Thread-Level Parallelism

15-213: Introduction to Computer Systems
26th Lecture, November 29, 2016

Instructor:
Phil Gibbons
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 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)\times n)/2\]

- **Partition values 1, ..., n-1 into** \(t\) ranges
  - \([n/t]\) 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: \texttt{psum-mutex}

Simplest approach: Threads sum into a global variable protected by a semaphore \texttt{mutex}.

```c
void *\texttt{sum_mutex}(void *\texttt{vargp}); /* Thread routine */

/* Global shared variables */
long \texttt{gsum} = 0; /* Global sum */
long \texttt{nelems_per_thread}; /* Number of elements to sum */
\texttt{sem_t \texttt{mutex}}; /* Mutex to protect global sum */

int \texttt{main}(int \texttt{argc}, char **\texttt{argv})
{
    long \texttt{i}, \texttt{nelems}, \texttt{log_nelems}, \texttt{nthreads}, \texttt{myid}[MAXTHREADS];
    \texttt{pthread_t \texttt{tid}[MAXTHREADS]};

    /* Get input arguments */
    \texttt{nthreads} = \texttt{atoi(ARGV[1])};
    \texttt{log_nelems} = \texttt{atoi(ARGV[2])};
    \texttt{nelems} = (1L << \texttt{log_nelems});
    \texttt{nelems_per_thread} = \texttt{nelems} / \texttt{nthreads};
    \texttt{sem_init(\&mutex, 0, 1)};
}
```

\texttt{psum-mutex.c}
Simplest approach: Threads sum into a global variable protected by a semaphore mutex.

```c
/* 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;
```

Thread ID
Thread routine
Thread arguments (void *p)
**psum-mutex** Thread Routine

- Simplest approach: Threads sum into a global variable protected by a semaphore mutex.

```c
/* 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, \( n=2^{31} \)

<table>
<thead>
<tr>
<th>Threads (Cores)</th>
<th>1 (1)</th>
<th>2 (2)</th>
<th>4 (4)</th>
<th>8 (8)</th>
<th>16 (8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>psum-mutex (secs)</td>
<td>51</td>
<td>456</td>
<td>790</td>
<td>536</td>
<td>681</td>
</tr>
</tbody>
</table>

- 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 $\text{psum}[i]$
- Main waits for threads to finish, then sums elements of $\text{psum}$
- Eliminates need for mutex synchronization

```c
/* 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 Performance

- Orders of magnitude faster than psum-mutex

![Graph showing performance comparison between psum-array and psum-mutex]
Next Attempt: \texttt{psum-local}

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

```c
/* Thread routine for psum-local.c */
void *sum_local(void *vargp)
{
    long myid = *((long *)vargp); /* Extract thread ID */
    long start = myid \times \text{nelems\_per\_thread}; /* Start element index */
    long end = start + \text{nelems\_per\_thread}; /* End element index */
    long \texttt{i}, sum = 0;

    for (i = start; i < end; i++) {
        sum += i;
    }
    \texttt{psum[myid] = sum;}
    return NULL;
}
```

\texttt{psum-local.c}
**psum-local Performance**

- Significantly faster than `psum-array`

![Graph showing performance comparison between `psum-array` and `psum-local`](image)

**Parallel Summation**

<table>
<thead>
<tr>
<th>Threads (cores)</th>
<th>Elapsed seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1(1)</td>
<td>5.36</td>
</tr>
<tr>
<td>2(2)</td>
<td>4.24</td>
</tr>
<tr>
<td>4(4)</td>
<td>2.54</td>
</tr>
<tr>
<td>8(8)</td>
<td>1.64</td>
</tr>
<tr>
<td>16(8)</td>
<td>0.94</td>
</tr>
</tbody>
</table>

`psum-array` vs. `psum-local`
Characterizing Parallel Program Performance

- \( p \) processor cores, \( T_k \) is the running time using \( k \) cores

- **Def. Speedup:** \( S_p = \frac{T_1}{T_p} \)
  - \( S_p \) 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:** \( E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \)
  - Reported as a percentage in the range \((0, 100]\)
  - Measures the overhead due to parallelization

- Is super-linear speed-up (*\( S_p > p \), *\( E_p > 100\% \)) possible?
  - Yes: Due to hyperthreading and cache effects
# Performance of `psum-local`

<table>
<thead>
<tr>
<th>Threads (t)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores (p)</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Running time ($T_p$)</td>
<td>1.98</td>
<td>1.14</td>
<td>0.60</td>
<td>0.32</td>
<td>0.33</td>
</tr>
<tr>
<td>Speedup ($S_p$)</td>
<td>1</td>
<td>1.74</td>
<td>3.30</td>
<td>6.19</td>
<td>6.00</td>
</tr>
<tr>
<td>Efficiency ($E_p$)</td>
<td>100%</td>
<td>87%</td>
<td>82%</td>
<td>77%</td>
<td>75%</td>
</tr>
</tbody>
</table>

- 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
  ▪ T₉ = 0.9 * 10/9 + 0.1 * 10 = 1.0 + 1.0 = 2.0
  ▪ Least possible running time:
    ▪ Tₙ = 0.1 * 10.0 = 1.0

■ Limit on strong scaling: fixed problem size, increasing cores
■ Not on weak scaling: problem size scales with increasing cores
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 (when value=p, break tie by array index)
  - 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

X

L' | p | R

L' | p3 | R' | R3

L' | p | R'
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 of size N
  - 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)

```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;
}
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
  - \( k^{\text{th}} \) 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
Parallelizing Partitioning Step

Parallel partitioning based on global p

Reassemble into partitions
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
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
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

```c
int a = 1;
int b = 100;
Thread1:  
  Wa:  a = 2;
  Rb:  print(b);
Thread2:  
  Wb:  b = 200;
  Ra:  print(a);
```
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

```
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

- 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);
```

Thread1 Cache
- E: a:2

Thread2 Cache
- E: b:200

Main Memory
- T1: a:1
- T2: b:100
Snoopy Caches

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

Thread1 Cache
- S a:2
- S b:200

Thread2 Cache
- S a:2
- S b:200

Main Memory
- S a:2
- S b:200

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

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

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

int a = 1;
int b = 100;

print 2
print 200
Non-Sequentially Consistent Scenario

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

**Fix:** Add `SFENCE` instructions between `Wa` & `Rb` and `Wb` & `Ra`
Recap

- **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