Lecture 6
Announcements

- A2 will be posted by Monday at 9AM
Today’s lecture

• Cache Coherence and Consistency
• False sharing
• Parallel sorting
Recapping from last time: Bang’s memory hierarchy

- Each core of bang has…
  - Private L1 caches (instructions and data)
  - Shared L2 cache
- `/sys/devices/system/cpu/cpu*/cache/index*/`
- Login to bang `qlogin` node and view the files
Cache Coherence

• What happens if two cores have a cached copy of a shared memory location and one of them writes that location?
• If one writes to the location, *all* others must *eventually* see the write
• Cache coherence is the consistency of shared data across multiple caches
What happens to P1’s copy of x?

A. It could be invalidated
B. It could be updated to x==2
C. We can’t say
D. A and B [this should be worded as ‘or’]
E. A and C
Cache Coherence in action

- P0 & P1 load X from main memory into cache
- P0 stores 2 into X
- The memory system doesn’t have a coherent value for X
Cache Coherence Protocols

- Ensure that all processors *eventually* see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write
SMP architectures

• Employ a snooping protocol to ensure coherence

• Cache controllers listen to bus activity updating or invalidating cache as needed

Patterson & Hennessey
Can we keep adding more processors to a snooping bus without performance consequences?

A. Yes
B. No
C. Not sure
Memory consistency and correctness

• The cache coherence policy tells us that a write will *eventually* become visible to other processors

• The memory consistency model tells us *when* this will happen, that is, when a written value will be seen by a reader

• **But:** Even if memory is consistent, changes don’t propagate instantaneously

• These give rise to correctness issues involving program behavior and the use of appropriate synchronization
How can we characterize a memory consistency model with respect to ensuring program correctness?

A. Necessary
B. Sufficient
C. Both A & B
Memory consistency

• A memory system is consistent if the following 3 conditions hold
  ‣ Program order (you read what you wrote)
  ‣ Definition of a coherent view of memory (“eventually”)
  ‣ Serialization of writes
    (a single frame of reference)
• We’ll look at each condition in turn
Program order

• If a processor writes and then reads the same location X, and there are no other intervening writes by other processors to X, then the read will always return the value previously written.

\[ X = 2 \]

\[ \text{Memory} \]

\[ P \]

\[ X := 2 \]

\[ X := 2 \]
Definition of a coherent view of memory

- If a processor $P$ reads from location $X$ that was previously written by a processor $Q$, then the read will return the value previously written, if a sufficient amount of time has elapsed between the read and the write.
Serialization of writes

• If two processors write to the same location $X$, then other processors reading $X$ will observe the same the sequence of values in the order written.

• If 10 and then 20 is written into $X$, then no processor can read 20 and then 10.
What does memory consistency buy us?

• It enables us to write correct programs that share data
• Think about using a lock to protect access to a shared counter, say in processor self-scheduling

```java
boolean getChunk(int& startRow) {
    my_mutex.lock();
    k = _counter;
    _counter += _chunk;
    my_mutex.unlock();
    if (k > (_n – _chunk))
        return false;
    startRow = k;
    return true;
}
```

A memory system is consistent if the following 3 conditions hold

1. Program order: you read what you wrote
2. Definition of a coherent view of memory (“eventually”)
3. Serialization of writes: a single frame of reference
Consistency in practice

- Assume that there is ..
  - A bus-based snooping cache
  - A buffer between CPU and Cache that delays the writes
- Initially $A = B = 0$

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A=1$</td>
<td>$B=1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><strong>if</strong> ($B==0$)</td>
<td><strong>if</strong> ($A==0$)</td>
</tr>
<tr>
<td>Critical section</td>
<td>Critical section</td>
</tr>
</tbody>
</table>
If memory is inconsistent, it possible that both if statements evaluate to true and hence both cores enter a critical section?

A. Yes
B. No
C. Not sure

<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A=1</td>
<td>B=1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>if (B==0)</td>
<td>if (A==0)</td>
</tr>
<tr>
<td>Critical section</td>
<td>Critical section</td>
</tr>
</tbody>
</table>
Today’s lecture

• Cache Coherence and Consistency
• False sharing
• Sorting
False sharing

• Even if two cores don’t share the same memory location, there can be overheads if they write to the same cache line

• We call this “false sharing” because we don’t share any data
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated
False sharing

• P1 reads the location written by P0
• P1 then writes a different location in the same block of memory
False sharing

• P1’s write updates main memory
• Snooping protocol invalidates the corresponding block in P0’s cache
False sharing

• Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Is false sharing a correctness or performance issue?

A. Correctness
B. Performance
Avoiding false sharing

• Cleanly separate locations updated by different processors
  ‣ Manually assign scalars to a pre-allocated region of memory using pointers
  ‣ Spread out the values to coincide with a cache line boundaries
Example of false sharing

- Reduce number of accesses to shared state
- Use a local variable and write only at the end of many updates
- To allocate an aligned block of memory, use `memalign`
  https://www.chemie.fu-berlin.de/chemnet/use/info/libc/libc_3.html#SEC28

```c
static int counts[];
for (int k = 0; k < reps; k++)
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            counts[TID]++;

int _count = 0;
for (int k = 0; k < reps; k++) {
    for (int r = first; r <= last; ++r)
        if ((values[r] % 2) == 1)
            _count++;
    counts[TID] = _count;
}
```

4.7s, 6.3s, 7.9s, 10.4 [NT=1,2,4,8] 3.4s, 1.7s, 0.83, 0.43 [NT=1,2,4,8]
Today’s lecture

• Cache Coherence and Consistency
• False sharing
• Parallel sorting
Parallel Sorting

• Sorting is fundamental algorithm in data processing
  ‣ Given an unordered set of keys $x_0$, $x_1$, ..., $x_{N-1}$
  ‣ Return the keys in sorted order

• The keys may be character strings, floating point numbers, integers, or any object for which the relations $>$, $<$, and $==$ hold

• We’ll assume integers

• In practice, we sort on external media, i.e. disk, but we’ll consider in-memory sorting See: http://sortbenchmark.org

• There are many parallel sorts. We’ll implement Merge Sort in A2
Serial Merge Sort algorithm

- A divide and conquer algorithm
- We stop the recursion when we reach a certain size $g$
- Sort each piece with a fast local sort
- We merge data in odd-even pairs
- Each partner get the smallest (largest) $\frac{N}{P}$ values, discards the rest
- Running time of the merge in $O(m+n)$, assuming 2 vectors of size $m$ & $n$
Why might the lists to be merged have different sizes?

A. Because the median value might not be in the middle
B. Because the mean value might not be in the middle
C. Both A&B
D. Not sure
Parallel Merge Sort algorithm

- At each level of recursion we use \( x^2 \) the number of cores as at the previous level.
- When we are running on all the cores, we stop spawning threads & switch to the serial merge sort algorithm.
- As with the serial algorithm, we stop the recursion when we reach a certain size \( g \).
- Threads merge data in odd-even pairs.
- The simplest algorithm uses a sequential merge, but we’ll implement a parallel merge.
- But start with the serial merge!
In general, $N/g << N/$# threads and you’ll reach the ‘g’ limit before the thread limit.
Serial Merge

-1 3 7 9 11

Thread 0

2 4 8 12 14

Thread 1

• Merge Step
• Left most thread does the merging
  -1 3 7 9 11 2 4 8 12 14
• Sorts the merged list
  -1 2 3 4 7 8 9 11 2 14
• Parallelism diminishes as we move up the recursion tree
• There is only $O(\log n)$ parallelism, but if we stop the recursion before reaching the bottom of the tree, it’s much smaller
What is the running time of parallel merge sort (with serial merge)?

A. $O(N)$
B. $O(N \log N)$
C. $O(N^2)$
Assignment #1

• Implement parallel merge sort
• When this is running correctly, and you have conducted a strong scaling study…
• Implement parallel merge and determine how much it helps