Parallel Computer Architecture

Parallel computer architecture is the design of systems that use more than one processor or core to work on tasks at the same time. By sharing work among many processors, such systems can solve large or complex problems faster and more efficiently. Parallel architectures organize resources like memory and interconnects so that processors can cooperate on jobs. The ultimate goal is higher **speedup** (running faster on many processors) and **efficiency**, allowing students and engineers to tackle real-world problems with big data or heavy computation.

Memory Organizations

Memory organization in parallel computers describes how all the processors share and access data. Different designs trade off complexity, speed, and scalability:

- **Bus and Cache:** A *bus* is a common wiring ("shared communication path") that connects processors, memory, and I/O devices. Processors place addresses and data on the bus to access memory. Because bus access can be slow and contended, each processor also uses a *cache* – a small, fast memory close to the CPU. The cache stores copies of data or instructions the processor is likely to reuse, reducing the time to fetch data from main memory. In short, cache memory holds "recently used data" so that the CPU can read or write it quickly, improving performance. If multiple processors have caches, extra work is needed to keep data consistent (cache coherence), which we discuss below.

- **Backplane Bus Systems:** These systems connect multiple processors and peripherals on a single motherboard or "backplane" via one system bus. Such designs are simple and inexpensive, and they work well for a small number (around a dozen) of processors. However, since all processors share the same bus, only one processor can use it at any time. In practice this means the *effective bandwidth* drops as more processors contend for the bus. In other words, adding processors increases bus contention and slows down communication between processors and memory. Thus backplane buses are limited to modestly sized systems (typically under 16–32 CPUs).

- **Shared Memory Organizations:** In shared-memory systems, all processors access a **single global memory**. Each processor can read or write any location in that memory, communicating by using shared variables. This model is "tightly coupled": processors see the same memory contents (modulo synchronization) and do not need explicit messages. Shared-memory systems are often easier to program, since data can be shared simply via variables. However, they require hardware and software support for **cache coherence** and careful synchronization (locks, barriers, etc.) so that processors do not overwrite each other's data. A simple form is **Uniform Memory Access (UMA)**: every processor has equal-speed access to all memory. In contrast, **Non-Uniform Memory Access (NUMA)** systems have memory spread across nodes; each processor has a local memory that it accesses faster than remote memory. A NUMA machine's memories form a global address space, but access time varies by location. These designs scale to larger systems, at the cost of more complex memory management.

- **Cache Memory Organizations:** With caches in multiple processors, we must decide how writes propagate. Three common policies are:

- **Write-Through:** Each store operation updates both the cache and the main memory immediately. This is simple and keeps memory always current, but slows writes because both the cache and memory must be updated.
- **Write-Back:** The processor updates the cache only and marks the data as "dirty"; the main memory is updated later when the cache line is replaced. This reduces memory traffic but means other processors might see stale data until the cache is flushed.
- **Write-Around:** On a write miss, data is written directly to main memory without bringing it into cache. This avoids polluting the cache if the written data isn't reused soon.

Whenever multiple processors have caches, a **cache coherence protocol** ensures consistency. For example, if one CPU changes a value in its cache, other CPUs' caches must either update or invalidate their copies. Cache coherence guarantees that all processors ultimately see the same value for any shared data, preventing errors.

- **Consistency Models:** These define the order in which memory operations appear to execute. **Sequential consistency** is a strict model: it ensures that "the result of any execution is the same as if all operations were executed in some sequential order, and the operations of each individual processor appear in this sequence in program order". In simpler terms, every read and write looks as if the processors ran one instruction at a time, even though they actually overlapped. This model is easy to reason about, but limits hardware optimizations. **Weaker models** allow the hardware to reorder reads and writes for speed. They may give different processors different views of memory order, so programmers must use synchronization (locks, barriers) to enforce correctness. In practice, parallel systems often use a consistency model that balances performance and ease of programming.

Pipelining and Superscalar Techniques

To make each processor execute instructions faster, CPUs use *pipelining*. A pipeline splits instruction execution into stages, like an assembly line. While one instruction is being decoded, another is fetched, and a third is executed, all at the same time but in different stages. This overlaps work and boosts throughput (instructions per second).

- **Linear vs. Nonlinear Pipelines:** A *linear pipeline* has stages arranged one after another in a fixed order (e.g. Fetch → Decode → Execute → Memory → Write-back). Each instruction goes through each stage in sequence. In contrast, *nonlinear pipelines* have multiple paths or concurrent operations. For example, one pipeline stage might split into two parallel units that do different work, allowing two parts of an instruction to proceed in parallel. Nonlinear designs can increase parallelism within the CPU at the cost of more complex control.

- **Instruction Pipeline Design:** A classic example is the 5-stage integer pipeline. Its stages are:

- **Instruction Fetch (IF)** – read the next instruction from memory.
- **Instruction Decode (ID)** – figure out what the instruction does.
- **Execute (EX)** – perform the arithmetic or logic operation.
- **Memory Access (MEM)** – read or write data to memory if needed.
- **Write-Back (WB)** – write any results back to the register file.

In a pipelined CPU, different instructions occupy different stages simultaneously. For example, while Instruction 1 is in EX, Instruction 2 can be in ID, and Instruction 3 can be in IF. This overlapping greatly increases instruction throughput. The pipeline is most efficient if each stage takes about the same time and the stages are balanced.

- **Arithmetic Pipeline Design:** Separate pipelines are often used for complex arithmetic like floating-point multiplication or division. An arithmetic pipeline breaks a calculation into sequential steps. For example, floating-point addition can be split into stages that compare exponents, align mantissas, perform the add/subtract, and normalize the result. Each of these steps becomes a pipeline stage. As one number pair is having its mantissas added, another number pair can be in the exponent comparison stage, and so on. In this way, an *arithmetic pipeline* can process many independent arithmetic operations at once.

- **Superscalar and Super-Pipelining:** To further increase throughput, modern CPUs use *superscalar* and *super-pipelining* techniques. A **superscalar** processor issues (starts) multiple instructions in the same clock cycle by having several execution units (e.g. one integer ALU and one floating-point unit). In one cycle it might dispatch two or four instructions if they are independent. Superscalar CPUs can thus complete more than one instruction per cycle on average. In the example of a dual-issue superscalar CPU, if one instruction is an integer add and another is a floating-point multiply, both can execute in parallel in one cycle.

Super-pipelining means making the pipeline deeper by adding more stages. For instance, a simple IF stage might be split into two shorter stages (IF1, IF2). This allows a higher clock rate (shorter stage logic) but can increase the number of hazards and overhead. Super-pipelining alone does not change the parallelism of issue, but it can raise the processor's maximum speed. In practice, modern designs often use both deeper pipelines and multiple issue to maximize performance.

Parallel Computer Models

Different parallel architectures organize processors and memory in different ways. The following models represent major styles of parallel computing:

- **Evolution of Architecture:** Early computers were single-processor machines. As demand grew, designers added multiple CPUs on one motherboard (multiprocessors) or multiple cores on one chip (multicore). Today's trend includes specialized parallel units, especially GPUs (graphics processors), which have thousands of simple cores. These advancements owe much to **VLSI (Very Large Scale Integration)**, which allows millions of transistors on a chip. VLSI technology makes it possible to place many cores, caches, and even whole processors on a single chip. The result is compact, fast, and power-efficient devices – for example, modern smartphones contain multicore CPUs and GPUs all on one chip.

- **Multiprocessors and Multi-computers:** A **multiprocessor** system has multiple CPUs or cores sharing the same memory (a single address space). These are "tightly coupled" systems – often found in desktops and servers – where processors communicate simply by reading and writing shared memory. On the other hand, a **multi-computer** (or cluster) connects separate computers over a network; each node has its own local memory. These are "loosely coupled" systems. Processors must send messages over the network to share data. Distributed-memory clusters (using technologies like MPI) are common in supercomputing and data centers. In summary, **multiprocessors** use shared memory and are easy to program but hard to scale, while **multi-computers** use message passing and scale well but require explicit communication management.

- **Vector and SIMD Supercomputers:** Early supercomputers used **vector processors**. A vector processor applies a single instruction to long arrays (vectors) of data efficiently[16]. For example, it might add two 1,000-element vectors by repeatedly applying "add" in a tight loop, using specialized hardware that feeds the pipeline very quickly. Vector machines are especially good at scientific simulations and signal processing. **SIMD (Single Instruction, Multiple Data)** machines are similar in spirit: they have many simple processing elements that perform the same operation on different pieces of data simultaneously. GPUs are essentially SIMD machines: a GPU core executes the same instruction on many pixels or vertices in parallel. SIMD excels at tasks like image processing or matrix math, where the same computation is repeated over large data sets.

- **VLSI Models:** Very Large-Scale Integration refers to packing a huge number of transistors onto chips. With VLSI, designers can put multiple simple processors, caches, and interconnects on one chip. This enables massively parallel systems in a small form factor (e.g. a chip with dozens or hundreds of cores). Low-power, high-speed chips used in mobile devices and modern GPUs all rely on VLSI technology. As transistors became cheaper and faster, architects have leveraged VLSI to increase parallelism rather than just speed up a single core.

- **Dataflow Machines:** Dataflow is a more experimental model. Instead of following a fixed program order, a *dataflow machine* executes instructions as soon as their input data is available. In other words, every operation waits for its operands, and then fires. There is no program counter in the traditional sense. This can expose a lot of fine-grained parallelism, since independent operations can run out-of-order. In practice, pure dataflow hardware is rare, but the concept influenced parallel software and some specialized hardware (like certain signal processing units). Dataflow excels at concurrency: operations happen whenever their inputs arrive, which can lead to high utilization in the presence of many data streams.

Parallel Computer Architectures

Real-world parallel systems must address communication, interconnects, and performance. Key points in designing such architectures include:

- **Design Issues:** A parallel machine must consider how processors are connected (interconnect), how memory is managed, how tasks are synchronized, and how the system can grow (scalability). For example, adding more CPUs should not bottleneck on a single shared bus or a slow network. Designers choose topologies and protocols to keep processors busy and minimize waiting for data.

- **Communication Models:** There are two primary ways processors exchange data. In the **shared-memory model**, all processors see a global memory; they communicate by reading and writing shared variables. This makes programming simpler (e.g. using threads), but requires coherence and can suffer contention on memory access. In the **message-passing model**, each processor has private memory; processors explicitly send and receive messages (packets of data) to communicate. Libraries like MPI implement this. Message passing is common in distributed clusters and requires the programmer to manage communication. In simple terms: shared-memory hides communication behind memory accesses, while message-passing requires explicit sends/receives.

- **Interconnection Networks:** The topology of the network linking processors is crucial. Simple networks include a single **bus** (all-to-all but only one use at a time) or a **ring** (each CPU connected to two neighbors). More complex topologies are grids (2D **mesh**), hypercubes (each processor has links to form an n-dimensional cube), fat trees, etc. Each topology has trade-offs in cost, bandwidth, and latency. For instance, a mesh scales better than a bus, but a hypercube has very low maximum distance between any two nodes. The choice affects how quickly data can move between processors and how well the system scales.

- **Performance:** Parallel performance is measured by **speedup** and **efficiency**. *Speedup* is how much faster a program runs on p processors versus one processor: speedup $= T(1)/T(p)$. *Efficiency* is speedup divided by p, indicating how well processors are utilized. Perfect linear speedup (speedup = p) is ideal but rare due to overhead and communication. Another key concept is **scalability**: how the system's performance improves as we add more processors. This depends on task granularity, load balancing, and how well communication is overlapped with computation.

- **SIMD Computers:** SIMD architectures (as above) include **array processors** and **vector processors**. An array processor has a grid of simple ALUs, all driven by a single instruction stream. A vector processor (as in supercomputers) has special vector registers and pipelines that operate on long vectors. These systems are ideal for data-parallel tasks like image transforms or scientific loops.

- **Shared Memory Multiprocessors:** These systems often follow one of several models:

- **UMA (Uniform Memory Access):** All processors have equal-speed access to all memory. A classic UMA system is a symmetric multiprocessor (SMP).
- **NUMA (Non-Uniform Memory Access):** Each processor has faster access to its own local memory and slower access to remote memory. Access time depends on which memory bank holds the data. NUMA systems scale to more processors by combining clusters of UMA nodes with a fast interconnect.
- **COMA (Cache-Only Memory Architecture):** This is a special kind of NUMA system where all memory is treated like a big cache. Data blocks can migrate dynamically to where they are needed, and there is no fixed "home" memory for data. In effect, every memory location in the system caches frequently used data, reducing remote access costs. (COMA systems automatically move or replicate data so that it stays near the processors that use it.)

- **Message-Passing Multi-Computers:** These machines have no shared memory. Each processor has its own local memory, and processors communicate by sending messages over a network. This model underlies distributed computing and high-performance clusters (often programmed with MPI or similar libraries). Message-passing designs avoid cache coherence issues (since each piece of data lives in one place) and can scale to thousands of nodes, but they require explicit communication and synchronization by the programmer.

Conclusion

Parallel computer architecture combines clever memory design, pipelined processors, and efficient interconnects to speed up computation. By organizing how memory is shared (from simple buses to NUMA and COMA) and how processors overlap work (via pipelining, superscalar issue, and SIMD/vector units), these systems achieve high throughput. Choosing the right communication model (shared memory vs. message passing) and network topology is crucial for scalability. In the end, understanding concepts like cache coherence, pipeline hazards, and speedup lets engineers build machines that solve today's large-scale problems much faster than a single processor could.