Unit 1

Syllabus - Parallelism Fundamentals – Key Concepts and Challenges – Overview of Parallel computing – Flynn's Taxonomy – Multi-Core Processors – Shared vs Distributed memory. Performance of Parallel Computers, Performance Metrics for Processors, Parallel Programming Models, Parallel Algorithms.

Parallelism Fundamentals - Key Concepts and Challenges

Before taking a discussion on Parallel Computing, first, let's take a look at the background of **computations** of computer software and **why it failed** for the modern era. Computer software was written conventionally for **serial computing**. This meant that to solve a problem, an **algorithm divides the problem** into smaller instructions. These **discrete instructions** are then **executed** on the Central Processing Unit of a computer **one by one**. Only after one instruction is finished, next one starts. A real-life example of this would be people standing in a queue waiting for a movie ticket and there is only a cashier. The cashier is giving tickets one by one to the persons. The complexity of this situation increases when there are 2 queues and only one cashier.

So, in short, Serial Computing is following:

1. In this, a problem statement is broken into discrete instructions.
2. Then the instructions are executed one by one.
3. Only one instruction is executed at any moment of time.

Look at **point 3**. This was **causing a huge problem** in the computing industry as only one instruction was getting executed at any moment of time. This was a **huge waste of hardware resources** as only one part of the hardware will be running for particular instruction and of time. As **problem statements** were getting **heavier and bulkier**, so does the amount of **time in execution** of those statements. Examples of processors are Pentium 3 and Pentium 4. Now let's come back to our real-life problem. We could definitely say that **complexity will decrease** when there are **2 queues and 2 cashiers** giving tickets to 2 persons simultaneously. This is an example of Parallel Computing.

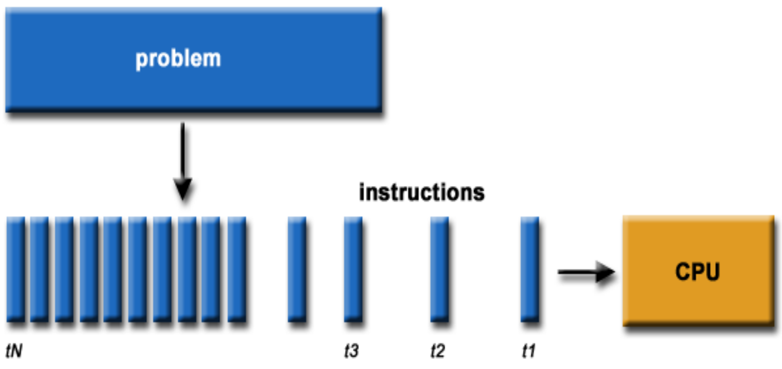


Fig. 1 Serial processing

Fundamentals of parallel programming

Parallel computation can often be a **bit more complex** compared to standard serial applications. It is the **use of multiple processing elements** simultaneously for solving any problem. Problems are **broken down into**

instructions and are **solved concurrently** as each resource that has been applied to work is working at the same time. It progresses **much faster**, and you are able to finish the task within desired time duration. This principle is the **central idea behind parallel computation**. You can dramatically cut down on **computation by splitting** one large task into **smaller tasks** that multiple processors can perform all at once. With parallel processes a task that would normally take several weeks can potentially be reduced to several hours.

Limitations of Parallel Computing:

- It addresses such as **communication and synchronization** between multiple sub-tasks and processes which is **difficult to achieve.**
- The **algorithms must be managed** in such a way that they can be handled in a parallel mechanism.
- The algorithms or programs must have **low coupling** and **high cohesion**. But it's **difficult to create** such programs.
- More **technically skilled** and **expert programmers** can code a parallelism-based program well.

Challenges of Parallel Computing

1. The Hardware Model

An **ideal processor** is one where **all constraints on ILP** (Instruction-level Parallelism) are **removed.** The only **limits on ILP** in such a processor are those imposed by the **actual data flows** through either **registers or memory.** The **assumptions made** for an ideal or perfect processor are as follows:

1. **Register renaming** - There are an **infinite number** of **virtual registers** available, and hence **all WAW** (write after write) and **WAR** (write after read) **hazards are avoided** and an unbounded number of instructions can begin execution simultaneously.

2. **Branch prediction** - Branch **prediction is perfect**. All conditional branches are **predicted exactly.**

3. **Jump prediction -** All jumps (including jump register used for return and computed jumps) are **perfectly predicted**. When **combined** with perfect **branch prediction**, this is equivalent to having a processor with **perfect speculation** and an **unbounded buffer** of instructions available for execution.

4. **Memory address alias analysis -** All memory **addresses are known** exactly, and a **load** can be moved before a **store** provided that the **addresses are not identical**. Note that this implements perfect address alias analysis.

5. **Perfect caches** - All **memory accesses** take **1 clock cycle**. In practice, superscalar processors will typically **consume large amounts** of ILP hiding cache misses, making these results highly optimistic.

To **measure** the available **parallelism,** a **set of programs** was compiled and optimized with the standard **MIPS optimizing compilers.** The programs were instrumented and **executed to produce a trace** of the instruction and data references. **Every instruction** in the trace is then **scheduled** as early as possible, **limited** only by the **data dependences**. Since a **trace is used,** perfect **branch prediction** and perfect **alias analysis** are easy to do. With these mechanisms, instructions may be **scheduled much earlier** than they would otherwise, moving across large numbers of instructions on which they are **not data dependent,** including branches, since branches are perfectly predicted.

2. Limitations on the Window Size and Maximum Issue Count

To build a processor that even comes close to **perfect branch prediction** and perfect **alias analysis** requires **extensive dynamic analysis**, since **static compile time** schemes **cannot be perfect**. Of course, most realistic **dynamic schemes** will **not be perfect**, but the use of dynamic schemes will provide the ability to **uncover parallelism** that cannot be analyzed by static compile time analysis. Thus, a dynamic processor might be able to **more closely match** the amount of parallelism uncovered by our ideal processor.

3. The Effects of Realistic Branch and Jump Prediction

Our ideal processor assumes that **branches** can be **perfectly predicted**: The **outcome** of any branch in the program is **known** before the first instruction is executed! Of course, **no real processor** can ever **achieve this.** We assume a **separate predictor** is used for jumps. **Jump predictors** are important primarily with the **most accurate branch predictors**, since the branch **frequency is higher** and the **accuracy** of the branch predictors dominates.

1. **Perfect** - All branches and jumps are **perfectly predicted** at the **start of execution.**

2. **Tournament-based branch predictor** - The prediction scheme uses a **correlating** 2-bit predictor and a **non-correlating 2-bit predictor** together with a selector, which chooses the **best predictor** for each **branch.**

4. The Effects of Finite Registers

Our ideal processor **eliminates all name dependences** among register references using an infinite set of virtual registers. To date, the **IBM Power 5** has provided the largest numbers of virtual registers: **88 additional floating-**point and **88 additional integer registers**, in addition to the **64 registers** available in the base architecture. All **240 registers are shared** by two threads when executing in multithreading mode, and all are available to a single thread when in single-thread mode.

5. The Effects of Imperfect Alias Analysis

Our optimal model assumes that it can **perfectly analyze all memory dependences,** as well as **eliminate all** register **name dependences.** Of course, **perfect alias analysis** is **not possible** in practice: The analysis cannot be perfect at compile time, and it **requires** a potentially **unbounded number of comparisons** at run time (since the number of simultaneous memory references is unconstrained). The three models are

1. **Global/stack perfect** - This model does **perfect predictions** for **global and stack references** and assumes all **heap references** conflict. This model represents an **idealized version** of the best compiler-based analysis schemes currently in production. Recent and **ongoing research on alias analysis for pointers** should improve the handling of pointers to the heap in the future.

2. **Inspection** - This model **examines the accesses** to see if they can be determined **not to interfere** at compile time. For example, if an access uses R10 as a base register with an offset of 20, then another access that uses R10 as a base register with an offset of 100 cannot interfere, assuming R10 could not have changed. In addition, addresses based on registers that point to different allocation areas (such as the global area and the stack area) are assumed never to alias. This analysis is similar to that performed by many existing commercial compilers, though newer compilers can do better, at least for loop oriented programs.

3. **None** - All **memory references** are assumed to **conflict**.

Applications of Parallel Computing:

- Databases and Data mining.
- Real-time simulation of systems.
- Science and Engineering.
- Advanced graphics, augmented reality, and virtual reality.

Overview of Parallel computing

Parallel processing is a **term used** to denote a **large class of techniques** that are used to provide **simultaneous data-processing tasks** for the purpose of increasing the computational speed of a computer system. Instead of **processing each instruction sequentially** as in a conventional computer, a parallel processing system is able to perform concurrent data processing to **achieve faster execution time.** For example, while an instruction is being executed in the ALU, the next **instruction can be read from memory.** The system may have **two or more ALUs** and be able to **execute two or more instructions** at the same time. Furthermore, the system may have two or more processors operating concurrently.

The **purpose of parallel processing** is to **speed up** the computer processing capability and **increase** its throughput, that is, the amount of processing that can be accomplished during a given interval of time. The amount of **hardware increases with parallel processing**, and with it, the cost of the system increases. However, technological developments have reduced hardware costs to the point where parallel processing techniques are **economically feasible**. Parallel processing can be **viewed from various**

levels of complexity. At the lowest level, we **distinguish between parallel** and **serial operations** by the type of registers used.

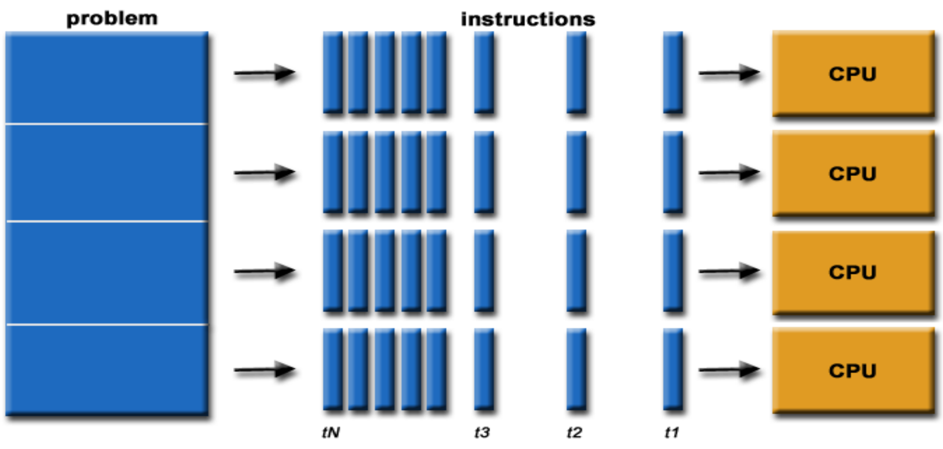


Fig. 2 Parallel processing

Shift registers operate in **serial fashion one bit at a time,** while registers with parallel load operate with **all the bits** of the word simultaneously. Parallel processing at a **higher level of complexity** can be achieved by having a **multiplicity of functional units** that perform identical or different operations simultaneously. Parallel processing is **established by distributing the data** among the multiple functional units. For example, the arithmetic, logic, and shift operations can be separated into three units and the operands diverted to each unit under the supervision of a control unit.

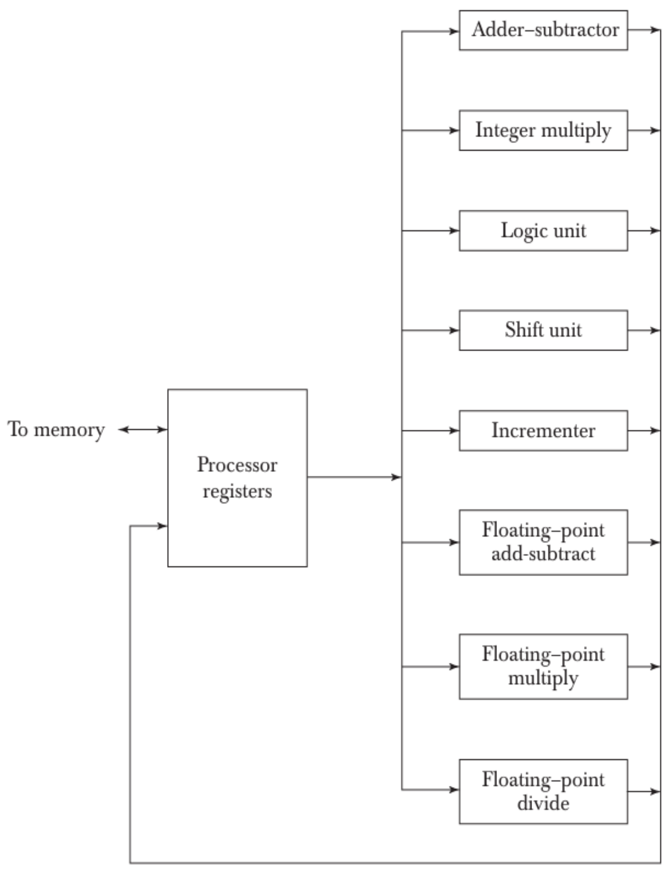


Fig. 3 Processor with multiple functional units.

Figure 3 shows one possible way of **separating the execution unit** into **eight functional units** operating in parallel. The operands in the registers are applied to one of the units depending on the operation specified by the instruction associated with the operands. The operation performed in each functional unit is indicated in each block of the diagram. The adder and integer multiplier perform the arithmetic operations with integer numbers. The floating-point operations are separated into three circuits operating in parallel. The logic, shift, and increment operations can be performed concurrently on different data. All units are independent of each other, so one number can be shifted while another number is being incremented.

How parallel computation works

Parallel computation connects **multiple processors** to **memory** that is either **pooled or connected** via **high-speed networks**. Here are three different types of parallel computation.

1. **Shared Memory Model:** In a shared memory model **all processors** to have **access to a pool of common memory** that they can freely use.

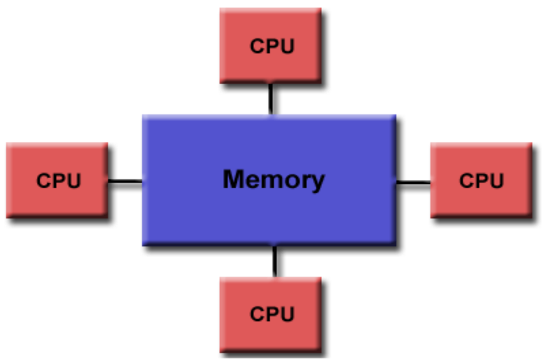


Fig. 4 Shared Memory model

2. **Distributed Memory Model:** In a distributed memory model, a **separate segment of memory** is available to **each processor.** Because memory isn't shared inherently, **information** that must be **shared** between processes is sent **over a network.**

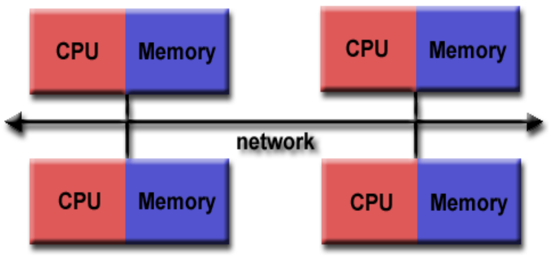


Fig. 5 Distributed Memory model

3. **Distributed/Shared Model:** A split distributed/shared model is a **hybrid between a shared and distributed model** and has the properties of both. Each **separate set of processors sharing a set of common memory** is called a node.

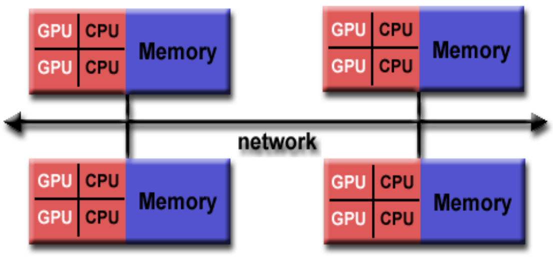


Fig. 6 Distributed/Shared Memory model

Tools for Parallel Programming - Two common solutions for creating parallel code are **OpenMP and MPI.**

1. **OpenMP -** OpenMP ("**Open Multi-Processing**") is a compiler-side application programming interface (API) for creating code that can run on a system of threads. **No external libraries** are required in order to parallelize your code. OpenMP is often considered **more user friendly** with **thread safe** methods and **parallel** sections of code that can be set with simple scoping. OpenMP is, however, limited to the amount of threads available on a node – in other words, it follows a **shared memory model.** On a node with **64 CPUs,** you can use no more than **64 processors.**

2. **MPI -** MPI ("**Message Passing Interface**") is a **library standard** for handling **parallel processing**. Unlike OpenMP, MPI has much **more flexibility** in how individual processes handle memory. MPI is also **compatible** with **multi-node** structures, allowing for **very large**, multi-node applications (i.e, **distributed memory models**). MPI is, however, often considered **less accessible** and **more difficult to learn**. Regardless, learning the **library provides a user** with the ability to **maximize processing ability**. MPI is a library standard, meaning there are several libraries based on MPI that you can use to develop parallel code. OpenMPI and Intel MPI are solutions available on most CURC systems.

Advantages

1. It **saves time** and **money** as many resources working together will reduce the time and cut potential costs.
2. It can be impractical to **solve larger problems** on Serial Computing.
3. It can take advantage of **non-local resources** when the local resources are finite.
4. Serial Computing **'wastes' the potential computing power,** thus Parallel Computing makes better work of the hardware.

SERIAL COMPUTING	**PARALLEL COMPUTING**
One task is completed at a time and all the tasks are executed by the processor in sequence	Multiple tasks are completed at a time by different processors
There is a single processor	There are multiple processors
Lower performance	Higher performance
Workload of the processor is higher	Workload per processor is lower
Data transfers are in bit-by-bit format	Data transfers are in byte form (8 bits)
Requires more time to complete the task	Requires less time to complete the task
Cost is lower	Cost is higher

Types of Parallelism:

1. **Bit-level parallelism –** It is the form of parallel computing which is based on the **increasing processor's size**. It **reduces** the **number of instructions** that the system must execute in order to perform a task on large-sized data. *Example:* Consider a scenario where an 8-bit processor must compute the sum of two 16-bit integers. It must first sum up the 8 lower-order bits, then add the 8 higher-order bits, thus requiring two instructions to perform the operation.

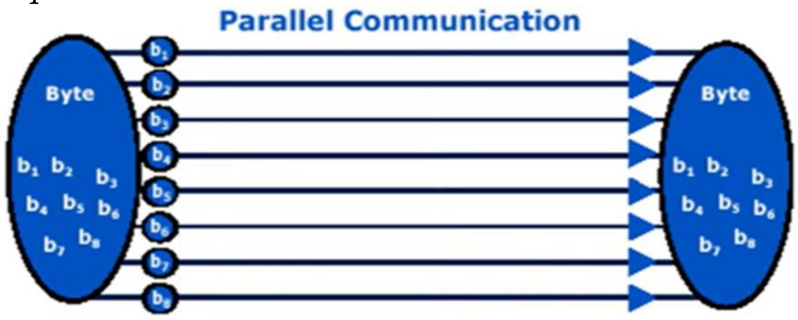

Parallel Communication
Byte
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8
Byte
b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8

Fig. 7 Bit level parallelism

2. **Instruction-level parallelism –** It refers to the situation where **different instructions** of a program are executed by different processing elements. Most processors have **several execution units** and can execute several instructions (usually machine level) at the same time. A processor can **only address less than one instruction** for each clock cycle phase.

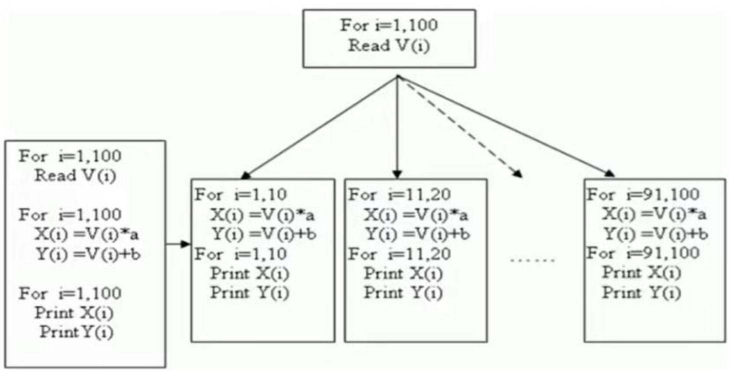


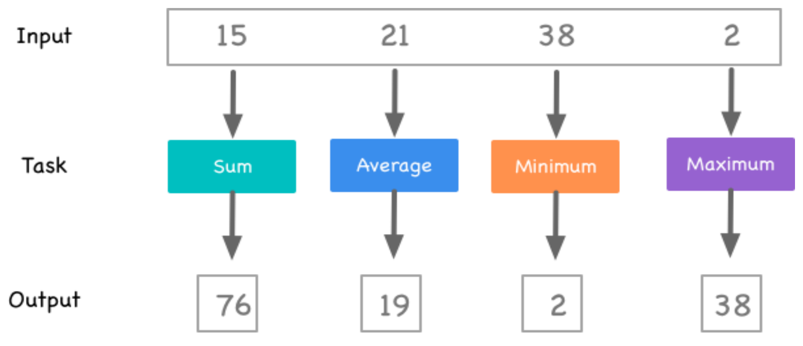
Fig. 8 Instruction level parallelism

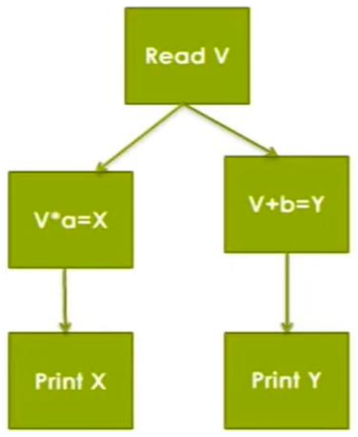
These instructions **can be re-ordered** and grouped which are later on **executed concurrently** without affecting the result of the program. This is called instruction-level parallelism. Good compilers can reorder instructions to maximize instruction throughput.

3. **Task Parallelism –** Task parallelism employs the **decomposition** of a **task into subtasks** and then allocating each of the subtasks for execution. The processors perform the execution of sub-tasks **concurrently.** In task parallelism **instead of dividing** up the **data** and doing the **same work on different processors,** we **divide the operations** to apply in different parallel units. Hence, **different tasks** are applied to the **same data** at the **same time.**

   For example, if we want to get several aggregations from the same data like sum, average, minimum, maximum etc. This is possible since there are **no dependencies** between the tasks, and so they can **run in parallel**. Each of these types require different program decomposition and aggregation schemes, and it is important that we have a clear idea of which type of problem we are trying to optimize.

Fig. 9 a) b) Task level parallelism

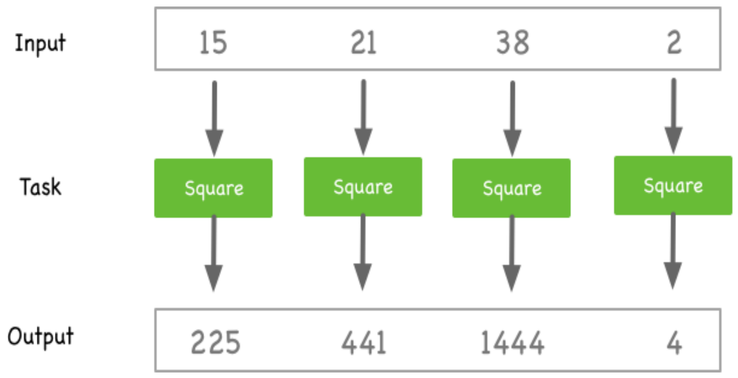




Read vector (V)
Multiply (V) *(a)=X
Sum (V)+(b)=Y
Print X, Y

4. **Data-level parallelism (DLP) –** Instructions from a **single stream operate concurrently** on several data – Limited by non-regular data manipulation patterns and by memory bandwidth. When we **partition the data** used in **solving** the problem among the **cores**, and each core **carries out more or less** similar operations on its part of the data. For example, if we want to *square* all the integers in an array, we can divide parts of the data between different parallel units and perform the squaring operations in parallel in each.

Fig. 10 Data level parallelism



5. Additionally, tasks are **often classified based** to how often their **subtasks need to synchronize** or communicate with each other.

    1. It exhibits **coarse-grained parallelism** if they do not communicate many times per second, and
    2. A task is said to exhibit **fine-grained parallelism** if its subtasks must communicate several times per second.
    3. It exhibits **embarrassing parallelism** if they rarely or never have to communicate. Embarrassingly parallel tasks are considered the easiest to parallelize.

Why parallel computing?

- The whole real-world **runs in dynamic nature** i.e. many things happen at a certain time but at different places **concurrently**. This data is **extensively huge to manage.**
- Real-world data needs **more dynamic simulation and modeling**, and for achieving the same, parallel computing is the key.
- Parallel computing provides concurrency and **saves time and money.**
- **Complex, large datasets**, and their management can be organized only and only using parallel computing's approach.
- Ensures the **effective utilization** of the resources. The hardware is guaranteed to be **used effectively** whereas in serial computation only some part of the hardware was used and the rest rendered idle.
- Also, it is **impractical to implement** real-time systems using serial computing.

Flynn's Taxonomy

To understand **parallel computing** better, a classification scheme for computer **architecture known** as *Flynn's Taxonomy* is commonly used. A multifunctional organization is usually **associated with a complex control** unit to coordinate all the activities among the various components. There are a **variety of ways** that parallel processing can be **classified.** It can be considered from the **internal organization of the processors**, from the interconnection structure between processors, or from the **flow of information** through the system.

Von Neumann architecture

The von Neumann architecture is a **fundamental computer architecture** model that describes the structure of most traditional computers. It was proposed by John von Neumann in the **1940s** and has been the basis for the **design of general-purpose computers** since then. The von Neumann architecture **consists of four main components**: the central processing unit (CPU), memory, input/output (I/O) devices, and a control unit.

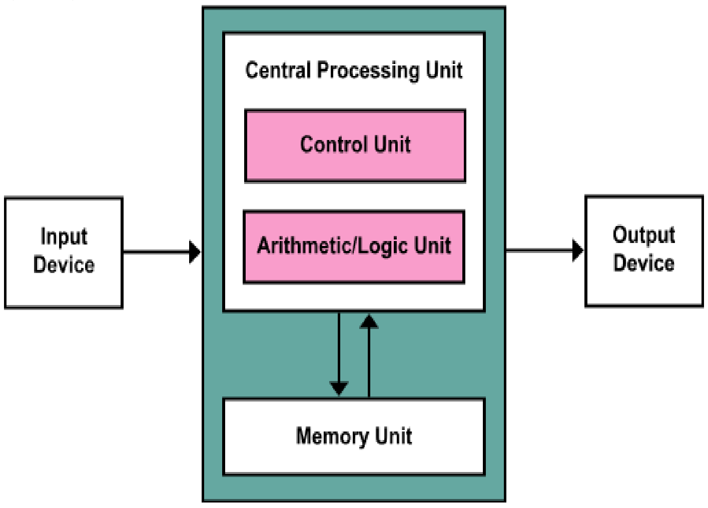


Fig. 11 Von Neumann architecture of simple computer

- **Central Processing Unit (CPU):** The part of a computer that carries out instructions and performs arithmetic, logical, and control operations.

- **Memory:** A place where the computer stores and retrieves data and instructions. Memory is divided into two types: primary memory, such as Random Access Memory (RAM), and secondary memory, like hard disk drives and solid-state drives.

- **Input-Output (I/O) devices:** Components responsible for interfacing the computer with the external world. Examples of I/O devices include keyboards, mouse, printers, and monitors.

- **System Bus:** A communication pathway that connects the CPU, memory, and I/O devices, enabling data and control signals to flow between these components.

The Von Neumann Architecture is characterized by its **simplicity and unified approach** to handling **instructions and data**. This design principle has a significant influence on the overall structure and operation of the computer system. Key features of the architecture include:

- **Unified memory structure:** Both **instructions and data** are stored together in the **same memory.**

- **Sequential instruction processing:** Program **instructions** are executed one after another in a **linear sequence.**

- **Shared system bus:** Components are **interconnected** through a **central communication pathway,** allowing for efficient communication and coordination.

- **Modularity:** The architecture is suitable for a **wide range** of computer systems, from simple **microcontrollers** to complex supercomputers, by **scaling memory and processing** capabilities.

Von Neumann Architecture also has certain **limitations:**

- **Von Neumann bottleneck:** The **single system bus** can be a performance bottleneck, as it **restricts the speed** at which data and instructions can be transferred between components. This limitation can lead to **slower overall processing** performance when dealing with large amounts of data or complex tasks.

- **Sequential instruction processing:** The linear execution of program instructions can **limit parallelism** and reduce the potential for performance improvement through parallel processing techniques, such as those used in modern multi-core processors.

- **Increased power consumption:** As components are **continuously working** together, the demand for **power increases**, resulting in **higher energy consumption** and potential heat dissipation issues.

One classification introduced by **M. J. Flynn** considers the organization of a computer system by the **number of instructions and data items** that are manipulated simultaneously. The normal operation of a computer is to **fetch instructions** from memory and **execute** them in the processor. The sequence of instructions read from memory constitutes an instruction stream. The operations performed on the data in the processor constitutes a data stream.

This system classifies a computer based on the number of **instruction streams (S)** and **data streams (M)** it can manage simultaneously. An instruction stream is the set of instructions that makes up a process, and a data stream is the set of data to be processed. Parallel processing **may occur** in the **instruction stream,** in the **data stream,** or **in both**. Flynn's classification **divides computers** into **four major** groups as follows.

1. **Single instruction stream, single data stream (SISD) -** SISD **systems are equivalent** to the classical **sequential von Neumann** architecture that **execute a single instruction** at a time and can fetch or store one data item at a time. It represents the organization of a **single computer** containing a **control unit,** a **processor unit,** and a **memory unit.** Instructions are **executed sequentially** and the system **may or may not** have internal parallel processing capabilities.

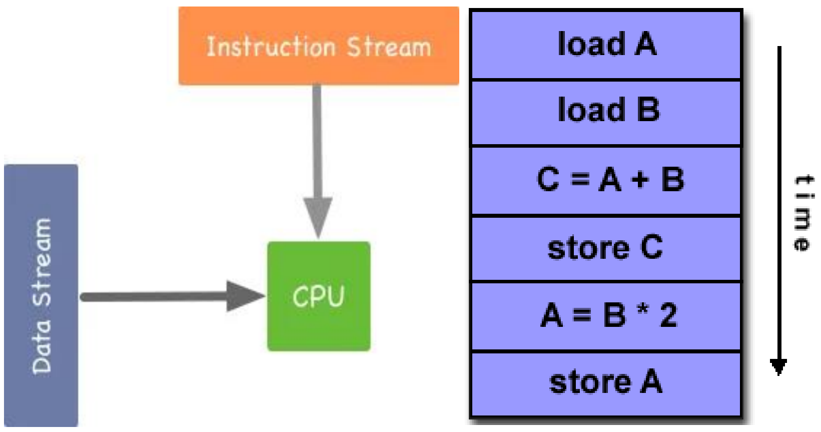


Fig. 12 SISD model

Parallel processing in this case may be achieved by means of **multiple functional units** or by **pipeline processing.**

2. **Single instruction stream, multiple data stream (SIMD) -** SIMD systems are present in most **current parallel** systems such as **GPUs** and desktop **CPUs,** where the same instruction is run on multiple data items at the same time. SIMD represents an organization that includes **many processing units** under the supervision of a **common**

control unit. The different processing units work **synchronously** and are there much **easier to implement**.

These are ideal for parallelizing simple loops that operate on large arrays of data, which is accomplished by dividing data among the processors and having the processors apply the same instructions to these subsets. All processors receive the same instruction from the control unit but **operate on different** items of data. The **shared memory unit** must contain multiple modules so that it can communicate with all the processors simultaneously.

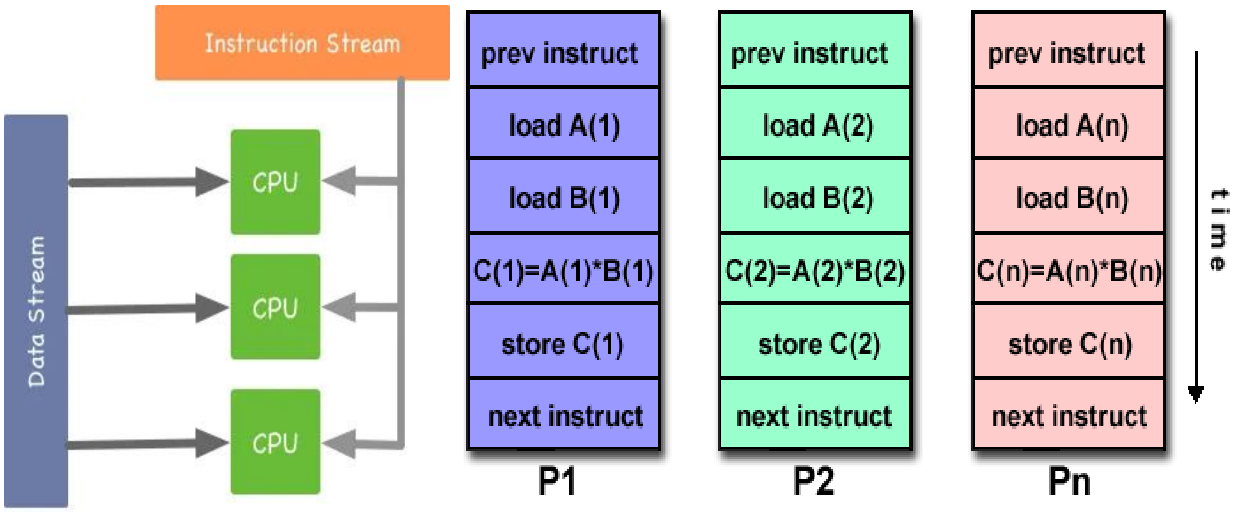


Fig. 13 SIMD model

3. **Multiple instruction stream, single data stream (MISD) -** MISD structure is only of **theoretical interest** since **no practical system** has been **constructed** using this organization.

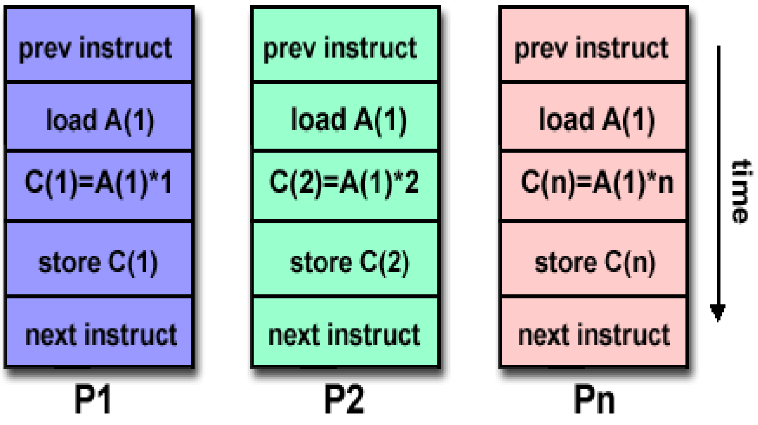


Fig. 14 MISD model

4. **Multiple instruction stream, multiple data stream (MIMD) -** MIMD organization refers to a computer system **capable of processing several programs** at the same time. MIMD systems support multiple simultaneous instruction streams operating on multiple data streams. These consist of a **collection of fully independent processing units** or cores that **work asynchronously**.

   Most multiprocessor and multi computer systems can be classified in this category. MIMD is **more flexible** than SIMD and thus more generally applicable, but it is inherently **more expensive** than SIMD. Hence, MIMD is usually **popular parallel architecture** in use for SMPs and distributed clusters that are used in more specialized use-cases. You may also find these further classified into **SPMD** and **MPMD** systems.

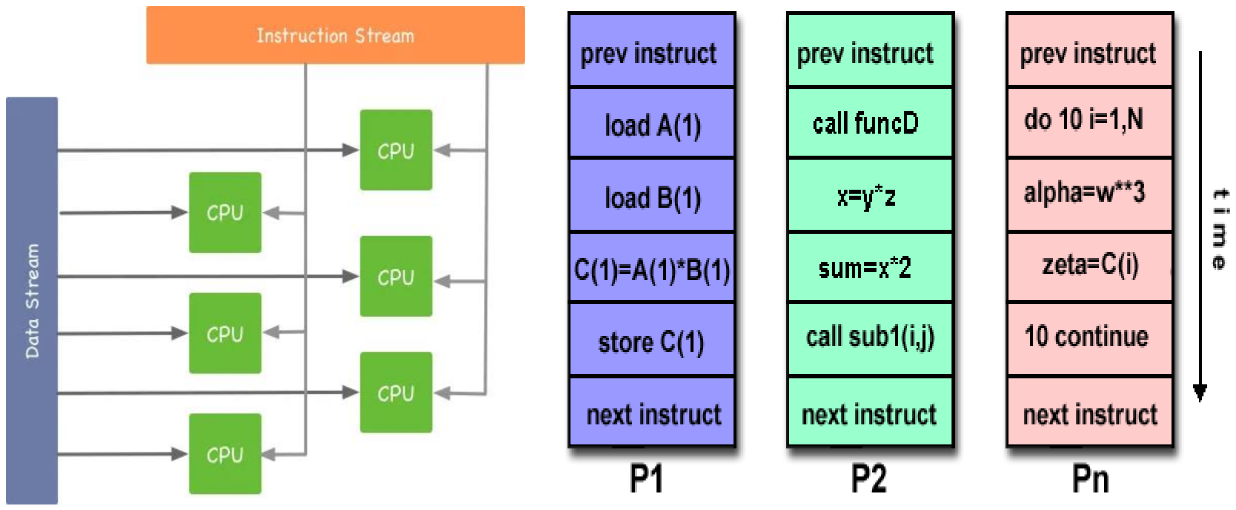


Fig. 15 MIMD model

Flynn's classification depends on the **distinction between** the **performance of the control unit** and the **data-processing unit**. It emphasizes the **behavioural characteristics of the computer** system rather than its operational and structural interconnections. Parallel processing computers are required to **meet the demands** of **large-scale computations** in many scientific, engineering, military, medical, artificial intelligence, and basic research areas. The following are **some representative applications** of parallel processing computers: Numerical weather forecasting, computational aerodynamics, finite-eleent analysis, remote-sensing applications, genetic engineering, computer-asseted tomography, and weapon research and defence.

Multi-Core Processors

There are **two methods** for creating systems of computers with **multiple processors** or processor cores:

1. **Multiprocessor organization** and
2. **Multicore organization**.

Both strategies aim to **boost** a computer's **processing power** by enabling it to handle several tasks at once. Several separate processors linked by a **communication network** make up a multiprocessor system in most cases. Each processor can carry out a **unique set of instructions** and has a separate local memory. The throughput of the entire system can be increased by these processors working on several tasks concurrently.

Multiprocessor System - A system with a multiprocessor has **several CPUs or processors.** These systems **execute multiple instructions** concurrently. **Throughput** improves as a result. The remaining CPUs will keep **operating normally** even if one CPU fails. Multiprocessors are therefore **more dependable.**

Multiprocessor systems can **take advantage of distributed** or shared memory. To execute **instructions concurrently**, each processor in a shared memory multiprocessor shares the main memory and peripherals. In these systems, the **main memory is accessed by all CPUs** via a single bus. As bus traffic increases, the majority of CPUs will be inactive. The **symmetric multiprocessor** is another name for this kind of multiprocessor. It gives each CPU access to a single memory region. A distributed memory multiprocessor **contains private memory** for each CPU. To do the computational duties, all processors can use local data. If remote data is required, the processor may access the main memory or interact with other processors over the bus.

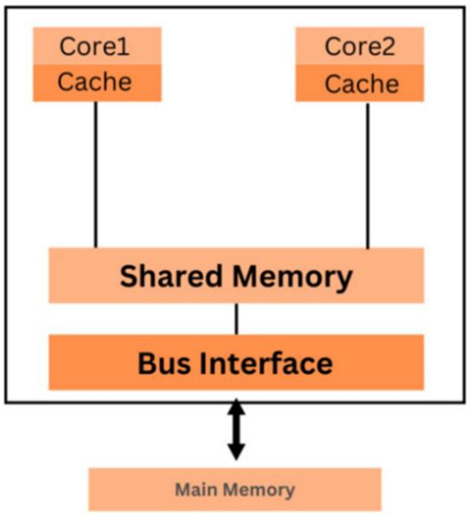


Fig. 16 Multiprocessor Architecture

Some of the **use cases** of Multiprocessor Organization.

1. High-performance computing clusters
2. Database management systems
3. Web servers
4. Virtualization and cloud computing
5. Real-time systems

Benefits of Multiprocessor System

- A multiprocessor system's **processing power increases** significantly because more tasks may be carried out simultaneously by more processors **working in parallel.**
- A multiprocessor system's **greater processing capability** makes it possible to **process huge amounts** of data more effectively, which **boosts system throughput.**
- **Workloads** can be **distributed more evenly** among several processors in a multiprocessor system, maximizing the usage of system resources.
- By enabling work to be automatically moved to other processors in the case of a failure, a multiprocessor system can provide **fault tolerance.**

Drawbacks of the Multiprocessor System

- A multiprocessor system's **design and implementation** are more **difficult** than those of one processor system.
- Task **synchronization and coordination** across numerous processors can be **difficult**, especially when using shared resources.
- Performance may be impacted by the extra communication and **synchronization overhead** that systems with multiple processors may experience.

Multi-core Processor - A multi-core processor is an **integrated circuit** that **combines two or more processors** to execute numerous tasks at once, **minimise power consumption,** and **boost performance**. It has multiple instructions, which means that various cores **run different threads** that access different regions of memory. It typically has two or more processors that **read and execute** computer instructions. It's also known as **MIMD** because all of the **processors are on the same chip.** A multi-core processor is a **single chip with multiple processing** units, or "cores," each of which may **execute distinct tasks.**

For example, while conducting many tasks at once, such as watching a movie and using WhatsApp, one core will perform activities like viewing a movie while the second core performs another WhatsApp job. **Parallel computing** is the **foundation** upon which the multi-core processor concept is built. Because it contains two or more central processing units (CPUs) in a single chip, this can dramatically improve computer speed and efficiency. A **dual-core configuration** is comparable to having several **different processors** installed on the **same computer,** but the connection between them is faster because the two CPUs are plugged into the **same socket.**

Several **instructions in parallel** may be executed by individual cores, **boosting the speed** of software built to make use of the architecture's unique features. As compared to a **single-core processor**, a dual-core processor usually is **twice as powerful** in ideal circumstances. In actuality, **performance gains** of around **50%** are expected: a dual-core CPU is roughly 1.5 times as powerful as a single-core processor.

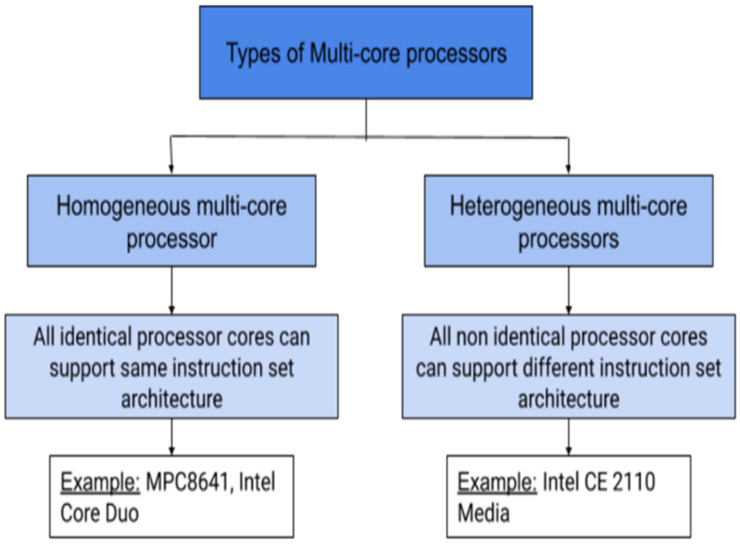

Types of Multi-core processors
Homogeneous multi-core processor
Heterogeneous multi-core processors
All identical processor cores can support same instruction set architecture
All non identical processor cores can support different instruction set architecture
Example: MPC8641, Intel Core Duo
Example: Intel CE 2110 Media

Fig. 17 Types of Multi-core Processor

Multi-core Architecture:

The multi-core processor design facilitates the **interaction between** all available **cores**, and all processing **jobs** are split and assigned as needed. After all **processing processes are done,** the processed data from each core is **delivered back** to the computer's mainboard (motherboard) via a single **common gateway.** In terms of total performance, this strategy outperforms a single-core CPU.

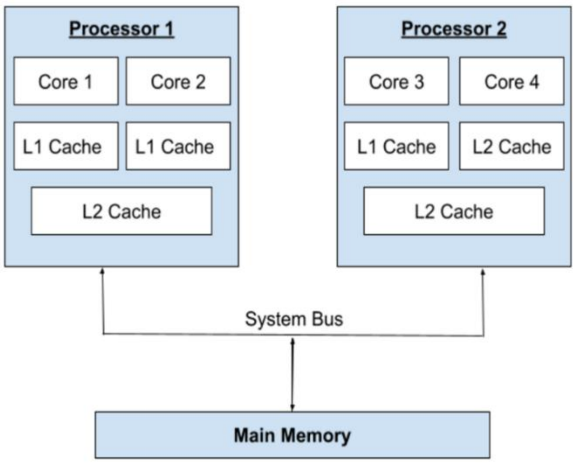


Fig. 18 Types of Multi-core Processor

A multi-core processor's architecture is shown in Figure 18. Two cores are present in the processors. The two processors share the same system bus and main memory. Each core has its own L1 cache, also known as Level 1 cache. The L2 cache, also known as the Level 2 cache, acts as a central repository for the various L1 caches. If the CPU can't find the data it needs to complete the next task in the L1 cache, it can look in the L2 cache. The L2 cache is slower than the L1 cache, although it has more memory. Because the L2 cache is shared throughout the cores, it may be used to its maximum potential. When compared to a single-core CPU of comparable speed, this strategy considerably improves performance.

Multicore processors working

The **heart** of every processor is an execution engine, also known as a **core.** The core is designed to **process instructions** and data according to the direction of software programs in the computer's memory. Over the years, designers found that every **new processor design had limits**. Numerous technologies were developed to **accelerate performance,** including the following ones:

- **Clock speed.** One approach was to make the processor's clock faster. The clock is the "drumbeat" used to **synchronize the processing** of **instructions and data** through the processing engine. Clock speeds have **accelerated from** several **megahertz** to several **gigahertz** (GHz) today.

- **Hyper-threading.** Another approach involved the handling of multiple instruction **threads**. With hyper-threading, processor cores are designed to **handle two separate instruction** threads at the same time. When properly **enabled and supported** by both the computer's **firmware and operating system** (OS), hyper-threading techniques enable **one physical core** to function as **two logical cores**. The logical abstraction of the physical processor added little real **performance** to the processor other than to help streamline the behaviour of multiple simultaneous applications running on the computer.

- **More chips.** The next step was to **add processor chips** to the processor package, which is the physical device that **plugs into the motherboard.** A dual-core processor includes two separate processor cores. A quad-core processor includes four separate cores. The **multicore approach** is almost **identical** to the use of **multiprocessor** motherboards, which have two or four separate processor sockets.

Multicore chips have **several issues** to consider.

1. Addition of more processor cores **doesn't automatically improve** computer performance. The **OS and applications** must direct software program instructions to recognize and use the multiple cores. This must be done in parallel, using various threads to different cores within the processor package.

2. The performance benefit of additional cores is not a direct multiple. **Adding** a second core does **not double** the processor's **performance**, or a quad-core processor does not multiply the processor's performance by a factor of four. This happens because of the shared elements of the processor, such as access to internal memory or caches, external buses and computer system memory. The benefit of multiple cores can be substantial, but there are practical limits.

   Example - Consider the analogy of cars on a road. Each car might be a processor, but each car must share the common roads and traffic limitations. More cars can transport more people and goods in a given time, but more cars also cause congestion and other problems.

The following are some of the **benefits of** having a **shared cache between cores:**

1. If one of the **cores isn't using the cache**, the other can, ensuring that the resources are used efficiently.
2. Provides programmers with **more freedom** by allowing for wider data sharing possibilities between cores with threads running in parallel.
3. The data for the other core can be **pre-processed or post processed** by one of the cores.
4. Allows for **effective data sharing** between cores and allows data requests to be addressed at shared-cache levels rather than in system memory.

The following are some of the **benefits of a multi-core processor** over a single-core processor:

1. **Reliability:** Because the software is distributed over multiple cores, it remains unaffected even if one component fails. In the event of a failure, just one core is affected.
2. **Multitasking:** A multi-core CPU is utilized by the operating system for executing multiple programs simultaneously.
3. **Performance:** A multi-core processor is more efficient than a single-core processor, allowing processes to run faster.
4. **Power Consumption:** In a multi-core CPU, the part that generates heat is the only part that will be used. This results in lower power consumption and, as a result, lower battery utilisation.

Disadvantages of using Multi-core processor:

1. **Application Speed**: When an application is processing, the multi-core processor can bounce from one core to another filling up the cache. This results in more time to finish the process.

2. **Analysis**: Additional memory models need to be added in multi-core processors due to its ability to multitask, making the interference analysis complex with the increase in the number of cores.

3. **Jitter**: Jitters can develop in a multi-core processor as more cores cause more interference. This can result in frequent failures in the operating system.

4. **Software Interference**: Spatial and temporal isolation are some common problems caused due to resource sharing.

Applications of Multi-core processors:

1. Games with high graphics, such as Overwatch and Star Wars Battlefront, as well as 3D games.
2. The multicore processor is more appropriate in Adobe Premiere, Adobe Photoshop, iMovie, and other video editing software.
3. Solid works with computer-aided design (CAD).
4. High network traffic and database servers.
5. Industrial robots, for example, are embedded systems.

Shared vs Distributed memory

Shared Memory - Shared memory is the **memory** which **all the processors can access**. In hardware point of view, it means **all the processors** have **direct access** to the common **physical memory** through **bus** based (usually using wires) access. These processors can **work independently** while they all access the same memory. Any **change** in the variables stored in the memory is **visible by all processors** because at any given moment all they see is a copy or picture of entire variables stored in the memory and they can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

Shared memory parallel **computers vary widely**, but generally have in common the ability for **all processors to access all memory** as global address space. Multiple **processors** can **operate independently** but **share the same memory resources**. Changes in a **memory location effected** by one processor are **visible to all** other processors. Shared memory machines can be **divided into two main classes** based upon memory access times: UMA and NUMA. Uniform Memory Access (UMA):Non-Uniform Memory Access (NUMA)

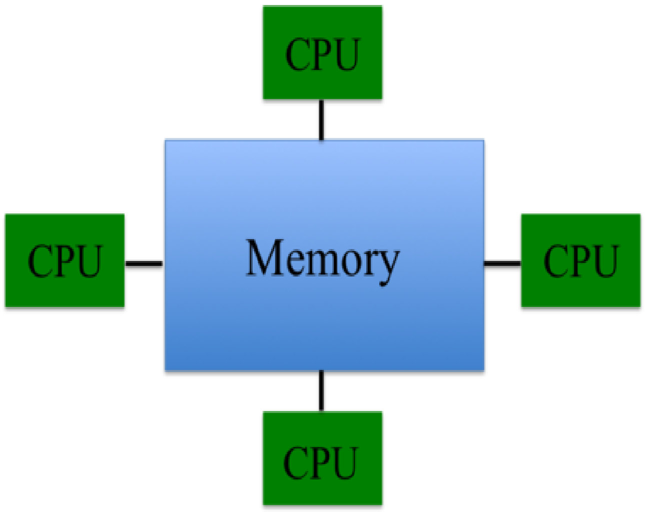


Fig. 19 UMA

Uniform Memory Access (UMA):

- Most commonly represented today by **Symmetric Multiprocessor (SMP)** machines
- **Identical processors**
- **Equal access** and access **times to memory**
- Sometimes called **CC-UMA - Cache Coherent UMA.** Cache coherent means if **one processor updates a location** in shared memory, all the **other processors know** about the update. Cache coherency is **accomplished** at the **hardware level.**

Non-Uniform Memory Access (NUMA):

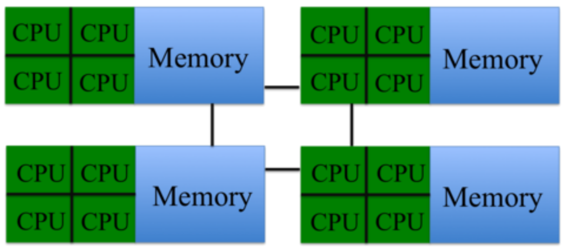


Fig. 20 NUMA

- Often made by **physically linking** two or more SMPs
- One SMP can **directly access memory** of another SMP
- **Not all** processors have **equal access time** to **all memories**
- Memory access across **link is slower**
- If **cache coherency** is maintained, then may also be **called CC-NUMA - Cache Coherent NUMA**

Advantages

1. **Global address space provides** a user-friendly programming perspective to memory

2. **Data sharing** between tasks is both **fast and uniform** due to the proximity of memory to CPUs

Disadvantages

1. Primary disadvantage is the **lack of scalability** between memory and CPUs.
2. Adding more CPUs can geometrically **increases traffic** on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
3. Programmer **responsibility for synchronization** constructs that insure "correct" access of global memory.
4. **Expense:** it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Shared Memory: UMA vs. NUMA

UMA	NUMA
Most commonly represented today by Symmetric Multiprocessor **(SMP)** machines **Identical processors.**	Non-Uniform Memory Access (NUMA) Often made by **physically linking** two or more SMPs
Share **common memory area.**	One SMP can **directly access memory** of another SMP.
Equal access and **access times** to memory.	**Not** all processors have **equal access** time to all memories.
Sometimes **called CC-UMA -** Cache Coherent UMA.	Sometimes called **CC-NUMA -** Cache Coherent NUMA.
Memory access across is **faster**.	Memory access across link is **slower.**
Cache coherent means if one **processor updates** a location in shared memory, all the other processors **know about the update.**	Cache coherency **can be maintained**
Cache coherency is accomplished at the **hardware level.**	**Cache coherency** is accomplished at the **hardware level.**

Distributed Memory - Distributed memory in hardware sense, refers to the case where the **processors can access other processor's memory** only **through network.** In software sense, it means each **processor** only can **see local machine memory** directly and should use **communications through network** to access memory of the other processors.

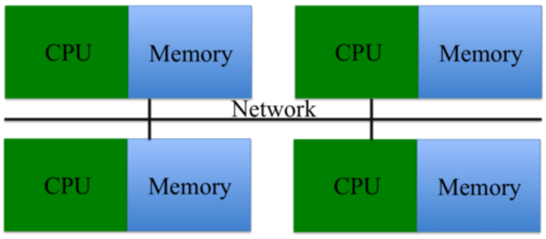
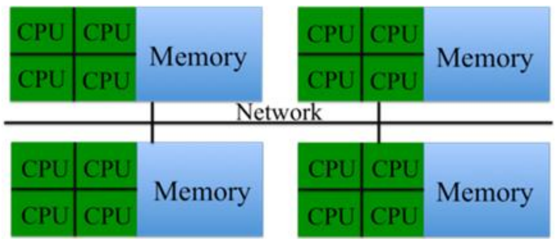


Fig. 21 Distributed Memory

Fig. 22 Hybrid Memory



Hybrid Memory - Combination of the **two kinds of memory** is what usually is used in today's **fast supercomputers.** The hybrid memory system is basically a **network of shared memories.** Within each shade's component, the **memory is accessible**

to all the CPUs, and in addition, they can access the tasks and information stored on other units through the network.

Advantages:

1. **Memory is scalable** with number of processors. **Increase** the number of **processors** and the **size of memory** increases proportionately.
2. Each processor can **rapidly access its own memory** without interference and without the overhead incurred with trying to maintain cache coherency.
3. **Cost effectiveness:** can use commodity, off-the-shelf processors and networking.

Disadvantages

1. The **programmer is responsible** for many of the details associated with **data communication between processors.**
2. It may be **difficult to map existing data structures,** based on global memory, to this memory organization.

Shared Memory		**Distributed Memory**
Shared memory varies widely, but generally have in common the ability for **all processors** to access all memory as **global address space.**	**1**	Like shared memory systems, distributed memory systems vary widely but **share a common characteristic.**
Identical processors have **equal access** and access **times to memory**	**2**	Processors have **different access** and access **times to memory**
All the **processors have direct access** to the common physical memory **through bus.**	**3**	Distributed memory systems require a **communication network** to connect **inter-processor memory.**
Processors **don't have** their own **local memory.**	**4**	Processors **have** their own **local memory.**
Memory addresses in one processor **map to another processor**, so there is **concept of global address** space across all processors.	**5**	Memory addresses in one processor **do not map to another processor**, so there is **no concept of global address** space across all processors.
Processor don't have its own local memory. Processors can **work independently** while they all **access the same memory.**	**6**	Because each processor has its **own local memory,** it operates independently.
Changes in a **memory location effected** by one processor are **visible to all** other processors, so concept of **cache coherency apply.**	**7**	**Changes** makes to its **local memory have no effect** on the memory of other processors, so concept of **cache coherency does not apply.**
When a **processor needs access** to data in **another processor,** it accesses all memory as **global address space.**	**8**	When a **processor needs access** to data in **another processor,** it is usually the **task of the programmer** to explicitly define how and when data is communicated.
One processor updates a location		**Synchronization** between tasks is

in shared memory, all the **other processors know** about the update is called cache coherency. Cache coherency is **accomplished** at the **hardware level.**	**9**	likewise the **programmer's responsibility.**
Through **buses** all the **processors have direct access** to the common physical memory.	**10**	The **network "fabric"** used for data **transfer varies widely**, though it can be as simple as Ethernet.

Performance of Parallel Computers

The **development** of parallel programming created the **need of performance metrics** and a software tool to **evaluate** the performance of a parallel algorithm in order **to decide** whether its **use is convenient or not.** The **focus** of parallel computing is to **solve large problems** in a relatively **short time.** The **factors that contribute** to the achievement of this objective are, for example,

1. The type of hardware used
2. The degree of parallelism of the problem, and
3. Which parallel programming model is adopted, and many other parameters.

To facilitate this, analysis of basic concepts was introduced, which compares the parallel algorithm obtained from the original sequence. The performance is **achieved by analyzing** and **quantifying** the **number of threads** and/or the number of **processes used**. To analyze this, a few **performance indexes** are introduced: **speedup, efficiency, and scaling.**

The **limitations** of a parallel computation are **introduced by the Ahmdal's law** to evaluate the **degree of the efficiency** of parallelization of a sequential algorithm we have the **Gustafson's law.**

1. Speedup - Speedup is the measure that displays the **benefit of solving a problem** in parallel. It is defined as the **ratio of the time taken** to solve a problem on a **single processing element**, **T_S**, to the **time required** to solve the same problem on **p identical processing** elements, **Tp.**

$S = T_s/T_p$ We have a **linear speedup,** where if ***S=p***, it means that the **speed of execution increases** with the number of processors. Of course, this is an ideal case. While the **speedup is absolute** when S<p, ***Ts*** is the execution time of the **best sequential algorithm**, the **speedup is relative** when S>p, ***Ts*** is the execution time of the **parallel algorithm** for a single processor.

- $S = p$ is linear or ideal speedup
- $S < p$ is real speedup
- $S > p$ is super linear speedup

2. Efficiency - In an ideal world, a parallel system with ***p*** **processing elements** can give us a **speedup equal to *p*.** However, this is **very rarely achieved**. Usually, some **time is wasted** in either idling or communicating.

Efficiency is a **performance metric** estimating **how well-utilized** the processors are in solving a task, compared to **how much effort is wasted** in communication and synchronization.

We denote it by ***E*** and can define it as $\mathbf{E = S/p = T_s/pT_p}$. The algorithms with **linear speedup** have the value of ***E = 1*;** in other cases, the value of *E* is less than 1. The three cases are identified as follows:

- When $E = 1$, it is a linear case
- When $E < 1$, it is a real case
- When $E \ll 1$, it is a problem that is parallelizable with low efficiency

3. Scaling - Scaling is defined as the **ability to be efficient** on a parallel machine. It **identifies the computing power** (speed of execution) in proportion with the **number of processors.** By increasing the size of the problem and at the same time the number of processors, there will be no loss in terms of performance. The scalable system, **depending on** the increments of the **different factors,** may maintain the same efficiency or improve it. $\mathbf{R_p = O_p / O_1}$

$\mathbf{O_1}$ is the total number of operations performed by one processing unit and $\mathbf{O_p}$ is the total number of operations performed by p processing units

4. Amdahl's law - Amdahl's law is a widely used law **used to design processors** and **parallel algorithms**. It states that the **maximum speedup t**hat can be achieved is limited by the **serial component** of the program:

$\mathbf{S = 1/1\text{-}P}$, where ***1 – P*** denotes the **serial component** (not parallelized) of a program. This means that for, as **an example,** a program in which 90 percent of the code can be made parallel, but 10 percent must remain serial, the maximum achievable **speedup is 9** even for an infinite number of processors.

5. Gustafson's law - Gustafson's law is based on the following considerations:

- While **increasing** the **dimension of a problem,** its **sequential parts remain constant**
- While **increasing** the **number of processors**, the **work required** on each of them still **remains the same**

This states that $\boldsymbol{S(P) = P\text{–}a\ (P\text{–}1)}$, where ***P*** is the **number of processors**, ***S*** is the **speedup**, and ***a*** is the **non-parallelizable fraction** of any parallel process. This is in contrast to Amdahl's law, which takes the **single-process execution time** to be the fixed quantity and compares it to a **shrinking per process parallel execution time.** Thus, Amdahl's law is based on the **assumption of a fixed problem size**; it assumes that the overall workload of a program does not change with respect to the machine size (that is, the number of processors).

Gustafson's law **addresses the deficiency** of Amdahl's law, which does not take into account the **total number of computing resources** involved in solving a task. It suggests that the **best way to set the time** allowed for the

solution of a parallel problem is to **consider all** the **computing resources** and on the basis of this information, it fixes the problem.

Performance Metrics for Processors

It is important to study the **performance of parallel programs** with a view to **determining the best algorithm,** evaluating **hardware platforms**, and **examining the benefits** from parallelism. A number of **metrics** have been used **based on the desired outcome** of performance analysis.

There are 2 distinct classes of performance metrics:

1. **Performance metrics for processors/cores –** assess the performance of a **processing unit,** normally done by **measuring the speed** or the number of operations that it does in a **certain period of time.**

   Some of the best-known metrics are:

   1. **MIPS –** Millions of Instructions Per Second
   2. **MFLOPS –** Millions of FLoating point Operations Per Second
   3. **SPECint –** SPEC (Standard Performance Evaluation Corporation) benchmarks that evaluate processor performance on integer arithmetic (first release in 1992)
   4. **SPECfp –** SPEC benchmarks that evaluate processor performance on floating
   5. **SPECfp –** SPEC benchmarks that evaluate processor performance on floating point operations (first release in 1989)
   6. **Whetstone –** synthetic benchmarks to assess processor performance on floating point operations (first release in 1972)
   7. **Dhrystone –** synthetic benchmarks to assess processor performance on integer arithmetic (first release in 1984)

2. **Performance metrics for parallel applications –** assess the performance of a **parallel application**, normally done by **comparing the execution time** with **multiple processing units** against the execution time with just one processing unit. We are mostly interested in metrics that measure the performance of parallel applications.

   Some of the best-known metrics are:

   1. Speedup
   2. Efficiency
   3. Scaling/Redundancy
   4. Execution Time
   5. Total Parallel Overhead
   6. Cost

1. Speedup - Speedup is the measure that displays the **benefit of solving a problem** in parallel. It is defined as the **ratio of the time taken** to solve a problem on a **single processing element**, **T_S**, to the **time required** to solve the same problem on **p identical processing** elements, **Tp.**

$S = T_s/T_p$ We have a **linear speedup,** where if ***S*=*p***, it means that the **speed of execution increases** with the number of processors. Of course, this is an ideal case. While the **speedup is absolute** when S<p, ***Ts*** is the execution time of the **best sequential algorithm**, the **speedup is relative** when S>p, ***Ts*** is the execution time of the **parallel algorithm** for a single processor.

- $S = p$ is linear or ideal speedup
- $S < p$ is real speedup
- $S > p$ is super linear speedup

2. Efficiency - In an ideal world, a parallel system with ***p*** **processing elements** can give us a **speedup equal to *p*.** However, this is **very rarely achieved**. Usually, some **time is wasted** in either idling or communicating. Efficiency is a **performance metric** estimating **how well-utilized** the processors are in solving a task, compared to **how much effort is wasted** in communication and synchronization.

We denote it by ***E*** and can define it as **$E = S/p = T_s/pT_p$**. The algorithms with **linear speedup** have the value of ***E* = *1*;** in other cases, the value of *E* is less than 1. The three cases are identified as follows:

- When $E = 1$, it is a linear case
- When $E < 1$, it is a real case
- When $E << 1$, it is a problem that is parallelizable with low efficiency

3. Scaling - Scaling is defined as the **ability to be efficient** on a parallel machine. It **identifies the computing power** (speed of execution) in proportion with the **number of processors.** By increasing the size of the problem and at the same time the number of processors, there will be no loss in terms of performance. The scalable system, **depending on** the increments of the **different factors,** may maintain the same efficiency or improve it. **$R_p = O_p / O_1$**

O_1 is the total number of operations performed by one processing unit and **O_p** is the total number of operations performed by p processing units

4. Execution Time - The serial runtime of a program is the **time elapsed between the beginning and the end** of its execution on a sequential computer. The parallel runtime is the time that elapses from the **moment a parallel computation starts** to the moment the **last processing element** finishes execution. We denote the serial runtime by TS and the parallel runtime by TP.

5. Total Parallel Overhead - The overheads incurred by a parallel program are **encapsulated into a single expression** referred to as the overhead function. We define **overhead function** or total overhead of a parallel system as the **total time collectively spent** by all the processing elements over and above that required by the **fastest known sequential algorithm** for solving the same problem on a single processing element. We **denote** the overhead function of a parallel system by the **symbol T_o**. The **total time spent** in solving a problem summed over **all processing elements** is **pT_P** . **T_S units** of this time are spent **performing useful work**, and the remainder is overhead. Therefore, the overhead function (T_o) is given by $\mathbf{T_o = pT_p - T_s}$

6. Cost - We define the cost of solving a problem on a parallel system as the **product of parallel runtime** and the **number of processing elements** used. Cost reflects the **sum of the time** that each processing element spends solving the problem. **Efficiency** can also be expressed as the ratio of the execution time of the **fastest known sequential algorithm** for solving a problem to the **cost of solving the same problem** on p processing elements. The **cost of solving a problem** on a single processing element is the execution time of the fastest **known sequential algorithm**. A parallel system is said to be **cost-optimal** if the cost of solving a problem on a parallel computer has the **same asymptotic growth** (in Q terms) as a function of the input size as the fastest-known sequential algorithm on a single processing element. Cost is sometimes referred to as work or processor-time product, and a cost-optimal system is also known as a $\mathbf{pT_P}$ **- optimal system.**

There also some laws/metrics that try to explain and assert the potential performance of a parallel application. The best known are:

1. Amdahl's law
2. Gustafson-Barsis' law
3. Karp-Flatt metric
4. Isoefficiency metric

1. Amdahl's law - Amdahl's law is a widely used law **used to design processors** and **parallel algorithms**. It states that the **maximum speedup t**hat can be achieved is limited by the **serial component** of the program:

S = 1/1-P, where ***1 – P*** denotes the **serial component** (not parallelized) of a program. This means that for, as **an example,** a program in which 90 percent of the code can be made parallel, but 10 percent must remain serial, the maximum achievable **speedup is 9** even for an infinite number of processors.

2. Gustafson's law - Gustafson's law is based on the following considerations:

- While **increasing** the **dimension of a problem,** its **sequential parts remain constant**
- While **increasing** the **number of processors**, the **work required** on each of them still **remains the same**

This states that ***S(P) = P–a (P–1)***, where ***P*** is the **number of processors**, ***S*** is the **speedup**, and ***a*** is the **non-parallelizable fraction** of any parallel process. This is in contrast to Amdahl's law, which takes the **single-process execution time** to be the fixed quantity and compares it to a **shrinking per process parallel execution time.** Thus, Amdahl's law is based on the **assumption of a fixed problem size**; it assumes that the overall workload of a program does not change with respect to the machine size (that is, the number of processors).

Gustafson's law **addresses the deficiency** of Amdahl's law, which does not take into account the **total number of computing resources** involved in solving a task. It suggests that the **best way to set the time** allowed for the **solution of a parallel problem** is to **consider all** the **computing resources** and on the basis of this information, it fixes the problem.

3. **Karp-Flatt metric**

It was proposed by Alan H Karp and Horace P. Flatt in 1990.The Karp-Flatt Metric is a measure of **parallelization of code** in **parallel processor systems.** This metric exists in addition to Amdahl's Law and the Gustafson's Law as an indication of the extent to which a particular computer code is parallelized.

Given a parallel computation exhibiting speedup psi on **p** processors, where **p > 1**, the experimentally determined serial fraction **e** is defined to be the Karp - Flatt Metric viz: **e = frac { frac{1} {psi} - frac{1} { p {1-frac{1} } {p}**

The **less the value of e** the better the parallelization.

There are many ways to measure the performance of a parallel algorithm running on a parallel processor. The Karp-Flatt metric **defines a metric** which **reveals aspects** of the performance that are **not easily discerned** from other metrics. A pseudo-"derivation" of sorts follows from Amdahl's Law, which can be written as: **T(p) = Ts + frac{Tp} {p}**

Where: **p** is the **number of processors**
T(p) is the **total time** taken for **code execution** in a **p-processor** system
Ts is the **time taken** for the **serial part** of the **code to run**
Tp is the **time taken** for the parallel part of the **code to run in one processor**

with the obvious result obtained by substituting **p = 1** then **T(1) = Ts + Tp**,

if we define the serial fraction **e = frac{Ts} {T(1)}** then the equation can be re-written as: **T(p) = T(1) e + frac{T(1) (1-e)}{p}**

In terms of the speedup
psi = frac{T(1)} {T(p)}
frac{1} {psi} = e + frac{1-e} {p}

Solving for the serial fraction, we get the Karp-Flatt metric as above.

4. **Isoefficiency metric**

The isoefficiency function is one such metric. It **relates the size** of the **problem** being solved to the **number of processors required** to maintain the efficiency at a fixed value. For a given problem size, as we **increase** the number of **processing elements**, the **overall efficiency** of the parallel system **goes down.** This phenomenon is common to all parallel systems. In many cases, the **efficiency** of a parallel system **increases** if the **problem size is increased** while keeping the number of **processing elements constant.**

Parallel execution time can be expressed as a **function of problem size,** overhead function, and the number of processing elements.
Parallel runtime: $T_P = (W + T_o(W, P))/ P$

Speedup $\mathbf{S} = W/T_P$
$= W_P/W+T_o(W, \mathbf{p})$

Efficiency $\mathbf{E} = S/P$
$= W/W+T_o(W, \mathbf{p})$
$= 1/1+T_o(W, p)/W$
$= T_o(W,P)/ W$
$= 1-E/E$
$\mathbf{W} = (E/1-E)\ T_o(W, p)$

If $\mathbf{K}= E/1-E$ be constant then, $\mathbf{W} = \mathbf{K}T_o(W, p)$

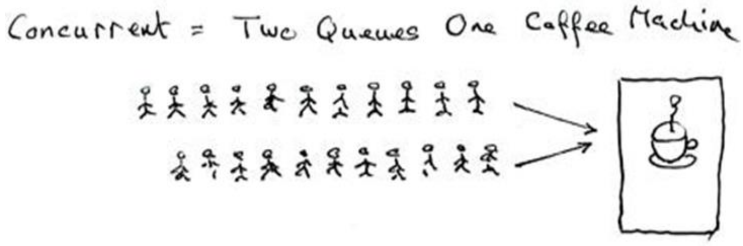
Parallel Programming Models

With the basics of the parallel hardware explained, its time to learn how they are programmed. Although processors have had **some form of parallelism** built-in for a long time, due to the traditional way of learning programming **our mental models** have been wired to **think sequentially**. Getting used to **parallelizing the problem** and debugging it is far from **trivial**. But, because of the reasons mentioned to earlier, it has become imperative to learn parallel programming in order to **get the maximum efficiency** from our hardware.

Concurrency and Parallelism

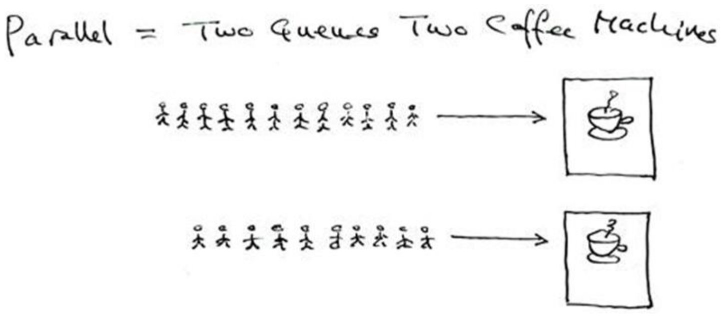
Before going any further it is important to **differentiate** the two terms **Concurrency *and* Parallelism.** Most people tend to use them interchangeably, which often **leads to misconceptions** about what parallel programming is. They are both **closely related concepts** but aren't quite the same.

Fig 23 Concurrency



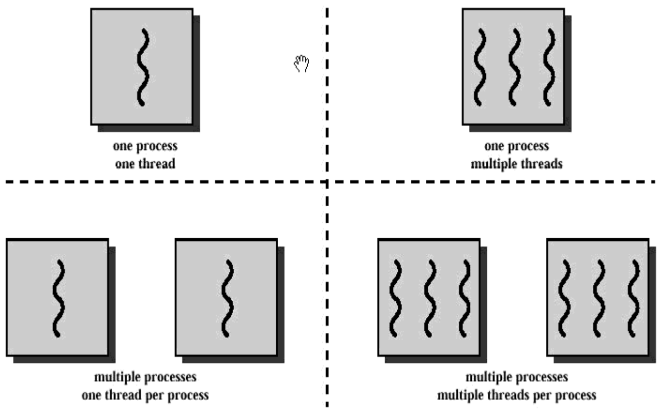
Concurrency is a much broader, general problem than parallelism. It is concerned with the **handling of more than one task at the same time** by a program to **increase its responsiveness.** This doesn't necessarily require parallel hardware and can happen by **interleaving the execution steps** of each task via **time-sharing slices**. An example is how the OS GUI allows multitasking by running many applications at the same time.

Fig 24 Parallelism



Parallelism one the other hand is concerned with **running more than one task simultaneously** on a multi-core processor to **increase the speed** of programs. One of the reasons to do so may be to **achieve concurrency** but not necessarily so. For example, if you want to convert a large number of colour photos into B/W, you can simultaneously run the B/W filter on 4 photos on a quad-core processor by batching them up in groups of four.

Fig, 25 Process and thread



Processes and Threads

Parallel programs are executed by **multi-core processors** such that one or **multiple control flows** are executed on each processor. **Depending** on their **coordination**, these control flows are referred to as **processes *or* threads**. In the simplest terms, a process *is an* ***instance of an executing program.*** Threads are generalizations of the process concept and are sometimes also **referred as a *subprocess.*** A process can consist of several threads which share a **common address space** whereas each process works on a different address space.

Which of these **two constructs is more suitable** for a given parallel computing problem **depends on the physical memory organization** of the execution environment. **Processes** are usually more **suitable for distributed memory** systems whereas **threads** are typically used for **shared memory** systems. **Another factor** that dictates which model we choose is also the restrictions in the **design of programming language** we are using. Statically-typed **compiled programming languages** like C++, Java and C# primarily **use OS threads**, while due to restriction such as the infamous Global Interpreter Lock (GIL), **dynamic interpreted languages** like Python and Ruby tend to favour **processes** to utilize all the available cores.

Parallel Models - The need for a parallel model arises in order **to understand the strategy** that is used for the **partitioning of data** and the ways in which these **data are being processed.** Therefore every model being used provides proper structuring based on **two techniques**. They are as follows:

1. Selection of **proper partitioning** and
2. **mapping techniques** and proper **use of strategy** in order to reduce the interaction.

Types of Parallel Models

1. The Data-Parallel Model - The data-parallel model algorithm is one of the **simplest models** of all other parallel algorithm models. In this model, the **tasks** that need to be carried out are **identified first** and then **mapped to the processes.** This mapping of tasks onto the processes is being **done statically or semi-statically.** In this model, the **task** that is being **performed** by every process is the same or **identical** but the **data** on which these **operations or tasks** are performed is **different.**

The problem to be solved is **divided into a number of tasks** on the basis of data partitioning. Here data partitioning is being used because all the operations performed by each process are **similar and proper uniform partitioning of data** followed by static mapping assures the proper load balancing. **Example:** Dense Matrix Multiplication

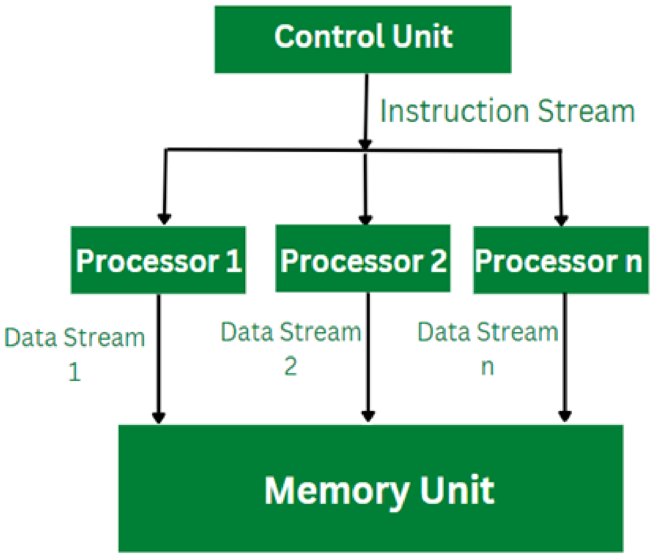


Fig. 26 Dense Matrix Multiplication

In the above example of dense matrix multiplication, the instruction stream is being divided into the available number of processors. Each processor computes the data stream it is allocated with and accesses the memory unit for read and write operation. As shown in the above figure, the data stream 1 is allocated to processor 1, once it computes the calculation the result is being stored in the memory unit.

2. The Task Graph Model - The **task dependency graph** is being used by the parallel algorithms for **describing the computations** it performs. Therefore, the use of **interrelationships** among the **tasks** in the task dependency graph can be used for reducing the interaction costs.

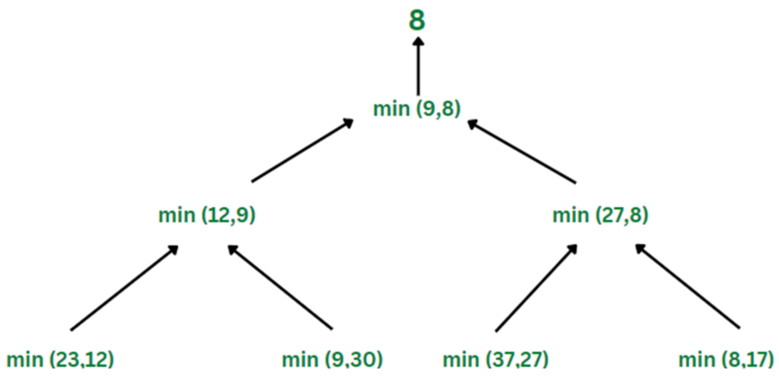


Fig. 27 finding the Minimum Number

This model can be **used effectively** for solving problems in which **tasks** are associated with a **large amount of data** as **compared** to that **actual computation**. The parallelism that is described with the task dependency graph where **each task is an independent** task is known as **task parallelism**. The task graph model is majorly used for the implementation of parallel quick sort, a parallel algorithm based on divide and conquer. **Example:** Finding the minimum number

In the above example of finding the minimum number, the task graph model works parallelly in order to find the minimum number in the given stream. As shown in the above figure, the minimum of 23 and 12 is computed and passed on further by one process, similarly at the same time the minimum of 9 and 30 is calculated and passed on to the further process. This approach of computation requires less time and effort.

3. Work Pool Model - The work pool model is also **known as the task pool model**. This model makes use of a **dynamic mapping approach** for task assignment in order to **handle load balancing**. The size of some processes or tasks is small and requires less time. Whereas some tasks are of large size and therefore require more time for processing. In order to **avoid the inefficiency** load balancing is required.

The **pool of tasks** is created. These tasks are **allocated to the processes** that are idle in the runtime. This work pool model can be **used in** the **message-passing approach** where the data that is associated with the tasks is smaller than the computation required for that task. In this model, the task is moved without causing more interaction overhead. **Example:** Parallel tree search

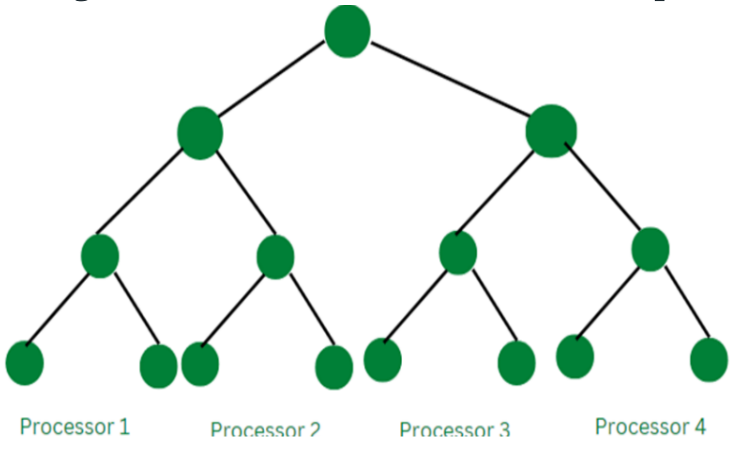


Fig. 28 Parallel Search Tree

In the above example of the parallel search tree, that uses the work pool model for its computation uses four processors simultaneously. The four sub-tress are allocated to four processors and they carry out the search operation.

4. Master-Slave Model - Master Slave Model is also **known as Manager-worker** model. The **work** is being **divided** among the process. In this model, there are **two different types** of processes namely **master process** and **slave process**. **One or more** process acts as a **master** and the remaining **all other** process acts as a **slave**. **Master allocates the tasks** to the slave processes according to the requirements. The **allocation** of tasks **depends on the size** of that task. If the size of the task can be calculated on a prior basis the master allocates it to the required processes.

If the **size** of the task **cannot be calculated** prior the master allocates some of the **work to every process at different times**. The master-slave model works more efficiently when work has to be done in different phases where the master assigns different slaves to perform tasks at different phases. In the master-slave model, the **master is responsible** for the **allocation** of tasks and **synchronizing** the activities of the slaves. The master-slave model is generally efficient and used for shared address space and message-passing paradigms. **Example:** Distribution of workload across multiple slave nodes by the master process

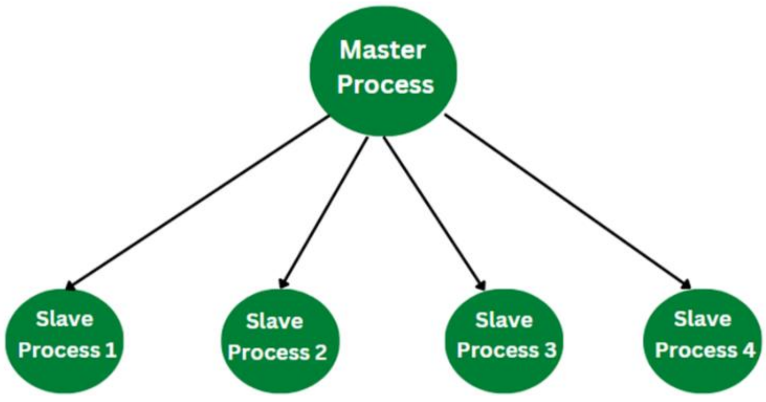


Fig. 29 Distribution of workload across multiple slave nodes by the master process

As shown in the example of the Master-Slave model, the distribution of workload is being done across multiple processes. As shown in the above diagram, one node is the master process that allocates the workload to the other four slave processes. In this way, each sub-computation is carried out by multiple slave processes.

5. The Pipeline Model - The Pipeline Model is also known as the **Producer-Consumer model**. This model is **based on** the **passing of a data stream** through the processes that are arranged in succession. Here a **single task** goes through all the other **processes**. They are then accessed by the required processes **in a sequential manner**. Once the processing of **one process is finished** it goes to the **next present process**. In this model, the pipeline acts as a chain of producers and consumers.

This pipeline of producers and consumers can also be **arranged in a directed graph-**like fashion rather than a **linear chain.** The approach of Static mapping is being used for mapping of tasks onto the processes. **Example:** Parallel LU factorization algorithm

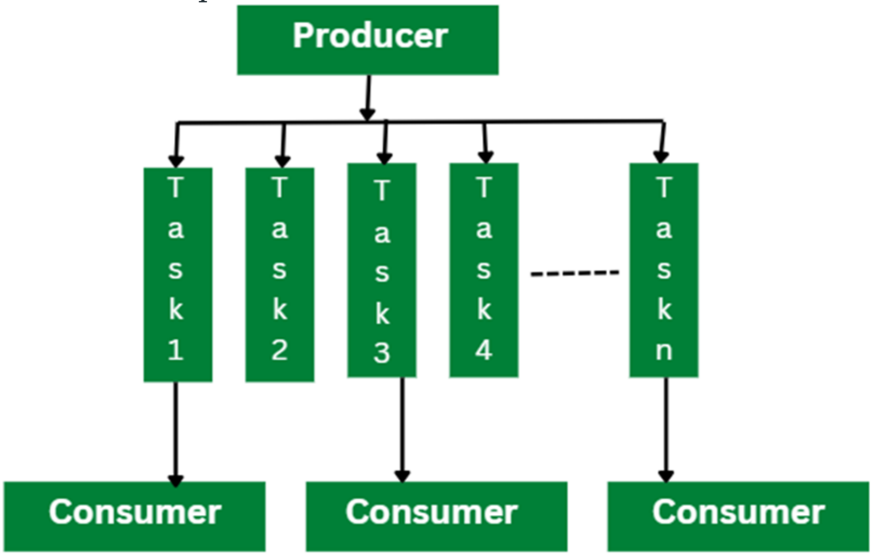


Fig. 30 Parallel factorization algorithm

As shown in the above diagram, the Parallel LU factorization algorithm uses the pipeline model (Matrix multiplication L lower triangle & U upper triangle in matrix). In this model, the producer reads the input matrix and generated the tasks that are required for computing the LU factorization as an output. The producer divides this input matrix into a smaller size of multiple tasks and shares them into a shared task queue. The consumers then retrieve these blocks and perform the LU factorization on each independent block.

6. Hybrid Model - A hybrid model is the **combination of more than one** parallel model. This combination can be applied **sequentially or hierarchically** to the different phases of the parallel algorithm. The model that can be efficient for performing the task is selected as a model for that particular phase. **Example:** A combination of master-slave, work pool, and data graph model.

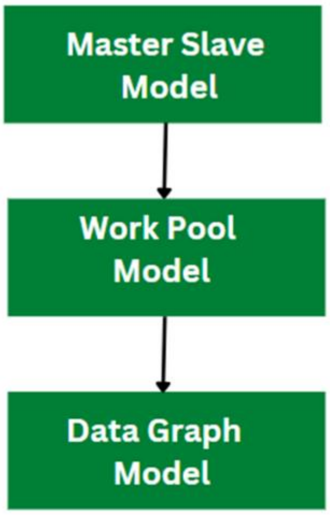


Fig. 31 A combination of master slave, work pool and data graph model

As shown in the above hybrid model where three different models are used at each phase master-slave model, the work pool model, and the data graph model. Consider the above example where the **master-slave model is** used for the **data transformation** task. The master process distributes the task to multiple slave processes for parallel computation. In the second phase **work pool model** is used for **data analysis** and similarly **data graph model** is used for making the **data visualization.** In this way, the **operation is** carried out in **multiple phases** and by using different parallel algorithm models at each phase.

Parallel Algorithms

Divide-and-conquer

A divide-and-conquer algorithm **splits the problem** to be **solved into subproblems** that are easier to solve than the original problem, solves the subproblems, and **merges the solutions** to the subproblems to construct a solution to the original problem. The divide-and-conquer paradigm **improves program modularity**, and often leads to **simple and efficient** algorithms.

It has therefore proven to be a **powerful tool** for sequential algorithm designers. Divide-and-conquer plays an even more **prominent role in parallel** algorithm design. Because the **subproblems created in the first step** are typically independent, they can be solved in parallel. Often the subproblems are **solved recursively** and thus the next **divide step yields** even **more subproblems** to be solved in parallel. As a consequence, even divide-and-conquer algorithms that were designed for sequential machines typically have some inherent parallelism.

However, that in order for divide-and-conquer to **yield a highly parallel algorithm,** it is often necessary to parallelize the divide step and the merge step, so that they can all be solved in parallel. As an **example** of parallel **divide-and-conquer,** consider the sequential **merge sort algorithm.** Merge sort takes a **sequence of n keys** as input and returns the keys in sorted order. It works by splitting the keys into two sequences of n/2 keys, recursively sorting each sequence, and then merging the two sorted sequences of n/2 keys into a sorted sequence of n keys. To analyze the sequential running time of merge sort we note that two sorted sequences of n/2 keys can be merged in **O(n) tim**e. Hence the running time can be specified by the recurrence T(n)

$$T(n) = \begin{cases} 2T(n/2) + O(n) & n > 1 \\ O(1) & n = 1 \end{cases}$$

Using a technique **called pipelined** divide-and-conquer the depth of merge sort can be further **reduced to O(log n)**. Divide-and-conquer has proven to be **one of the most powerful techniques** for solving problems in parallel.

Randomization

Random numbers are **used in** parallel algorithms to ensure that processors can **make local decisions** which, with **high probability**, add up to good **global decisions**. Here we consider **three uses** of randomness.

1. **Sampling:** One use of randomness is to **select a representative sample** from a set of elements. Often, a problem can be solved by

selecting a sample, **solving the problem on that sample**, and then using the solution for the sample to guide the solution for the original set. **For example**, suppose we want to **sort** a collection of **integer keys**. This can be accomplished by **partitioning** the keys **into buckets** and then sorting within each bucket. For this to work well, the **buckets** must represent **non-overlapping intervals** of integer values, and each bucket must contain approximately the **same number of keys.** Random sampling is used to **determine the boundaries** of the intervals. First each **processor selects a random** sample of its keys. Next all of the selected **keys are sorted** together. Finally these **keys** are **used as** the **boundaries**. Such random sampling is also **used in** many parallel **computational geometry,** graph, and **string matching** algorithms.

2. **Symmetry breaking:** Another use of randomness is in symmetry breaking. For example, consider the problem of **selecting a large independent set** of vertices in a graph in parallel. A set of vertices is independent if **no two are neighbors**. Imagine that each vertex must decide, in parallel with all other vertices, whether to join the set or not. Hence, if **one vertex chooses to join** the set, then **all** of its **neighbors** must **choose not to join** the set. The choice is **difficult** to make simultaneously by each vertex if the **local structure** at each vertex **is the same**. As it turns out, the impasse can be resolved by using randomness to break the symmetry between the vertices.

3. **Load balancing:** A third use of randomness is load balancing. One way to **quickly partition** a large number of data items into a collection of approximately **evenly sized subsets** is to randomly assign each element to a subset. This technique works best when the average size of a subset is **at least logarithmic in the size** of the original set.

Parallel pointer techniques

Many of the **traditional sequential techniques** for manipulating lists, trees, and graphs **do not translate** easily into parallel techniques. **For example**, techniques such as **traversing** the elements of a linked list, **visiting** the nodes of a tree in **postorder**, or performing a **depth-first traversal** of a graph appear to be inherently sequential. Fortunately these techniques can often be **replaced by parallel techniques** with roughly the same power.

1. **Pointer jumping -** One of the **oldest parallel pointer techniques** is pointer jumping. This technique can be applied to either **lists or trees.** In each pointer jumping step, each **node** in parallel **replaces its pointer** with that of its **successor (or parent)**. For **example,** one way to label each node of an **n-node list (or tree)** with the label of the **last node (or root)** is to use pointer jumping. After **at most ⌈log n⌉** steps, every node points to the same node, the end of the list (or root of the tree).

2. **Euler tour -** An Euler tour of a **directed graph** is a path through the graph in which **every edge is traversed** exactly once. In an **undirected graph** each edge is typically **replaced** with two **oppositely**

directed edges. The Euler tour of an **undirected tree** follows the **perimeter** of the tree visiting each **edge twice**, once on the way down and once on the way up. By keeping a **linked structure** that represents the Euler tour of a tree it is possible to **compute many functions** on the tree, such as the size of each subtree. This technique **uses linear work**, and **parallel depth** that is independent of the depth of the tree. The Euler tour can often be used to **replace a standard traversal** of a tree, such as a depth-first traversal.

3. **Graph contraction -** Graph contraction is an **operation** in which a graph is **reduced in size** while **maintaining** some of its **original structure**. Typically, after performing a graph contraction operation, the **problem is solved recursively** on the contracted graph. The solution to the problem on the **contracted graph** is then used to form the final solution. For **example**, one way to partition a graph into its connected components is to first **contract the graph** by merging some of the vertices with neighboring vertices, then **find the connected components** of the contracted graph, and finally **undo the contraction** operation. Many **problems** can be solved by contracting **trees**, in which case the technique is called **tree contraction.**

4. **Ear decomposition** - An ear decomposition of a graph is a **partition** of its **edges** into an **ordered collection of paths.** The **first path** is a **cycle**, and the others are called **ears**. The **end-points** of each **ear** are **anchored** on previous paths. Once an **ear decomposition** of a graph is found, it is not difficult to determine if **two edges** lie on a **common cycle**. This information can be **used in algorithms** for determining biconnectivity, triconnectivity, 4-connectivity, and planarity. An ear decomposition can be found in **parallel using linear work** and **logarithmic depth,** independent of the structure of the graph. Hence, this technique can be **used to replace** the standard sequential technique for solving these problems, depth-first search.