CS 211: Computer Architecture

Instructor: Prof. Bhagi Narahari
Dept. of Computer Science
Course URL: www.seas.gwu.edu/~narahari/cs211/

Computer Architecture

- Part I: Processor Architectures
  - starting with simple ILP using pipelining
  - explicit ILP - EPIC
  - key concept: issue multiple instructions/cycle

- Part II: Multi Processor Architectures
  - move from Processor to System level
  - can utilize all the techniques covered thus far
    - i.e., the processors used in a multi-processor can be EPIC
  - move from fine grain to medium/coarse grain
  - assume all processor issues are resolved when discussing system level Multiprocessor design

Some key problems in Multiprocessor Architectures

- Shared Memory vs Distributed Memory
  - Introduction of network into the architecture

- Synchronization and coordination
  - Barrier, critical section

- Network topology and latency

- Flynn's Classification: SIMD, MIMD...

- Parallel algorithm design

- Cache coherence in MIMD shared memory

MIMD

- Multiple instructions, Multiple Data

- shared memory or distributed memory

- Each processor executes its own program
  - processor must store inst and data
  - larger memory required
  - more complex (than SIMD) processors
  - can also have heterogeneous processors
Moving from multiprocessor to distributed systems

- CS 251: Distributed Systems course

Local Calls (Subroutines)

Main program

Library

Remote Calls (Sockets)

Local computer

Remote computer

Object Request Broker (ORB)

Local computer

Remote computer
Java Remote Method Invocation (RMI)

- Parallel algorithms
  - Computational model
- Cache coherency in shared memory systems
- Cluster computing architectures

Scalability

- Performance must scale with
  - system size
  - problem/workload size
- Amdahl’s Law
  - perfect speedup cannot be achieved since there is a inherently sequential part to every program
- Scalability measures
  - Efficiency (speedup per processor)

Parallel Algorithms

- Solving problems on a multiprocessor architecture requires design of parallel algorithms
- How do we measure efficiency of a parallel algorithm?
  - 10 seconds on Machine 1 and 20 seconds on machine 2 – which algorithm is better?
Parallel Algorithm Complexity

- Parallel time complexity
  - Express in terms of input size and System size (num of processors)
  - T(N,P): input size N, P processors
  - Relationship between N and P
    - Independent size analysis – no link between N and P
    - Dependent size – P is a function of N; eg. P=N/2

- Speedup: how much faster than sequential
  - S(P) = T(N,P) / T(N,1)

- Efficiency: speedup per processor
  - S(P)/P

Parallel Computation Models

- Shared Memory
  - Protocol for shared memory...what happens when two processors/processes try to access same data
    - EREW: Exclusive Read, Exclusive Write
    - CREW: Concurrent Read, Exclusive Write
    - CRCW: Concurrent read, Concurrent write

- Distributed Memory
  - Explicit communication through message passing
    - Send/Receive instructions

Formal Models of Parallel Computation

- Alternating Turing machine
- P-RAM model
  - Extension of sequential Random Access Machine (RAM) model
- RAM model
  - One program
  - One memory
  - One accumulator
  - One read/write tape

P-RAM model

- P programs, one per processor
- One memory
  - In distributed memory it becomes P memories
- P accumulators
- One read/write tape
- Depending on shared memory protocol we have
  - CREW P-RAM
  - EREW PRAM
  - CRCW PRAM
**PRAM Model**

- Assumes synchronous execution
- Idealized machine
  - Helps in developing theoretically sound solutions
  - Actual performance will depend on machine characteristics and language implementation

**PRAM Algorithms -- Summing**

- Add N numbers in parallel using P processors
  - How to parallelize?
  - Compute partial sums in a processor, add partial sums in next step
  - N numbers in N proc
  - Next step, N/2 partial sums - use N/4 proc to add them in one step to create N/4 partial sums
  - How many steps?

**Parallel Summing**

- Using N/2 processors to sum N numbers in O(\(\log N\)) time
- Independent size analysis:
  - Do sequential sum on N/P values and then add in parallel
  - Time = O(N/P + log P)

**Analysis of Parallel Sum Algo**

- Speedup = Seq.time/ Parallel time
  \[ T(1/T(N,N)) = N/\log(N) \]
- Efficiency = Speedup/Processors = 1/\log N
- What if we use N/\log N processors?
  - Add N/(N/\log N) numbers sequentially in each P
  - Time = O(N/(N/\log N)) = O(log N)
  - Then do parallel sum of N/\log N partial sums using N/\log N processors
    - Time = O((N/\log N) = O(log N)
  - Total time complexity = O(log N)
  - Speedup T(N, N/\log N) = N/\log N
  - Efficiency = Speedup/Proc = 1
**Parallel Sorting on CREW PRAM**

- Sort N numbers using P processors
  - Assume P unlimited for now.
- Given an unsorted list \((a_1, a_2, \ldots, a_n)\)
  created sorted list \(W\), where \(W[i] < W[i+1]\)
- Where does \(a_1\) go?
  - Compute “rank” of \(a_1\): how many numbers is it greater than?
  - Computing rank requires comparisons between elements

- Using \(P = N^2\) processors
- For each processor \(P(i,j)\) compare \(a_i > a_j\)
  - If \(a_i > a_j\) then \(R[i,j] = 1\) else 0
  - Time = \(O(1)\)
- For each “row” of processors \(P(i,j)\) for \(j = 1\) to \(j = N\) do parallel sum to compute rank
  - Compute \(R[i] = \text{sum of } R[i,j]\)
  - Time = \(O(\log N)\)
- Write \(a_i\) into \(W[R(i)]\)
- Total time complexity = \(O(\log N)\)

**Analysis of Parallel Sorting**

- Time \(T(N, N^2) = O(\log N)\)
- Speedup = \(N \log N / \log N = N\)
- Efficiency = \(N/N^2 = 1/N\)
- Use same trick as before:
  - \((N/ \log N)\) processors in each row, \(N\) rows
  - Each proc does \((N/(N/\log N))\) compares in \(O(\log N)\) time
  - Use \(N/\log N\) proc to add \(N\) numbers in time \(O(\log N)\)
  - Total time = \(O(\log N)\)
  - Efficiency = \(\log N/N\)

**Parallel Matrix Multiplication**

For \(i = 1\) to \(N\) do
  for \(j = 1\) to \(N\) do
    for \(k = 1\) to \(N\) do
      \(S[i,j,k] = A[i,k]*B[k,j];\)
      \(C[i,j] = C[i,j] + C[i,j,k]\)

How many \(S[i,j,k]\) ?
How many times is each \(S[i,j,k]\) used?
Parallel Matrix Multiplication

Rewrite the equation in terms of \( S[i,j,k] \):
First compute all \( S[i,j,k] \)
For \( i=1 \) to \( N \)
  for \( j=1 \) to \( N \)
    for \( k=1 \) to \( N \)
      \( C[i,j] = C[i,j] + S[i,j,k] \);

The above is \( N^2 \) independent values \( C[i,j] \)
each is \( N \) additions
Using \( N^3 \) processors, we can do this in \( \log N \) time!

Parallel Algorithms

- Design of parallel algorithm has to take system architecture into consideration
- Must minimize interprocessor communication in a distributed memory system
  - Communication time is much larger than computation
  - Comm. Time can dominate computation if not problem is not "partitioned" well
- Efficiency

Computation-Communication Tradeoffs

- As we scale processors, more parallelism in architecture
  - Can decrease compute time
  - More processors have to communicate
    - Time to communicate/synchronize?
- Hidden issues?
  - Memory: what happens when we increase processors/system-size?
  - I/O time: how does this balance against compute and communication time?
- Degree of parallelism in the application?

Flynn’s Classification: SIMD Architectures

- Single Instruction stream, Multiple Data stream
  - Each processor executes same instruction on different data
- Efficient in applications requiring large data processing
  - Low level image processing
  - Multimedia processing
  - Scientific computing
- Synchronization Implicit
  - All processors are in lock step with control unit
SIMD Architectures

- **Control Unit (CU)**
  - Broadcasts instructions to processors
  - Has memory for program
  - Executes control flow instructions (branches)

- **Processing elements (PE)**
  - Data distributed among PE memories
  - Each PE can be enabled or disabled using Mask
  - MASK instruction broadcast by CU

SIMD... PE Organization

- **Simple processors**
  - Do not need to fetch instructions
  - Can be simple microcontrollers
- **CPU**
- **Local memory to store data**
- **General purpose registers**
- **Address register – addr of PE**
- **Data transfer registers for network (DTR_in, DTR_out)**
- **Status flag – enabled/disabled**
- **Index register – used in mem access**
  - Offset by x_i in mem i of PE i

Processing on SIMD

- CU broadcasts instructions
- PE executes – can be simple decode and execute units
- CU can also broadcast a data value
- Time taken to process a task is time to complete tasks on all processors
- All processors execute same instruction in one cycle
  - Note also that processors are hardware synchronized (tied to same clock ?)

Matrix Multiplication on SIMD using CU

- Assume we have N processors in SIMD configuration
- Algorithm to multiply N by N matrix using the CU to broadcast an element
  - For i := 1 to N do
    - For j := 1 to N do
      - C[i,j] := 0
      - for k :=0 to N do
  - Note each row of A is required N times for C[i,1],C[i,2],…..C[i,N]
Matrix Multiplication on SIMD using CU

- Assume each processor $P_k$ stores column $k$ of matrix $B$
- CU can broadcast the current value of $A$
- Each processor $k$ computes $C[i,k]$ for all values of $i$
  - Processor $k$ computes column $k$ of result matrix

Sample Code

For $i := 1$ to $N$ do
  In Parallel for ALL processors $P_k$
    (i.e., enable all processors)
    Broadcast $i$ /* send value of $i$ to all proc */
    $C[i] := 0$ /*initialize $C[i]$ to 0 in all proc $k$ */
  For $j := 1$ to $N$ do
    Broadcast $j$
    Broadcast $A[i,j]$
    $\text{MULT } A[i,j], B[j] \rightarrow \text{temp}$
    $\text{ADD } C[i], \text{temp} \rightarrow C[i]$
  Endfor (j loop)
Endfor

Time Analysis

- There are $N^2$ iterations at control unit
  - Time taken is $N^2$
- Instructions are broadcast to all PE
- Essentially the $k$ loop has been parallelized
  - Using $k$ processors
- Requires each processor store $N$ elements of $B$ and $N$ elements of result matrix
- Ideal speedup and efficiency
  - Got speedup of $N$ using $N$ processors for 100% efficiency

Storage Rules

- In previous example, the algorithm required that a row of $B$ be executed upon at each cycle
  - Since $B$ was stored column wise, this was not a problem
- What if a column of $B$ has to be processed at each cycle
  - Since an entire column is stored in one processor, this requires $N$ cycles
  - No speedup and waste of $N$ processors
- Need to come up with better ways to store matrices
  - Allow row or column fetching in parallel
  - Skew Storage Rules allow this
Multiprocessor cache design
Cluster and Networked Computing

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

- What about cache consistency?

**What Does Coherency Mean?**

- Informally:
  - "Any read must return the most recent write"
  - Too strict and too difficult to implement
- Better:
  - "Any write must eventually be seen by a read"
  - All writes are seen in proper order ("serialization")
- Two rules to ensure this:
  - "If P writes x and P1 reads it, P's write will be seen by P1 if the read and write are sufficiently far apart"
  - Writes to a single location are serialized: seen in one order
    - Latest write will be seen
    - Otherwise could see writes in illogical order (could see older value after a newer value)
Potential HW Coherence 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)

- **Directory-Based Schemes (discuss later):**
  - Keep track of what is being shared in 1 centralized place (logically)
  - Distributed memory => distributed directory for scalability (avoids bottlenecks)
  - Send point-to-point requests to processors via network
  - Scales better than Snooping
  - Actually existed BEFORE Snooping-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 any copies
  - Read miss: memory is always up-to-date

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

Basic Snoopy Protocols

- **Write Invalidate versus Broadcast:**
  - Invalidate requires one transaction per write-run
    - Broadcast will require write updates for each write, even if done by the same processor
  - Invalidate uses spatial locality: one transaction per block
    - Broadcast has to work with each word that is updated
  - Broadcast has lower latency between write and read
  - More bus traffic created by Broadcast
    - Invalidate scales better...
      - More common solution today

An Example Snoopy Protocol

- Invalidation protocol, write-back cache
- Each block of memory is in one state:
  - Clean in all caches and up-to-date in memory (Shared)
  - OR Dirty in exactly one cache (Exclusive)
  - OR Not in any caches
- Each cache block is in one state (track these):
  - Shared: block can be read
  - OR Exclusive: cache has only copy, its writable, and dirty
  - OR Invalid: block contains no data
- Read misses: cause all caches to snoop bus
- Writes to clean line are treated as misses
States and Events

- Each cache line/block is in:
  - Shared, Exclusive, Invalid
  - Processor can read line in shared state
  - Processor can write into exclusive line that it owns
  - Modifying a shared line requires changing state to exclusive via invalidating all copies of this block in other caches
    - Intention to modify must be broadcast to all other caches so that each cache can invalidate the copy it holds

- States and Events

  - Each cache line is in Shared/Exclusive/Invalid state
  - Transitions between states occur in response to local events in CPU and events in other caches observed on bus
    - Local CPU events: Read Miss, Read Hit, Write hit, Write Miss
    - Bus events: bus write miss, bus read miss

State Transitions & CPU Events

- CPU read hit: never a problem, and no state change
- CPU write hit for a line in exclusive state: no problem, no state change
- CPU Write hit for line in shared state:
  - Change state to exclusive
  - Signal write miss to other caches (using bus) causing them to invalidate their copies
- CPU Read miss or Write miss:
  - Signal the event on the bus
  - Bring the missing line/block into cache in shared or exclusive mode

State transitions & Bus Events

- Write miss on bus for cache line in exclusive state
  - Write back line/block to memory
  - Change state to invalid
- Read miss on bus for cache in exclusive state
  - Write back to memory
  - Change state to shared
- request for access to exclusive line appears on bus
  - use of writeback policy forces owner of line/block to intervene to make sure stale data not used
  - thus, in addition to writeback, transitions from exclusive to shared and invalid states require memory accesses as a result of read or write miss be inhibited
  - A memory access in progress may be aborted
Example states transitions

- CPU read miss: write back cache line, signal read miss on bus
  > Read miss is encountered for some desired line
  > Exclusive line (which is dirty) must be written back to memory before it can be replaced with the desired line
  > New line assumes shared state because miss occurred for a read operation

Snoopy-Cache State Machine-I

- State machine for CPU requests for each cache block

Snoopy-Cache State Machine-II

- State machine for CPU requests for each cache block

Snoopy-Cache State Machine-III

- State machine for CPU requests for each cache block and for Bus requests for each cache block
## Example

### Processor 1

<table>
<thead>
<tr>
<th>Step</th>
<th>State</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
</tr>
</tbody>
</table>

### Processor 2

<table>
<thead>
<tr>
<th>Step</th>
<th>State</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P2: Write 20 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>20</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Excl.</td>
<td>A2</td>
<td>40</td>
</tr>
</tbody>
</table>

### Bus Memory

<table>
<thead>
<tr>
<th>Step</th>
<th>State</th>
<th>Addr</th>
<th>Value</th>
<th>Proc. Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>10</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Excl.</td>
<td>A1</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 20 to A1</td>
<td>Excl.</td>
<td>A1</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Excl.</td>
<td>A2</td>
<td>40</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Memory

- **Remote Write**: In this state, the memory receives a write operation from a remote processor.
- **Write Back**: In this state, the memory updates its contents based on a write operation.
- **Invalid**: In this state, the memory indicates that the requested block is invalid.
- **Shared**: In this state, the memory indicates that multiple processors have access to the same block.
- **Exclusive**: In this state, the memory indicates that a single processor has access to the block.
- **CPU Read hit**: Indicates a successful read operation from the CPU.
- **CPU Write hit**: Indicates a successful write operation from the CPU.
- **Remote Read Miss**: Indicates a missed read operation from a remote processor.
- **CPU Write Miss**: Indicates a missed write operation from the CPU.

### Example: Step 1

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.

### Example: Step 2

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.

### Example: Step 3

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 != A2.
Example: Step 4

<table>
<thead>
<tr>
<th>Step</th>
<th>State</th>
<th>Addr</th>
<th>Value</th>
<th>Action</th>
<th>Proc.</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Start</td>
<td>A1</td>
<td>10</td>
<td>WrMs</td>
<td>P1</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Share</td>
<td>A1</td>
<td>10</td>
<td>RdMs</td>
<td>P2</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P1: Write 20 to A1</td>
<td>Share</td>
<td>A1</td>
<td>20</td>
<td>WrMs</td>
<td>P1</td>
<td>A1</td>
<td>20</td>
</tr>
<tr>
<td>P2: Write 40 to A2</td>
<td>Start</td>
<td>A2</td>
<td>40</td>
<td>WrMs</td>
<td>P2</td>
<td>A2</td>
<td>40</td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2.

Example: Step 5

<table>
<thead>
<tr>
<th>Step</th>
<th>State</th>
<th>Addr</th>
<th>Value</th>
<th>Action</th>
<th>Proc.</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1: Write 10 to A1</td>
<td>Start</td>
<td>A1</td>
<td>10</td>
<td>WrMs</td>
<td>P1</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P2: Read A1</td>
<td>Share</td>
<td>A1</td>
<td>10</td>
<td>RdMs</td>
<td>P2</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P1: Read A1</td>
<td>Share</td>
<td>A1</td>
<td>10</td>
<td>RdDa</td>
<td>P1</td>
<td>A1</td>
<td>10</td>
</tr>
<tr>
<td>P2: Write 20 to A2</td>
<td>Start</td>
<td>A2</td>
<td>20</td>
<td>WrMs</td>
<td>P2</td>
<td>A2</td>
<td>20</td>
</tr>
</tbody>
</table>

Assumes initial cache state is invalid and A1 and A2 map to same cache block, but A1 ≠ A2.

Implementation Complications

- **Write Races:**
  - Cannot update cache until bus is obtained
  - Otherwise, another processor may get bus first, and then write the same cache block!
  - Two step process:
    - Arbitrate for bus
    - Place miss on bus and complete operation
  - If miss occurs to block while waiting for bus, handle miss (invalidate may be needed) and then restart.
  - Split transaction bus:
    - Bus transaction is not atomic: can have multiple outstanding transactions for a block
    - Multiple misses can interleave, allowing two caches to grab block in the Exclusive state
    - Must track and prevent multiple misses for one block
- **Must support interventions and invalidations**

Implementing Snooping Caches

- Bus serializes writes, getting bus ensures no one else can perform memory operation
- On a miss in a write back cache, may have the desired copy and it's dirty, so must reply
- Add extra state bit to cache to determine shared or not
Larger MPs

- Separate Memory per Processor
- Local or Remote access via memory controller
- 1 Cache Coherency solution: non-cached pages
- Alternative: directory per cache that tracks state of every block in every cache
  - Which caches have a copies of block, dirty vs. clean, ...
- Info per memory block vs. per cache block?
  - PLUS: In memory => simpler protocol (centralized/done location)
  - MINUS: In memory => directory is \( f(\text{memory size}) \) vs. \( f(\text{cache size}) \)
- Prevent directory as bottleneck?
  distribute directory entries with memory, each keeping track of which Procs have copies of their blocks

Another MP Issue:
Memory Consistency Models

- What is consistency? When must a processor see the new value? e.g., seems that
  P1: \( A = 0; \) P2: \( B = 0; \)
  \( \ldots \) \( A = 1; \) \( B = 1; \)
  \( \ldots \) \( \) \( \) \( \) \( \) \( \)
  L1: if \( (B == 0) \) ... 
  L2: if \( (A == 0) \) ...
- Impossible for both if statements L1 & L2 to be true?
  - What if write invalidate is delayed & processor continues?
- Memory consistency models:
  what are the rules for such cases?
- Sequential consistency: result of any execution is the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above
  - SC: delay all memory accesses until all invalidates done

Memory Consistency Model

- Schemes faster execution to sequential consistency
- Not really an issue for most programs; they are synchronized
  - A program is synchronized if all access to shared data are ordered by synchronization operations
    - write (x)
    - ... 
    - release (s) (unlock)
    - ... 
    - acquire (s) (lock)
    - ... 
    - read(x)
- Only those programs willing to be nondeterministic are not synchronized: "data race": outcome \( f(\text{proc. speed}) \)
- Several Relaxed Models for Memory Consistency since most programs are synchronized; characterized by their attitude towards: RAR, WAR, RAW, WAW to different addresses

Summary

- Caches contain all information on state of cached memory blocks
- Snooping and Directory Protocols similar; bus makes snooping easier because of broadcast (snooping => uniform memory access)
- Directory has extra data structure to keep track of state of all cache blocks
- Distributing directory => scalable shared address multiprocessor => Cache coherent, Non uniform memory access
Cluster Computing - What is it?

- CS 235- Cluster and High performance Computing, Fall 2005 course

How to Run Applications Faster?

- There are 3 ways to improve performance:
  - Work Harder
  - Work Smarter
  - Get Help
- Computer Analogy
  - Using faster hardware
  - Optimized algorithms and techniques used to solve computational tasks
  - Multiple computers to solve a particular task

Scalable Parallel Computer Architectures

- Taxonomy
  - based on how processors, memory & interconnect are laid out
- Massively Parallel Processors (MPP)
- Symmetric Multiprocessors (SMP)
- Cache-Coherent Nonuniform Memory Access (CC-NUMA)
- Distributed Systems
- Clusters

Taxonomy of Architectures

- Simple classification by Flynn:
  (No. of instruction and data streams)
  - SISD - conventional
  - SIMD - data parallel, vector computing
  - MISD - systolic arrays
  - MIMD - very general, multiple approaches.
- Current focus is on MIMD model, using general purpose processors or multicomputers.
**Design Space of Competing Computer Architecture**

- **Size Scalability**
  - Distributed System
  - Present Cluster
  - MPP
  - CC-NUMA
  - SMP
  - Future Cluster
  - Single System Image

**Scalable Parallel Computer Architectures**

- **CC-NUMA**
  - A scalable multiprocessor system having a cache-coherent nonuniform memory access architecture
  - Every processor has a global view of all of the memory
- **Distributed systems**
  - Considered conventional networks of independent computers
  - Have multiple system images as each node runs its own OS
  - The individual machines could be combinations of MPPs, SMPs, clusters, and individual computers
- **Clusters**
  - A collection of workstations of PCs that are interconnected by a high-speed network
  - Work as an integrated collection of resources
  - Have a single system image spanning all its nodes

---

**Cluster Computer and its Architecture**

- A cluster is a type of parallel or distributed processing system, which consists of a collection of interconnected stand-alone computers cooperatively working together as a single, integrated computing resource.
- A node
  - A single or multiprocessor system with memory, I/O facilities, and OS
  - Generally 2 or more computers (nodes) connected together
  - In a single cabinet, or physically separated and connected via a LAN
  - Appear as a single system to users and applications
  - Provide a cost-effective way to gain features and benefits

**Cluster Computer Architecture**

- Parallel Applications
- Sequential Applications
- Cluster Middleware
- Single System Image and Availability Infrastructure
- High-Speed Network/Switch
Motivation for using Clusters

- Surveys show utilisation of CPU cycles of desktop workstations is typically <10%.
- Performance of workstations and PCs is rapidly improving
- As performance grows, percent utilisation will decrease even further!
- Organisations are reluctant to buy large supercomputers, due to the large expense and short useful life span.

Clustering Today

- Clustering gained momentum when 3 technologies converged:
  - 1. Very HP Microprocessors
    - workstation performance = yesterday supercomputers
  - 2. High speed communication
    - Comm. between cluster nodes >> between processors in an SMP.

Computing Power (HPC) Drivers

Solving grand challenge applications using computer *modeling, simulation* and

*Life Sciences, Aerospace, E-commerce/anything, CAD/CAM, Digital Biology, Military Applications*

Example Clusters: Berkeley NOW

- 100 Sun UltraSparcs
- 200 disks
- Myrinet SAN
- 160 MB/s
- Fast comm.
- AM, MPI, ...
- Ether/ATM switched external net
- Global OS
- Self Config
Cluster of SMPs (CLUMPS)

- Four Sun E5000s
  - 8 processors
  - 4 Myricom NICs each
- Multiprocessor, Multi-NIC, Multi-Protocol
- NPACI => Sun 450s

Prominent Components of Cluster Computers (I)

- Multiple High Performance Computers
  - PCs
  - Workstations
  - SMPs (CLUMPS)
  - Distributed HPC Systems leading to Metacomputing

Prominent Components of Cluster Computers (II)

- State of the art Operating Systems
  - Linux (Beowulf)
  - Microsoft NT (Illinois HPVM)
  - SUN Solaris (Berkeley NOW)
  - IBM AIX (IBM SP2)
  - HP UX (Illinois - PANDA)
  - Mach (Microkernel based OS) (CMU)
  - Cluster Operating Systems (Solaris MC, SCO Unixware, MOSIX (academic project))
  - OS glueing layers (Berkeley Glunix)

Prominent Components of Cluster Computers (III)

- High Performance Networks/Switches
  - Ethernet (10Mbps)
  - Fast Ethernet (100Mbps)
  - Gigabit Ethernet (1Gbps)
  - SCI (Dolphin - MPI-12micro-sec latency)
  - ATM
  - Myrinet (1.2Gbps)
  - Digital Memory Channel
  - FDDI
Prominent Components of Cluster Computers (IV)

- Network Interface Card
  - Myrinet has NIC
  - User-level access support

Prominent Components of Cluster Computers (V)

- Fast Communication Protocols and Services
  - Active Messages (Berkeley)
  - Fast Messages (Illinois)
  - U-net (Cornell)
  - XTP (Virginia)

Prominent Components of Cluster Computers (VI)

- Cluster Middleware
  - Single System Image (SSI)
  - System Availability (SA) Infrastructure
- Hardware
  - DEC Memory Channel, DSM (Alewife, DASH), SMP Techniques
- Operating System Kernel/Gluing Layers
  - Solaris MC, Unixware, GLUnix
- Applications and Subsystems
  - Runtime systems (software DSM, PFS etc.)
  - Resource management and scheduling software (RMS)
- SSI Concept is key to cluster concept!

Cluster Middleware & SSI

- SSI
  - Supported by a middleware layer that resides between the OS and user-level environment
  - Middleware consists of essentially 2 sublayers of SW infrastructure
    - SSI infrastructure
      - Glue together OSs on all nodes to offer unified access to system resources
      - System availability infrastructure
        - Enable cluster services such as checkpointing, automatic failover, recovery from failure, & fault-tolerant support among all nodes of the cluster
What is Single System Image (SSI)?

- A single system image is the illusion, created by software or hardware, that presents a collection of resources as one, more powerful resource.
- SSI makes the cluster appear like a single machine to the user, to applications, and to the network.
- A cluster without a SSI is not a cluster.

Single System Image Boundaries

- Every SSI has a boundary
- SSI support can exist at different levels within a system, one able to be build on another.

Prominent Components of Cluster Computers (VII)

- Parallel Programming Environments and Tools
  - Threads (PCs, SMPS, NOW...)
    - Posix Threads
    - MPI
    - POSIX Threads
    - Linux, NT, on many Supercomputers
    - PVM
    - Software DSMs (Shmem)
  - Compilers
    - C/C++/Java
    - Parallel programming with C++ (MIT Press book)
    - RAD (rapid application development tools)
    - GUI based tools for VP modeling
    - Debuggers
    - Performance Analysis Tools
    - Visualization Tools

Programming Environments and Tools (II)

- Message Passing Systems (MPI and PVM)
  - Allow efficient parallel programs to be written for distributed memory systems
  - 2 most popular high-level message-passing systems – PVM & MPI
  - MPI
    - Both an environment & a message-passing library
  - PVM
    - A message passing specification, designed to be standard for distributed memory parallel computing using explicit message passing
    - Attempt to establish a practical, portable, efficient, & flexible standard for message passing
    - Generally, application developers prefer MPI, as it is fast becoming the de facto standard for message passing.
Prominent Components of Cluster Computers (VIII)

- Applications
  - Sequential
  - Parallel / Distributed (Cluster-aware app.)
    - Grand Challenging applications
    - Weather Forecasting
    - Quantum Chemistry
    - Molecular Biology Modeling
    - Engineering Analysis (CAD/CAM)
  - PDBs, web servers, data-mining

Key Operational Benefits of Clustering

- High Performance
- Expandability and Scalability
- High Throughput
- High Availability

Definition: Embedded System

- Embedded system: any device that includes a programmable computer but is not itself a general-purpose computer.
- Take advantage of application characteristics to optimize the design:
  - don’t need all the general-purpose bells and whistles.

Defining Embedded Systems

- An embedded system may take on several definitions. Similarities amongst these definitions, however, may include:
  - Constrained system resources
  - Minimized human-machine interface (if at all)
  - Singly-focused application that runs when power is applied and terminates when power source is turned off or depleted.
Modern Embedded Systems

- Personal digital assistant (PDA).
- Consumer electronics (e.g. digital cameras and household appliances)
- Cell phone
- Automobile engines, fuel control, etc.
- Global Positioning System (GPS) units
- Printers
- Home automation systems
- Manufacturing plants – process control

Embedded Systems and Pervasive Computing?

- Embedded system
  - Driving technology
  - Individual components
- Pervasive Computing
  - Using embedded devices/systems to provide a pervasive computing environment
    - Networked, ubiquitous applications using embedded systems

Embedding a computer

Recognize This System?
### A little history

- Automobiles used microprocessor-based engine controllers starting in 1970’s.
  - Control fuel/air mixture, engine timing, etc.
  - Multiple modes of operation: warm-up, cruise, hill climbing, etc.
  - Provides lower emissions, better fuel efficiency.
- First microprocessor was Intel 4004 in early 1970’s
- What types of processors account for over 95% of all processors sold today?

### BMW 850i brake and stability control system

- Anti-lock brake system (ABS): pumps brakes to reduce skidding.
- Automatic stability control (ASC+T): controls engine to improve stability.
- ABS and ASC+T communicate.
  - ABS was introduced first—needed to interface to existing ABS module.

### BMW 850i, cont’d.

- Anti-lock brake system (ABS): pumps brakes to reduce skidding.
- Automatic stability control (ASC+T): controls engine to improve stability.
- ABS and ASC+T communicate.
  - ABS was introduced first—needed to interface to existing ABS module.

### Challenges in embedded system design

- How much hardware do we need?
  - How big is the CPU? Memory?
- How do we meet our deadlines?
  - Faster hardware or cleverer software?
- How do we minimize power?
  - Turn off unnecessary logic? Reduce memory accesses?
Software Development Process

- There are many ways to develop applications for embedded systems.
- Unlike general-purpose platform application development, you may not (generally) develop applications on the target embedded system platform.
- Typically, a general-purpose host computer is used in conjunction with a cross compiler/assembler to create the binaries for the target system.
- The resultant code is downloaded to the target system and tested.

Program design and analysis for Embedded Systems

- Optimizing for execution time.
- Optimizing for energy/power.
- Optimizing for program size.

Issues in Compiling for Embedded Systems

- Code Generation for Specialized Architecture
- Code size
  - Using techniques for performance optimizations, such as loop unrolling etc., can increase code size
- Timing requirements
  - To get precise timing may need to use assembly language
  - Not always "fast as you can" – may need minimum time, and duration
- Energy/Power optimization
  - Can we control power through software??
  - Instruction power – use of low power instructions
  - Dynamic voltage scaling
    - Memory power optimization
      - Placement of data
      - Dynamic power control of memory modules
- For i=1 to N A[i]=A[i]*2; For j=1 to N B[i]=B[i]+10;
CS 211: Summary

- Overview of Computer Architecture Design and Analysis
  - Designing for Performance
  - ILP concepts
  - Memory design
- Hardware/software interface and role of s/w techniques in processor performance
- Final Exam; May 5th, 6:40pm-8:40pm
  - Focus on materials after midterm BUT cumulative
  - Similar format to midterm
  - Programming project 3 due May 12th

Implementing Snooping Caches

- Multiple processors must be on bus, access to both addresses and data
- Add a few new commands to perform coherency, in addition to read and write
- Processors continuously snoop on address bus
  - If address matches tag, either invalidate or update
- Since every bus transaction checks cache tags, could interfere with CPU cache access:
  - solution 1: duplicate set of tags for L1 caches just to allow checks in parallel with CPU
  - solution 2: L2 cache already duplicate, provided L2 obeys inclusion with L1 cache
    - block size, associativity of L2 affects L1

Distributed Directory MPs

Directory Protocol

- Similar to Snoopy Protocol: Three states
  - Shared: ≥ 1 processors have data, memory up-to-date
  - Uncached (no processor has it; not valid in any cache)
  - Exclusive: 1 processor (owner) has data; memory out-of-date
- In addition to cache state, must track which processors have data when in the shared state (usually bit vector, 1 if processor has copy)
- Keep it simple(r):
  - Writes to non-exclusive data
    - write miss
  - Processor blocks until access completes
  - Assume messages received and acted upon in order sent
Directory Protocol

- No bus and don’t want to broadcast:
  - interconnect no longer single arbitration point
  - all messages have explicit responses
- Terms: typically 3 processors involved
  - Local node where a request originates
  - Home node where the memory location of an address resides
  - Remote node has a copy of a cache block, whether exclusive or shared
- Example messages on next slide:
  P = processor number, A = address