Thread-Level Parallelism

15-213: Introduction to Computer Systems
26th Lecture, July 28, 2016

Instructor:

Brian Railing

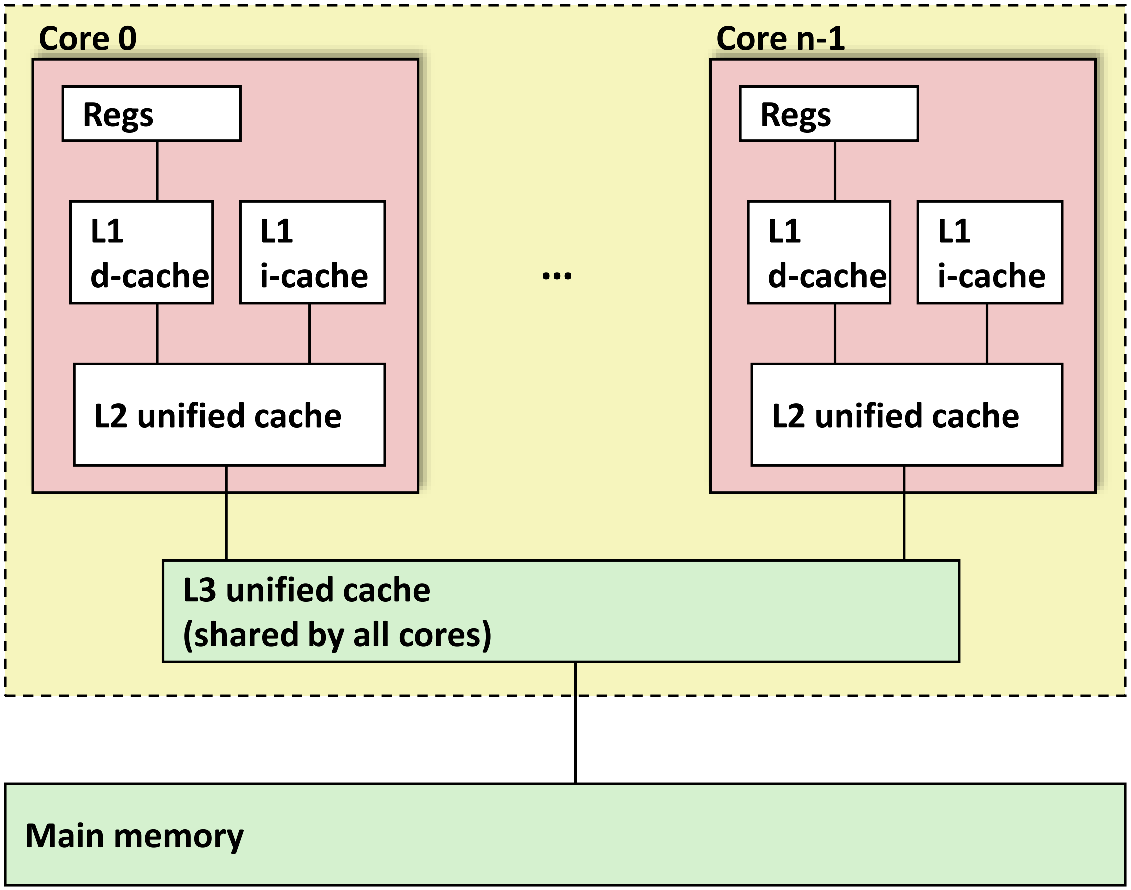
Today

- **Parallel Computing Hardware**
  - Multicore
    - Multiple separate processors on single chip
  - Hyperthreading
    - Efficient execution of multiple threads on single core
- **Thread-Level Parallelism**
  - Splitting program into independent tasks
    - Example 1: Parallel summation
  - Divide-and conquer parallelism
    - Example 2: Parallel quicksort
- **Consistency Models**
  - What happens when multiple threads are reading & writing shared state

Exploiting parallel execution

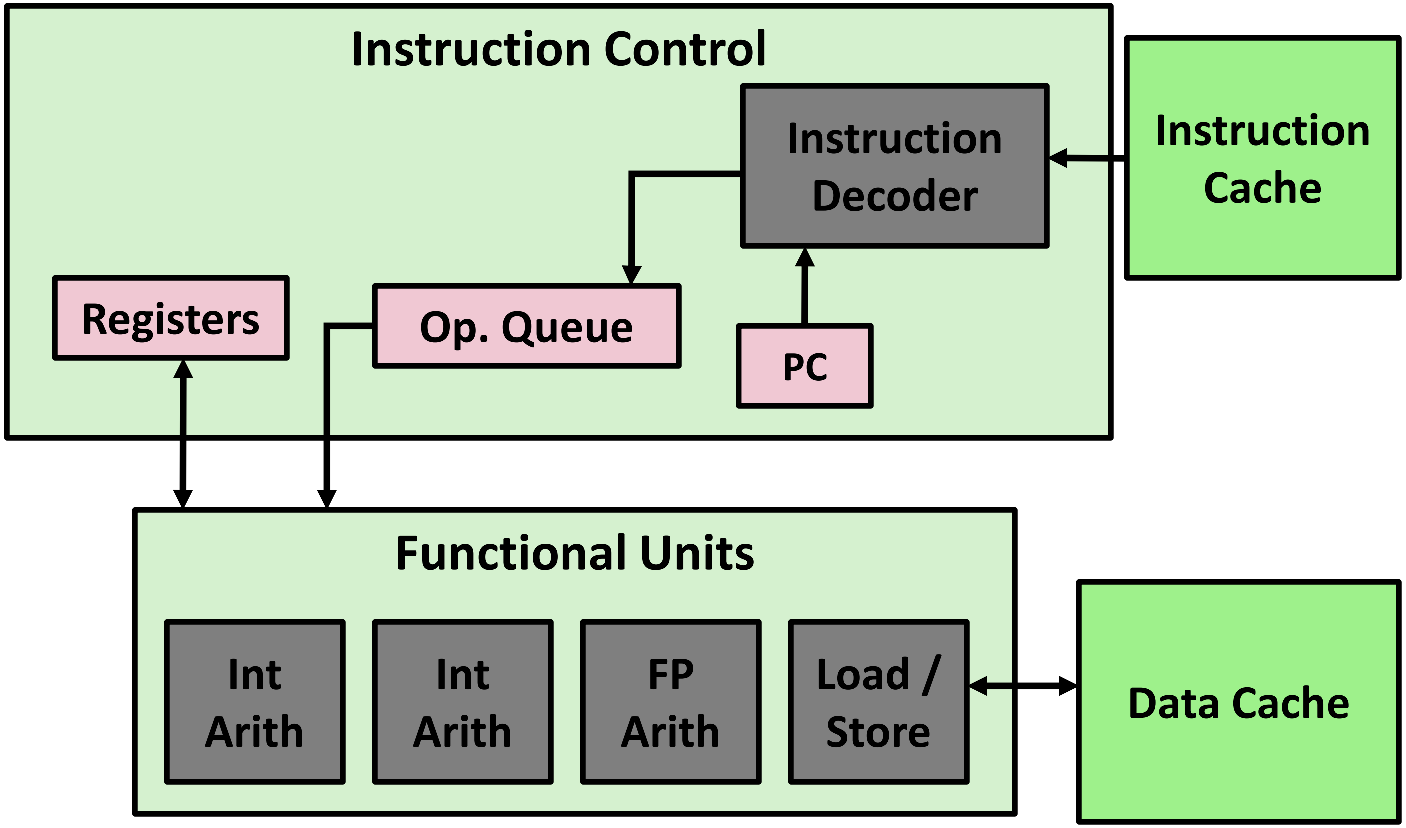
- **So far, we've used threads to deal with I/O delays**
  - e.g., one thread per client to prevent one from delaying another
- **Multi-core/Hyperthreaded CPUs offer another opportunity**
  - Spread work over threads executing in parallel
  - Happens automatically, if many independent tasks
    - e.g., running many applications or serving many clients
  - Can also write code to make one big task go faster
    - by organizing it as multiple parallel sub-tasks

Typical Multicore Processor



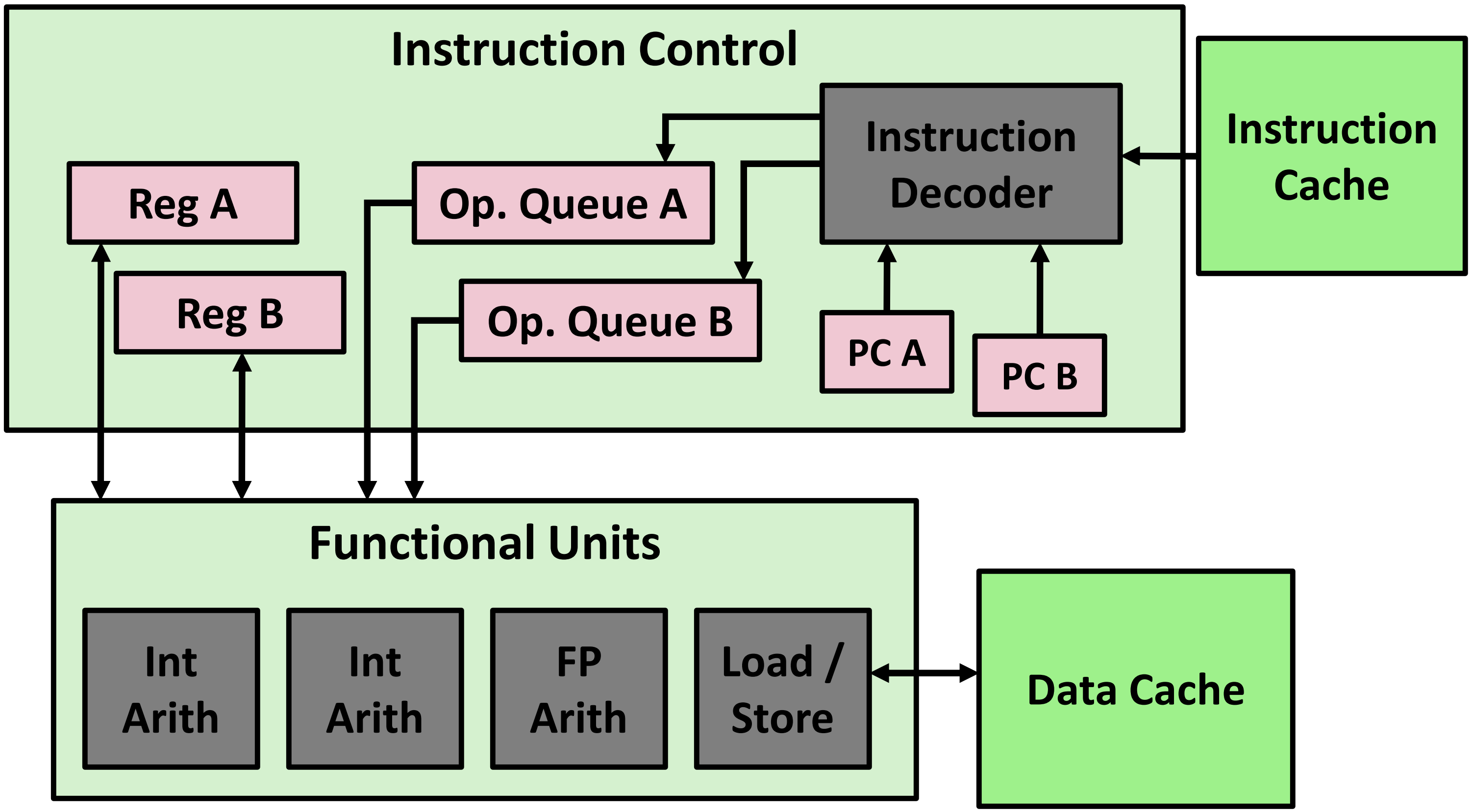
- **Multiple processors operating with coherent view of memory**

Out-of-Order Processor Structure



- Instruction control dynamically converts program into stream of operations
- Operations mapped onto functional units to execute in parallel

Hyperthreading Implementation



- Replicate enough instruction control to process K instruction streams
- K copies of all registers
- Share functional units

Benchmark Machine

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

Example 1: Parallel Summation

- **Sum numbers *0, …, n-1***
  - Should add up to $((n-1)*n)/2$
- **Partition values *1, …, n-1* into *t* ranges**
  - $\lfloor n/t \rfloor$ values in each range
  - Each of *t* threads processes 1 range
  - For simplicity, assume *n* is a multiple of *t*

- **Let's consider different ways that multiple threads might work on their assigned ranges in parallel**

First attempt: psum-mutex

- **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);

    return 0;
}
```

psum-mutex.c

psum-mutex Thread Routine

- 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

- **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

- **Nasty surprise:**
  - **Single thread is very slow**
  - **Gets slower as we use more cores**

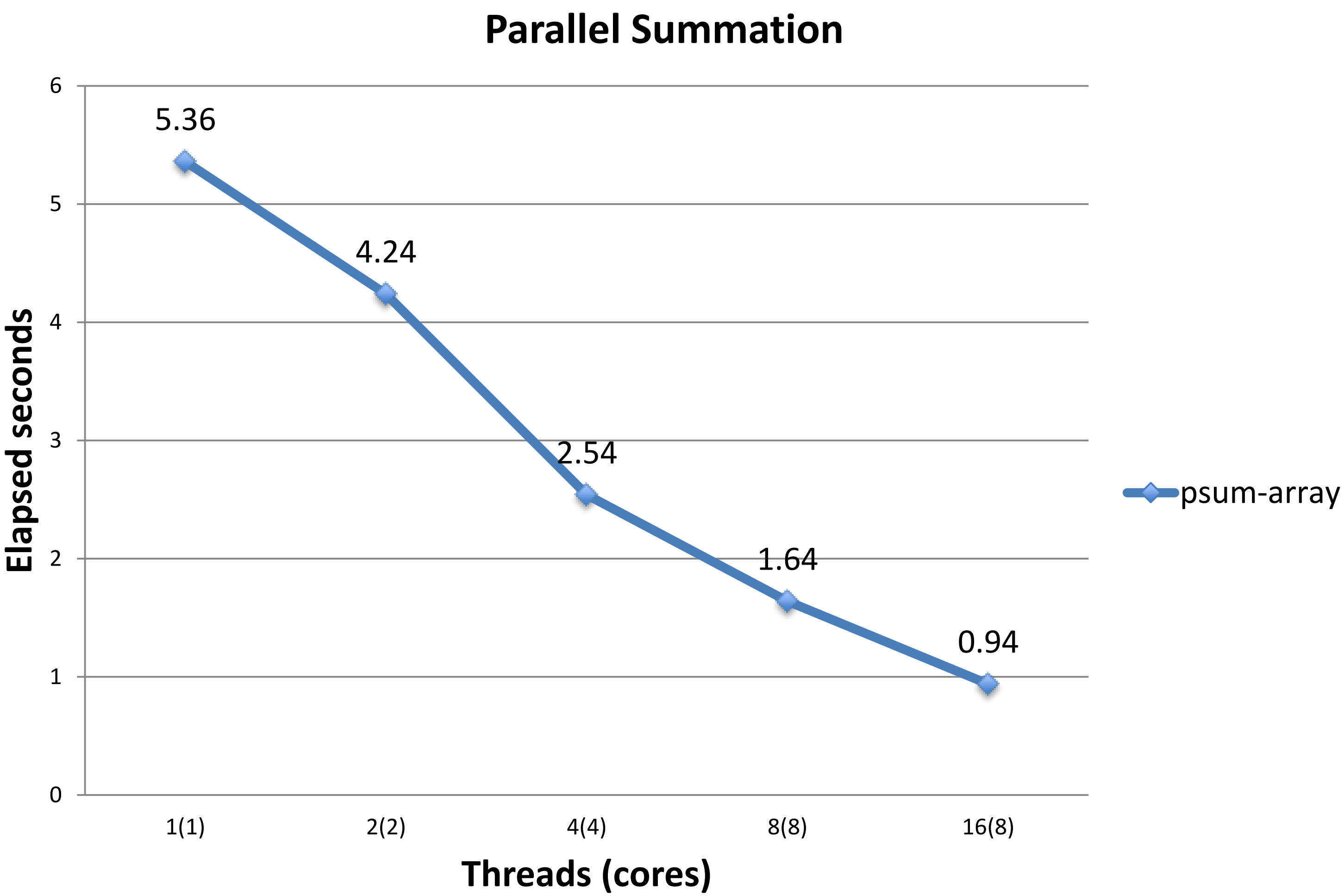
Next Attempt: psum-array

- Peer thread `i` sums into global array element `psum[i]`
- Main waits for theads to finish, then sums elements of `psum`
- Eliminates need for mutex synchronization

```
/* 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

- Orders of magnitude faster than `psum-mutex`



Next Attempt: `psum-local`

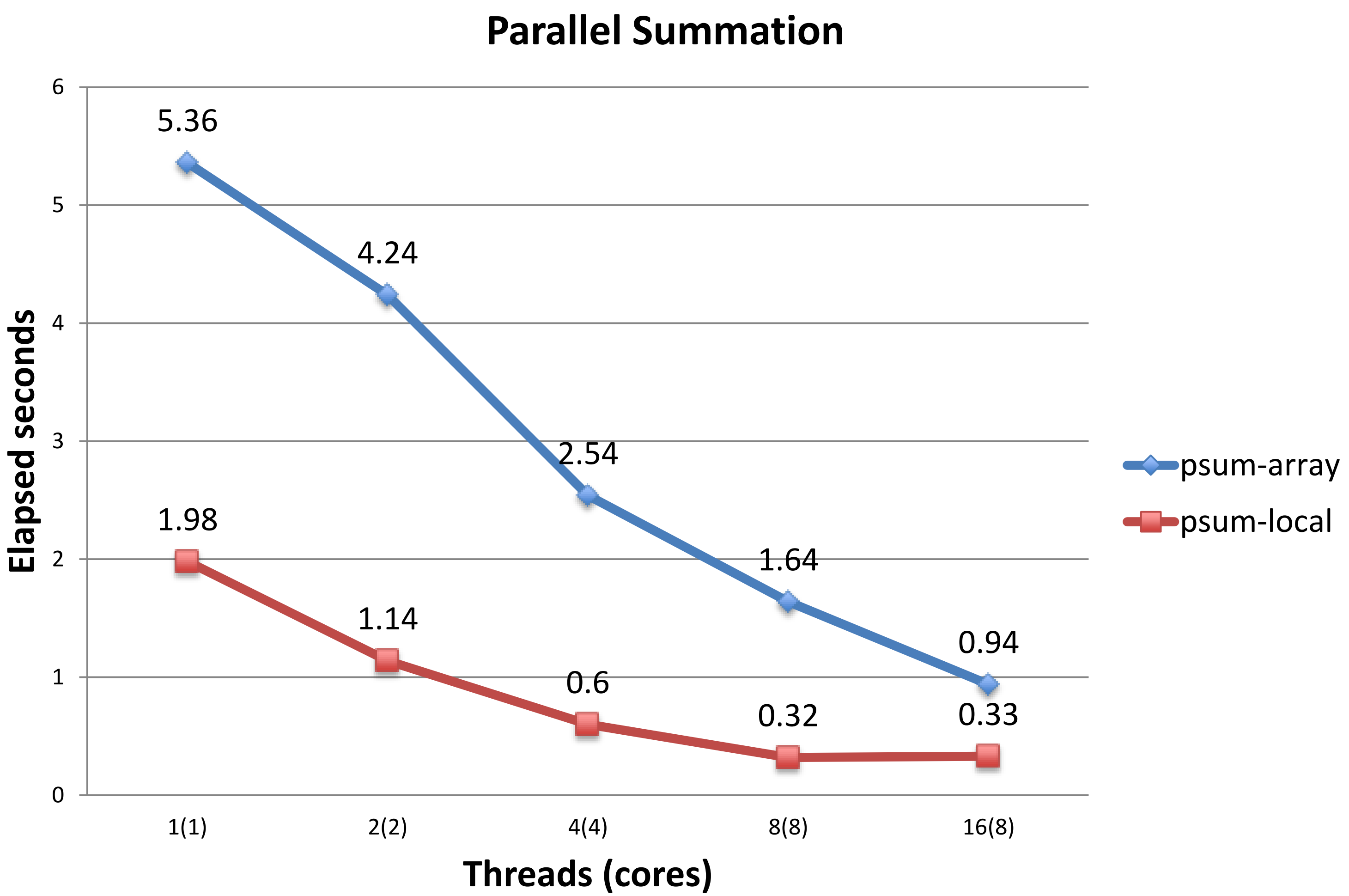
- Reduce memory references by having peer thread i sum into a local variable (register)

```
/* 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

- Significantly faster than `psum-array`



Characterizing Parallel Program Performance

- ***p* processor cores, T_k is the running time using *k* cores**

- ***Def.* *Speedup:* $S_p = T_1 / T_p$**
  - S_p is *relative speedup* if T_1 is running time of parallel version of the code running on 1 core.
  - S_p is *absolute speedup* if T_1 is running time of sequential version of code running on 1 core.
  - Absolute speedup is a much truer measure of the benefits of parallelism.

- ***Def.* *Efficiency:* $E_p = S_p / p = T_1 / (pT_p)$**
  - Reported as a percentage in the range (0, 100].
  - Measures the overhead due to parallelization

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%

- **Efficiencies OK, not great**
- **Our example is easily parallelizable**
- **Real codes are often much harder to parallelize**
  - **e.g., parallel quicksort later in this lecture**

Amdahl's Law

- Gene Amdahl (Nov. 16, 1922 – Nov. 10, 2015)

- **Captures the difficulty of using parallelism to speed things up.**
- **Overall problem**
  - T Total sequential time required
  - p Fraction of total that can be sped up ($0 \leq p \leq 1$)
  - k Speedup factor
- **Resulting Performance**
  - $T_k = pT/k + (1-p)T$
    - Portion which can be sped up runs k times faster
    - Portion which cannot be sped up stays the same
  - Least possible running time:
    - $k = \infty$
    - $T_\infty = (1-p)T$

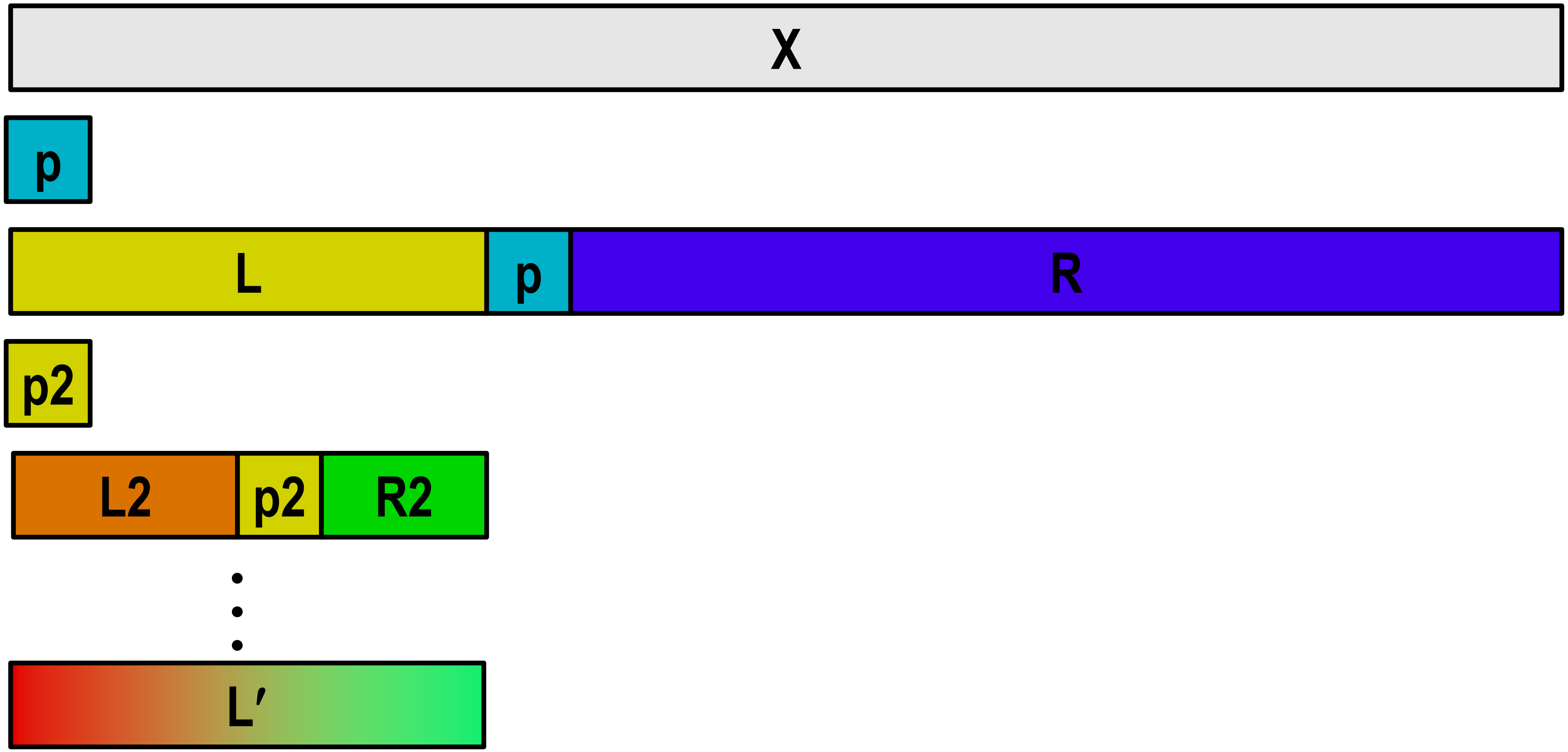
Amdahl's Law Example

- **Overall problem**
  - T = 10 Total time required
  - p = 0.9 Fraction of total which can be sped up
  - k = 9 Speedup factor
- **Resulting Performance**
  - $T_9 = 0.9 * 10/9 + 0.1 * 10 = 1.0 + 1.0 = 2.0$
  - Least possible running time:
    - $T_\infty = 0.1 * 10.0 = 1.0$

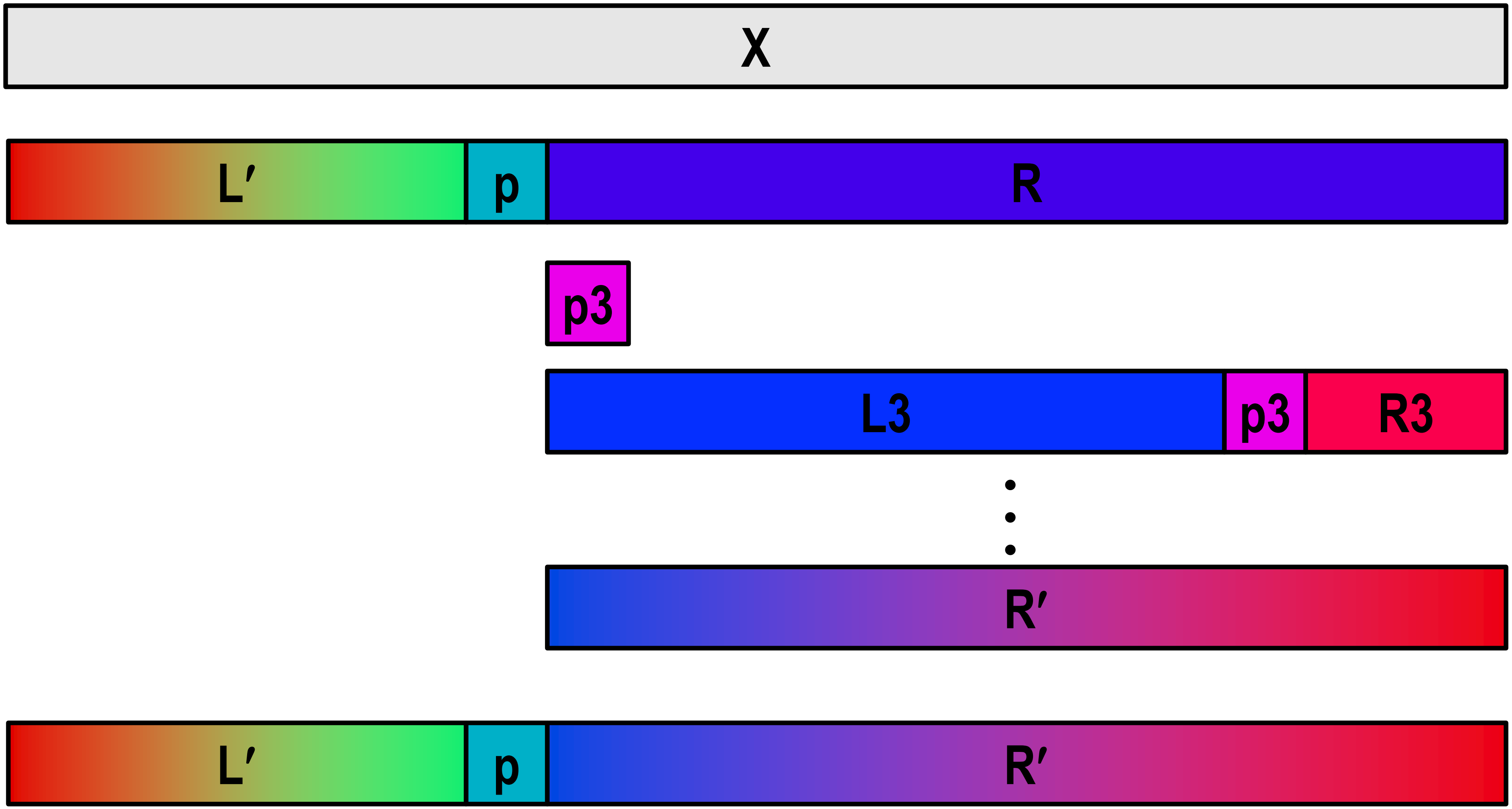
A More Substantial Example: Sort

- **Sort set of N random numbers**
- **Multiple possible algorithms**
  - Use parallel version of quicksort
- **Sequential quicksort of set of values X**
  - Choose “pivot” p from X
  - Rearrange X into
    - L: Values $\leq$ p
    - R: Values $\geq$ p
  - Recursively sort L to get L′
  - Recursively sort R to get R′
  - Return L′ : p : R′

Sequential Quicksort Visualized



Sequential Quicksort Visualized



Sequential Quicksort Code

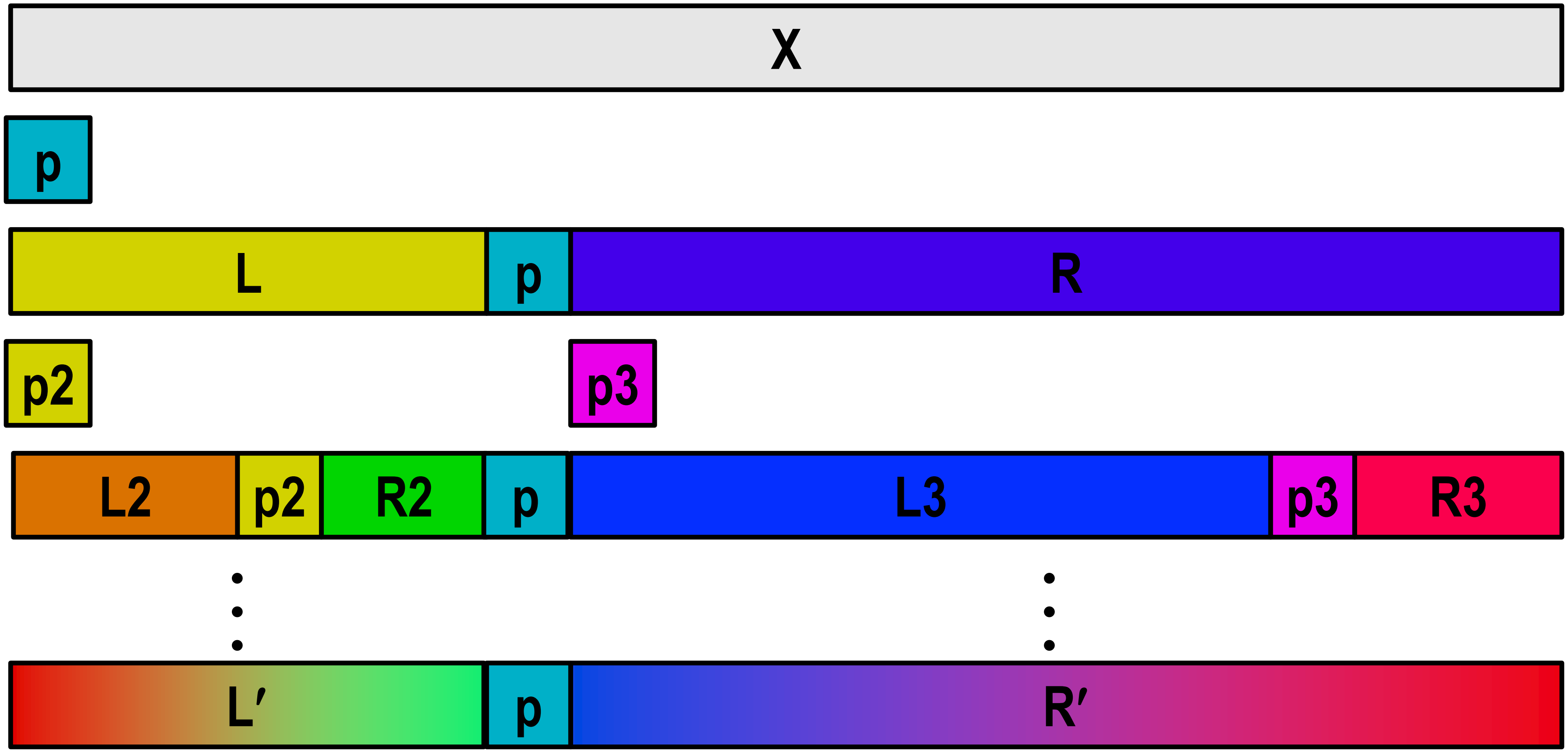
```
void qsort_serial(data_t *base, size_t nele) {
  if (nele <= 1)
    return;
  if (nele == 2) {
    if (base[0] > base[1])
      swap(base, base+1);
    return;
  }

  /* 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);
}
```

- **Sort nele elements starting at base**
  - Recursively sort L or R if has more than one element

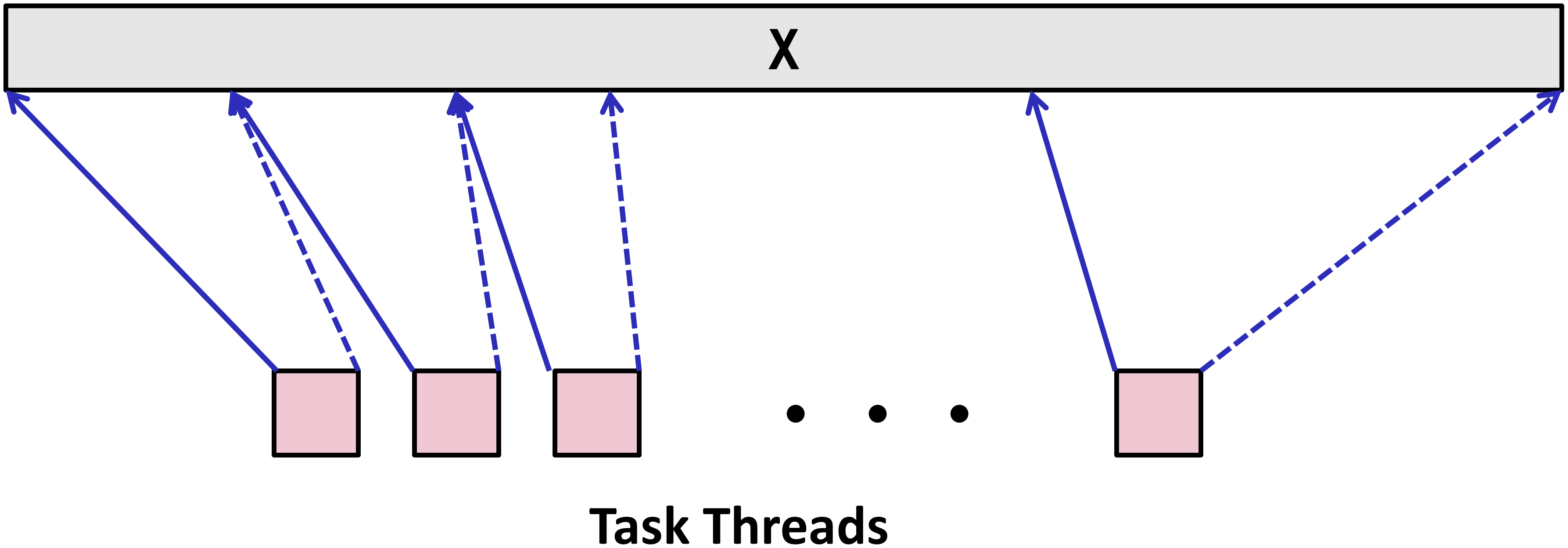
Parallel Quicksort

- **Parallel quicksort of set of values X**
  - If N ≤ Nthresh, do sequential quicksort
  - Else
    - Choose "pivot" p from X
    - Rearrange X into
      - L: Values ≤ p
      - R: Values ≥ p
    - Recursively spawn separate threads
      - Sort L to get L′
      - Sort R to get R′
    - Return L′ : p : R′

Parallel Quicksort Visualized

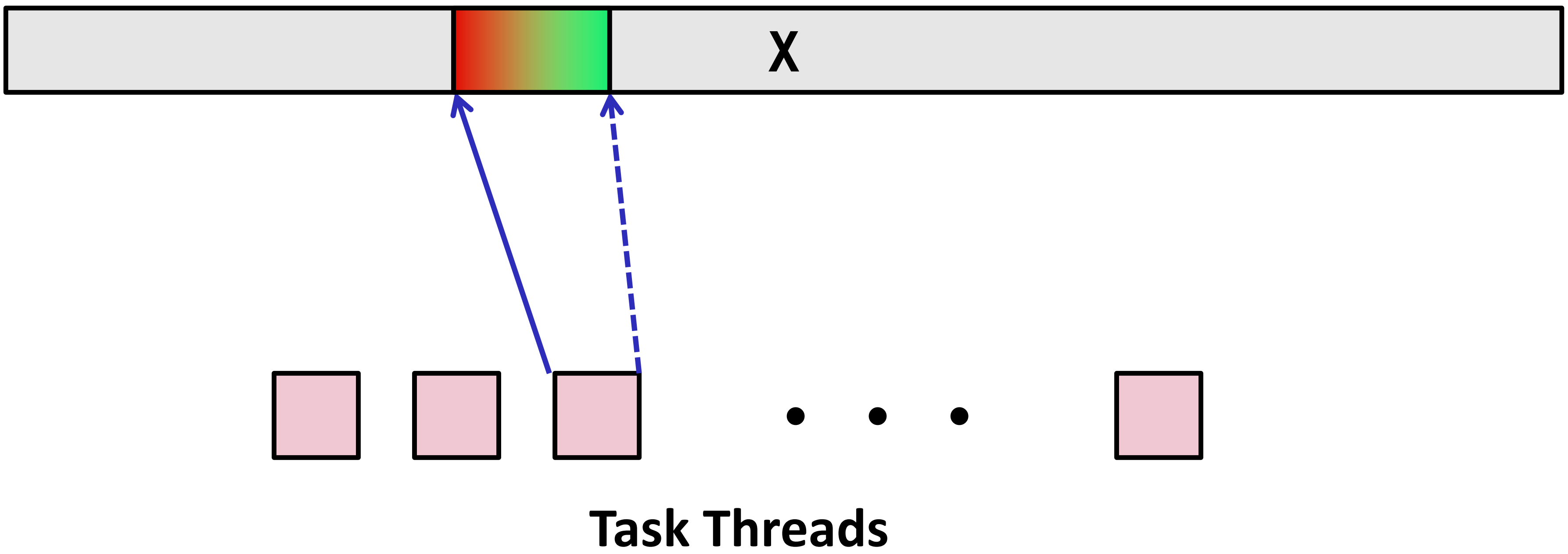


Thread Structure: Sorting Tasks



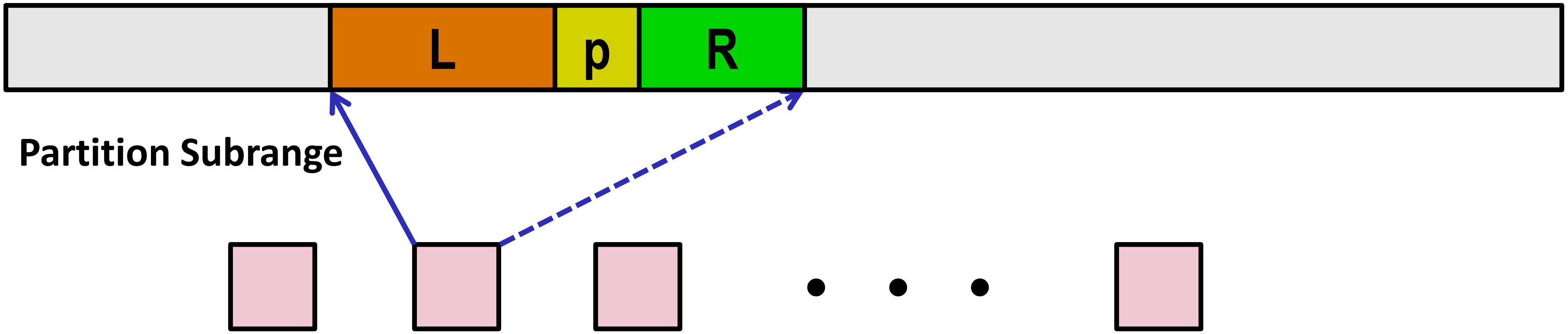
- **Task: Sort subrange of data**
  - Specify as:
    - **`base`**: Starting address
    - **`nele`**: Number of elements in subrange
- **Run as separate thread**

Small Sort Task Operation

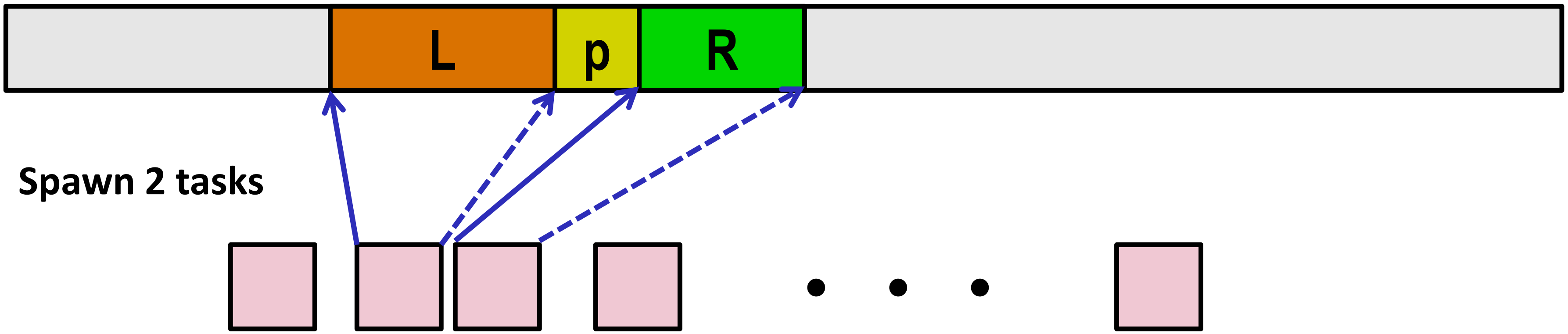


- Sort subrange using serial quicksort

Large Sort Task Operation



Partition Subrange



Spawn 2 tasks

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);
}
```

- **Sets up data structures**
- **Calls recursive sort routine**
- **Keeps joining threads until none left**
- **Frees data structures**

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);
}
```

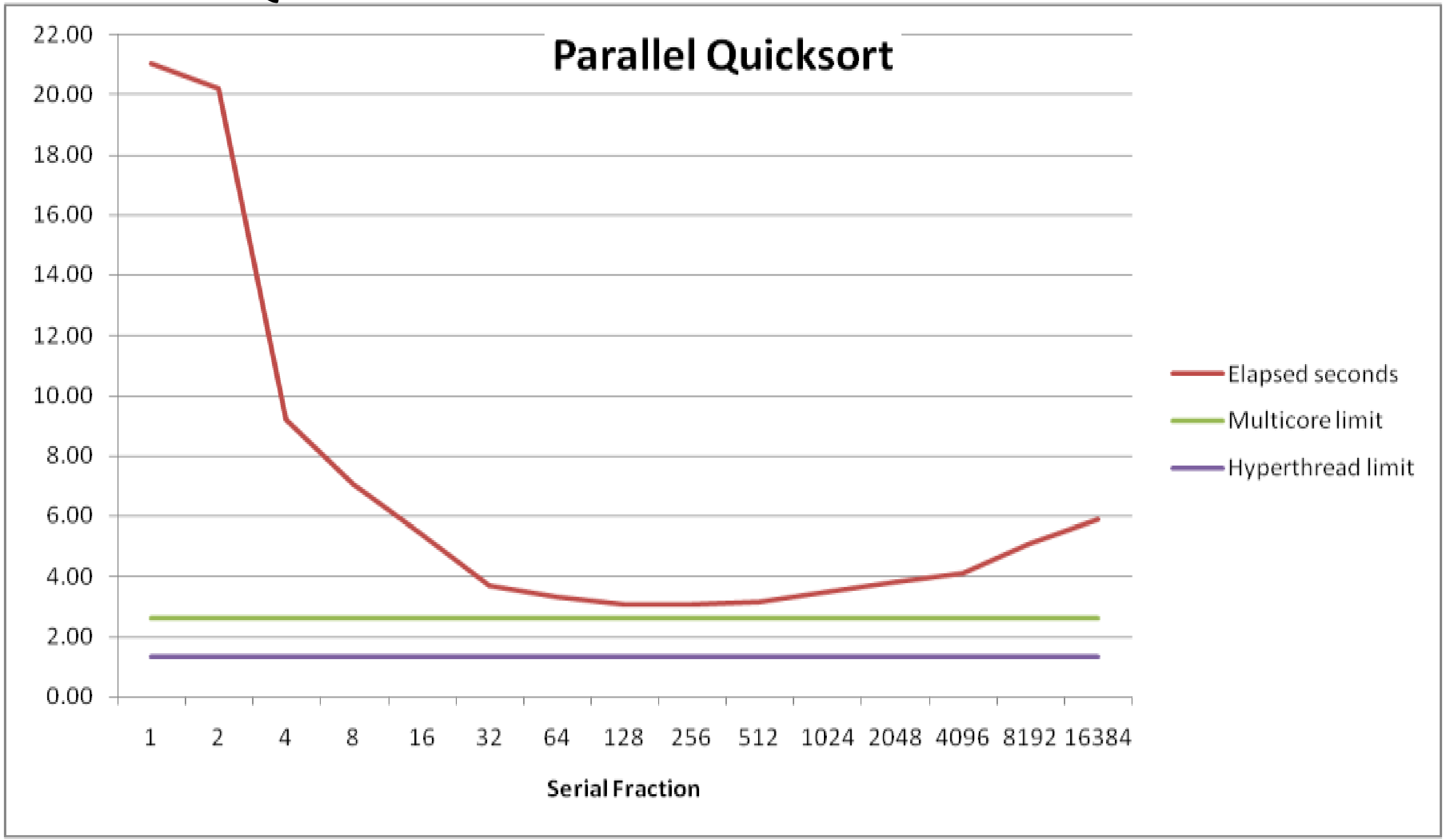
- Small partition: Sort serially
- Large partition: Spawn new sort task

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;
}
```

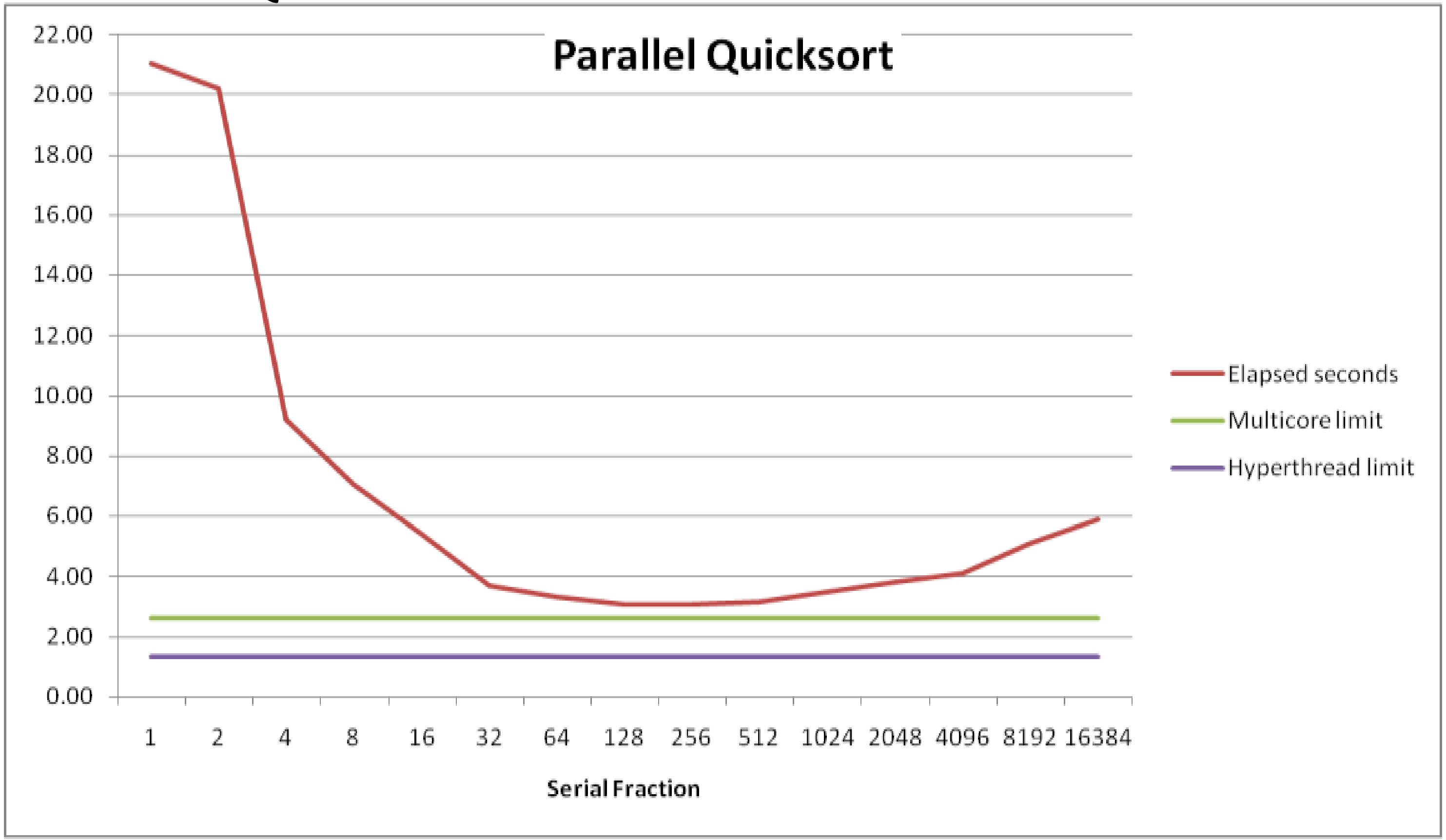
- Get task parameters
- Perform partitioning step
- Call recursive sort routine on each partition

Parallel Quicksort Performance



- Serial fraction: Fraction of input at which do serial sort
- Sort 2^{27} (134,217,728) random values
- Best speedup = 6.84X

Parallel Quicksort Performance

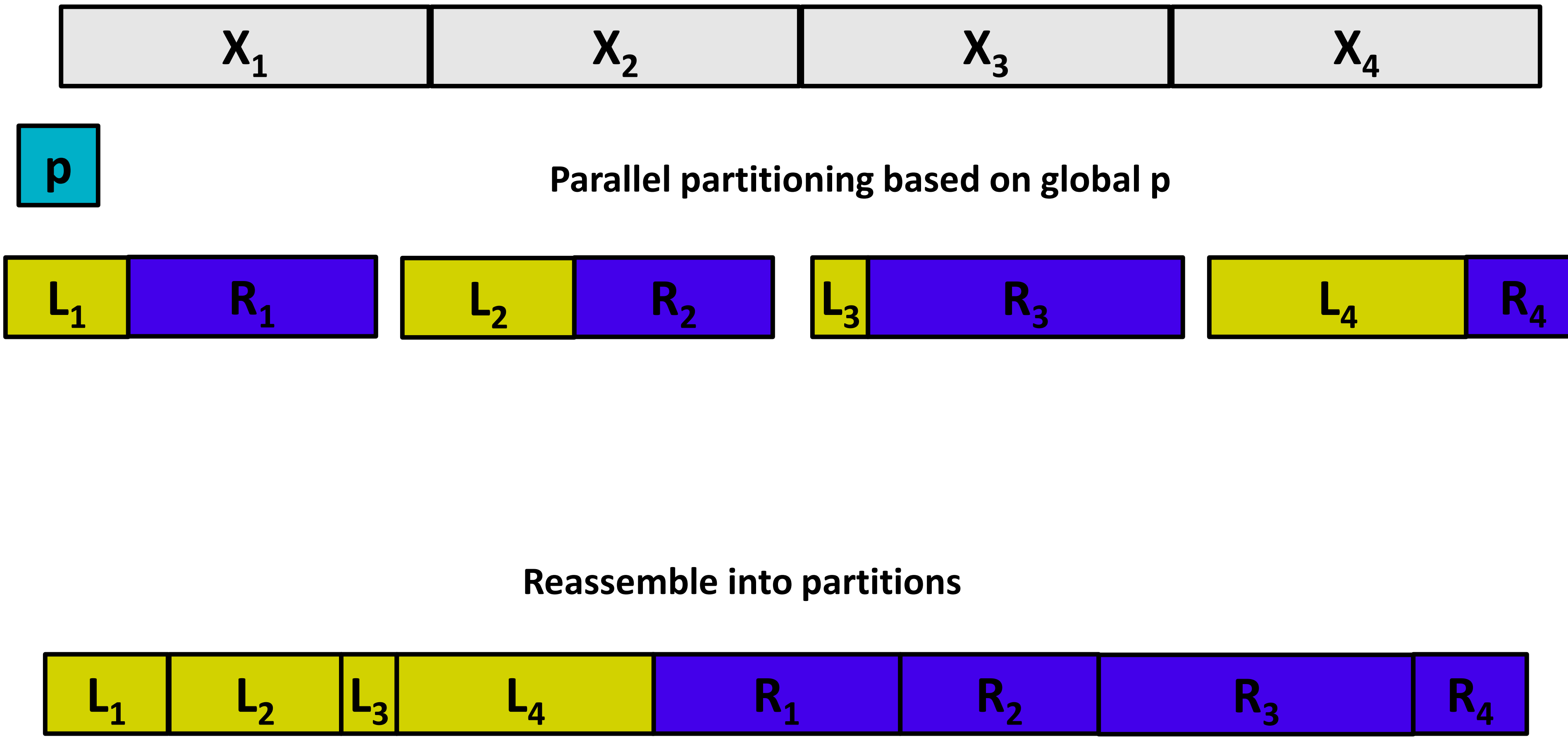


- **Good performance over wide range of fraction values**
  - F too small: Not enough parallelism
  - F too large: Thread overhead + run out of thread memory

Amdahl's Law & Parallel Quicksort

- **Sequential bottleneck**
  - Top-level partition: No speedup
  - Second level: $\leq$ 2X speedup
  - kth level: $\leq 2^{k-1}$X speedup
- **Implications**
  - Good performance for small-scale parallelism
  - Would need to parallelize partitioning step to get large-scale parallelism
    - Parallel Sorting by Regular Sampling
      - H. Shi & J. Schaeffer, J. Parallel & Distributed Computing, 1992

Parallelizing Partitioning Step



Experience with Parallel Partitioning

- **Could not obtain speedup**
- **Speculate: Too much data copying**
  - Could not do everything within source array
  - Set up temporary space for reassembling partition

Lessons Learned

- **Must have parallelization strategy**
  - Partition into K independent parts
  - Divide-and-conquer
- **Inner loops must be synchronization free**
  - Synchronization operations very expensive
- **Beware of Amdahl's Law**
  - Serial code can become bottleneck
- **You can do it!**
  - Achieving modest levels of parallelism is not difficult
  - Set up experimental framework and test multiple strategies

Memory Consistency

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

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

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

Thread consistency constraints

Wa → **Rb**

Wb → **Ra**

- **What are the possible values printed?**
  - Depends on memory consistency model
  - Abstract model of how hardware handles concurrent accesses
- **Sequential consistency**
  - Overall effect consistent with each individual thread
  - Otherwise, arbitrary interleaving

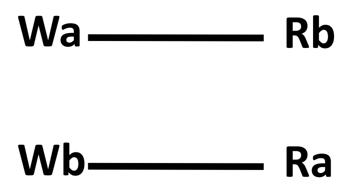
Sequential Consistency Example

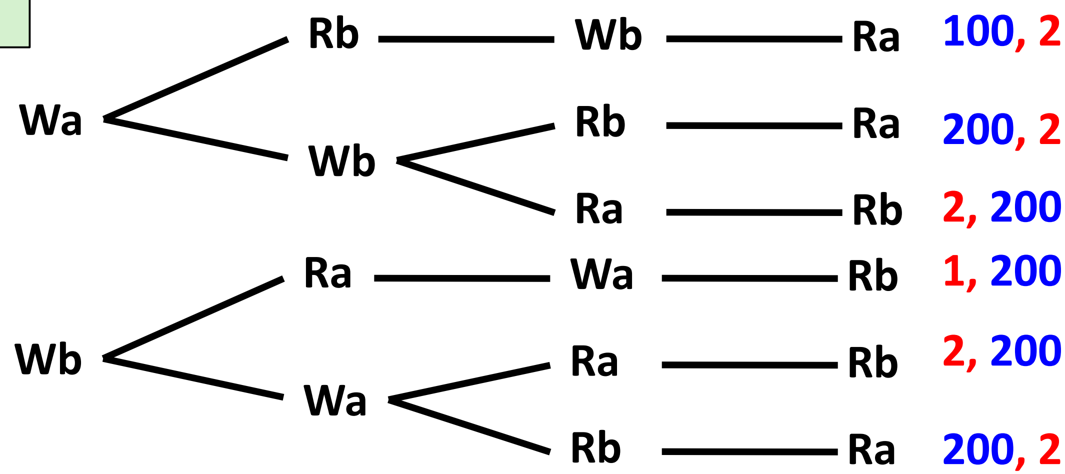
```
int a = 1;
int b = 100;
```

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

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

Thread consistency constraints





- **Impossible outputs**
  - 100, 1 and 1, 100
  - Would require reaching both Ra and Rb before Wa and Wb

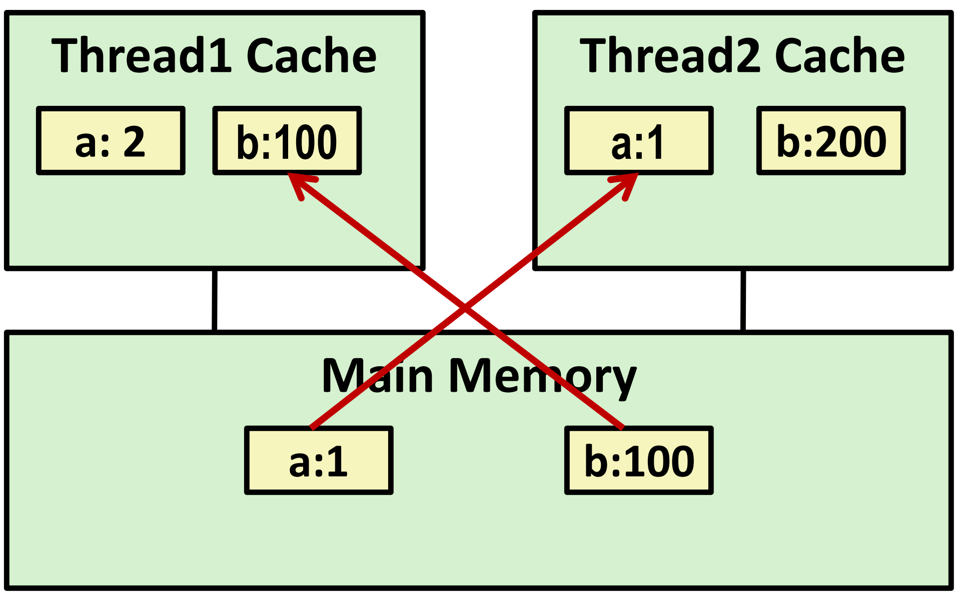
Non-Coherent Cache Scenario

- 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

Snoopy Caches

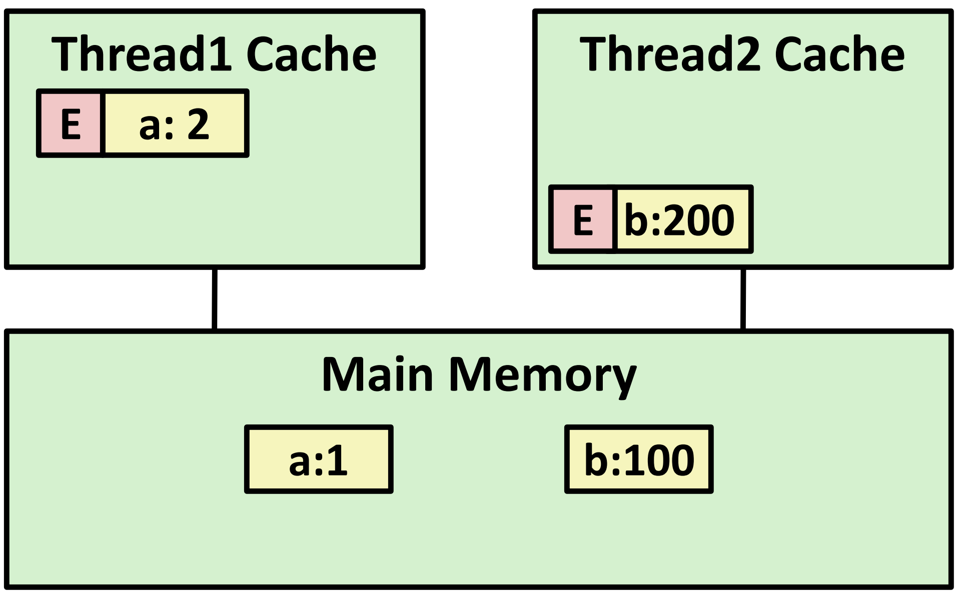
- **Tag each cache block with state**

Invalid	Cannot use value
Shared	Readable copy
Exclusive	Writeable copy

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

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

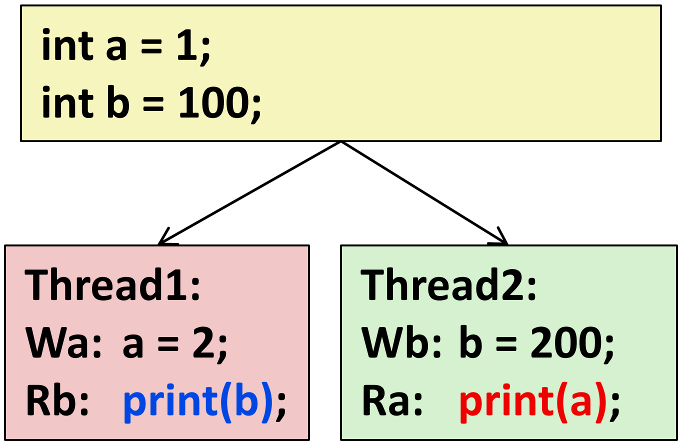
```
Thread2:
Wb:  b = 200;
Ra:   print(a);
```

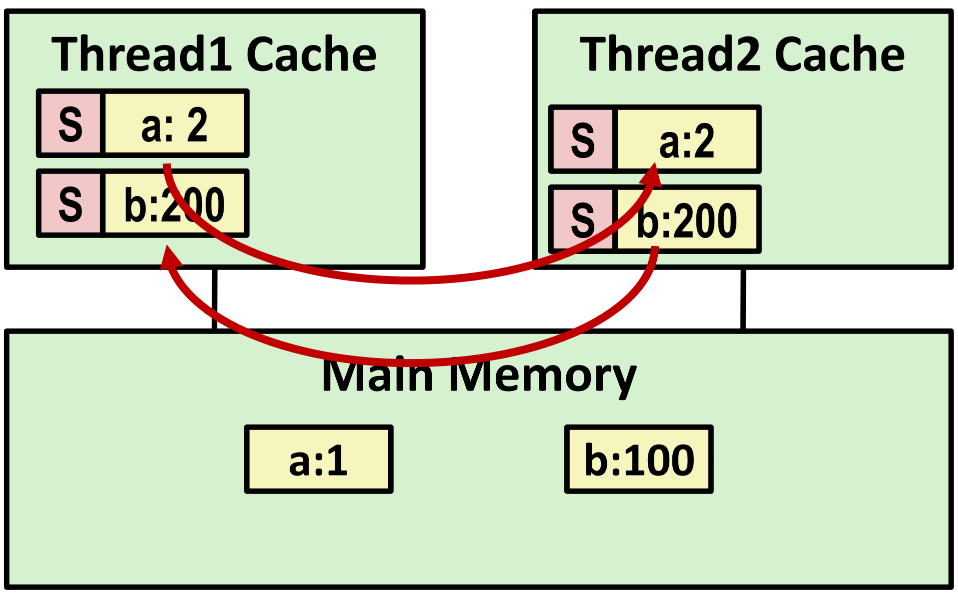


Snoopy Caches

- **Tag each cache block with state**

Invalid	Cannot use value
Shared	Readable copy
Exclusive	Writeable copy





print 2

print 200

- When cache sees request for one of its E-tagged blocks
  - Supply value from cache
  - Set tag to S