Thread-Level Parallelism

15-213 / 18-213: Introduction to Computer Systems
26th Lecture, Apr. 26, 2012

Instructors:
Todd Mowry and Anthony Rowe

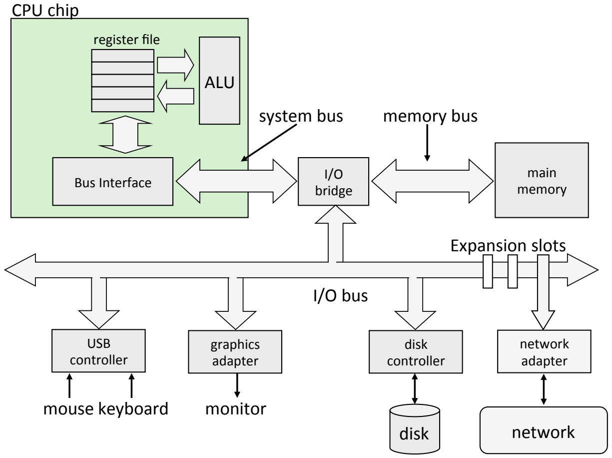
Today

- **Parallel Computing Hardware**
  - Multicore
    - Multiple separate processors on single chip
  - Hyperthreading
    - Multiple threads executed on a given processor at once
- **Thread-Level Parallelism**
  - Splitting program into independent tasks
    - Example: Parallel summation
    - Some performance artifacts
  - Divide-and conquer parallelism
    - Example: Parallel quicksort

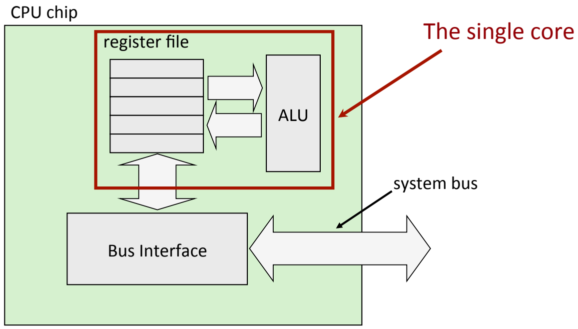
Why Multi-Core?

- Traditionally, single core performance is improved by increasing the clock frequency...
- ...and making deeply pipelined circuits...
- Which leads to...
  - Heat problems
  - Speed of light problems
  - Difficult design and verification
  - Large design teams
  - Big fans, heat sinks
  - Expensive air-conditioning on server farms
- Increasing clock frequency no longer the way to go forward

Single Core Computer

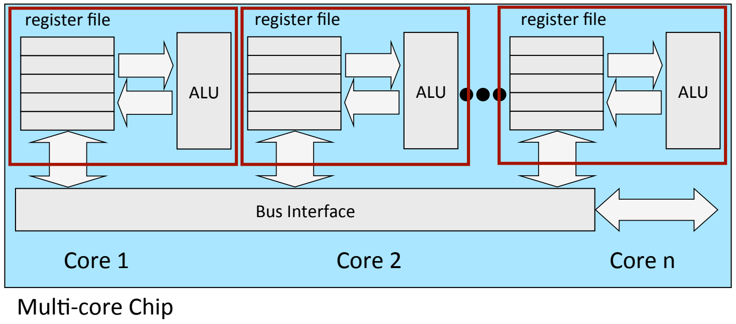


Single Core Processor (chip)

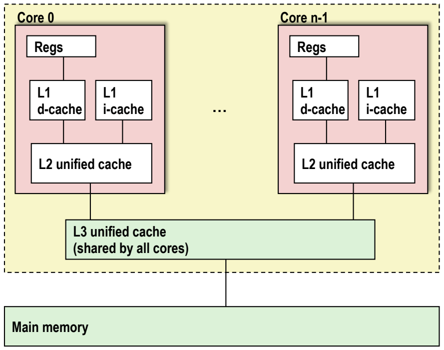


Multi-Core Architecture

- Somewhat recent trend in computer architecture
- Replicate many cores on a single die



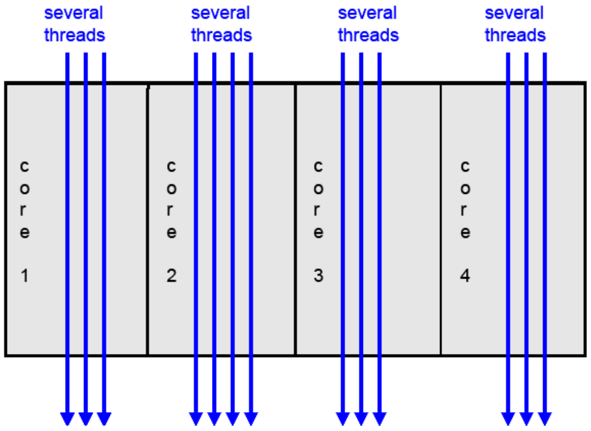
Multi-core Processor



- **Intel Nehalem Processor**
  - E.g., Shark machines
  - Multiple processors operating with coherent view of memory

---

Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction With the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today:
  - Mac OS X, Linux, Windows, …

Flavors of Parallelism

- Instruction Level Parallelism (ILP)
- Thread Level Parallelism (TLP)
- Simultaneous Multi-Threading (SMT)

Instruction-Level Parallelism

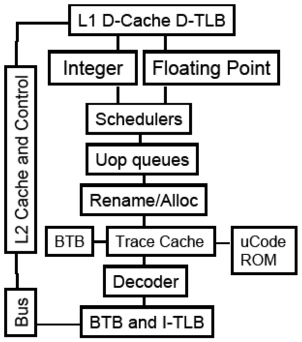
- Parallelism at the machine-instruction level
- Achieved in the processor with
  - Pipeline
  - Re-ordered instructions
  - Split into micro-instructions
  - Aggressive branch prediction
  - Speculative execution
- ILP enabled rapid increases in processor performance
  - Has since plateaued

Thread-level Parallelism

- Parallelism on a coarser scale
- Server can serve each client in a separate thread
  - Web server, database server
- Computer game can do AI, graphics, physics, UI in four different threads
- Single-core superscalar processors cannot fully exploit TLP
  - Thread instructions are interleaved on a coarse level with other threads
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP

Simultaneous Multithreading (SMT)

- Complimentary technique to multi-core
- Addresses the stalled pipeline problem
  - Pipeline is stalled waiting for the result of a long operation (float?)
  - ... or waiting for data to arrive from memory (long latency)
- Other execution units are idle


L1 D-Cache D-TLB
Integer
Floating Point
Schedulers
Uop queues
Rename/Alloc
BTB
Trace Cache
uCode ROM
Decoder
BTB and I-TLB
L2 Cache and Control
Bus

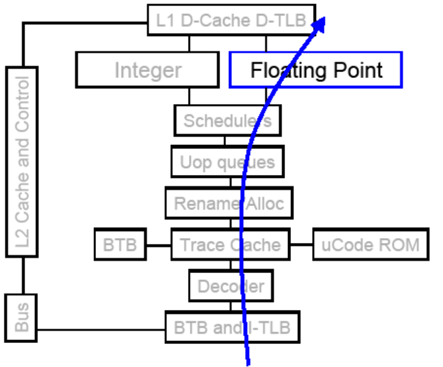
Source: Intel

SMT

- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple “threads”
- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

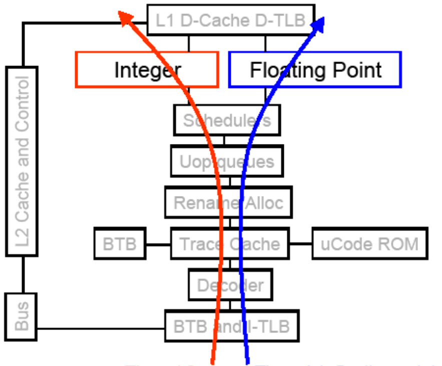
---

Without SMT, only a single thread can run at any given time


L1 D-Cache D-TLB
Integer
Floating Point
Schedulers
Uop queues
Rename/Alloc
BTB
Trace Cache
uCode ROM
Decoder
BTB and I-TLB
L2 Cache and Control
Bus

Thread 1: floating point

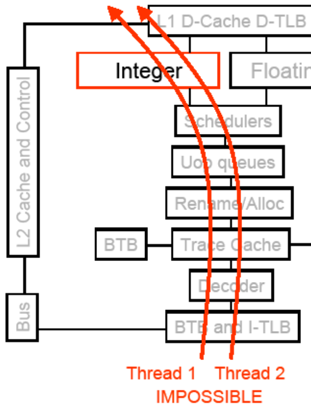
SMT processor: both threads can run concurrently


L1 D-Cache D-TLB
Integer
Floating Point
Schedulers
Uop queues
Rename/Alloc
BTB
Trace Cache
uCode ROM
Decoder
BTB and I-TLB
L2 Cache and Control
Bus

Thread 2: integer operation

Thread 1: floating point

But: Can't simultaneously use the same functional unit

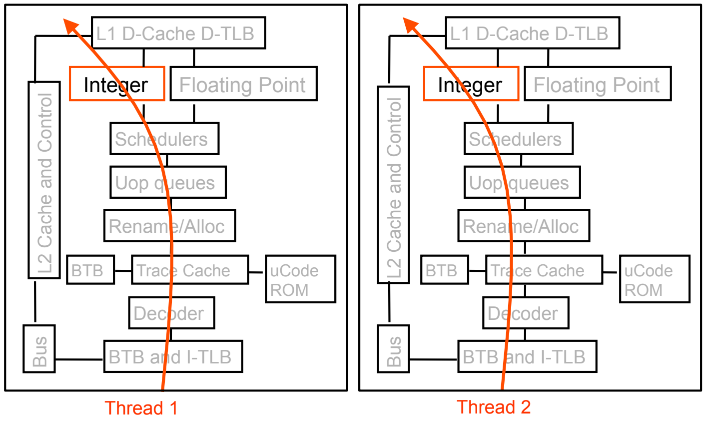


This scenario is impossible with SMT on a single core (assuming a single integer unit)

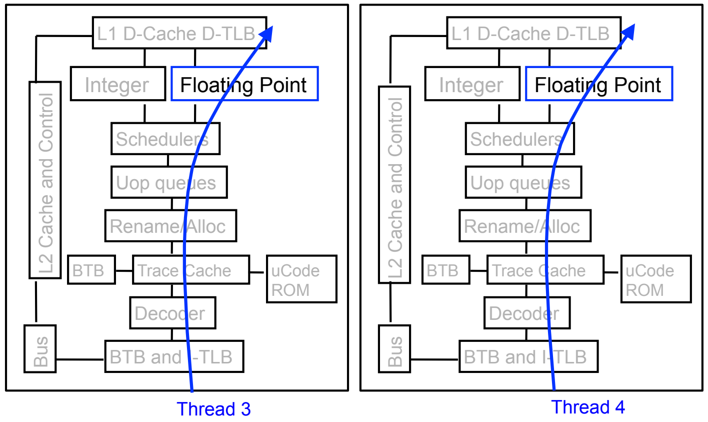
SMT is not a "true" parallel processor

- Enables better threading (e.g. up to 30%)
- OS and applications perceive each simultaneous thread as a separate "virtual processor"
- The chip has only a single copy of each resource
- Compare to multi-core:
  - Each core has its own copy of resources

Multi-core: Threads run on separate cores



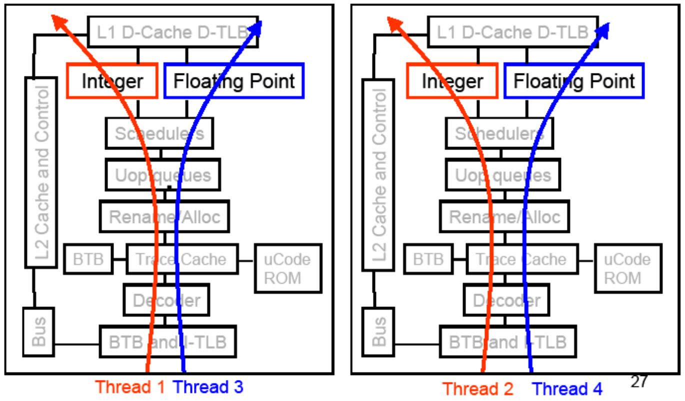
Multi-core: Threads run on separate cores



Combining Multi-core and SMT

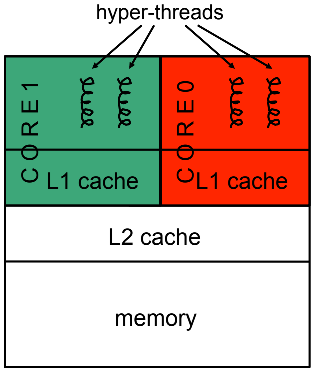
- Cores can be SMT-enabled (or not)
- The different combinations:
  - Single-core, non-SMT: standard uniprocessor
  - Single-core, with SMT
  - Multi-core, non-SMT
  - Multi-core, with SMT: our fish machines
- The number of SMT threads is determined by hardware design
  - 2, 4 or sometimes 8 simultaneous threads
- Intel calls them “Hyper-threads”

SMT Dual-core: all four threads can run concurrently

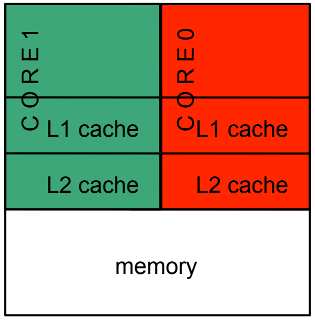


SMT/Multi-Core and the Memory Hierarchy

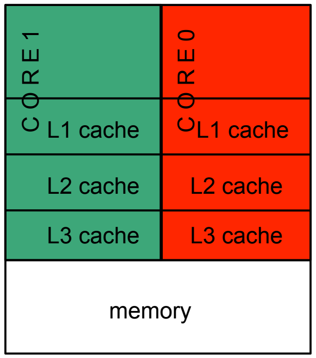
- SMT is a sharing of pipeline resources
  - Thus all caches are shared
- Multi-core chips:
  - L1 caches are private (i.e. each core has its own L1)
  - L2 cache private in some architectures, shared in others
  - Main memory is always shared
- Example: Fish machines
  - Dual-core Intel Xeon processors
  - Each core is hyper-threaded
  - Private L1, shared L2 caches



Designs with Private L2 Caches



Examples: AMD Opteron, AMD Athlon, Intel Pentium D



Example: Intel Itanium 2

Quad Core 2 Duo shares L2 in pairs of cores

Private vs Shared Cache

- Advantages of Private Cache
  - Closer to the core, so faster access
  - No contention for core access -- no waiting while another core accesses
- Advantages of Shared Cache
  - Threads on different cores can share same cache data
  - More cache space is available if a single (or a few) high-performance threads run
- Cache Coherence Problem
  - The same memory value can be stored in multiple private caches
  - Need to keep the data consistent across the caches
  - Many solutions exist
    - Invalidation protocol with bus snooping, ...

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 CPUs offer another opportunity**
  - Spread work over threads executing in parallel on N cores
  - 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
- **Shark machines can execute 16 threads at once**
  - 8 cores, each with 2-way hyperthreading
  - Theoretical speedup of 16X
    - never achieved in our benchmarks

Summation Example

- **Sum numbers 0, …, N-1**
  - Should add up to (N-1)*N/2
- **Partition into K ranges**
  - ⌊N/K⌋ values each
  - Accumulate leftover values serially
- **Method #1: All threads update single global variable**
  - 1A: No synchronization
  - 1B: Synchronize with pthread semaphore
  - 1C: Synchronize with pthread mutex
    - "Binary" semaphore. Only values 0 & 1

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];
```

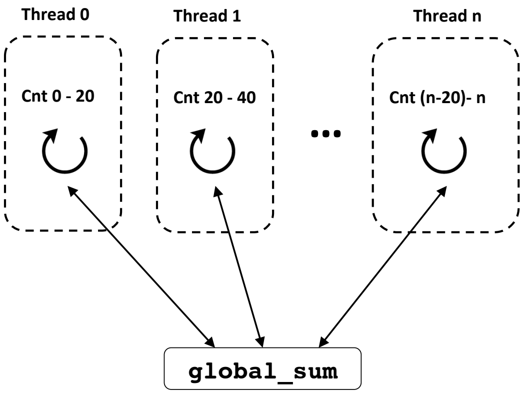
Accumulating in Single Global Variable: Operation

```
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;
```

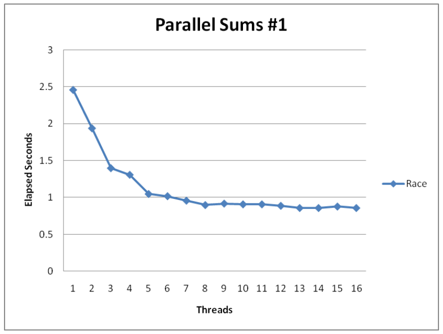
Thread Function: No Synchronization

```
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;
}
```

Accumulating in Single Global Variable: Illustration



Unsynchronized Performance



- N = 2^{30}
- Best speedup = 2.86X
- Gets wrong answer when > 1 thread!

Thread Function: Semaphore / Mutex

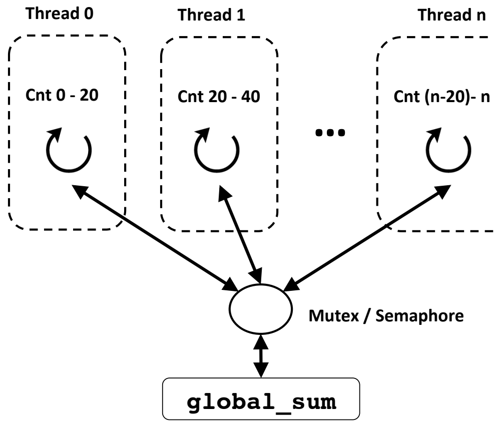
Semaphore

```
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;
}
```

Mutex

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

Accumulating with Mutex / Semaphore



Thread 0

Cnt 0 - 20

Thread 1

Cnt 20 - 40

•••

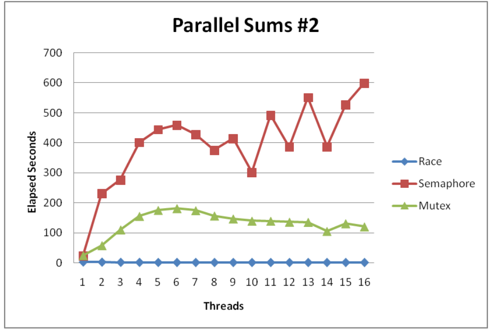
Thread n

Cnt (n-20)- n

Mutex / Semaphore

`global_sum`

Semaphore / Mutex Performance



- **Terrible Performance**
  - 2.5 seconds ➔ ~10 minutes
- **Mutex 3X faster than semaphore**
- **Clearly, neither is successful**

Separate Accumulation

- **Method #2: Each thread accumulates into separate variable**
  - 2A: Accumulate in contiguous array elements
  - 2B: Accumulate in spaced-apart array elements
  - 2C: Accumulate in registers

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

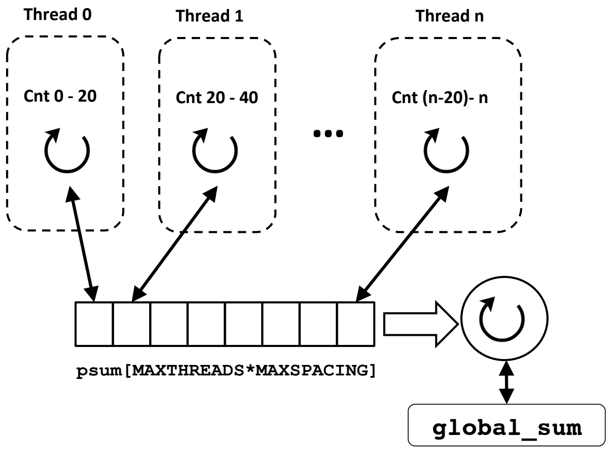
Separate Accumulation: Operation

```
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;
```

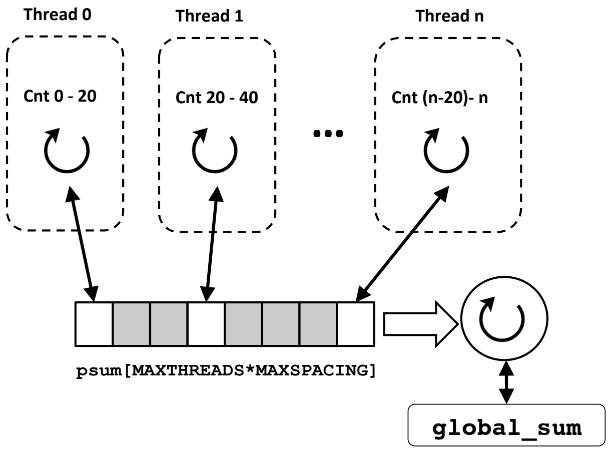
Thread Function: Memory Accumulation

```
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;
}
```

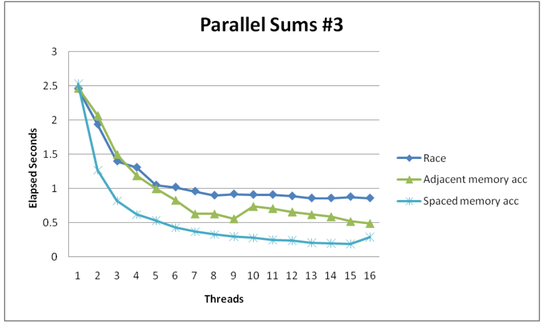
Accumulating into memory (no spacing)



Accumulating into memory (spacing)

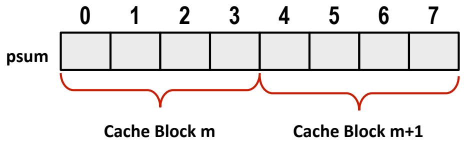


Memory Accumulation Performance



- **Clear threading advantage**
  - Adjacent speedup: 5 X
  - Spaced-apart speedup: 13.3 X (Only observed speedup > 8)
- **Why does spacing the accumulators apart matter?**

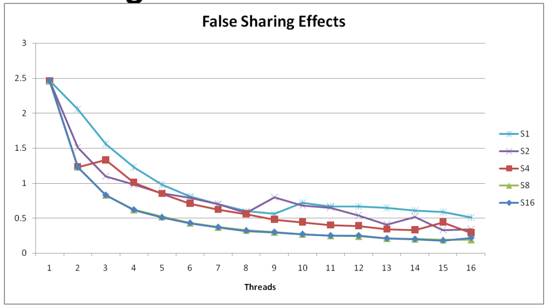
False Sharing



- **Coherency maintained on cache blocks**
- **To update psum[i], thread i must have exclusive access**
  - Threads sharing common cache block will keep fighting each other for access to block

---

False Sharing Performance

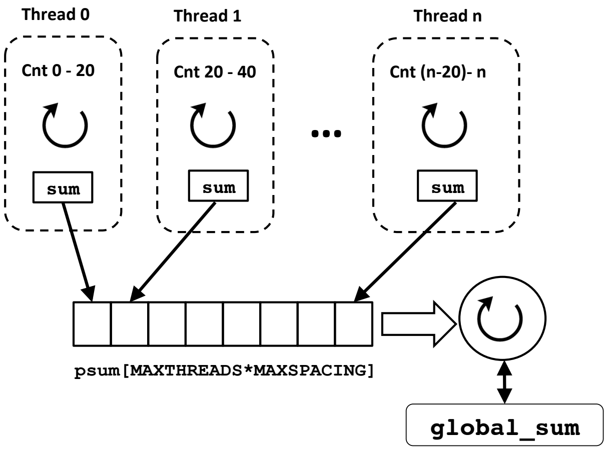


  - Best spaced-apart performance 2.8 X better than best adjacent
- **Demonstrates cache block size = 64**
  - 8-byte values
  - No benefit increasing spacing beyond 8

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;
}
```

Accumulating into register



Thread 0

Thread 1

Thread n

Cnt 0 - 20

Cnt 20 - 40

Cnt (n-20)- n

`sum`

`sum`

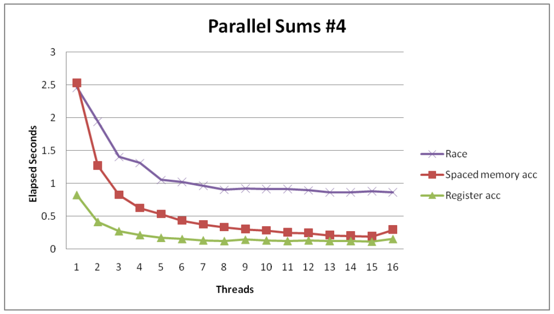
`sum`

…

`psum[MAXTHREADS*MAXSPACING]`

`global_sum`

Register Accumulation Performance



- **Clear threading advantage**
  - Speedup = 7.5 X
- **2X better than fastest memory accumulation**

Amdahl's Law

- **Overall problem**
  - T Total time required
  - p Fraction of total that can be sped up ($0 \le p \le 1$)
  - 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
  - Maximum possible speedup
    - $k = \infty$
    - $T_\infty = (1-p)T$

Amdahl's Law Example

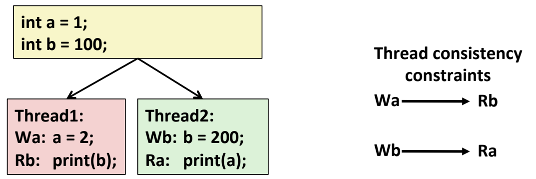
- **Overall problem**
  - T = 10 Total time required
  - p = 0.9 Fraction of total which can be sped up
  - k = 9 Speedup factor
- **Resulting Performance**
  - $T_9 = 0.9 * 10/9 + 0.1 * 10 = 1.0 + 1.0 = 2.0$
  - Maximum possible speedup
    - $T_\infty = 0.1 * 10.0 = 1.0$

Memory Consistency

- **There are different memory consistency models**
  - Abstract model of how hardware handles concurrent accesses
- **Most systems provide "sequential consistency"**
  - Overall effect consistent with each individual thread
  - But, the threads can be interleaved in any way
    - like when one-thread-at-a-time, but with constant interleaving
- **So, no correctness effects**
  - But, there can be performance effects
    - related to keeping cached values consistent
    - copying data from one cache to another is sorta like a cache miss

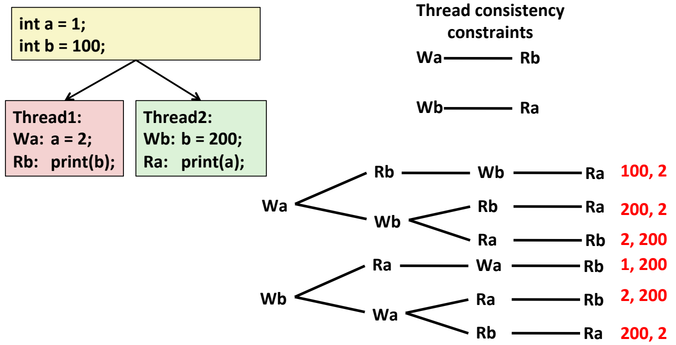
---

Memory Consistency



- **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

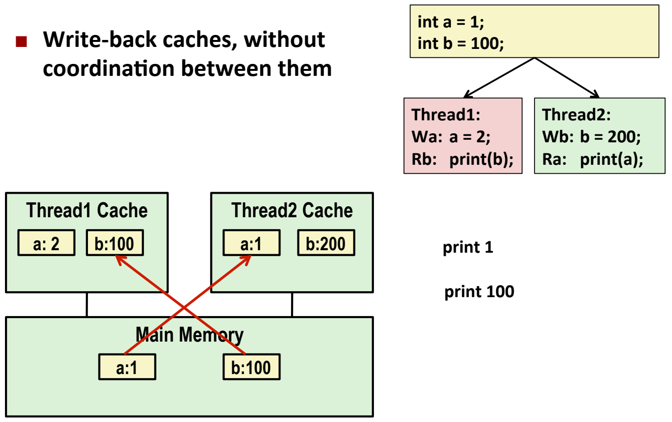
Sequential Consistency Example

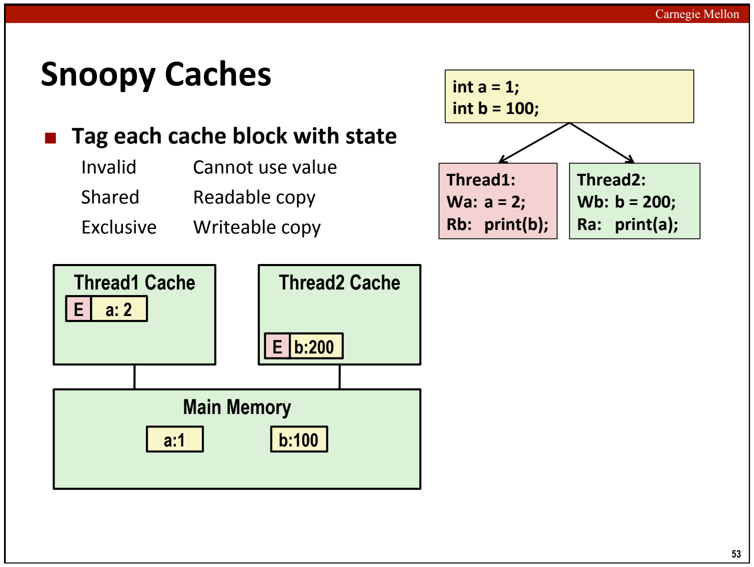


- **Impossible outputs**
  - 100, 1 and 1, 100
  - Would require reaching both Ra and Rb before Wa and Wb

Non-Coherent Cache Scenario

- **Write-back caches, without coordination between them**





Snoopy Caches

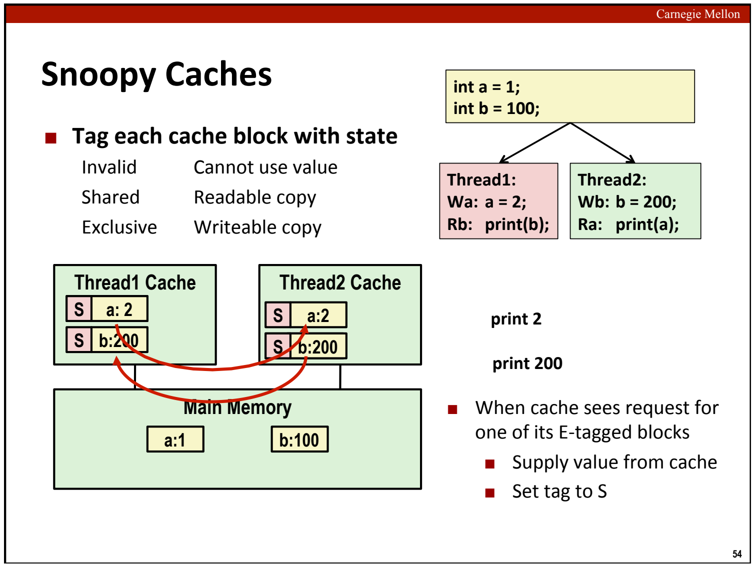
- **Tag each cache block with state**

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

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

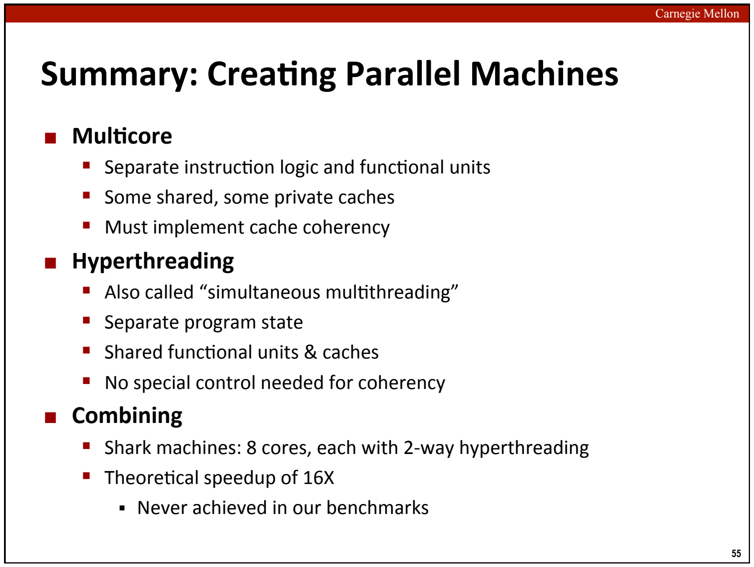
```
Thread1:
Wa:  a = 2;
Rb:  print(b);
```

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

print 2

print 200

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



Summary: Creating Parallel Machines

- **Multicore**
  - Separate instruction logic and functional units
  - Some shared, some private caches
  - Must implement cache coherency
- **Hyperthreading**
  - Also called “simultaneous multithreading”
  - Separate program state
  - Shared functional units & caches
  - No special control needed for coherency
- **Combining**
  - Shark machines: 8 cores, each with 2-way hyperthreading
  - Theoretical speedup of 16X
    - Never achieved in our benchmarks