Non-Uniform Memory Access (NUMA) Architecture and Multicomputers

Parallel and Distributed Computing

Department of Computer Science and Engineering (DEI)
Instituto Superior Técnico

October 2, 2012
Outline

- Distributed memory systems
  - Non-Uniform Memory Access (NUMA) architecture
    - Memory coherence and consistency
  - Multicomputers
    - Message Passing

- Network topologies

- Latency reduction
Uniform Memory Access (UMA) architecture
also known as
Symmetric Shared-Memory Multiprocessors (SMP)

Limited scalability!
Non-Uniform Memory Access (NUMA) architecture

or

Multicomputers

Larger access time to remote data.
Distributed-Memory Systems

Two programming models for distributed memory systems:

**Non-Uniform Memory Access (NUMA) architecture**
Memories are physically separated, but can be addressed as one logically shared addressed space (also known as Distributed Shared Memory, DSM).

**Multicomputers**
Each processor has its own private address space, not accessible by the other processors. Data sharing requires explicit messages among processors (also known as Message Passing Multiprocessors).
all processors share the same address space
⇒ a given memory address returns the same data for any processor

data always accessed through load/store instructions, wherever it may reside

access time is highly dependent on whether data is in local or remote memory (NUMA)
Distributed shared memory can be implemented through a distributed virtual memory scheme:

- logic address space is divided into pages
- page table keeps state of each page (similar to a directory)
  - not-present: pages in remote memories
  - shared: page available in local memory, but more copies exist
  - exclusive: page only available in local memory
- access to one of these pages causes a page-fault → page request sent to remote processor
NUMA Memory Coherence

Memory coherence:

- a read of $M[X]$ by processor $P$ that follows a write by $P$ to $M[X]$, and with no write to $M[X]$ by another processor in between, always returns the value written by $P$.

- a read of $M[X]$ by $P_i$ that follows a write by of $M[X]$ by $P_j$ always returns the value written by $P_j$ if the read and write are sufficiently separated in time and no other writes to $M[X]$ occur between the two accesses.

- write to the same location are serialized; two writes to the same location by any two processors are seen in the same order by all processors.

No single bus to serialize requests!

**Synchronous write**: block processor until confirmation of write operation.
Multicomputers - Message Passing

- Each processor has its own private resources and address space, not accessible by the other processors, hence acts like an independent computer.

- Data sharing requires explicit message requests through the interconnect network.

Asymmetrical multicompiler: one processor (front-end) is the access point to all processors (back-end) and manages the distribution of the workload.

Symmetrical multicompiler: every processor at the same level.
NUMA vs Multicomputers

Advantages of NUMA (DSM):

- programming is simpler
- no compiler libraries are necessary
Advantages of NUMA (DSM):

- programming is simpler
- no compiler libraries are necessary

Advantages of Multicomputers (Message Passing):

- simpler hardware
- explicit communication
- possible to emulate DSM
Network of Workstations (NOW): computers’ primary purpose is not parallel computing; use free CPU power; potential diversity of architectures and operating systems
Types of Distributed Memory Systems

Network of Workstations (NOW): computers’ primary purpose is not parallel computing; use free CPU power; potential diversity of architectures and operating systems

Cluster: co-located collection of COTS computers and switches dedicated to run parallel jobs
Types of Distributed Memory Systems

Network of Workstations (NOW): computers’ primary purpose is not parallel computing; use free CPU power; potential diversity of architectures and operating systems

Cluster: co-located collection of COTS computers and switches dedicated to run parallel jobs

Grid: large collection of widely distributed clusters and NOWs
Types of Distributed Memory Systems

Network of Workstations (NOW): computers’ primary purpose is not parallel computing; use free CPU power; potential diversity of architectures and operating systems

Cluster: co-located collection of COTS computers and switches dedicated to run parallel jobs

Grid: large collection of widely distributed clusters and NOWs

Commercial: costume hardware / network to provide a good balance between processing and communication speed
Packet switching: messages are sent one at a time over a shared medium that connects to all processors. Typically each message is divided into packets.
Interconnection Networks

Packet switching: messages are sent one at a time over a shared medium that connects to all processors. Typically each message is divided into packets.

Circuit switching: support point-to-point connection between pairs of processors.
Circuit Switching Networks

Metrics for circuit switching network topology evaluation:

**Bisection bandwidth**: bandwidth available between halves of the network

**Diameter**: largest number of switches in the path between any two nodes

**Cost**: required hardware and wires

**Scalability**: how the above parameters grow with the number of processors
Circuit Switching Networks

Bus
Circuit Switching Networks

Bus

Ring
Circuit Switching Networks

Bus

Ring

2D Grid / Mesh
Circuit Switching Networks

Bus

Ring

2D Grid / Mesh

2D Torus
Circuit Switching Networks

More sophisticated topologies:

4D Hypercube
Circuit Switching Networks

More sophisticated topologies:

4D Hypercube

Tree / Fat Tree
Butterfly
Circuit Switching Networks

For a network with $n$ processors:

<table>
<thead>
<tr>
<th>Type</th>
<th>Switches</th>
<th>Diameter</th>
<th>Bisection</th>
<th>Ports / switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D Mesh</td>
<td>$n$</td>
<td>$2(\sqrt{n} - 1)$</td>
<td>$\sqrt{n}$</td>
<td>4</td>
</tr>
<tr>
<td>Tree</td>
<td>$2n - 1$</td>
<td>$2 \log n$</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Hypercube</td>
<td>$n$</td>
<td>$\log n$</td>
<td>$n/2$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>Butterfly</td>
<td>$n(\log n + 1)$</td>
<td>$\log n$</td>
<td>$n$</td>
<td>4</td>
</tr>
<tr>
<td>one2one</td>
<td>$n^2$</td>
<td>1</td>
<td>$2^{n/2}$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
Memory coherence rules are a guarantee for the correct behavior of parallel applications.

Particularly relevant for:
- transactional systems
- distributed processing
Latency Reduction Techniques

Memory coherence rules are a guarantee for the correct behavior of parallel applications.

Particularly relevant for:
- transactional systems
- distributed processing

When developing parallel applications, the programmer can in some cases disregard these rules to achieve better performance
⇒ asynchronous message send / receive
Asynchronous Send

Synchronous send / receive

Task A
- computation
- wait
- sending
- computation

Task B
- computation
- receiving
- computation

send_sync()

receive_sync()
Asynchronous Send

Synchronous send / receive

Task A
- computation
- wait
- sending
- computation

Task B
- computation
- receiving
- computation

Asynchronous send (write): write to a temporary buffer

Task A
- computation
- buffer
- computation

Task B
- computation
- receiving
- computation
Asynchronous Receive

Synchronous send / receive

send_sync()

Task A

computation | wait | sending | computation

Task B

computation | receiving | computation

receive_sync()
Asynchronous Receive

Synchronous send / receive

Task A
- computation
- wait
- sending
- computation

Task B
- computation
- receiving
- computation

Asynchronous receive (read): execute command before data is needed

Task A
- computation
- sending
- computation

Task B
- computation
- computation

send_sync()
receive_sync()
Review

- Distributed memory systems
  - Non-Uniform Memory Access (NUMA) architecture
    - Memory coherence and consistency
  - Multicomputers
    - Message Passing

- Network topologies

- Latency reduction
Next Class

- Dependency Graphs
- Parallel Programming Methodology