Classification of Parallel Architectures

Flynn’s Taxonomy
- SISD: Single instruction single data
  - Classical von Neumann architecture
- SIMD: Single instruction multiple data
- MISD: Multiple instructions single data
  - Non existent, just listed for completeness
- MIMD: Multiple instructions multiple data
  - Most common and general parallel machine
Single Instruction Multiple Data

- Also known as Array-processors
- A single instruction stream is broadcasted to multiple processors, each having its own data stream
  - Still used in some graphics cards today

![Diagram of Single Instruction Multiple Data]

Multiple Instructions Multiple Data (I)

- Each processor has its own instruction stream and input data
- Very general case
  - every other scenario can be mapped to MIMD
- Further breakdown of MIMD usually based on the memory organization
  - Shared memory systems
  - Distributed memory systems
Shared memory systems (I)

- All processes have access to the same address space
  - E.g. PC with more than one processor
- Data exchange between processes by writing/reading shared variables
  - Shared memory systems are easy to program
  - Current standard in scientific programming: OpenMP
- Two versions of shared memory systems available today
  - Symmetric multiprocessors (SMP)
  - Non-uniform memory access (NUMA) architectures

Symmetric multi-processors (SMPs)

- All processors share the same physical main memory

- Memory bandwidth per processor is limiting factor for this type of architecture
- Typical size: 2-32 processors
NUMA architectures (I)

- Some memory is closer to a certain processor than other memory
  - The whole memory is still addressable from all processors
  - Depending on what data item a processor retrieves, the access time might vary strongly

NUMA architectures (II)

- Reduces the memory bottleneck compared to SMPs
- More difficult to program efficiently
  - E.g. first touch policy: data item will be located in the memory of the processor which uses a data item first
- To reduce effects of non-uniform memory access, caches are often used
  - ccNUMA: cache-coherent non-uniform memory access architectures
- Largest example as of today: SGI Origin with 512 processors
Distributed memory machines (I)

- Each processor has its own address space
- Communication between processes by explicit data exchange
  - Sockets
  - Message passing
  - Remote procedure call / remote method invocation

Distributed memory machines (II)

- Performance of a distributed memory machine strongly depends on the quality of the network interconnect and the topology of the network interconnect
  - Of-the-shelf technology: e.g. fast-Ethernet, gigabit-Ethernet
  - Specialized interconnects: Myrinet, Infiniband, Quadrics, 10G Ethernet ...
Distributed memory machines (III)

- Two classes of distributed memory machines:
  - Massively parallel processing systems (MPPs)
    - Tightly coupled environment
    - Single system image (specialized OS)
  - Clusters
    - Of-the-shelf hardware and software components such as
      - Intel P4, AMD Opteron etc.
      - Standard operating systems such as LINUX, Windows, BSD UNIX

MIMD (III)

- Important metrics:
  - Latency:
    - minimal time to send a very short message from one processor to another
    - Unit: ms, µs
  - Bandwidth:
    - amount of data which can be transferred from one processor to another in a certain time frame
    - Units: Bytes/sec, KB/s, MB/s, GB/s
      Bits/sec, Kb/s, Mb/s, Gb/s, baud
Hybrid systems

• E.g. clusters of multi-processor nodes

Shared memory vs. distributed memory machines

Shared memory machines:
• Compiler directives (e.g. Open MP)
• Threads (e.g. POSIX Threads)

Distributed memory machines:
• Message Passing (e.g. MPI, PVM)
• Distributed Shared Memory (e.g. UPC, CoArrayFortran)
• Remote Procedure Calls (RPC/RMI)

Message passing widely used in (scientific) parallel programming
• price/performance ratio of ‘message passing hardware’
• very general concept
Performance Metrics (I)

- **Speedup**: how much faster does a problem run on \( p \) processors compared to 1 processor?
  \[
  S(p) = \frac{T_{\text{total}}(1)}{T_{\text{total}}(p)}
  \]
  - Optimal: \( S(p) = p \) (linear speedup)

- **Parallel Efficiency**: Speedup normalized by the number of processors
  \[
  E(p) = \frac{S(p)}{p}
  \]
  - Optimal: \( E(p) = 1.0 \)

Performance Metrics (II)

- Example: Application A takes 35 min. on a single processor, 27 on two processors and 18 on 4 processors.

\[
S(2) = \frac{35}{27} = 1.29 \\
S(4) = \frac{35}{18} = 1.94
\]

\[
E(2) = \frac{1.29}{2} = 0.645 \\
E(4) = \frac{1.94}{4} = 0.485
\]
Amdahl’s Law (I)

- Most applications have a (small) sequential fraction, which limits the speedup

\[ T_{total} = T_{sequential} + T_{parallel} = fT_{Total} + (1 - f)T_{Total} \]

\( f \): fraction of the code which can only be executed sequentially

\[ S(p) = \frac{T_{total}(1)}{(f + \frac{1-f}{p})T_{total}(1)} = \frac{1}{f + \frac{1-f}{p}} \]

Example for Amdahl’s Law
Amdahl’s Law (II)

- Amdahl’s Law assumes, that the problem size is constant
- In most applications, the sequential part is independent of the problem size, while the part which can be executed in parallel is not.

Performance Metrics (III)

- **Scaleup**: ratio of the execution time of a problem of size $n$ on 1 processor to the execution time of the same problem of size $n*p$ on $p$ processors

\[ S_r(p) = \frac{T_{\text{total}}(1,n)}{T_{\text{total}}(p,n*p)} \]

- Optimally, execution time remains constant, e.g.

\[ T_{\text{total}}(p,n) = T_{\text{total}}(2p,2n) \]
Cache Coherence

- A more detailed look at shared memory systems leads to
  - Caches between memory and CPU
- Question: What happens if both caches hold a copy of the same data item and
  - CPU 1 modifies the data and CPU 2 tries to read it at the same time?
    - Don’t want a CPU to read ‘old’ values
  - Both CPU’s try to write the same item?
    - Don’t want both CPU’s to have a different view
    - Requires write serialization

Cache Coherence Protocols

- Two basic schemes:
  - Directory based: the status (shared/not shared) of a cache block is stored in a memory location
  - Snooping: each cache block has a copy of the sharing status of the block.
    - Caches are kept on a bus
    - Caches ‘snoop’ the bus in order to determine whether a copy of block has been modified
    - Dominating for small number of cores in today’s PC’s
Cache Coherence Protocols (II)

- Two forms of implementing cache coherence using bus snooping
  - **Write invalidate**: a write operation by a CPU invalidates the cache blocks of the same data items
    - Other CPU has to re-fetch the data item
  - **Write update**: a write operation by a CPU broadcasts the value on the bus to all other caches
    - Have to track which data items are held by other caches in order to minimize data transfer

Write invalidate vs. Write update

- Multiple writes to the same word (without any read operation in-between) requires multiple broadcasts in the write update protocol, but only one invalidate operation in the write invalidate protocol
- Multi-word write requires one invalidate operation per cache block for the write-invalidate protocol, requires however the broadcast of every word in the write-update protocol
- A read after write by different processes might be more efficient by the write update protocol

→ because of bus and memory bandwidth issues, write invalidate is the dominant protocol
Cache Coherence Protocols (III)

- A bus connects the different caches
- In order to perform an invalidate operation, processor needs to acquire bus access
  - Automatically serializes multiple writes
  - All writes are complete (e.g. it is not possible, that the first part of a multi-word write contains data from Proc. A and the second part from Proc. B if both write exactly the same memory locations)
- Extra bit on the cache tag indicates whether a cache block is shared
  - A write operation changes the bit from shared to exclusive