Parallel Computing

Parallel computing means solving computing problems by performing many calculations simultaneously on multiple processors or cores. Large tasks are divided into smaller subtasks that run in parallel, with each processor working on its part. This approach is much faster than traditional serial computing, where one core runs instructions one after another. Parallelism has become crucial because single-core CPU speeds have plateaued due to physical limits. To handle today's data-intensive and real-time applications (e.g. smartphone apps, scientific simulation, AI), modern systems use multi-core chips, GPUs and clusters of processors. For example, modern smartphones and supercomputers alike use parallel hardware so that tasks like opening an app or running climate models complete hundreds of times faster than they would on a single processor.

Need for Parallelism: We use parallel computing mainly to get results faster and solve bigger problems. By "throwing more resources at a task," a parallel system can greatly reduce computation time[3]. For instance, summing a billion numbers takes only seconds on a 100-core machine instead of minutes on one core. Some problems are so large or complex (like climate modeling or large database queries) that a single processor could never finish in a reasonable time. In fact, many **Grand Challenge** scientific problems require the high speed of parallel machines (petaflops of performance) that serial computers can't handle[4]. Parallelism also lets computers run many things at once (like handling email, video, and background tasks in a phone simultaneously), making better use of modern multi-core hardware. In short, parallel computing saves time and enables new applications by using multiple processors together[3][2].

Paradigms of Parallel Computing

Parallel computing supports different **paradigms** or models of operation. Broadly, these can be divided into *synchronous* and *asynchronous* styles:

- **Synchronous paradigms:** In synchronous parallelism, all processors operate in lockstep under a global clock or barrier. Examples include **Vector/Array** processing and **SIMD (Single Instruction, Multiple Data)** systems. A vector processor applies one instruction to an entire array ("vector") of data in a pipelined fashion[5]. Similarly, a SIMD machine broadcasts the same instruction to many processing units at once, each working on different data elements[6]. For instance, a graphics processor (GPU) executes one shader program on hundreds of pixels in parallel, all in sync. Another example is a **systolic array**, a grid of simple processors where data pulses rhythmically through the network (analogous to blood flowing through a heart)[7]. In such arrays, each processor performs a small task each clock cycle, passing results to neighbors, which greatly speeds up repetitive computations like digital signal processing or certain neural-network operations[7]. These synchronous designs are deterministic and repeatable by design.

- **Asynchronous paradigms:** In asynchronous parallelism, processors work independently without a global clock. The most common asynchronous model is **MIMD (Multiple Instruction, Multiple Data)**, where each processor may execute its own instruction stream on its own data[8]. This is how today's multi-core PCs and server clusters work: different cores or nodes run different parts of a program at their own pace. For example, in a parallel web search engine, each server (processor) independently handles different queries or data shards without strict synchronization. Asynchronous models can adapt to tasks that differ or complete unevenly. A key pattern here is the **reduction paradigm**, where many processors each work on a piece of data and then combine (reduce) their results into a final answer. For example, to compute the sum of a large array, each processor sums a segment, and then these partial sums are added together (often in a tree structure) to get the total. Reduction operations (sums, maxima, logical ORs, etc.) are fundamental in parallel algorithms for combining results efficiently. In practice, an architecture or program may mix paradigms (e.g. an MIMD cluster where each node uses SIMD internally).

Hardware Taxonomy: Flynn's Classification

Flynn's classical taxonomy classifies computer architectures by how many instruction and data streams they support[9]. The four categories are:

- **SISD (Single Instruction, Single Data):** A traditional serial computer. One processor executes one instruction stream on one data stream at a time[10]. Personal computers running one task at once (in an old single-core design) are SISD machines.

- **SIMD (Single Instruction, Multiple Data):** One instruction is applied simultaneously to many data items. All processing units execute the same operation in parallel on different data elements[6]. This is ideal for vector math and image processing. Modern CPUs and GPUs have SIMD units (e.g. Intel's SSE/AVX, ARM's NEON) for speeding up array operations.

- **MISD (Multiple Instruction, Single Data):** Multiple instruction streams operate on the *same* data stream. This is rare in practice, with few real examples[11]. One conceivable use is running several filters in parallel on one data signal (e.g. multiple encryption algorithms on the same data), but MISD architectures are mainly theoretical.

- **MIMD (Multiple Instruction, Multiple Data):** Each processor has its own instruction and data streams. This is the most common parallel architecture today[8]. Multi-core servers, clusters of PCs, and grid computers are MIMD systems. They allow fully independent tasks and can run in **synchronous** or **asynchronous** mode[8]. MIMD systems also often include SIMD units internally (for example, each CPU core is MIMD but has SIMD vector instructions).

Each category has examples: SISD (classic uniprocessors), SIMD (Cray supercomputers, GPUs), MISD (very few/none), and MIMD (most parallel machines). Flynn's scheme helps us see that parallel systems range from simple (SIMD pipelines) to complex (networked MIMD).

Software Taxonomy: Kung's Taxonomy

Kung's taxonomy classifies parallel architectures by how they handle data flow and control flow[12]. It defines three software-oriented categories:

- **Processor Arrays:** These target a *data-flow* model. A grid of processors passes data from one to the next, so computations flow through the array. Systolic arrays are an example. Processor arrays suit numerical and scientific tasks where the same operation streams through data[13].

- **Multithreaded (Control-Flow):** These are built for a *control-flow* model. A processor (or chip) executes multiple threads in parallel, each fetching its own instructions. This is how conventional multi-core and many-threaded architectures work. They are used for general-purpose and application-level parallelism, like web servers handling many threads[14].

- **Hybrid:** Hybrid architectures combine aspects of both arrays and multithreading[15]. For example, a GPU is a kind of hybrid: it is essentially a large processor array (many ALUs in parallel) controlled by a few CPUs. Hybrids aim to support both intensive numeric/data tasks and flexible multithreading, useful in workloads like AI where you mix data processing and program control.

Kung's classification gives a finer view than Flynn's by focusing on programming models (data vs. control) rather than just instruction/data streams.

SPMD Model

The **SPMD (Single Program, Multiple Data)** model is a common way to write parallel programs[16]. In SPMD, all processors run the *same* program but on different pieces of data. Each processor (or core) has its own local data partition. For example, in a parallel simulation, every process runs the same code, but each reads different input or

works on a different spatial region. The processors may diverge in execution (taking different branches or work schedules) but conceptually share one program. SPMD is effectively a form of MIMD, and it underlies models like MPI and OpenMP: you launch many copies of the program and each copy (process or thread) handles part of the problem. This model simplifies development because you write only one program text, yet run it many times in parallel.