Thread-Level Parallelism

15-213 / 18-213: Introduction to Computer Systems
26th Lecture, April 25, 2013

Instructors:
Seth Copen Goldstein, Anthony Rowe, and Greg Kesden

Today

- Parallel Computing Hardware
  - Multicore
    - Multiple separate processors on single chip
  - Others
- Thread-Level Parallelism
  - Splitting program into independent tasks
    - Example: Parallel summation
  - Some performance artifacts
    - Example: Parallel quicksort

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 (not covered)
  - Theoretical speedup of 16X
    - never achieved in our benchmarks

Multicore Processor

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

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

- **Partition into K ranges**
  - \(\lfloor N/K \rfloor\) 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**

```c
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];
```

**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!

Semaphore / Mutex Performance

- Terrible Performance
  - 2.5 seconds \( \rightarrow \) ~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

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);
```

/* 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

```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

Summary of Summation

<table>
<thead>
<tr>
<th>Method</th>
<th>Seq time</th>
<th>Max Speedup</th>
<th>Speedup on 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Race</td>
<td>2.5</td>
<td>2.9</td>
<td>2.9</td>
</tr>
<tr>
<td>Semaphore</td>
<td>2.5</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Mutex</td>
<td>2.5</td>
<td>1.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Adj Mem</td>
<td>2.5</td>
<td>5.0</td>
<td>4.2</td>
</tr>
<tr>
<td>Adj Mem</td>
<td>2.5</td>
<td>13.3</td>
<td>6.2</td>
</tr>
<tr>
<td>Register Accum</td>
<td>0.8</td>
<td>7.5</td>
<td>7.5</td>
</tr>
</tbody>
</table>
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

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

Parallel Quicksort Data Structures

```c
/* Structure that defines sorting task */
typedef struct {
  data_t *base;
  size_t nele;
  pthread_t tid;
} sort_task_t;

volatile int ntasks = 0;
volatile int ctasks = 0;
sort_task_t **tasks = NULL;

static void init_task(size_t nele) {
  ctasks = 64;
  tasks = (sort_task_t **) Calloc(ctasks, sizeof(sort_task_t *));
  ntasks = 0;
  Sem_init(&tmutex, 0, 1);
  nele_max_serial = nele / serial_fraction;
}
```

- Data associated with each sorting task
  - base: Array start
  - nele: Number of elements
  - tid: Thread ID
- Generate list of tasks
  - Must protect by mutex

Parallel Quicksort Initialization

- Task queue dynamically allocated
- Set \( N_{\text{thresh}} = N/F \):
  - \( N \): Total number of elements
  - \( F \): Serial fraction
    - Fraction of total size at which shift to sequential quicksort
Parallel Quicksort: Accessing Task Queue

```
static sort_task_t *new_task(data_t *base, size_t nele) {
    P(&tmutex);
    if (ntasks == ctasks) {
        ctasks *= 2;
        tasks = (sort_task_t **) Realloc(tasks, ctasks * sizeof(sort_task_t *));
    }
    int idx = ntasks++;
    sort_task_t *t = (sort_task_t *) Malloc(sizeof(sort_task_t));
    tasks[idx] = t;
    V(&tmutex);
    t->base = base;
    t->nele = nele;
    t->tid = (pthread_t) 0;
    return t;
}
```

- Dynamically expand by doubling queue length
  - Generate task structure dynamically (consumed when reap thread)
- Must protect all accesses to queue & ntasks by mutex

Parallel Quicksort: Top-Level Function

```
void tqsort(data_t *base, size_t nele) {
    int i;
    init_task(nele);
    tqsort_helper(base, nele);
    for (i = 0; i < get_ntasks(); i++) {
        P(&tmutex);
        sort_task_t *t = tasks[i];
        V(&tmutex);
        Pthread_join(t->tid, NULL);
        free((void *) t);
    }
}
```

- Actual sorting done by tqsort_helper
- Must reap all of the spawned threads
  - All accesses to task queue & ntasks guarded by mutex

Parallel Quicksort: Recursive function

```
void tqsort_helper(data_t *base, size_t nele) {
    if (nele <= nele_max_serial) {
        /* Use sequential sort */
        qsort_serial(base, nele);
        return;
    }
    sort_task_t *t = new_task(base, nele);
    Pthread_create(&t->tid, NULL, sort_thread, (void *) t);
}
```

- If below Nthresh, call sequential quicksort
- Otherwise create sorting task

Parallel Quicksort: Sorting Task Function

```
static void *sort_thread(void *vargp) {
    sort_task_t *t = (sort_task_t *) vargp;
    data_t *base = t->base;
    size_t nele = t->nele;
    size_t m = partition(base, nele);
    if (m > 1)
        tqsort_helper(base, m);
    if (nele-1 > m+1)
        tqsort_helper(base+m+1, nele-m-1);
    return NULL;
}
```

- Same idea as sequential quicksort
**Parallel Quicksort Performance**

- Sort $2^{37}$ (134,217,728) random values
- Best speedup = 6.84X

**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**
  - $T_k = pT/k + (1-p)T$
    - 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**
  - $T_9 = 0.9 \times 10/9 + 0.1 \times 10 = 1.0 + 1.0 = 2.0$
  - 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 strategy**
  - Partition into \( K \) independent parts
  - Divide-and-conquer

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

- **Watch out for hardware artifacts**
  - Sharing and false sharing of global data

- **You can do it!**
  - Achieving modest levels of parallelism is not difficult

Extra slides

Out-of-Order Processor Structure

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

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

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

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

Main Memory

Thread1 Cache
- a: 2
- b: 100

Thread2 Cache
- a: 1
- b: 200

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

Main Memory

Thread1 Cache
- a: 2
- E

Thread2 Cache
- b: 200
- E

int a = 1;
int b = 100;

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

print 2
print 200

 Crucial concept: Thread Safety

- Functions called from a thread must be thread-safe

  **Def:** A function is thread-safe iff it will always produce correct results when called repeatedly from multiple concurrent threads.

- Classes of thread-unsafe functions:
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions
Thread-Unsafe Functions (Class 1)

- Failing to protect shared variables
  - Fix: Use P and V semaphore operations
  - Example: goodcnt.c
  - Issue: Synchronization operations will slow down code

Thread-Unsafe Functions (Class 2)

- Relying on persistent state across multiple function invocations
  - Example: Random number generator that relies on static state

```c
static unsigned int next = 1;

/* rand: return pseudo-random integer on 0..32767 */
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Thread-Safe Random Number Generator

- Pass state as part of argument
  - and, thereby, eliminate static state

```c
/* rand_r - return pseudo-random integer on 0..32767 */
int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

- Consequence: programmer using rand_r must maintain seed

Thread-Unsafe Functions (Class 3)

- Returning a pointer to a static variable
  - Fix 1. Rewrite function so caller passes address of variable to store result
    - Requires changes in caller and callee
  - Fix 2. Lock-and-copy
    - Requires simple changes in caller (and none in callee)
    - However, caller must free memory.

```c
/* lock-and-copy version */
char *ctime_ts(const time_t *timep, char *privatep)
{
    char *sharedp;
    P(&mutex);
    sharedp = ctime(timep);
    strcpy(privatep, sharedp);
    V(&mutex);
    return privatep;
}
```
Thread-Unsafe Functions (Class 4)

- Calling thread-unsafe functions
  - Calling one thread-unsafe function makes the entire function that calls it thread-unsafe
  - Fix: Modify the function so it calls only thread-safe functions 😊

Reentrant Functions

- Def: A function is **reentrant** iff it accesses no shared variables when called by multiple threads.
  - Important subset of thread-safe functions
    - Require no synchronization operations
    - Only way to make a Class 2 function thread-safe is to make it reentrant (e.g., `rand_r`)

All functions

<table>
<thead>
<tr>
<th>Thread-safe functions</th>
<th>Thread-unsafe functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reentrant functions</td>
<td></td>
</tr>
</tbody>
</table>

Thread-Safe Library Functions

- All functions in the Standard C Library (at the back of your K&R text) are thread-safe
  - Examples: `malloc`, `free`, `printf`, `scanf`
- Most Unix system calls are thread-safe, with a few exceptions:

<table>
<thead>
<tr>
<th>Thread-unsafe function</th>
<th>Class</th>
<th>Reentrant version</th>
</tr>
</thead>
<tbody>
<tr>
<td>asctime</td>
<td>3</td>
<td>asctime_r</td>
</tr>
<tr>
<td>ctime</td>
<td>3</td>
<td>ctime_r</td>
</tr>
<tr>
<td>gethostbyaddr</td>
<td>3</td>
<td>gethostbyaddr_r</td>
</tr>
<tr>
<td>gethostbyname</td>
<td>3</td>
<td>gethostbyname_r</td>
</tr>
<tr>
<td>inet_ntoa</td>
<td>3</td>
<td>(none)</td>
</tr>
<tr>
<td>localtime</td>
<td>3</td>
<td>localtime_r</td>
</tr>
<tr>
<td>rand</td>
<td>2</td>
<td>rand_r</td>
</tr>
</tbody>
</table>

Today

- Thread safety
- 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