What is Parallel Computer Architecture?

• A Parallel Computer is a collection of processing elements that cooperate to solve large problems fast
  – how large a collection?
  – how powerful are the elements?
  – how does it scale up?
  – how do they cooperate and communicate?
  – how is data transmitted between processors?
  – what are the primitive abstractions?
  – how does it all translate to performance?
Parallel Computation: Why and Why Not?

• Pros
  – Performance
  – Cost-effectiveness (commodity parts)
  – Smooth upgrade path
  – Fault Tolerance

• Cons
  – Difficult to parallelize applications
  – Requires automatic parallelization or parallel program development
  – Software! AAHHHH!

Flynn Categories

• SISD (Single Instruction Single Data)
  – Uniprocessors

• MISD (Multiple Instruction Single Data)
  – ???

• SIMD (Single Instruction Multiple Data)
  – Examples: Illiac-IV, CM-2, Intel MMX
    » Simple programming model
    » Low overhead
    » Flexibility
    » All custom processors

• MIMD (Multiple Instruction Multiple Data)
  – Examples: Intel 4-way SMP, SUN ES3000, SGI Origin, Cray T3D
    » Flexible
    » Use off-the-shelf microprocessors
Communication Models

• Shared Memory
  – Processors communicate with shared address space
  – Easy on small-scale machines
  – Advantages:
    » Model of choice for uniprocessors, small-scale MPs
    » Ease of programming
    » Lower latency
    » Easier to use hardware controlled caching

• Message passing
  – Processors have private memories, communicate via messages
  – Advantages:
    » Less hardware, easier to design
    » Focuses attention on costly non-local operations

• Can support either model on either HW base

Simple Problem

for i = 1 to N
    A[i] = (A[i] + B[i]) * C[i]
    sum = sum + A[i]

• How do you make this loop parallel to run on many processors?
Simple Problem

- Split the loops
  // Independent iterations
  // Run on up to N processors
  for i = 1 to N
      A[i] = (A[i] + B[i]) * C[i]
  // One last loop to run on one processor
  for i = 1 to N
      sum = sum + A[i]

Small Scale Shared Memory Multiprocessors

- Small number of processors connected to one shared memory
- Memory is equidistant from all processors (UMA)
- Kernel can run on any processor (symmetric MP)
- Intel dual/quad Pentium, IBM, SUN, Compaq, almost everyone
- Some are moving on-chip (e.g., IBM)
Large Scale Shared Memory Multiprocessors

- 100s to 1000s of nodes (processors) with single shared physical address space
- Use General Purpose Interconnection Network
  - Still have cache coherence protocol
  - Use messages instead of bus transactions
  - No hardware broadcast
- Communication Assist
- Cray T3D, T3E, Compaq EV7, SUN ES3000

Message Passing Architectures

- Cannot directly access memory on another node
- IBM SP-2, Intel Paragon
- Cluster of workstations
Important Communication Properties

- Bandwidth
  - Need high bandwidth in communication
  - Cannot scale, but stay close
  - Limits may be in network, memory, and processor
  - Overhead to communicate is a problem in many machines

- Latency
  - Affects performance, since processor may have to wait
  - Affects ease of programming, since requires more thought to overlap communication and computation

- Latency Hiding
  - How can a mechanism help hide latency?
  - Examples: overlap message send with computation, prefetch

Small-Scale—Shared Memory

- Caches serve to:
  - Increase bandwidth versus bus/memory
  - Reduce latency of access
  - Valuable for both private data and shared data

- What about cache coherence?
Cache Coherence Problem (Initial State)

Time

P1

Interconnection Network / Bus

P2

Main Memory

ld r2, x

Cache Coherence Problem (Step 1)

Time

P1

Interconnection Network / Bus

P2

Main Memory

ld r2, x
Cache Coherence Problem (Step 2)

P1

Id r2, x

Interconnection Network / Bus

x

Main Memory

P2

Id r2, x

Cache Coherence Problem (Step 3)

P1

Id r2, x
add r1, r2, r4
st x, r1

Interconnection Network / Bus

x

Main Memory

P2

Id r2, x

© Alvin R. Lebeck 1999
The Problem of Cache Coherence (4)

Coherence vs. Consistency

- Intuition says loads should return latest value
  - what is latest?
- Coherence concerns only one memory location
- Consistency concerns apparent ordering for all locations
- A Memory System is Coherent if
  - can serialize all operations to that location such that,
  - operations performed by any processor appear in program order
    - program order = order defined program text or assembly code
  - value returned by a read is value written by last store to that location
Why Coherence != Consistency

/* initial A = B = flag = 0 */
P1
A = 1; while (flag == 0); /* spin */
B = 1; print A;
flag = 1; print B;

Intuition says printed A = B = 1
Coherence doesn’t say anything, why?

The Sequential Consistency Memory Model

sequential processors issue memory ops in program order

switch randomly set after each memory op
Sufficient Conditions for Sequential Consistency

- Every processor issues memory ops in program order
- Processor must wait for store to complete before issuing next memory operation
- After load, issuing proc waits for load to complete, and store that produced value to complete before issuing next op
- Easily implemented with shared bus.

Potential Solutions

- Snooping Solution (Snoopy Bus):
  - Send all requests for data to all processors
  - Processors snoop to see if they have a copy and respond accordingly
  - Requires broadcast, since caching information is at processors
  - Works well with bus (natural broadcast medium)
  - Dominates for small scale machines (most of the market)
    - probably won’t scale beyond 2-4 processors
- Directory-Based Schemes
  - Keep track of what is being shared in one centralized place
  - Distributed memory => distributed directory (avoids bottlenecks)
  - Send point-to-point requests to processors
  - Scales better than Snoop
  - Actually existed BEFORE Snoop-based schemes
Basic Snoopy Protocols

- **Write Invalidate Protocol:**
  - Multiple readers, single writer
  - Write to shared data: an invalidate is sent to all caches which snoop and invalidate any copies
  - Read Miss:
    - Write-through: memory is always up-to-date
    - Write-back: snoop in caches to find most recent copy

- **Write Broadcast (Update) Protocol:**
  - Write to shared data: broadcast on bus, processors snoop, and update copies
  - Read miss: memory is always up-to-date

- **Write serialization: bus serializes requests**
  - Bus is single point of arbitration

Snoopy Cache-Coherence Protocols

- **Bus provides serialization point for consistency**
  - but, but, what about write-buffers? Later in the semester....

- **Each cache controller “snoops” all bus transactions**
  - relevant transactions if for a block it contains
  - take action to ensure coherence
    - invalidate
    - update
    - supply value
  - depends on state of the block and the protocol

- **Simultaneous Operation of Independent Controllers**
Snoopy Design Choices

- Controller updates state of blocks in response to processor and snoop events and generates bus actions
- Often have duplicate cache tags
- Snoopy protocol
  - set of states
  - state-transition diagram
  - actions
- Basic Choices
  - write-through vs. write-back
  - invalidate vs. update

The Simple Invalidate Snoopy Protocol

- Write-through, no-write-allocate cache
- Actions: PrRd, PrWr, BusRd, BusWr
A 3-State Write-Back Invalidation Protocol

• 2-State Protocol
  + Simple hardware and protocol
  – Bandwidth (every write goes on bus!)

• 3-State Protocol (MSI)
  – Modified
    » one cache has valid/latest copy
    » memory is stale
  – Shared
    » one or more caches have valid copy
  – Invalid

• Must invalidate all other copies before entering modified state
• Requires bus transaction (order and invalidate)

MSI Processor and Bus Actions

• Processor:
  – PrRd
  – PrWr
  – Writeback on replacement of modified block

• Bus
  – Bus Read (BusRd) Read without intent to modify, data could come from memory or another cache
  – Bus Read-Exclusive (BusRdX) Read with intent to modify, must invalidate all other caches copies
  – Writeback (BusWB) cache controller puts contents on bus and memory is updated
  – Definition: cache-to-cache transfer occurs when another cache satisfies BusRd or BusRdX request

• Let’s draw it!
An example

<table>
<thead>
<tr>
<th>Proc Action</th>
<th>P1 State</th>
<th>P2 state</th>
<th>P3 state</th>
<th>Bus Act</th>
<th>Data from</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. P1 read u</td>
<td>S</td>
<td>--</td>
<td>--</td>
<td>BusRd</td>
<td>Memory</td>
</tr>
<tr>
<td>2. P3 read u</td>
<td>S</td>
<td>--</td>
<td>S</td>
<td>BusRd</td>
<td>Memory</td>
</tr>
<tr>
<td>3. P3 write u</td>
<td>I</td>
<td>--</td>
<td>M</td>
<td>BusRdX</td>
<td>Memory or not</td>
</tr>
<tr>
<td>4. P1 read u</td>
<td>S</td>
<td>--</td>
<td>S</td>
<td>BusRd</td>
<td>P3's cache</td>
</tr>
<tr>
<td>5. P2 read u</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>BusRd</td>
<td>Memory</td>
</tr>
</tbody>
</table>

- Single writer, multiple reader protocol
- Why Modified to Shared?
- What if not in any cache?
  - Read, Write produces 2 bus transactions!
4-State (MESI) Invalidation Protocol

• Often called the Illinois protocol
• Modified (dirty)
• Exclusive (clean unshared) only copy, not dirty
• Shared
• Invalid
• Requires shared signal to detect if other caches have a copy of block
• Cache Flush for cache-to-cache transfers
  – Only one can do it though
• What does state diagram look like?

4-State Write-back Update Protocol

• Dragon (Xerox PARC)
• States
  – Exclusive (E): one copy, clean, memory is up-to-date
  – Shared-Clean (SC): could be two or more copies, memory unknown
  – Shared-Modified (SM): could be two or more copies, memory stale
  – Modified (M)
• Adds Bus Update Transaction
• Adds Cache Controller Update operation
• Must obtain bus before updating local copy
• What does state diagram look like?
  – let’s look at the actions first
Basic Snoopy Protocols

- **Write Invalidate versus Broadcast:**
  - Invalidate requires one transaction per write-run
  - Invalidate uses spatial locality: one transaction per block
  - Broadcast has lower latency between write and read
  - Broadcast: BW (increased) vs. latency (decreased) tradeoff

<table>
<thead>
<tr>
<th>Name</th>
<th>Protocol Type</th>
<th>Memory-write policy</th>
<th>Machines using</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write Once</td>
<td>Write invalidate</td>
<td>Write back after first write</td>
<td>First snoopy protocol.</td>
</tr>
<tr>
<td>Synapse N+1</td>
<td>Write invalidate</td>
<td>Write back</td>
<td>1st cache-coherent MPs</td>
</tr>
<tr>
<td>Berkeley</td>
<td>Write invalidate</td>
<td>Write back</td>
<td>Berkeley SPUR</td>
</tr>
<tr>
<td>Illinois</td>
<td>Write invalidate</td>
<td>Write back</td>
<td>SGI Power and Challenge</td>
</tr>
<tr>
<td>“Firefly”</td>
<td>Write broadcast</td>
<td>Write back private, Write through shared</td>
<td>SPARCCenter 2000</td>
</tr>
</tbody>
</table>