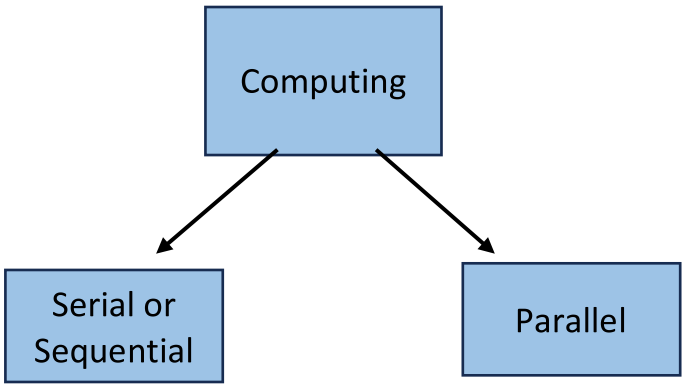
UNIT-1: TOPIC 1

Introduction To Parallel Computing

Basics

"**Computing"** refers to the process of using computers to perform various tasks, such as data processing, information storage, and solving complex problems.

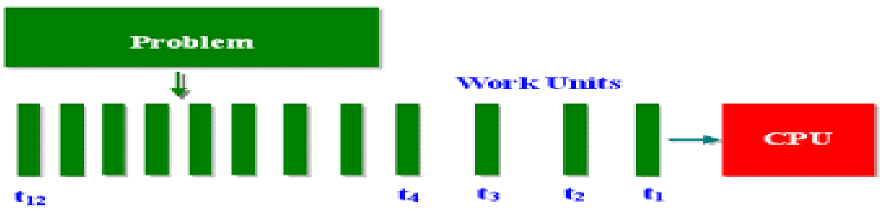
This computing can be either done in Serial Way known as **Serial Computing** or in Parallel Way known as **Parallel Computing**



Serial Computing:

Serial computing refers to traditional computing where tasks are executed sequentially, one after the other, using a single processor. In serial computing, each instruction or task must wait for the previous one to complete before it can be executed. This approach limits the speed and efficiency of processing, especially when dealing with complex or time-consuming tasks.

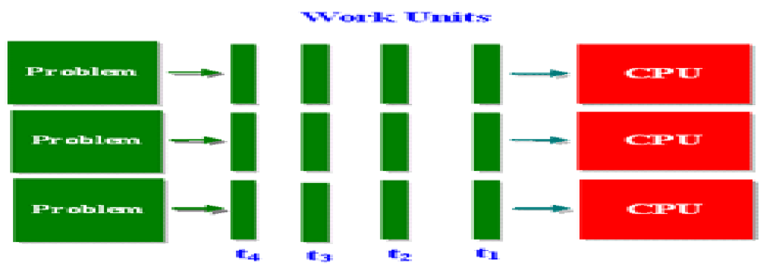
Example of Serial Computing: Consider a task of sorting a large dataset of numbers in ascending order. In a serial computing environment, a single processor would go through the entire dataset, comparing and rearranging numbers one pair at a time until the entire dataset is sorted. This process occurs sequentially, and each comparison and rearrangement must wait for the previous one to finish.



Parallel computing

Parallel computing is a type of computation in which multiple processors or computers work together to solve a problem. Instead of one single processor handling the entire task, parallel computing divides the task into smaller sub-tasks that can be processed simultaneously. This simultaneous processing can lead to significant improvements in computational speed and efficiency.

Example of Parallel Computing: Using the same example of sorting a large dataset, parallel computing would involve dividing the dataset into smaller chunks, and each chunk is sorted independently by a separate processor. These processors work in parallel, sorting their respective chunks simultaneously. Once all processors have completed sorting their portions, the sorted chunks can be combined to produce the final sorted dataset.



DEFINITION Parallel computing is the practice of identifying and exposing parallelism in algorithms, expressing this in our software, and understanding the costs, benefits, and limitations of the chosen implementation.

Benefits of Parallel Computing

- **Faster Run Time with More Compute Cores:** Parallelization involves dividing a task into smaller sub-tasks that can be executed simultaneously, utilizing multiple cores to process the data. This approach can significantly reduce the time required to complete the task, as each core works on a separate portion of the problem concurrently.
- **Larger Problem Sizes with More Compute Nodes:** With more nodes, you can break down your problem into smaller pieces that each node can work on simultaneously, which is especially beneficial for handling larger datasets and more complex simulations.
- **Energy Efficiency by Doing More with Less:** In the context of parallel computing, the concept of "doing more with less" often revolves around optimizing energy efficiency while achieving better computational performance.This can be achieved by making use of dynamic

resource allocation and workload consolidation to ensure that the number of processors used is proportional to the workload. Turn off or put to sleep any unused processors.
The energy consumption for your application can be estimated using the formula
P = (N Processors) × (R Watts/Processors) × (T hours)
where P is the energy consumption, N is the number of processors, R is the thermal design power,and T is the application run time.

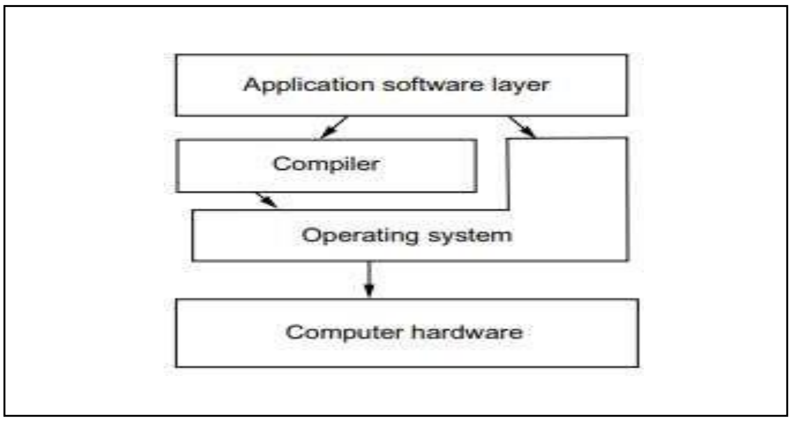
- **Scalability:** Parallel computing can be easily scaled by adding more processors, which further enhances performance. Serial computing does not scale in this manner, as it relies on a single processor.
- **Parallel Computing Can Reduce Costs:** As technology advances, the cost of individual processors and memory decreases. Parallel computing systems can take advantage of these cost reductions, making it more economical to build high-performance computing clusters or data centers.

Applications of Parallel Computing:

- **Scientific Simulations:** Used in fields such as physics, chemistry, and engineering for complex simulations.
- **Big Data Processing:** Parallel computing is crucial in processing vast amounts of data in fields like data analytics and machine learning.
- **Weather Forecasting:** Enables complex weather simulations and predictions.
- **Video and Image Processing:** Parallelism accelerates tasks like video rendering and image recognition.
- **Financial Modelling:** Used for risk analysis, option pricing, and other complex financial calculations.

How parallel computing Works

As a developer, you are responsible for the application software layer, which includes your source code.



In the source code, you make choices about the programming language and parallel software interfaces you use to leverage the underlying hardware. Additionally, you decide how to break up your work into parallel units. A compiler is designed to translate your source code into a form the hardware can execute. With these instructions at hand, anOS manages executing these on the computer hardware.

Parallel Approach models are used to express parallelization in an application software layer that gets mapped to the computer hardware through the compiler and the OS. Parallel computing approaches involve various models and paradigms that define how tasks are divided, coordinated, and executed in parallel systems. Here are some common parallel computing approach models:

Parallel Approach Models

Hardware Models

Distributed Memory Architecture: A Cross-Node Parallel Method
Shared Memory Architecture: An On-Node Parallel Method
Vector Units: Multiple Operations With One Instruction
Accelerator Device: A Special-Purpose Add-On Processor

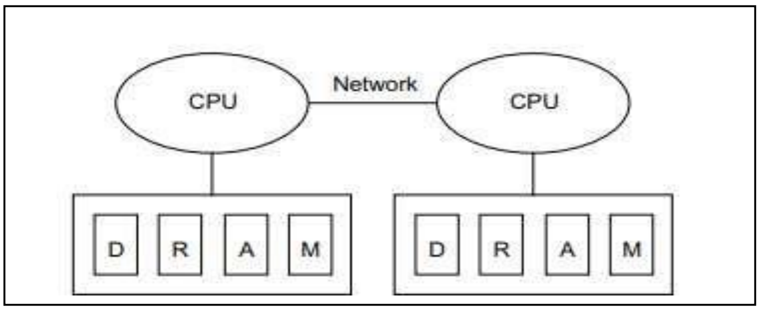
Software Models

Process-based parallelization: Message passing
Thread-based parallelization: Shared data via memory
Vectorization: Multiple operations with one instruction
Stream processing: Through specialized processors

Hardware Models

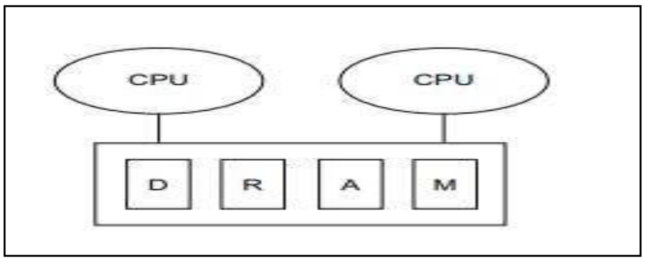
Distributed Memory Architecture: A Cross-Node Parallel Method**:**

Distributed Memory Architecture, also known as distributed memory parallelism, is a parallel computing method where multiple processors or nodes in a cluster have their own private memory. These nodes are connected via a network, and they communicate and coordinate with each other by passing messages. In this architecture, each node operates independently and has its own local memory, and data sharing is achieved explicitly through message passing.In the context of



distributed memory architecture, a "cross-node parallel method" refers to parallel processing techniques that involve distributing tasks across multiple nodes in a cluster. Each node works on its subset of the data or a specific portion of the computation. Communication and coordination between nodes are essential, as tasks often depend on results or data computed on other nodes.

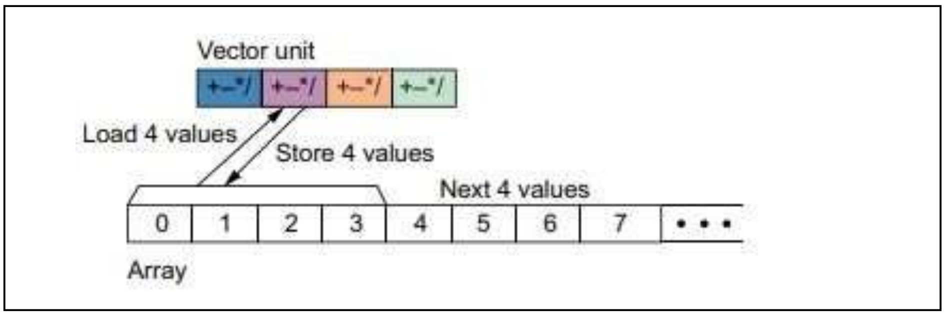
Shared Memory Architecture: An On-Node Parallel Method



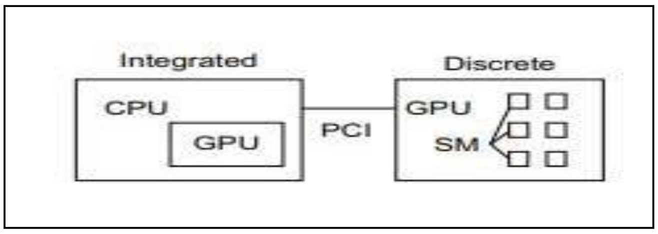
In shared memory architecture, multiple processors or cores share a single, unified memory space. This shared memory can be accessed and modified by any processor within the system. On-node parallelism, within the context of shared memory architecture, refers to parallel processing techniques that occur on a single computing node. In this approach, multiple threads or processes run concurrently on the same node, accessing shared memory to perform computations.

Vector Units: Multiple Operations With One Instruction Vector units, also known as vector processors, are specialized hardware units that can perform multiple operations with a single instruction. These units are designed to process vectors, which are arrays of data elements, simultaneously. Vector processing is particularly useful in scenarios where the same operation needs to be performed on a large set of data elements.

Vector processing example with four array elements operated on simultaneously



Accelerator Device: A Special-Purpose Add-On Processor



GPUs come in two varieties: integrated and discrete. Discrete or dedicated GPUs typically have a large number of streaming multiprocessors and their own DRAM. Accessing data ona discrete GPU requires communication over a PCI bus

An accelerator device, often referred to as an accelerator, is a specialized hardware component (GPU)designed to perform specific types of computational tasks or workloads efficiently. Accelerators are typically used in conjunction with a central processing unit (CPU) and are especially well-suited for workloads that can benefit from parallel processing and offloading certain tasks from the CPU. Accelerators are sometimes called "add-on processors" because they augment the processing capabilities of a system.

General Heterogeneous Parallel Architecture Model

Now let's combine all of these different hardware architectures into one model . Two nodes, each with two CPUs, share the same DRAM memory. Each CPU is a dual-core processor with an integrated GPU. A discrete GPU on the PCI bus also attaches to one of the CPUs. Though the CPUs share main memory, these are commonly in different Non-Uniform Memory Access (NUMA) regions. This means that accessing the second CPU's memory is more expensive than getting at it's own memory

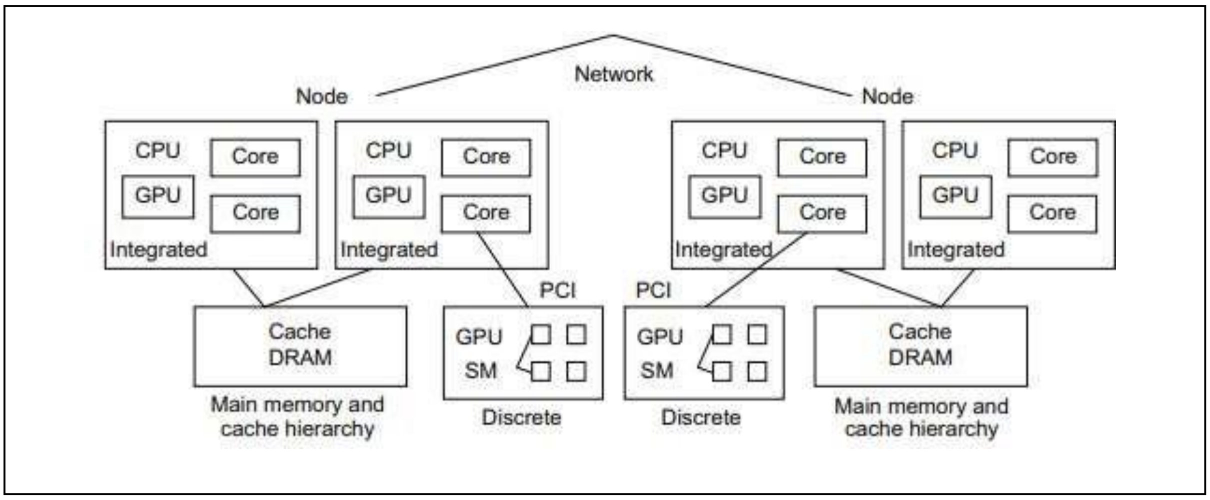
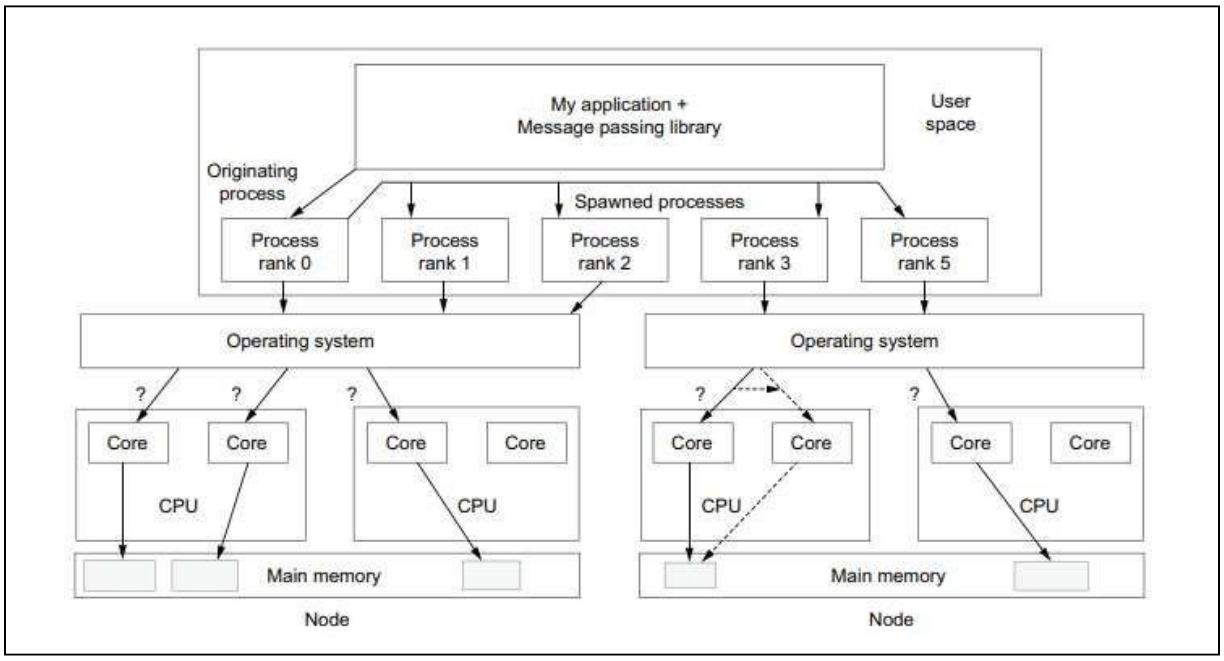


Fig 5: A general heterogeneous parallel architecture model consisting of two nodes connected by a network.Each node has a multi-core CPU with an integrated and discrete GPU and some memory (DRAM).

Software Models

- **Process-based parallelization**: Message passing Process-based parallelization, particularly through message passing, is a common approach in parallel computing. It involves dividing a task into multiple processes or threads that run independently on separate computing nodes or cores. These processes communicate and coordinate with each other by sending and receiving messages. Message passing is a method of inter-process communication where data and instructions are exchanged between processes to synchronize and share information. This approach is widely used in distributed memory systems, such as clusters and supercomputers.

Fig 6 : The message passing library spawns processes. The OS places the processes on the



cores of two nodes.The question marks indicate that the OS controls the placement of the processes and can move these during run time as indicated by the dashed arrows. The OS also allocates memory for each process from the node's main memory

- **Thread-based parallelization**: Shared data via memory

Thread-based parallelization involves dividing a task into multiple threads that share the same memory space within a single process. These threads can run concurrently on multiple CPU cores, and they communicate and coordinate by accessing shared data in the shared memory. This approach is commonly used in multi-core processors and symmetric multiprocessing (SMP) systems.

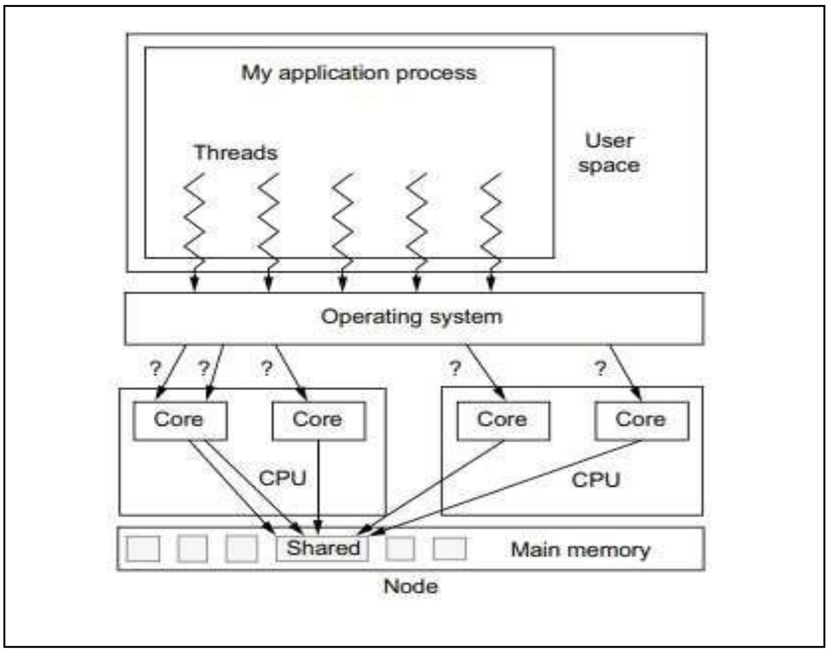


Fig 7:The application process in a thread-based approach to parallelization spawns threads. The threads are restricted to the node's domain. The question marks show that the OS decides where to place thethreads. Some memory is shared between threads.

- **Vectorization**: Multiple operations with one instruction

Vectorization is a parallel computing technique that enables processors to perform multiple operations with a single instruction. It takes advantage of SIMD (Single Instruction, Multiple Data) capabilities found in modern processors, including CPUs and GPUs. SIMD allows a single instruction to operate on multiple data elements simultaneously, which can significantly accelerate computations involving large datasets.

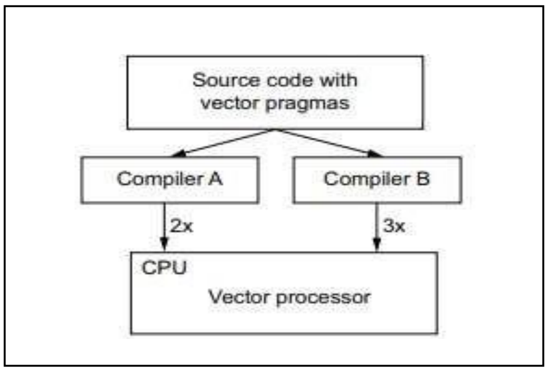


Fig 8 :Vector instructions in source code returning different performance levels from compilers

- **Stream processing**: Through specialized processors

Stream processing, often referred to as stream computing or data stream processing, is a computing paradigm where data is continuously processed as it is generated or ingested, rather than being stored in traditional databases or file systems. Stream processing is particularly useful for handling large volumes of real-time data from various sources, such as sensors, social media, financial transactions, and IoT devices. Specialized processors designed for stream processing accelerate the analysis and manipulation of data streams, ensuring timely and efficient processing

In the stream processing approach, data and compute kernel are offloaded to the GPU and its streaming multiprocessors. Processed data, or output, transfers back to the CPU for file IO or other work

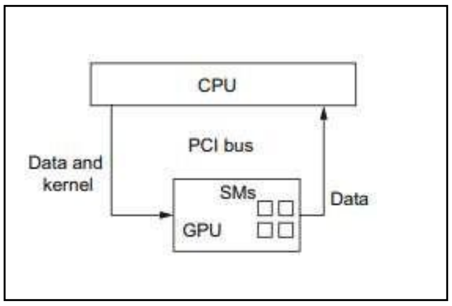


Fig 9 : Stream Processing Through Specialized Processors

UNIT-2

TOPIC 2

Fundamental laws

Fundamental laws in parallel computing, such as Amdahl's Law and Gustafson's Law, are essential for understanding the limitations and possibilities of parallel processing. These laws provide valuable insights into how the speedup of a parallel algorithm is affected by various factors.

What is Speedup?

Speedup in parallel computing refers to the performance improvement achieved by using multiple processors or computing resources to solve a problem compared to using a single processor. It is a measure of how much faster a parallel algorithm or system can complete a task compared to a serial (single-processor) implementation of the same task. Speedup is a crucial metric for evaluating the effectiveness of parallel computing systems.

The speedup (S) can be calculated using the following formula:

$S = T_{\text{serial}} / T_{\text{parallel}}$

Where:

- Tserial is the execution time of the task using a single processor (serial execution time). Tparallel is the execution time of the task using multiple processors (parallel execution time).

A speedup value greater than 1 indicates that the parallel implementation is faster than the serial implementation. Ideally, in a perfectly parallelizable task, doubling the number of processors would ideally halve the execution time, resulting in a speedup of 2. However, achieving perfect linear speedup is rare in real-world scenarios due to factors such as communication overhead, load balancing issues, and synchronization constraints between processors.

Amdahl's Law is a fundamental principle in parallel computing that expresses the potential speedup of a parallel algorithm as a function of the proportion of the algorithm that can be parallelized. It was formulated by Gene Amdahl in 1967 and is represented by the following formula:

$$SpeedUp(N) = \frac{1}{S + \frac{P}{N}}$$

Where:

- **Speedup** is the improvement in performance achieved by parallelizing a computation compared to executing it sequentially.
- **P** is the proportion of the algorithm that can be parallelized (a value between 0 and 1).
- S is the serial fraction
- N – no.of processors/nodes/cores

Amdahl's Law highlights the limitations of parallel computing. It states that the speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program. In other words, if only a portion of a program can be parallelized (the rest being inherently sequential), then no matter how many processors are added, there will always be a limit to the speedup that can be achieved.

For example, if 90% of a program can be parallelized (P = 0.9) and the parallel portion runs on 5 processors, the maximum speedup that can be achieved according to Amdahl's Law is:

Speedup=1/(0.1+(0.9/5))=3.57

In this case, even though 90% of the program can be parallelized and runs on 5 processors, the maximum speedup achievable is approximately 3.57 times faster compared to the sequential execution due to the presence of the 10% sequential portion.

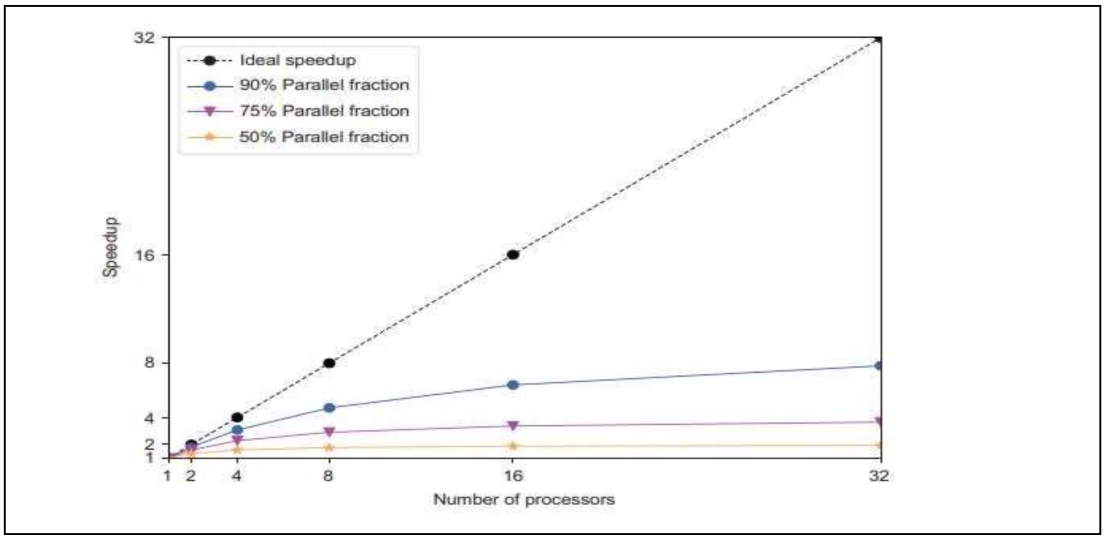


Fig : Speedup for a fixed-size problem according to Amdahl's Law is shown as a function of the number of processors. Lines show ideal speedup when 100% of an algorithm is parallelized, and for 90%, 75%, and 50%. Amdahl's Law states that speedup is limited by the fractions ofcode that remain serial.

Gustafson's Law(Gustafson Barsis Law), formulated by John L. Gustafson, provides a different perspective on parallel computing compared to Amdahl's Law. Unlike Amdahl's Law, which focuses on fixed problem sizes, Gustafson's Law takes into account varying problem sizes. The basic idea behind Gustafson's Law is that as the size of the problem increases, the impact of the parallelizable portion of the program becomes more significant, leading to better scalability. In other words, with larger problem sizes, parallel systems can achieve higher levels of speedup.

The formula for Gustafson's Law is as follows:

SpeedUp(N) = N – S * (N – 1) where N is the number of processors, and S is the serial fraction

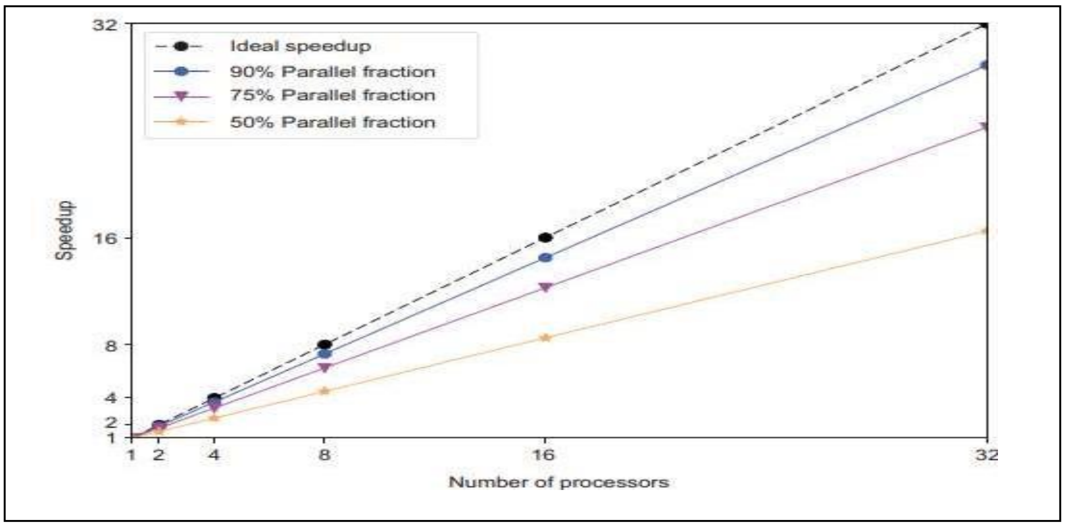


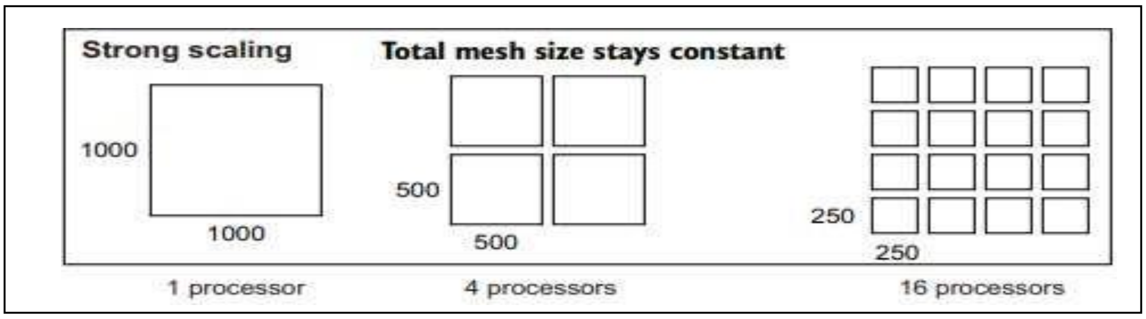
Fig :Speedup for when the size of a problem grows with the number of available processors accordingto Gustafson-Barsis's Law is shown as a function of the number of processors. Lines show ideal speedup when 100% of an algorithm is parallelized, and for 90%, 75%, and 50%

Strong scaling and weak scaling are two different metrics used to evaluate the performance of parallel computing systems, and they provide insights into how well a parallel algorithm or application can handle an increasing workload or an increasing number of processors. Here's a comparison of strong scaling and weak scaling:

Strong Scaling:

Definition: Strong scaling measures how the execution time of a fixed problem size decreases as the number of processors increases. In other words, it assesses how well a parallel system performs when the size of the problem remains constant, but the number of processors used to solve the problem increases.

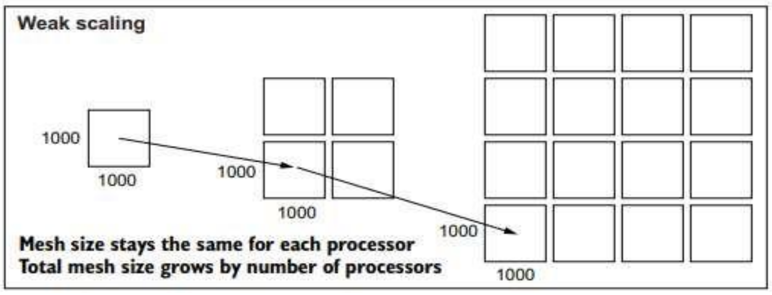
Objective: The goal of strong scaling is to reduce the execution time for a fixed problem size by utilizing more processors. It aims to speed up the solution of a specific problem.



2. Weak Scaling:

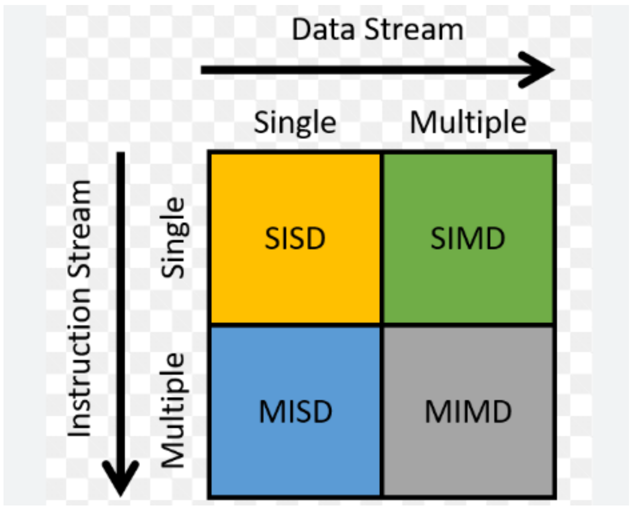
Definition: Weak scaling measures how the execution time changes as both the problem size and the number of processors increase proportionally. In other words, it assesses how well a parallel system can handle larger workloads by adding more processors as the problem size grows.

Objective: The goal of weak scaling is to maintain a constant workload per processor as the size of the problem and the number of processors increase. It aims to solve larger problems in approximately the same amount of time per processor.


Weak scaling
1000
1000
1000
1000
1000
1000
Mesh size stays the same for each processor
Total mesh size grows by number of processors

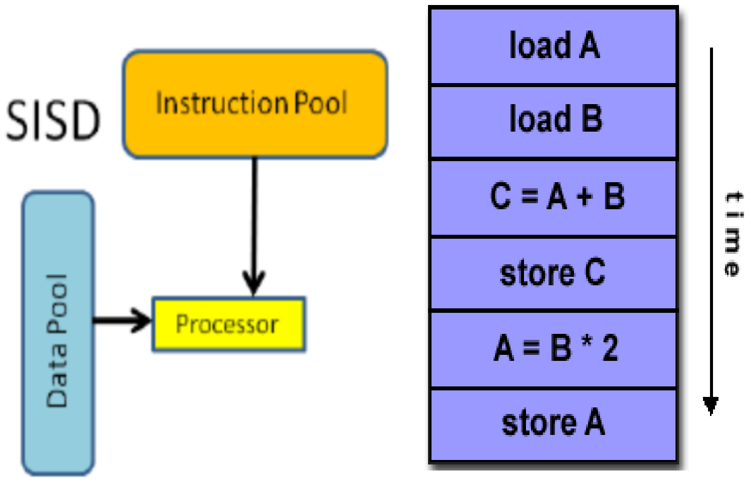
Parallel Approaches (Flynn’s Classification)

Flynn's classification is essential in the field of parallel computing because it provides a framework for understanding and categorizing different types of computer architectures based on the number of instruction streams and data streams. This classification is named after Michael J. Flynn, who introduced it in 1966. Flynn’s taxonomy is a useful tool for understanding different types of computer architectures and their strengths and weaknesses.

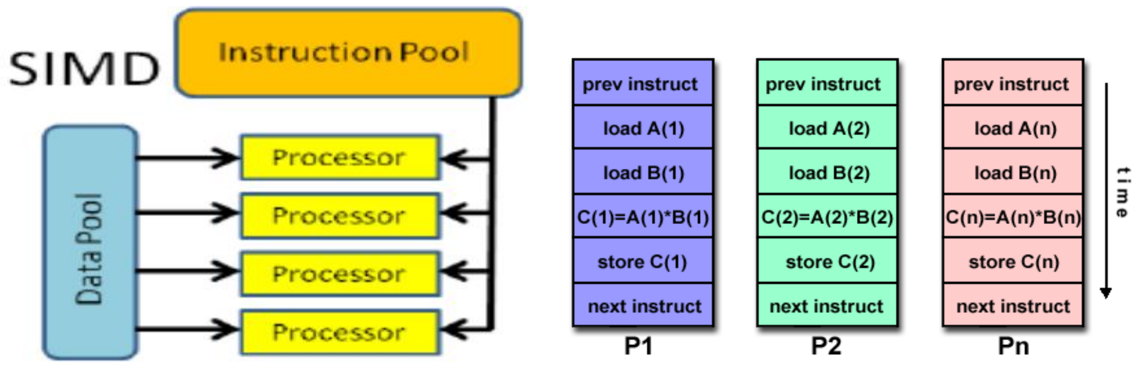


The taxonomy highlights the importance of parallelism in modern computing and shows how different types of parallelism can be exploited to improve performance.It helps in designing and analyzing parallel processing systems

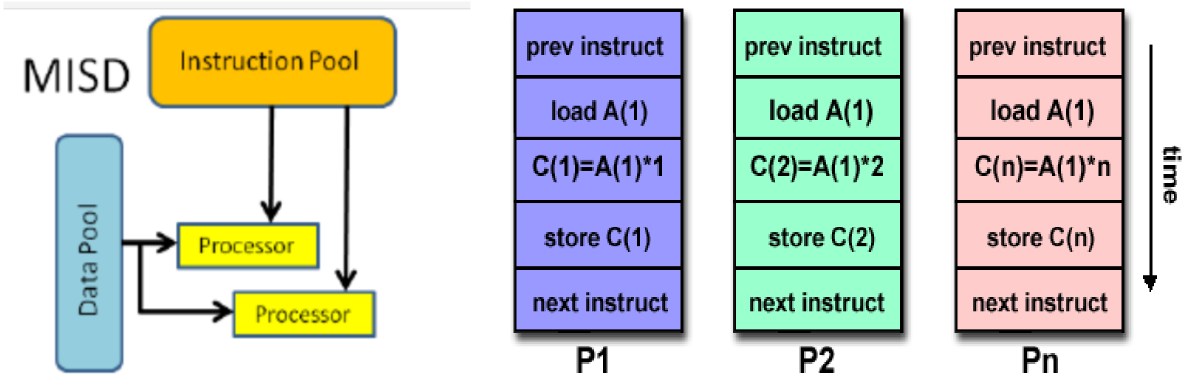
1. Single Instruction Single Data (**SISD**): In a SISD architecture, there is a single processor that executes a single instruction stream and operates on a single data stream. This is the simplest type of computer architecture and is used in most traditional computers.



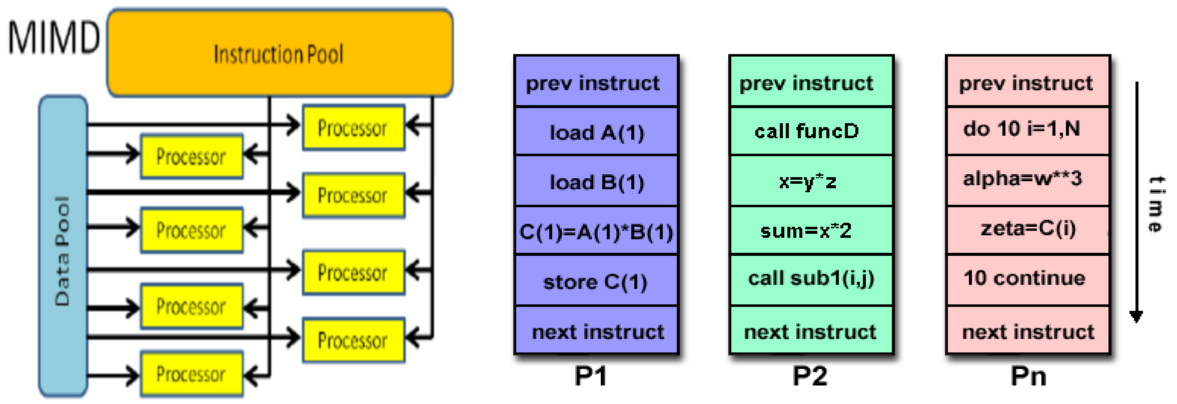
2. Single Instruction Multiple Data (**SIMD**): In a SIMD architecture, there is a single processor that executes the same instruction on multiple data streams in parallel. This type of architecture is used in applications such as image and signal processing.



3. Multiple Instruction Single Data (**MISD**): In a MISD architecture, multiple processors execute different instructions on the same data stream. This type of architecture is not commonly used in practice, as it is difficult to find applications that can be decomposed into independent instruction streams.



4. Multiple Instruction Multiple Data (**MIMD**): In a MIMD architecture, multiple processors execute different instructions on different data streams. This type of architecture is used in distributed computing, parallel processing, and other high-performance computing applications.



UNIT-1

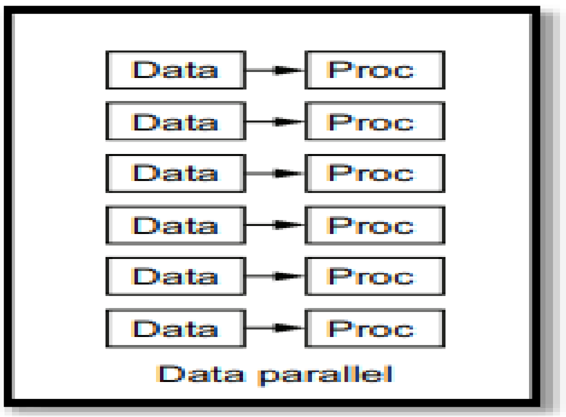
TOPIC 3

Parallel strategies

Parallel strategies" typically refer to techniques and methods for parallel processing, which is the simultaneous execution of multiple tasks or processes to improve the efficiency and performance of a computer system. Parallel strategies are commonly used in various computing domains, such as high-performance computing and distributed systems, to speed up computations and handle large volumes of data. Here are some common parallel strategies:

Data Parallel Approach

Data parallelism involves performing the same operation on multiple data elements simultaneously. This strategy is often used in applications where the same operation can be applied to different pieces of data independently.



Scenario: Imagine you're running a data analysis task on a large dataset of customer reviews for a product. Your goal is to perform sentiment analysis on each review to determine if it's positive, negative, or neutral. The sentiment analysis process is computationally intensive, and you want to speed it up using data parallelism.

Data Parallelism in Sentiment Analysis:

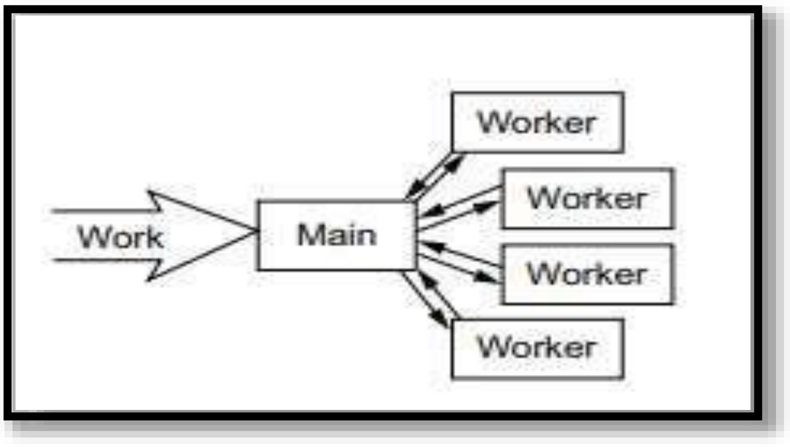
1. **Data Preparation**: You have a dataset of 1,000,000 customer reviews. To apply data parallelism, you divide this dataset into smaller, non-overlapping subsets. Let's say you split it into four subsets, each containing 250,000 reviews.
2. **Parallel Processing**: You have a sentiment analysis model that can analyze reviews. You set up four separate processing units (e.g., CPU cores or machines in a cluster), each responsible for analyzing one subset of reviews. Each processing unit loads its assigned subset of data.

3. **Analysis**: Each processing unit applies the sentiment analysis model to its subset of reviews independently and simultaneously. For instance:
    - Processing Unit 1 analyzes reviews 1 to 250,000.
    - Processing Unit 2 analyzes reviews 250,001 to 500,000.
    - Processing Unit 3 analyzes reviews 500,001 to 750,000.
    - Processing Unit 4 analyzes reviews 750,001 to 1,000,000.
4. **Aggregation**: As each processing unit finishes its analysis, it generates results, such as counts of positive, negative, and neutral reviews within its subset. These results are temporarily stored.
5. **Combining Results**: After all processing units have completed their work, you combine the results. You sum up the counts from each processing unit to get the overall sentiment analysis results for the entire dataset.

Task Parallelism(Main-Worker Approach)

Task parallelism involves executing multiple independent tasks or processes in parallel. Each task can perform different operations and may not necessarily operate on the same data. Task parallelism is common in applications where different tasks can be performed concurrently without dependencies between them.

In the main-worker approach, one processor schedules and distributes the tasks for all the workers, and each worker checks for the next work item as it returns the previous completed



task.

Example: Web Server Handling Requests

Consider a web server handling incoming HTTP requests. Each incoming request is an independent task that can be processed concurrently. The tasks include tasks like parsing the request, querying the database, and generating the response. In a task parallelism scenario:

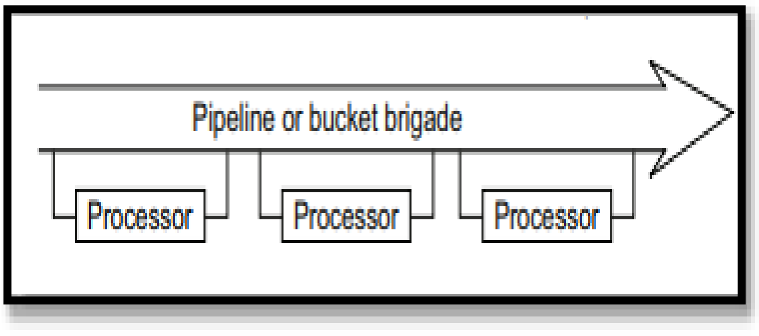
1. **Task 1: Parsing Request**
    - This task involves parsing the incoming HTTP request to extract information like the requested URL, parameters, and headers.
2. **Task 2: Database Query**
    - This task involves querying a database to fetch data related to the request, such as user information or product details.
3. **Task 3: Generating Response**
    - This task involves generating an HTML response based on the parsed request and data retrieved from the database.

In a task parallelism setup, these tasks can be executed concurrently by multiple threads or processes, allowing the server to handle multiple incoming requests simultaneously without waiting for one task to complete before starting the next.

Bucket-brigade Parallelism:

A **bucket brigade** is a method of manually transporting items or materials from one location to another by forming a line of people, each of whom carries an item and passes it to the next person. This technique is similar to how buckets of water might be passed along a line of people to put out a fire, which is where the term "bucket brigade" originated.

In parallel computing, the concept of bucket-brigade parallelism involves breaking down a task into smaller subtasks, where each subtask is processed independently and passed to the next processing unit for further computation. This technique allows for efficient parallel processing of tasks and is often used in scenarios where tasks can be divided into smaller, manageable parts.



Example: Manufacturing Assembly Line

Let's say we have a manufacturing assembly line for producing smartphones. The assembly line consists of three stages: A, B, and C. Each stage represents a specific task in the smartphone assembly process.

1. **Stage A - Component Assembly**:
    - Worker A assembles the basic components of the smartphone, such as the circuit board, battery, and display. Once Worker A finishes assembling a smartphone, it passes it to Stage B.
2. **Stage B - Software Installation**:
    - Worker B installs the operating system and necessary software onto the smartphone assembled by Worker A. After software installation, the smartphone is passed to Stage C.
3. **Stage C - Quality Control and Packaging**:
    - Worker C checks the smartphone for quality control, ensuring that all components are working correctly and the software is functioning as intended. If the smartphone passes quality control, it is packaged and prepared for shipment.

In this example, each stage (A, B, and C) represents a processing step, similar to the stages in a bucket-brigade parallelism scenario.

Parallel speedup versus comparative speedups.

Parallel speedup and comparative speedup are two different metrics used to evaluate the performance improvement achieved by parallel processing.

Parallel speedup measures how much faster a parallel algorithm runs compared to its sequential (single-processor) counterpart. It quantifies the performance improvement gained by using multiple processing units in parallel. Parallel speedup is calculated using the following formula:

Parallel Speedup=Sequential Execution Time/Parallel Execution

In this formula:

- **Sequential Execution Time** is the time taken by the algorithm to execute sequentially on a single processor.
- **Parallel Execution Time** is the time taken by the parallel algorithm to execute on multiple processors.

Comparative speedup: Comparative speedup is between architectures. This is usually a performance comparison between two parallel implementations or other comparison between reasonably constrained sets of hardware. For example, it may be between a parallel MPI implementation on all the cores of the node of a computer versus the GPU(s) on a node

UNIT 1

TOPIC 4

Performance limits and profiling:

Applications Potential Performance Limits & determine your hardware capabilities

In parallel processing, understanding performance limits and profiling the application are crucial steps to optimize the execution of parallel programs.

Performance limits refer to the maximum achievable performance of a computing system or application under specific conditions. These limits are determined by various factors and constraints and play a crucial role in understanding the capabilities and limitations of a system. Understanding these performance limits is essential for designing efficient algorithms, optimizing software, and choosing appropriate hardware configurations. It also guides researchers and engineers in developing new technologies to overcome existing limitations and improve overall computing performance.

Profiling tools are used to gather detailed information about the behavior of a parallel program. By understanding performance limits, utilizing profiling tools, and optimizing the code based on the profiling results, developers can enhance the efficiency of parallel applications, leading to improved speedup and overall performance.

Application's potential performance limits

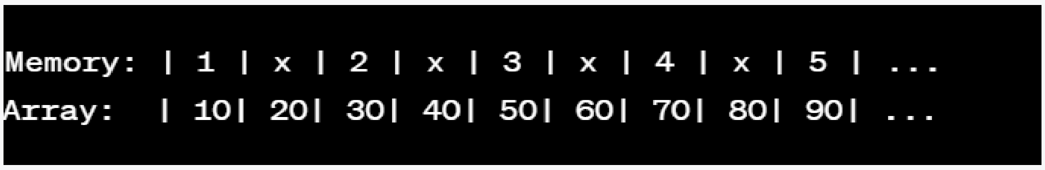
- Flops (floating-point operations)
- Ops (operations) that include all types of computer instructions
- Memory bandwidth: Rate at which the data is transferred
- Memory latency: Time required for the first byte or word of data to be transferred
- Instruction queue (instruction cache)
- Networks
- Disk
- Machine Balance: Number of flops executed /memory bandwidth
- Arithmetic Intensity: Number of flops executed per memory operation

All of these limitations can be divided into two major categories : Speeds are how fastoperations can be done. It includes all types of computer operations. But to be able to do the operations, you must get the data there. This is where feeds come in. Feeds include the memory bandwidth through the cache hierarchy, as well as network and disk bandwidth.

For many applications, the memory bandwidth limit can be difficult especially dealing with non-contiguous bandwidth.It is also known as strided memory access or non-contiguous memory access, refers to the manner in which data elements are accessed in memory. In contrast to contiguous memory access, where elements are stored in consecutive memory locations, non-contiguous memory access involves accessing elements that are not stored sequentially in memory.

Non-Contiguous Memory Access:

Now, consider a situation where the array elements are scattered in memory with a stride of 2. This is a non-contiguous memory access pattern:



```
Memory: | 1 | x | 2 | x | 3 | x | 4 | x | 5 | ...
Array:  | 10| 20| 30| 40| 50| 60| 70| 80| 90| ...
```

In this case, accessing every second element (stride = 2) would mean accessing memory locations 1, 2, 3, 4, 5, etc., but the elements are not stored sequentially.

When your program needs to access such non-contiguous elements, it may lead to inefficiencies due to increased cache misses and a higher likelihood of accessing data from main memory rather than the faster cache memory.

Determine your Hardware capabilities:

To determine the Performance of hardware the following metrics are used:

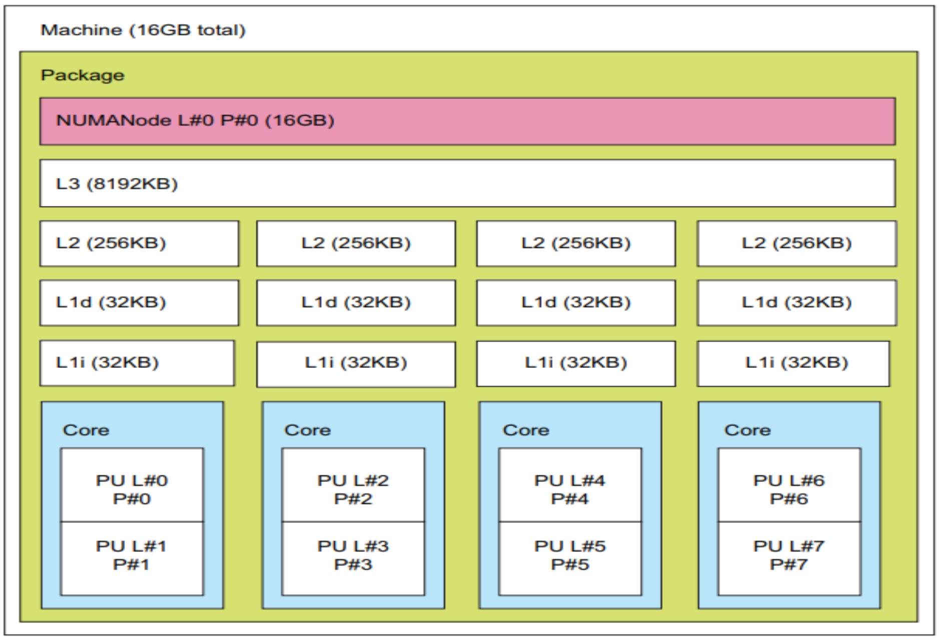
- The rate at which floating-point operations can be executed (FLOPs/s)
- The rate at which data can be moved between various levels of memory (GB/s)
- The rate at which energy is used by your application (Watts)

In determining hardware performance and calculating the metrics, we use a mixture of theoretical and empirical measurements.

Theoretical measurements provide an upper limit to what a system can achieve. For instance, in parallel computing, theoretical analysis can reveal the maximum speedup or efficiency that a parallel algorithm can achieve in an ideal scenario**.**

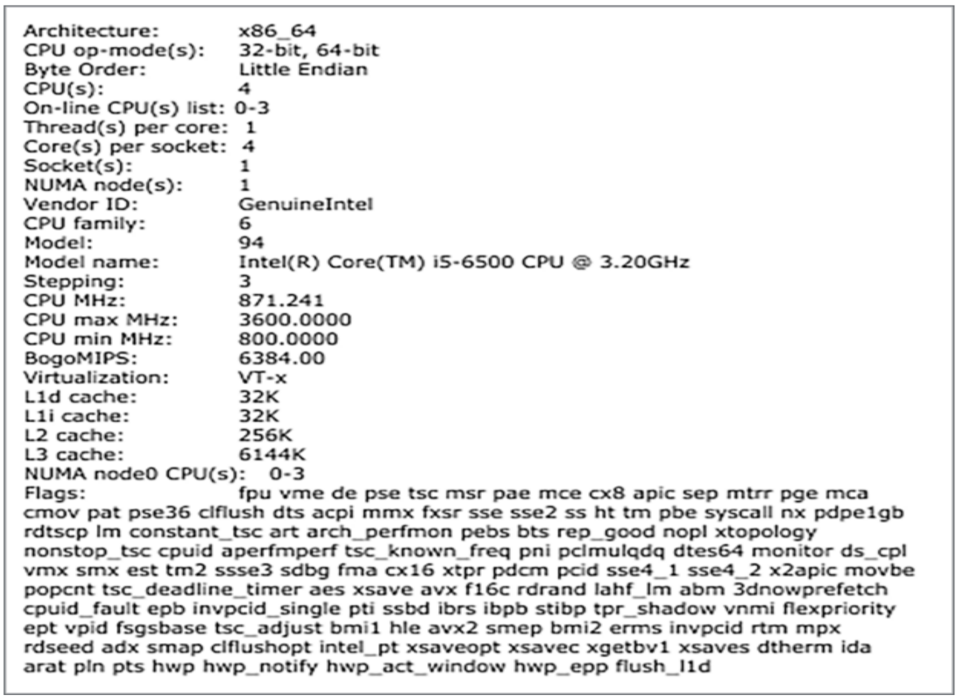
Real-world Validation is done by **Empirical measurements**, they provide concrete evidence of how a system performs under real-world conditions, accounting for various factors like I/O operations, network latency, and concurrency issues.

One of the best tools for understanding the hardware you run is the lstopo program(graphical view) and lscpu for text view. lstopo is bundled with the hwloc package that comes with nearly every MPI distribution. This command outputs a graphical view of the hardware on your system. Figure below shows the output for a Mac laptop in **graphical view.**



Text view

The information from the lspci command and the /proc/cpuinfo file helps to determine the number of processors, the processor model, the cache sizes, and the clock frequency for the system



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                4
On-line CPU(s) list: 0-3
Thread(s) per core:  1
Core(s) per socket:  4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 94
Model name:            Intel(R) Core(TM) i5-6500 CPU @ 3.20GHz
Stepping:              3
CPU MHz:               871.241
CPU max MHz:           3600.0000
CPU min MHz:           800.0000
BogoMIPS:              6384.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              6144K
NUMA node0 CPU(s):   0-3
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology
nonstop_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq dtes64 monitor ds_cpl
vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4_1 sse4_2 x2apic movbe
popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch
cpuid_fault epb invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi flexpriority
ept vpid fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx
rdseed adx smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm ida
arat pln pts hwp hwp_notify hwp_act_window hwp_epp flush_l1d
```

Calculating theoretical maximum flops

Theoretical FLOPS=Number of Cores×Clock Speed×FLOPs per Cycle per Core

Where:

- **Number of Cores:** This represents the total number of processor cores in the computing system.
- **Clock Speed:** This indicates the clock speed of each core in the system, typically measured in Hertz (Hz) or Gigahertz (GHz). It represents the number of cycles the processor can execute per second.
- **FLOPs per Cycle per Core:** This signifies the number of floating-point operations a core can perform in a single clock cycle. Modern processors often perform multiple FLOPs per cycle due to features like SIMD (Single Instruction, Multiple Data) operations.

For example, let's consider a system with 4 cores, each operating at 3.0 GHz, and capable of executing 4 FLOPs per cycle per core (assuming SIMD operations are utilized):

Theoretical FLOPS=4 cores×3.0 GHz×4 FLOPs per cycle per Core

Theoretical FLOPS=48 GFLOPS

The memory hierarchy and theoretical memory bandwidth

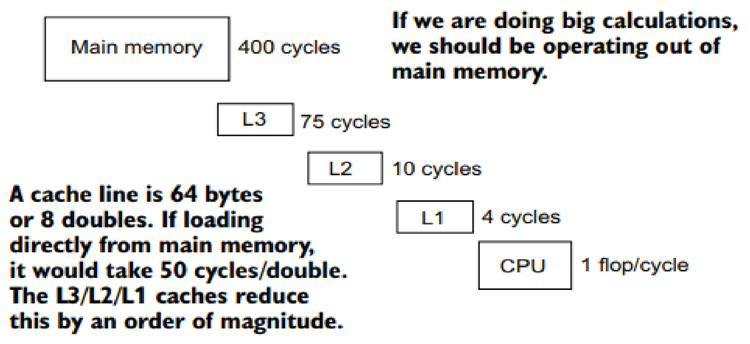

Main memory
400 cycles
If we are doing big calculations, we should be operating out of main memory.
L3
75 cycles
L2
10 cycles
L1
4 cycles
CPU
1 flop/cycle
A cache line is 64 bytes or 8 doubles. If loading directly from main memory, it would take 50 cycles/double. The L3/L2/L1 caches reduce this by an order of magnitude.

Figure 3.5 Memory hierarchy and access times. Memory is loaded into cache lines and stored at each level of the cache system for reuse.

We can calculate the theoretical memory bandwidth of the main memory using the memory chips specifications.

The general formula is B T = MTR × Mc × Tw × Ns = Data Transfer Rate × Memory Channels × Bytes Per Access × Sockets

Processors are installed in a socket on the motherboard. The motherboard is the main system board of the computer, and the socket is the location where the processor is inserted. Most motherboards are single-socket, where only one processor can be installed. Dual-socket motherboards are more common in high-performance computing systems. Two processors can be installed in a dual-socket motherboard, giving us more processing cores and more memory bandwidth.

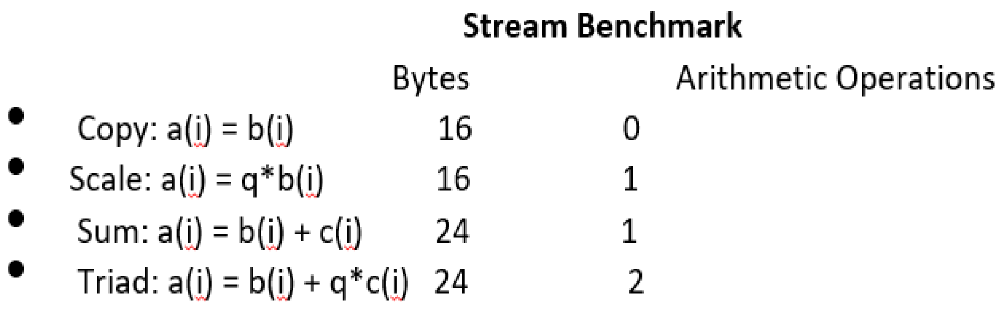
Empirical measurement of bandwidth and flop

The empirical bandwidth is the measurement of the fastest rate that memory can be loaded from main memory into the processor. If a single byte of memory is requested, it takes 1 cycle to retrieve it from a CPU register. If it is not in the CPU register, it comes from the L1 cache. If it is not in the L1 cache, the L1 cache loads it from L2 and so on to main memory. If it goes all the way to main memory, for a single byte of memory, it can take around 400 clock cycles. This time required for the first byte of data from each level of memory is called the memory latency.

Two different methods are used for measuring the bandwidth: **the STREAM Benchmark** and the **roofline model** measured by the Empirical Roofline Toolkit.

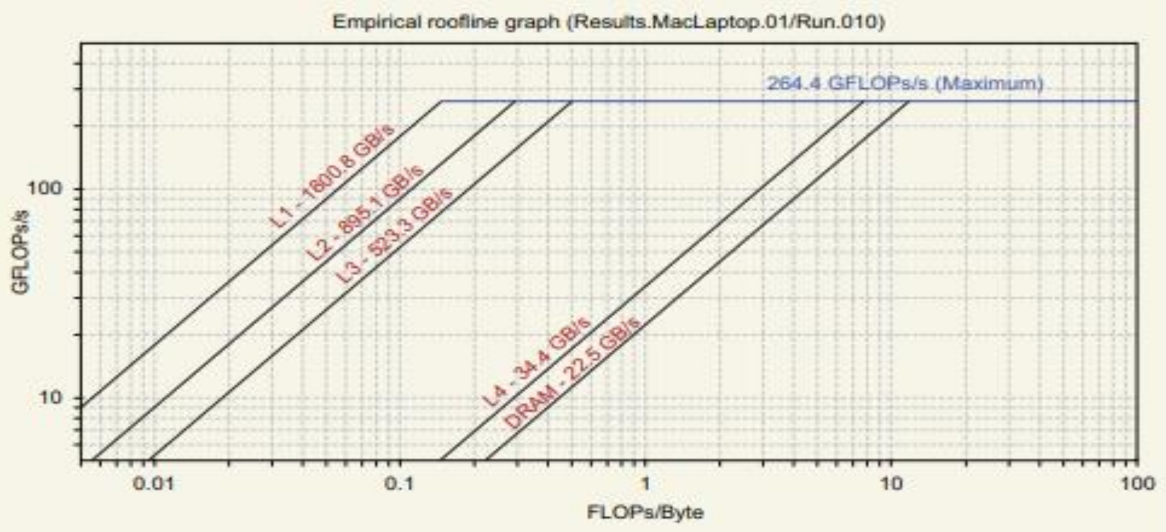
Key Differences:

- **Focus:** STREAM primarily focuses on memory bandwidth, providing quantitative measurements. In contrast, the Roofline Model provides a graphical representation of performance bottlenecks, considering both computational capabilities and memory bandwidth.
- **Representation:** STREAM results in a numerical measurement (memory bandwidth in bytes per second), while the Roofline Model is a graphical representation that helps visualize performance limitations.
- **Insights:** STREAM provides detailed insights into memory subsystem performance, whereas the Roofline Model offers a high-level overview of an application's performance efficiency concerning hardware constraints.



Stream Benchmark

	Bytes	Arithmetic Operations
• Copy: a(i) = b(i)	16	0
• Scale: a(i) = q*b(i)	16	1
• Sum: a(i) = b(i) + c(i)	24	1
• Triad: a(i) = b(i) + q*c(i)	24	2



Roofline for 2017 Mac Laptop shows the maximum FLOPs as a horizontal line, and the maximum bandwidth from various cache and memory levels as sloped lines.

Calculating the machine balance between flops and bandwidth

The machine balance is the flops divided by the memory bandwidth.

We can calculate both a theoretical machine balance (MB_T) and an empirical machine balance (MB_E) like so:

$MB_T = F_T / B_T$

$MB_E = F_E / B_E$

UNIT 1

TOPIC 5

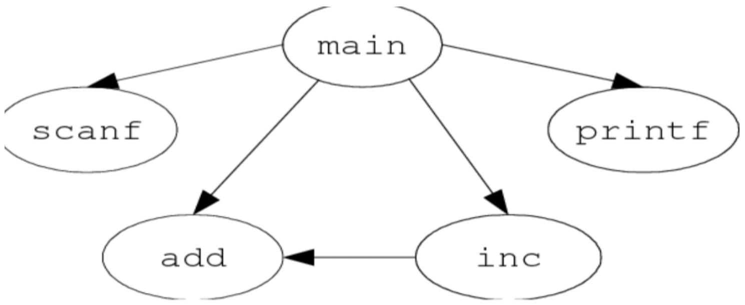
Characterizing your application: Profiling

Now that you have some sense of what performance you can get with the hardware, you need to determine what are the performance characteristics of your application. Additionally, you should develop an understanding of how different subroutines and functions depend on each other.

Profiling tools:

Using call graphs for hot-spot and dependency analysis

In the context of parallel programming, call graphs are diagrams that represent the calling relationships between different functions or methods in a parallel program. They illustrate how functions or tasks invoke each other and provide a visual representation of the program's control flow. Analyzing call graphs in parallel programming can provide valuable insights into the program's structure, dependencies, and potential performance optimizations. By analyzing these call graphs, developers can identify hot-spots—functions or tasks that consume a significant amount of computational time. Optimizing these hot-spots is essential for improving overall parallel program performance.



Empirical measurement of processor clock frequency and energy consumption

Empirical Measurement of Processor Clock Frequency:

1. **Profiling Tools:** Profiling tools like Intel VTune Profiler or AMD CodeXL can provide insights into various performance metrics, including processor clock frequency. These tools often offer visualizations and detailed reports for better analysis.
2. **Benchmarking Suites:** Benchmarking tools like SPEC CPU benchmarks or HPC Challenge benchmarks often include components that measure processor clock frequencies. Running these benchmarks can provide detailed information about the processor's performance characteristics.

Empirical Measurement of Energy Consumption:

1. **Power Measurement Tools:** Use power measurement tools and hardware devices to measure the power consumption of your system. Power meters and sensors can be

attached to the system to measure real-time power usage. Tools like Intel Power Gadget or Linux's `powerstat` can help measure power usage.

2. **Energy Profilers:** Some profiling tools, like Intel VTune Profiler, also offer energy profiling capabilities. They can provide insights into energy consumption patterns at different parts of your code. These tools often correlate energy consumption with specific functions or code regions.

Tracking memory during run time

Tracking memory usage during runtime in parallel computing is crucial for optimizing performance, detecting memory leaks, and ensuring efficient memory management. Several techniques and tools can help you monitor memory usage in parallel applications. Here are some approaches to tracking memory during runtime in parallel computing environments:

Profiling Tools:

1. **Valgrind Massif:** Valgrind is a powerful instrumentation framework. Massif, a Valgrind tool, can profile heap memory usage over time, showing memory consumption patterns. It's particularly useful for detecting memory leaks and understanding how memory usage evolves during program execution.
2. **Intel VTune Profiler:** VTune Profiler provides memory analysis capabilities, including memory usage tracking. It can analyze memory consumption at various levels, from individual functions to entire applications, in both serial and parallel contexts.
3. **OpenMP/MPI Memory Profilers:** Many parallel programming frameworks like OpenMP and MPI provide their memory profiling tools. For example, OpenMP has tools like Score-P, and MPI has memory profiling features integrated into MPI implementations.

UNIT-1

TOPIC-6

Parallel algorithms and patterns

A parallel algorithm is a step-by-step computational procedure or set of rules designed to be executed on parallel computing architectures. These algorithms are specifically crafted to take advantage of parallel processing capabilities, where multiple processors or cores can work together to solve a problem.

Parallel patterns are like reusable blueprints that help programmers apply proven methods to solve specific types of problems efficiently. These patterns guide the decomposition of tasks and data, providing a framework for creating effective parallel algorithms.

Example : Parallel Algorithm for Finding the Maximum Element:

Suppose you have a large array of numbers, and you want to find the maximum element using a parallel algorithm based on the "Divide and Conquer" pattern. In this example, the "Divide and Conquer" pattern is applied to find the maximum element in an array. The array is divided into smaller subarrays, and the maximum values of these subarrays are found in parallel. Finally, the maximum among these partial maximums is selected as the maximum element of the entire array.

Algorithm analysis for parallel computing applications

The goal of algorithm analysis is to compare different algorithms that are used to solve the same problem. One of the more traditional ways to evaluate algorithms is by looking at their algorithmic complexity.

DEFINITION: Algorithmic complexity is a measure of the number of operations that it would take to complete an algorithm. Algorithmic complexity is a property of the algorithm and is a measure of the amount of work or operations in the procedure.

Complexity is usually expressed in asymptotic notation. Using asymptotic notation, you can analyze and compare algorithms efficiently and make informed decisions when choosing the most suitable algorithm for a particular problem, taking into account both time and space complexity

The three main types of asymptotic notation are:

1. **Big O Notation (O-notation):**
    - Big O notation describes the upper bound or worst-case time complexity of an algorithm. It represents an approximation of how an algorithm's running time increases as the input size grows. It provides an upper limit on the number of basic operations an algorithm performs.
2. **Theta Notation (Θ-notation):**

- Theta notation provides a tight bound, expressing both the upper and lower bounds of an algorithm's time complexity. It characterizes the average-case behavior of an algorithm.

3.**Omega Notation (Ω-notation):**

- Omega notation describes the lower bound or best-case time complexity of an algorithm. It provides a way to express how quickly the algorithm can solve a problem in the most favorable circumstances.

Performance models versus algorithmic complexity

Performance models are broader and more practical in nature. They encompass various aspects of system performance, including algorithmic efficiency, but also consider factors related to specific hardware, software, and real-world scenarios.

Algorithmic complexity, often expressed using asymptotic notations like Big O, Theta, and Omega, focuses on analyzing the efficiency of algorithms in terms of their time and space requirements as a function of the input size. It provides a theoretical framework for characterizing how an algorithm's performance scales as the input size grows towards infinity.

Example: finding the sum of all elements in an array.

Algorithmic Complexity (Big O Notation):

- **Time Complexity:**$O(n)$ - Linear time complexity, where n is the size of the input array. The algorithm processes each element once.
- **Space Complexity:** $O(1)$ - Constant space complexity, as it uses only a few variables regardless of the input size.

Performance Model Considerations:

- **Hardware Differences:** Different computers might execute the same algorithm at different speeds due to variations in processor capabilities.
- **Compiler Optimizations:** Compilers can optimize the code differently, affecting the execution time.
- **Parallelization:** Divide the array into chunks and calculate the partial sums concurrently using parallel processing techniques, especially for large arrays.
- **Memory Optimization:** For very large arrays, consider memory-efficient data structures or algorithms to reduce memory usage.

Parallel algorithms

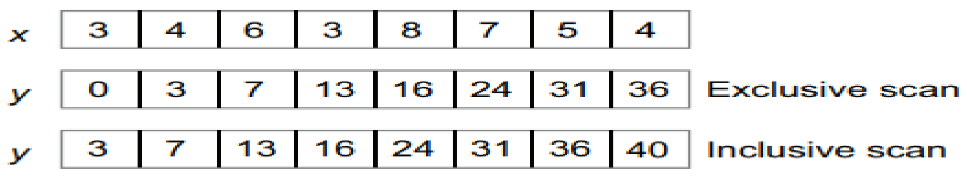
Parallel algorithms are designed to efficiently solve computational problems by utilizing multiple processing units (such as CPU cores, GPUs, or distributed computing nodes) simultaneously. They are crucial in high-performance computing (HPC) and parallel processing environments where large datasets and complex computations need to be handled efficiently. Parallel algorithms aim to break down tasks into smaller subtasks that can be processed independently and concurrently, leading to significant speedup in overall computation time. Here are some common types of parallel algorithms:

- **Parallel Merge Sort:** Divide the sorting task into smaller parts, sort them independently, and then merge the sorted parts in parallel.
- **Parallel QuickSort:** A parallel version of the quicksort algorithm that partitions the data and sorts partitions concurrently.

Prefix sum

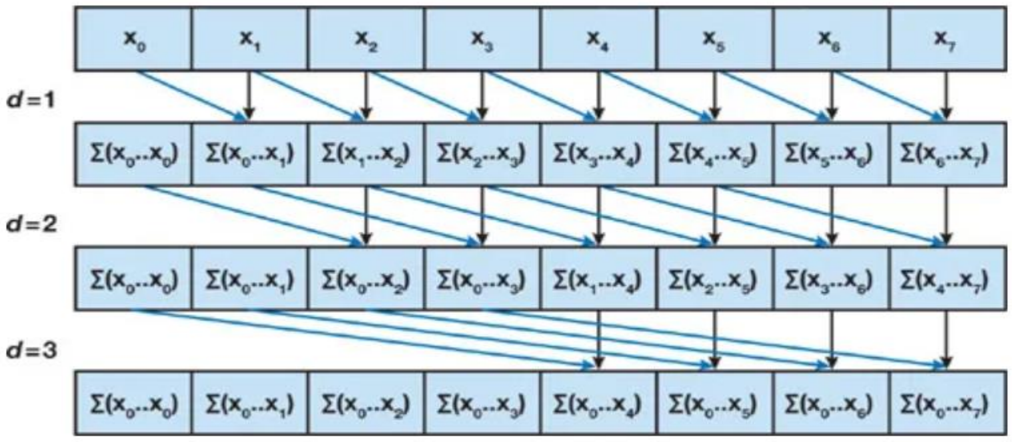
The prefix sum, also known as the scan operation, is a fundamental parallel pattern in computer science and parallel computing. Given an input array of elements, the prefix sum operation computes a new array where each element is the sum of all elements in the input array up to and including the corresponding element's position. There are two common types of prefix sum operations: exclusive and inclusive.

- **Exclusive Prefix Sum:** The result at each position does not include the element at that position.
  - Input: [a, b, c, d]
  - Output (Exclusive): [0, a, a+b, a+b+c]
- **Inclusive Prefix Sum:** The result at each position includes the element at that position.
  - Input: [a, b, c, d]
  - Output (Inclusive): [a, a+b, a+b+c, a+b+c+d]

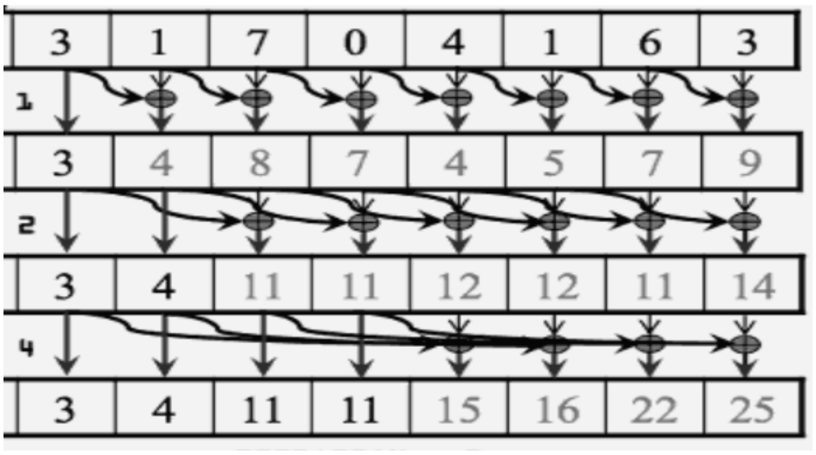


x	3	4	6	3	8	7	5	4	
y	0	3	7	13	16	24	31	36	Exclusive scan
y	3	7	13	16	24	31	36	40	Inclusive scan

Step-efficient parallel scan operation(Inclusive Prefix sum)

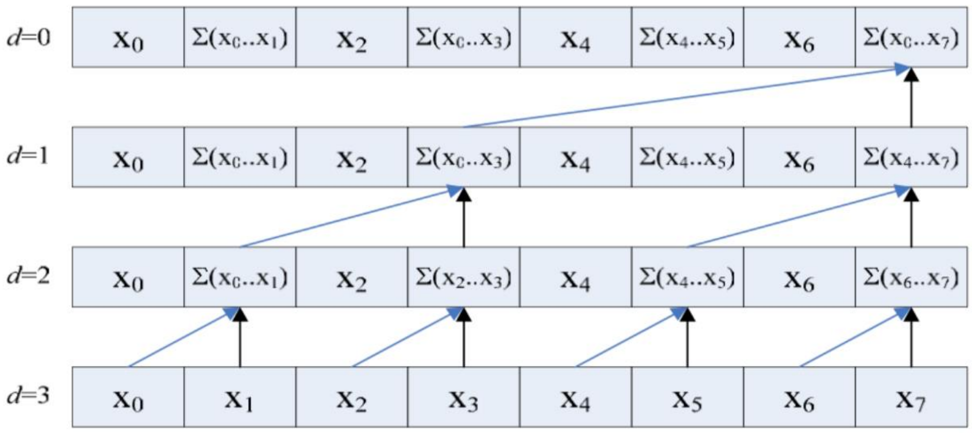


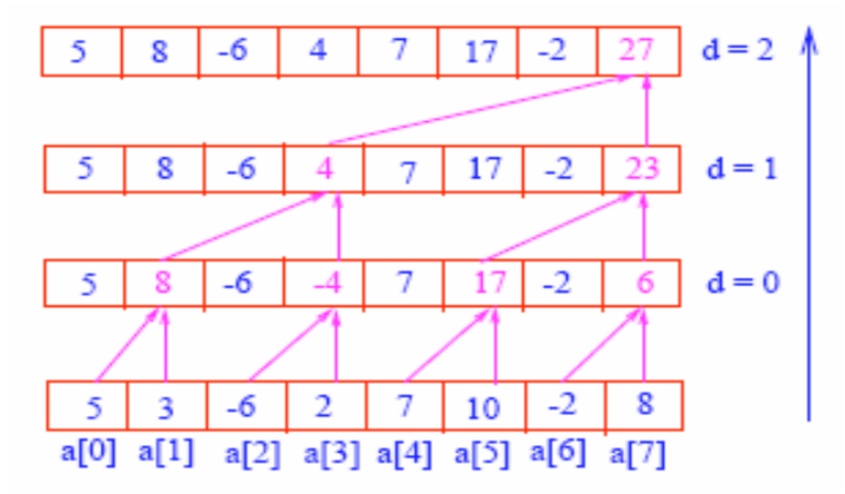
Parallel Prefix Scan



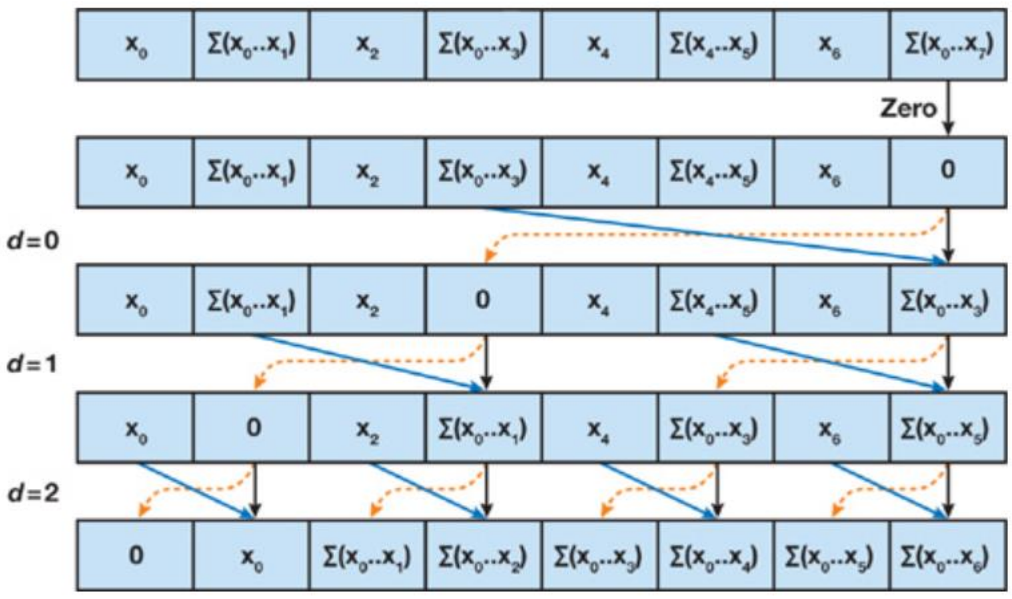
Work-efficient parallel scan operation(Exclusive Prefix sum)

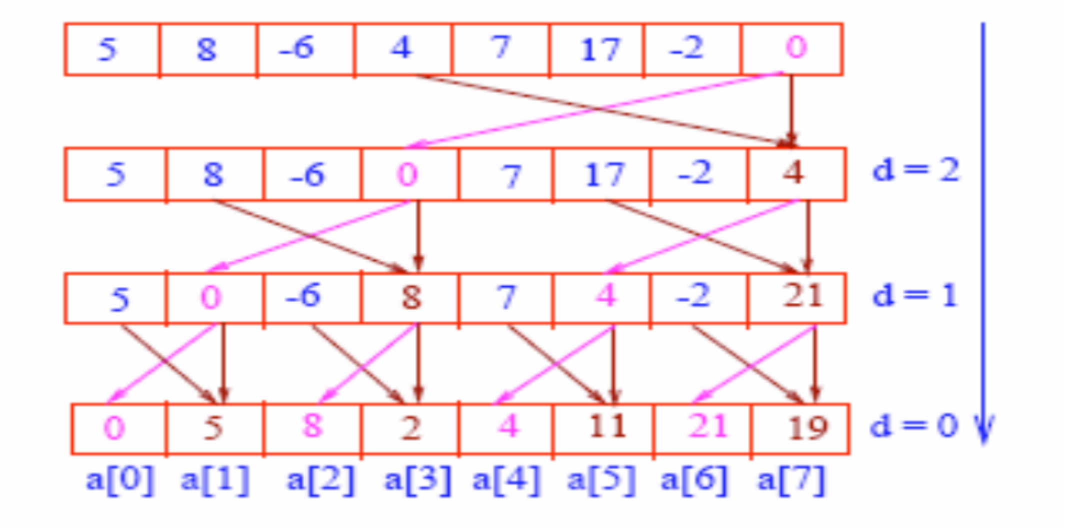
The work-efficient parallel scan operation uses two sweeps through the arrays. The first sweep is called an upsweep, though it is more of a right sweep.





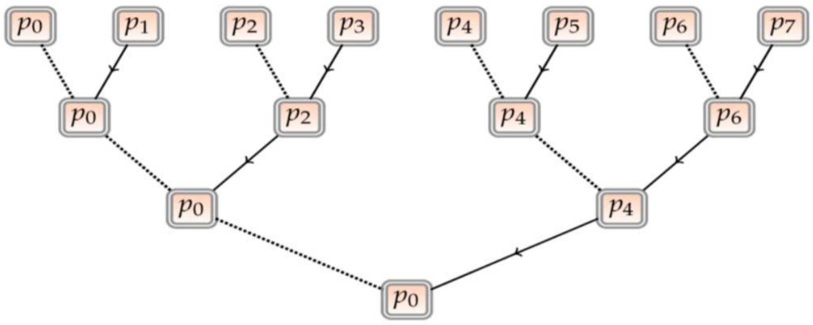
The second phase, known as the downsweep phase, is more of a left sweep. The output of upsweep is provided as input to downsweep.It starts by setting the last value to zero and then does another tree-based sweep to get the final result.





Parallel global sum

The "parallel global sum" refers to the problem of computing the sum of elements across multiple processors or nodes in a parallel or distributed computing environment.



Though the process is simple it has some problems like Changing the order of additions changes the answer in finite-precision arithmetic. This is problematic because a parallel calculation changes the order of the additions. There is even a worse case for additions of finite precision values when adding two values that are almost identical, but of different signs. This subtraction of one value from another when these are nearly the same causes a catastrophic cancellation.

Catastrophic cancellation occurs when the operands are subject to rounding errors.

For example, if there are two measures L1=253.5cm long and the otherL2 =252.5cm long

Approximations could come out to be

L1 = 254cm and L2 = 252cm L1-L2 = 2cm

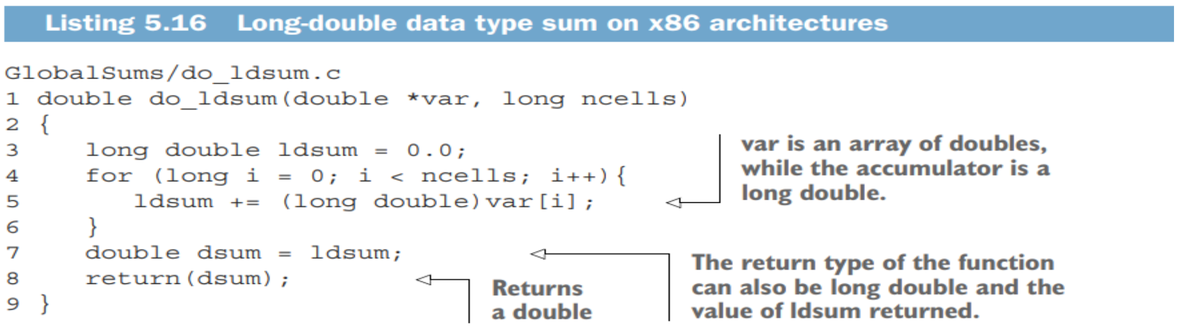
Actual difference is L1-L2 = 1 cm

There are several solutions for addressing the global sum . The list of possible techniques presented here includes

- Long-double data type
- Pairwise summation
- Kahan summation
- Knuth summation—uses same method of pairwise
- Quad-precision summation

Long-double data type

The easiest solution is to use the long-double data type on a x86 architecture. On this architecture, a long-double is implemented as an 80-bit floating-point number in hardware giving an extra 16-bits of precision. Unfortunately, this is not a portable technique



Listing 5.16 Long-double data type sum on x86 architectures

```
GlobalSums/do_ldsum.c
double do_ldsum(double *var, long ncells)
{
   long double ldsum = 0.0;
   for (long i = 0; i < ncells; i++){
      ldsum += (long double)var[i];
   }
   double dsum = ldsum;
   return(dsum);
}
```

var is an array of doubles, while the accumulator is a long double.

The return type of the function can also be long double and the value of ldsum returned.

Returns a double

Pairwise summation

Pairwise summation, also known as pairwise addition, is a method used to sum a sequence of numbers in a way that reduces the effects of numerical errors, particularly in floating-point arithmetic. This technique is commonly employed in scientific computing and numerical analysis to improve the accuracy of summation operations.

How Pairwise Summation Works:

1. **Pairing the Numbers:**

- Given a sequence of numbers, they are paired up. If there are an odd number of elements, one number is left unpaired.

2. **Pairwise Addition:**
    - Within each pair, the two numbers are added together to create intermediate sums.
3. **Summing the Intermediate Sums:**
    - The intermediate sums obtained from pairwise addition are then summed together using the same pairwise summation method.

- **Kahan summation**

Kahan summation, also known as compensated summation or Kahan summation algorithm, is a method used to reduce the numerical error that accumulates during the summation of a large number of floating-point values.

How Kahan Summation Works:

In standard floating-point summation, when adding a small number to a large number, the small number can be "lost" in the least significant bits of the large number, leading to a loss of precision. Kahan summation addresses this issue by using a compensation term to keep track of the lost precision.

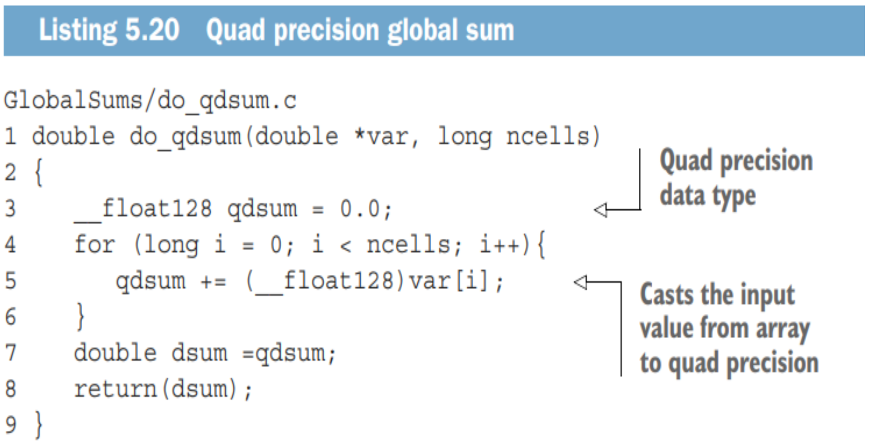
1. **Initialization:**
    - Initialize the sum and the compensation term to zero.
2. **Iterative Addition:**

- For each number to be added:
- Add the number to the current sum.
- Calculate the difference between the updated sum and the original sum (this difference is the lost precision).
- Add this difference to the compensation term.

3. **Final Result:**
    - The final result is the sum adjusted by the compensation term.

Quad-precision summation

Quad-precision summation refers to performing arithmetic operations with numbers represented in quadruple-precision floating-point format. In the IEEE 754 floating-point standard, quadruple-precision is a 128-bit data type, providing higher precision compared to single-precision (32-bit) and double-precision (64-bit) floating-point number.



Listing 5.20 Quad precision global sum

GlobalSums/do_qdsum.c

```
double do_qdsum(double *var, long ncells)
{
   __float128 qdsum = 0.0;
   for (long i = 0; i < ncells; i++){
      qdsum += (__float128)var[i];
   }
   double dsum =qdsum;
   return(dsum);
}
```