FIT3143

Parallel Computing

Week 1: Introduction to Parallel Computing

Vishnu Monn and ABM Russel

Unit Topics

Week	Lecture Topic	Tutorial	Laboratory	Remarks
1	Introduction to Parallel Computing and Distributed Systems	None	Lab Week 01 - Introduction to Linux & Setting up VM (Not assessed)	Group formation for lab activities and assignments (Two students per group)
2	Parallel computing on shared memory (POSIX and OpenMP)	Tutorial Week 02 - Introduction to Parallel Computing	Lab Week 02 - C Primer (Not assessed)	
3	Inter Process Communications; Remote Procedure Calls	Tutorial Week 03 - Shared memory parallelism	Lab Week 03 - Threads (POSIX)	Assignment 1 specifications released
4	Parallel computing on distributed memory (MPI)	Tutorial Week 04 - Inter process communication	Lab Week 04 - Threads (OpenMP)	
5	Synchronization, MUTEX, Deadlocks	Tutorial Week 05 - Distributed memory parallelism	Lab Week 05 - Message passing interface	Assignment 2 specifications released
6	Election Algorithms, Distributed Transactions, Concurrency Control	Tutorial Week 06 - Synchronization	Lab Week 06 - Communication patterns	
7	Faults, Distributed Consensus, Security, Parallel Computing	Tutorial Week 07 - Transactions and concurrency	Lab Week 07 - Parallel data structure (I)	Assignment 1 due
8	Instruction Level Parallelism	Tutorial Week 08 - Consensus	1st assignment interview	
9	Vector Architecture	Tutorial Week 09 - Instruction level parallelism	Lab Week 09 - Parallel data structure (II)	
10	Data Parallel Architectures, SIMD Architectures	Tutorial Week 10 - Vector architecture	Lab Week 10 - Master slave	
11	Introduction to MIMD, Distributed Memory MIMD Architectures	Tutorial Week 11 - Data parallelism	Lab Week 11 - Performance tuning	Assignment 2 due
12	Recent development in Parallel Computing - Software, Hardware, Applications etc. (GPGPU etc.), Exam Revision	Tutorial Week 12 - Revision	2nd assignment code demo and interview	

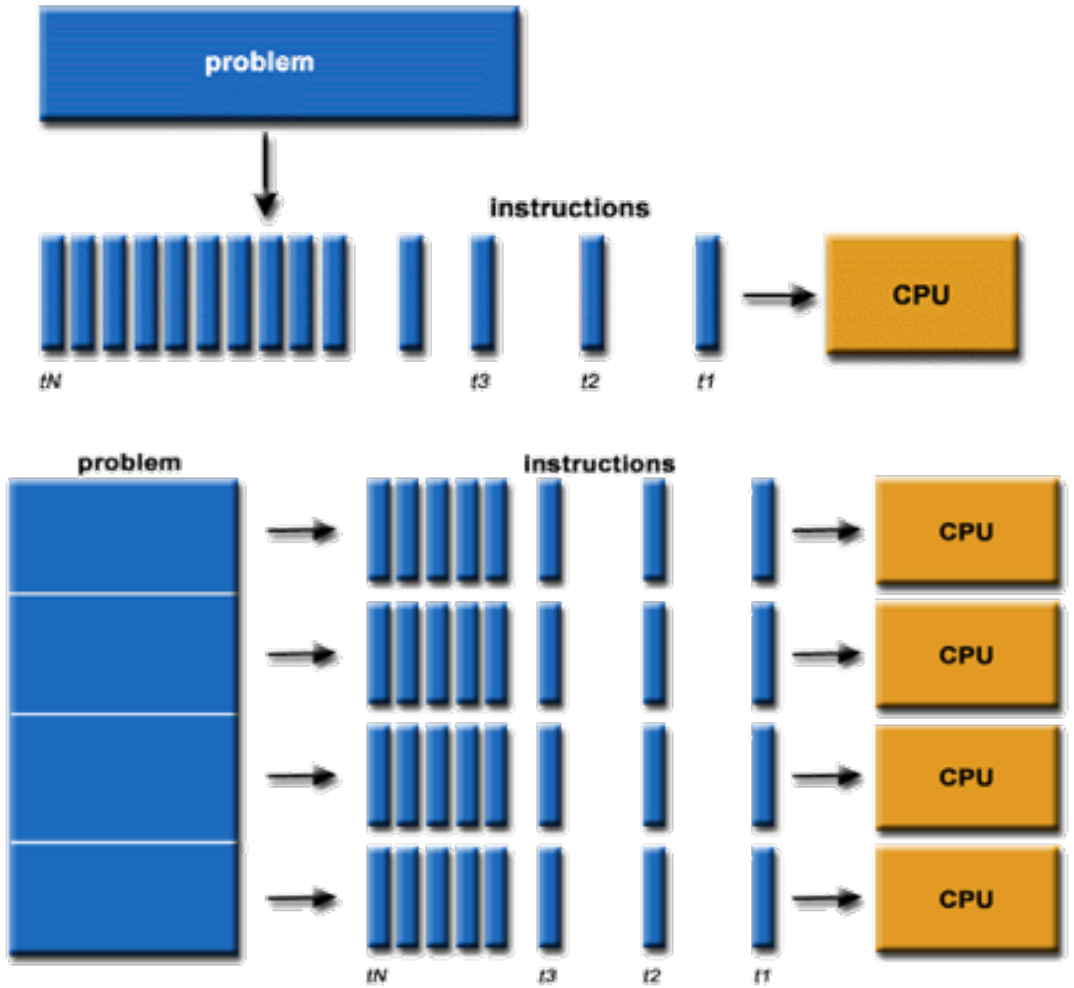
Assessment	Weight
Assignment 1	15%
Assignment 2	25%
Lab-work and Quizzes	10%
Final exam	50%

Today

- Parallel computing concept and applications
- Parallel computing models
- Distributed computing
- Parallel computing performance
- Parallel vs. distributed vs. asynchronous computing

Parallel computing concept and applications

Serial vs Parallel



Serial Computing

Parallel Computing

Adapted from https://computing.llnl.gov/tutorials/parallel_comp/

The demand for computational speed

- **Numerical modeling**
- **Simulation of scientific and engineering problems.**
- Computations must be completed within a "**reasonable**" time period.

Grand challenge problems

- Modeling large DNA structures
- Global weather forecasting
- Modeling motion of astronomical bodies

Serial or Parallel Machines

- Serial: Using a single Von Neumann machine.
- Parallel: Teraflops supercomputers.

FLUX: Serial machines

Reasons for not using existing serial machines:

A. Serial machines may not be able to reach the solution within a reasonable amount of time if using a single Von Neumann machine.

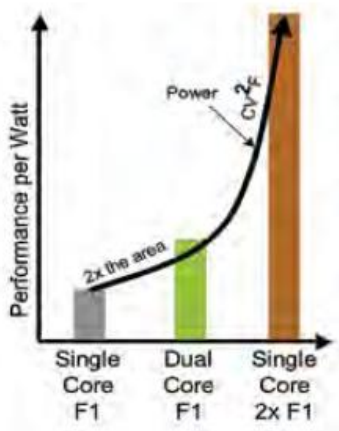
B. The **limits** imposed by physical constraints such as the speed of light

C. Cost of designing and manufacturing new chips. Current speed of chips - may reach a few billion Hz but will not grow enormously beyond that.

JYGAFY5

Parallelism for Mainstream Computing

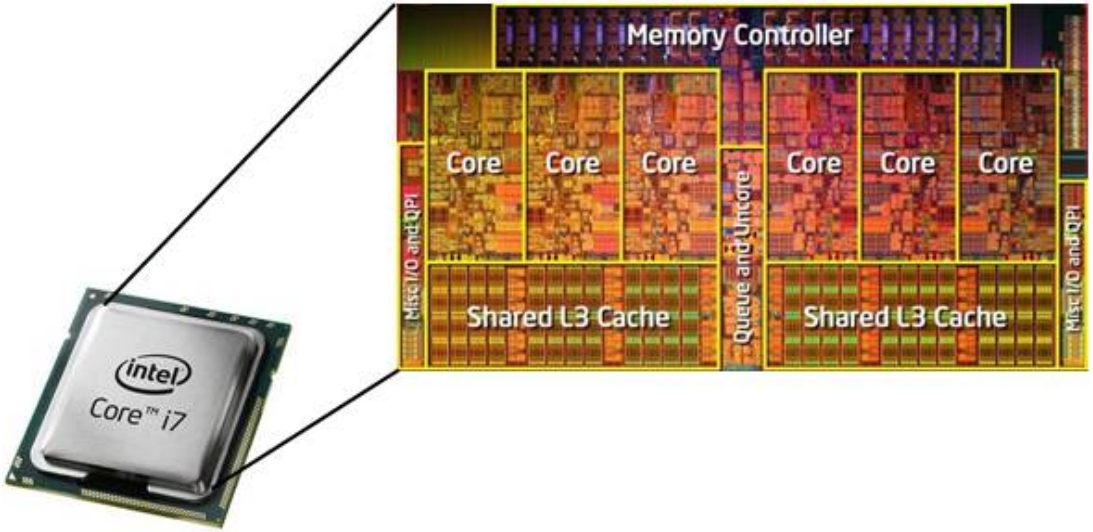
- In single core processor, the performance of the processor will not gain much simply by the increasing operating frequency:
- Memory wall
  - The processor was incapable of optimizing the performance by increasing the operating frequency as the actual cause is the increasing gap between the CPU and memory throughput (data transfer rate).
- ILP (Instruction Level Parallelism) wall
  - Difficulty in full parallel instruction processing.
- Power wall
  - Power consumption doubled following thedoubling of operating frequency. Faster clock speeds require higher input voltages. Every transistor leaks a small amount of current 1% clock increase results in a 3% power increase

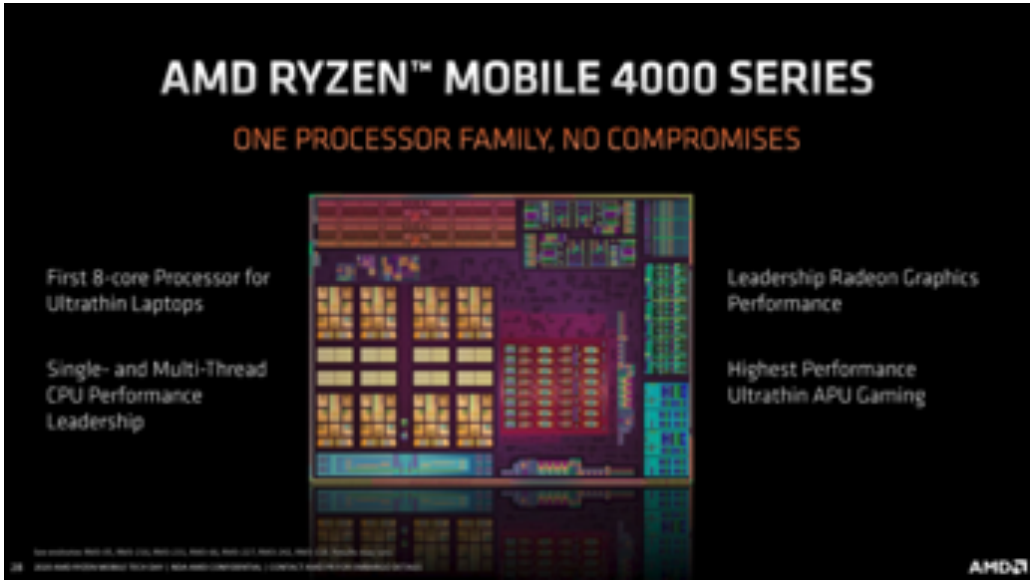


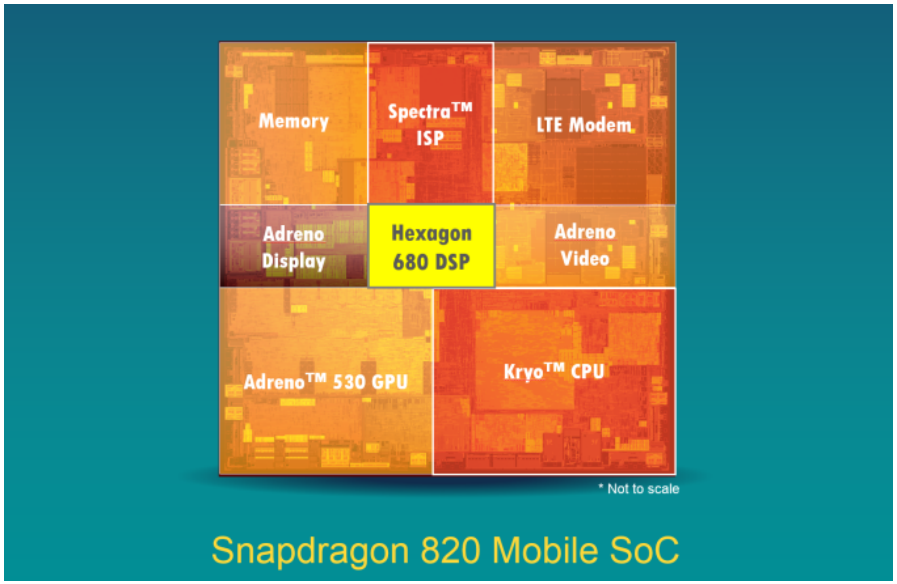
Types of parallelism

- Bit level parallelism
  - Parallelism achieved by doubling word size
- Instruction level parallelism
  - Superscalar processor with multiple executing units and out of order execution.
- Data level parallelism
  - Each processor performs the same task on different pieces of distributed data
- Task level parallelism
  - Each processor executes a different thread (or process) on the same or different data

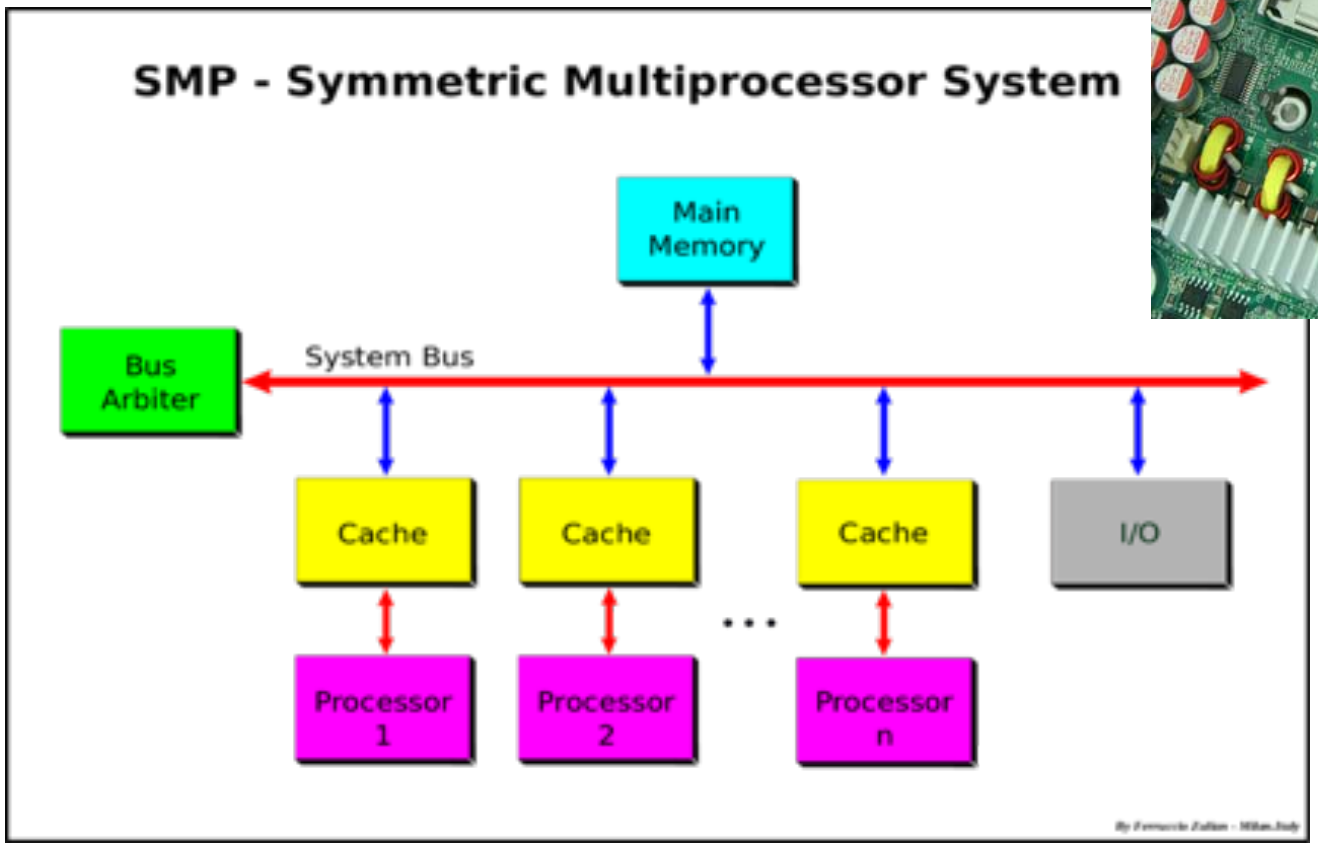
Multi core processor



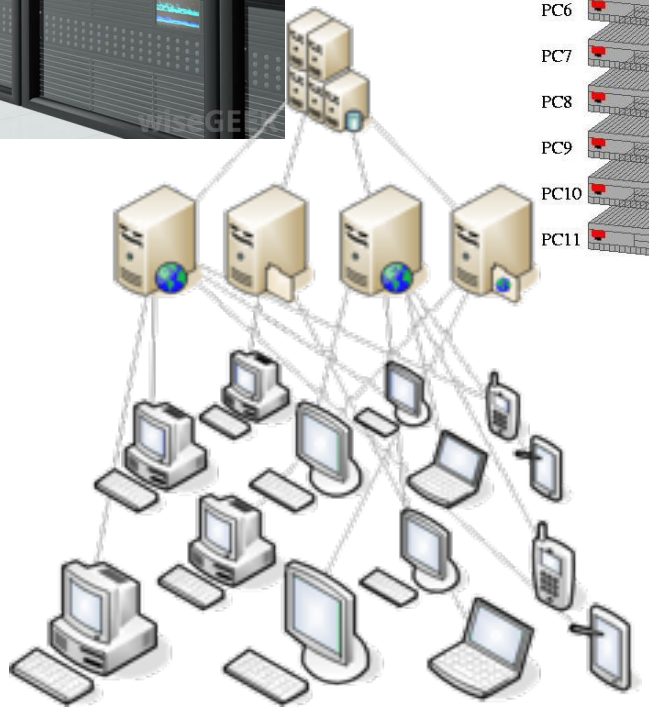


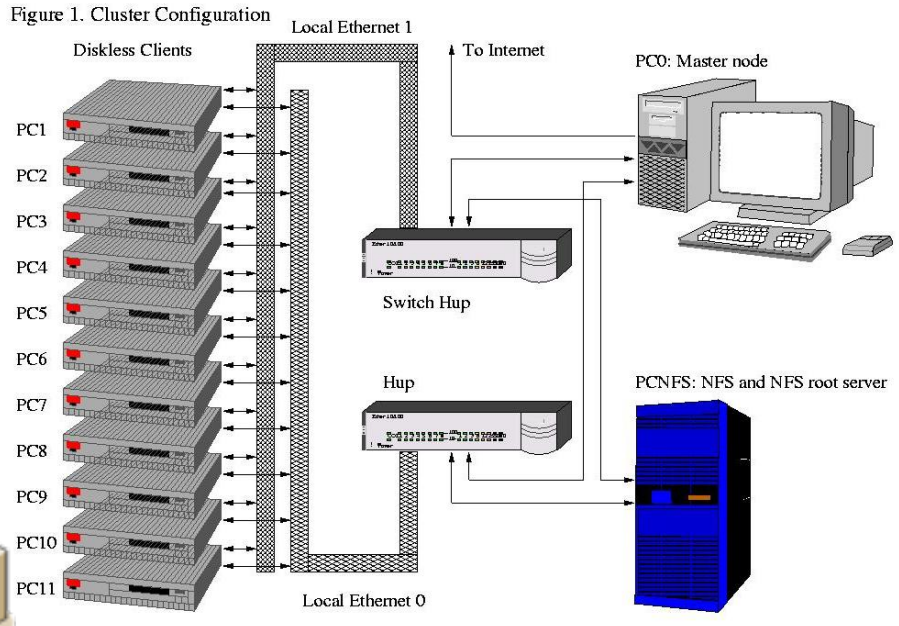


SMP - Symmetric multiprocessor



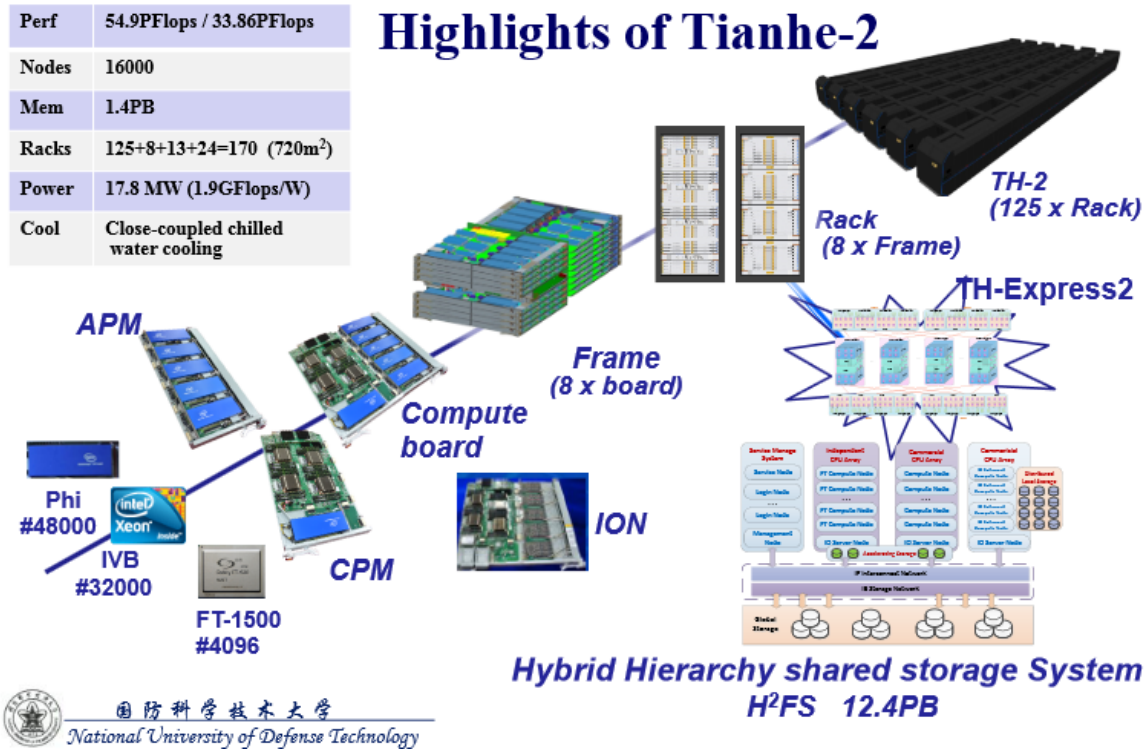
Cluster & Distributed/Grid computing



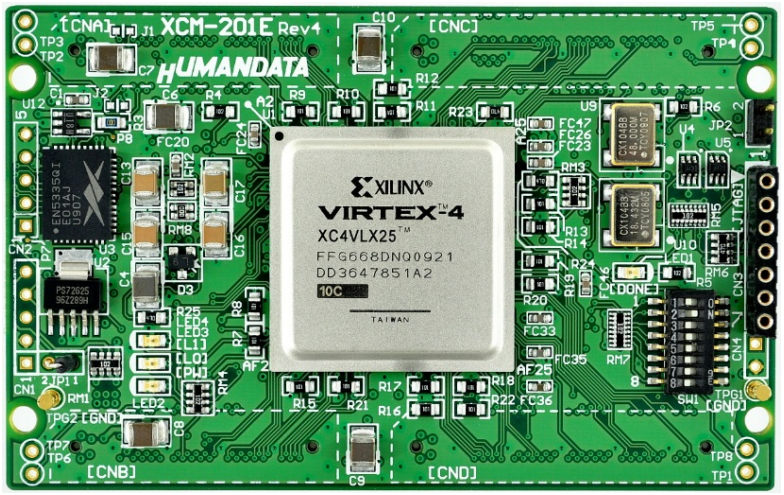


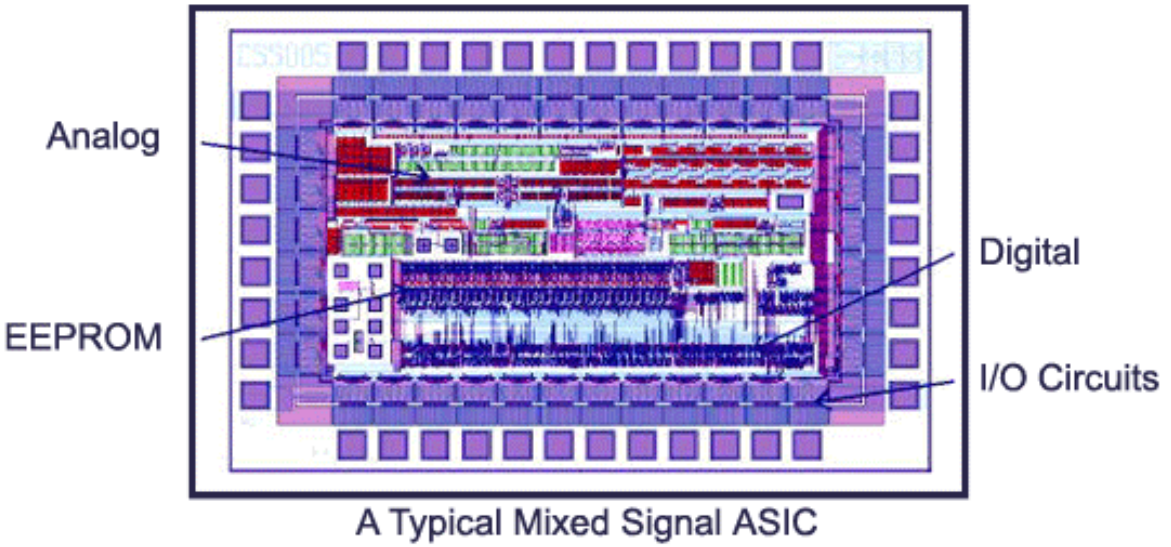
Massively parallel computing / supercomputing

Perf	54.9PFlops / 33.86PFlops
Nodes	16000
Mem	1.4PB
Racks	125+8+13+24=170 (720m^2)
Power	17.8 MW (1.9GFlops/W)
Cool	Close-coupled chilled water cooling



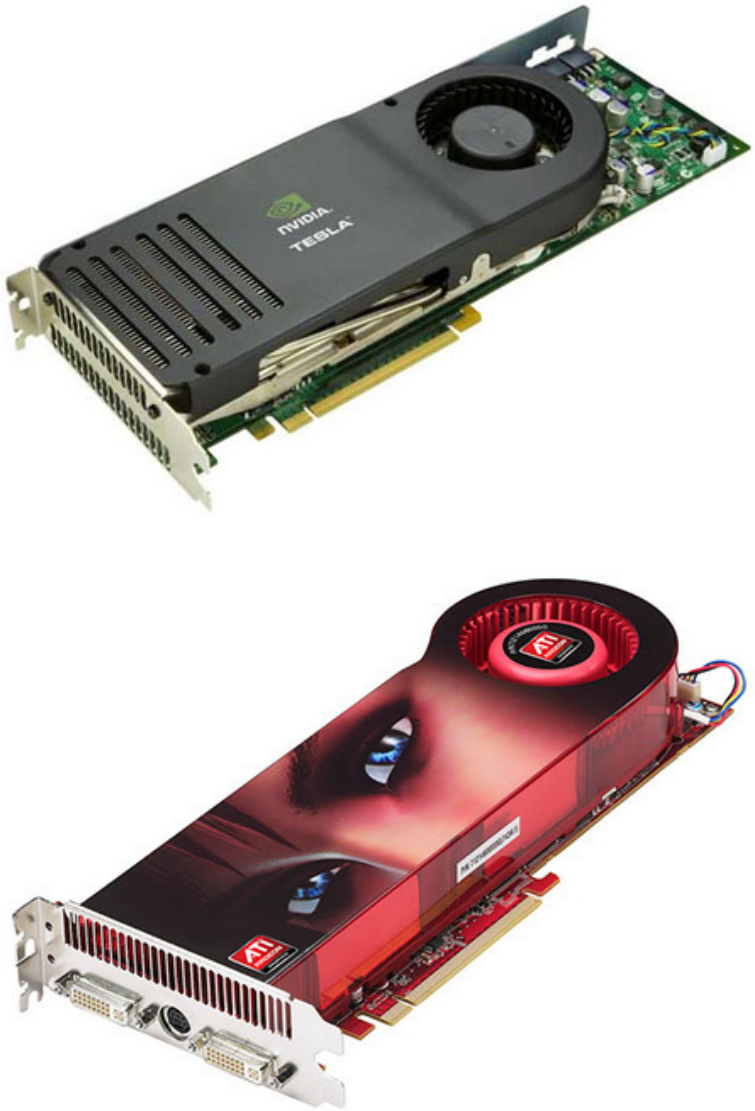
Reconfigurable computing (FPGA) & ASIC

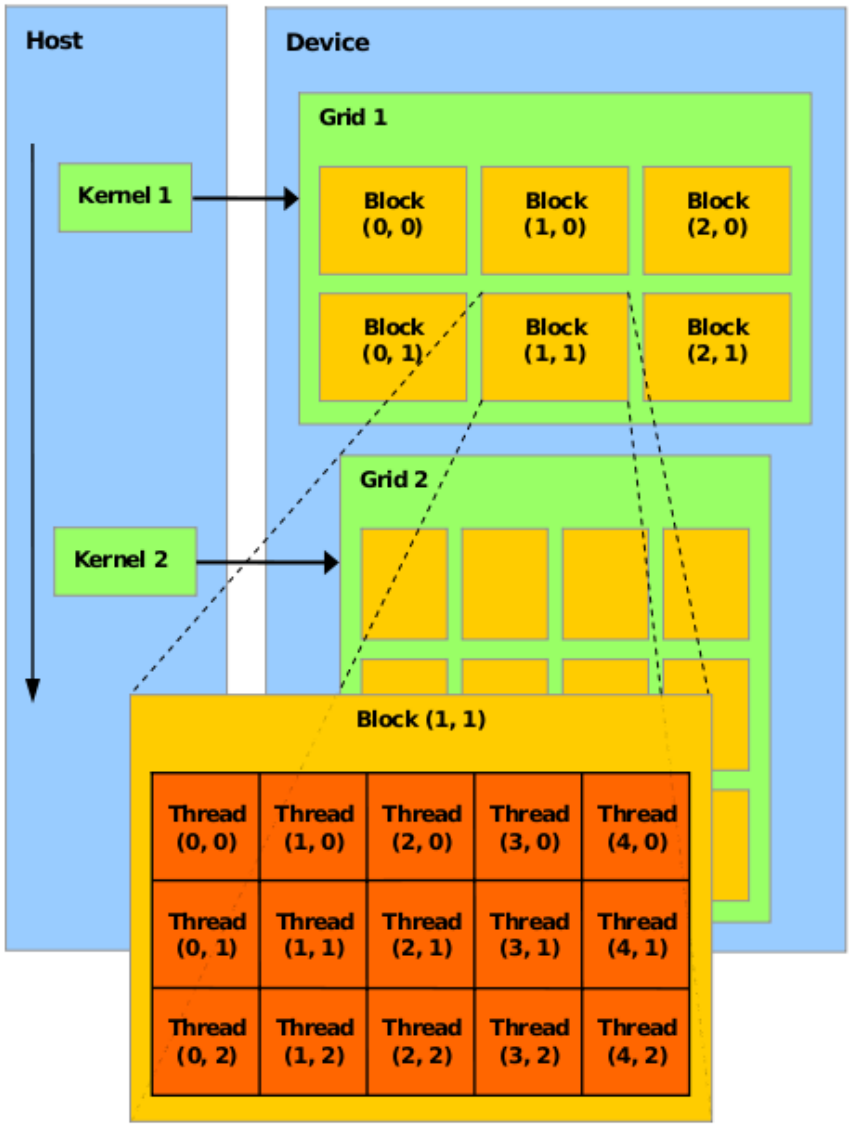




A Typical Mixed Signal ASIC

General-purpose computing on graphics processing units (GPGPU)





Embarrassingly parallel computation

- An **ideal** parallel computation is one that can be immediately divided into completely independent parts that can be executed simultaneously – this is what is known as an *embarrassingly-parallel computation*, or *job-level parallelism*.

- In this context, a network is a collection of integrated processors that are communicating and sharing information to speed up the solution to a single task.

Issues

- How big of a collection?
- What kind of processors?
- How closely integrated is the collection of processors? **Tightly coupled**? **Loosely coupled**?
- How do the processors communicate?

Job = Task1 + Task2 + Task3 + …

- What type of information is shared?
- Focus on a single task

Parallel computing Models

Classification

- Parallel computers can be **classified** by:
  - Type and number of processors.
  - Interconnection scheme of the processors
  - Communication scheme.
  - Input/output operations.

Conventional computer

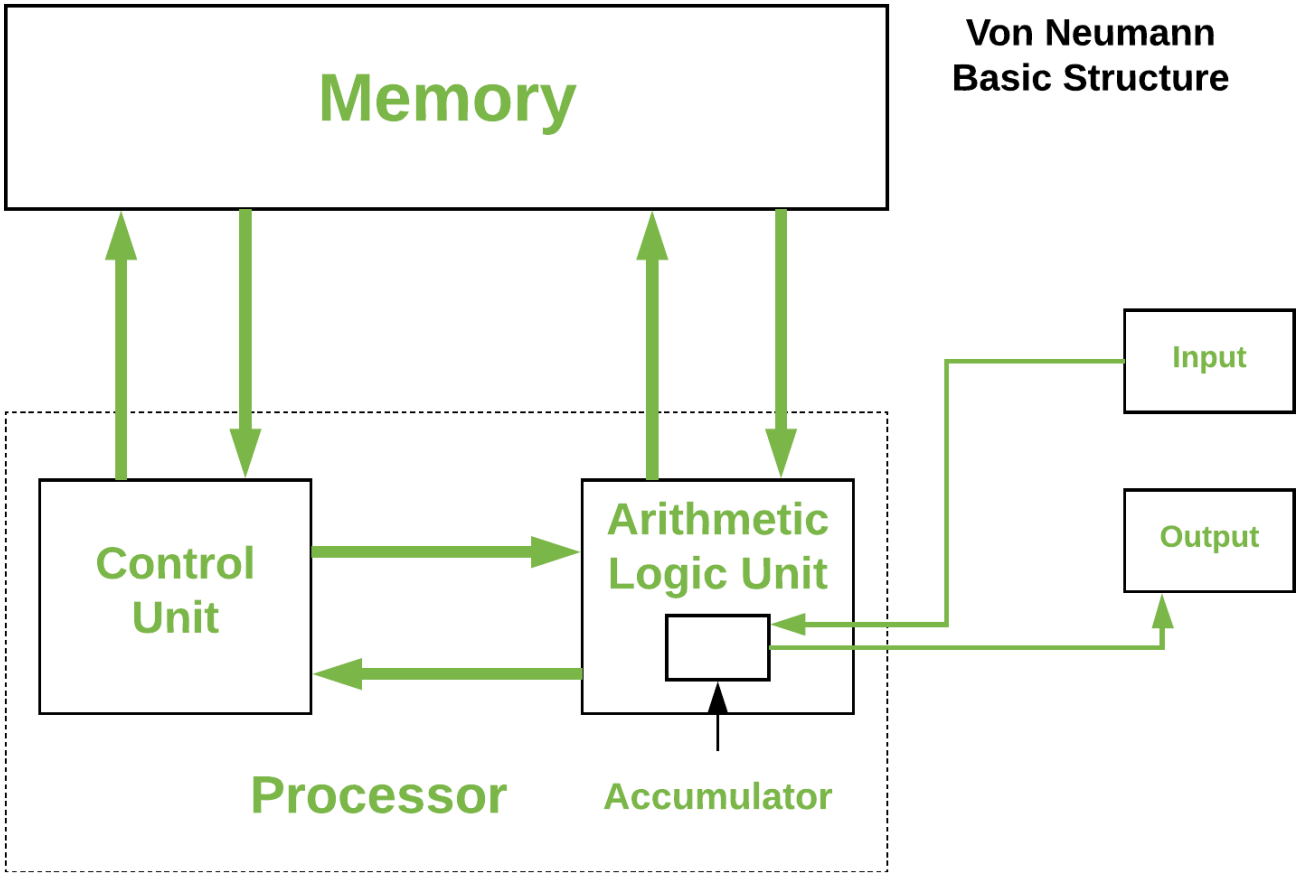


Image link: https://media.geeksforgeeks.org/wp-content/uploads/basic_structure.png

Multi-processor computer architecture

- Flynn's Classification
- Shared memory model
- Distributed memory model
- Distributed shared memory model

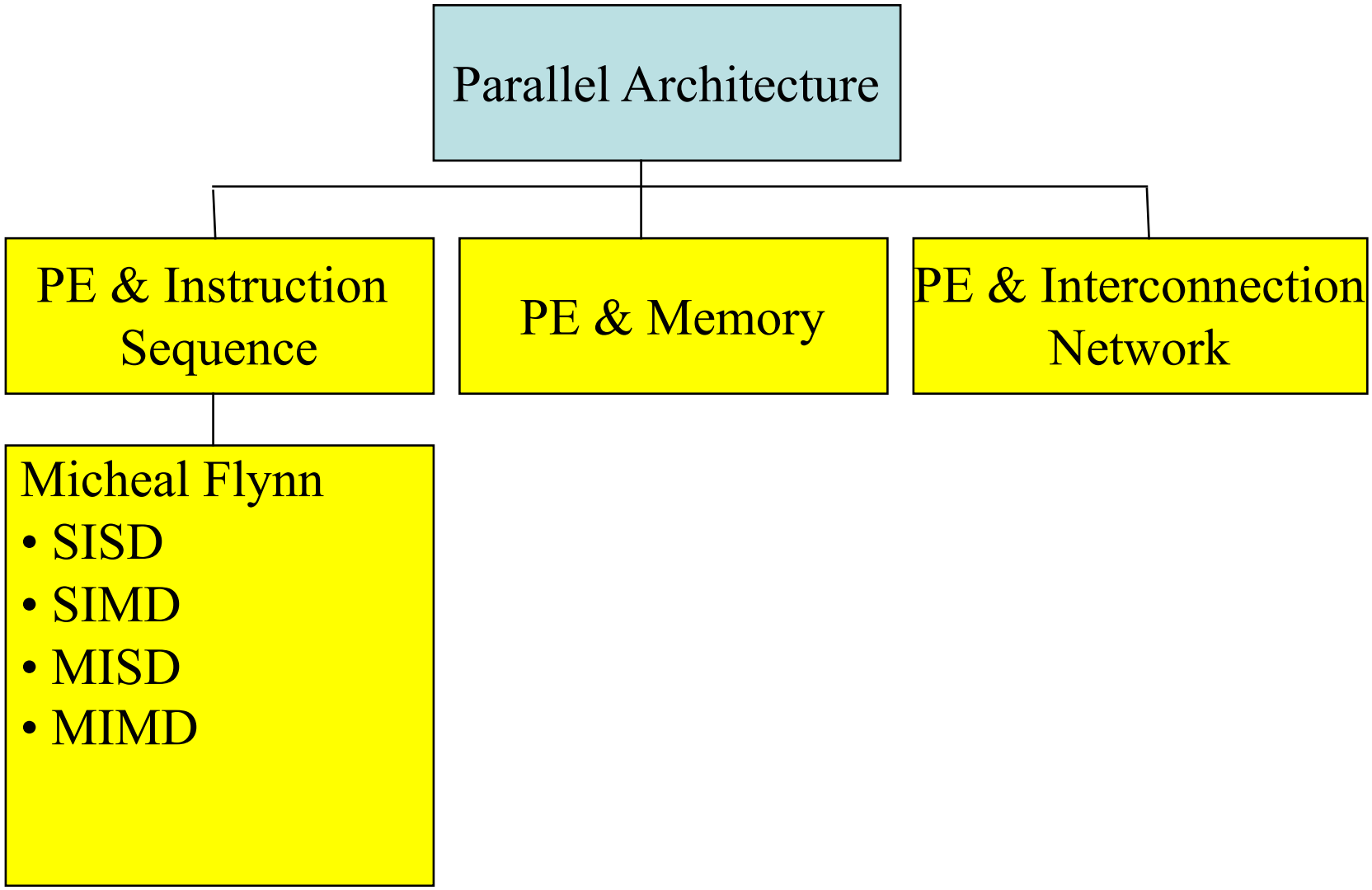
Taxonomy (Classification)

- A taxonomy of parallel architectures can be built based on three relationships:

1. **Relationship between PE and the instruction sequence executed**
2. **Relationship between PE and the memory**
3. **Relationship between PE and the interconnection network**

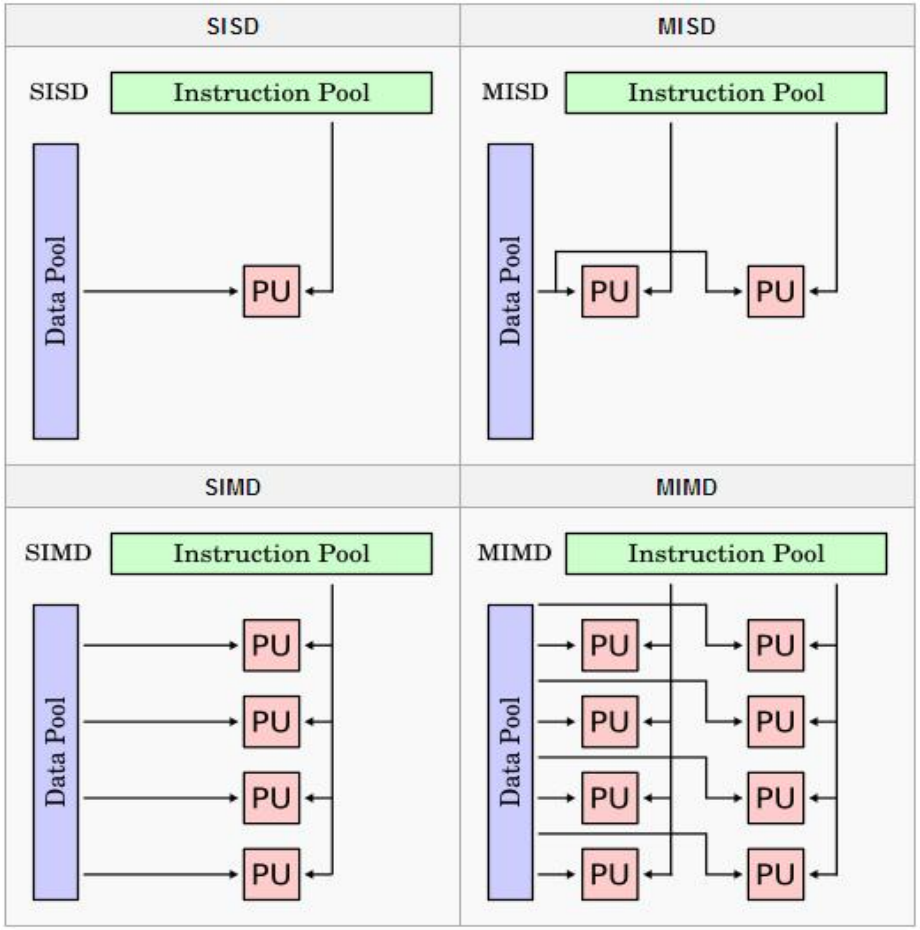
PE: Processing Element (CPU)

Parallel Architecture

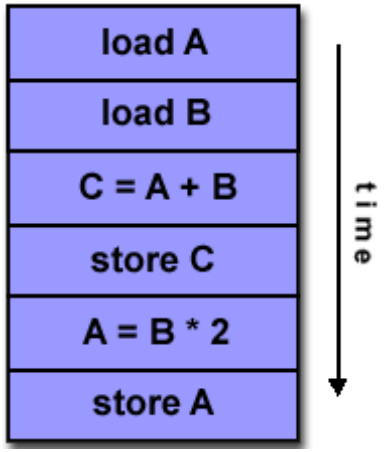


Flynn taxonomy

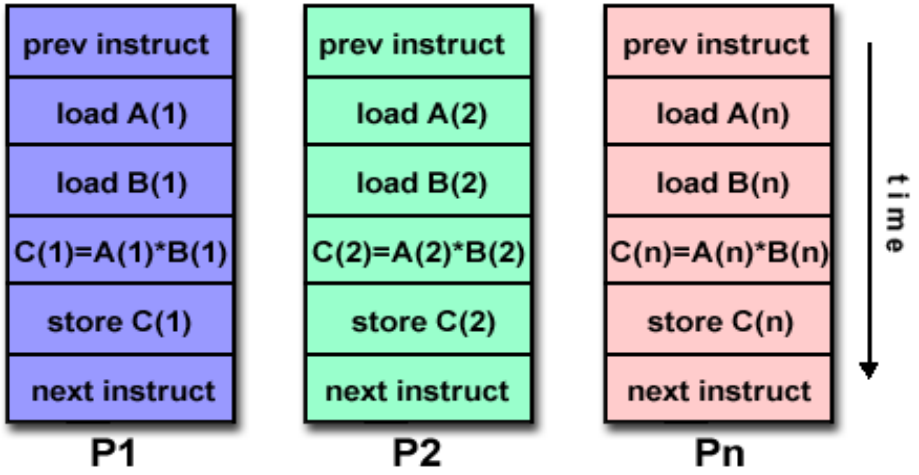
- Michael Flynn (1966) developed a taxonomy of parallel systems based on the number of independent *instruction and data streams.*



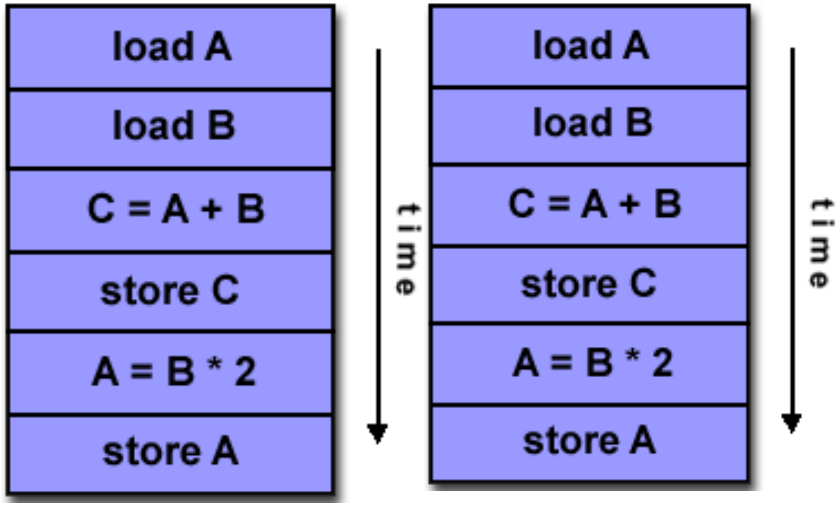
Flynn's Classification



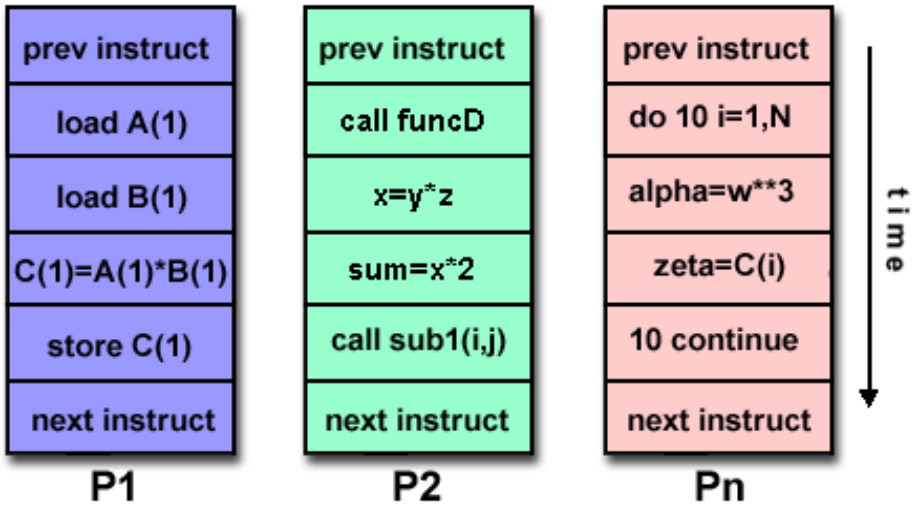
Single Instruction, Single Data (SISD)



Single Instruction, Multiple Data (SIMD)



Multiple Instruction, Single Data (MISD)



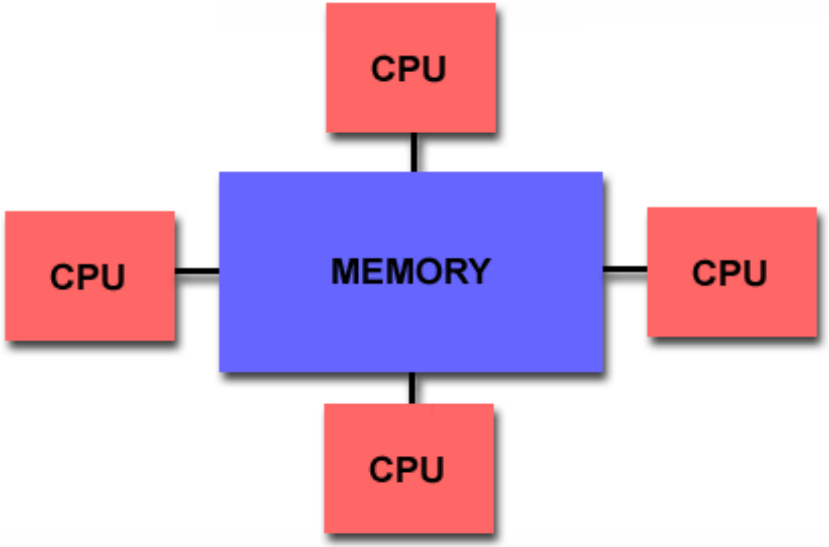
Multiple Instruction, Multiple Data (MIMD)

Parallel Computer Memory Architectures

- Broadly divided into three categories
  - Shared memory
  - Distributed memory
  - Hybrid

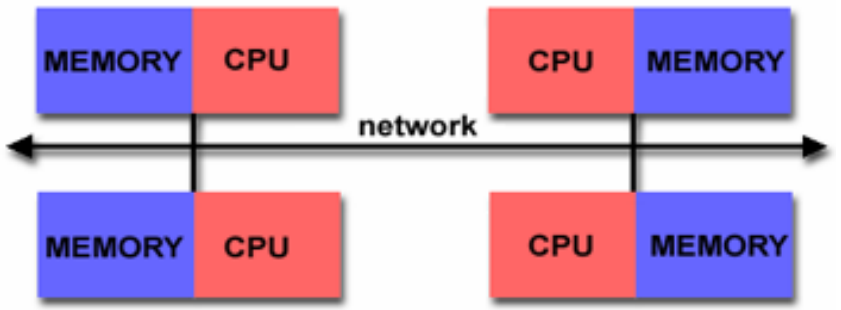
Shared Memory

- 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***.



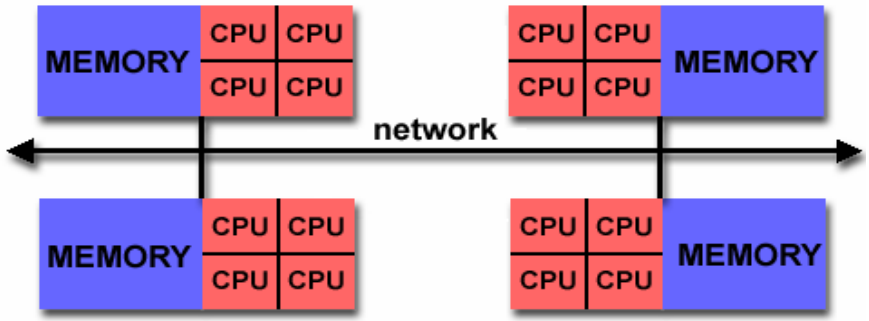
Distributed Memory

- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory. There is no concept of global address space across all processors.
- Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
- 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. Synchronization between tasks is likewise the programmer's responsibility.
- The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.



Hybrid

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
- Advantages and Disadvantages: whatever is common to both shared and distributed memory architectures.



Parallel Programming Models

Parallel Programming Models

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid
- Parallel programming models exist as an abstraction above hardware and memory architectures.
- Single Program Multiple Data (SPMD) structure
- Multiple Program Multiple Data (MPMD) structure

How to create executable code for a shared memory multiprocessor

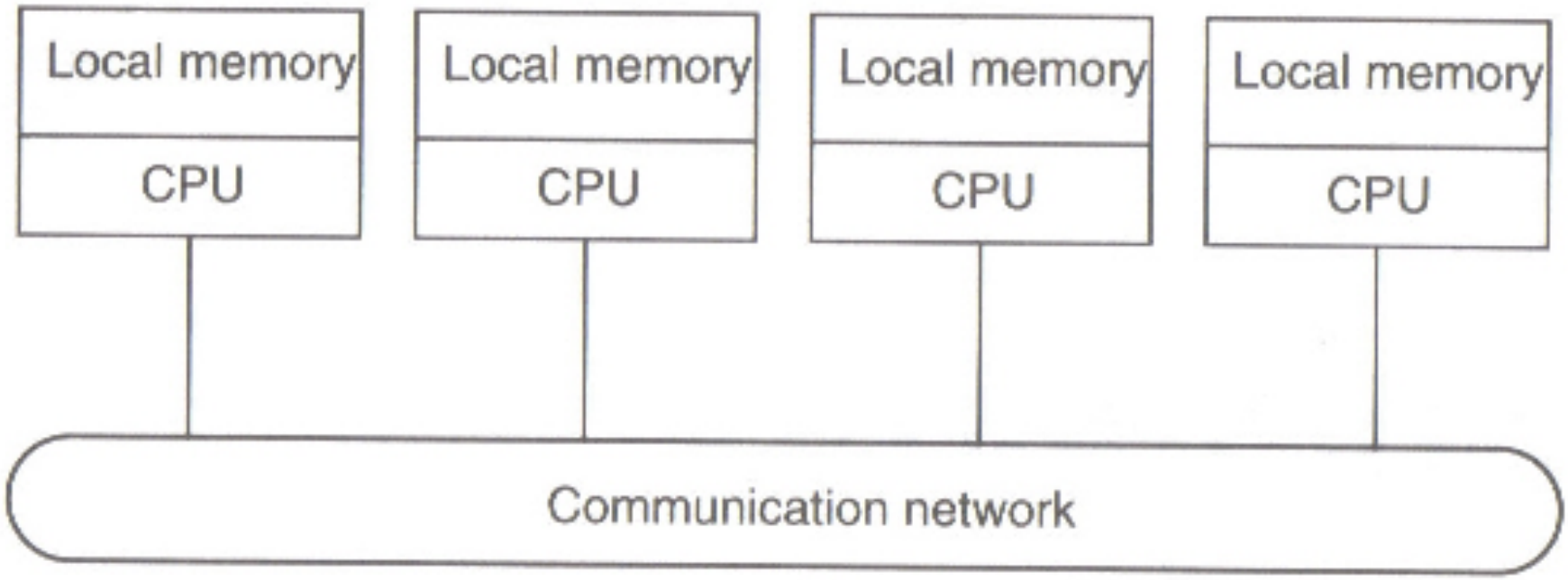
1. Parallel Programming Languages
2. Threads
3. Sequential programming

Programming message-passing multicomputer

- **Dividing** the problem into parts that are intended to be executed simultaneously to solve the problem
- Common approach is to use message-passing library routines that are inserted into a conventional sequential program for message passing.
- A problem is divided into a number of concurrent processes that may be executed on a different computer.
- Processes **communicate by sending messages**; this will be the only way to distribute data and results between processes.

Distributed Computing

Loosely coupled systems



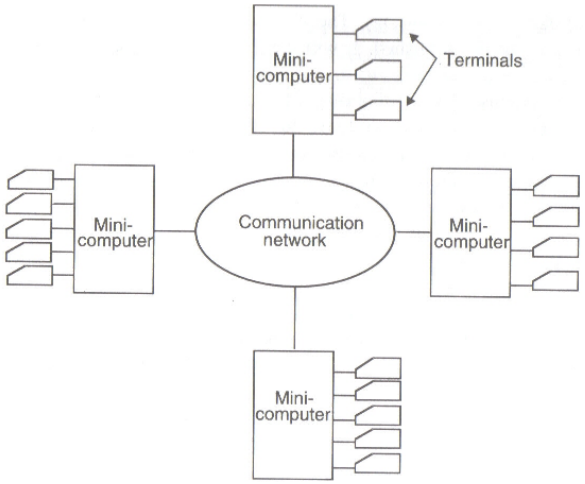
Evolution of Distribution System

- Two advances as the reason for spread of distributed systems
  - Powerful micro-processor
  - Computer network

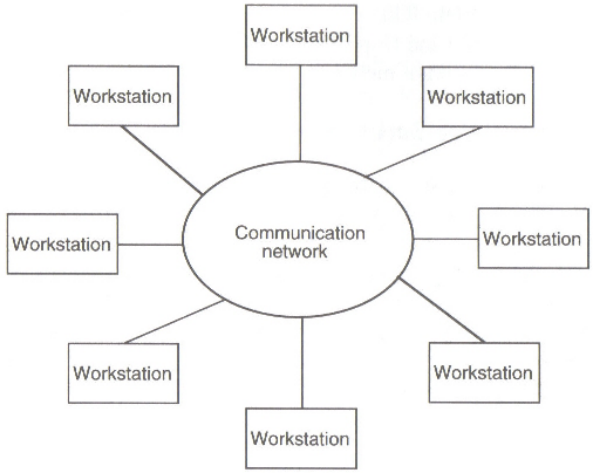
Distributed computing system models

- Various models are used for building distributed computing systems.These models can be broadly classified into five categories:
  - Minicomputer model
  - Workstation model
  - Workstation-server model
  - Processor-pool model
  - Hybrid model

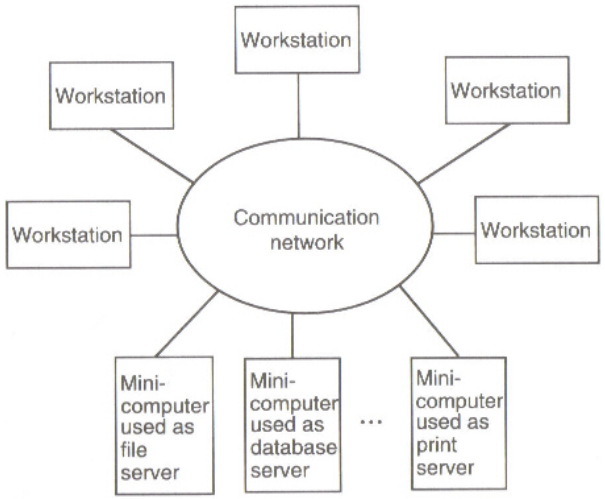
Distributed computing system models



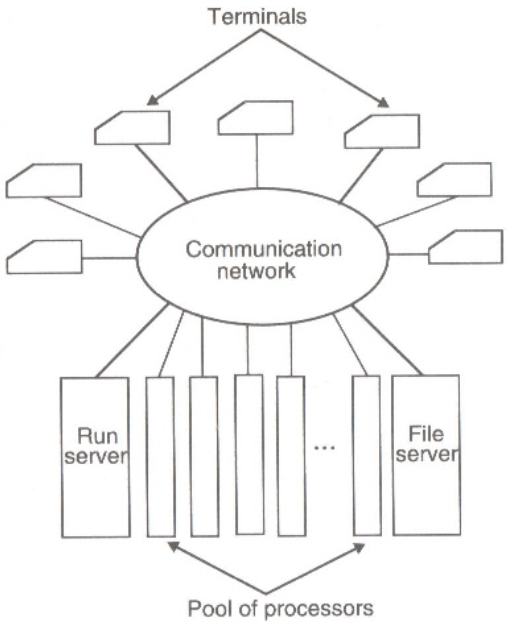
1. Minicomputer model



2. Workstation model



3. Workstation-server model



4. Processor-pool model

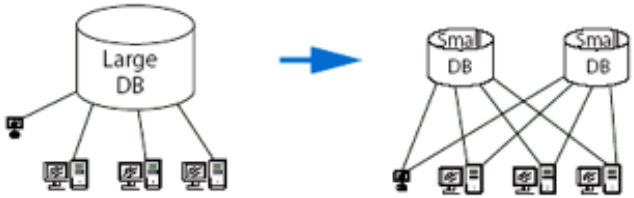
5. Hybrid Model

Distribution Model

- File Model - Resources are modeled as files. Remote resources are accessible simply by accessing files.
- Remote Procedure Call Model - Resource accesses are modeled as function calls. Remote resources can be accessed by calling functions.
- Distributed Object Model - Resources are modeled as objects which are a set of data and functions to be performed on the data. Remote resources are accessible simply by accessing an object.

Advantages of Distributed Systems

- Economics: Microprocessors offer a better price / performance than mainframes.
- Speed: A distributed system may have more total computing power than a mainframe. E.g. one large database may be split into many small databases. In that way, we may improve response time.
- Inherent distribution: Some application like banking, inventory systems involve spatially separated machines.



Advantages of Distributed Systems

- Reliability: If 5% of the machines are downed, the system as a whole can still survive with a 5% degradation of performance.
- Incremental growth: Computing power can be added in small increments
- Sharing: Allow many users access to a common database and peripherals.
- Communication: Make human-to-human communication easier.
- Effective Resource Utilization: Spread the workload over the available machines in the most cost effective way.

Disadvantages of Distributed Systems

- Software: It is harder to develop distributed software than centralized one.
- Networking: The network can saturate or cause other problems.
- Security: Easy access also applies to secret data.

Challenges of Distributed Systems

- Heterogeneity - Within a distributed system, we have variety in networks, computer hardware, operating systems, programming languages, etc.
- Openness - New services are added to distributed systems. To do that, specifications of components, or at least the interfaces to the components, must be published.
- Transparency - One distributed system looks like a single computer by concealing distribution.
- Performance - One of the objectives of distributed systems is achieving high performance out of cheap computers.

Challenges of Distributed Systems

- Scalability - A distributed system may include thousands of computers. Whether the system works is the question in that large scale.
- Failure Handling - One distributed system is composed of many components. That results in high probability of having failure in the system.
- Security - Because many stake-holders are involved in a distributed system, the interaction must be authenticated, the data must be concealed from unauthorized users, and so on.
- Concurrency - Many programs run simultaneously in a system and they share resources. They should not interfere with each other.

Reliability and Performance

- **Reliability**
  - High probability to have faulty components in a distributed system. It is theoretically possible to build a distributed system such that if a machine goes down, the other machine takes over the job.
  - Availability: The fraction of time that the system is available. $R = \frac{usable\_time}{total\_time}$
  - Fault tolerance: Distributed systems can hide failures from the users.
- **Performance**
  - Maximum aggregate performance of the system can be measured in terms of Maximum aggregate floating-point operation.
  - P = N*C*F*R, where P performance in flops, N number of nodes, C number of CPUs, F floating point ops per clock period - FLOP, R clock rate.The similar measures with MOP/MIP.

Scalability and Utilization

- **Scalability**
  - Computed as S = T(1) / T(N), where T(1) is the wall clock time for a program to run on a single processor, T(N) is the runtime over N processors.
  - A scalability figure close to N means the program scales well. Scalability metric helps estimate the optimal number of processors for an application.
- **Utilization**
  - Calculated as, U = S(N)/N
  - Values close to unity or 100% are ideally sought.

Parallel Computing Performance

Potential for increased computational speed

- In all forms of MIMD multiprocessor/multicomputer systems, it is necessary to divide the computations into tasks or processes that can be executed simultaneously.
- The size of a process can be described by its granularity, which is related to the number of processors being used.
  - In coarse granularity, each process contains a large number of sequential instructions and takes substantial time to execute.
  - In fine granularity, a process might consists of few instructions or perhaps even one instruction.
  - Medium granularity describes the middle ground.
- Sometimes granularity is defined as the size of the computation between communication or synchronization points.

Granularity Metric

- Ratio
  - $$\frac{Computation}{Communication} = \frac{Computation\ Time}{Communication\ Time} = \frac{t_{comp}}{t_{comm}}$$
- Speed up factor
  - $$S(n) = \frac{\text{Execution time using one processor (single processor system)}}{\text{Execution time using a multiprocessor with n processors}} = \frac{t_S}{t_P}$$
- The **maximum speedup is n** with n processors (linear speed up).
  - $$S_{(n)} = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with processors}}$$

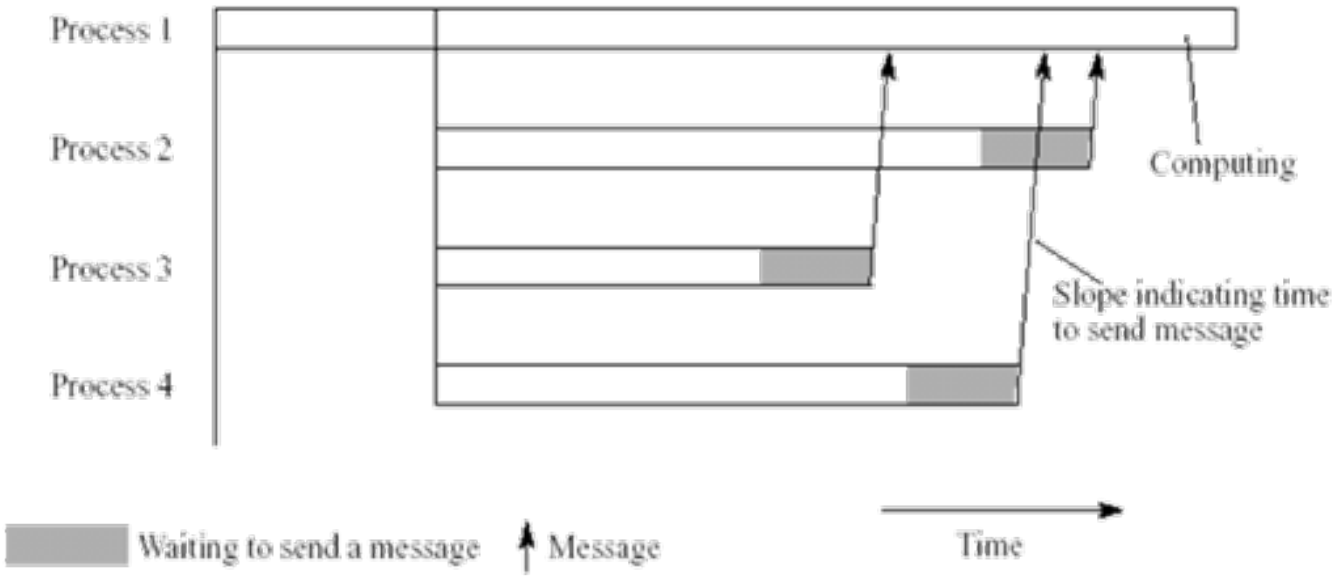
Superlinear Speedup

- where $S(n) > n$, may be seen on occasion, but usually this is due to using a suboptimal sequential algorithm or some unique feature of the architecture that favors the parallel formation.
- One common reason for superlinear speedup is the **extra memory** in the multiprocessor system which can hold more of the problem data at any instant, it leads to less relatively slow disk-memory traffic.
- Superlinear speedup can occur in search algorithms

Space-time diagram of a message passing program

- It is reasonable to expect that some part of a computation cannot be divided at all into concurrent processes and must be performed serially.

- During the initialization period or period before concurrent processes are being setup, only one processor is doing useful work, but for the rest of the computation, additional processors are operating in parallel.

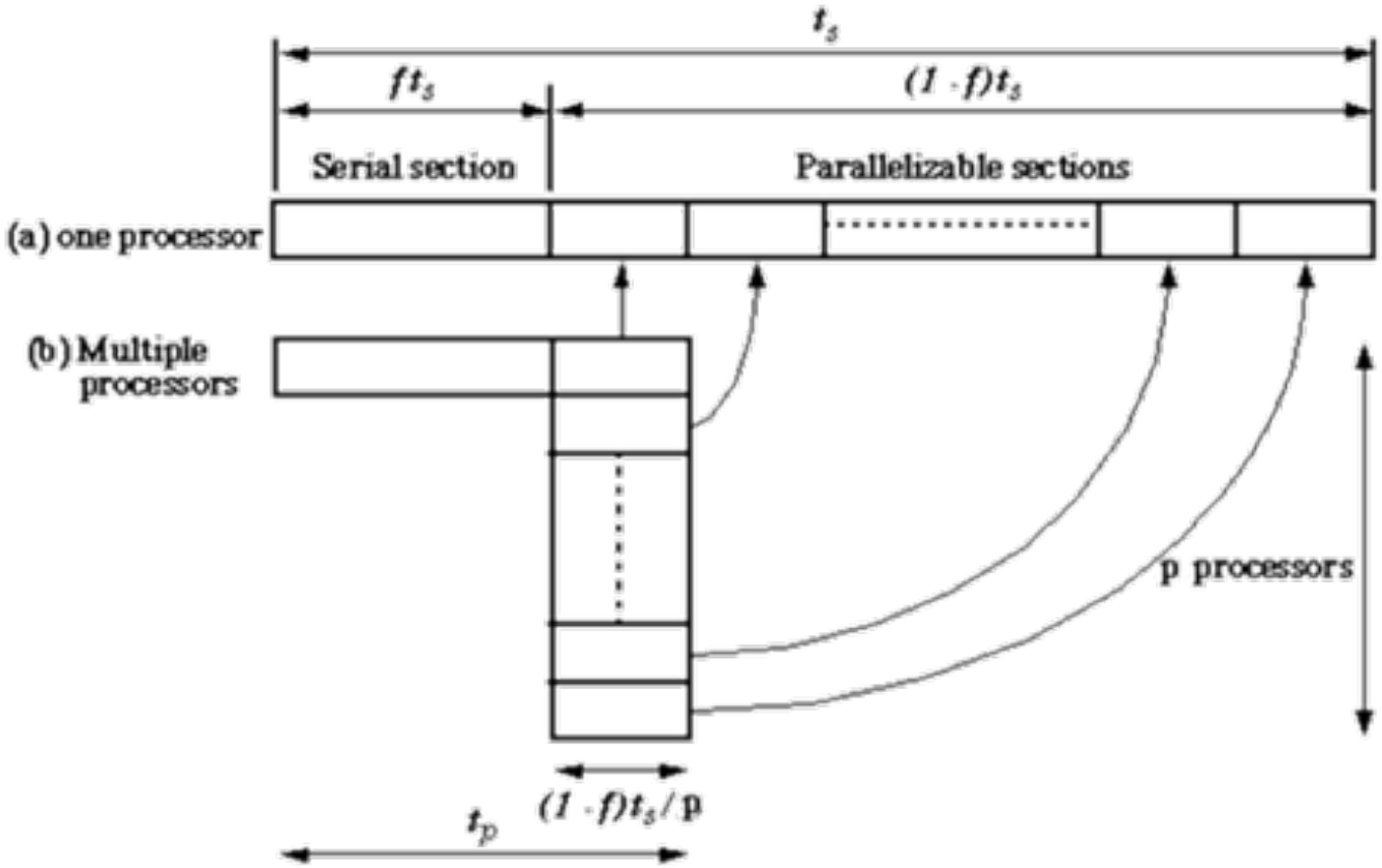


Maximum speed

- If the fraction of the computation that cannot be divided into concurrent tasks is f, the time to perform the computation with *p* processors is given by $ft_s + (1-f)\,t_s/p$ as shown in the next slide.
- Illustrated is the case with a single serial part at the beginning of the computation which cannot be parallelized, but the remaining parts could be distributed throughout the computation.
- The speed up factor S(p) is given by

$$S(p) = \frac{t_S}{ft_S + (1-f)t_S / p} = \frac{p}{1+(p-1)f}$$

Parallelizing sequential problem-Amdahl's law



Alternative:

$$\frac{1}{r_s + \frac{r_p}{p}}$$

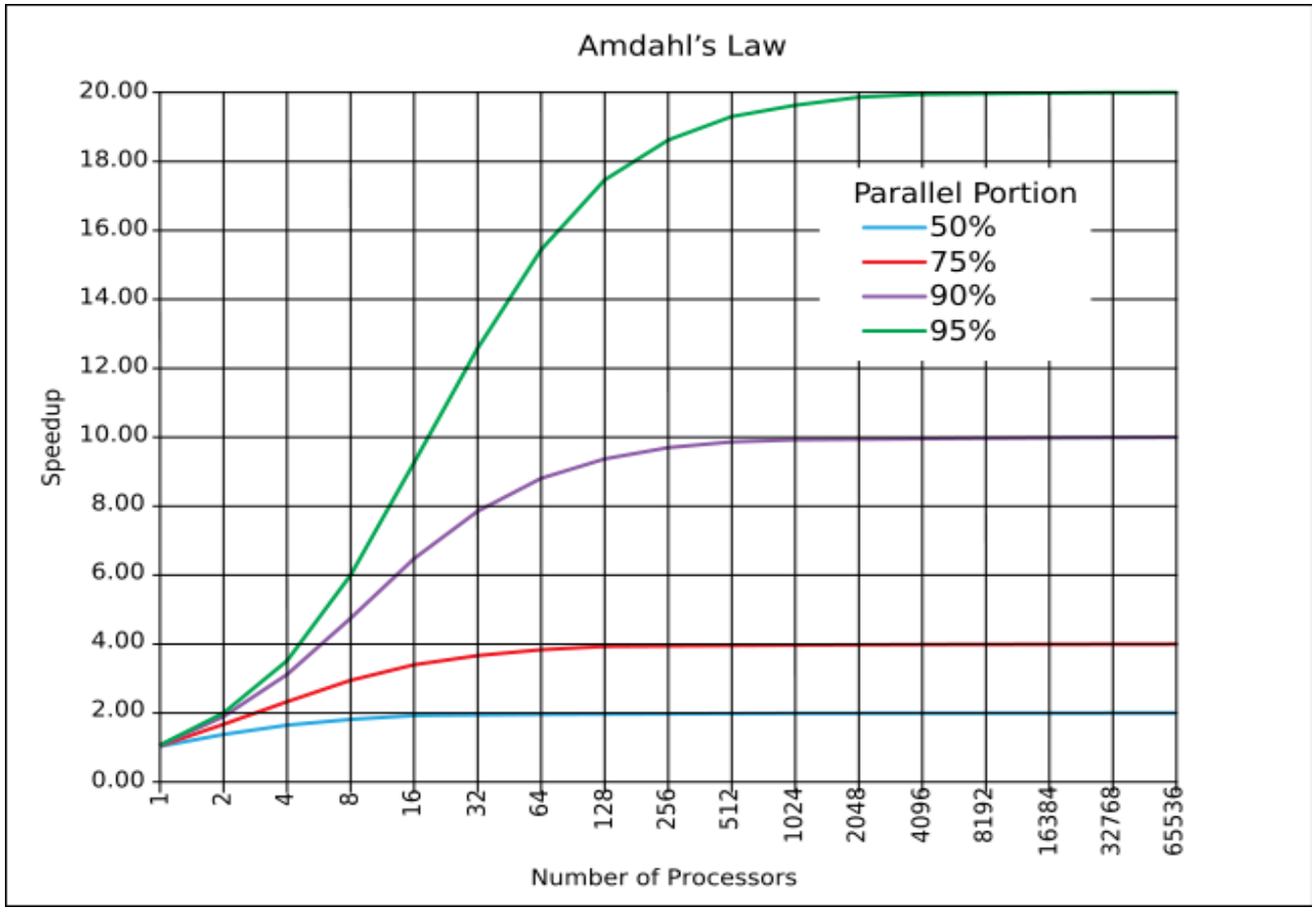
r_s is the serial ratio (non-parallelizable portion)

r_p is the parallel ratio (parallelizable portion)

p is the number of processors

Amdahl's law

- Even with infinite number of processors, maximum speedup limited to 1/*f*. i.e., S(n) =1/f, $n -> infinity$



Efficiency

$$\text{Efficiency} = \frac{\text{Speedup}}{\text{Processors}}$$

$$\varepsilon(n,p) \le \frac{\sigma(n)+\varphi(n)}{p\sigma(n)+\varphi(n)+p\kappa(n,p)}$$

0 <= e(*n*,*p*) <= 1

Amdahl's law

$$\psi(n,p) \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p+\kappa(n,p)} \le \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\varphi(n)/p}$$

Let *f* = s(*n*)/(s(*n*) + j(*n*)); i.e., *f* is the fraction of the code which is inherently sequential

$$\psi \le \frac{1}{f+(1-f)/p}$$

Examples

- 95% of a program's execution time occurs inside a loop that can be executed in parallel. What is the maximum speedup we should expect from a parallel version of the program executing on 8 CPUs?

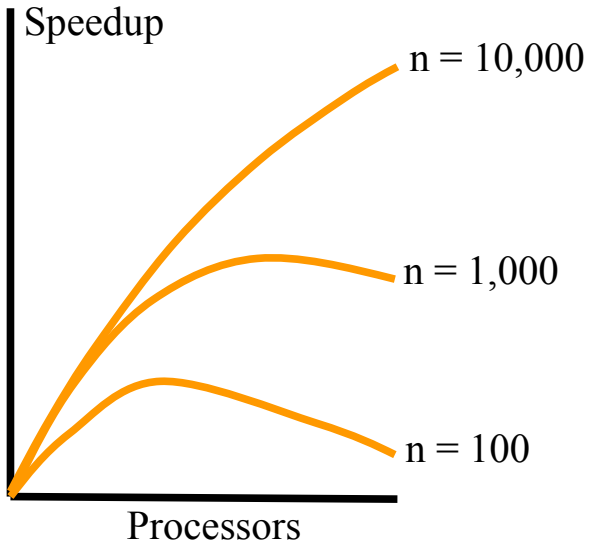
$$\psi \leq \frac{1}{0.05 + (1 - 0.05)/8} \cong 5.9$$

- 20% of a program's execution time is spent within inherently sequential code. What is the limit to the speedup achievable by a parallel version of the program?

$$\lim_{p \to \infty} \frac{1}{0.2 + (1 - 0.2)/p} = \frac{1}{0.2} = 5$$

Amdahl's law limitation

- Limitations of Amdahl's Law
  - Ignores k(*n*,*p*) - overestimates speedup
  - Assumes f constant, so underestimates speedup achievable
- Amdahl Effect
  - Typically k(*n*,*p*) has lower complexity than j(*n*)/*p*
  - As *n* increases, j(*n*)/*p* dominates k(*n*,*p*)
  - As n increases, speedup increases
  - As n increases, sequential fraction f decreases.



Gustafson's law

- **Gustafson's Law** (also known as **Gustafson-Barsis' law,** 1988) states that any sufficiently large problem can be efficiently parallelized.

- Gustafson's Law is closely related to Amdahl's law, which gives a limit to the degree to which a program can be sped up due to parallelization.

$$S_s(p) = p + (1 - p) f t_s$$

- where *p* is the number of processors, *s* is the speedup, and *f* the non-parallelizable part of the process

- Gustafson's law addresses the shortcomings of Amdahl's law, which cannot scale to match availability of computing power as the machine size increases.

Scaled speedup Factor

- If we fix the parallel execution time t_p and the fraction of the process which cannot be parallelized *f*, we can use Gustafson's law to figure out the speed-up factor given the number of processors, *p*.
- This is called the *scaled* speedup factor because the t_p is scaled to 1.

$$S_s(p) = \frac{ft_s + (1-f)t_s}{ft_s + (1-f)t_s / p} = \frac{p + (1-p)ft_s}{1} = p + (1-p)ft_s$$

  - $ft_s + (1-f)t_s/p = 1$
  - $pft_s+(1-f)t_s = p$
  - $t_s = p + (1-p)\, ft_s$

Parallel vs. distributed vs. asynchronous computing

Parallel vs. Distributed computing

S.NO	PARALLEL COMPUTING	DISTRIBUTED COMPUTING
1	Many operations are performed simultaneously	System components are located at different locations
2	Single computer is required	Uses multiple computers
3	Multiple processors perform multiple operations	Multiple computers perform multiple operations
4	It may have shared or distributed memory	It have only distributed memory
5	Processors communicate with each other through bus	Computer communicate with each other through message passing.
6	Improves the system performance	Improves system scalability, fault tolerance and resource sharing capabilities

Parallel vs. Asynchronous computing

- Both parallel and asynchronous programming models perform similar tasks and functions in modern programming languages. However, these models have conceptual differences.
- Asynchronous programming is used to avoid “blocking” within a software application. Example, database queries or network connections are best implemented using asynchronous programming.
  - An asynchronous call spins off a thread (e.g. an I/O thread) to complete the target task.
  - An asynchronous calls prevents the user interface from the “freeze” effect.
- As for parallel programming, the main task is segmented into smaller tasks, to be executed by a set of threads within the reach of a common variable pool.
  - Parallel programming can also prevent user interface “freeze” effect when running computational expensive tasks on a CPU.
- Key difference: In an asynchronous call, control over threads is limited and is system dependent. In parallel programming, the user has more control over task distribution, based on the number of available logical processors.

Lab Week 1 Overview

- Introduction and setup eFolio
- Run Lab VM
- Introduction to Linux

Tutorial Week 2 Overview

- Distributed Systems and Parallel Computing
- Tightly coupled system and loosely coupled system
- Reliability in the context of distributed storage

Next week: Shared Memory Parallel Computing

- POSIX Threads
- OpenMP