CC-NUMA (3)
Synchronization Operations
Synchronizing Operations

• Examples
  – *locks* to gain exclusive access for manipulation of shared variables
  – *barrier synchronization* to ensure all processors have reached a program point

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

- Possible atomic machine operations
  In the following, < ... > refers to atomic execution of action within the brackets, 
m is a memory location, and r1, r2 are processor registers

  - read and write
    <r1 := m>
    <m := r1>

  - exchange(m,r1)
    <r1, m := m, r1>

  - test and set(m,r1,r2)
    <if (m == r1) then m := r2>

  - fetch and add(m,r1,r2)
    <r2 := m + r1; m := r2>

  - load-linked(r1,m) and store-conditional(m,r2)
    <r1 := m>; .... ; <m := r2 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 some hardware-level special coherence protocol

• Load-linked (LL) / Store conditional (SC)
  – LLfetches 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 \( r2 := \text{fetch-and-add}(m,r1) \) 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
    
    \[
    \begin{align*}
    & \{r1 == 1\} \\
    & \text{lock: exch r1, key ; spin until zero obtained} \\
    & \text{cmpi r1, 0 ;} \\
    & \text{bne lock ;} \\
    & \{\text{lock obtained}\}
    \end{align*}
    \]

  - to release, exchange with key
    
    \[
    \begin{align*}
    & \{r1 == 0\} \\
    & \text{unlock: exch r1, key} \\
    & \{\text{lock released}\}
    \end{align*}
    \]

- 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 improve serialization on 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 \( i \)

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

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

\( \text{arrive}[i : i+p-1 \text{ mod } p] > 0 \)
Dissemination barrier: example \( (p = 4) \)

```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;
}
```

```
s = 4

s = 2

s = 1
```

```
0          0          0          0
```