Objectives of the Course

• Parallel Systems:
  – Understanding the current state-of-the-art in parallel programming technology
  – Getting familiar with existing algorithms for number of application areas

• Distributed Systems:
  – Understanding the principals of distributed programming
  – Learning how to use sockets and RMI technology in JAVA

• Completion of a research paper
Parallel Architecture: motivation

• The Von Newman model
  – Bottlenecks:
    • The CPU-memory bottleneck
    • The CPU execution rate

• Improvements to the basic model
  – Memory interleaving and caching
  – Instruction/execution pipelining
Memory Interleaving

To speed up the memory operations (read and write), the main memory of \( 2^n = N \) words can be organized as a set of \( 2^m = M \) independent memory modules (where \( m < n \)) each containing \( 2^{n-m} \) words. If these M modules can work in parallel (or in a pipeline fashion), then ideally an M fold speed improvement can be expected. The n-bit address is divided into an m-bit field to specify the module, and another (n-m)-bit field to specify the word in the addressed module. The field for specifying the modules can be either the most or least significant m bits of the address.

For example, these are the two arrangements of \( M = 2^m = 2^2 = 4 \) modules (\( m = 2 \)) of a memory of \( 2^n = 2^4 = 16 \) words (\( n = 4 \)):

**High-order arrangement**

<table>
<thead>
<tr>
<th></th>
<th>M0</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>00</td>
<td>10</td>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>00</td>
<td>11</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

**Low-order arrangement**

<table>
<thead>
<tr>
<th></th>
<th>M0</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
<td>00</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>00</td>
<td>01</td>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>00</td>
<td>10</td>
<td>00</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>00</td>
<td>11</td>
<td>00</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>01</td>
<td>00</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>5</td>
<td>01</td>
<td>01</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td>00</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
<td>00</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>01</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
In general, the CPU is more likely to need to access the memory for a set of consecutive words (either a segment of consecutive instructions in a program or the components of a data structure such as an array, the interleaved (low-order) arrangement is preferable as consecutive words are in different modules and can be fetched simultaneously. In case of high-order arrangement, the consecutive words are usually in one module, having multiple modules is not helpful if consecutive words are needed.

**Example:** A memory of $2^{16}$ words (n=16) with $2^{4} = 16$ modules (m=4) each containing $2^{n-m} = 2^{12} = 4K$ words:

![Memory Interleaving Diagram](image-url)
Figure 2.1  The evolution of a typical sequential computer: (a) a simple sequential computer; (b) a sequential computer with memory interleaving; (c) a sequential computer with memory interleaving and cache; and (d) a pipelined processor with $d$ stages.

Copyright (r) 1994 Benjamin/Cummings Publishing Co.
Figure 2.1 Example of a two-way superscalar execution of instructions.

(a) Three different code fragments for adding a list of four numbers.

(b) Execution schedule for code fragment (i) above.

(c) Hardware utilization trace for schedule in (b).
Logical and Physical Organization

- The two fundamental aspects of parallel computing from a programmer perspective:
  - ways of expressing parallel tasks (*control structure*)
    - MIMD, SIMD (Single/Multiple Instruction, Multiple Data)
  - Mechanisms for specifying task-to-task interaction (*communication model*)
    - Main classification: message passing vs. shared memory

- The physical organization of a machine is often (but not necessarily) related to the logical view
  - Good performance requires good matching between the two views
The von Neumann model is also called *Single Instruction stream – Single Data stream* (SISD)

Bottleneck are CPU rate and CPU-memory ➔ Multiply CPUs (MIMD, SPMD) or just the PEs (SIMD) and related memory

- SIMD model: same instruction executed synchronously by all execution units on different data
- MIMD(and SPMD) model: each processor is capable of executing its own program
SIMD vs. MIMD

• SIMD: a single global *control unit* multiple PE
• MIMD: multiple, full blown processors
• Examples
  – SIMD: Illiac IV, CM-2, MasPar MP-1 and MP-2
  – MIMD: CM-5, paragon
  – SPMD: Origin 2000, Cary T3E, Clusters

Figure 2.2  A typical SIMD architecture (a) and a typical MIMD architecture (b). Copyright (c) 1994 Benjamin/Cummings Publishing Co.
SIMD vs. MIMD (II)

• In general MIMD is more flexible,
• SIMD pros:
  – Requires less hardware: single control unit
  – Faster communication: single clock means synchronous operation, transfer of data is very much like a register transfer
• SIMD cons
  – Best suited for data-parallel programs
  – Different nodes cannot execute different instructions in same clock cycle – conditional statement examples!
Figure 2.4  Executing a conditional statement on an SIMD computer with four processors: (a) the conditional statement; (b) the execution of the statement in two steps.
A Different Taxonomy

- SISD, SIMD, MIMD refer mainly to the processor organization
- With respect to the memory organization, the two fundamental models are:
  - Distributed memory architecture
    - Each processor has its own private memory
  - Shared address space architecture
    - Processors have access to the same address space
Memory Organizations (II)

- Shared-address-space computers can have a local memory to speed access to non-shared data
  - Figure (b) and (c) in previous slide
  - So called *Non Uniform Memory Access* (NUMA) as opposed to *Uniform Memory Access* (UMA) has different access times depending on location of data
- To alleviate speed difference, local memory can also be used to cache frequently used shared data
  - Use of cache introduces the issue of cache coherence,
  - In some architectures local memory is entirely used as cache – so called *cache-only memory access* (COMA)