Computer Architecture
A Quantitative Approach, Fifth Edition

Chapter 5

Multiprocessors and Thread-Level Parallelism

 1

Introduction

Introduction

- Thread-Level parallelism
  - Have multiple program counters
  - Uses MIMD model
  - Targeted for tightly-coupled shared-memory multiprocessors
- For *n* processors, need *n* threads
- Amount of computation assigned to each thread = grain size
  - Threads can be used for data-level parallelism, but the overheads may outweigh the benefit

 2

Why Parallelism?

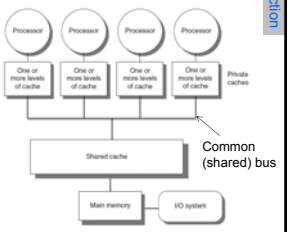
Introduction

- Faster (better performance)
- Less power consumption (think of N processors running at frequency *f* vs. one processor running at *Nf*). **But**, there is also static power
- Limits for single processor performance (scalability)
- Howe much parallelism in the application (Amdahl's law)
- Redundancy and reliability

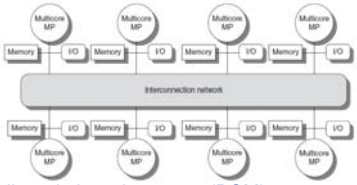
 3

Types -- SMP

- Symmetric multiprocessors (SMP)
  - Small number of cores
  - AKA Tightly coupled multiprocessors
  - Share single memory with uniform memory latency
  - Bus is a bottleneck
  - Most of the communication is handled by OS/HW
  - Existing multi-core



Types -- DSM



- Distributed shared memory (DSM)
  - Memory distributed among processors
  - Loosely Coupled multiprocessors
  - Non-uniform memory access/latency (NUMA)
  - Processors connected via direct (switched) and non-direct (multi-hop) interconnection networks
  - Communication handled by programmer (message passing) (Synchronization explicitly required

Ocean Kernel


```
Procedure Solve(A)
begin
  diff = done = 0;
  while (!done) do
     diff = 0;
     for i ← 1 to n do
       for j ← 1 to n do
         temp = A[i,j];
         A[i,j] ← 0.2 * (A[i,j] + neighbors);
         diff += abs(A[i,j] – temp);
       end for
     end for
     if (diff < TOL) then done = 1;
  end while
end procedure
```

Slide credit: Rajeev Balasubramonian

Shared Address Space Model


```
int  n, nprocs;
float  **A, diff;
LOCKDEC(diff_lock);
BARDEC(bar1);

main()
begin
  read(n); read(nprocs);
  A ← G_MALLOC();
  initialize (A);
  CREATE (nprocs,Solve,A);
  WAIT_FOR_END (nprocs);
end main
```

```
procedure Solve(A)
  int i, j, pid, done=0;
  float temp, mydiff=0;
  int mymin = 1 + (pid * n/procs);
  int mymax = mymin + n/nprocs -1;
  while (!done) do
    mydiff = diff = 0;
    BARRIER(bar1,nprocs);
    for i ← mymin to mymax
      for j ← 1 to n do
        …
      endfor
    endfor
    LOCK(diff_lock);
    diff += mydiff;
    UNLOCK(diff_lock);
    BARRIER (bar1, nprocs);
    if (diff < TOL) then done = 1;
    BARRIER (bar1, nprocs);
  endwhile
```

Slide credit: Rajeev Balasubramonian

7

Message Passing Model


```
main()
  read(n); read(nprocs);
  CREATE (nprocs-1, Solve);
  Solve();
  WAIT_FOR_END (nprocs-1);

procedure Solve()
  int i, j, pid, nn = n/nprocs, done=0;
  float temp, tempdiff, mydiff = 0;
  myA ← malloc(…)
  initialize(myA);
  while (!done) do
    mydiff = 0;
    if (pid != 0)
      SEND(&myA[1,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      SEND(&myA[nn,0], n, pid+1, ROW);
    if (pid != 0)
      RECEIVE(&myA[0,0], n, pid-1, ROW);
    if (pid != nprocs-1)
      RECEIVE(&myA[nn+1,0], n, pid+1, ROW);
    for i ← 1 to nn do
      for j ← 1 to n do
        …
      endfor
    endfor
    if (pid != 0)
      SEND(mydiff, 1, 0, DIFF);
      RECEIVE(done, 1, 0, DONE);
    else
      for i ← 1 to nprocs-1 do
        RECEIVE(tempdiff, 1, *, DIFF);
        mydiff += tempdiff;
      endfor
      if (mydiff < TOL) done = 1;
      for i ← 1 to nprocs-1 do
        SEND(done, 1, I, DONE);
      endfor
    endif
  endwhile
```

Slide credit: Rajeev Balasubramonian

8

Design issue

- Shared memory synchronization
  - How to handle locks, atomic operations
- Cache coherence
  - How to ensure correct operation in the presence of private caches
- Memory consistency: Ordering of memory operations
  - What should the programmer expect the hardware to provide?
- Shared resource management
- Communication: Interconnects

Slide credit : Onur Mutlu

Programming Issues

- Load imbalance
  - How to partition a single task into multiple tasks
- Synchronization
  - How to synchronize (efficiently) between tasks
  - How to communicate between tasks
  - Locks, barriers, pipeline stages, condition variables, semaphores, atomic operations, …
- Ensuring correct operation while optimizing for performance

Slide credit : Onur Mutlu

 10

Example

- 2 processors trying to enter a critical section
- What could go wrong
- Blackboard ….

 11

Cache Coherence

- Processors may see different values through their caches:

Time	Event	Cache contents for processor A	Cache contents for processor B	Memory contents for location X
0				1
1	Processor A reads X	1		1
2	Processor B reads X	1	1	1
3	Processor A stores 0 into X	0	1	0

 12

Memory ordering

- In a single processor
  - Load and stores are executed according to program order
  - Sometimes, out-of-order execution, but that doesn't change the semantics
- Same thing happens every time we run the program (good for debugging)

 13

Memory ordering

- multiprocessors
  - Memory operations happens concurrently
  - We need some sort of global order
  - If completely independent, we don't care
  - The problem is when they share some data.

 14

Cache Coherence

- Coherence: How do other processors see a memory update?
- Writes to the same location by any two processors are seen in the same order by all processors
- Consistency
  - When a written value will be returned by a read
  - If a processor writes location A followed by location B, any processor that sees the new value of B must also see the new value of A

 15

Cache Coherence -- more

- A memory system is coherent if

1. A read by P to location X that follows a write by P to location X with no writes to X in between (by any processor) returns the value written by P.
2. A read by processor p1 to X that follows a write by P2 to X returns the value written by P2 if the read and write are sufficiently separated in time, and no other writes to X occurred between the two accesses.
3. Writes to the same location are *serialized* Two writes by two processors to the same location are seen in the same order by all processors

 16

Enforcing Coherence

- Coherent caches provide:
  - *Migration*: movement of data
  - *Replication*: multiple copies of data
- Cache coherence protocols
  - Directory based
    - Sharing status of each block kept in one location (distributed memory model).
  - Snooping
    - Each core tracks sharing status of each block (SMP).

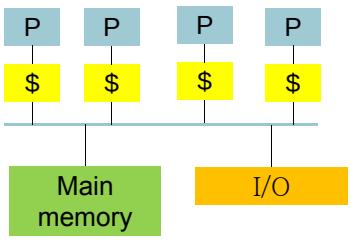
 17

Cache Coherence Protocols

1. **Directory based** — Sharing status of a block of physical memory is kept in just one location, the directory
2. **Snooping** — Every cache with a copy of data also has a copy of sharing status of block, but no centralized state is kept
   - All caches are accessible via some broadcast medium (a bus or switch)
   - All cache controllers monitor or snoop on the medium to determine whether or not they have a copy of a block that is requested on a bus or switch access

18

SMP or Centralized Shared Memory



19

Snooping Protocols

- The processor may have an exclusive access to the data, in this case the processor may change it. This is knows as *write invalidate*

Processor activity	Bus	content of A	Content of B	Memory
				0
A reads X	Miss	0	------	0
B reads X	Miss	0	0	0
A writes X	INV X	1	----	~~0~~
B reads X	Miss	1	1	1

20

Snooping Protocols

- The alternative is to update *write update* or *write broadcast* and is only done for shared blocks

Processor activity	Bus	content of A	Content of B	Memory
				0
A reads X	Miss	0	------	0
B reads X	Miss	0	0	0
A writes X	INV X	1	1	1
B reads X	Miss	1	1	1

21

Snoopy Coherence Protocols

- Write invalidate
  - On write, invalidate all other copies
  - Use bus itself to serialize
    - Write cannot complete until bus access is obtained

Processor activity	Bus activity	Contents of processor A's cache	Contents of processor B's cache	Contents of memory location X
				0
Processor A reads X	Cache miss for X	0		0
Processor B reads X	Cache miss for X	0	0	0
Processor A writes a 1 to X	Invalidation for X	1		0
Processor B reads X	Cache miss for X	1	1	1

- Write update
  - On write, update all copies

Comparison

- Multiple writes to the same word with no intervening reads require multiple write broadcast for an update protocol, and one invalidate for invalidate protocols.
- With multiword cache blocks, write to multiple words (bytes) in the same line require multiple broadcast, while only one invalidate (assuming no intervening reads).
- The delay between writing a word in a processor, and reading it by another processor is less in write update

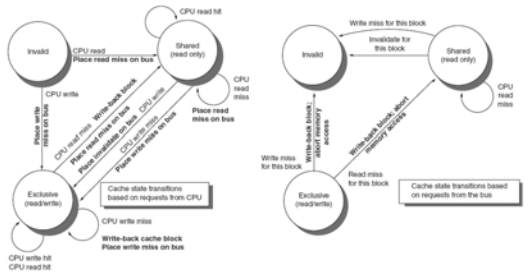
Snooping Coherence Protocols

- Locating an item when a read miss occurs
  - In write-back cache, the updated value must be sent to the requesting processor
- Cache lines marked as shared or exclusive/modified
  - Only writes to shared lines need an invalidate broadcast
    - After this, the line is marked as exclusive

Snooping Coherence Protocols

Request	Source	State of addressed cache block	Type of cache action	Function and explanation
Read hit	Processor	Shared or modified	Normal hit	Read data in local cache.
Read miss	Processor	Invalid	Normal miss	Place read miss on bus.
Read miss	Processor	Shared	Replacement	Address conflict miss: place read miss on bus.
Read miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place read miss on bus.
Write hit	Processor	Modified	Normal hit	Write data in local cache.
Write hit	Processor	Shared	Coherence	Place invalidate on bus. These operations are often called upgrade or *ownership* misses, since they do not fetch the data but only change the state.
Write miss	Processor	Invalid	Normal miss	Place write miss on bus.
Write miss	Processor	Shared	Replacement	Address conflict miss: place write miss on bus.
Write miss	Processor	Modified	Replacement	Address conflict miss: write-back block, then place write miss on bus.
Read miss	Bus	Shared	No action	Allow shared cache or memory to service read miss.
Read miss	Bus	Modified	Coherence	Attempt to share data: place cache block on bus and change state to shared.
Invalidate	Bus	Shared	Coherence	Attempt to write shared block; invalidate the block.
Write miss	Bus	Shared	Coherence	Attempt to write shared block; invalidate the cache block.
Write miss	Bus	Modified	Coherence	Attempt to write block that is exclusive elsewhere; write-back the cache block and make its state invalid in the local cache.

Snooping Coherence Protocols

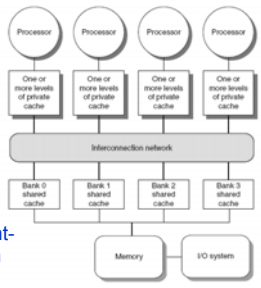


Snoopy Coherence Protocols

- Complications for the basic MSI protocol:
  - Operations are not atomic
    - E.g. detect miss, acquire bus, receive a response
    - Creates possibility of deadlock and races
    - One solution: processor that sends invalidate can hold bus until other processors receive the invalidate
- Extensions:
  - Add exclusive state to indicate clean block in only one cache (MESI protocol)
    - Prevents needing to write invalidate on a write
  - Owned state

Coherence Protocols: Extensions

- Shared memory bus and snooping bandwidth is bottleneck for scaling symmetric multiprocessors
  - Duplicating tags
  - Place directory in outermost cache
  - Use crossbars or point-to-point networks with banked memory



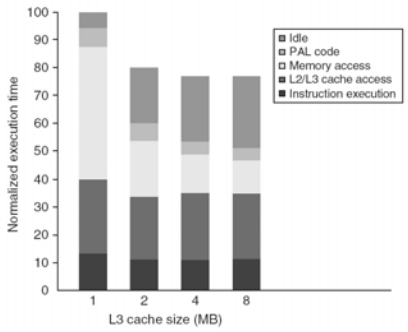
Coherence Protocols

- AMD Opteron:
  - Memory directly connected to each multicore chip in NUMA-like organization
  - Implement coherence protocol using point-to-point links
  - Use explicit acknowledgements to order operations

Performance

- Coherence influences cache miss rate
  - Coherence misses
    - True sharing misses
      - Write to shared block (transmission of invalidation)
      - Read an invalidated block
    - False sharing misses
      - Read an unmodified word in an invalidated block

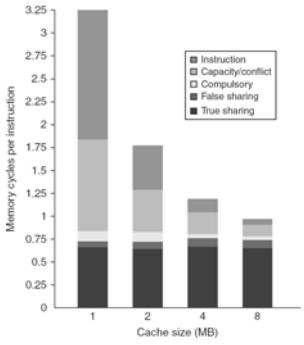
Performance Study: Commercial Workload



Performance of Symmetric Shared-Memory Multiprocessors

31

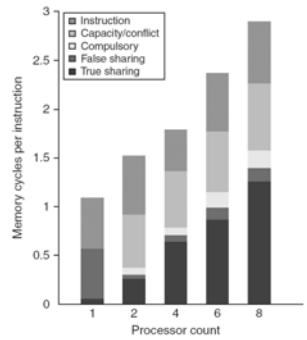
Performance Study: Commercial Workload



Performance of Symmetric Shared-Memory Multiprocessors

32

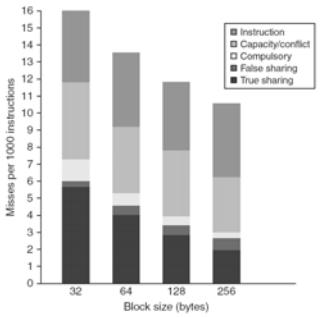
Performance Study: Commercial Workload



Performance of Symmetric Shared-Memory Multiprocessors

33

Performance Study: Commercial Workload

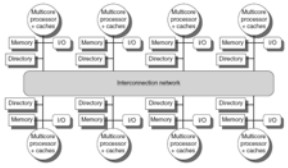


Performance of Symmetric Shared-Memory Multiprocessors

34

Directory Protocols

- Directory keeps track of every block
  - Which caches have each block
  - Dirty status of each block
- Implement in shared L3 cache
  - Keep bit vector of size = # cores for each block in L3
  - Not scalable beyond shared L3
- Implement in a distributed fashion:



Distributed Shared Memory and Directory-Based Coherence

35

Directory Protocols

- For each block, maintain state:
  - Shared
    - One or more nodes have the block cached, value in memory is up-to-date
    - Set of node IDs
  - Uncached
  - Modified
    - Exactly one node has a copy of the cache block, value in memory is out-of-date
    - Owner node ID
- Directory maintains block states and sends invalidation messages

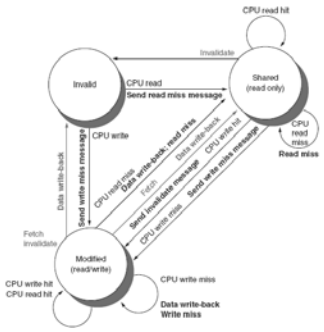
Distributed Shared Memory and Directory-Based Coherence

36

Messages

Message type	Source	Destination	Message contents	Function of this message
Read miss	Local cache	Home directory	P, A	Node P has a read miss at address A; request data and make P a read sharer.
Write miss	Local cache	Home directory	P, A	Node P has a write miss at address A; request data and make P the exclusive owner.
Invalidate	Local cache	Home directory	A	Request to send invalidates to all remote caches that are caching the block at address A.
Invalidate	Home directory	Remote cache	A	Invalidate a shared copy of data at address A.
Fetch	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; change the state of A in the remote cache to shared.
Fetch/invalidate	Home directory	Remote cache	A	Fetch the block at address A and send it to its home directory; invalidate the block in the cache.
Data value reply	Home directory	Local cache	D	Return a data value from the home memory.
Data write-back	Remote cache	Home directory	A, D	Write-back a data value for address A.

Directory Protocols



Directory Protocols

- For uncached block:
  - Read miss
    - Requesting node is sent the requested data and is made the only sharing node, block is now shared
  - Write miss
    - The requesting node is sent the requested data and becomes the sharing node, block is now exclusive
- For shared block:
  - Read miss
    - The requesting node is sent the requested data from memory, node is added to sharing set
  - Write miss
    - The requesting node is sent the value, all nodes in the sharing set are sent invalidate messages, sharing set only contains requesting node, block is now exclusive

Directory Protocols

- For exclusive block:
  - Read miss
    - The owner is sent a data fetch message, block becomes shared, owner sends data to the directory, data written back to memory, sharers set contains old owner and requestor
  - Data write back
    - Block becomes uncached, sharer set is empty
  - Write miss
    - Message is sent to old owner to invalidate and send the value to the directory, requestor becomes new owner, block remains exclusive

Synchronization

- Basic building blocks:
  - Atomic exchange
    - Swaps register with memory location
  - Test-and-set
    - Sets under condition
  - Fetch-and-increment
    - Reads original value from memory and increments it in memory
  - Requires memory read and write in uninterruptable instruction
  - load linked/store conditional
    - If the contents of the memory location specified by the load linked are changed before the store conditional to the same address, the store conditional fails

Implementing Locks

- Spin lock
  - If no coherence:

```
        DADDUI  R2,R0,#1
lockit: EXCH    R2,0(R1)    ;atomic exchange
        BNEZ    R2,lockit   ;already locked?
```

  - If coherence:

```
lockit: LD      R2,0(R1)    ;load of lock
        BNEZ    R2,lockit   ;not available-spin
        DADDUI  R2,R0,#1    ;load locked value
        EXCH    R2,0(R1)    ;swap
        BNEZ    R2,lockit   ;branch if lock wasn't 0
```


Implementing Locks

- Advantage of this scheme: reduces memory traffic

Step	P0	P1	P2	Coherence state of lock at end of step	Bus/directory activity
1	Has lock	Begins spin, testing if lock = 0	Begins spin, testing if lock = 0	Shared	Cache misses for P1 and P2 satisfied in either order. Lock state becomes shared.
2	Set lock to 0	(Invalidate received)	(Invalidate received)	Exclusive (P0)	Write invalidate of lock variable from P0.
3		Cache miss	Cache miss	Shared	Bus/directory services P2 cache miss; write-back from P0; state shared.
4		(Waits while bus/ directory busy)	Lock = 0 test succeeds	Shared	Cache miss for P2 satisfied
5		Lock = 0	Executes swap, gets cache miss	Shared	Cache miss for P1 satisfied
6		Executes swap, gets cache miss	Completes swap: returns 0 and sets lock = 1	Exclusive (P2)	Bus/directory services P2 cache miss, generates invalidate; lock is exclusive.
7		Swap completes and returns 1, and sets lock = 1	Enter critical section	Exclusive (P1)	Bus/directory services P1 cache miss; sends invalidate and generates write-back from P2.
8		Spins, testing if lock = 0			None

Models of Memory Consistency

Processor 1:	Processor 2:
A=0	B=0
…	…
A=1	B=1
if (B==0) …	if (A==0) …

- Should be impossible for both if-statements to be evaluated as true
  - Delayed write invalidate?
- Sequential consistency:
  - Result of execution should be the same as long as:
    - Accesses on each processor were kept in order
    - Accesses on different processors were arbitrarily interleaved

Implementing Locks

- To implement, delay completion of all memory accesses until all invalidations caused by the access are completed
  - Reduces performance!
- Alternatives:
  - Program-enforced synchronization to force write on processor to occur before read on the other processor
    - Requires synchronization object for A and another for B
      - "Unlock" after write
      - "Lock" after read

Relaxed Consistency Models

- Rules:
  - X → Y
    - Operation X must complete before operation Y is done
    - Sequential consistency requires:
      - R → W, R → R, W → R, W → W
  - Relax W → R
    - "Total store ordering"
  - Relax W → W
    - "Partial store order"
  - Relax R → W and R → R
    - "Weak ordering" and "release consistency"

Relaxed Consistency Models

- Consistency model is multiprocessor specific
- Programmers will often implement explicit synchronization
- Speculation gives much of the performance advantage of relaxed models with sequential consistency
  - Basic idea: if an invalidation arrives for a result that has not been committed, use speculation recovery