Review and Wrap-Up

Mark Greenstreet

CpSc 418 – Nov. 28, 2013

Lecture Outline

Review and Everything Else

- Review
  - Scan
  - Producer-Consumer
  - Bitonic Sorting
  - ...
- Everything Else
  - Energy and Computing
  - Tilera/Raw
  - Silicon Photonics
  - nano-tubes, graphene, MEMs
  - Computing for the next 10+ years
  - My research
- Correctness of shared memory programs
  - Bad stuff: Races, deadlock, livelock
  - Good stuff: Invariants

Scan

- How to design `Leaf1`, `Leaf2`, and `Combine`
- …

Other HW4 stuff

- Q2.a is easy.
- What is “show” as in “Show that `F` commutes with `my_merge`”?
  - You need to show that the claim holds for all cases.
  - Your argument needs to be convincing.
  - You need to convince the reader (me, the TA’s etc.) that the claim holds.
    - This may not mean showing every last detail of the derivation.
    - But you do need to show enough that the pieces we fill-in are things like being able to conclude that if $x \leq y - 1$ then $x < y$, simple algebra, etc.
  - You need to convince the reader that you really understood the full argument.
    - No gaps in the proof that I could probably fill in but leave doubts about whether you got stuck.
  - Statement/reason proofs are great.
    - If you tell me why you can make an inference, then I’ll believe that you understood it.
    - “It’s obvious” is not a good “reason”.
    - “algebra” or “implied by steps 2, 3, and 5” can be very good reaons.

Producer-Consumer

- Problem statement:
  - The producer generates a sequence of data values: v_1, v_2, ...
  - The consumer reads this sequence from the producer.
  - If the consumer is ready to read a value and none is available from the producer, then the consumer stalls until the a data value is available.
  - Likewise, we can implement this interface with a fixed-capacity buffer.
    - ★ In this case, if the producer generates a value and there is no empty space available in the buffer, the producer stalls until the value can be written to the buffer.
- We'll look at an implementation using a shared, fixed-sized array as a buffer.

Producer-Consumer: try 1

```
Value buffer[n]; // shared buffer
int wptr, rptr; // indices for current write and read positions
int next(int i) { // cyclic successor of i
    return((i+1) % n);
}
void put(Value v) { // called by producer
    if(next(wptr) != rptr) {
        buffer[wptr] = v;
        wptr = next(wptr);
    } else ???
}
Value take() { // called by consumer
    if(rptr != wptr) {
        Value v = buffer[rptr];
        rptr = next(rptr);
        return(v);
    } else ???
}
```

Producer-Consumer: try 2

```
void put(Value v) { // called by producer
   while(next(wptr) == rptr); // wait for empty space
   buffer[wptr] = v;
   wptr = next(wptr);
}
Value take() { // called by consumer
   while(rptr == wptr); // wait for data to arrive
   Value v = buffer[rptr];
   rptr = next(rptr);
   return(v);
}
```

What's wrong with this solution?

Condition Variables (try cond-1)

- `wait(cond);` this thread waits until a signal is sent to `cond`.
- `signal(cond);` this thread sends a signal to `cond`.

Producer-Consumer: try 3

```
Cond w_cond, r_cond; // condition variables
void put(Value v) { // called by producer
   int oldwptr = wptr;
   if(next(wptr) == rptr)
      wait(w_cond);
   buffer[wptr] = v;
   wptr = next(wptr);
   if(oldwptr == rptr)
      signal(r_cond);
}
Value take() { // called by consumer
   int oldrptr = rptr;
   if(rptr == wptr)
      wait(r_cond);
   Value v = buffer[rptr];
   rptr = next(rptr);
   if(next(wptr) == oldrptr)
      signal(w_cond);
   return(v);
}
```

Mutex Variables

- `lock(mutex);` this thread acquires a lock on `mutex`.
  - Only one thread can have the lock at a time.
  - If a thread θ_i attempts to lock a mutex that thread θ_j has already locked, then thread θ_i will block.
- `unlock(mutex);` this thread releases its lock on mutex.
  - If one or more threads are blocked trying to lock the mutex, then one of them will acquire the lock.
  - If multiple threads are waiting for the mutex, an arbitrary one gets it.
  - There is no promise or intent of first-come-first-served awarding of the mutex to waiting threads.

Producer-Consumer: try 4

```
Mutex m; // a mutex variable
void put(Value v) { // called by producer
   int oldwptr = wptr;
   lock(m);
   if(next(wptr) == rptr)
      wait(w_cond);
   buffer[wptr] = v;
   wptr = next(wptr);
   if(oldwptr == rptr)
      signal(r_cond);
   unlock(m);
}
Value take() { // called by consumer
   int oldrptr = rptr;
   lock(m);
   if(rptr == wptr)
      wait(r_cond);
   Value v = buffer[rptr];
   rptr = next(rptr);
   if(next(wptr) == oldrptr)
      signal(w_cond);
   unlock(m);
   return(v);
}
```

Condition variables and mutexes

- We need a mutex with each condition variable
  - Otherwise, we can't safely check the wait condition.
- If the thread needs to wait, then the mutex needs to be unlocked after the thread is waiting for the signal.
  - But, if the thread is waiting for a signal, then it's blocked,
  - ...and it can't do anything.
  - In particular, it can't unlock the mutex.
- Solution: the `wait` function handles the mutex lock:
  - When the thread is suspended, `wait` unlocks the mutex.
  - When the thread is resumed, `wait` relocks the mutex.

Producer-Consumer: final solution

```
void put(Value v) { // called by producer
   int oldwptr = wptr;
   lock(m);
   if(next(wptr) == rptr)
      wait(w_cond, m);
   buffer[wptr] = v;
   wptr = next(wptr);
   if(oldwptr == rptr)
      signal(r_cond);
   unlock(m);
}
Value take() { // called by consumer
   int oldrptr = rptr;
   lock(m);
   if(rptr == wptr)
      wait(r_cond, m);
   Value v = buffer[rptr];
   rptr = next(rptr);
   if(next(wptr) == oldrptr)
      signal(w_cond);
   unlock(m);
   return(v);
}
```

We could unlock the mutex while updating `buffer`, `rptr`, and `wptr`. Should we?

Mutexes

The mutex type: `pthread_mutex_t`

- declare and initialize a mutex:

```
pthread_mutex_t my_mutex;
pthread_mutex_init(&my_mutex, NULL);
```

- using a mutex:
  - `pthread_mutex_lock(&my_mutex);`
  - `pthread_mutex_unlock(&my_mutex);`
  - `pthread_mutex_trylock(&my_mutex);`
  - `pthread_mutex_destroy(&my_mutex);`
- usage:
  - Typically, a mutex is associated with a shared data structure.
  - A thread acquires the mutex before accessing the data structure.

Condition Variables

The condition variable type: `pthread_cond_t`

- declare and initialize a condition variable:

```
pthread_cond_t my_cond;
pthread_cond_init(&my_cond, NULL);
```

- using a condition:
  - `pthread_cond_wait(&my_cond);`
  - `pthread_cond_signal(&my_cond);`
  - `pthread_cond_broadcast(&my_cond);`
  - `pthread_cond_destroy(&my_cond);`
- condition variables and locks:

Spurious wake-ups

- Threads can wake-up “spontaneously”
  - This arises from performance optimizations in the OS.
  - There are races that are better to expose to the application than it would be to create a sequential bottleneck in the kernel.
- **WRONG:**

```
if(condition)
    wait(cond, m);
```

- **RIGHT:**

```
while(condition)
    wait(cond, m);
```

Producer-Consumer: final version

```
void put(Value v) { // called by producer
   int oldwptr = wptr;
   lock(m);
   while(next(wptr) == rptr)
      wait(w_cond, m);
   buffer[wptr] = v;
   wptr = next(wptr);
   if(oldwptr == rptr)
      signal(r_cond);
   unlock(m);
}
Value take() { // called by consumer
   int oldrptr = rptr;
   lock(m);
   while(rptr == wptr)
      wait(r_cond, m);
   Value v = buffer[rptr];
   rptr = next(rptr);
   if(next(wptr) == oldrptr)
      signal(w_cond);
   unlock(m);
   return(v);
}
```

Bitonic Merge

Convert a bitonic sequence to a monotonic one.

- Let $x_0, x_1, \ldots, x_{N-1}$ be a bitonic sequence, with N even.
- Let

$$\begin{aligned} y_i &= \min(x_i, x_{i+\frac{N}{2}}) &&, \text{if } 0 \leq i < \frac{N}{2} \\ &= \max(x_i, x_{i-\frac{N}{2}}) &&, \text{if } \frac{N}{2} \leq i < N \end{aligned}$$

- Then
  - Either $y_0, \ldots y_{\frac{N}{2}-1}$ is all zeros or $y_{\frac{N}{2}}, \ldots y_{N-1}$ is all ones, and is bitonic.

    Proof:
    - ★ If $x_0, \ldots x_{\frac{N}{2}-1}$ or $x_{\frac{N}{2}}, \ldots x_{N-1}$ is clean, then either $y_0, \ldots y_{\frac{N}{2}-1}$ or $y_{\frac{N}{2}}, \ldots y_{N-1}$ is clean, and the other is just a copy of the other half of x and therefore bitonic.
    - ★ If neither $x_0, \ldots x_{\frac{N}{2}-1}$ nor $x_{\frac{N}{2}}, \ldots x_{N-1}$ are clean, $x_0, \ldots x_{\frac{N}{2}-1}$ is positive monotonic and $x_{\frac{N}{2}}, \ldots x_{N-1}$ is negative monotonic, and the result follows by an argument like the one we used for Shear sort.
    - ★ Note that in the second case, the bitonic part can be either $\nearrow\searrow$ or $\searrow\nearrow$ bitonic.

Bitonic Merge: The Big-Picture

- Big picture: the largest element of $y_0, \ldots y_{\frac{N}{2}-1}$ is less than or equal to the smallest element of $y_{\frac{N}{2}}, \ldots y_{N-1}$.
- Now, recurse to convert $y_0, \ldots y_{\frac{N}{2}-1}$ and $y_{\frac{N}{2}}, \ldots y_{N-1}$ into monotonic sequences.

Bitonic Sort

Assume N is a power of 2.

- Sorting an array with one element is easy.
- Sorting an array with two elements is a single-compare and-swap.
- To sort an array with four elements:
  - Sort elements x_0 and x_1 in ascending order.
  - Sort elements x_2 and x_3 in descending order.
  - Now, the list $[x_0, x_1, x_2, x_3$ is $\nearrow\searrow$ bitonic.
  - Use a 4-way merge.
- To sort an array with N elements ($N > 2$):
  - Sort elements $x_0, \ldots, x_{\frac{N}{2}-1}$ in ascending order.
  - Sort elements $x_{\frac{N}{2}}, \ldots, x_{N-1}$ in descending order.
  - Now, the list $[x_0, x_1, \ldots, x_{N-1}$ is $\nearrow\searrow$ bitonic.
  - Use a N-way merge.

That's Odd (1 of 3)

What if N is odd?

- Let $x_0, x_1, \ldots, x_{N-1}$ be a bitonic sequence, with N odd.
- Let

$$\begin{array}{rcll} y_i & = & \min(x_i, x_{i+\frac{N+1}{2}}) & \text{, if } 0 \leq i < \frac{N-1}{2} \\ & = & x_i & \text{, if } i = \frac{N-1}{2} \\ & = & \max(x_i, x_{i-\frac{N+1}{2}}) & \text{, if } \frac{N+1}{2} \leq i < N \end{array}$$

- Then
  - Either $y_0, \ldots y_{\frac{N-1}{2}-1}$ is all zeros or $y_{\frac{N-1}{2}}, \ldots y_{N-1}$ is all ones.
    Proof:
    - ⋆ Pretty much like the case when N is even, with some extra care for $y_{\frac{N-1}{2}}$.
    - ⋆ Assume x is $\nearrow\searrow$ bitonic. The argument for the other case is equivalent.
    - ⋆ If $x_{\frac{N-1}{2}}$ is 0, see slide 22.
    - ⋆ Else $x_{\frac{N-1}{2}}$ is 1, see slide 23.

That's Odd (2 of 3)

If $x_{\frac{N-1}{2}}$ is 0,

- Either $x_0, \ldots x_{\frac{N-1}{2}-1}$ is constant zero or $x_{N+1}2, \ldots x_{N-1}$ is constant zero.
- $y_0, \ldots x_{\frac{N-1}{2}-1}$ is constant zero.
- $y_{\frac{N+1}{2}}, \ldots y_{N-1}$ is $\nearrow\searrow$ bitonic.
- $0, y_{\frac{N+1}{2}}, \ldots y_{N-1}$ is $\nearrow\searrow$ bitonic.
- $y_{\frac{N-1}{2}}, y_{\frac{N+1}{2}}, \ldots y_{N-1}$ is $\nearrow\searrow$ bitonic.

That's Odd (3 of 3)

- If $x_{\frac{N-1}{2}}$ is 1, then
  - $\star$ If $x_{\frac{N+1}{2}} = 0$, then
    - $x_{\frac{N+1}{2}}, \ldots, x_{N-1}$ is constant 0.
    - $x_0, \ldots, x_{\frac{N-1}{2}}$ is positive monotonic.
    - $y_{\frac{N+1}{2}}, \ldots, y_{N-1} = x_0, \ldots, x_{\frac{N-1}{2}}$.
    - $0, y_{\frac{N+1}{2}}, \ldots, y_{N-1}$ is bitonic.
    - $y_{\frac{N-1}{2}}, y_{\frac{N+1}{2}}, \ldots, y_{N-1}$ is bitonic.
- Short version: if N is odd:
  - Perform a round of compare-and-swap operations with stride $\frac{N+1}{2}$.
  - Perform bitonic merge on $y_0, \ldots, y_{\frac{N-1}{2}-1}$, and a separate merge on $y_{\frac{N-1}{2}}, \ldots y_{N-1}$.

    The first sequence has $\lfloor \frac{N}{2} \rfloor$ elements and the second has $\lceil \frac{N}{2} \rceil$ elements.

Bitonic Time

- A M-way merge has $\lceil \log_2(M) \rceil$ stages of compare-and-swap elements.
  - Each stage has $\sim M/2$ compare and swap operations.
  - The merge can be done in $O(\log(M))$ parallel time with $O(M \log(M))$ compare-and-swap operations.
- Bitonic sort of N elements requires merges of size N, $N/2$, $\ldots$, 2.
  - Bitonic sort can be done in $O(\log^2(N))$ parallel time.
  - A total of $O(N \log^2(N))$ compare-and-swap operations are performed.
- Beware of communication overheads
  - A time cost of $\log^2(N)\lambda$ for communication if we don't worry about bandwidth.
  - No matter how you arrange the processors, bitonic sort requires several exchanges of the full data set across any network bisection.
  - If the network bisection bandwidth is $o(N)$, then this becomes the bottleneck.

Energy and Computing

- Power consumption is the key performance limitter for sequential computing.
  - This is why the world of computing has gone parallel.
  - Parallelism from fine-grained, data-parallelism of GPUs to big cloud/cluster computers.
  - Communication is the key consideration of parallel performance
  - Then energy to compute something is strongly connected to:
    - how many bits have to move,
    - how far they have to move,
    - how fast they need to get there.
  - Counting operations is at best a very indirect measure of the resources (time, energy, etc.) needed for the computation.
- Communication costs:
  - Fixed cost model: λ
    - Reminds us the communication is expensive.
    - Ignores constraints of network topology.
  - Network cross-section bandwidth critical for many computations.
    - Sorting is an example.

Other ways to compute

- RAW/Tilera: http://tilera.com/,
  http://dx.doi.org/10.1109/MM.2002.997877
- Silicon photonics:
  http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumbe
- nano-tubes, graphene
- other?

My research

- It's really cool.
- Let me tell you about it...

Finally, the final

- Review sessions
  - Monday, Dec. 2, 10:30am-12noon, ICCS X836
  - Tuesday, Dec. 3, 10:30am-12noon, ICCS X836