SPM: Parallel Architectures

Academic year 2024-2025

Prof. Massimo Torquati
massimo.torquati@unipi.it

Outline

- Classification of parallel architectures
  - Flynn’s Taxonomy
  - MIMD architectures
  - Shared Memory (SHM) vs. Distributed Memory (DM) systems
  - Programming considerations for SHM and DM systems

Why Parallel Architectures?

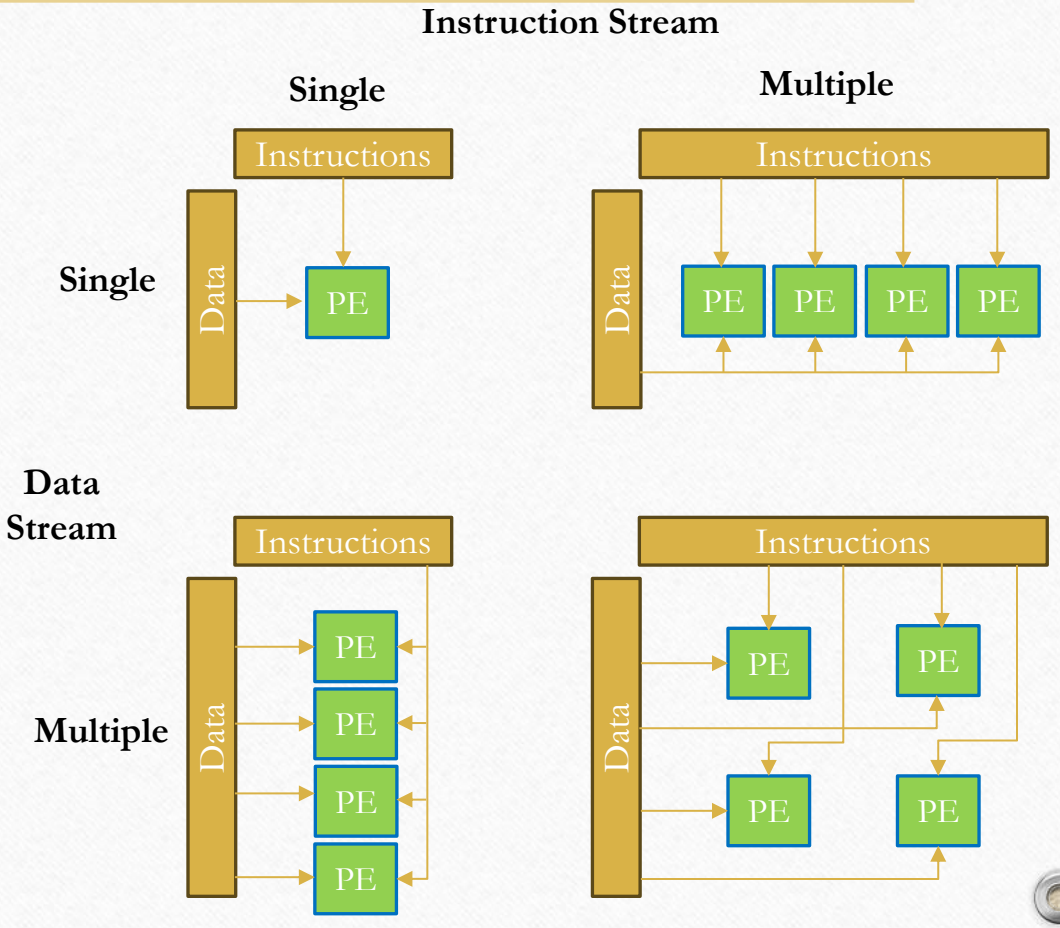
- Increased performance demand to solve challenging problems in scientific and engineering applications
- **Scalability**: to enable scale-out by adding more nodes to the system
- **Fault tolerance**: to ensure reliability, if one or more nodes fail, other nodes can still be used
- Parallelism applies at all levels of system design
  - Microarchitectures, memory systems, interconnections, I/O, ….
- Essential for modern computing:
  - Parallel architectures play an important role in the evolution of computer systems
  - Today' supercomputers often rely on off-the-shelf components (CPU and GPUs). However, the overall architecture goes beyond simply assembling these components. Interconnections, cooling, and SW stack (including management) are carefully customized for maximum performance.

Classifying Parallel Architectures

- Two significant aspects characterize parallel architectures
  - The system architecture of the single-node
  - The interconnection network connecting multiple nodes forming the parallel system
- No single classification fully captures the diversity of parallel architectures, as systems often share overlapping features.
- Common classification criteria:
  - Flynn's taxonomy: well-recognized and straightforward, but nowadays not so representative, yet instructive
  - Memory organization: shared-memory, distributed-memory, hybrid systems
  - Core count & Processing model: CMP, multiprocessors, distributed systems, supercomputers
  - Interconnection networks: Evaluate communication among PEs and memory modules

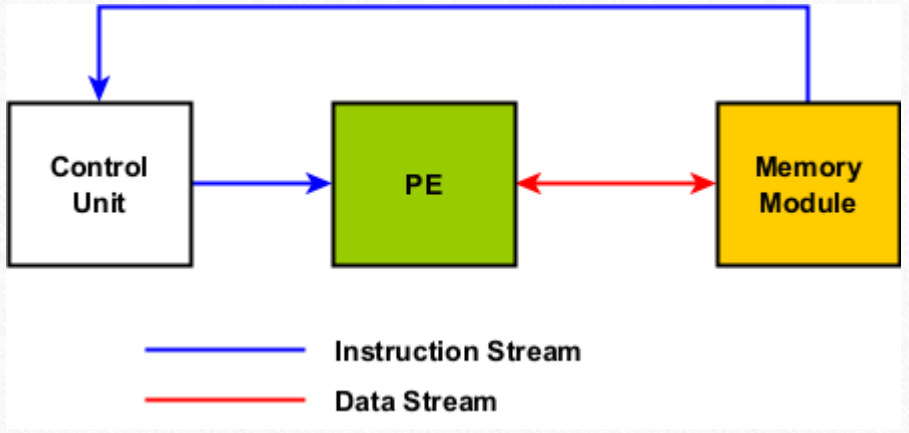
Flynn's Taxonomy (1966)

- Classification based on the number of **instructions and data streams**
  - **SISD** (*Single Instruction, Single Data*) refers to the traditional von Neumann architecture where a single sequential processing element (PE) operates on a single stream of data
  - **SIMD** (Single Instruction, Multiple Data) executes the same operation on multiple data items simultaneously
  - **MISD** (*Multiple Instruction, Single Data*) employs multiple PEs to execute different instructions on a single stream of data (rarely used)
  - **MIMD** (*Multiple Instruction, Multiple Data*) uses multiple PEs to execute different instructions on different data streams



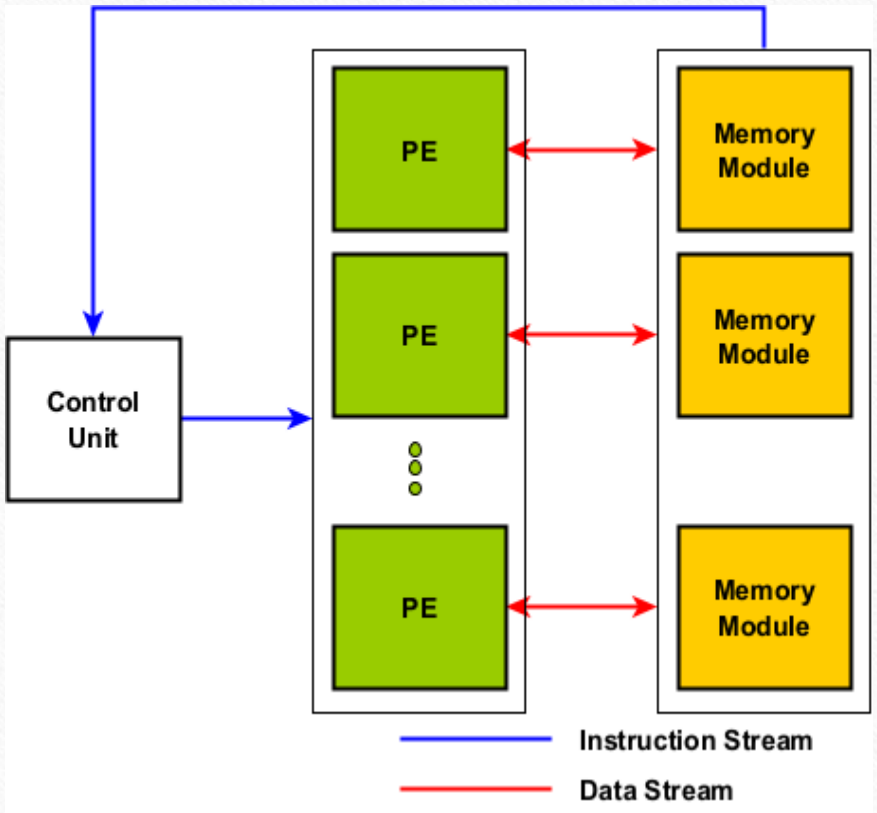
Single Instruction, Single Data Stream (SISD)

- Serial (non-parallel) computer
- Single stream of instructions executed serially
  - Single PE, data loaded/stored from/to a single memory
- The processor executes a single instruction at a time operating on data stored in a single memory
- Example: Uniprocessor/unicore machine
  - One core of a CMP machine can be considered a SISD-like machine



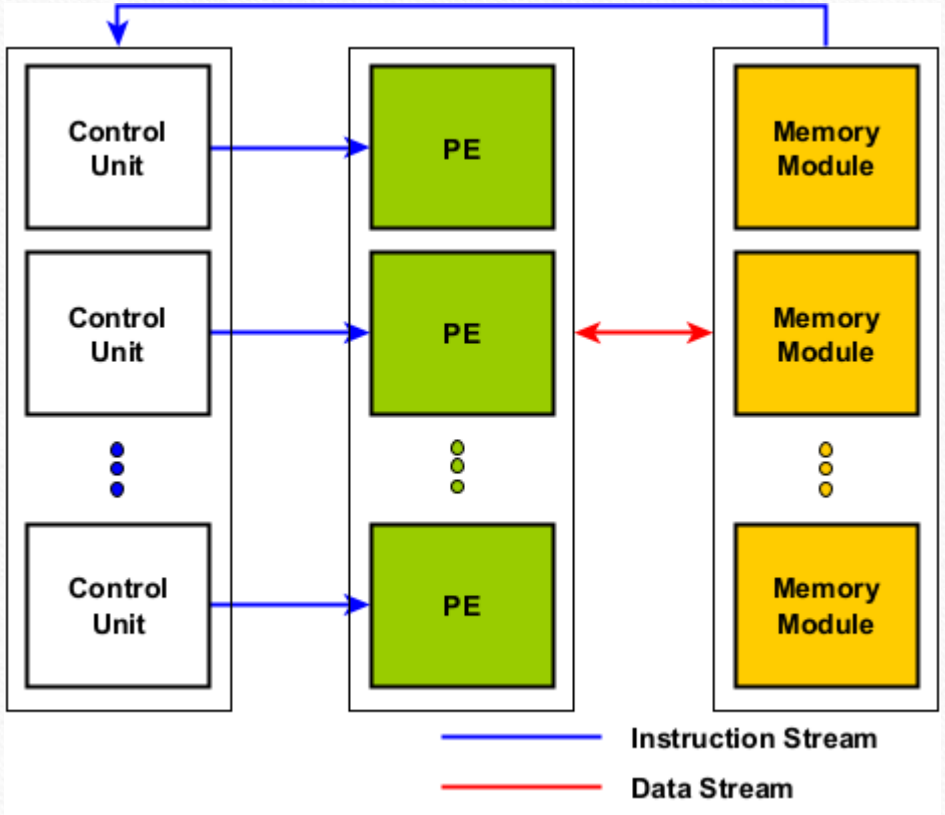
Single Instruction, Multiple Data Stream (SIMD)

- All PEs/CPUs execute the same instruction at any given clock cycle on a different set of data in parallel
  - **SIMD systems relate to data parallelism**
- Each CPU operates on different data streams, and usually each CPU has an associated data memory module
- The execution is synchronous in **locksteps**
- Examples:
  - Vector units of modern pipeline/superscalar CPUs
  - GPUs: they implement SIMD execution within Streaming Multiprocessors. However, at a broader architectural level, they follow the SIMT model, which generalizes SIMD by supporting thread divergence and independent execution paths



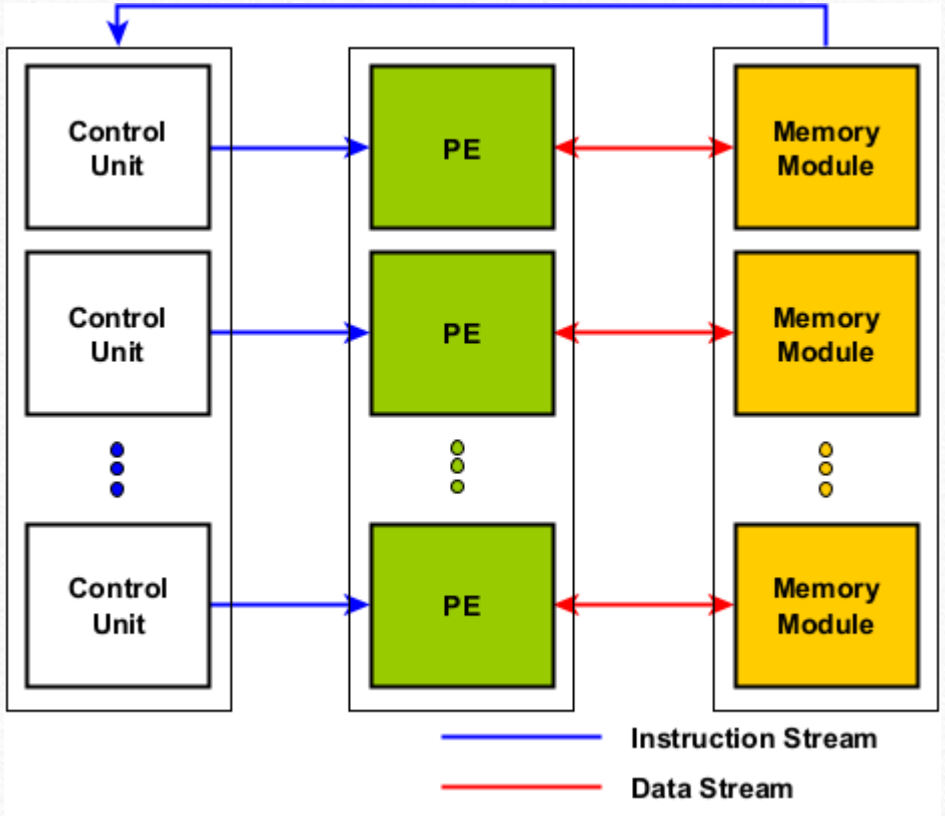
Multiple Instruction, Single Data Stream (MISD)

- All PEs/CPUs execute a **different instruction sequence** on a single data stream
- This type of system is not general-purpose and not commercially implemented
  - One application of this model is in mission-critical systems for fault tolerance reasons
    - The same data is processed by multiple machines, and the decision is taken considering (for example) the majority principle.
    - Different algorithms are run on distinct processors using the same input data



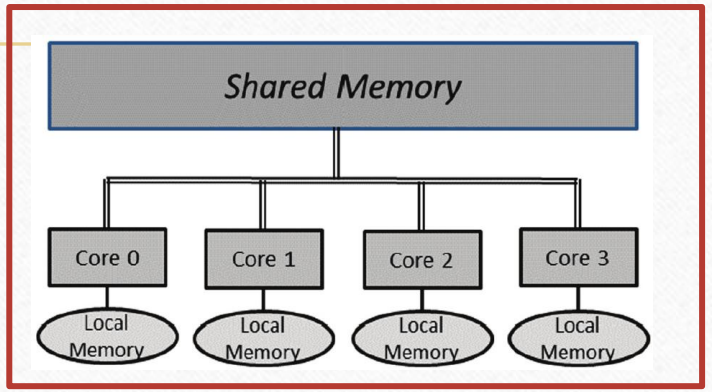
Multiple Instruction Stream, Multiple Data Stream (MIMD)

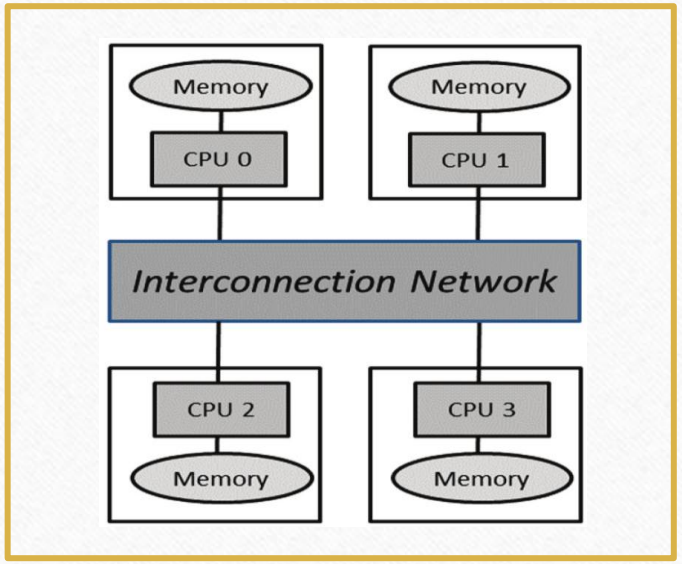
- A set of PEs simultaneously execute different instruction sequences on different data streams
- MIMD architectures are the most commonly used parallel architectures
- Each processor can execute all instructions
- Can be further classified based on memory organization and interconnection topologies used
- Examples:
  - Standard off-the-shelf CMPs (Intel Xeon, AMD EPYC, IBM Power 10)
  - Clusters and most supercomputers



Classification based on the memory system

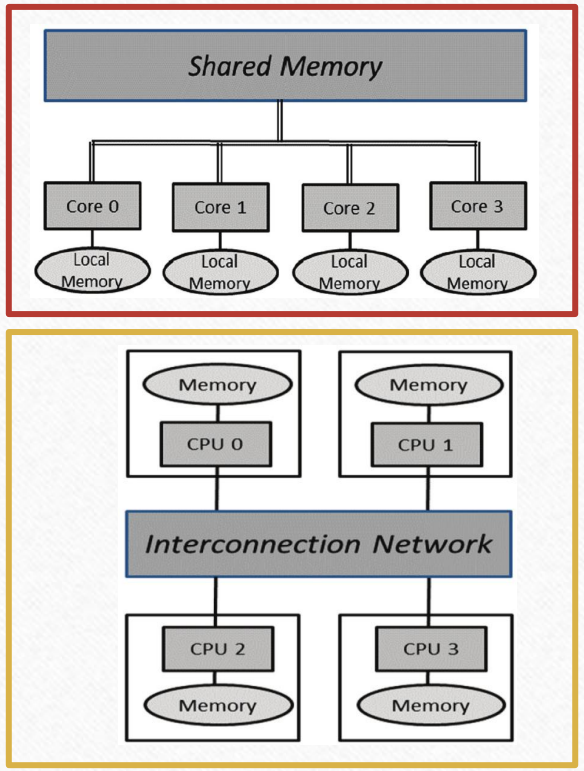
- Here we implicitly refer to MIMD parallel architectures, i.e., the most general ones
- Considering the **memory system**, we have
  - **Shared-Memory** architectures
  - **Distributed-Memory** architectures
- Shared Memory architectures are also known as **multiprocessors**
- Distributed Memory architectures are also known as **multicomputers**





Classification based on the memory system

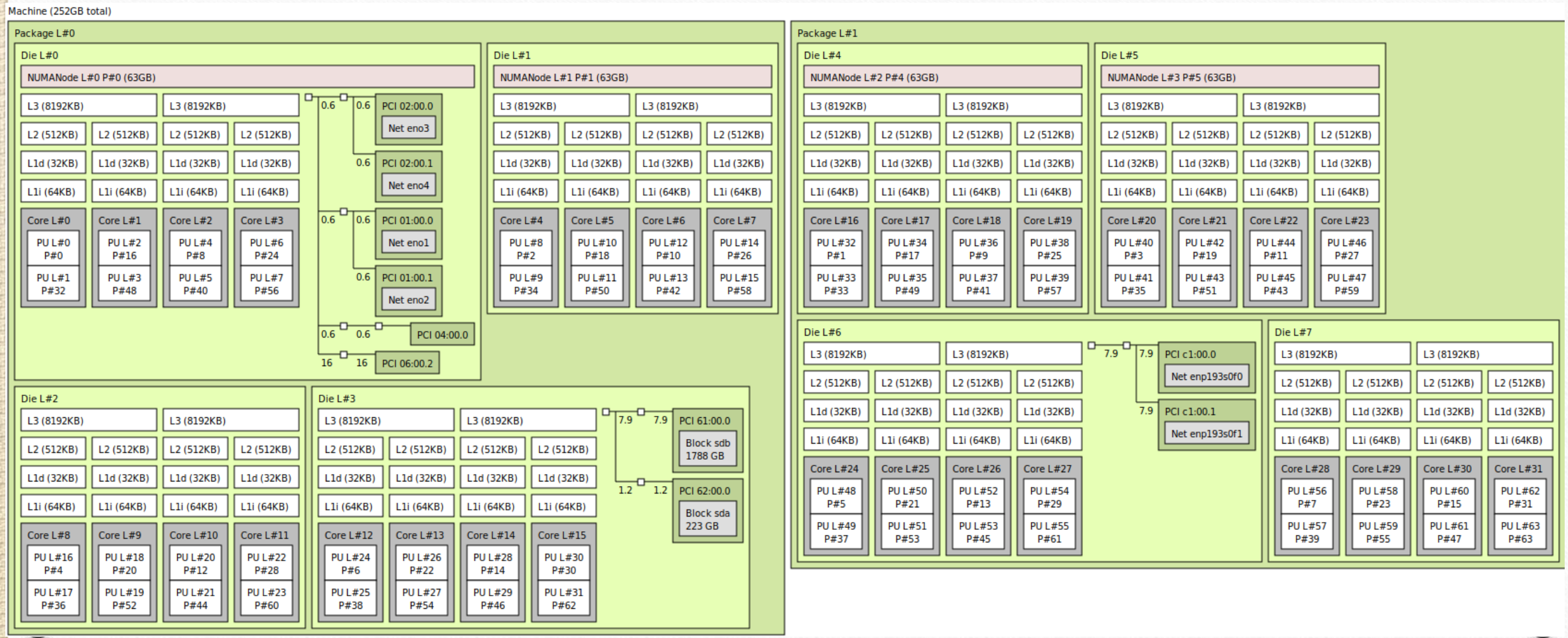
- **SHared-Memory (SHM)** systems can be classified as
  - **uniform** (SMP – Symmetric Multiprocessor)
    - All processors/cores have (roughly) equal access time to memory
    - This organization is called Uniform Memory Access (UMA)
    - Example: Single socket Intel Xeon-based system, where all cores share one memory controller
  - **non-uniform** (NUMA – Non-Uniform Memory Access)
    - Each processor/core (or group of cores) has its local memory
    - Non-uniform memory access time, i.e., the memory access time is asymmetric depending on which memory is accessed
    - In **NUMA multiprocessors**, the memory is still shared among PEs, but it is physically distributed
    - Example: AMD EPYC chips, which have a NUMA organization even in a single-socket configuration (chip composed of multiple chiplets)



NUMA multicores

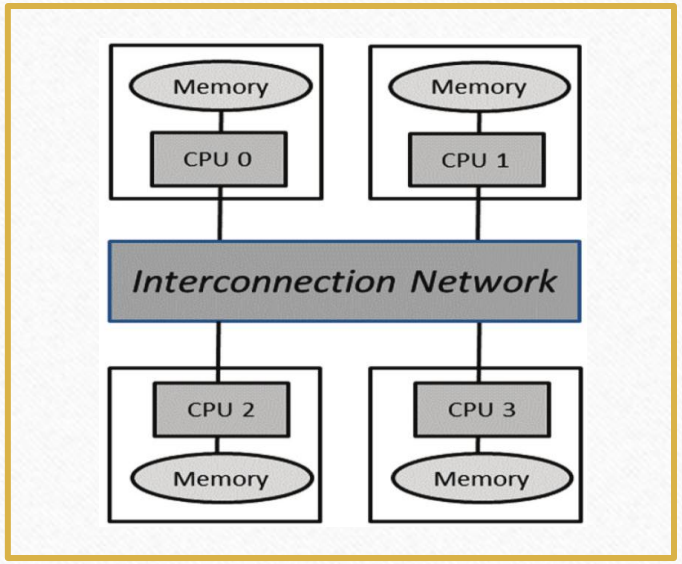
- Multi-socket server architectures and also single-socket design that use multiple dies (also called chiplets)
  - A NUMA node corresponds to an entire CPU (socket) or a chiplet with its local memory and set of cores
  - Each socket (or chiplet) has its memory controller and local memory
  - To maintain a single, shared address space, the nodes are connected by a high-speed Network on Chip (NoC)
  - The OS, tries to allocate memory «close to» where the allocating thread is running to reduce memory access costs
  - Each socket (or chiplet) has a shared Last Level Cache (LLC), usually L3

NUMA multicore example: spmnuma



Classification based on the memory system

- **Distributed Memory (DM)** systems are inherently NUMA
  - Each processor has its private memory; it can be an SPM or a NUMA multiprocessor
  - The address space of distinct nodes is disjoint
    - However, ccNUMA and COMA-like multicomputers were also built in the past as both commercial products and research prototypes
      - Examples: SGI Origin 2000 series and MIT Alewife
  - Processors communicate via explicit messages through the network
  - Examples: Clusters, supercomputers, edge and cloud nodes

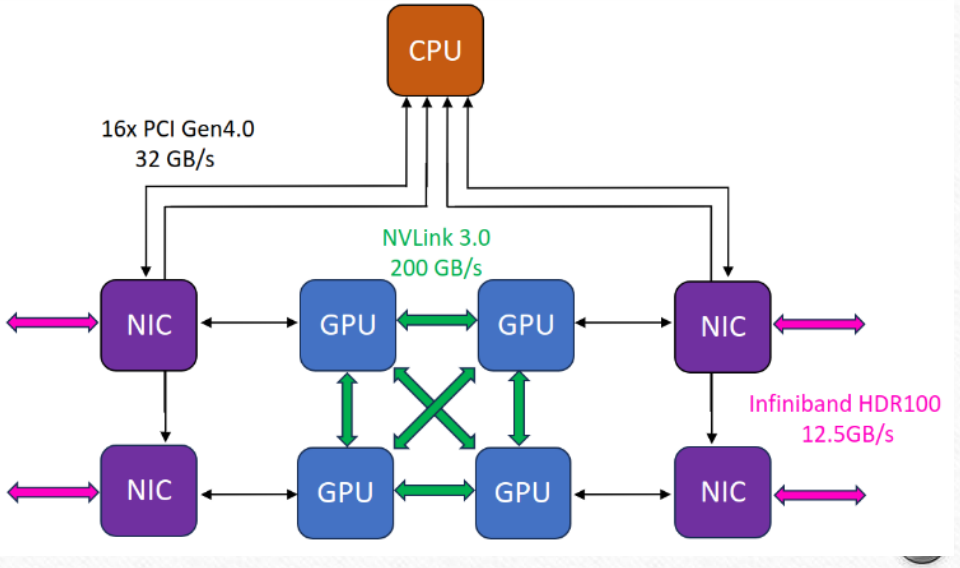


Classification based on the cores count

- Considering the number of cores (thus FLOPS) in general-purpose MIMD machines
  - $O(10^1 \div 10^2)$ cores, for a single multiprocessor chip (CMP)
    - E.g.,: The 4th generation of AMD EPYC CPU has 64 cores (128 threads)
  - $O(10^2 \div 10^3)$ cores, for a Shared-Memory tightly-coupled multiprocessor
    - E.g.,: HPE Superdome Flex series, large ccNUMA multiprocessor.
  - $O(10^3 \div 10^5)$ for Distributed-Memory loosely-coupled systems, i.e., from small to large compute clusters (comprising GP cores and GPU cores)
    - E.g.,: small-medium cluster with 16-128 nodes up to large cluster where nodes have one or more GPUs
  - $O(10^5 \div 10^6)$ top supercomputers
    - E.g.,: the Leonardo supercomputer

Core count of Leonardo

- The Leonardo supercomputer @CINECA comprises:
  - A **Data-Centric Module** composed of **1536** nodes each with 2x Intel Sapphire Rapids @2.0 GHz, 56 cores. More than **172K** general purpose cores in total
    - *Rpeak*: 1536 $nodes \times 2\ sockets \times 2 \times 10^9 Hz \times 56\ core \times 32\ FLOPS\ /10^{12}$ =11.010 PFLOPS
  - A **Booster Module** composed of **3456** nodes each with one Intel Xeon 8358 @2.6 GHz, 32 cores, plus 4 NVIDIA A100 SXM4 64GB. More than **1,8M** cores in total (CPU + GPUs cores)
    - *Rpeak* more than 306 PFLOPS

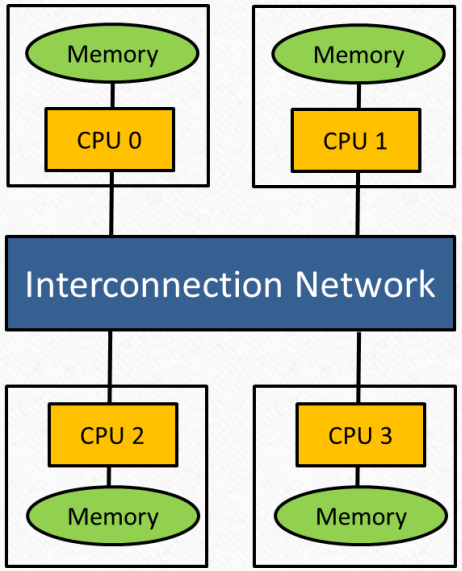


Parallel systems goals & challenges

- **Shared-memory systems**
  - **Key focus**: primarily on the memory organization (memory hierarchy, processor-memory interconnections)
  - **Goals**: minimize memory contention and mitigate the *von Neumann bottleneck*
  - **Challenges**: cache-coherence, memory consistency, thread synchronization
- **Distributed-memory systems**
  - **Key focus**: primarily on the interconnection network topology
  - **Goals**: reduce communication costs (reducing latency, increasing available bandwidth)
    - Communications among nodes is a form of I/O for the single node
  - **Challenges**: fast messaging protocols/libraries, message routing and flow control

DM systems

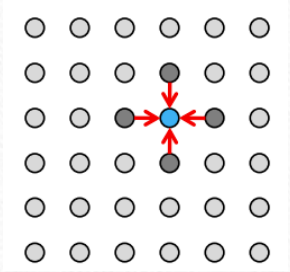
- Each **node** is a complete computer system, usually an SMP or NUMA multiprocessor.
- Processes on different nodes communicate **explicitly** by sending messages across a network
  - Reference library in HPC: **MPI** (e.g. MPI_Send, MPI_Recv, MPI_Bcast, MPI_Reduce, MPI_AllToAll)
- Depending on the network (and other aspects), we can further classify distributed systems
  - Clusters, Cloud, geographically distributed systems, pervasive infrastructures, ….
- We are interested in systems with high-performance network topologies, and homogeneous nodes, i.e., compute clusters
- Examples of high-performance network topologies: Mesh, Fat Tree, Dragonfly

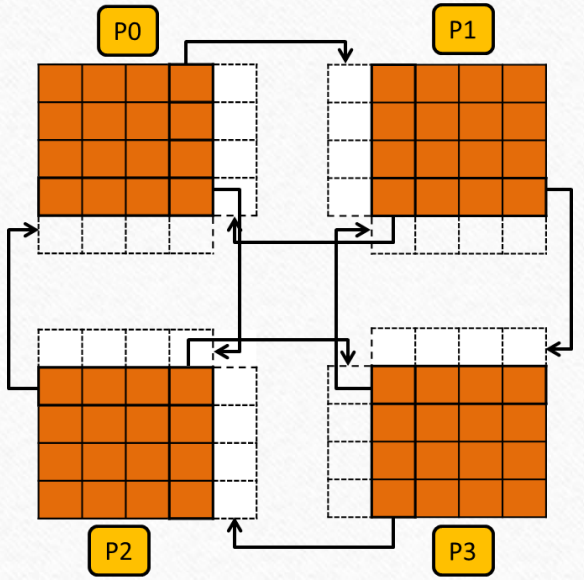


DM systems

- Example of an application running on a distributed memory system
  - **Stencil computation**. Distributed memory partitioning of a $N \times N$ matrix (8×8 in our example) onto N processes (4 in our example: P0, P1, P2, P3) running on M nodes (4), for the implementation of a 5-point stencil code. Access to the neighboring cells requires the sending and receiving of data between pairs of processes using explicit messages.

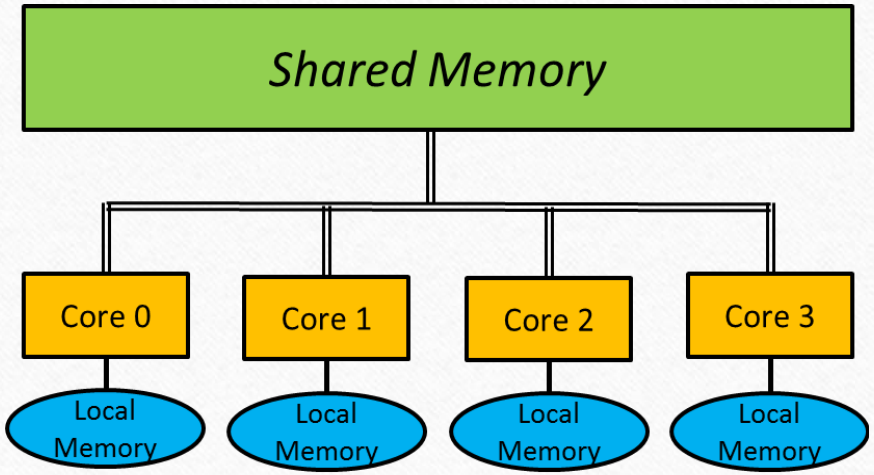
2D 5-point stencil





SHM systems

- They comprise a (modest) number of cores, all with direct hardware access to a shared memory space
  - SHM accessible through an interconnection network (e.g., shared bus, crossbar switch, fat tree, …)
  - In CMPs, all cores share the main memory
- Even in UMA systems, each core has a small local memory (e.g. L1-cache) to mitigate expensive accesses to main memory (*von Neumann bottleneck*)
  - Modern CMPs support automatic cache coherence and typically have 2-3 levels of cache (some private and some shared)
  - Some CMPs are ccNUMA
- Reference programming models: Pthreads, C++ threads, OpenMP



SHM vs DM systems

- Distributed Memory systems are more scalable, more costly, and less energy efficient
  - A modern high-end CMP node (equipped with GPUs) has more computing power than a supercomputer of 10 years ago at a fraction of the cost and power consumed
- From the standpoint of the runtime system programmer (i.e., who implements the mechanisms and software layers to simplify parallel programming)
  - For shared memory systems, the physical shared memory can be used directly for fast synchronization and communication between processes/threads. However, efficient management of locking and synchronization is generally a critical point of paying attention
  - For distributed memory systems, the most important aspect is to reduce the cost of communications as much as possible (i.e., I/O). How?
    - **Overlapping of I/O and computation**, reducing memory copies for I/O, using fast messaging protocols (e.g., Remote Direct Memory Access - RDMA)

Programming SHM systems

- Shared-Memory systems
  - **Key concept**: the primary way to program SHM systems is by exploiting the physical shared memory **through thread-level parallelism**
  - **Reference model**: **Shared Variables programming model** (e.g., Pthread, C++-threads, OpenMP)
    - Distict processes on the same node (having disjoint address spaces) may communicate via Shared Memory buffers (e.g., POSIX SHM, Sys V)
  - **Challenges**:
    - synchronization issues
    - memory hierarchy exploitations

Programming DM systems

- Distributed-Memory systems
  - **Key concept**: the primary way to exploit parallelism is through **process-level parallelism**, which involves running multiple processes simultaneously on different nodes
  - **Reference model**: **Message Passing programming model** (e.g., POSIX socket, MPI)
    - There were many attempts to provide a transparent shared-memory abstraction atop distributed-memory systems (the so-called *SW-DSM* – Software-based Distributed Shared Memory).
      - They were not enough scalable due to high synchronization costs
    - An example of a modern SW-DSM implementation is **CUDA Unified Memory** within a single node
      - Provide a unified memory space across CPU cores and GPU cores with automatic management
  - **Challenges**:
    - hide communication overheads
    - explicit synchronization via messages

Considerations on the programming model

- **Message Passing (MP) vs Shared-Variable (SV) programming models**
  - **Generality**
    - MP is more general because it can be used in both DM and SHM systems
      - MPI-based parallel code can be executed efficiently on both DM and SHM systems
    - In SHM systems MP can be implemented using SV
  - **Scalability** MP is generally more scalable than SV
  - **Intrusiveness and Complexity**
    - MP is more intrusive as it requires explicit communication for data movement and synchronization
    - More verbose and error-prone code
    - In the SV model, managing synchronization to avoid race conditions can still be complex and error-prone
  - **Performance**
    - For MP depends on the communication layer
    - For SV depends on the memory access contention and in general to the memory hierarchy

Suggested Readings

- Chapter 1 Section 1.2 «Parallelism Basics» of the Parallel Programming Concept and Practice book
- Chapter 3 Section 3.2 «Flynn’s Taxonomy» of the Parallel Programming Concept and Practice book
- Additional materials:
  - J. Diaz, C. Muñoz-Caro and A. Niño, "A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era," in IEEE Transactions on Parallel and Distributed Systems, vol. 23, no. 8, pp. 1369-1386, Aug. 2012, DOI: 10.1109/TPDS.2011.308 https://ieeexplore.ieee.org/document/6122018