Thread-Level Parallelism

15-213 / 18-213: Introduction to Computer Systems
“26th” Lecture, August 1, 2018

Instructor:
Sol Boucher
Today

- **Parallel Computing Hardware**
  - Multicore
    - Multiple separate processors on single chip
  - Hyperthreading
    - Efficient execution of multiple threads on single core

- **Consistency Models**
  - What happens when multiple threads are reading & writing shared state

- **Thread-Level Parallelism**
  - Splitting program into independent tasks
    - Example: Parallel summation
    - Examine some performance artifacts
  - Divide-and-conquer parallelism
    - Example: Parallel quicksort
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 instruction control to process K instruction streams
- K copies of all registers
- Share functional units
Benchmark Machine

- Get CPU specs from `lscpu` or `/proc/cpuinfo`
- Shark Machines
  - Intel Xeon E5520 @ 2.27 GHz
  - Nehalem, ca. 2010
  - 8 Cores
  - Each can do 2x hyperthreading
Exploiting parallel execution

- So far, we’ve used threads to handle multiple clients’ I/O
- Multicore 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
What are the possible values printed?

- Depends on memory consistency model
- Abstract model of how hardware handles concurrent accesses
Non-Coherent Cache Scenario

- Write-back caches, without coordination between them

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

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

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

print 1

print 100
Memory Consistency

What are the possible values printed?
- Depends on memory consistency model
- Abstract model of how hardware handles concurrent accesses
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

**Thread consistency constraints**

- **Wa** → **Rb**
- **Wb** → **Ra**

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

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

#### 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
```
Non-Sequentially Consistent Scenario

- Thread consistency constraints violated due to out-of-order execution

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

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

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

- Fix: Add SFENCE instructions between Wa & Rb and Wb & Ra
Snoopy Caches

- Tag each cache block with state
  - Invalid: Cannot use value
  - Shared: Readable copy
  - Exclusive: Writeable copy

```java
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

```java
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
Memory Models

- **Sequentially Consistent:**
  - Each thread executes in proper order, any interleaving
- **To ensure, requires**
  - Proper cache/memory behavior
  - Proper intra-thread ordering constraints
Today

- Parallel Computing Hardware
  - Multicore
    - Multiple separate processors on single chip
  - Hyperthreading
    - Efficient execution of multiple threads on single core

- Consistency Models
  - What happens when multiple threads are reading & writing shared state

- Thread-Level Parallelism
  - Splitting program into independent tasks
    - Example: Parallel summation
    - Examine some performance artifacts
  - Divide-and-conquer parallelism
    - Example: Parallel quicksort
Summation Example

- **Sum numbers 0, ..., N-1**
  - Should add up to \((N-1) \times N/2\)

- **Partition into K ranges**
  - \(\lfloor N/K \rfloor\) values each
  - Each of the \(t\) threads processes 1 range
  - 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;
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;
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

```c
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

```c
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;
}
```
Unsynchronized Performance

- $N = 2^{30}$
- Best speedup = 2.86X
- Gets wrong answer when > 1 thread!
Thread Function: Semaphore / Mutex

Semaphore

```c
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

```c
pthread_mutex_lock(&mutex);
    global_sum += i;
    pthread_mutex_unlock(&mutex);
```
Semaphore / Mutex Performance

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

What is main reason for poor performance?
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

```c
/* Partial sum computed by each thread */
data_t psum[MAXTHREADS*MAXSPACING];

/* Spacing between accumulators */
size_t spacing = 1;
```
Separate Accumulation: Operation

```c
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

Where is the mutex?

```c
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;
}
```
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

```c
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;
}
```
Register Accumulation Performance

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

Beware the speedup metric!
Lessons learned

- Sharing memory can be expensive
  - Pay attention to true sharing
  - Pay attention to false sharing

- Use registers whenever possible
  - (Remember cachelab)
  - Use local cache whenever possible

- Deal with leftovers

- When examining performance, compare to best possible sequential implementation
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 ≤ p
    - R: Values ≥ p
  - Recursively sort L to get L´
  - Recursively sort R to get R´
  - Return L´ : p : R´
Sequential Quicksort Visualized

X

p

L p R

p2

L2 p2 R2

L′
Sequential Quicksort Visualized
Sequential Quicksort Code

```c
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 $N \leq N\text{thresh}$, do sequential quicksort
  - Else
    - Choose “pivot” $p$ from $X$
    - Rearrange $X$ into
      - $L$: Values $\leq p$
      - $R$: Values $\geq p$
    - Recursively spawn separate threads
      - Sort $L$ to get $L'$
      - Sort $R$ to get $R'$
    - Return $L' : p : R'$
Parallel Quicksort Visualized

\[ X \]

\[ \begin{array}{c}
   \text{p} \\
   \text{L} \quad \text{p} \quad \text{R} \\
   \text{p2} \quad \text{p3} \\
   \text{L2} \quad \text{p2} \quad \text{R2} \quad \text{p} \quad \text{L3} \quad \ddots \quad \text{p3} \quad \text{R3} \\
   \ddots \quad \ddots \quad \ddots \\
   \text{L'} \quad \text{p} \quad \text{R'}
\end{array} \]
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)

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

```c
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);
}
```
Recursive sort routine (Simplified)

```c
/* 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 $2^{37} (134,217,728)$ 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

**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**
- $T_k = \frac{pT}{k} + (1-p)T$
  - 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**
  - $T_9 = 0.9 \times \frac{10}{9} + 0.1 \times 10 = 1.0 + 1.0 = 2.0$
  - Maximum possible speedup
    - $T_\infty = 0.1 \times 10.0 = 1.0$
Amdahl’s Law & Parallel Quicksort

- **Sequential bottleneck**
  - Top-level partition: No speedup
  - Second level: $\leq 2X$ speedup
  - $k^{th}$ 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
Lessons Learned

- Must have parallelization strategy
  - Partition into K independent parts
  - Divide-and-conquer

- Inner loops must be synchronization free
  - Synchronization operations very expensive

- Watch out for hardware artifacts
  - Need to understand processor & memory structure
  - Sharing and false sharing of global data

- 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