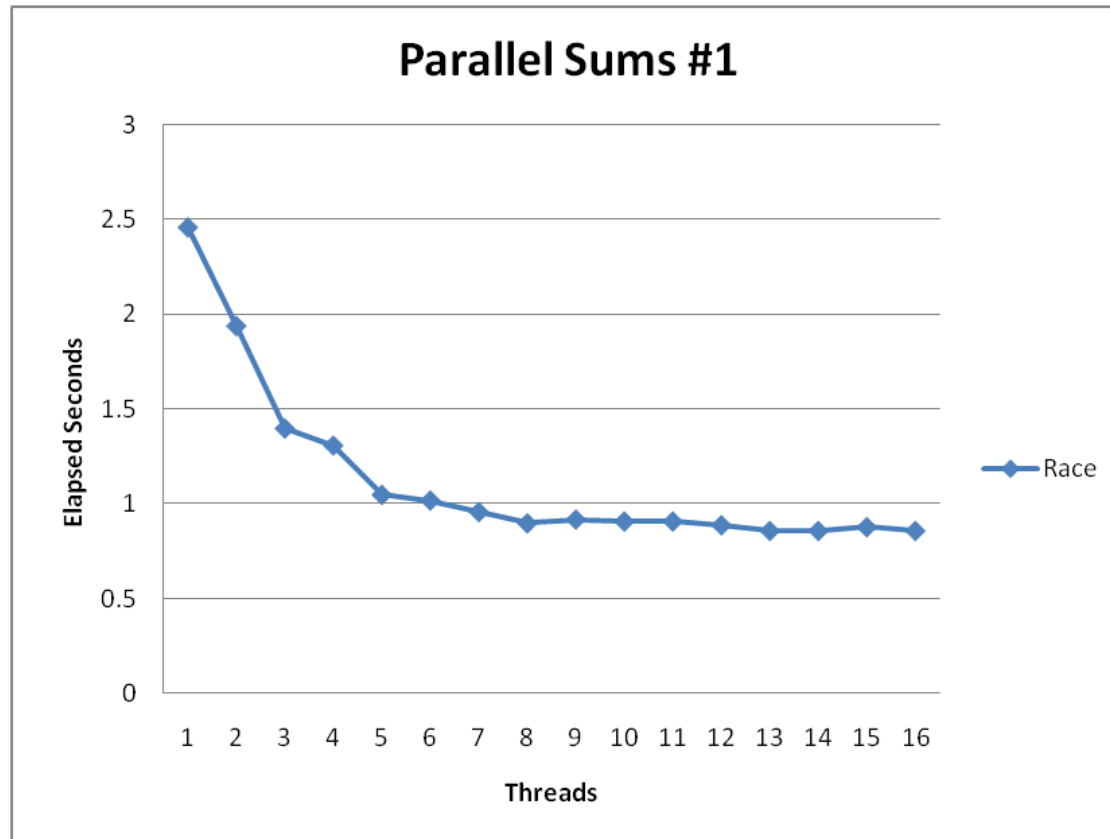
线程函数：无同步

Thread Function: No Synchronization

```
void *sum_race(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        global_sum += i;
    }
    return NULL;
}
```

无同步的性能 Unsynchronized Performance



- $N = 2^{30}$
- 最佳的加速比 Best speedup = 2.86X
- 当大于1个线程时得到错误的结果 Gets wrong answer when > 1 thread! 为何? Why?

线程函数：信号量/互斥锁

Thread Function: Semaphore / Mutex



信号量 Semaphore

```
void *sum_sem(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

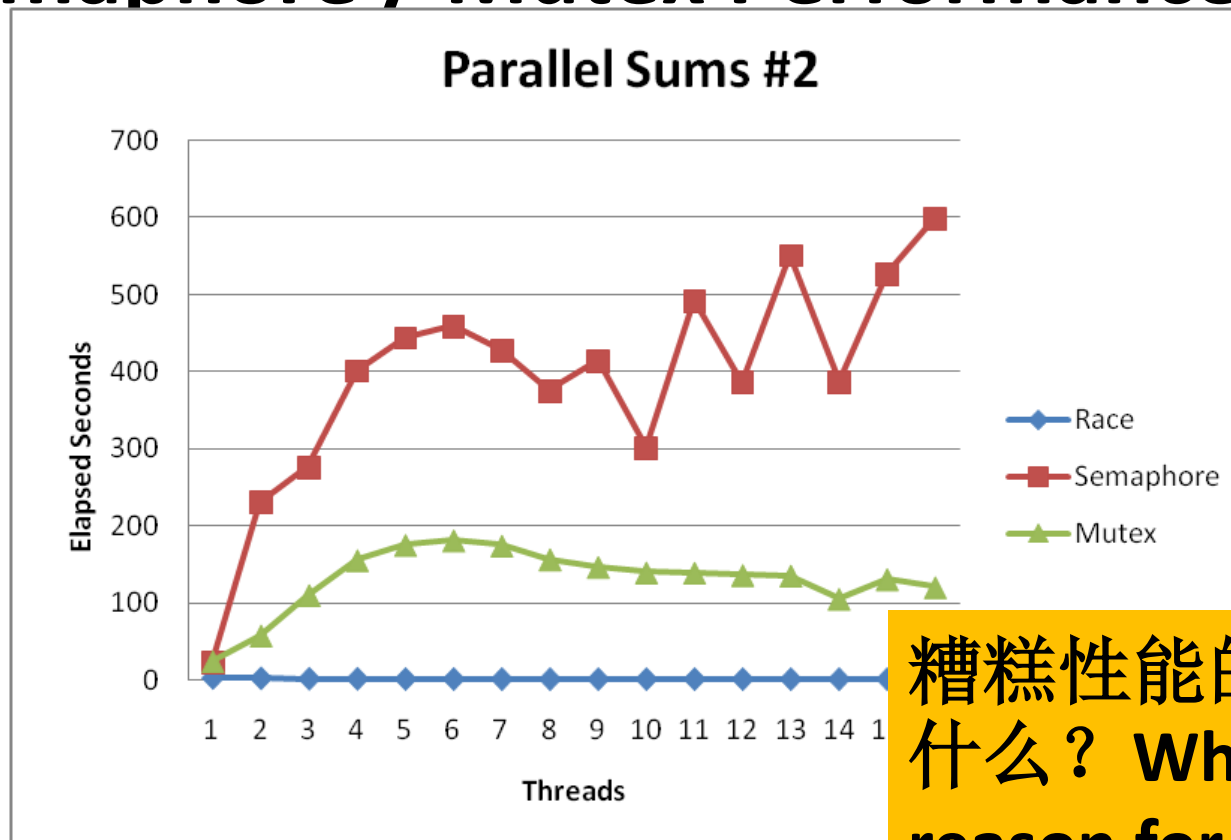
    for (i = start; i < end; i++) {
        sem_wait(&semaphore);
        global_sum += i;
        sem_post(&semaphore);
    }
    return NULL;
}
```

互斥锁 Mutex

```
pthread_mutex_lock(&mutex);
global_sum += i;
pthread_mutex_unlock(&mutex);
```

信号量/互斥锁性能

Semaphore / Mutex Performance



糟糕性能的主要原因是什么？ What is main reason for poor performance?

- 糟糕的性能 Terrible Performance
 - 2.5 seconds 秒 → ~10 minutes 分钟

- 互斥锁比信号量快3倍 Mutex 3X faster than semaphore
- 很明显，这些方法都不成功 Clearly, neither is successful

单独累积 Separate Accumulation



- **方法#2: 每个线程累积到单独的变量中 Method #2: Each thread accumulates into separate variable**
 - 2A: 在相邻数组元素中累加 2A: Accumulate in contiguous array elements
 - 2B: 在间隔开的数组元素中累加 2B: Accumulate in spaced-apart array elements
 - 2C: 在寄存器中累加 2C: Accumulate in registers

```
/* Partial sum computed by each thread */  
data_t psum[MAXTHREADS*MAXSPACING];  
  
/* Spacing between accumulators */  
size_t spacing = 1;
```

单独累积：操作

Separate Accumulation: Operation



```
nelems_per_thread = nelems / nthreads;

/* Create threads and wait for them to finish */
for (i = 0; i < nthreads; i++) {
    myid[i] = i;
    psum[i*spacing] = 0;
    Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
}
for (i = 0; i < nthreads; i++)
    Pthread_join(tid[i], NULL);

result = 0;

/* Add up the partial sums computed by each thread */
for (i = 0; i < nthreads; i++)
    result += psum[i*spacing];

/* Add leftover elements */
for (e = nthreads * nelems_per_thread; e < nelems; e++)
    result += e;
```

线程函数：内存累积

Thread Function: Memory Accumulation



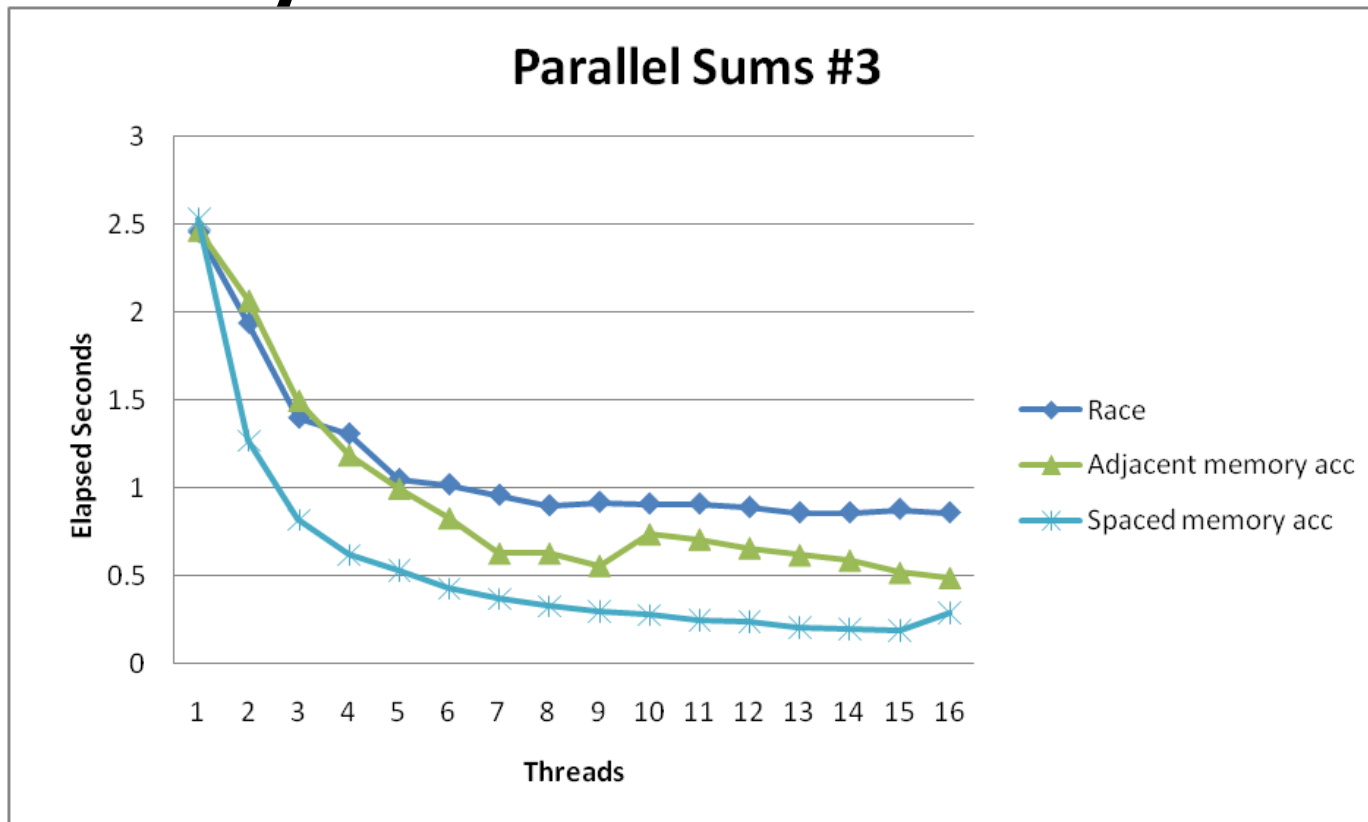
互斥锁在哪？ Where is the mutex?

```
void *sum_global(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    size_t index = myid*spacing;
    psum[index] = 0;
    for (i = start; i < end; i++) {
        psum[index] += i;
    }
    return NULL;
}
```


内存累积性能

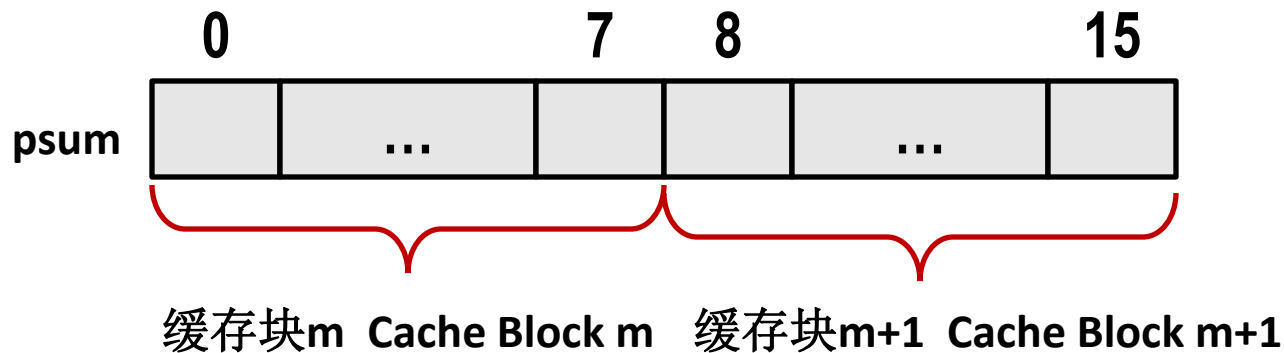
Memory Accumulation Performance



- 单独线程累积的优势 **Clear threading advantage**
 - 连续累积加速比: Adjacent speedup: 5 X
 - 间隔累积加速比: Spaced-apart speedup: 13.3 X (仅观察到加速比大于8
Only observed speedup > 8)
- 为什么进行间隔开累加性能更佳? **Why does spacing the accumulators apart matter?**



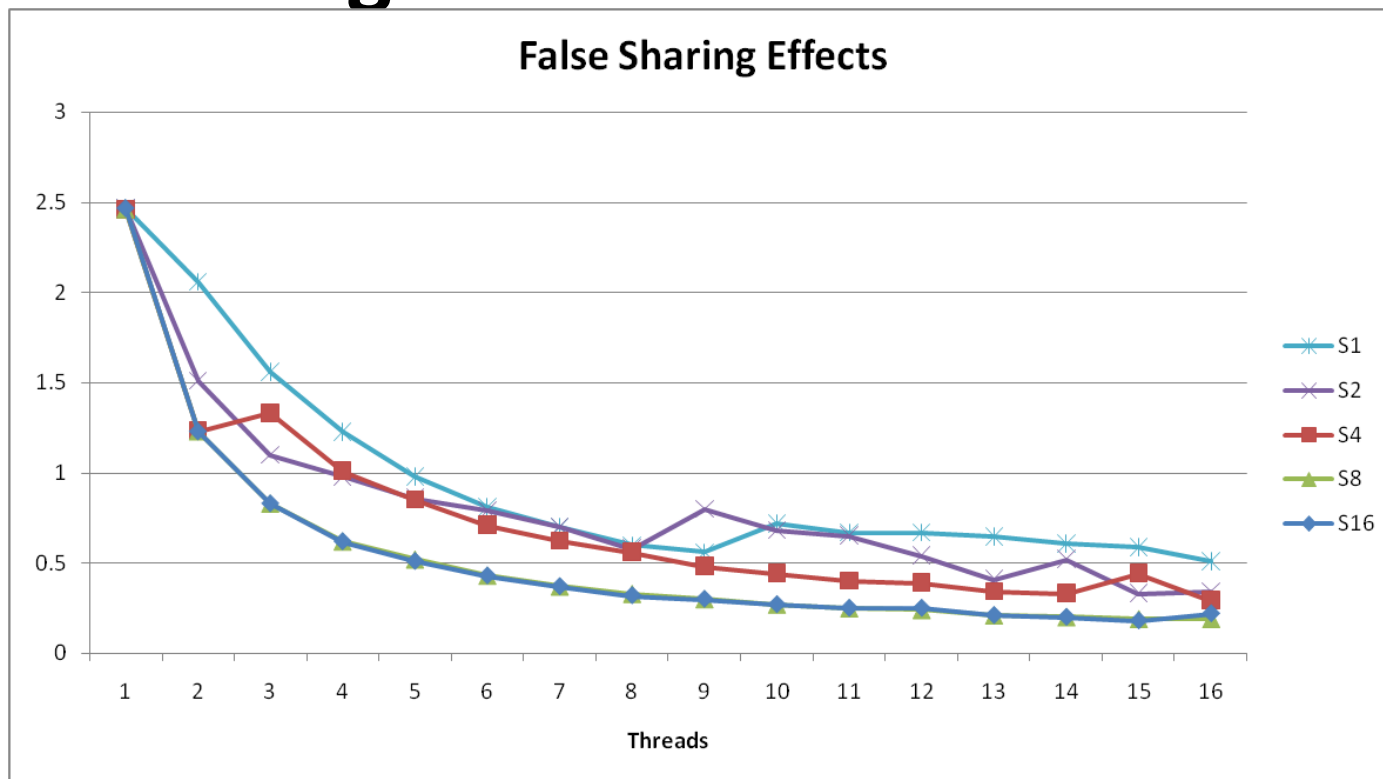
虚假共享 False Sharing



- 缓存块上保持一致性 Coherence maintained on cache blocks
- 要更新`psum[i]`，线程`i`必须具有独占访问权限 To update `psum[i]`, thread `i` must have exclusive access
 - 共享公共缓存块的线程将继续为访问块而相互争斗 Threads sharing common cache block will keep fighting each other for access to block

虚假共享的性能

False Sharing Performance



- 最佳间隔性能比最佳相邻性能高2.8倍 Best spaced-apart performance 2.8 X better than best adjacent
- 演示缓存块大小为64 Demonstrates cache block size = 64
 - 8字节值 8-byte values
 - 将间隔增加到8以上没有性能改善 No benefit increasing spacing beyond 8

线程函数：寄存器累积

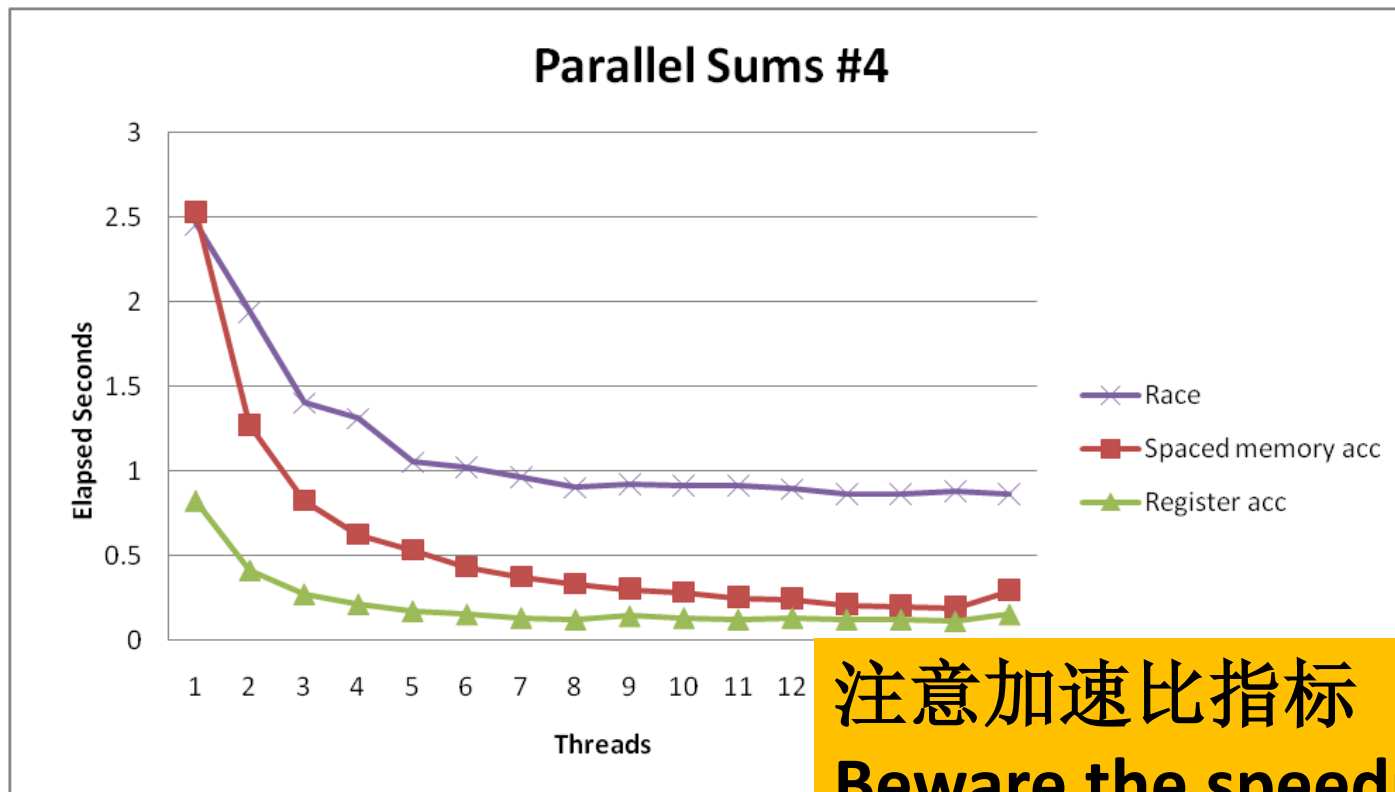
Thread Function: Register Accumulation



```
void *sum_local(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;
    size_t index = myid*spacing;
    data_t sum = 0;
    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[index] = sum;
    return NULL;
}
```

寄存器累积性能

Register Accumulation Performance



注意加速比指标

Beware the speedup metric!

- 单独线程累积优势 Clear threading advantage
 - 加速比/Speedup = 7.5 X
- 比最快的内存累积好2倍 2X better than fastest memory accumulation



经验教训 Lessons learned

- 共享内存可能开销很高 **Sharing memory can be expensive**
 - 关注真实共享 Pay attention to true sharing
 - 注意虚假共享 Pay attention to false sharing
- 尽可能使用寄存器 **Use registers whenever possible**
 - (记住cachelab Remember cachelab)
 - 尽可能使用本地缓存 Use local cache whenever possible
- 处理剩余的数据 **Deal with leftovers**
- 在检查性能时，与最佳顺序实现进行比较 **When examining performance, compare to best possible sequential implementation**

更重要的示例：排序

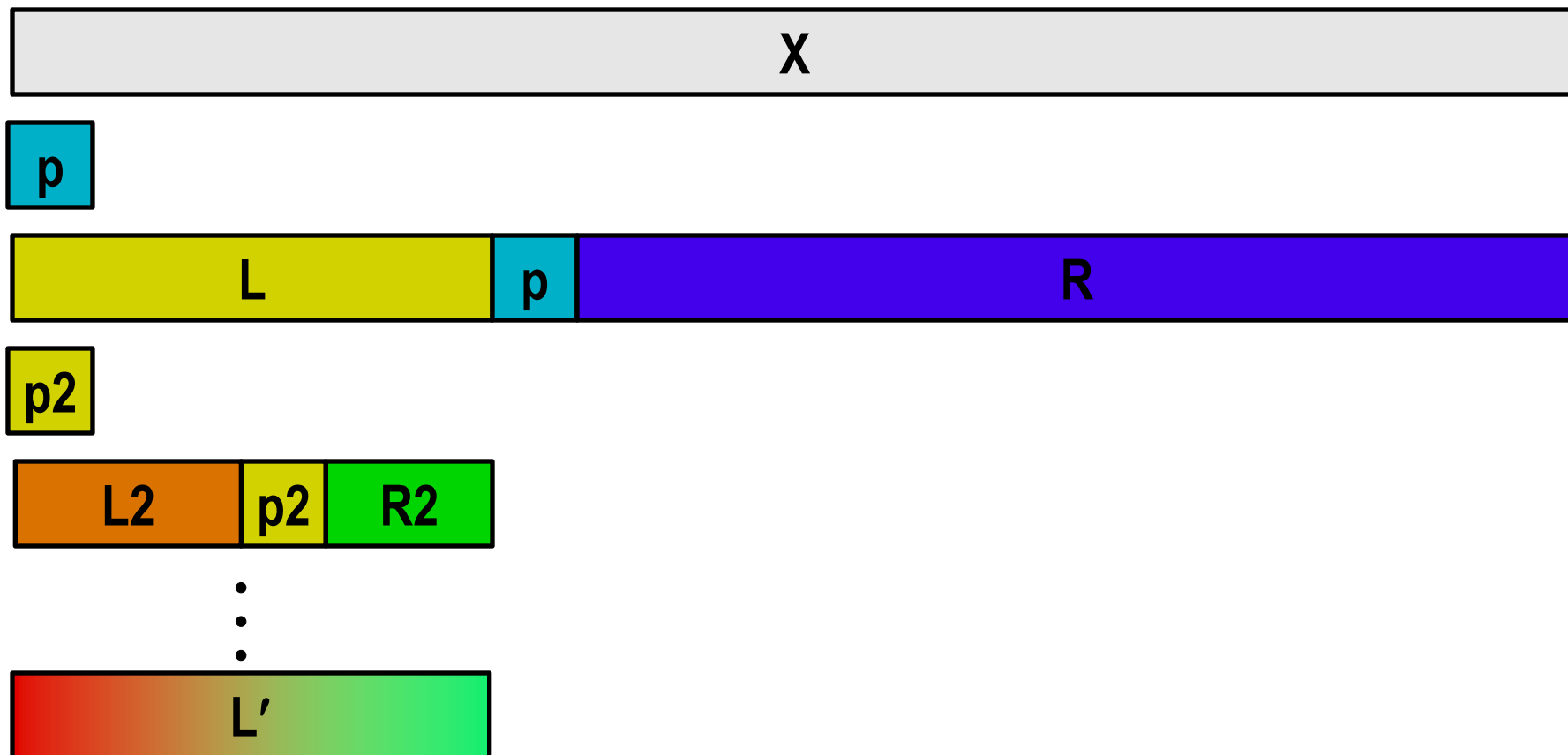
A More Substantial Example: Sort



- **N个随机数集合排序** Sort set of N random numbers
- **多种可能的算法** Multiple possible algorithms
 - 使用并行版本的快速排序 Use parallel version of quicksort
- **对X集合进行顺序快速排序** Sequential quicksort of set of values X
 - 从X选择“中心点”p Choose “pivot” p from X
 - 重新排列X Rearrange X into
 - 左边集合：值小于等于p L: Values \leq p
 - 右边集合：值大于等于p R: Values \geq p
 - 对左边集合进行递归排序得到L' Recursively sort L to get L'
 - 对右边集合进行递归排序得到R' Recursively sort R to get R'
 - 返回 Return L' : p : R'

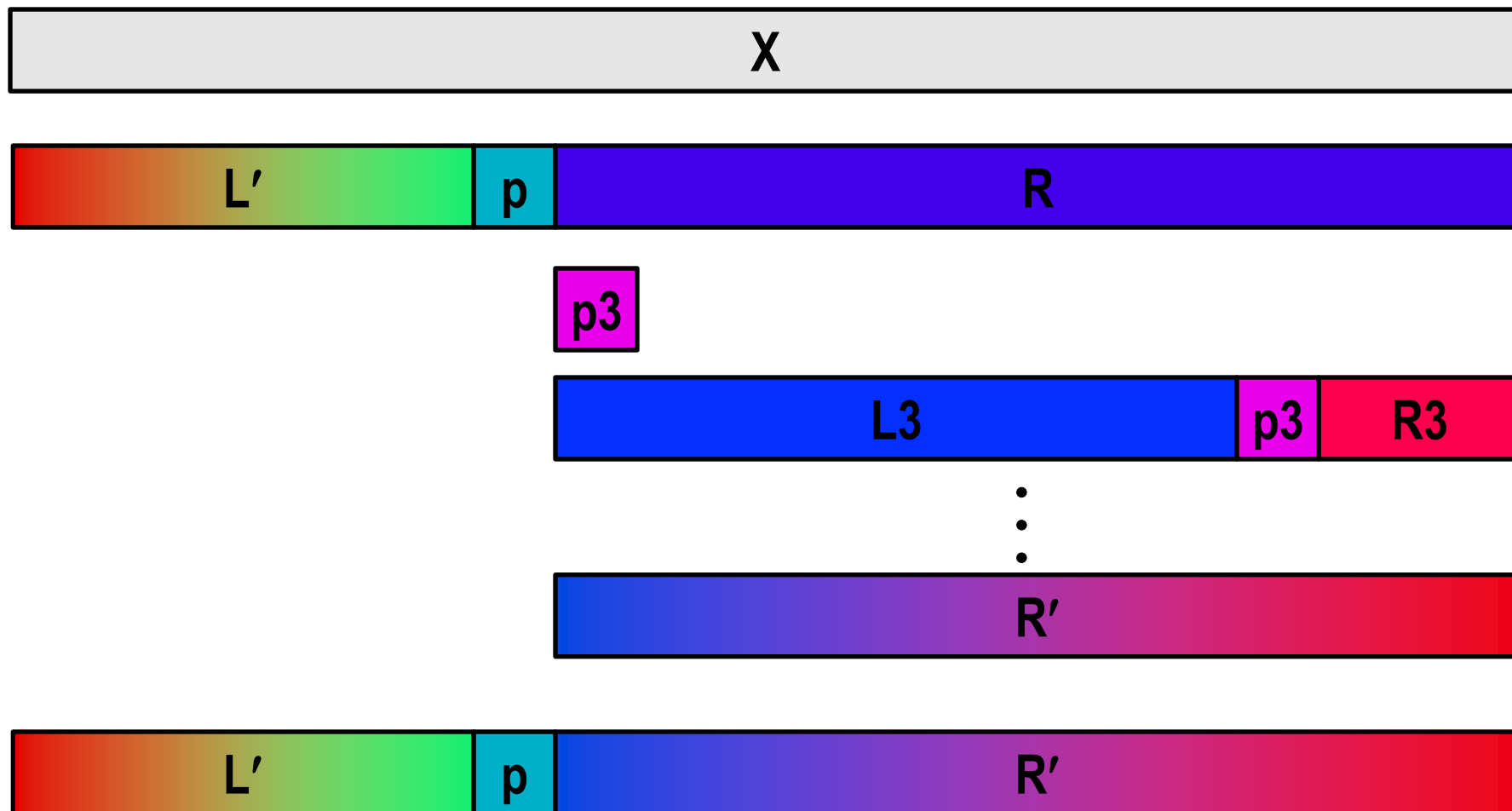
顺序快速排序可视化

Sequential Quicksort Visualized



顺序快速排序可视化

Sequential Quicksort Visualized



顺序快速排序代码

Sequential Quicksort Code



```
void qsort_serial(data_t *base, size_t nele) {
    if (nele <= 1)
        return;
    if (nele == 2) {
        if (base[0] > base[1])
            swap(base, base+1);
        return;
    }

    /* Partition returns index of pivot */
    size_t m = partition(base, nele);
    if (m > 1)
        qsort_serial(base, m);
    if (nele-1 > m+1)
        qsort_serial(base+m+1, nele-m-1);
}
```

- 从base开始对nele个元素排序 Sort nele elements starting at base
 - 如果有多于一个元素，则递归排序L或R Recursively sort L or R if has more than one element

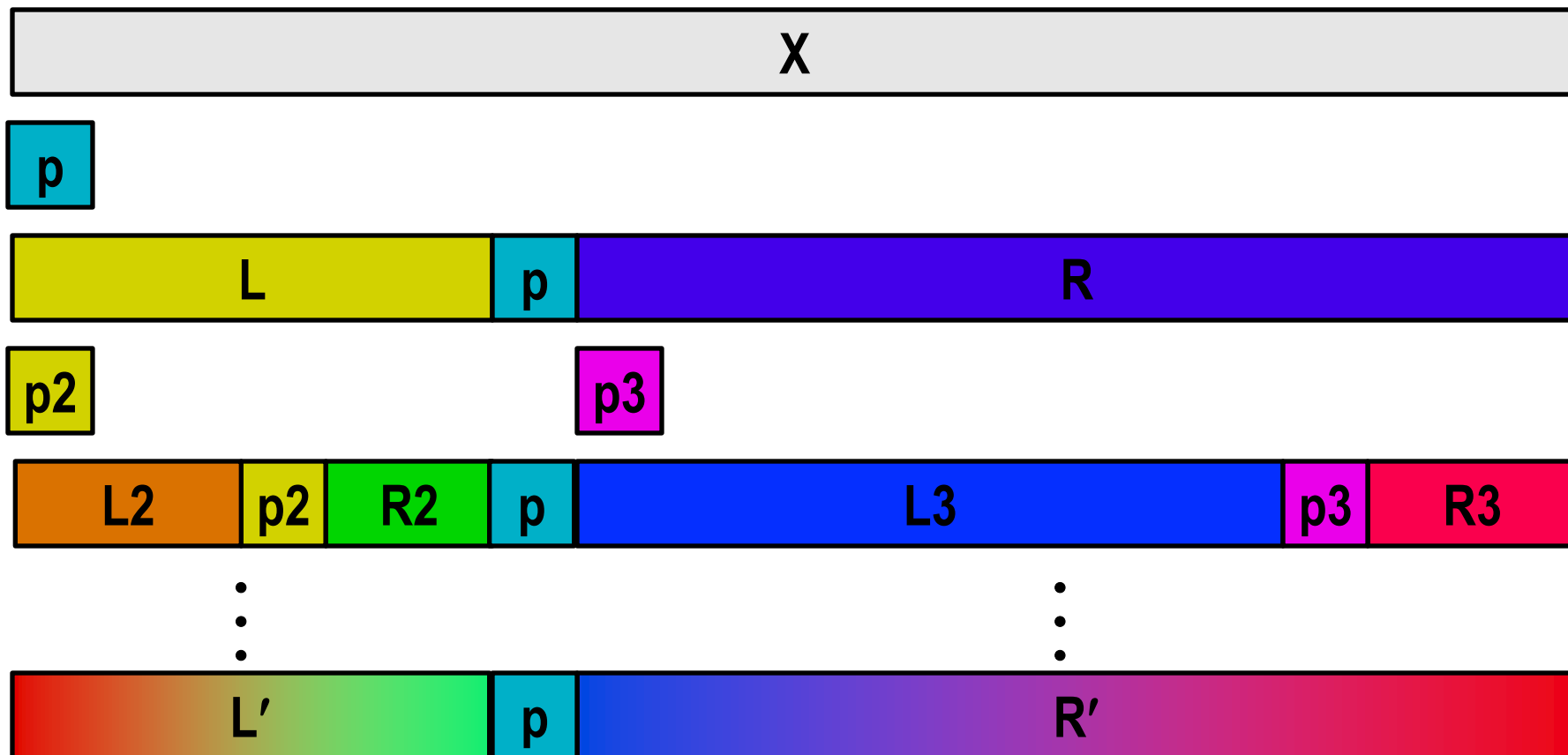
并行快速排序 Parallel Quicksort



- 集合X的并行快速排序 **Parallel quicksort of set of values X**
 - 如果N小于等于Nthresh, 执行顺序快速排序 If $N \leq N_{thresh}$, do sequential quicksort
 - 否则 Else
 - 从X选择“中心点”p Choose “pivot” p from X
 - 重新排列X Rearrange X into
 - 左集合: 值小于等于p L: Values $\leq p$
 - 右集合: 值大于等于p R: Values $\geq p$
 - 递归生成单独的线程 Recursively spawn separate threads
 - 排序L以获得L' Sort L to get L'
 - 排序R以获得R' Sort R to get R'
 - 返回 Return L' : p : R'

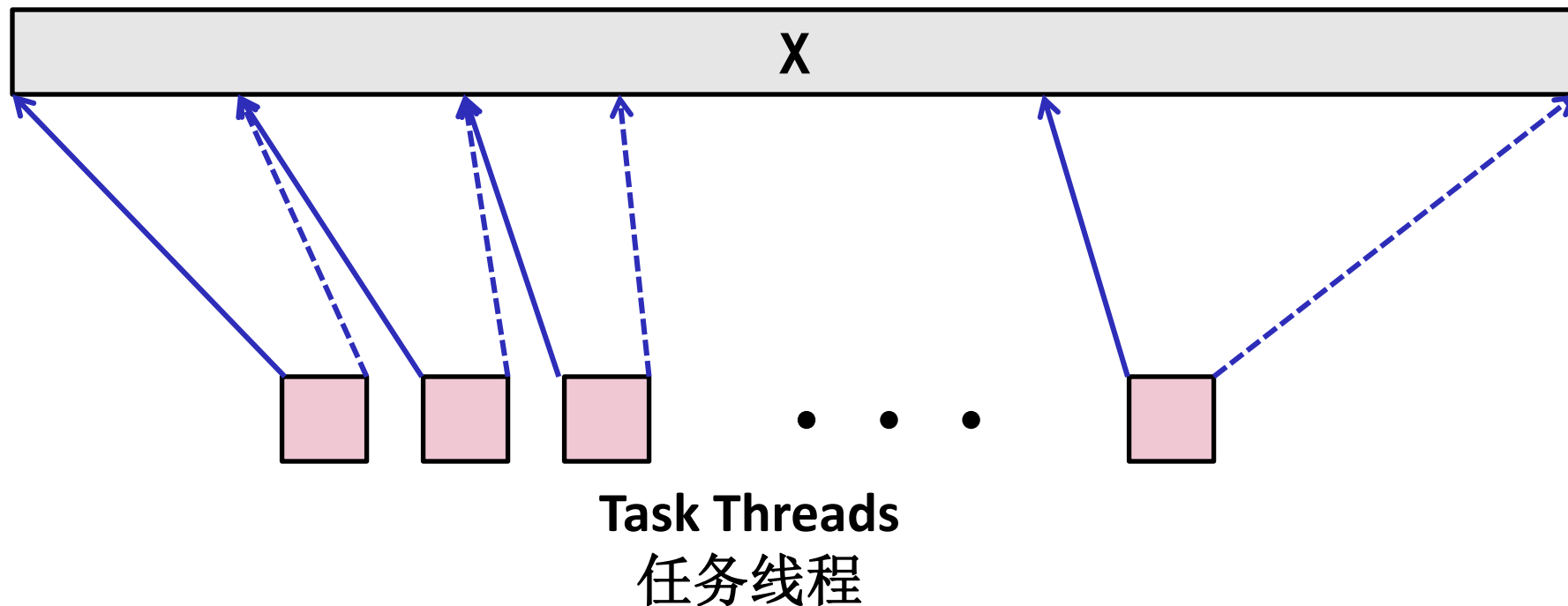
并行快速排序可视化

Parallel Quicksort Visualized



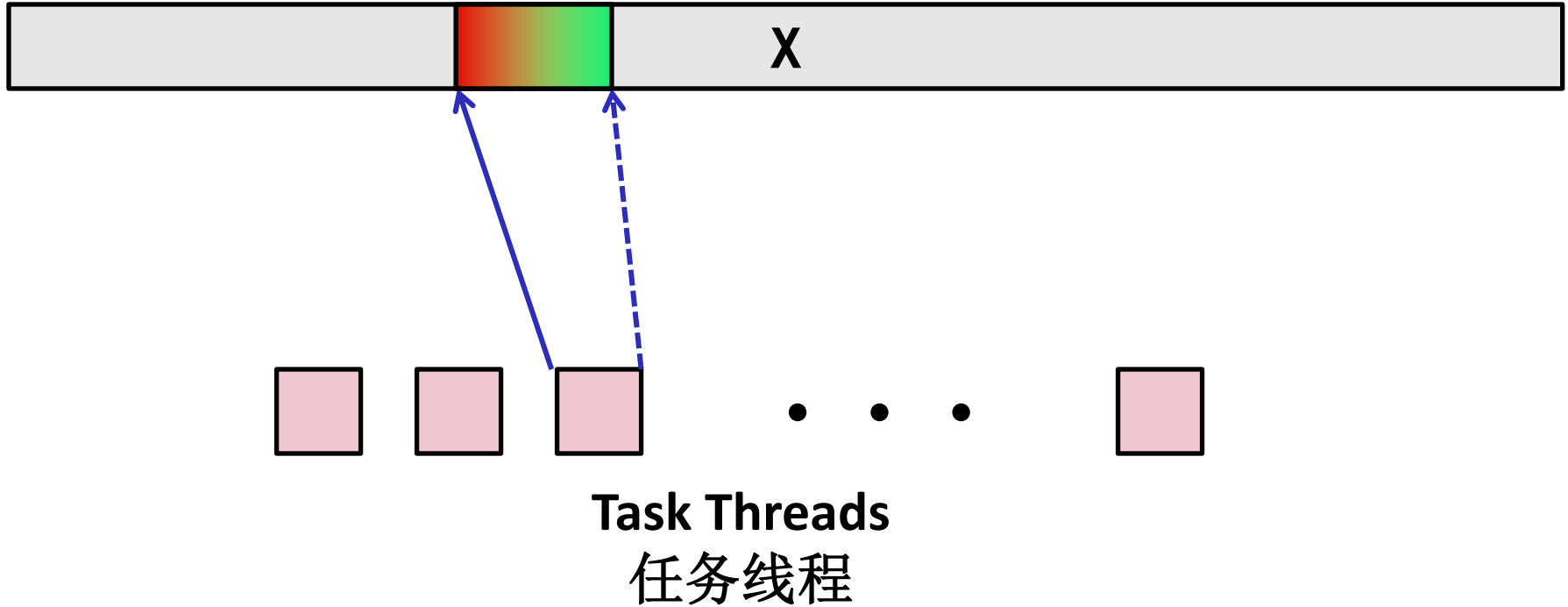
线程结构：排序任务

Thread Structure: Sorting Tasks



- 任务：排序子范围数据 **Task: Sort subrange of data**
 - 指定为： Specify as:
 - base：起始地址 **base**: Starting address
 - nele：子范围中的元素数 **nele**: Number of elements in subrange
- 作为单独线程运行 **Run as separate thread**

小排序任务操作 Small Sort Task Operation



- 排序子范围数据使用串行快速排序 Sort subrange using serial quicksort

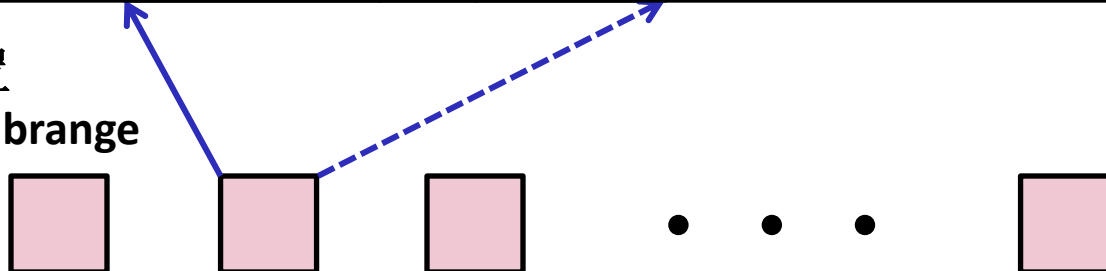
大排序任务操作

Large Sort Task Operation



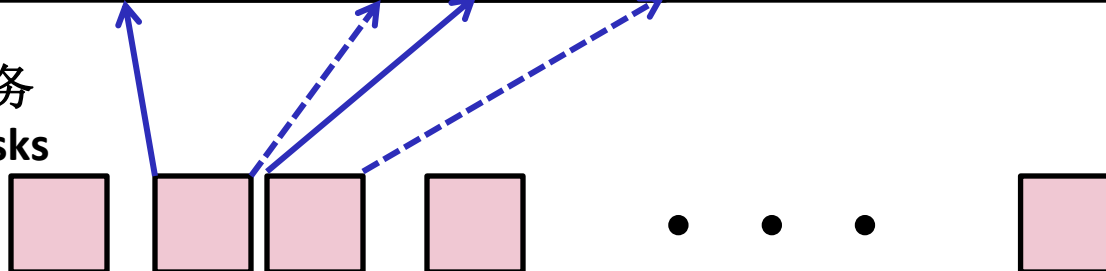
子范围位置

Partition Subrange



生成2个任务

Spawn 2 tasks





顶层函数（简化）

Top-Level Function (Simplified)

```
void tqsort(data_t *base, size_t nele) {  
    init_task(nele);  
    global_base = base;  
    global_end = global_base + nele - 1;  
    task_queue_ptr tq = new_task_queue();  
    tqsort_helper(base, nele, tq);  
    join_tasks(tq);  
    free_task_queue(tq);  
}
```

- 初始化数据结构 Sets up data structures
- 调用递归排序例程 Calls recursive sort routine
- 保持加入线程，直到没有剩余 Keeps joining threads until none left
- 释放数据结构 Frees data structures

递归排序例程（简化）

Recursive sort routine (Simplified)



```
/* Multi-threaded quicksort */
static void tqsort_helper(data_t *base, size_t nele,
                           task_queue_ptr tq) {
    if (nele <= nele_max_sort_serial) {
        /* Use sequential sort */
        qsort_serial(base, nele);
        return;
    }
    sort_task_t *t = new_task(base, nele, tq);
    spawn_task(tq, sort_thread, (void *) t);
}
```

- 小分区：按顺序排序 Small partition: Sort serially
- 大分区：生成新的排序任务 Large partition: Spawn new sort task

排序任务线程（简化）

Sort task thread (Simplified)

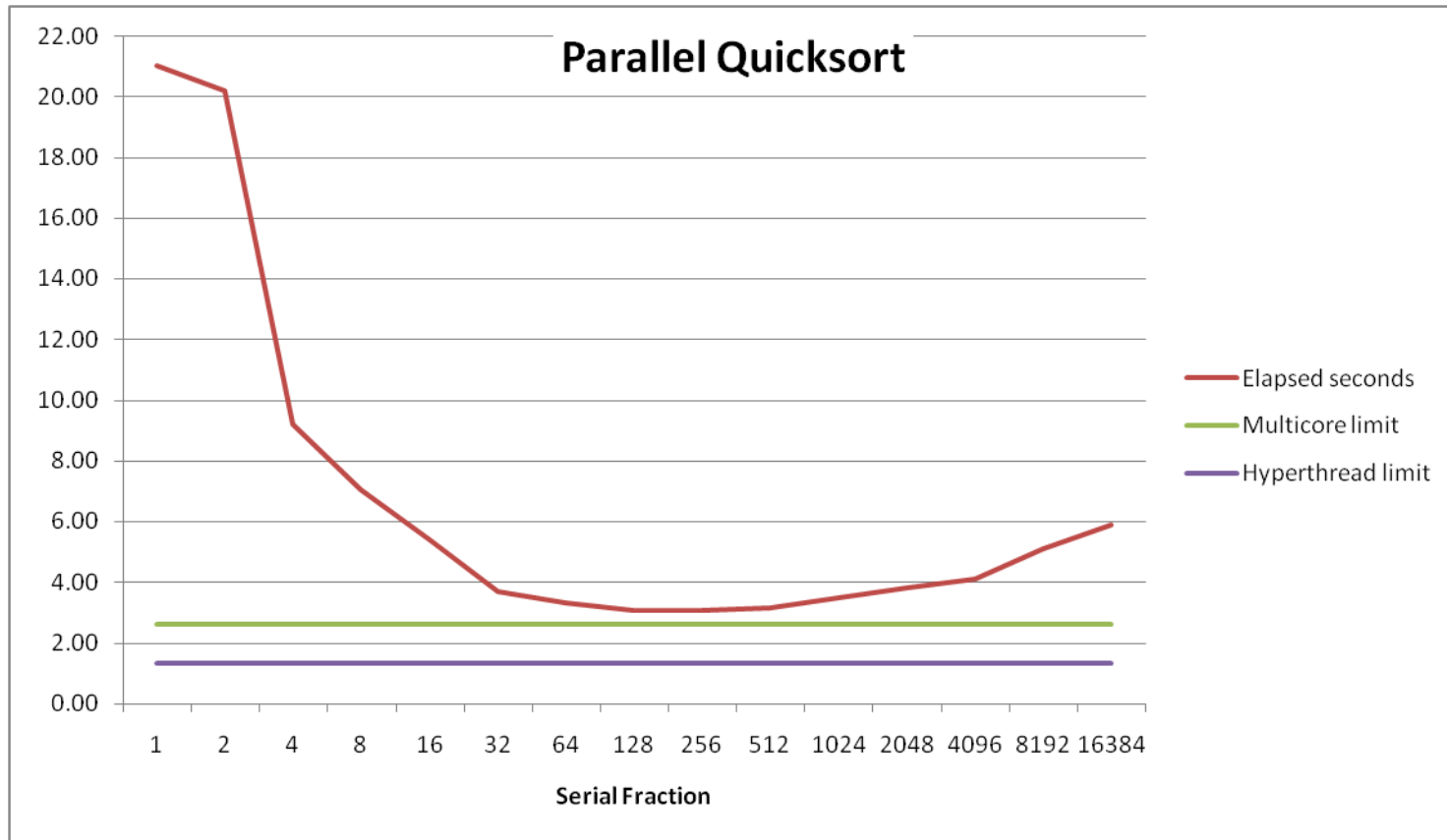


```
/* Thread routine for many-threaded quicksort */
static void *sort_thread(void *vargp) {
    sort_task_t *t = (sort_task_t *) vargp;
    data_t *base = t->base;
    size_t nele = t->nele;
    task_queue_ptr tq = t->tq;
    free(vargp);
    size_t m = partition(base, nele);
    if (m > 1)
        tqsort_helper(base, m, tq);
    if (nele-1 > m+1)
        tqsort_helper(base+m+1, nele-m-1, tq);
    return NULL;
}
```

- 获取任务参数 Get task parameters
- 执行分区步骤 Perform partitioning step
- 在每个分区上调用递归排序例程（如果部分大小大于1）
Call recursive sort routine on each partition (if size of part > 1)

并行快速排序性能

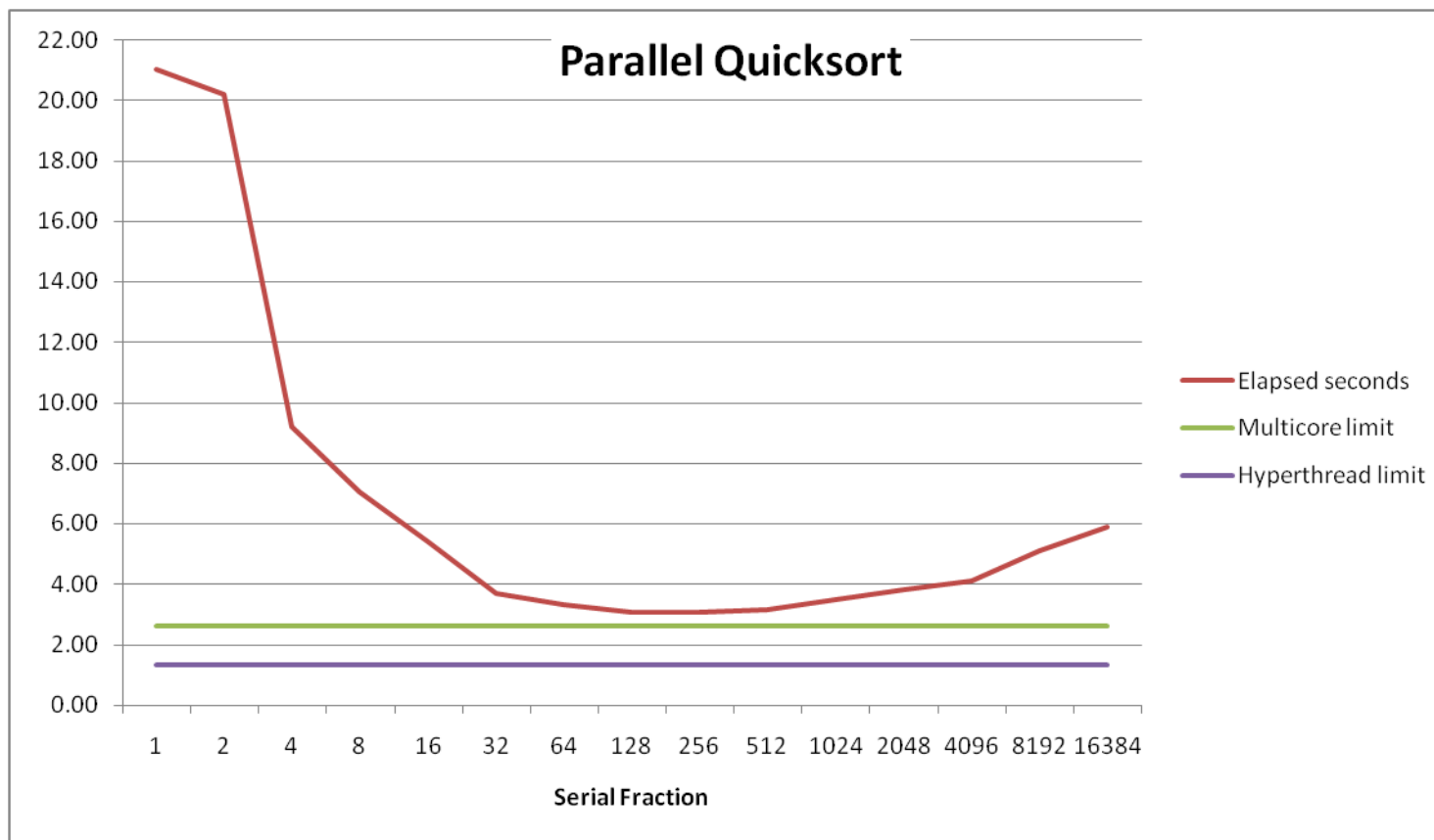
Parallel Quicksort Performance



- 串行比例：进行串行排序的输入比例 **Serial fraction: Fraction of input at which do serial sort**
- 排序128M随机值 **Sort 2^{27} (134,217,728) random values**
- 最佳加速比 **Best speedup = 6.84X**

并行快速排序性能

Parallel Quicksort Performance



■ 在广泛的串行比例范围内表现良好 Good performance over wide range of fraction values

- F太小：并行度不够 F too small: Not enough parallelism
- F太大：线程开销太高 F too large: Thread overhead too high

阿姆达尔定律（旅行模拟）

Amdahl's Law (Travel Analogy)



加速比
Speed-Up
1

- 从PIT直飞LHR Flying jet non-stop from PIT -> LHR: 7.5 Hours
- 或者，老式SST方式: Or, old fashioned SST way:
 - Fly jet from PIT -> JFK: 1.5 Hours
 - Fly SST from JFK -> LHR: 3.5 Hours 5 Hours 1.5x
- 或者，使用FTL Or, Using FTL:
 - Fly jet from PIT -> JFK: 1.5 Hours
 - Fly FTL from JFK -> LHR: .01 Hours 1.51 Hours ~5x
- 最好的加速比是5倍，即使是FTL，因为必须到达纽约 Best possible speed up is 5X, even with FTL because have to get to New York.
- PIT: 匹兹堡 LHR: 伦敦 JFK: 纽约
- SST: 超音速客机 FTL: 超光速



阿姆达尔定律 Amdahl's Law

■ 总体问题 Overall problem

- T所需的总顺序执行时间 T Total sequential time required
- p可加速的总比例 p Fraction of total that can be sped up ($0 \leq p \leq 1$)
- k加速系数 k Speedup factor

■ 最终性能 Resulting Performance

- $T_k = pT/k + (1-p)T$
 - 可以加速的部分速度快k倍 Portion which can be sped up runs k times faster
 - 无法加速的部分保持不变 Portion which cannot be sped up stays the same
- 最大可能加速比 Maximum possible speedup
 - $k = \infty$
 - $T_\infty = (1-p)T$

阿姆达尔定律（旅行模拟）

Amdahl's Law (Travel Analogy)



- 从PIT直飞LHR Flying jet non-stop from PIT -> LHR: 7.5 Hours
- 或者，老式SST方式: Or, old fashioned SST way:
 - Fly jet from PIT -> JFK: 1.5 Hours
 - Fly SST from JFK -> LHR: 3.5 Hours 5 Hours 1.5x
- 或者，使用FTL Or, Using FTL:
 - Fly jet from PIT -> JFK: 1.5 Hours
 - Fly FTL from JFK -> LHR: .01 Hours 1.51 Hours ~5x
- 最好的加速比是5倍，即使是FTL，因为必须到达纽约 Best possible speed up is 5X, even with FTL because have to get to New York.
 - $T=7.5, p=6/7.5=.8, k=\infty \Rightarrow T_{\infty} = (1-p)T=1.5$ 最大加速比
max speed-up =5x

阿姆达尔定律的示例

Amdahl's Law Example



■ 总体问题 Overall problem

- $T = 10$ Total time required 所需总时间
- $p = 0.9$ Fraction of total which can be sped up 可加速的总比例
- $k = 9$ Speedup factor 加速系数

■ 最终性能 Resulting Performance

- $T_g = 0.9 * 10/9 + 0.1 * 10 = 1.0 + 1.0 = 2.0$ (5倍加速比 a 5x speedup)

■ 最大可能加速比 Maximum possible speedup

- $T_\infty = 0.1 * 10.0 = 1.0$ (10倍加速比 a 10x speedup)
 - 拥有**无限的**并行计算资源! With **infinite** parallel computing resources!
- 极限加速比显示**算法**极限 Limit speedup shows **algorithmic** limitation

阿姆达尔定律和并行快速排序

Amdahl's Law & Parallel Quicksort



■ 顺序程序瓶颈 Sequential bottleneck

- 顶层分区：无加速 Top-level partition: No speedup
- 第二级：小于等于2倍加速比 Second level: $\leq 2X$ speedup
- 第k级：小于等于 2^{k-1} 加速比 k^{th} level: $\leq 2^{k-1}X$ speedup

■ 启示 Implications

- 小规模并行的良好性能 Good performance for small-scale parallelism
- 需要并行化分区步骤以获得大规模并行性 Would need to parallelize partitioning step to get large-scale parallelism
 - 基于规则抽样的并行排序 Parallel Sorting by Regular Sampling
 - “并行与分布式计算” H. Shi & J. Schaeffer, J. Parallel & Distributed Computing, 1992

经验教训 Lessons Learned



- **必须具有并行化策略 Must have parallelization strategy**
 - 划分为K个独立部分 Partition into K independent parts
 - 分而治之 Divide-and-conquer
- **内部循环必须无同步 Inner loops must be synchronization free**
 - 同步操作非常耗时 Synchronization operations very expensive
- **当心硬件瑕疵 Watch out for hardware artifacts**
 - 需要了解处理器和内存结构 Need to understand processor & memory structure
 - 共享和虚假共享全局数据 Sharing and false sharing of global data
- **当心阿姆达尔定律 Beware of Amdahl's Law**
 - 串行代码可能成为瓶颈 Serial code can become bottleneck
- **你能行！ You can do it!**
 - 实现适度的并行性并不困难 Achieving modest levels of parallelism is not difficult
 - 建立实验框架并测试多种策略 Set up experimental framework and test multiple strategies