Thread-Level Parallelism

CSE 2431: Introduction to Operating Systems
Reading: §12.6, [CSAPP]
Contents

• Parallel Computing Hardware
  – Multicore
    • Multiple separate processors on single chip
  – Hyperthreading
    • Replicated instruction execution hardware in each processor
  – Maintaining cache consistency

• Thread Level Parallelism
  – Splitting program into independent tasks
    • Example: Parallel summation
    • Some performance artifacts
  – Divide-and-conquer parallelism
    • Example: Parallel quicksort
Multicore Processor

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

```plaintext
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

**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

```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
Snoopy Caches (1)

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

Main Memory

- a: 1
- b: 100

Thread1 Cache

- E a: 2

Thread2 Cache

- E b: 200

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

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

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

int a = 2;
print(b);

Wb: b = 200;
Ra: print(a);
```
Snoopy Caches (2)

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

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

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

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

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

- print 2
- print 200
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 16×
    - Never achieved in benchmarks
Summation Example

- Sum numbers 0, …, \(N-1\)
  - Should add up to \((N-1)\times N/2\)
- 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 \texttt{pthread} semaphore
  - 1C: Synchronize with \texttt{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];
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 = $2.86 \times$
- 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 $\implies \sim 10$ minutes
- Mutex $3\times$ 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

```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;
```
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: $5 \times$
  – Spaced-apart speedup: $13.3 \times$ (Only observed speedup $> 8$)

• Why does spacing the accumulators apart matter?
False Sharing

- Coherency maintained on cache blocks
- To update $p\text{sum}[i]$, 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 $2.8 \times$ 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×
- 2× better than fastest memory accumulation
A More Interesting Example

• 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 (1)
Sequential Quicksort Visualized (2)
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 $N \leq N_{thresh}$, 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 $L' : p : R'$

• Degree of parallelism
  – Top-level partition: none
  – Second-level partition: $2 \times$
  – …
Parallel Quicksort Visualized

\[ X \]

\[ p \]

\[ L \quad p \quad R \]

\[ p2 \]

\[ L2 \quad p2 \quad R2 \quad p \quad L3 \quad p3 \quad R3 \]

\[ \ldots \]

\[ L' \quad p \quad R' \]
Parallel Quicksort Data Structures

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

```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;
sem_t tmutex;
```
Parallel Quicksort Initialization

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;
}

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

```c
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

```c
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

```c
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 (1)

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

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

• Task set data structure
  – Array of structs
    sort_task_t *tasks;
    • new_task returns pointer or integer index
  – Array of pointers to structs
    sort_task_t **tasks;
    • new_task dynamically allocates struct and returns pointer

• Reaping threads
  – Can we be sure the program won’t terminate prematurely?
Amdahl’s Law

• Overall problem
  – $T$: Total 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
  – $T_9 = 0.9 \times \frac{10}{9} + 0.1 \times 10 = 1.0 + 1.0 = 2.0$
  – 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: $\leq 2 \times$ speedup
  – $k^{th}$ level: $\leq 2^{k-1} \times$ speedup

• Implications
  – Good performance for small-scale parallelism
  – Would need to parallelize partitioning step to get large-scale parallelism
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