COMP 633 - Parallel Computing

Lecture 10
September 21, 2021

CC-NUMA (1)
CC-NUMA Implementation

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

• Single processor optimization
  – cache optimization
  – vectorization
  – general remarks

• Shared-memory multiprocessors
  – cache implementation
    • cache coherence
    • cache consistency
  – shared memory implementations
    • bus-based protocols
    • directory-based protocols

• OpenMP thread mapping to processors
Single-processor optimization

• Cache optimization
  – locality of reference
    • the unit of transfer to/from memory is a cache line (64 bytes)
    • maximize utility of the transferred data
      – an array of structs?
      – a struct of arrays?
  – keep in mind cache capacities
    • L1 and L2 are local to the core
    • L3 is local to the socket
    • first touch principle for page faults
      – the page frame is allocated in the physical memory attached to the socket
Single-processor optimization

• Vectorization
  – vector operations
    • generated by compiler based on analysis of data structures and loops
      – unrolls the loop iterations and generates vector instructions
    • dependencies between loop iterations can inhibit vectorization
    • automatic vectorization generally works quite well
      – icc can generate a vectorization report (see Intel Advisor: Vectorization)

• General remarks
  – use -Ofast flag for maximum analysis and optimization
    • use -O3 to retain precise floating point arithmetic
  – performance tuning can be time consuming
  – plan for parallelism
    • minimize arrays of pointers to dynamically allocated values
      – vectorization will be slowed by having to fetch all the values serially
    • avoid mixed reads and writes of shared data in a cache line
      – a write invalidate copies of the cache line held in other cores
Shared-memory multiprocessor implementation

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

- Why?
  - Correctness
    - memory consistency (or lack thereof) can be the source of very subtle bugs
  - Performance
    - cache coherence and synchronization mechanisms can have profound performance implications
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?
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
Implementation: FSM per cache line

- Action in response to CPU event

  - Invalid
    - CPU write
    - Eviction
    - Place write-miss on bus
  - Excl
    - CPU read
    - CPU write
  - Shared
    - CPU read
    - Eviction
    - Place read-miss on bus
  - Invalid
    - Write-miss for this block
    - Write-back block
  - Excl
    - Read-miss for this block
    - Write-back block
  - Shared
    - Write-miss for this block
    - Write-back block

- Action in response to bus event
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 clusters
  - 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 – invalid, shared, & exclusive states
- Home cluster is the owner for all its invalid and shared blocks
- Any one cache can own the only copy of a exclusive block
Cache coherence in DASH: Read miss

- Check local cluster caches first...
  - If found and SHARED then copy
  - If found and EXCL then make SHARED and copy

- If not found consult desired block’s home directory
  - If SHARED or UNCACHED then block is sent to requestor
  - If EXCL then request is forwarded to cluster where block is cached. Remote cluster makes block SHARED and sends copy to requestor

- To make a block SHARED
  - Send copy to owning cluster
  - mark SHARED
Writing processor must first become block’s owner

If block is cached at requesting processor and block is...
   – EXCL, then write can proceed
   – SHARED, then home directory must invalidate all copies and convert to EXCL

If block is not cached locally but is cached on the cluster
   – a local block transfer is performed (invalidating local copies)
   – home directory is updated to EXCL if the state was SHARED
Cache coherence in DASH: Writes

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

- If block is...
  - UNCAHCHED — Block is marked EXCL and sent to requestor
  - SHARED — Block is marked as EXCL and messages sent to caching clusters to invalidate their copies
  - EXCL — Request is forwarded to caching cluster. There the block is invalidated and forwarded to requestor
Intel cache coherence (skylake)

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

- up to 4 sockets
- up to 28 cores per socket
- up to 56 thread contexts (28 threads and 28 hyperthreads)
Mapping OpenMP threads to hardware (1)

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

Note: we use a fictional machine with 2 sockets and 4 cores with hyperthreads to illustrate these mappings.

Nearby threads-ids tend to share more lower-level cache.
Mapping OpenMP threads to hardware (2)

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

```
machine

socket 0
  core 0
  thread context
  0  4
  core 1
  2  6

socket 1
  core 0
  thread context
  1  5
  core 1
  3  7
OpenMP thread id
```
Mapping OpenMP threads to hardware (3)

- Mapping threads to maximize data locality and equal thread progress
  - KMP_AFFINITY = “granularity=fine,compact,1,0”
  - OMP_NUM_THREADS = 4
Mapping OpenMP threads to hardware (4)

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

![Diagram of machine with sockets and cores](image)

- OpenMP thread context:
  - Thread 0 and 4
  - Thread 2 and 6
  - Thread 1 and 5
  - Thread 3 and 7