COMP 633 - Parallel Computing

Lecture 10
September 27, 2018

CC-NUMA (1)
CC-NUMA implementation

• Reading for next time
  – Memory consistency models tutorial (sections 1-6, pp 1 -17)
Topics

• Objectives of the next few lectures
  – Examine some implementation issues in shared-memory multiprocessors
    • cache coherence
    • memory consistency
    • synchronization mechanisms

• Why?
  – Correctness
    • memory consistency can be the source of very subtle bugs
  – Performance
    • cache coherence and synchronization mechanisms can have profound performance implications
Cache-coherent shared memory multiprocessor

- Implementations
  - shared bus
    - bus may be a “slotted” ring
  - scalable interconnect
    - fixed per-processor bandwidth

- Effect of CPU write on local cache
  - write-through policy – value is written to cache and to memory
  - write-back policy – value written in cache only; memory updated upon cache line eviction

- Effect of CPU write on remote cache
  - update – remote value is modified
  - invalidate – remote value is marked invalid
Bus-Based Shared-Memory protocols

- “Snooping” caches
  - $C_i$ caches memory operations from $P_i$
  - $C_i$ monitors all activity on bus due to $C_h$ ($h \neq i$)

- **Update protocol with write-through cache**
  - between proc $P_i$ and cache $C_i$
    - read-hit from $P_i$ resolved from $C_i$
    - read-miss from $P_i$ resolved from memory and inserted in $C_i$
    - write (hit or miss) from $P_i$ updates $C_i$ and memory [write-through]

  - between cache $C_i$ and cache $C_h$
    - if $C_i$ writes a memory location cached at $C_h$, then $C_h$ is **updated** with new value

- consequences
  - every write uses the bus
  - doesn’t scale
Bus-Based Shared-Memory protocols

- Invalidation protocol with write-back cache
  - Cache blocks can be in one of three states:
    - INVALID — The block does not contain valid data
    - SHARED — The block is a current copy of memory data
      - other copies may exist in other caches
    - EXCLUSIVE — The block holds the only copy of the correct data
      - memory may be incorrect, no other cache holds this block

- Handling exclusively-held blocks
  - Processor events
    - cache is block “owner”
      » reads and writes are local
  - Snooping events
    - on detecting a read-miss or write-miss from another processor to an exclusive block
      » write-back block to memory
      » change state to shared (on ext read-miss) or invalid (on ext write-miss)
Invalidation protocol: example

- **P1**: 
  - R: Shared
  - W: Excl

- **P2**: 
  - R: Shared
  - W: Invalid

- **P3**: 
  - R: Shared
  - W: Invalid

- **X1**: 
  - Excl
  - Invalid

- **X2**: 
  - Excl

- **X3**: 
  - Invalid

- **X4**: 
  - Invalid

**Notes:**
- The diagram illustrates the invalidation protocol in a distributed system.
- Each node (P1, P2, P3) represents a processor or a node in the system.
- The rectangles (X1, X2, X3, X4) represent data blocks.
- The arrows show the direction of operations (read or write) and the colors indicate the state of the data blocks (shared, exclusive, invalid).
- The protocol ensures consistency and coherence across the distributed system.
Implementation: FSM per cache line

- Action in response to CPU event
  - Invalid
    - CPU read
    - CPU write
  - Shared
    - Eviction
    - Place read-miss on bus
    - CPU read
  - Excl
    - Write-back block
    - Place write-miss on bus

- Action in response to bus event
  - Invalid
    - Write-miss for this block
    - Read-miss for this block
  - Shared
    - Write-back block
  - Excl
    - Write-back block
Scalable shared memory: directory-based protocols

- The Stanford DASH multiprocessor
  - Processing clusters are connected via a scalable network
    - Global memory is distributed equally among clusters

- Caching is performed using an ownership protocol
  - Each memory block has a “home” processing cluster
  - At each cluster, a directory tracks the location & state of each cached block whose home is on the cluster
Directories

- Directories track location & state of all cache blocks
  - 16 MB cluster memories
  - 16 byte cache blocks
  - 2+ MB storage overhead per directory
Cache coherence in DASH

- Caching is based on an ownership model
  - uncached, clean, & dirty states
- Home cluster is the owner for all its uncached and clean blocks
- Any one cache can own the only copy of a dirty block
Cache coherence in DASH: Read miss

• Check local cluster caches first...
  – If found and CLEAN then copy
  – If found and DIRTY then make CLEAN and copy

• If not found consult desired block’s home directory
  – If CLEAN or UNCAHCEDED then block is sent to requestor
  – If DIRTY then request is forwarded to cluster where block is cached. Remote cluster makes block CLEAN and sends copy to requestor

• To make a block CLEAN
  – Send copy to owning cluster
  – mark CLEAN
Cache coherence in DASH: Writes

- Writing processor must first become block’s owner

- If block is cached at requesting processor and block is...
  - DIRTY, then write can proceed
  - CLEAN, then home directory must invalidate all copies

- If block is not cached locally but is cached on the cluster
  - a local block transfer is performed
  - home directory is updated if the state was CLEAN
Cache coherence in DASH: Writes

- If block is not cached on local cluster then block’s home directory is contacted.

- If block is...
  - UNCACHED — Block is marked DIRTY and sent to requestor.
  - CLEAN — Block is marked as DIRTY and messages sent to caching clusters to invalidate their copies.
  - DIRTY — Request is forwarded to caching cluster. There the block is invalidated and forwarded to requestor.
Intel cache coherence

- basically a directory-based protocol like DASH with 2 or 4 clusters
- each package (socket) is a cluster with p cores on a slotted ring
Intel physical organization

- up to 4 sockets
- up to 22 cores per socket
- up to 44 thread contexts
Mapping OpenMP threads to hardware

- Mapping threads to maximize data locality
  - `KMP_AFFINITY = “granularity=fine,compact”`

```
machine

socket 0  socket 1
  core 0  core 1  core 0  core 1
          0  1  2  3  4  5  6  7  OpenMP thread id
```
Mapping OpenMP threads to hardware

- Mapping threads to maximize bandwidth without data locality
  - `KMP_AFFINITY = “granularity=fine,scatter”`

```
0  4  2  6  1  5  3  7  OpenMP thread id
```

```
0
4
2
6
1
5
3
7
thread context
```
Mapping OpenMP threads to hardware

- Mapping threads to maximize data locality and equal thread progress
  - KMP_AFFINITY = "granularity=fine,compact,1,0"
  - OMP_NUM_THREADS = 4

```
0 4 1 5 2 6 3 7 OpenMP thread id
```
Mapping OpenMP threads to hardware

- Mapping threads to maximize bandwidth and equal thread progress
  - `KMP_AFFINITY = "granularity=fine,scatter"`
  - `OMP_NUM_THREADS = 4`

```
0 4 2 6 1 5 3 7
```

```
machine

socket 0

core 0

thread context

socket 1

core 1

OpenMP thread
```
Coherence and Consistency

- **Coherence**
  - behavior of a single memory location
  - viewed from a single processor
  - read returns “most recent” written value

- **Consistency**
  - behavior of multiple memory locations read and written by multiple processors
  - viewed from one or more of the processors
  - read may not return the “most recent” value
    - What are the permitted ordering among reads and writes of several memory locations?