COMP 633 - Parallel Computing

Lecture 12
October 4, 2018

CC-NUMA (3)
Synchronization Operations
Synchronizing Operations

• Examples
  – *locks* to gain exclusive access for manipulation of shared variables
  – barrier synchronization

• How are these efficiently implemented in a large cache-coherent shared memory multiprocessor?
Atomic operations in shared-memory multiprocessors

- Possible atomic machine operations
  In the following, < ... > refers to atomic execution of action within the brackets, \( m \) is a memory location, and \( r_1, r_2 \) are processor registers
  - read and write
    \[
    <r_1 := m>  \\
    <m := r_1>
    \]
  - exchange\((m, r_1)\)
    \[
    <r_1, m := m, r_1>
    \]
  - test and set\((m, r_1, r_2)\)
    \[
    <\text{if } (m == r_1) \text{ then } m := r_2>
    \]
  - fetch and add\((m, r_1, r_2)\)
    \[
    <r_2 := m + r_1; m := r_2>
    \]
  - load-linked\((r_1, m)\) and store-conditional\((m, r_2)\)
    \[
    <r_1 := m>; ....; <m := r_2 \text{ or } fail>
    \]
    - if \( m \) is updated by another processor between the read and write, the write to \( m \) will not be performed and the condition code cc will be set to fail
How implemented?

- Atomic read and write
  - simple to implement, difficult to use (recall memory consistency discussion)

- Exchange, test-and-set, fetch-and-add
  - require read-modify-write
    - Involves locking at some hardware level and/or a special coherence protocol

- Load-linked (LL) / Store conditional (SC)
  - LL fetches value into cache line (state = shared)
  - cache-line state is monitored
  - SC fails if cache line has invalid state at time of store
  - Example
    
    ;; implementation of r2 := fetch-and-add(m,r1) using LL/SC
    try:
    ll r3, m
    add r3, r1, r3 ; r3 := r3 + r1
    sc r3, m
    bcz try ; try again if sc fails
Lock/unlock using atomic operations

• Exchange lock
  – key holds access to the lock
    • key == 0 means lock available
  – to get access, a processor must exchange value 1 with key value 0
    \{r1 == 1\}
    lock: exch r1, key ; spin until zero obtained
    cmpi r1, 0 ;
    bne lock ;
    \{lock obtained\}
  – to release, exchange with key
    \{r1 == 0\}
    unlock: exch r1, key
    \{lock released\}

– what is the effect of spinning on an exchange lock in a CC-NUMA machine?
  • with single processor trying to obtain lock?
    – key is cache-resident in EXCLUSIVE state until released by other processor
  • with multiple processors trying to obtain lock?
    – each exchange brings key into cache and invalidates other copies requiring O(p) cache lines to be refreshed.
Improving cost of contended locks

• “Local” spinning using read-only copy of key
  – avoid coherence traffic while spinning
    lock: {r1 == 1}
    try:  lw r2, key
    cmpi r2,0
    bne try
    {lock observed available}
    exch r1, key
    cmpi r1, 0
    bne try
    {lock obtained}

• What happens with p processors spinning?
  – No coherence traffic when all processors have key in cache in “shared” state

• What happens when key is released with p processors spinning?
  – key is invalidated and up to p processors observe the lock available
  – up to p processors attempt an exchange
    • one succeeds
    • up to p-1 other processors perform an unsuccessful exch
      – each exch invalidates up to p-2 local copies of key
  – O(p^2) cache lines moved per lock release
Improving cost of lock release

- LL/SC makes an improvement
  - now 2p movements of cache line on release
    
    ```
    lock:  \{ r1 == 1 \}
    try:   ll  r2, key
           cmpi r2,0
           bne  try
           \{ lock observed available \}
    sc   r1, key
    bz   try
    \{ lock obtained \}
    ```

- basic problem
  - attempt to replicate contended value across caches
  - high cost when p processors contending

- Alternate approaches
  - exponential backoff
    - increase time to re-try with each failure
  - array lock: each process spins on different cache line
Barrier Synchronization

• Delay p processors until all have arrived at barrier
  – simple strategy
    • shared variables: count, release (initially with value 0)
    • in each processor
      lock; count = count + 1; unlock
      if (count == p) then release := 1
      local spinning while release == 0

  – How many cache line moves are required for p processors to pass the barrier?
    • p lock/unlock operations
    • each lock and unlock may have O(p) cache line moves
      – O(p^2) cache line moves in the presence of contention
      – Can we do better?
Barrier synchronization

• Barrier synchronization may have high contention on entry and on release
  – reduce contention on entry using backoff
    • exponential backoff in re-attempting lock acquisition
    • random delay in re-attempting lock acquisition
    • both approaches fully serialize entry to the barrier
      – O(2^p) cache block movements

  – reduce contention on entry and exit using a combining tree
    • O(1) contention in lock acquisition
    • O(p) cache line movements
    • O(lg p) lock acquisitions worst case delay
    • more parallelism in scalable shared memory multiprocessors
    • Sometimes implemented in hardware
Dissemination barrier

- Barrier using only atomic reads and writes
  - assume $p = 2^k$ processors
  - $\text{arrive}[0 : p-1]$ has initial value zero for all elements.
  - program executed by processor $i$

```c
int s = 1;
for (int j = 0; j < k; j++) {
    arrive[i] += 1;
    while (arrive[i] > arrive[(i+s) mod p]) { /* spin */
        s = 2 * s;
    }
} /* barrier synchronization achieved */
```

- \( \text{arrive}[i : i+s-1 \mod p] > 0 \)
- \( \text{arrive}[i : i+p-1 \mod p] > 0 \)