Thread-Level Parallelism
线程级并行

100076202：计算机系统导论

任课教师：

计卫星　　宿红毅　　张艳

原作者：

Randal E. Bryant and David R. O'Hallaron

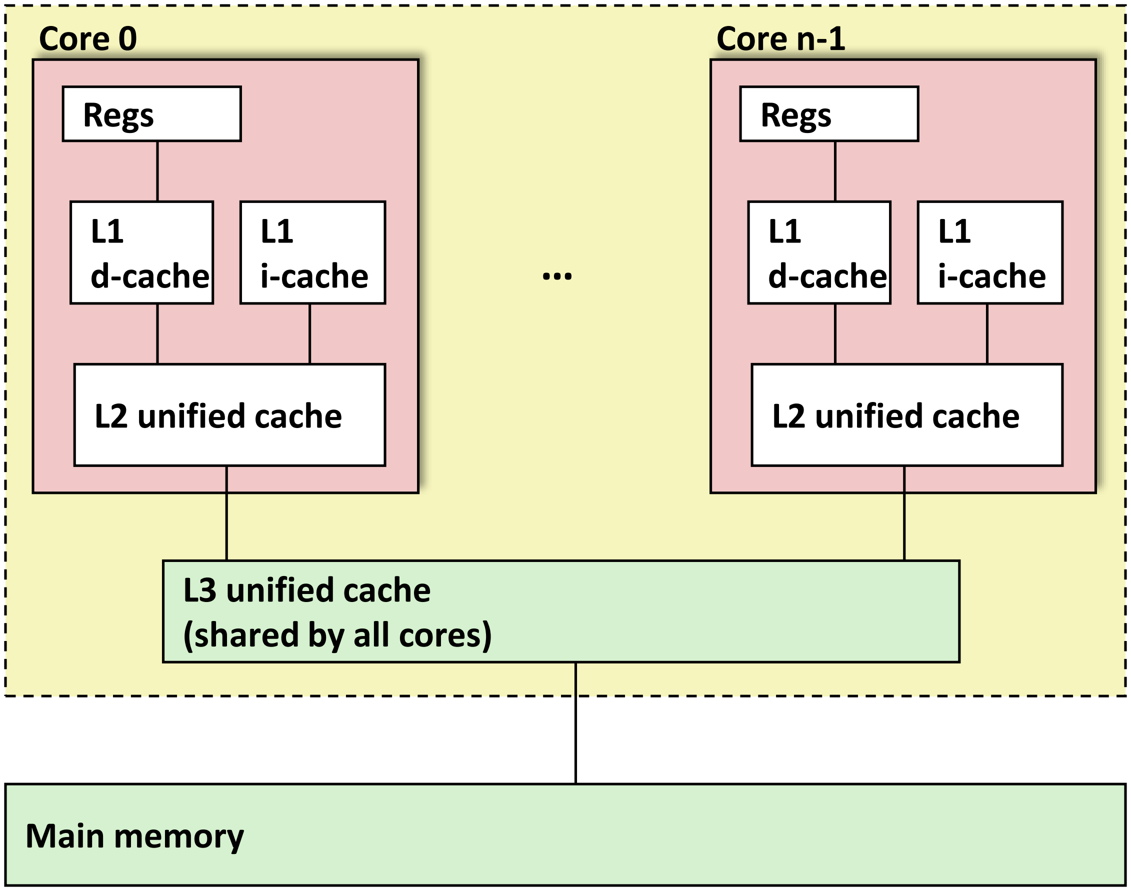
内容提纲/Today

- 并行计算硬件/**Parallel Computing Hardware**
  - 多核/Multicore
    - 一个芯片上多个独立的处理器/Multiple separate processors on single chip
  - 超线程/Hyperthreading
    - 单个核上执行多个线程/Efficient execution of multiple threads on single core
- 线程级并行/**Thread-Level Parallelism**
  - 将程序分为多个独立的任务/Splitting program into independent tasks
    - Example 1: Parallel summation
  - 分治并行/Divide-and conquer parallelism
    - Example 2: Parallel quicksort
- 一致性模型/**Consistency Models**
  - 当多个线程读写共享状态时会发生什么/What happens when multiple threads are reading & writing shared state

挖掘并行执行/Exploiting parallel execution

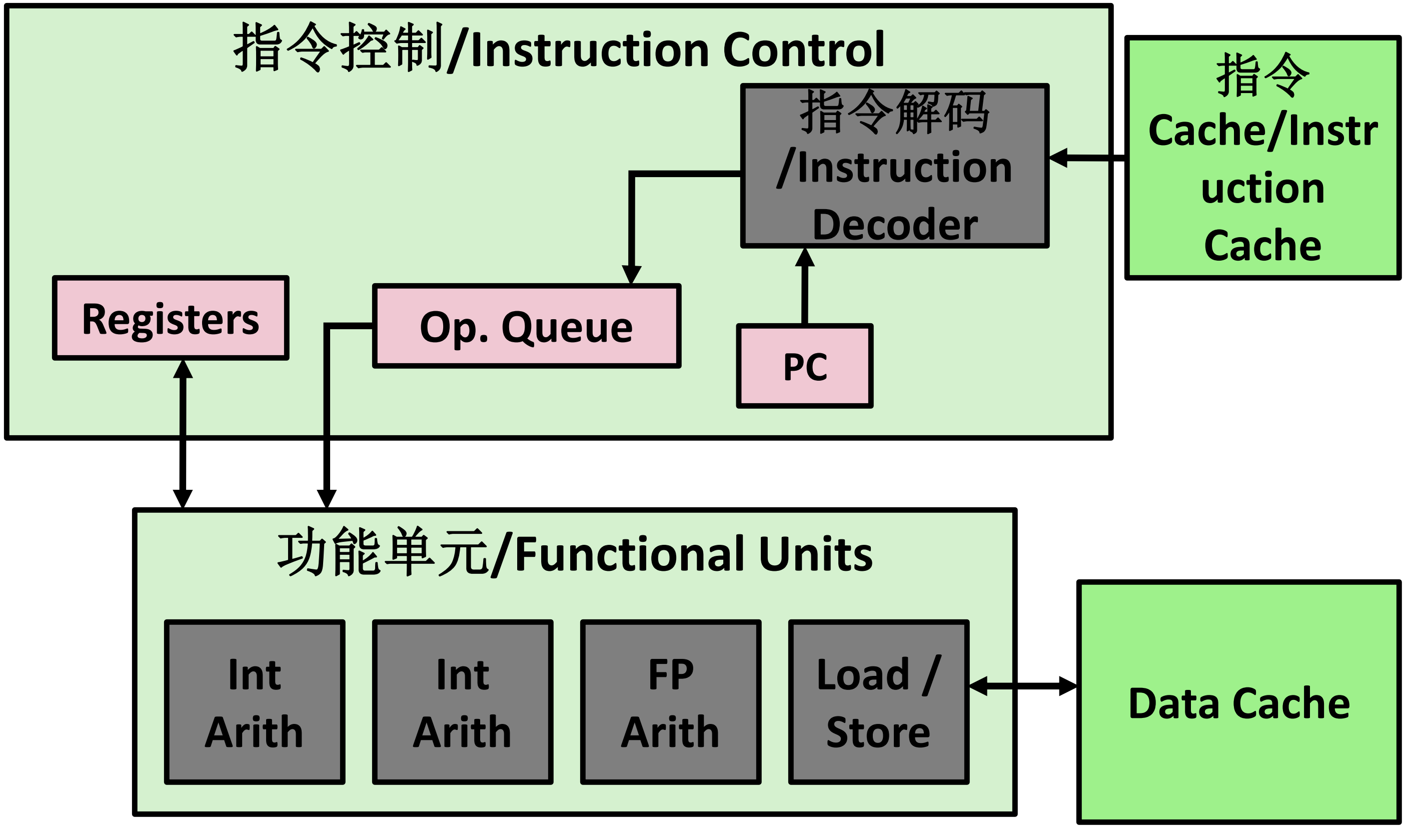
- **目前我们还是在用线程处理I/O延迟/So far, we've used threads to deal with I/O delays**
  - 例如为每个客户端设置一个线程以防止互相延迟/e.g., one thread per client to prevent one from delaying another
- **多核处理器/超线程CPU提供了其他的可能/Multi-core/Hyperthreaded CPUs offer another opportunity**
  - 将任务分布给线程并行执行/Spread work over threads executing in parallel
  - 如果有很多独立的任务则自动实现/Happens automatically, if many independent tasks
    - 例如有很多程序或者服务很多客户端/e.g., running many applications or serving many clients
  - 也可以编写代码实现一个任务的加速运行/Can also write code to make one big task go faster
    - 按照多个子任务并行组织/by organizing it as multiple parallel sub-tasks

典型多核处理器/Typical Multicore Processor



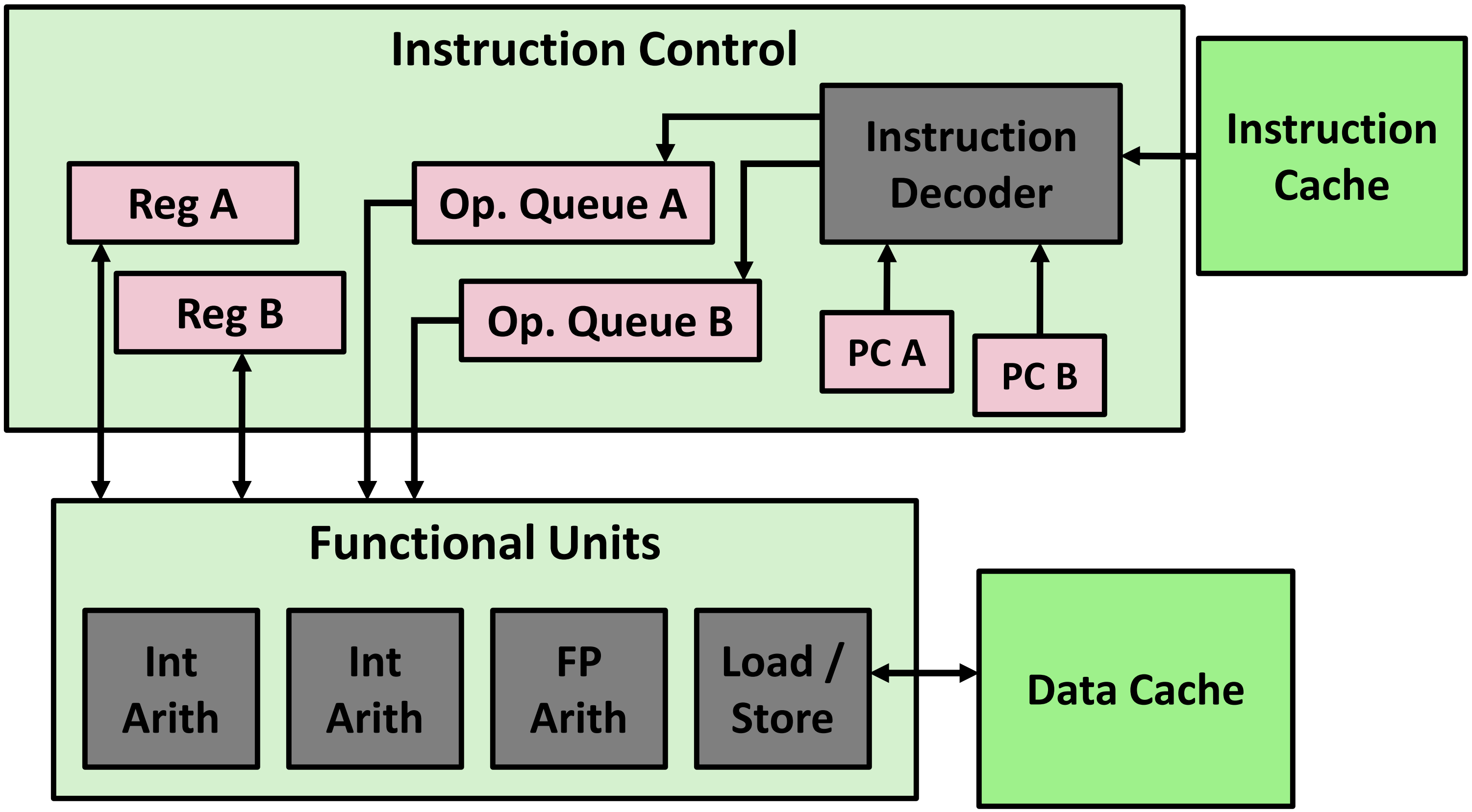
- 多个处理器在运行过程中对内存有一致的视图/Multiple processors operating with coherent view of memory

乱序处理器结构/Out-of-Order Processor Structure



- 指令控制自动将程序序转为操作流/Instruction control dynamically converts program into stream of operations
- 操作映射到功能单元上并行执行/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 enough instruction control to process K instruction streams
- 所有寄存器有K个副本/K copies of all registers
- 共享功能单元/Share functional units

测试机器/Benchmark Machine

- 从/proc/cpuinfo获取机器信息/Get data about machine from /proc/cpuinfo
- 机器Shark/Shark Machines
  - Intel Xeon E5520 @ 2.27 GHz
  - Nehalem, ca. 2010
  - 8 Cores
  - Each can do 2x hyperthreading

例1:并行求和/Example 1: Parallel Summation

- **对*0, …, n-1* 求和/Sum numbers *0, …, n-1***
  - Should add up to *((n-1)*n)/2*
- **将*1, …, n-1*划分为t个区间/Partition values *1, …, n-1* into *t* ranges**
  - 每个区间有⌊n/t⌋个值/⌊*n/t*⌋values in each range
  - 每个线程处理一个区间/Each of *t* threads processes 1 range
  - 简单起见，假设n是t的整数倍/For simplicity, assume *n* is a multiple of *t*

- **考虑一下多个线程在不同的区间上工作的不同方式/Let's consider different ways that multiple threads might work on their assigned ranges in parallel**

尝试1/First attempt: psum-mutex

- 简单方法：线程将求和结果合并到受mutex 保护的全局变量上/Simplest approach: Threads sum into a global variable protected by a semaphore mutex.

```
void *sum_mutex(void *vargp); /* Thread routine */

/* Global shared variables */
long gsum = 0;            /* Global sum */
long nelems_per_thread;   /* Number of elements to sum */
sem_t mutex;              /* Mutex to protect global sum */

int main(int argc, char **argv)
{
    long i, nelems, log_nelems, nthreads, myid[MAXTHREADS];
    pthread_t tid[MAXTHREADS];

     /* Get input arguments */
    nthreads = atoi(argv[1]);
    log_nelems = atoi(argv[2]);
    nelems = (1L << log_nelems);
    nelems_per_thread = nelems / nthreads;
    sem_init(&mutex, 0, 1);
```

psum-mutex.c

psum-mutex (cont)

- **Simplest approach: Threads sum into a global variable protected by a semaphore mutex.**

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

    /* Check final answer */
    if (gsum != (nelems * (nelems-1))/2)
        printf("Error: result=%ld\n", gsum);

    exit(0);
}
```

psum-mutex.c

psum-mutex Thread Routine/线程函数

- Simplest approach: Threads sum into a global variable protected by a semaphore mutex.

```
/* Thread routine for psum-mutex.c */
void *sum_mutex(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        P(&mutex);
        gsum += i;
        V(&mutex);
    }
    return NULL;
}
```

psum-mutex.c

`psum-mutex` Performance/性能

- **Shark machine with 8 cores, $n=2^{31}$**

Threads (Cores)	**1 (1)**	**2 (2)**	**4 (4)**	**8 (8)**	**16 (8)**
psum-mutex (secs)	51	456	790	536	681

- **意外的结果/Nasty surprise:**
  - **单个线程非常慢/Single thread is very slow**
  - **核越多越慢/Gets slower as we use more cores**

尝试2:/Next Attempt: `psum-array`

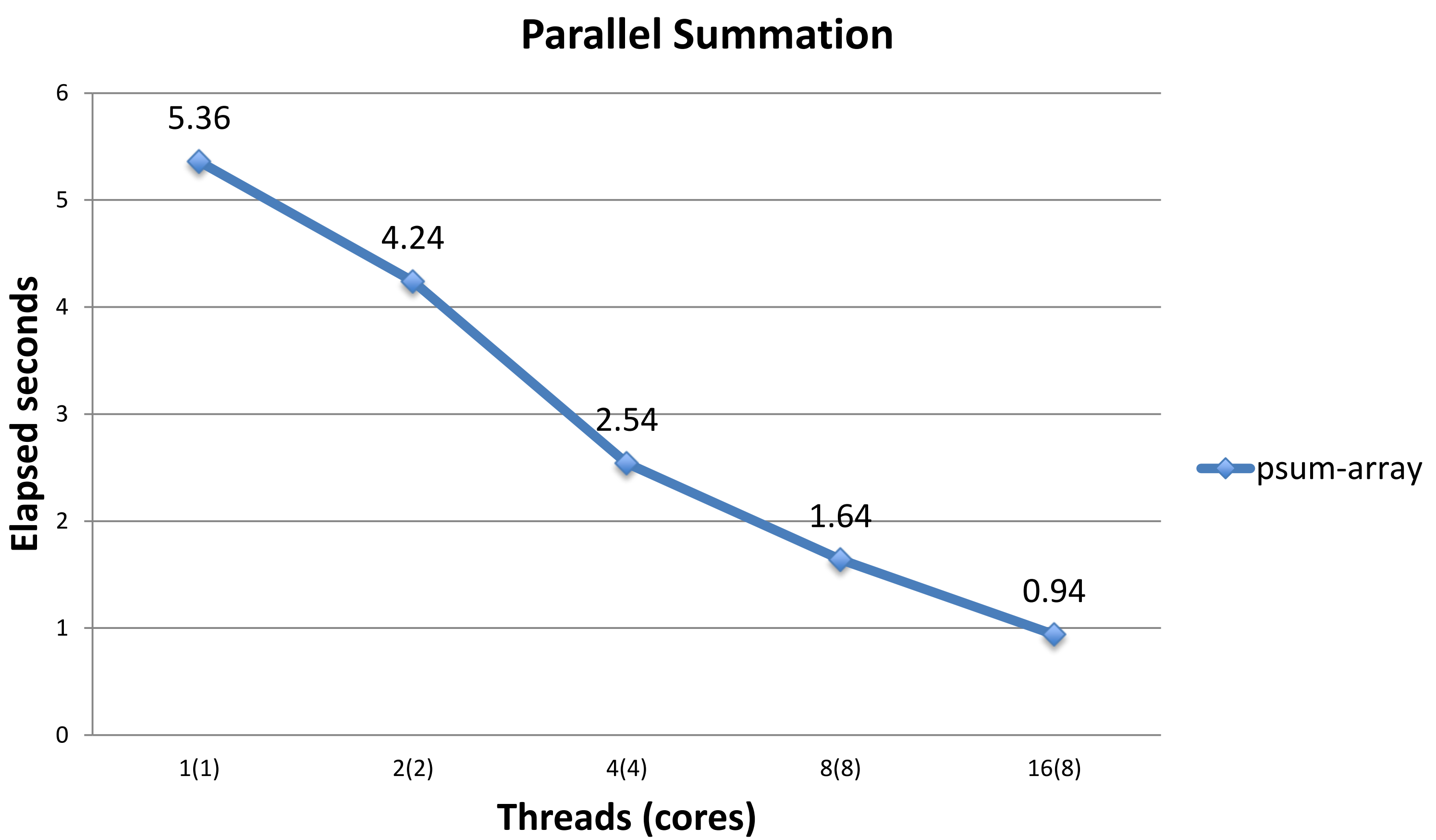
- 不同的线程归并到不同的数组元素/Peer thread `i` sums into global array element `psum[i]`
- 主线程等待其他线程完成，并对数组元素进行求和/Main waits for theads to finish, then sums elements of `psum`
- 消除了基于mutex的同步/Eliminates need for mutex synchronization

```c
/* Thread routine for psum-array.c */
void *sum_array(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i;

    for (i = start; i < end; i++) {
        psum[myid] += i;
    }
    return NULL;
}
```

psum-array.c

psum-array Performance/性能

- 比psum-mutex快一个量级/Orders of magnitude faster than psum-mutex



尝试3:/Next Attempt: psum-local

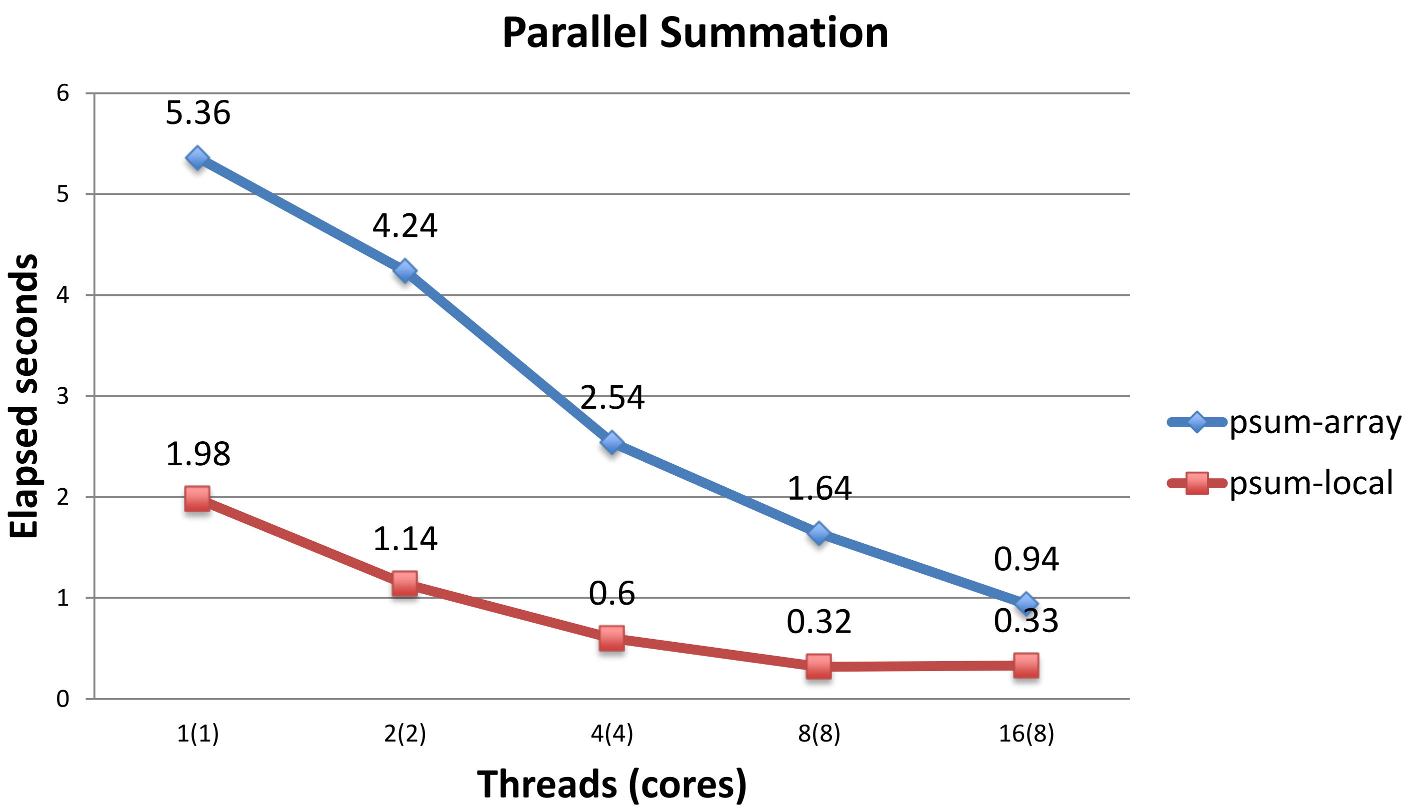
- 每个线程求和归并到局部变量/Reduce memory references by having peer thread i sum into a local variable (register)

```
/* Thread routine for psum-local.c */
void *sum_local(void *vargp)
{
    long myid = *((long *)vargp);          /* Extract thread ID */
    long start = myid * nelems_per_thread; /* Start element index */
    long end = start + nelems_per_thread;  /* End element index */
    long i, sum = 0;

    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[myid] = sum;
    return NULL;
}
```

psum-local.c

`psum-local` Performance/性能

- 比`psum-array`性能有了大幅提升/Significantly faster than `psum-array`



表征并行程序性能/Characterizing Parallel Program Performance

- ***p*表示处理器核数, T_k 表示使用k个核运行的时间/*p* processor cores, T_k is the running time using *k* cores**
- 加速比定义/***Def.*** ***Speedup:*** $S_p = T_1 / T_p$
  - S_p表示相对加速比，如果T_1是并行版本代码在1个核上的运行时间/S_p is *relative speedup* if T_1 is running time of parallel version of the code running on 1 core.
  - S_p表示绝对加速比，如果T_1是串行版本代码在1个核上的运行时间/S_p is *absolute speedup* if T_1 is running time of sequential version of code running on 1 core.
  - 绝对加速比能够更加真实的表示并行加速收益/Absolute speedup is a much truer measure of the benefits of parallelism.
- 并行效率定义/***Def.*** ***Efficiency:*** $E_p = S_p /p = T_1/(pT_p)$
  - 是(0, 100]之间的一个百分比/Reported as a percentage in the range (0, 100].
  - 测度的是并行带来的额外开销/Measures the overhead due to parallelization

`psum-local`性能/Performance of `psum-local`

Threads (t)	**1**	**2**	**4**	**8**	**16**
Cores (p)	1	2	4	8	8
Running time (T_p)	1.98	1.14	0.60	0.32	0.33
Speedup (S_p)	1	1.74	3.30	6.19	6.00
Efficiency (E_p)	100%	87%	82%	77%	75%

- 并行效率还可以，但是不是很好/**Efficiencies OK, not great**
- 我们的例子比较容易并行/**Our example is easily parallelizable**
- 实际代码更加难以并行/**Real codes are often much harder to parallelize**
  - 例如后面的**quicksort**例子/**e.g., parallel quicksort later in this lecture**

阿姆达尔定律/Amdahl's Law

- Gene Amdahl (Nov. 16, 1922 – Nov. 10, 2015)
- **描述了并行化的困难/Captures the difficulty of using parallelism to speed things up.**
- **问题概述/Overall problem**
  - T　Total sequential time required/串行运行时间
  - p　Fraction of total that can be sped up ($0 \le p \le 1$)/可并行加速比例
  - k　Speedup factor/加速因子
- **最终的性能/Resulting Performance**
  - $T_k = pT/k + (1-p)T$
    - 并行部分被加速k倍/Portion which can be sped up runs k times faster
    - 串行部分保持不动/Portion which cannot be sped up stays the same
  - 最短时间/Least possible running time:
    - $k = \infty$
    - $T_\infty = (1-p)T$

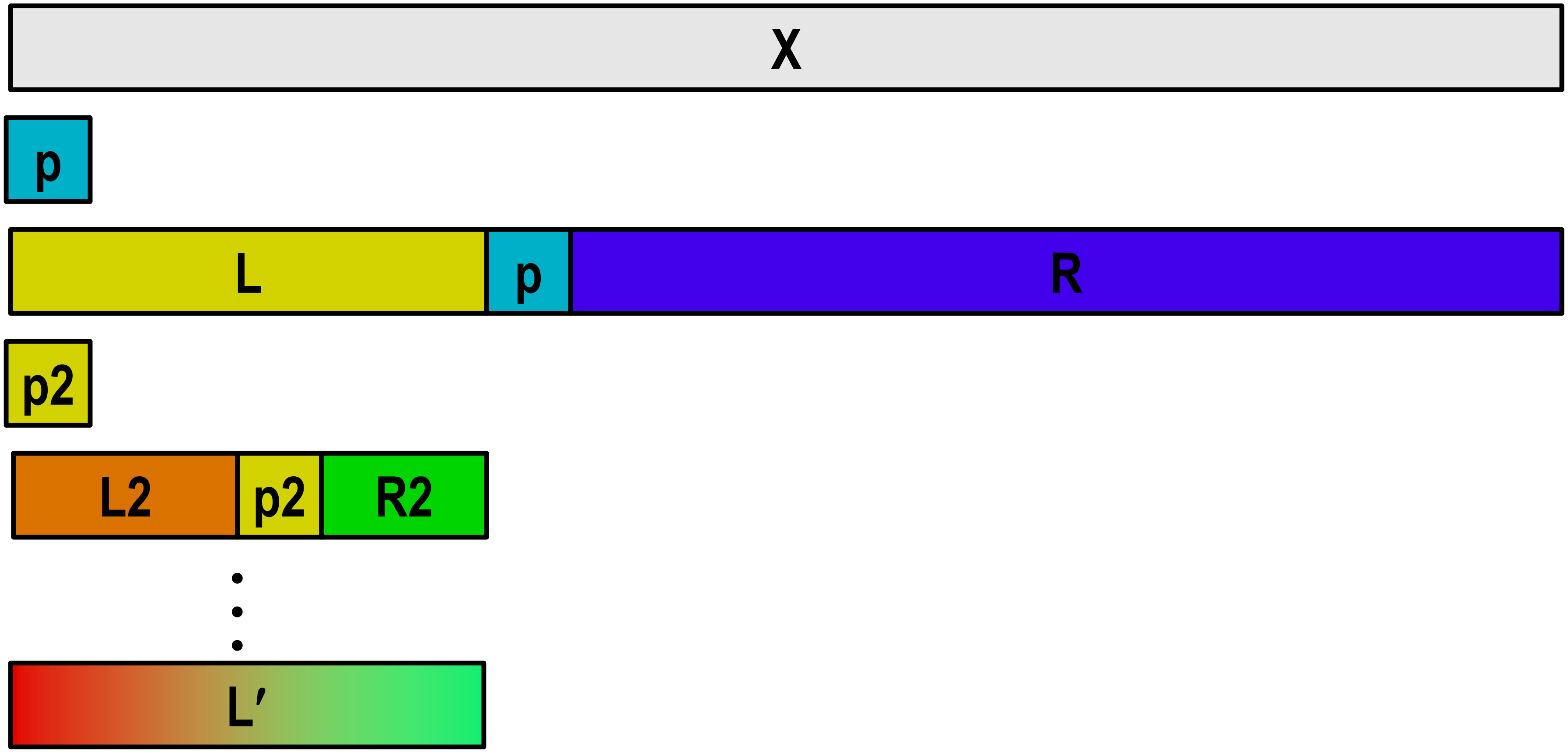
阿姆达尔定律举例/Amdahl's Law Example

- **问题概述/Overall problem**
  - T = 10 Total time required/总的运行时间
  - p = 0.9 Fraction of total which can be sped up/可并行部分加速比
  - k = 9 Speedup factor/加速因子
- **最终性能/Resulting Performance**
  - $T_9 = 0.9 * 10/9 + 0.1 * 10 = 1.0 + 1.0 = 2.0$
  - 最短时间/Least possible running time:
    - $T_\infty = 0.1 * 10.0 = 1.0$

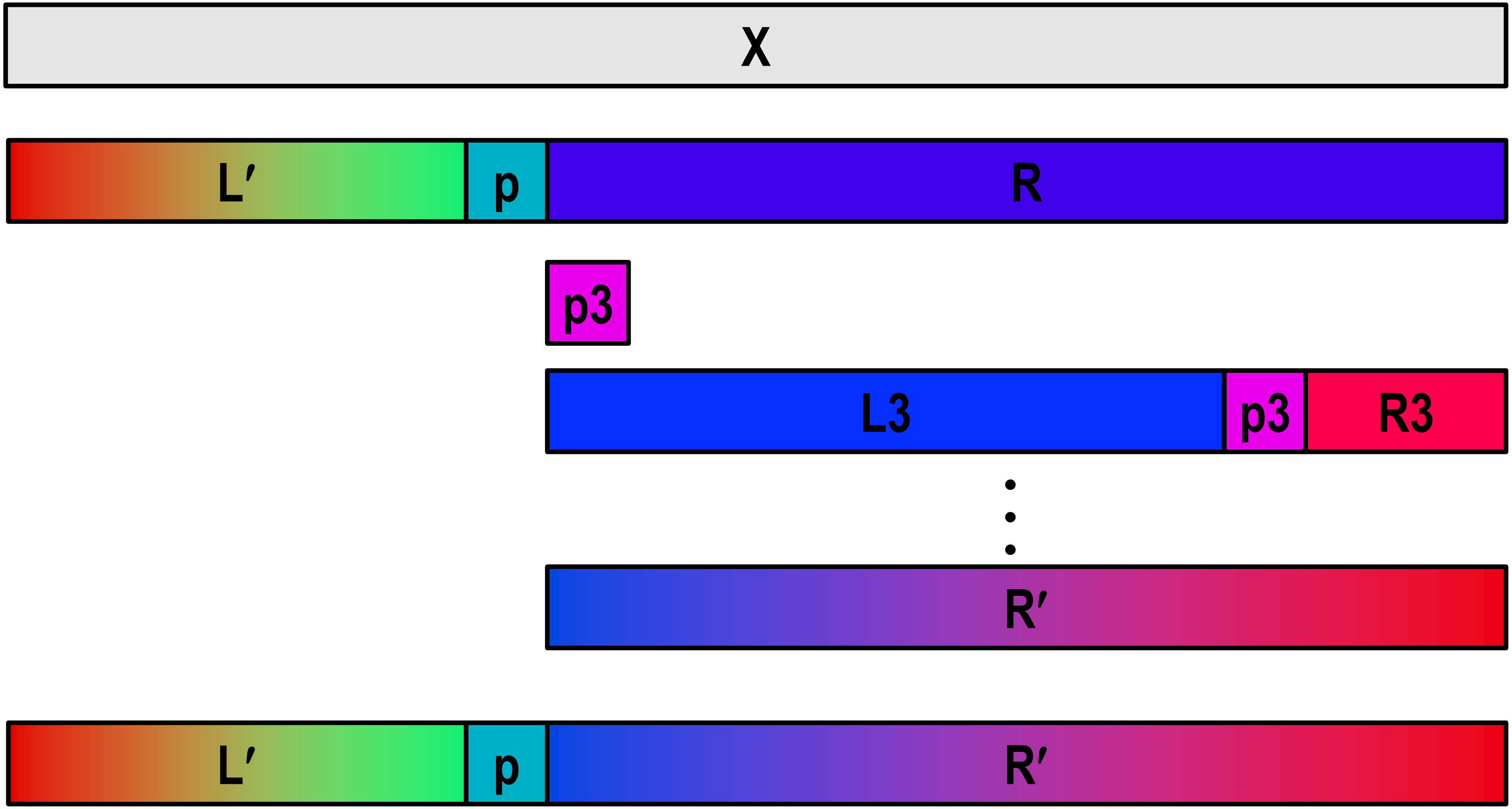
一个更复杂的例子/A More Substantial Example: Sort

- **对N个随机数排序/Sort set of N random numbers**
- **多个可能的算法/Multiple possible algorithms**
  - 使用quicksort的并行版本/Use parallel version of quicksort
- **对X数集的串行quicksort排序/Sequential quicksort of set of values X**
  - 从X中选择枢轴p/Choose “pivot” p from X
  - 对X划分/Rearrange X into
    - L: Values $\leq$ p
    - R: Values $\geq$ p
  - 对L递归排序形成L′/Recursively sort L to get L′
  - 对R递归排序形成R′/Recursively sort R to get R′
  - 返回/Return L′ : p : R′

串行Quicksort 可视化/Sequential Quicksort Visualized



串行Quicksort 可视化/Sequential Quicksort Visualized



串行Quicksort 代码/Sequential Quicksort Code

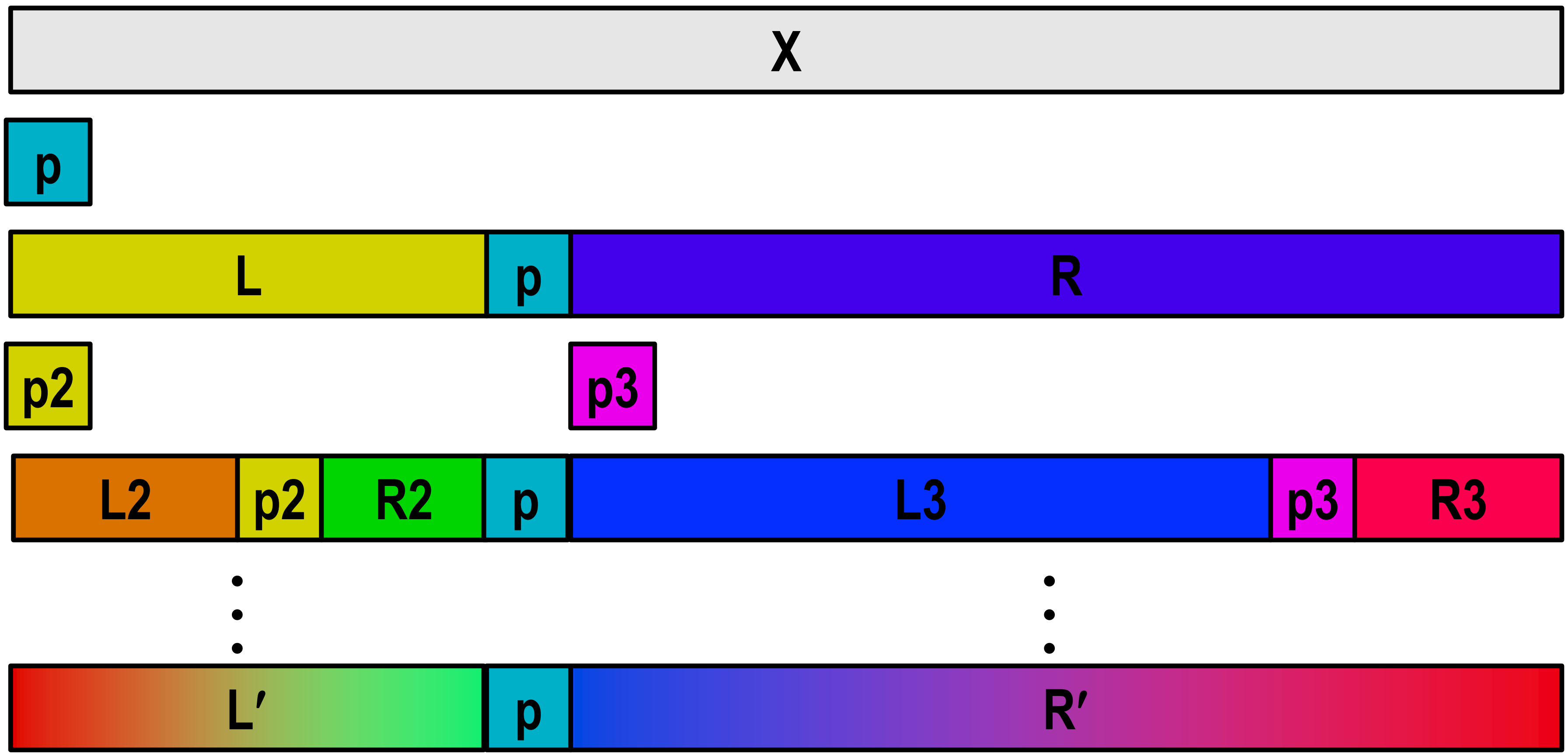
```
void qsort_serial(data_t *base, size_t nele) {
  if (nele <= 1)
    return;
  if (nele == 2) {
    if (base[0] > base[1])
      swap(base, base+1);
    return;
  }

  /* 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L或者R多于一个元素则递归排序/Recursively sort L or R if has more than one element

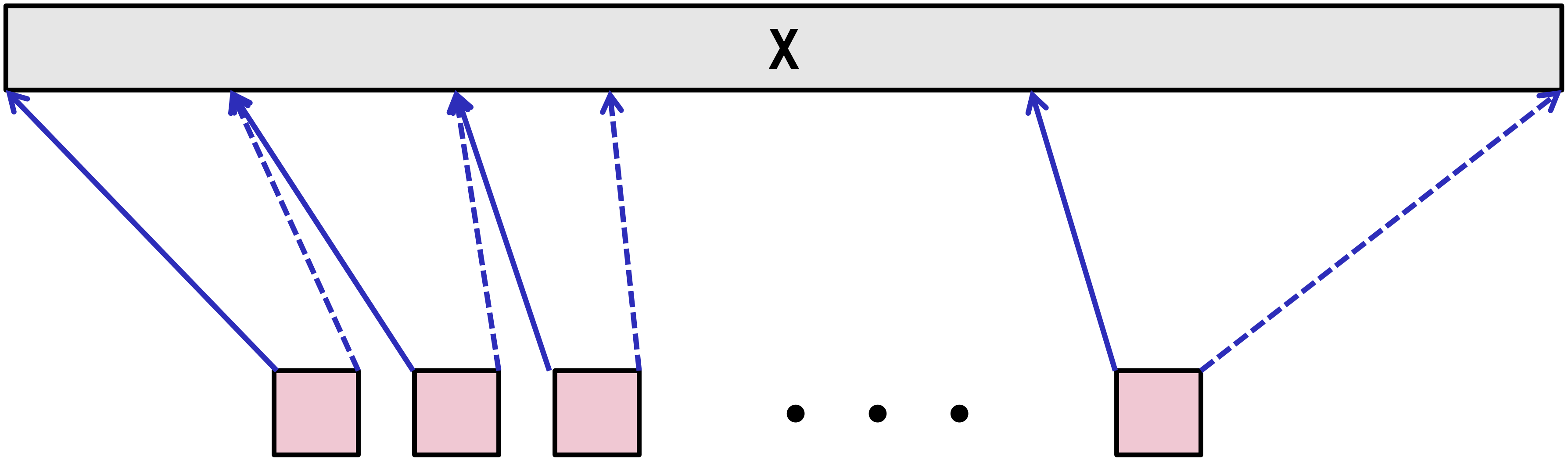
并行Quicksort/Parallel Quicksort

- **对数集X进行并行排序/Parallel quicksort of set of values X**
  - 如果N ≤ Nthresh， 则进行串行排序/If N ≤ Nthresh, do sequential quicksort
  - 否则/Else
    - 从X中选择枢轴p/Choose “pivot” p from X
    - 将X划分为/Rearrange X into
      - L: Values ≤ p
      - R: Values ≥ p
    - 递归创建独立线程进行排序/Recursively spawn separate threads
      - 对L进行排序形成L′/Sort L to get L′
      - 对R进行排序形成R′/Sort R to get R′
    - 返回/Return L′ : p : R′

并行Quicksort 可视化/Parallel Quicksort Visualized



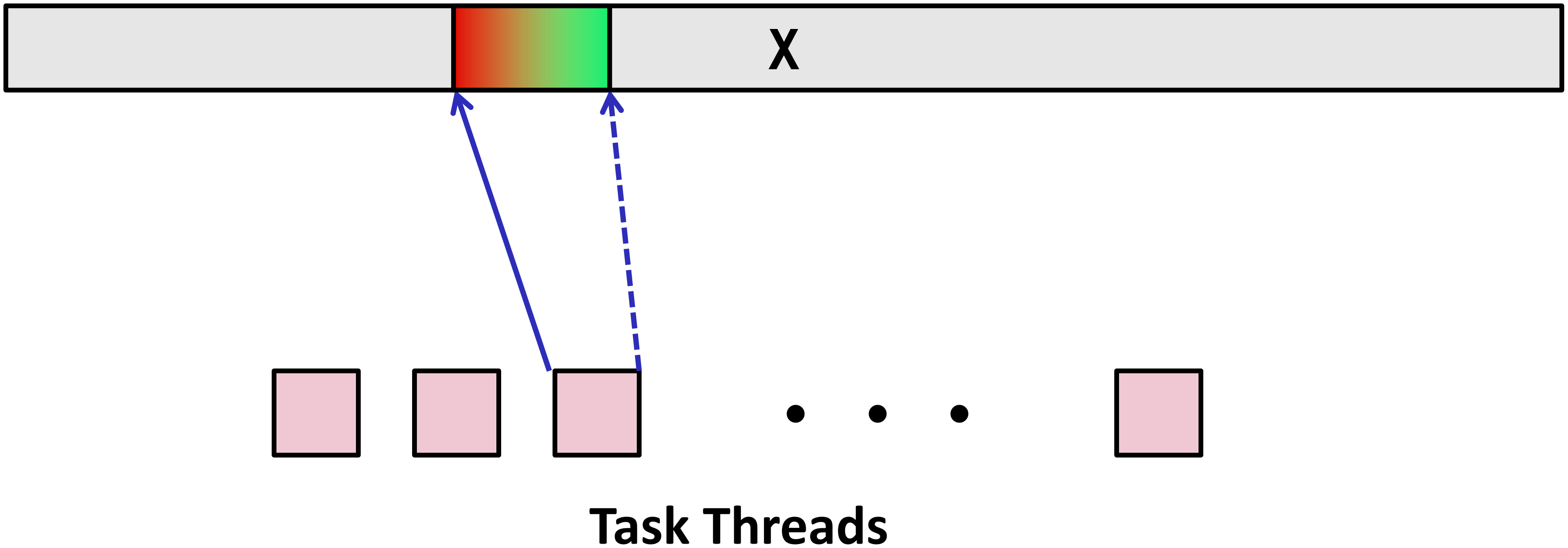
排序任务的线程结构/Thread Structure: Sorting Tasks



Task Threads

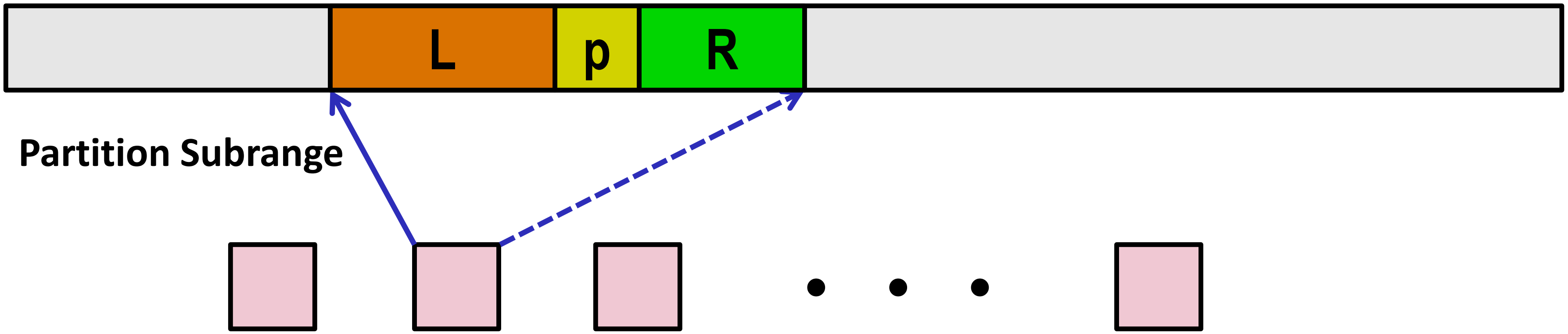
- **任务:对子区间进行排序/Task: Sort subrange of data**
  - 描述为/Specify as:
    - `base`: Starting address/开始地址
    - `nele`: Number of elements in subrange/子区间元素数量
- **按照独立线程运行/Run as separate thread**

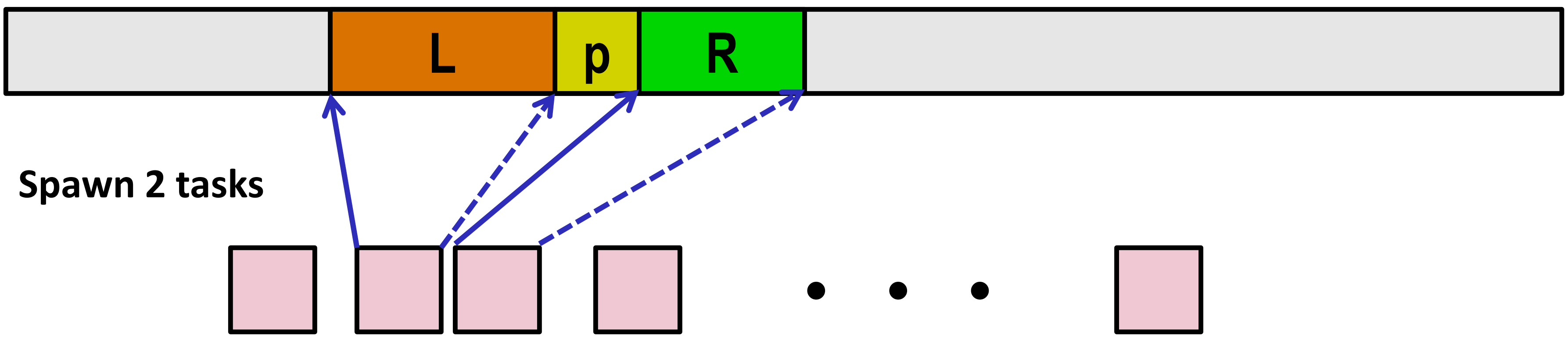
小规模排序任务运行/Small Sort Task Operation



- 使用串行quicksort 对子区间排序/Sort subrange using serial quicksort

大规模排序任务操作/Large Sort Task Operation





顶层函数（简化版）/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);
}
```

- 创建数据结构/Sets up data structures
- 递归调用排序函数/Calls recursive sort routine
- 持续对线程进行合并/Keeps joining threads until none left
- 释放数据结构Frees data structures

递归排序函数（简化版）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);
}
```

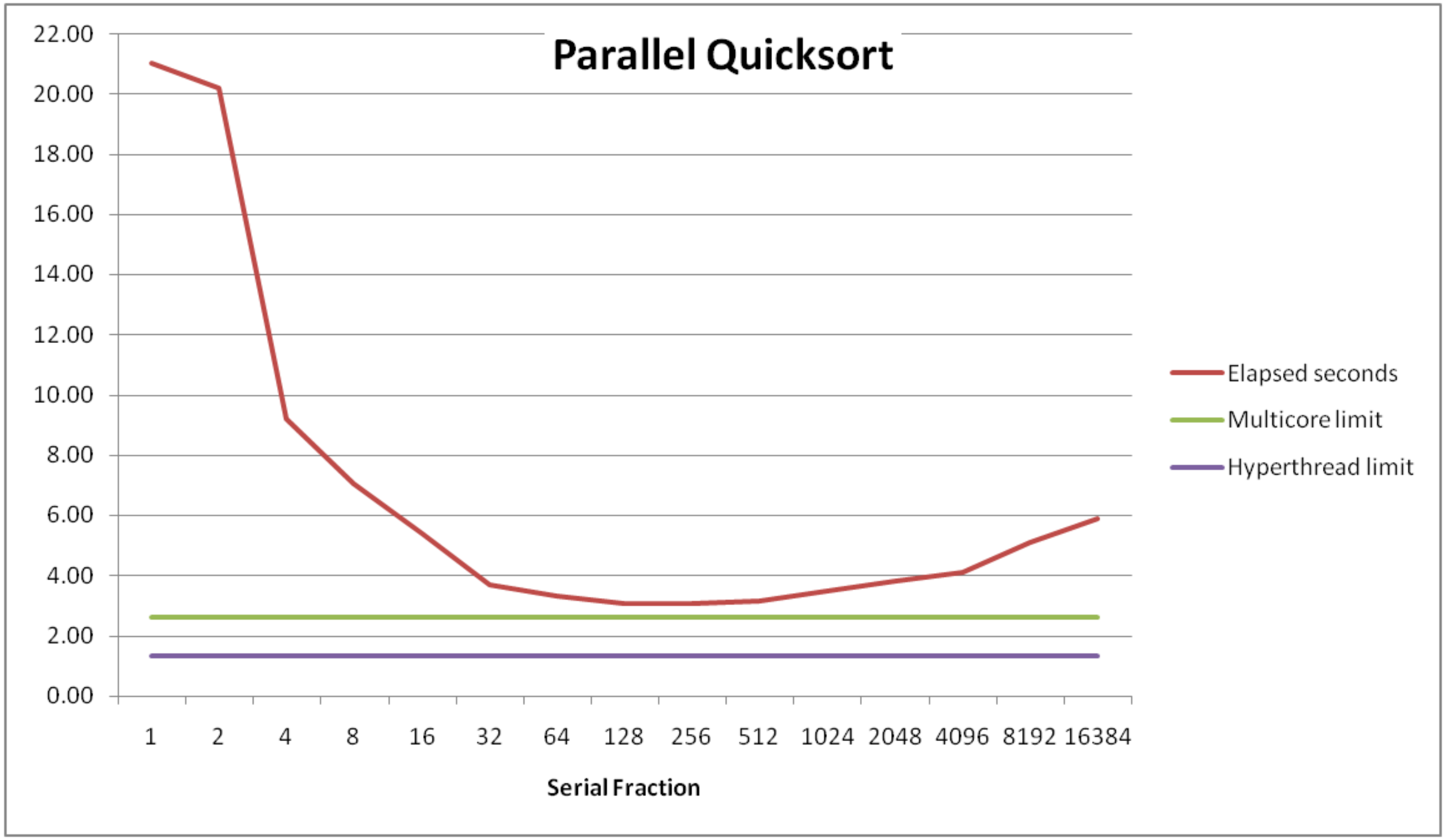
- 小区间：串行排序/**Small partition: Sort serially**
- 大区间：创建新的排序任务/**Large partition: Spawn new sort task**

排序任务线程（简化版）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;
}
```

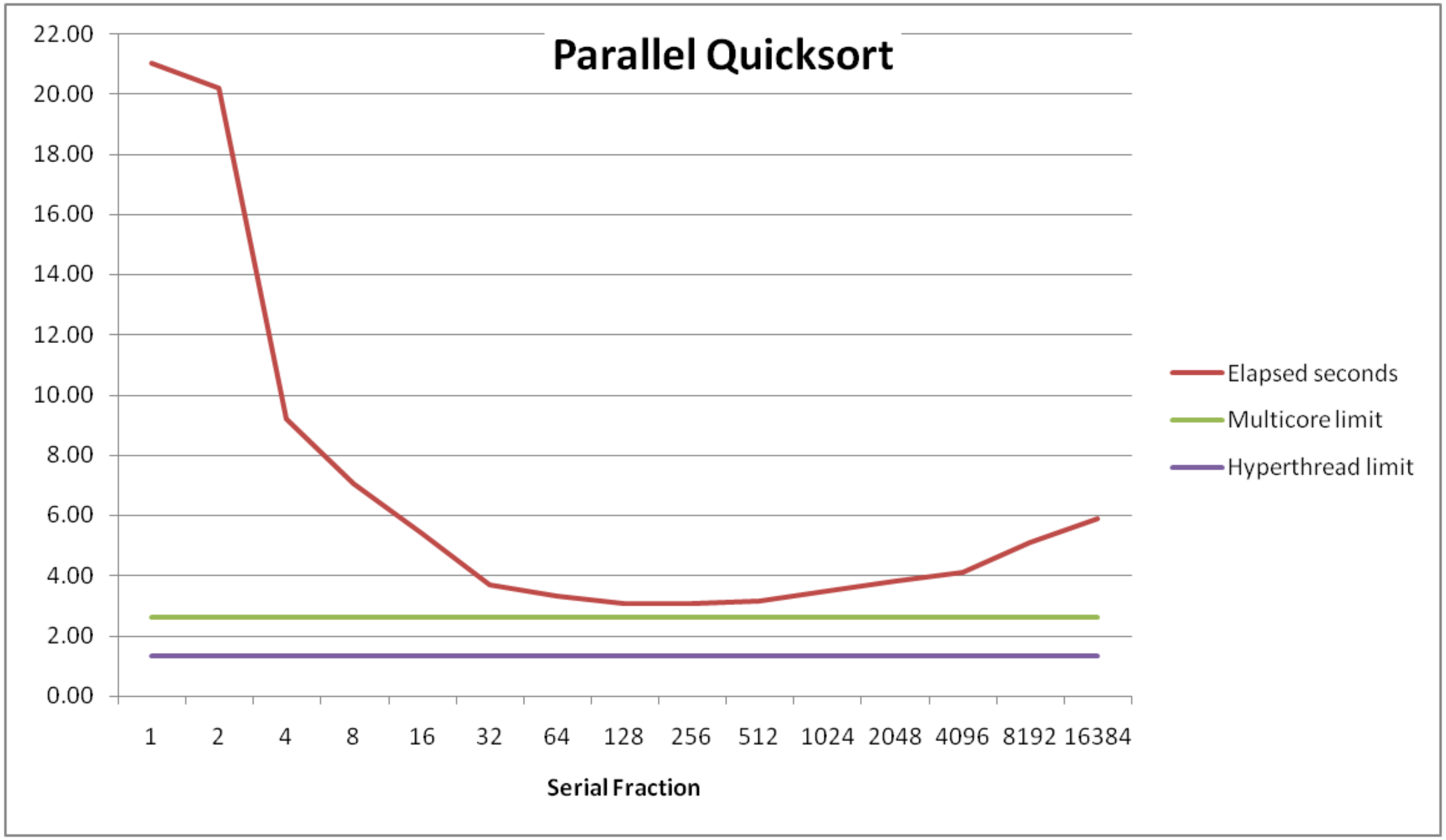
- 获得任务参数/Get task parameters
- 进行划分/Perform partitioning step
- 对每个划分调用递归排序函数/Call recursive sort routine on each partition

并行Quicksort性能/Parallel Quicksort Performance



- **串行占比:输入中串行排序的占比/Serial fraction: Fraction of input at which do serial sort**
- **对2^{27}进行排序/Sort 2^{27} (134,217,728) random values**
- **最好的加速比/Best speedup = 6.84X**

并行Quicksort性能/ Parallel Quicksort Performance

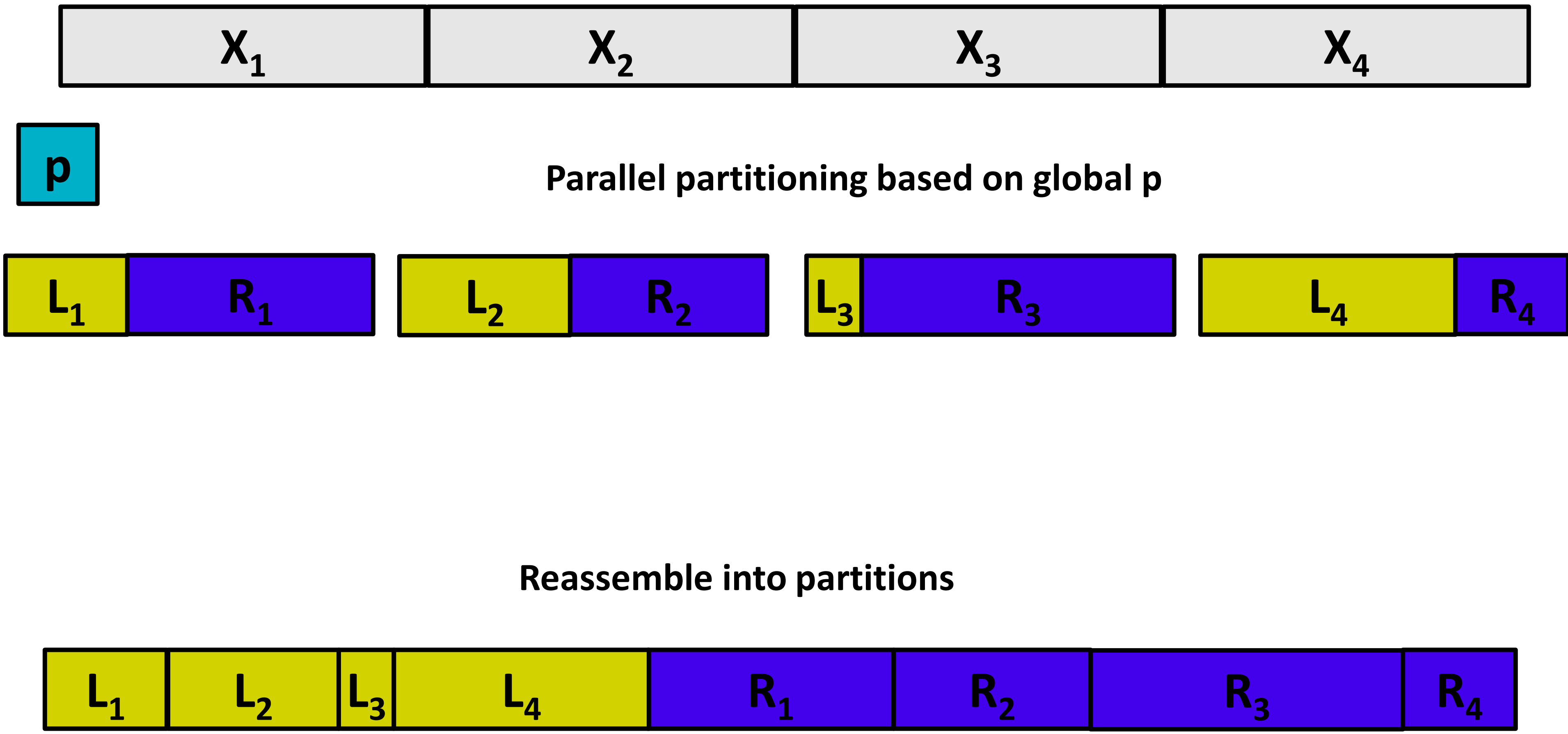


- 对于大部分占比都有比较好的性能/**Good performance over wide range of fraction values**
  - F太小:并行度不够/F too small: Not enough parallelism
  - F太大:线程开销较大+线程栈空间不够/F too large: Thread overhead + run out of thread memory

阿姆达尔定律和并行Quicksort/Amdahl's Law & Parallel Quicksort

- **串行瓶颈/Sequential bottleneck**
  - 顶层划分:无加速/Top-level partition: No speedup
  - 第二层: ≤ 2X 加速比/Second level: ≤ 2X speedup
  - 第k层: : ≤ 2^{k-1}X 加速比/k^{th} level: ≤ 2^{k-1}X speedup
- **启发/Implications**
  - 小规模并行具有比较好的性能/Good performance for small-scale parallelism
  - 需要对划分进行并行以实现更大规模的并行/Would need to parallelize partitioning step to get large-scale parallelism
    - 基于采样的并行排序/Parallel Sorting by Regular Sampling
      - H. Shi & J. Schaeffer, J. Parallel & Distributed Computing, 1992

划分并行/Parallelizing Partitioning Step



并行划分的经验/Experience with Parallel Partitioning

- **无法获得加速比/Could not obtain speedup**
- **原因分析：太多数据拷贝/Speculate: Too much data copying**
  - 无法在原有数组内完成/Could not do everything within source array
  - 创建临时空间以重新整合划分/Set up temporary space for reassembling partition

获得的教训/Lessons Learned

- **必须有并行化策略/Must have parallelization strategy**
  - 划分为K个独立的部分/Partition into K independent parts
  - 分治/Divide-and-conquer
- **内存循环不应该有同步/Inner loops must be synchronization free**
  - 同步操作开销过高/Synchronization operations very expensive
- **时刻记住阿尔达姆定律/Beware of Amdahl’s Law**
  - 串行代码可能成为瓶颈/Serial code can become bottleneck
- **你可以的/You can do it!**
  - 实现一定程度的并行并不困难/Achieving modest levels of parallelism is not difficult
  - 构建实验框架并测试不同的策略/Set up experimental framework and test multiple strategies

内存顺序一致性模型/Memory Consistency

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

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

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

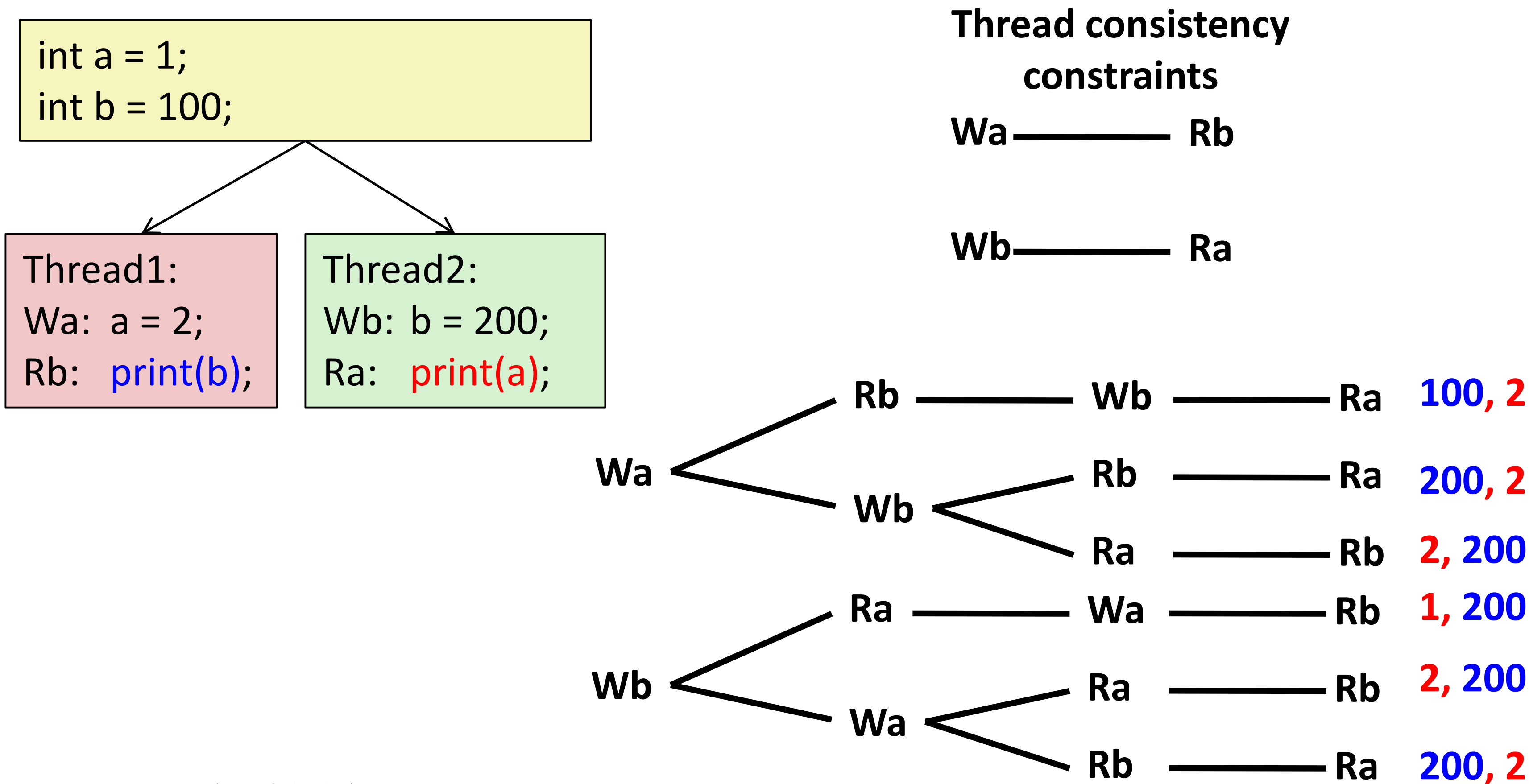
线程一致性限制/Thread consistency constraints

Wa $\rightarrow$ **Rb**

Wb $\rightarrow$ **Ra**

- **打印出来的值可能有哪些？/What are the possible values printed?**
  - 依赖于内存顺序一致性模型/Depends on memory consistency model
  - 是硬件如何实现并发访问的抽象模型/Abstract model of how hardware handles concurrent accesses
- **串行一致性模型/Sequential consistency**
  - 线程内满足程序序/Overall effect consistent with each individual thread
  - 其他任意交叉/Otherwise, arbitrary interleaving

串行顺序一致性举例/Sequential Consistency Example



- 不可能的输出/Impossible outputs
  - 100, 1 and 1, 100
  - 需要在Wa和Wb之前到达Ra和Rb/Would require reaching both Ra and Rb before Wa and Wb

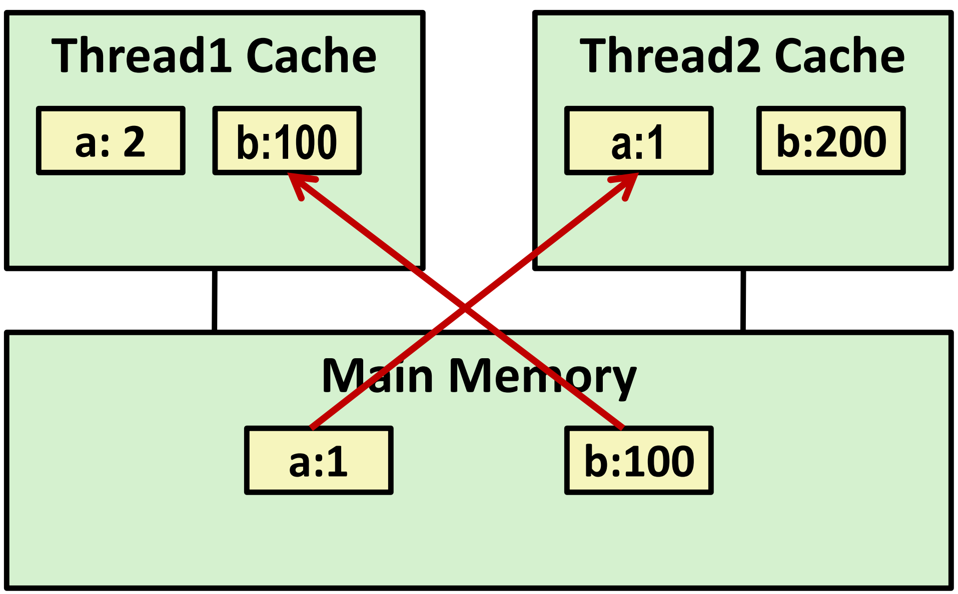
没有Cache一致性协议时/Non-Coherent Cache Scenario

- **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

总线侦听Cache一致性协议/Snoopy Caches

- **对每个Cache块打标签/Tag each cache block with state**

  非法/Invalid　不能用/Cannot use value

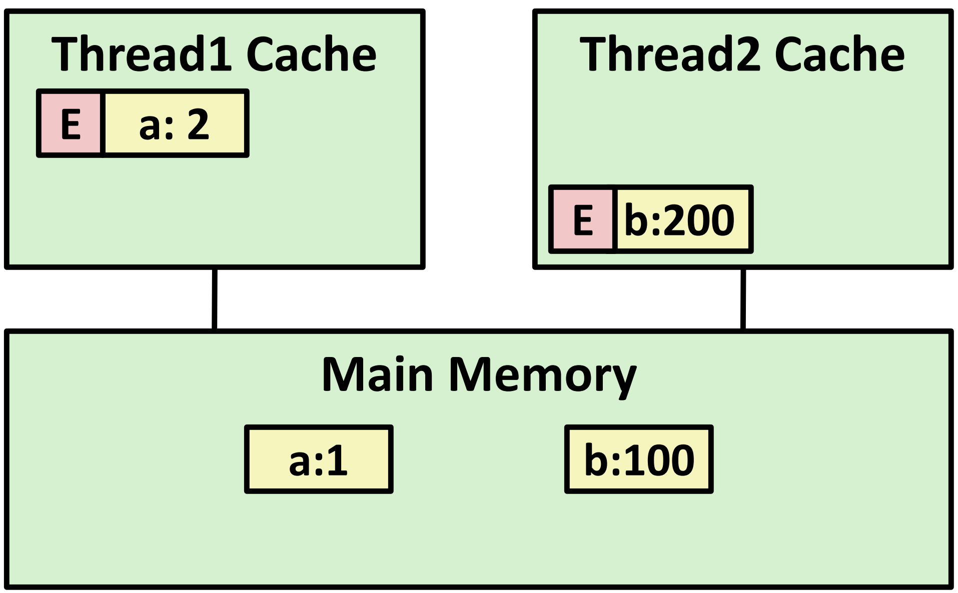
  共享/Shared　可读副本/Readable copy

  独占/Exclusive 可写副本/Writeable copy

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

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

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



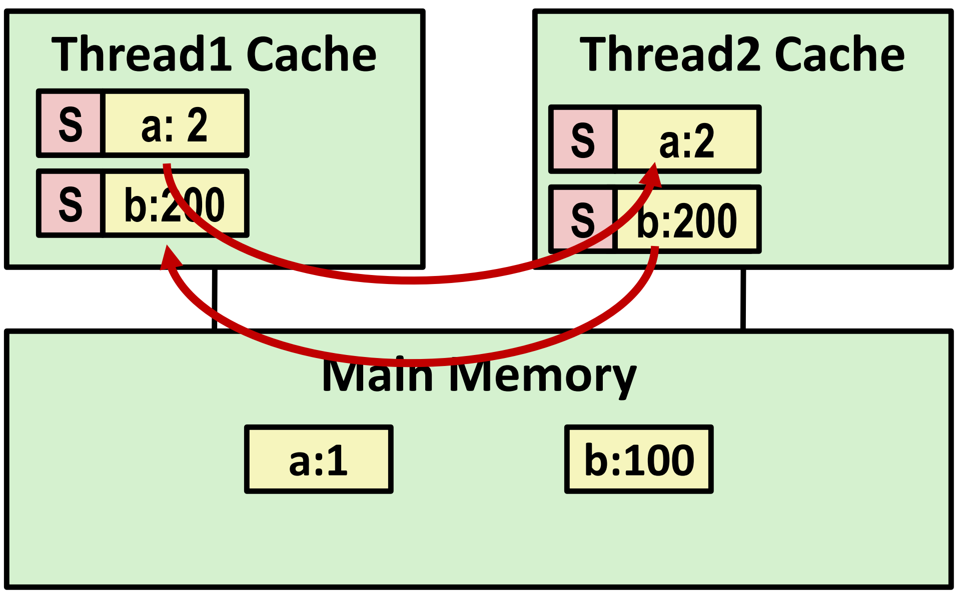
总线侦听Cache一致性协议/ Snoopy Caches

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

Thread1:	Thread2:
Wa: a = 2;	Wb: b = 200;
Rb: print(b);	Ra: print(a);

- **Tag each cache block with state**

Invalid	Cannot use value
Shared	Readable copy
Exclusive	Writeable copy



print 2

print 200

- 当cache看到某个对标记位E的块的请求/When cache sees request for one of its E-tagged blocks
  - 从自己的Cache提供数据/Supply value from cache
  - 将标记改为S/Set tag to S