MPCA UNIT 4

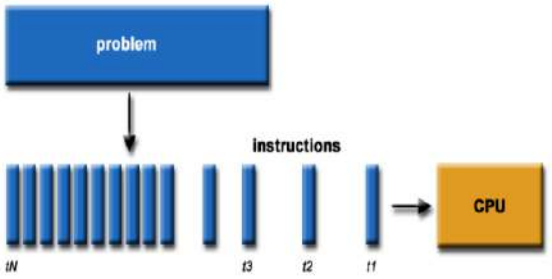
Introduction to Parallel Computing

Microprocessor & Computer Architecture (μpCA)

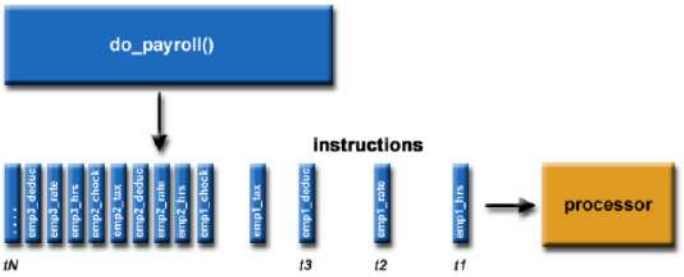
Sequential Computing

- Traditionally, software has been written for ***serial*** computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute at any moment in time.

Sequential Computing



Payroll Example

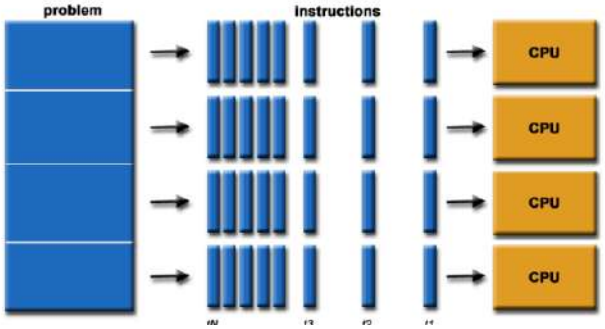


Microprocessor & Computer Architecture (μpCA)

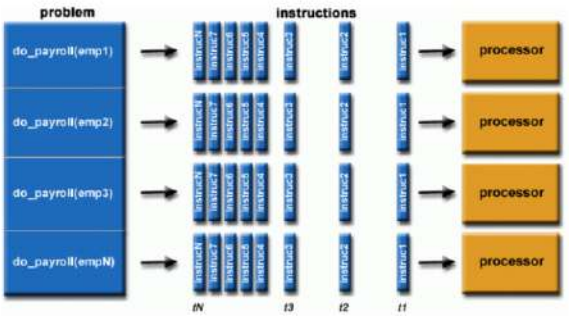
Parallel Computing

- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem.
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

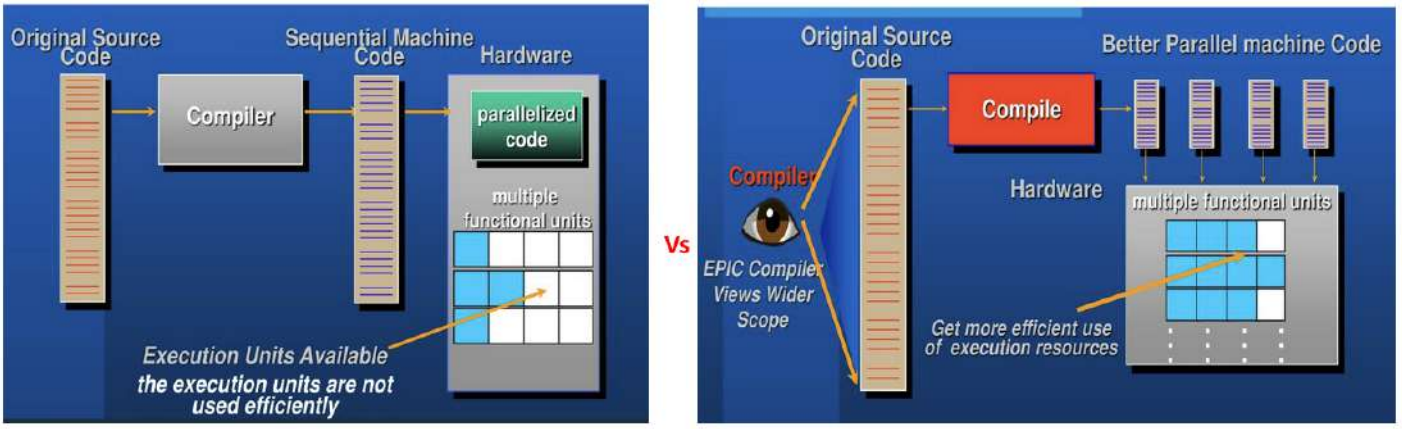
Parallel Computing



Payroll Example



Sequential Compiler Vs Parallel Compiler



The key difference is that the parallel compiler takes a holistic view of the target hardware architecture during compilation. It can analyze the code and generate parallelized machine code tailored for concurrent execution across multiple functional units. This approach exploits the parallel processing capabilities of modern hardware more effectively.

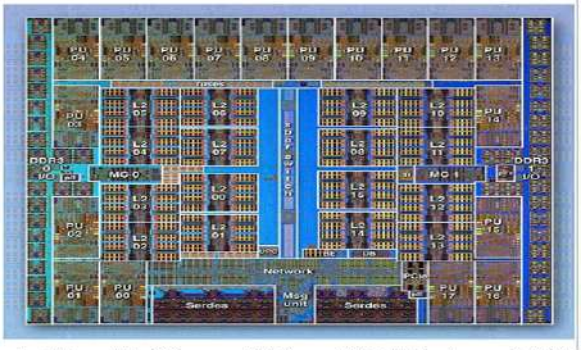
In contrast, the sequential approach first generates sequential machine code, then attempts to parallelize it, which may not fully leverage the hardware's parallel execution resources as efficiently as code parallelized during the compilation stage with full awareness of the hardware architecture.

A Computational Problem on a Parallel System

- The computational problem should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously
  - Execute multiple program instructions at any moment in time
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The compute resources are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network

Parallel Computers

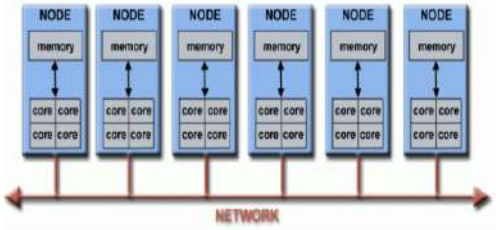
- Virtually all stand-alone computers today are parallel from a hardware perspective
  - Multiple functional units
    - L1 cache, L2 cache, branch, prefetch, decode,
    - floating-point, graphics processing (GPU), integer, etc.
  - Multiple execution units/cores
  - Multiple hardware threads



IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units(L2)

Microprocessor & Computer Architecture (µpCA)

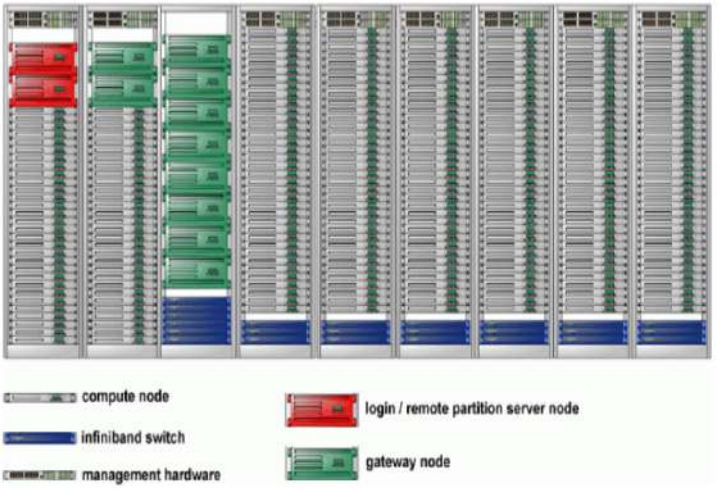
Parallel Computers



Example of typical LLNL parallel computer cluster

- Each compute node is a multi-processor parallel computer in itself
- Multiple compute nodes are networked together with an Infiniband network
- Special purpose nodes, also multi-processor, are used for other purposes

- Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters.



Microprocessor & Computer Architecture (µpCA)

Why use Parallel Computers ?

The Real World is Massively Parallel :

- In the natural world, many complex, interrelated events are happening at the same time, yet within a temporal sequence.
- Compared to serial computing, parallel computing is much better suited for modeling, simulating and understanding complex, real world phenomena.

- Main Reasons for using Parallel Programming is to
  - **SAVE TIME AND/OR MONEY**
  - **SOLVE LARGER / MORE COMPLEX PROBLEMS**

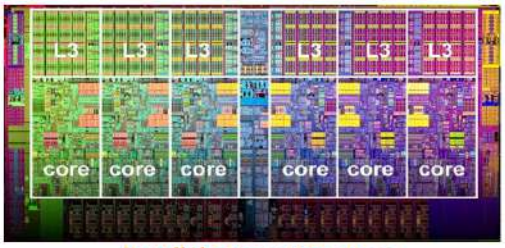
Working in parallel shortens completion time

Parallel computing can solve increasingly complex problems

Microprocessor & Computer Architecture (µpCA)

Why use Parallel Computers ?

- Main Reasons for using Parallel Programming is to
  - **PROVIDE CONCURRENCY**
  - **TAKE ADVANTAGE OF NON-LOCAL RESOURCES**
  - **MAKE BETTER USE OF UNDERLYING PARALLEL HARDWARE**



Parallel computing cores

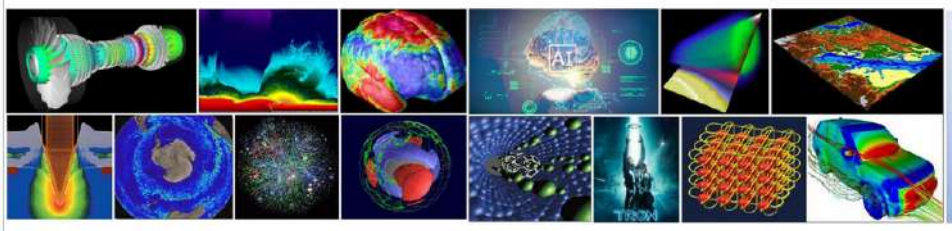
Collaborative networks

TAKE ADVANTAGE : NON-LOCAL RESOURCES

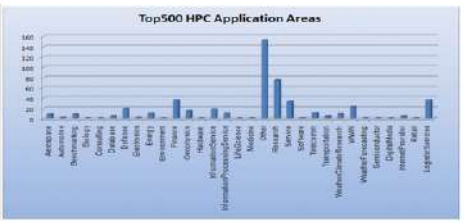
Microprocessor & Computer Architecture (µpCA)

Why use Parallel Computers ?

- Main Reasons for using Parallel Programming is to
  - **Science and Engineering**
  - **Industrial and Commercial**
  - **Global Applications**



simulating a range of complex physical phenomena



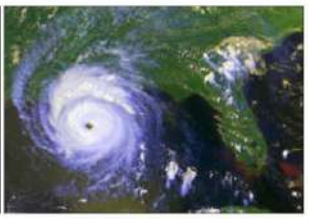
commercial applications

Top 500 Companies

Applications of Parallel Computing

Galaxy Formation

Planetary Movments



Climate Change

Rush Hour Traffic

Plate Tectonics

Weather

- Real world Phenomena can be simulated with parallel computing

Microprocessor & Computer Architecture (µpCA)

Applications of Parallel Computing

"Grand Challenge Problems"

- "Big Data", databases, data mining
- Artificial Intelligence (AI) & Machine Learning (ML)
- Web search engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Financial and economic modeling
- Management of national and multi-national corporations
- Advanced graphics and virtual reality, particularly in the entertainment industry
- Networked video and multi-media technologies

High Performance Computing

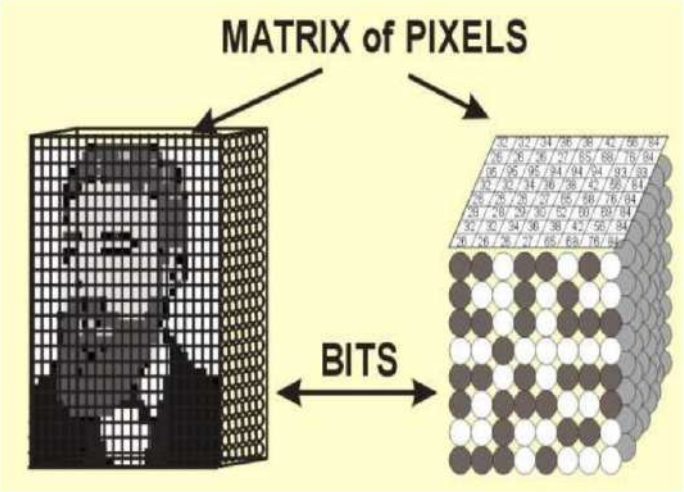
The use of the most efficient algorithms on computers
"capable of the highest performance" to solve the most demanding problems.

1. **Higher speed (solve problems faster)**
   Important when there are "hard" or "soft" deadlines;
   e.g., 24-hour weather forecast

2. **Higher throughput (solve more problems)**
   Important when we have many similar tasks to perform;
   e.g., Transaction processing

3. **Higher computational power (solve larger problems)**
   e.g., Weather forecast for a week rather than 24 hours,
   or with a finer mesh for greater accuracy

Quantifying the Problem



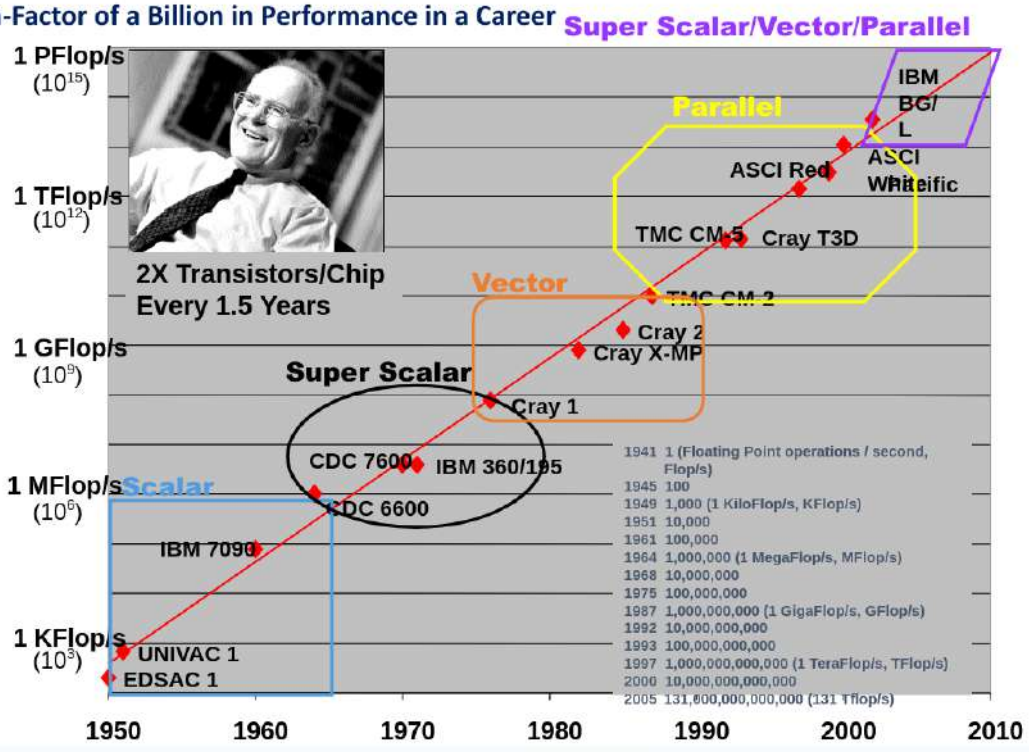
Large problems

- 10,000 x 10,000 x 10,000 grid
- 10^12 grid points
- 4x10^12 double variables
- 32x10^12 bytes
- 32 Tera-Bytes.

FLOPS, or FLOP/S: FLoating-point Operations Per Second

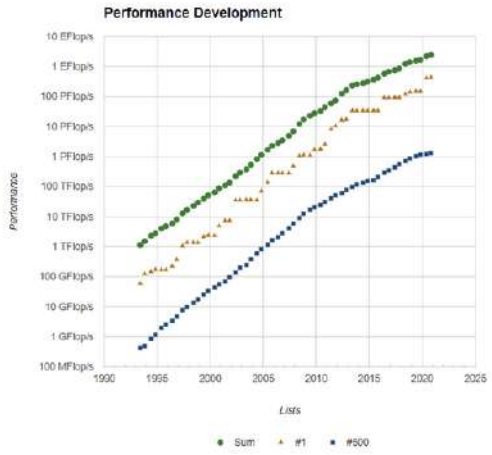
Name	Unit	Value
kiloFLOPS	kFLOPS	10^3
megaFLOPS	MFLOPS	10^6
gigaFLOPS	GFLOPS	10^9
teraFLOPS	TFLOPS	10^{12}
petaFLOPS	PFLOPS	10^{15}
exaFLOPS	EFLOPS	10^{18}
zettaFLOPS	ZFLOPS	10^{21}
yottaFLOPS	YFLOPS	10^{24}

A Growth-Factor of a Billion in Performance in a Career



Microprocessor & Computer Architecture (µpCA)

In progress : Top500.org



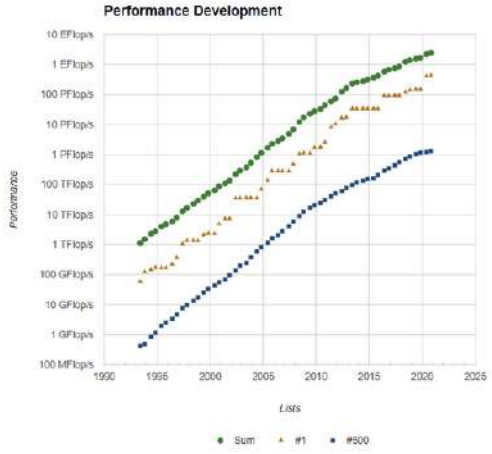
- During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computing.***

- In this same time period, there has been a greater than **500,000x** increase in supercomputer performance, with no end currently in sight.

- ***The race is already on for Exascale Computing!***
  - Exaflop = 10^{18} calculations per second

Microprocessor & Computer Architecture (µpCA)

In progress : Top500.org



- During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that ***parallelism is the future of computing.***

- In this same time period, there has been a greater than **500,000x** increase in supercomputer performance, with no end currently in sight.

- ***The race is already on for Exascale Computing!***
  - Exaflop = 10^{18} calculations per second

Microprocessor & Computer Architecture (µpCA)

Where Do We live?

Rank	System		Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
165	Supercomputer Education and Research Centre (SERC), Indian Institute of Science India	**SERC** - Cray XC40, Xeon E5-2680v3 12C 2.5GHz, Aries interconnect Cray Inc.	31,104	901.5	1,244.2	608
261	Indian Institute of Tropical Meteorology India	iDataPlex DX360M4, Xeon E5-2670 8C 2.600GHz, Infiniband FDR IBM	38,016	719.2	790.7	790
356	Indian Lattice Gauge Theory Initiative (ILGTI), Tata Institute of Fundamental Research (TIFR) India	**TIFR** - Cray XC30, Intel Xeon E5-2680v2 10C 2.8GHz, Aries interconnect , NVIDIA K20x Cray Inc.	11,424	558.8	730.7	320
392	Indian Institute of Technology Delhi India	HP Apollo 6000 Xl230/250 , Xeon E5-2680v3 12C 2.5GHz, Infiniband FDR, NVIDIA Tesla K40m HPE	22,572	524.4	1,170.1	498

Microprocessor & Computer Architecture (µpCA)

Top 5 Super Computers of India

SahasraT (Cray XC40)

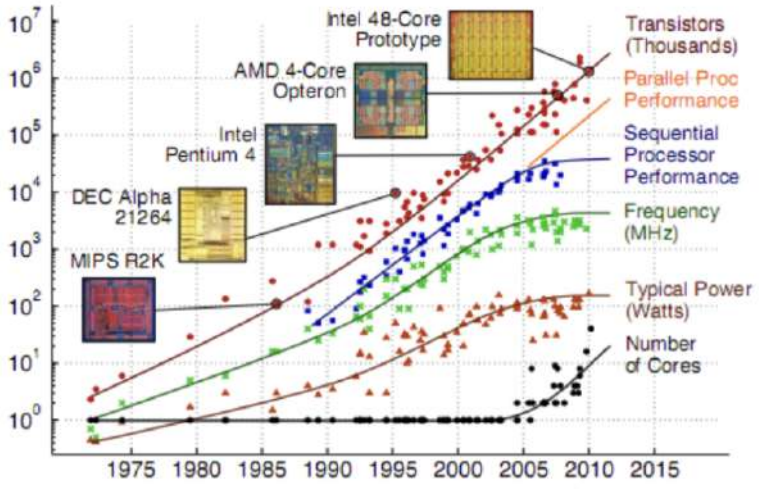
Aaditya (IBM/Lenovo System)

TIFR Colour Boson

IIT Delhi HPC
NVIDIA's GPU Tesla platform

CDACs Param Yuva-2

https://www.digit.in/general/top-5-supercomputers-in-india-29784.html



Parallel Computer Memory Architectures

Strong and weak scaling

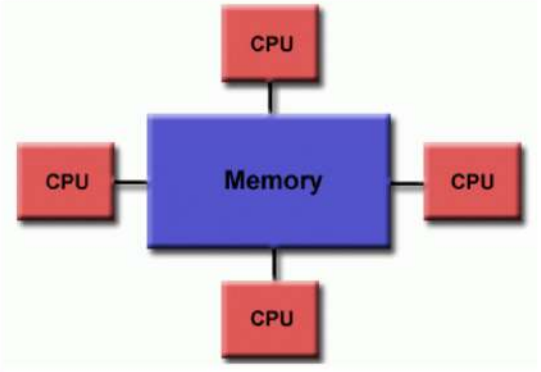
Shared Memory

General Characteristics

- 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.
- Historically, shared memory machines have been classified as ***UMA*** and ***NUMA***, based upon memory access times.

Uniform Memory Access (UMA)

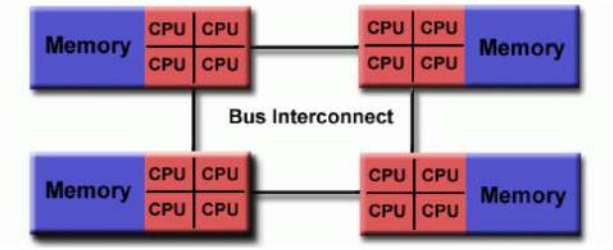
- Most commonly represented today by ***Symmetric Multiprocessor (SMP)*** machines
- Identical processors
- Equal access and access times to memory
- Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know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



Uniform memory access

Non-Uniform Memory Access (NUMA)

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



Non-uniform memory access

Advantages

- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

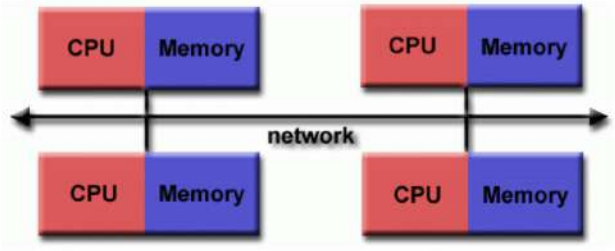
Disadvantages

- Primary disadvantage is the lack of scalability between memory and CPUs. Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management.
- Programmer responsibility for synchronization constructs that ensure "correct" access of global memory.

Distributed Memory

General Characteristics

- Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so 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.



Distributed memory

Advantages

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

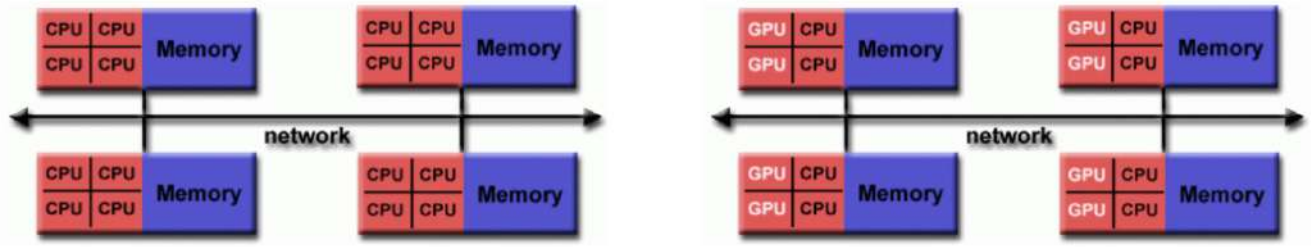
Disadvantages

- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access times - data residing on a remote node takes longer to access than node local data.

Hybrid Distributed-Shared Memory

General Characteristics

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.



- The shared memory component can be a shared memory machine and/or graphics processing units (GPU).
- The distributed memory component is the networking of multiple shared memory/GPU machines, which know only about their own memory - not the memory on another machine. Therefore, network communications are required to move data from one machine 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.
- Increased scalability is an important advantage
- Increased programmer complexity is an important disadvantage

The advantages and disadvantages listed apply to both the shared and distributed memory architectures depicted:

Advantages:

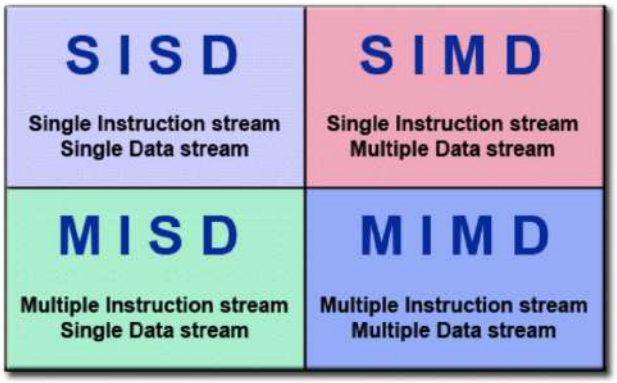
- Increased scalability is an important advantage common to both architectures. More CPUs/GPUs and memory can be added to scale up processing power and memory capacity within a shared memory system or across the networked distributed systems.

Disadvantages:

- Increased programmer complexity is an important disadvantage shared by both architectures. Coordinating access to shared memory within a machine or across distributed machines introduces complexities like synchronization, data consistency, and communication overhead that programmers must handle.

Flynn's Classical Taxonomy

- There are a number of different ways to classify parallel computers. Examples are available in the references.
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of *Instruction Stream* and *Data Stream*. Each of these dimensions can have only one of two possible states: *Single* or *Multiple*.
- The matrix below defines the 4 possible classifications according to Flynn:



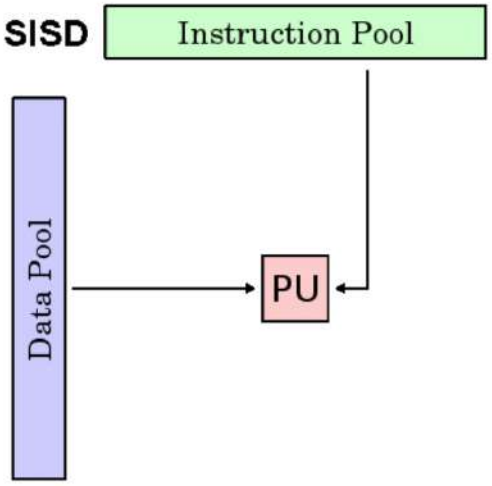
Flynn's taxonomy

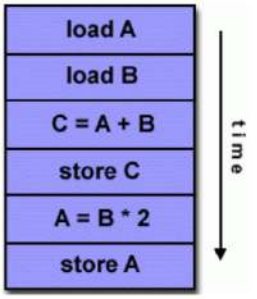
Analogy of Flynn's Classifications

- An analogy of Flynn's classification is the check-in desk at an airport
  - **SISD**: A single desk.
  - **SIMD:** Many desks and a supervisor with a megaphone giving instructions that every desk obeys.
  - **MIMD**: Many desks working at their own pace, synchronized through a central database.

Single Instruction, Single Data (SISD)

- A serial (non-parallel) computer
- **Single Instruction:** Only one instruction stream is being acted on by the CPU during any one clock cycle
- **Single Data:** Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest type of computer
- Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs.



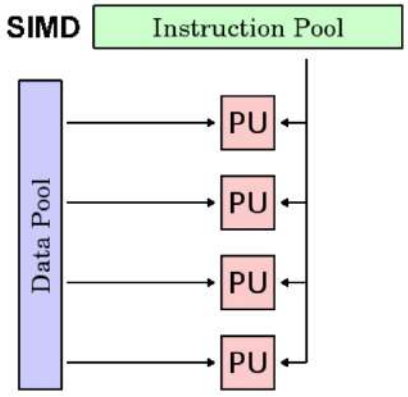


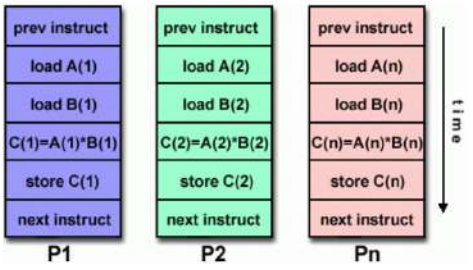
Few representatives are Intel Atom Family

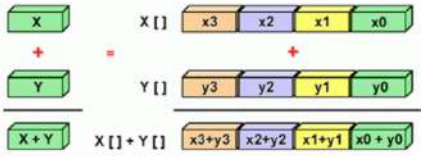
(Silverthorne, Lincroft, Diamondville, Pineview)

Single Instruction, Multiple Data (SIMD)

- A type of parallel computer
- **Single Instruction:** All processing units execute the same instruction at any given clock cycle
- **Multiple Data:** Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
  - Processor Arrays: Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV
  - Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

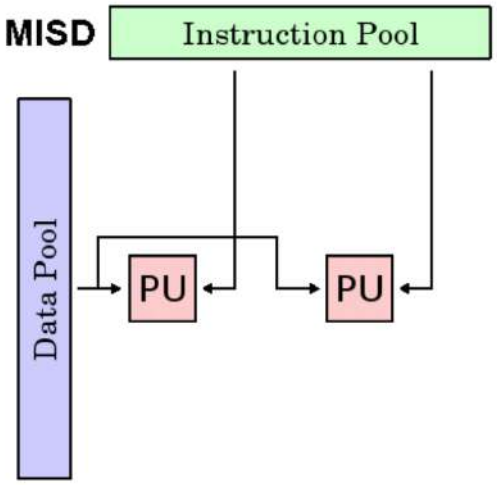


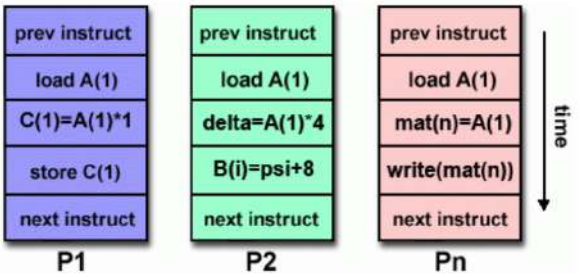




Multiple Instruction, Single Data (MISD)

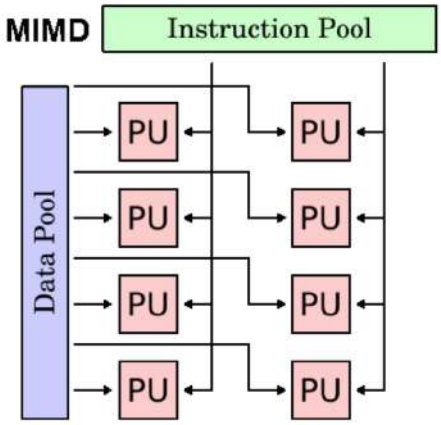
- A type of parallel computer
- **Multiple Instruction:** Each processing unit operates on the data independently via separate instruction streams.
- **Single Data:** A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever existed.
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.

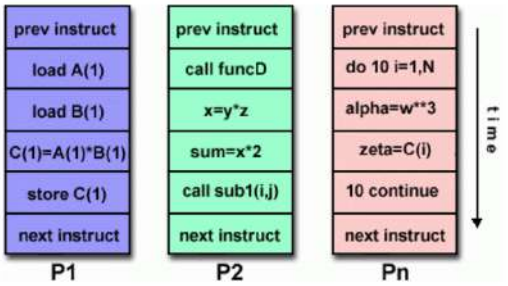




Multiple Instruction, Multiple Data (MIMD)

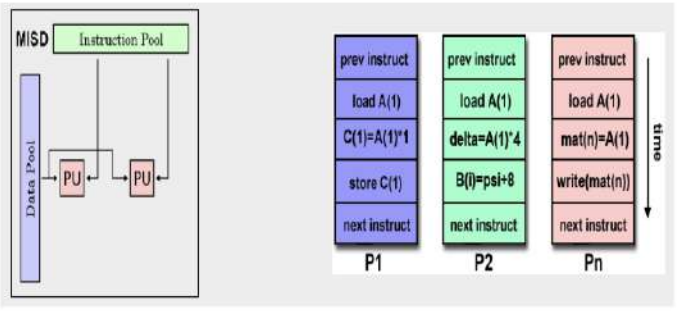
- A type of parallel computer
- **Multiple Instruction:** Every processor may be executing a different instruction stream
- **Multiple Data:** Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- **Note** Many MIMD architectures also include SIMD execution sub-components





MISD: Multiple Instruction Single Data (Does Not exist)

- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon University.
- A single data stream is fed into multiple processing units.
- Representatives: Systolic Arrays



Systolic Arrays

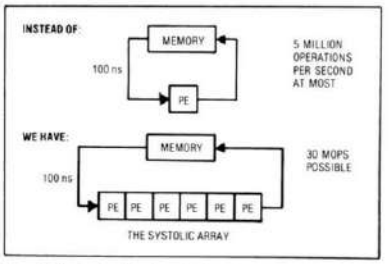


Figure 1. Basic principle of a systolic system.

Memory: heart
PEs: cells

- H. T. Kung, "Why Systolic Architectures?," IEEE Computer 1982.

Systolic Arrays :

Motivation:

- Design an accelerator that is
  - Simple and Regular design : # of Unique parts small and regular
  - High Concurrency : High Performance
  - Balanced Computation and I/O (bandwidth management).
- Idea: Replace a single Processing Element with a regular array of PEs and carefully orchestrate flow of data between PEs.

Systolic Arrays

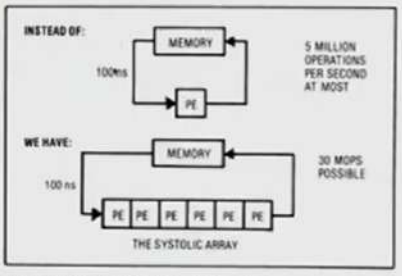


Figure 1. Basic principle of a systolic system.

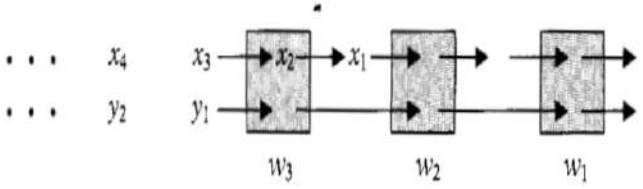
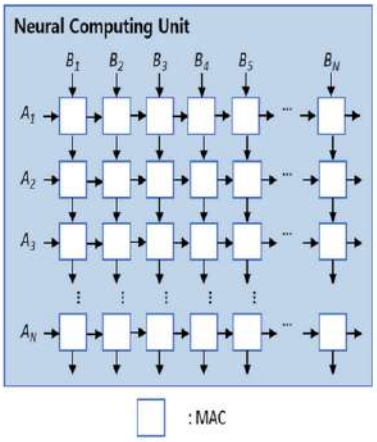


Figure 1. One-dimensional systolic array for implementing FIR filter.

Systolic Arrays : Matrix Representation



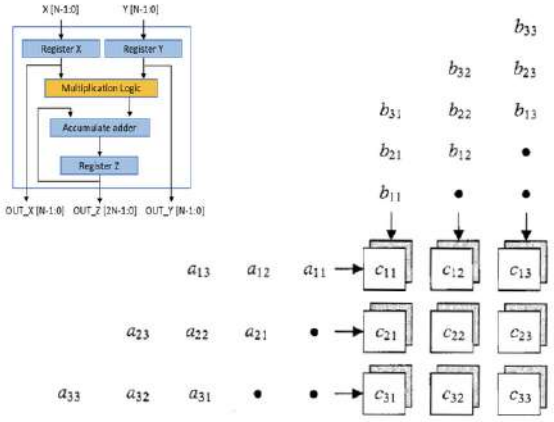
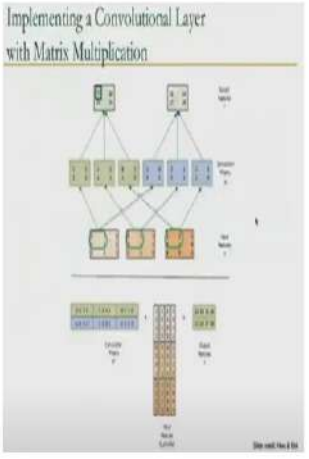
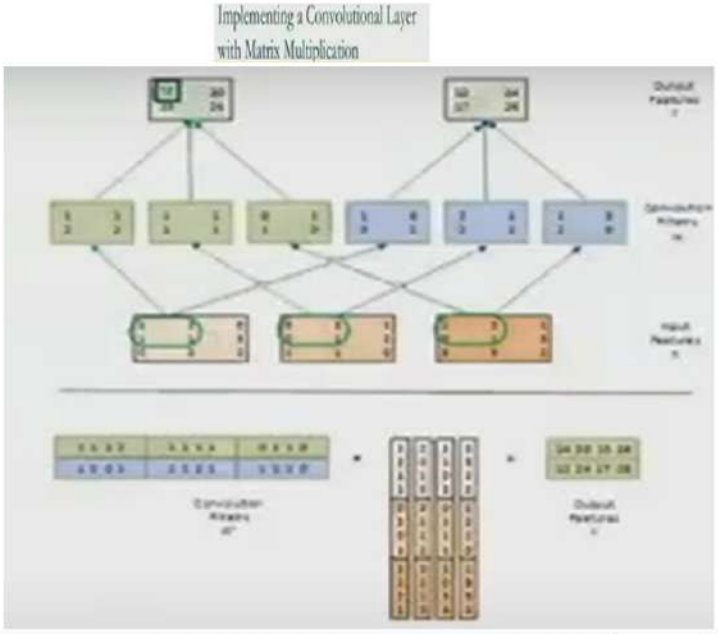


Figure 2. Two-dimensional systolic array for matrix multiplication.



Microprocessor & Computer Architecture (µpCA)

Systolic Arrays : Matrix Representation



Microprocessor & Computer Architecture (μpCA)

SIMD - Array processor

- Single Computer with Multiple parallel processors
- Processing Units are designed to work together under the supervision of a single control unit.
- Results in a single instruction stream and multiple data streams.

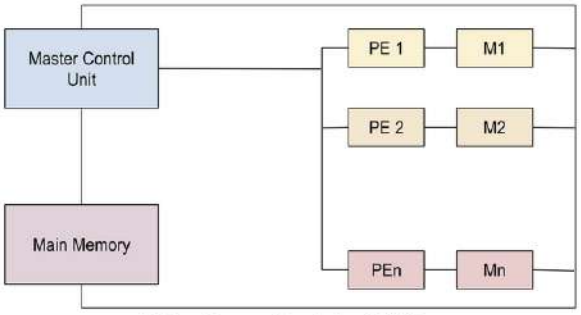


Fig-Array Processor Organisation in SIMD

Ex : An Array Processor general block diagram is shown.

- It comprises several identical processing elements (PEs), each with its local memory M
- An ALU and registers are included in each processor element.
- The master control unit controls the processing elements' actions
- It also decodes instructions and determines how they should be carried out

Microprocessor & Computer Architecture (μpCA)

SIMD - Array processor

- Single Computer with Multiple parallel processors
- Processing Units are designed to work together under the supervision of a single control unit.
- Results in a single instruction stream and multiple data streams.

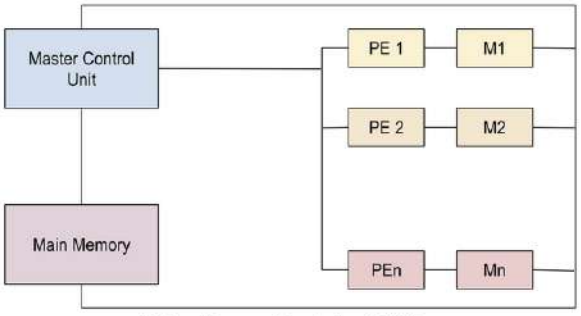


Fig-Array Processor Organisation in SIMD

Ex : An Array Processor general block diagram is shown.

- It comprises several identical processing elements (PEs), each with its local memory M
- An ALU and registers are included in each processor element.
- The master control unit controls the processing elements' actions
- It also decodes instructions and determines how they should be carried out

Supercomputing / High Performance Computing (HPC)
Using the world's fastest and largest computers to solve large problems.

Node
A standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc. Nodes are networked together to comprise a supercomputer.

CPU / Socket / Processor / Core
This varies, depending upon who you talk to. In the past, a CPU (Central Processing Unit) was a singular execution component for a computer. Then, multiple CPUs were incorporated into a node. Then, individual CPUs were subdivided into multiple "cores", each being a unique execution unit. CPUs with multiple cores are sometimes called "sockets" - vendor dependent. The result is a node with multiple CPUs, each containing multiple cores. The nomenclature is confused at times. Wonder why?

Microprocessor & Computer Architecture (μpCA)
Parallel Computing Concepts and Terminology

Task : A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.

Pipelining : Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.

Shared Memory: From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

Symmetric Multi-Processor (SMP): Shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources.

Distributed Memory: In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

Communications : Parallel tasks typically need to exchange data. There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

Synchronization : The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

Granularity : In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
Coarse: relatively large amounts of computational work are done between communication events
Fine: Relatively small amounts of computational work are done between communication events

Observed Speedup: Observed speedup of a code which has been parallelized, defined as:

$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

One of the simplest and most widely used indicators for a parallel program's performance.

Parallel Overhead

Required execution time that is unique to parallel tasks, as opposed to that for doing useful work. Parallel overhead can include factors such as:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel languages, libraries, operating system, etc.
- Task termination time

Massively Parallel

Refers to the hardware that comprises a given parallel system - having many processing elements. The meaning of "many" keeps increasing, but currently, the largest parallel computers are comprised of processing elements numbering in the hundreds of thousands to millions.

Embarrassingly (IDEALY) Parallel

Solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks.

Scalability

Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources. Factors that contribute to scalability include:

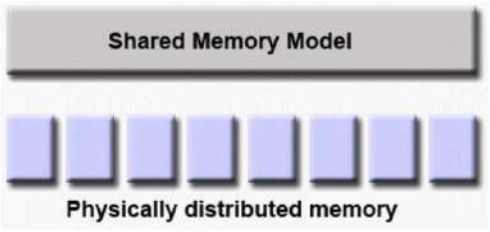
- Hardware - particularly memory-cpu bandwidths and network communication properties
- Application algorithm
- Parallel overhead related
- Characteristics of your specific application

Parallel Programming Models

- There are several parallel programming models in common use:
  - Shared Memory (without threads)
  - Threads
  - Distributed Memory / Message Passing
  - Data Parallel
  - Hybrid
  - Single Program Multiple Data (SPMD)
  - Multiple Program Multiple Data (MPMD)
- Parallel programming models exist as an abstraction above hardware and memory architectures.
- Although it might not seem apparent, these models are NOT specific to a particular type of machine or memory architecture. In fact, any of these models can (theoretically) be implemented on any underlying hardware. Two examples from the past are discussed below.

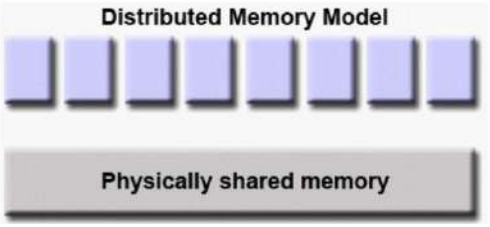
SHARED memory model on a DISTRIBUTED memory machine

Kendall Square Research (KSR) ALLCACHE approach. Machine memory was physically distributed across networked machines, but appeared to the user as a single shared memory global address space. Generically, this approach is referred to as "virtual shared memory".



DISTRIBUTED memory model on a SHARED memory machine

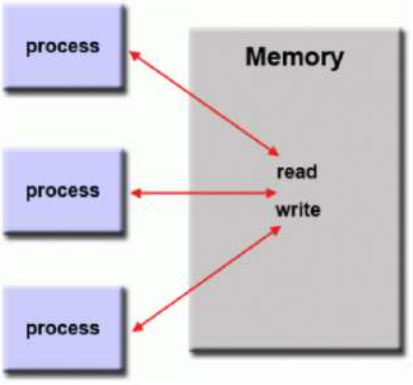
Message Passing Interface (MPI) on SGI Origin 2000. The SGI Origin 2000 employed the CC-NUMA type of shared memory architecture, where every task has direct access to global address space spread across all machines. However, the ability to send and receive messages using MPI, as is commonly done over a network of distributed memory machines, was implemented and commonly used.



- **Which model to use?** This is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.
- The following sections describe each of the models mentioned above, and also discuss some of their actual implementations.

Shared Memory Model (without threads)

- In this programming model, processes/tasks share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks / semaphores are used to control access to the shared memory, resolve contentions and to prevent race conditions and deadlocks.
- This is perhaps the simplest parallel programming model.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. All processes see and have equal access to shared memory. Program development can often be simplified.
- An important disadvantage in terms of performance is that it becomes more difficult to understand and manage *data locality*:
  - Keeping data local to the process that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processes use the same data.
  - Unfortunately, controlling data locality is hard to understand and may be beyond the control of the average user.



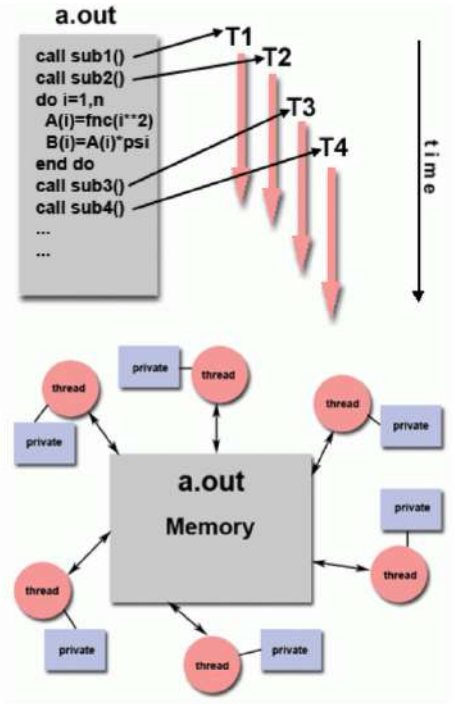
Shared memory model

Implementations

- On stand-alone shared memory machines, native operating systems, compilers and/or hardware provide support for shared memory programming. For example, the POSIX standard provides an API for using shared memory, and UNIX provides shared memory segments (shmget, shmat, shmctl, etc.).
- On distributed memory machines, memory is physically distributed across a network of machines, but made global through specialized hardware and software. A variety of SHMEM implementations are available: http://en.wikipedia.org/wiki/SHMEM.

Threads Model

- This programming model is a type of shared memory programming.
- In the threads model of parallel programming, a single "heavy weight" process can have multiple "light weight", concurrent execution paths.
- For example:
  - The main program **a.out** is scheduled to run by the native operating system. **a.out** loads and acquires all of the necessary system and user resources to run. This is the "heavy weight" process.
  - **a.out** performs some serial work, and then creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently.
  - Each thread has local data, but also, shares the entire resources of **a.out**. This saves the overhead associated with replicating a program's resources for each thread ("light weight"). Each thread also benefits from a global memory view because it shares the memory space of **a.out**.
  - A thread's work may best be described as a subroutine within the main program. Any thread can execute any subroutine at the same time as other threads.
  - Threads communicate with each other through global memory (updating address locations). This requires synchronization constructs to ensure that more than one thread is not updating the same global address at any time.
  - Threads can come and go, but **a.out** remains present to provide the necessary shared resources until the application has completed.



Threads model

Implementations

- From a programming perspective, threads implementations commonly comprise:
  - A library of subroutines that are called from within parallel source code
  - A set of compiler directives imbedded in either serial or parallel source code

In both cases, the programmer is responsible for determining the parallelism (although compilers can sometimes help).

- Threaded implementations are not new in computing. Historically, hardware vendors have implemented their own proprietary versions of threads. These implementations differed substantially from each other making it difficult for programmers to develop portable threaded applications.
- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP.***

POSIX Threads

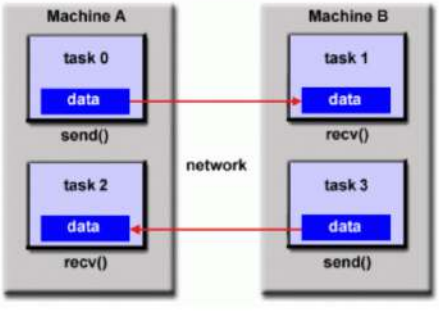
- Specified by the IEEE POSIX 1003.1c standard (1995). C Language only.
- Part of Unix/Linux operating systems
- Library based
- Commonly referred to as Pthreads.
- Very explicit parallelism; requires significant programmer attention to detail.

OpenMP

- Industry standard, jointly defined and endorsed by a group of major computer hardware and software vendors, organizations and individuals.
- Compiler directive based
- Portable / multi-platform, including Unix and Windows platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism". Can begin with serial code.
- Other threaded implementations are common, but not discussed here:
  - Microsoft threads
  - Java, Python threads
  - CUDA threads for GPUs

Distributed Memory / Message Passing Model

- This model demonstrates the following characteristics:
  - A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or across an arbitrary number of machines.
  - Tasks exchange data through communications by sending and receiving messages.
  - Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.



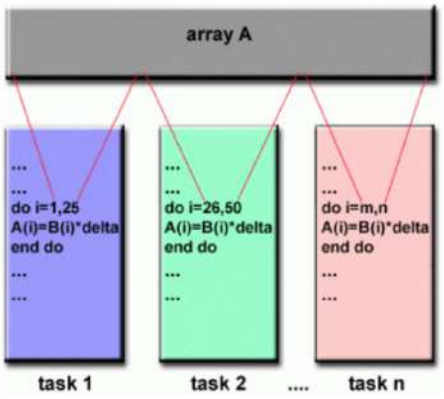
Distributed memory model

Implementations:

- From a programming perspective, message passing implementations usually comprise a library of subroutines. Calls to these subroutines are imbedded in source code. The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s. These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.
- Part 1 of the **Message Passing Interface (MPI)** was released in 1994. Part 2 (MPI-2) was released in 1996 and MPI-3 in 2012. All MPI specifications are available on the web at http://www.mpi-forum.org/docs/.
- MPI is the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work. MPI implementations exist for virtually all popular parallel computing platforms. Not all implementations include everything in MPI-1, MPI-2 or MPI-3.

Data Parallel Model

- May also be referred to as the **Partitioned Global Address Space (PGAS)** model.
- The data parallel model demonstrates the following characteristics:
  - Address space is treated globally
  - Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
  - A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
  - Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
- On shared memory architectures, all tasks may have access to the data structure through global memory.
- On distributed memory architectures, the global data structure can be split up logically and/or physically across tasks.



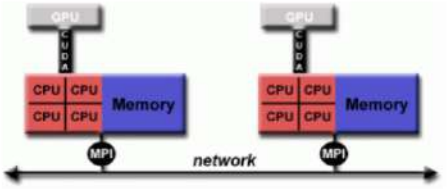
Data parallel model

Implementations:

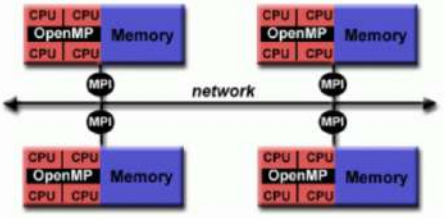
- Currently, there are several parallel programming implementations in various stages of developments, based on the Data Parallel / PGAS model.
- **Coarray Fortran:** a small set of extensions to Fortran 95 for SPMD parallel programming. Compiler dependent. More information: https://en.wikipedia.org/wiki/Coarray_Fortran
- **Unified Parallel C (UPC):** an extension to the C programming language for SPMD parallel programming. Compiler dependent. More information: https://upc.lbl.gov/
- **Global Arrays:** provides a shared memory style programming environment in the context of distributed array data structures. Public domain library with C and Fortran77 bindings. More information: https://en.wikipedia.org/wiki/Global_Arrays
- **X10: a PGAS** based parallel programming language being developed by IBM at the Thomas J. Watson Research Center. More information: http://x10-lang.org/
- **Chapel:** an open source parallel programming language project being led by Cray. More information: http://chapel.cray.com/

Hybrid Model

- A hybrid model combines more than one of the previously described programming models.
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP).
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI
- This hybrid model lends itself well to the most popular (currently) hardware environment of clustered multi/many-core machines.
- Another similar and increasingly popular example of a hybrid model is using MPI with CPU-GPU (graphics processing unit) programming.
  - MPI tasks run on CPUs using local memory and communicating with each other over a network.
  - Computationally intensive kernels are off-loaded to GPUs on-node.
  - Data exchange between node-local memory and GPUs uses CUDA (or something equivalent).
- Other hybrid models are common:
  - MPI with Pthreads
  - MPI with non-GPU accelerators



Hybrid model with MPI and CUDA

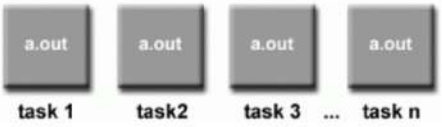


Hybrid model with MPI and OpenMP

SPMD and MPMD

Single Program Multiple Data (SPMD)

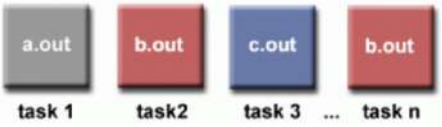
- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- SINGLE PROGRAM: All tasks execute their copy of the same program simultaneously. This program can be threads, message passing, data parallel or hybrid.
- MULTIPLE DATA: All tasks may use different data
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- The SPMD model, using message passing or hybrid programming, is probably the most commonly used parallel programming model for multi-node clusters.



SPMD model

Multiple Program Multiple Data (MPMD)

- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MULTIPLE PROGRAM: Tasks may execute different programs simultaneously. The programs can be threads, message passing, data parallel or hybrid.
- MULTIPLE DATA: All tasks may use different data
- MPMD applications are not as common as SPMD applications, but may be better suited for certain types of problems, particularly those that lend themselves better to functional decomposition than domain decomposition (discussed later under Partitioning).



MPMD model

Parallel Computing - Design Issues:

Partitioning: Splitting to Smaller Problem

Mapping: Distributing to Multiple processor

Communication: if Required (Depend on Topology)

Consolidating : The Final result

Designing Parallel Programs

Automatic vs. Manual Parallelization

- Designing and developing parallel programs has characteristically been a very manual process. The programmer is typically responsible for both identifying and actually implementing parallelism.
- Very often, manually developing parallel codes is a time consuming, complex, error-prone and iterative process.
- For a number of years now, various tools have been available to assist the programmer with converting serial programs into parallel programs. The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor.
- A parallelizing compiler generally works in two different ways:

Fully Automatic

- The compiler analyzes the source code and identifies opportunities for parallelism.
- The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
- Loops (do, for) are the most frequent target for automatic parallelization.

Programmer Directed

- Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
- May be able to be used in conjunction with some degree of automatic parallelization also.
- The most common compiler generated parallelization is done using on-node shared memory and threads (such as OpenMP).
- If you are beginning with an existing serial code and have time or budget constraints, then automatic parallelization may be the answer. However, there are several important caveats that apply to automatic parallelization:
  - Wrong results may be produced
  - Performance may actually degrade
  - Much less flexible than manual parallelization
  - Limited to a subset (mostly loops) of code
  - May actually not parallelize code if the compiler analysis suggests there are inhibitors or the code is too complex
- The remainder of this section applies to the manual method of developing parallel codes.

Thread Level & Instruction Level Parallelism

Thread Level

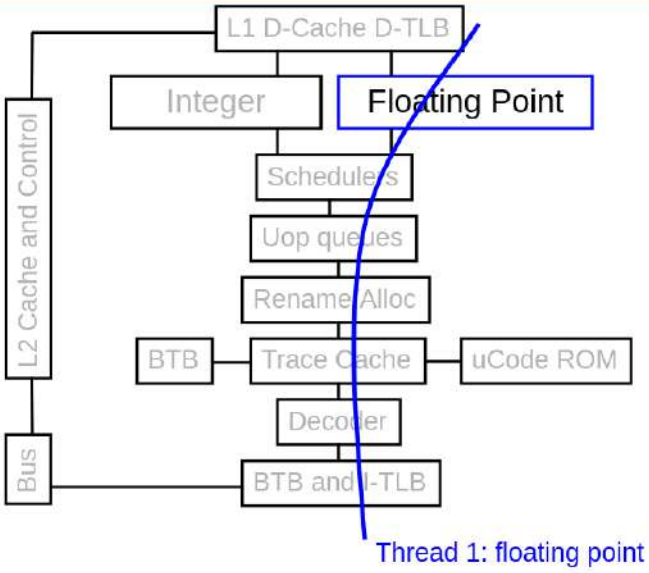
- Multi-threading vs Hyper-threading or Simultaneous Multi-threading.

Instruction Level

- Pipelining
- Super Pipelining
- Super Scalar
- Vector & Array Processing
- VLIW
- EPIC
- Parallel Computing vs Multicore Computing

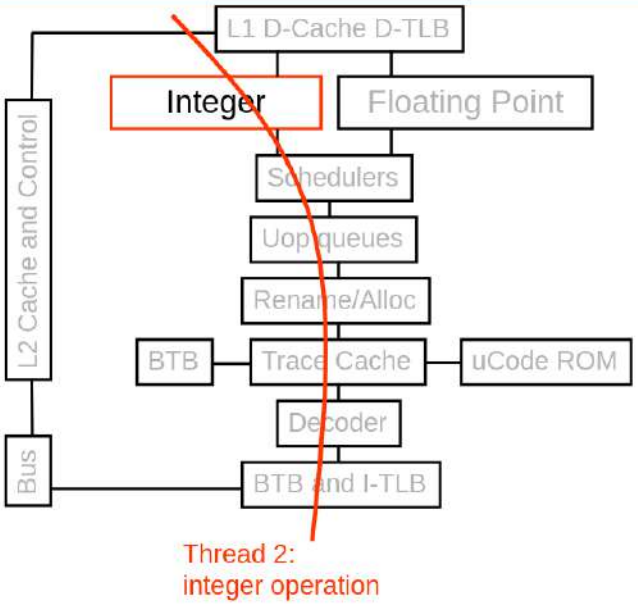
Microprocessor & Computer Architecture (µpCA)

Single thread can run at any given time



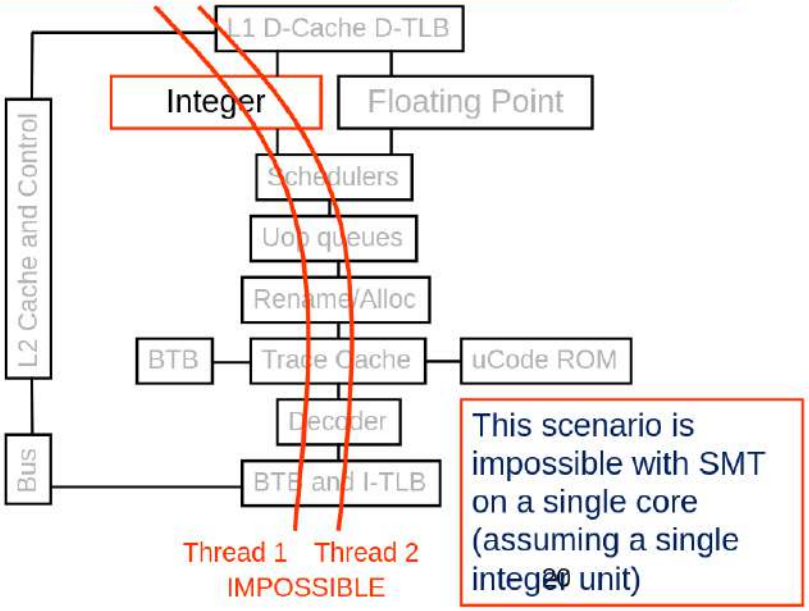
Microprocessor & Computer Architecture (µpCA)

Single thread can run at any given time

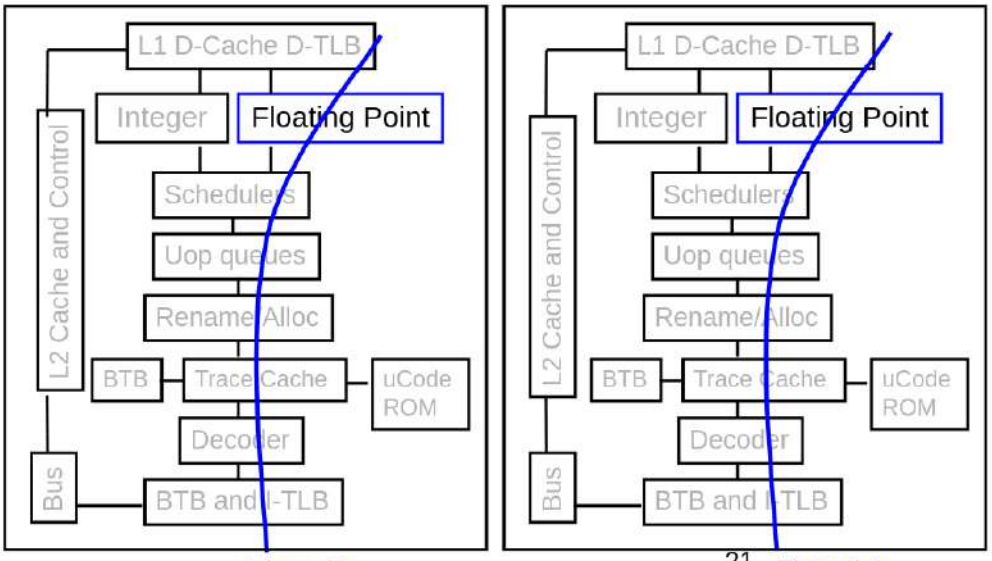


Microprocessor & Computer Architecture (µpCA)

But: Can't simultaneously use the same functional unit

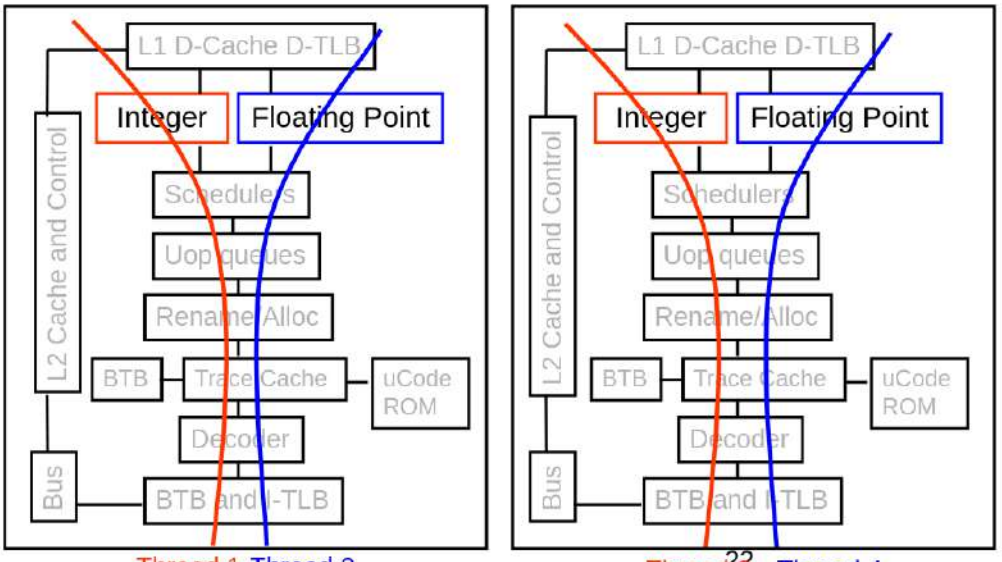


Multi-core: Threads can run on separate cores



Thread 3 Thread 4

SMT Dual-core: all four threads can run concurrently

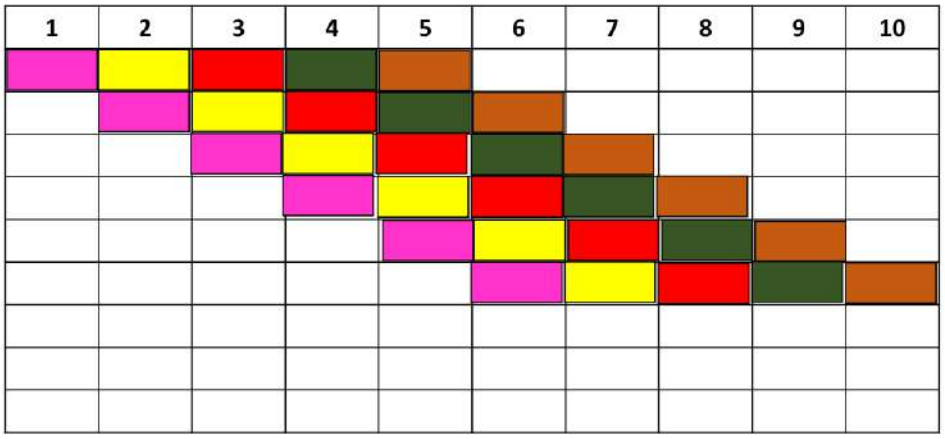


Thread 1 Thread 3 Thread 2 Thread 4

Microprocessor & Computer Architecture (µpCA)

Pipelining

Pipelining: Several instructions are simultaneously at different stages of their execution



Think of Executing following loop

```
for(i=1;i<=6;i++)
Out=i+i;
```


Microprocessor & Computer Architecture (µpCA)

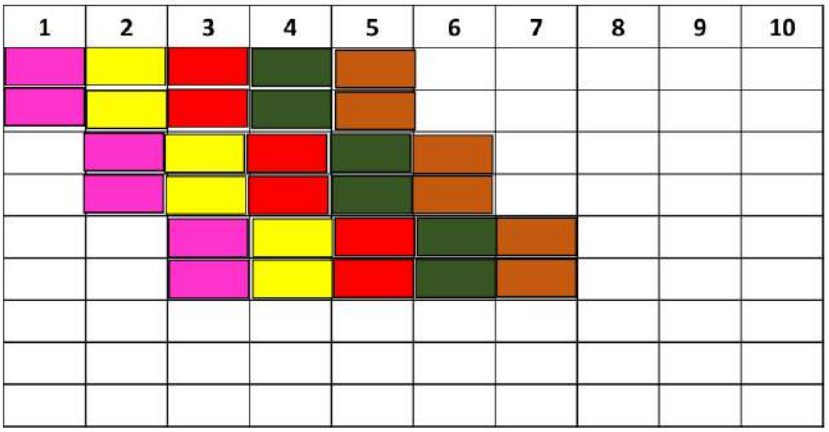
Super-Scalar

Superscalar: several instructions are simultaneously at the same stages of their execution

CPU can execute more than one Instructions per clock cycle

A Super-Scalar architecture includes parallel execution units which can execute instruction Simultaneously.

Super Pipelining

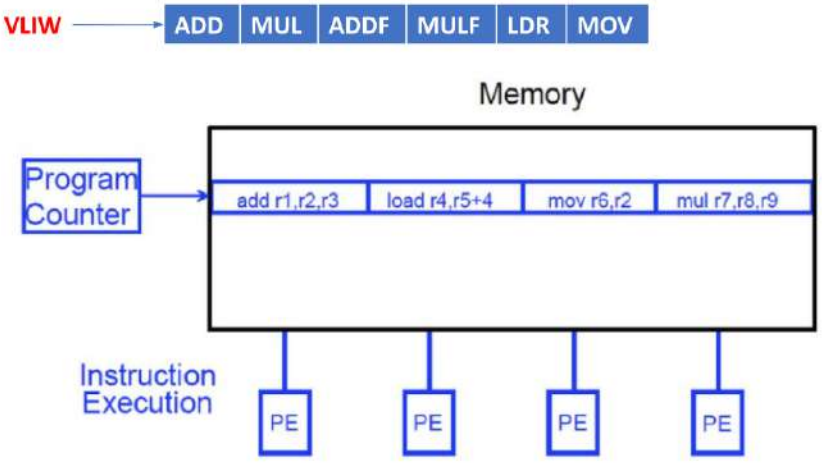


Think of Executing following loop

```
for(i=1;i<=6;i++)
Out=i+i;
```

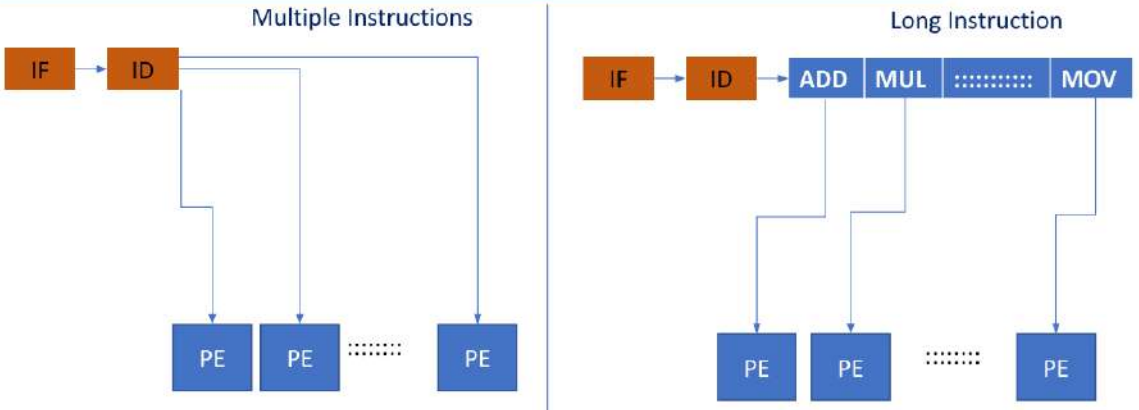

VLIW: Very Long Instruction Word

Multiple Independent Instructions are packed together by the Compiler



Microprocessor & Computer Architecture (µpCA)

VLIW vs Superscalar



Dynamic Scheduling
Complex Hardware

Static Scheduling Simpler Hardware
Compiler is used to

- Identify Independent Instruction
- Bundle the instructions

Microprocessor & Computer Architecture (µpCA)

VLIW: Very Long Instruction Word

Draw Back of VLIW

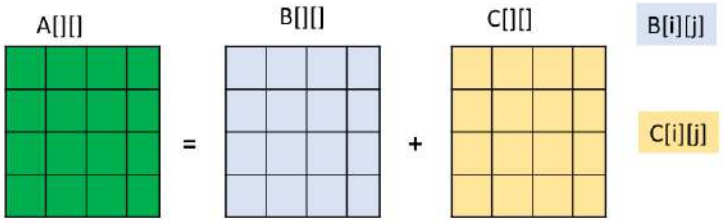
If compiler cannot find the independent instructions to for Long Instructions

- Need to Recompile the code
- Need to insert NOP's

Microprocessor & Computer Architecture (µpCA)

1. Adding Two 2D Matrices –Using a UniProcessor

```
for(i=0;i<row;i++)
  for(j=0;j<col;j++)
    A[i][j]=B[i][j]+C[i][j]
```



B[i][j]

C[i][j]

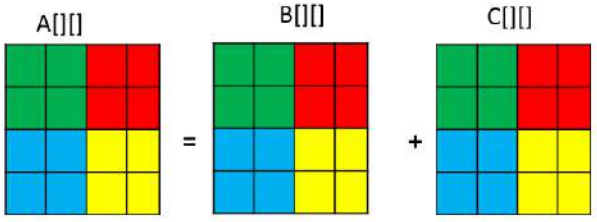
Uni Processor

Microprocessor & Computer Architecture (µpCA)

Adding Two Matrices on a Multi-Processors system

How to Utilize 4 Processors to perform Matrix addition?

```
for(i=0;i<row;i++)
  for(j=0;j<col;j++)
    A[i][j]=B[i][j]+C[i][j]
```



B[i][j]

C[i][j]

B[i][j]

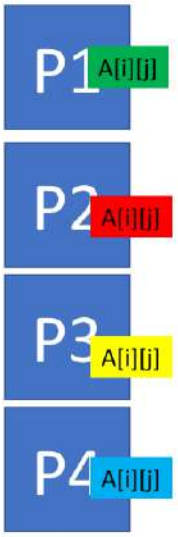
C[i][j]

B[i][j]

C[i][j]

B[i][j]

C[i][j]



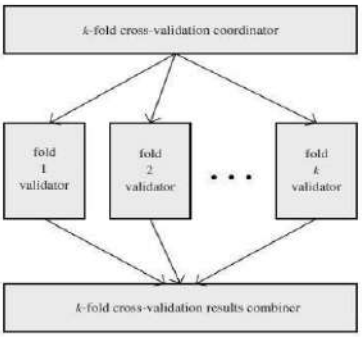
Everything is not Parallel

However, 4 Sequential operation in each Processor

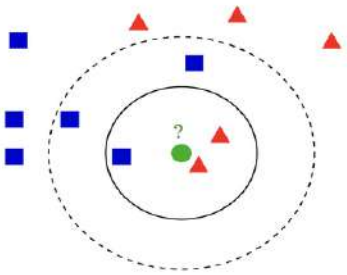
Microprocessor & Computer Architecture (μpCA)

2. Compute Euclidian Distance for a given dataset

- Common task used in machine learning which is ripe for parallelization is **distance calculation**
- Euclidean distance is a very common metric which requires calculation over and over again in numerous algorithms.
- The individual distance calculations of successive iterations are not dependent on other calculations of the same iteration, these calculations could be performed in parallel.



Parallel k-Nearest Neighbor

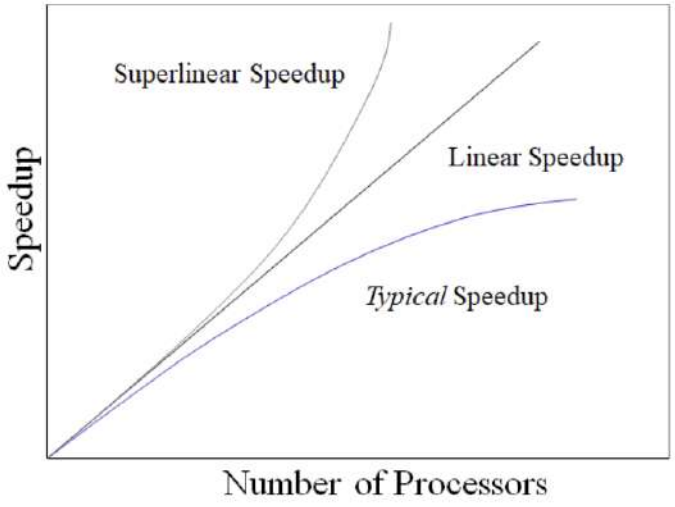


the key observation that makes distance calculation ripe for parallelization is that the individual distance computations between a given point and all other points are independent of each other. In other words, the calculation of the distance between point A and point B does not depend on the calculation of the distance between point A and point C, or any other point.

This independence allows us to split the distance calculations across multiple processors or cores, with each processor or core responsible for computing a subset of the distances in parallel. This parallel computation can significantly reduce the overall time required to perform the distance calculations, especially when dealing with large datasets or high-dimensional feature spaces

Microprocessor & Computer Architecture (μpCA)

Speed up vs Number of Processors



Can all types of problems be solved using Parallel Computing?

```
for(j=1;j<n;j++)
    A[j]=B[j]+C[j]+A[j-1]
```

The Above program cannot be fully parallelized due to dependency between the Instructions

Parallel Execution – Design Issues

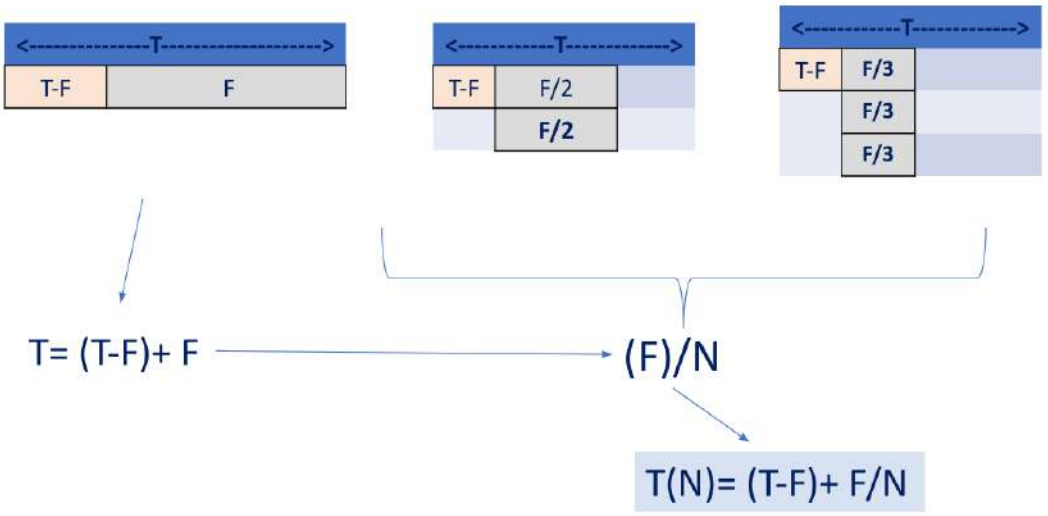
A program (or algorithm) which can be parallelized can be split up into two parts:

- A part which cannot be parallelized
- A part which can be parallelized

- T = Total time of serial execution
- T - F = Total time of non-parallizable part
- F = Total time of parallizable part (when executed serially, not in parallel)

T= F+ (T-F)

Parallel Execution – Design Issues



Parallel Computing - SOLVED EXAMPLE 1

The total time to execute a program is set to 1. The parallelizable part of the programs consumes 60% of the execution time. What is the execution time of the program when executed on 2 processor ?

Solution:

The parallelizable part is thus equal = **0.6.**

Time for non-parallelizable part is **1-0.6= 0.4 .**

The execution time of the program with a parallelization factor of 2 (2 threads or CPUs executing the parallelizable part, so N is 2) would be:

$$T(N) = (T - F) + F / N$$

$$T(2) = (1-0.6) + 0.6 / 2$$
$$= 0.4 + 0.6 / 2$$
$$= 0.4 + 0.3$$
$$= 0.7$$

Microprocessor & Computer Architecture (μpCA)

Parallel Computing - SOLVED EXAMPLE 1

Making the same calculation with a parallelization factor of **5 instead of 2** would look like this:

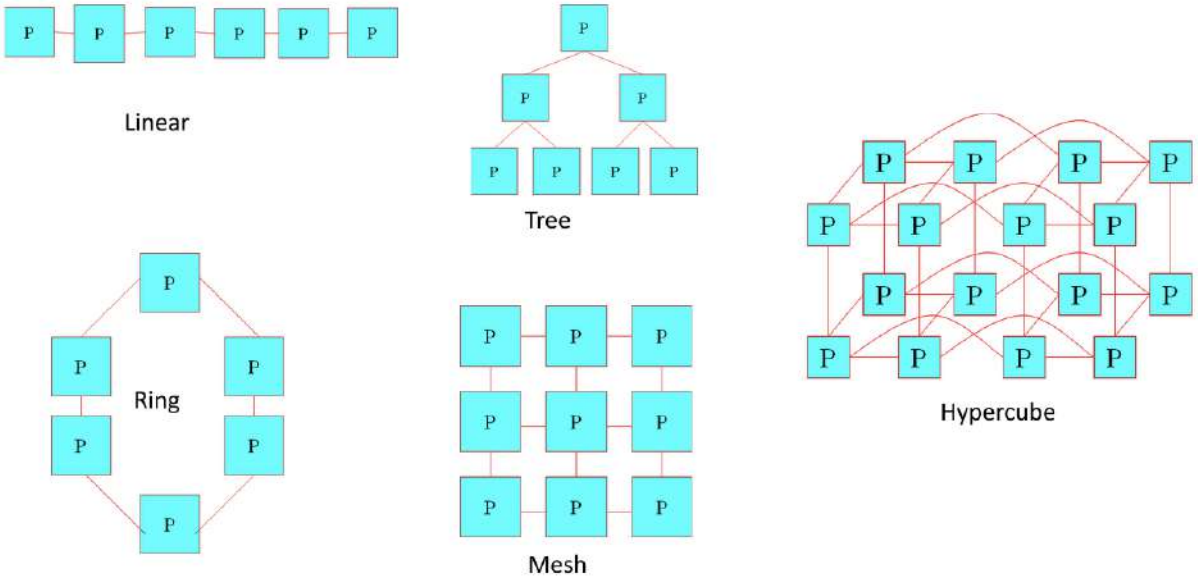
T(5) = (1-0.6) + 0.6 / 5
= 0.4 + 0.6 / 5
= 0.4 + 0.12
= 0.52

Conclusion, by increasing the number of processing unit will not contribute in improved execution time. Instead, parallelization in the program need to be improved.

i.e., writing parallel program make sense

Microprocessor & Computer Architecture (μpCA)

Issues with Communication Time & Parallel Computing Topology?

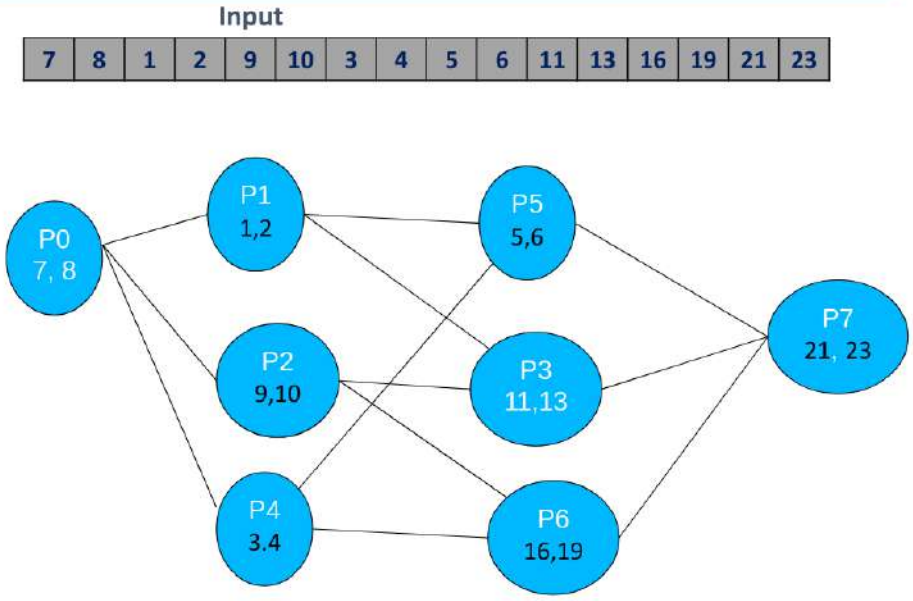


Parallel Computation - Challenges!!

- Different (Parallel) Programming Skills required as Topologies changes.
- Communication Cost (Time) changes according to Topologies.
- Through understanding of the Hardware is required to write program to utilize the computational power fully

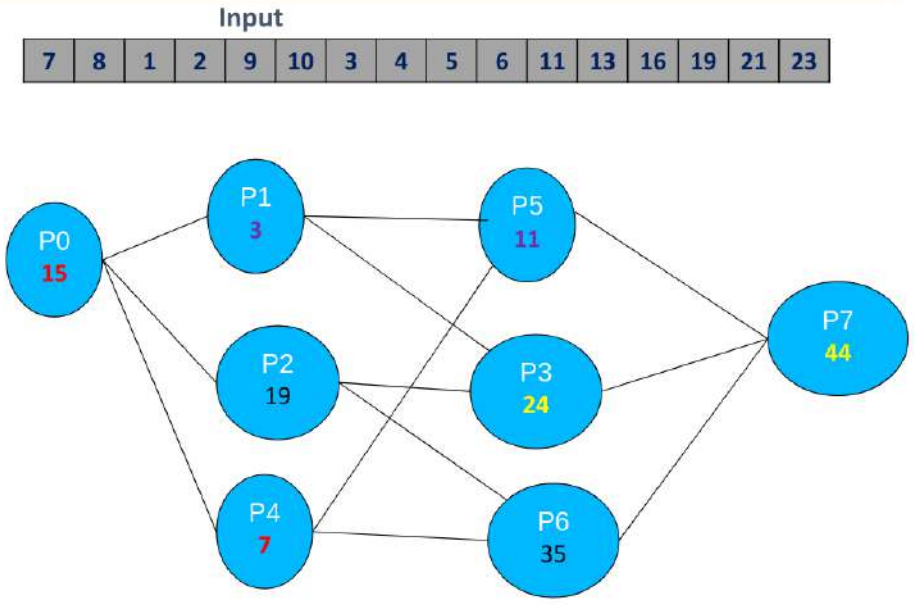
Microprocessor & Computer Architecture (µpCA)

Summation (Hypercube SIMD)



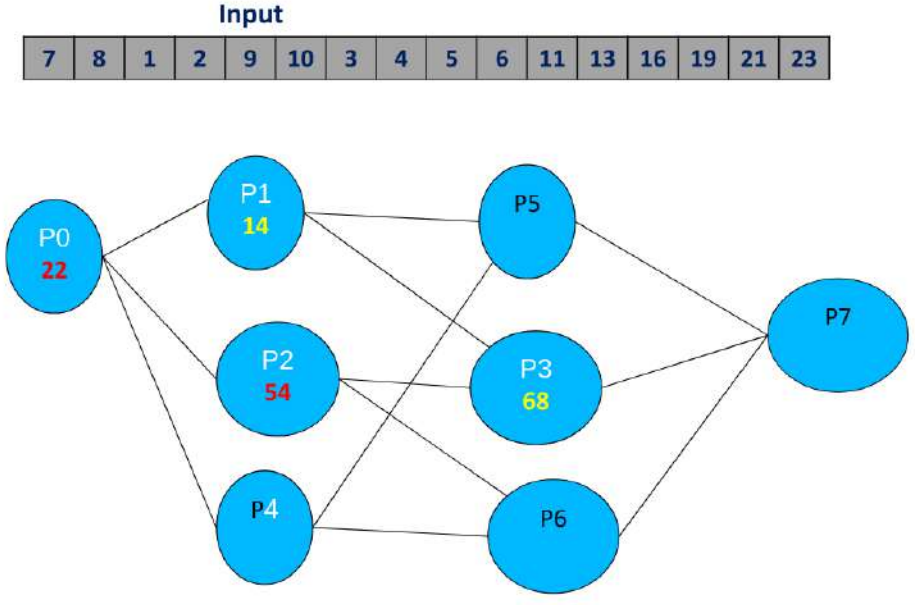
Microprocessor & Computer Architecture (μpCA)

Summation (Hypercube SIMD)



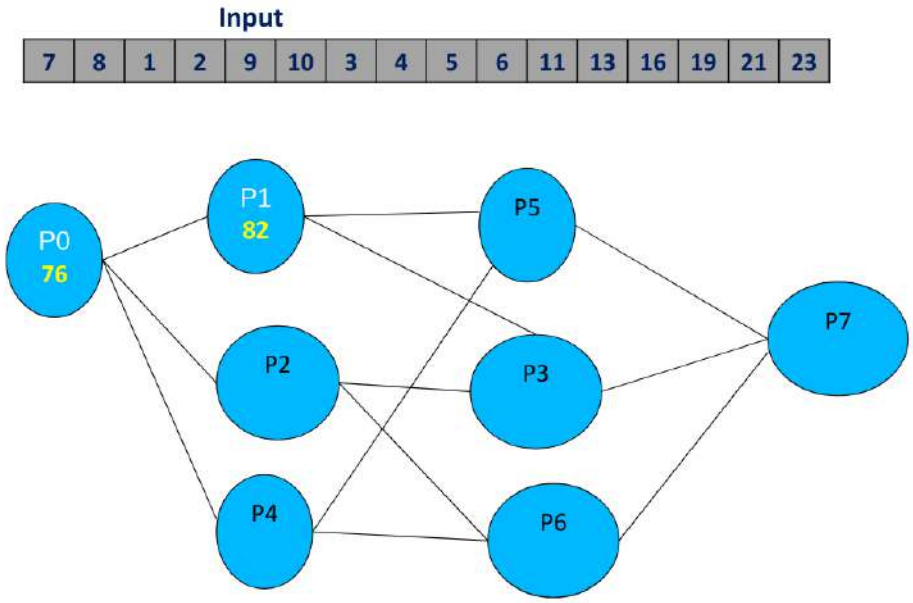
Microprocessor & Computer Architecture (μpCA)

Summation (Hypercube SIMD)



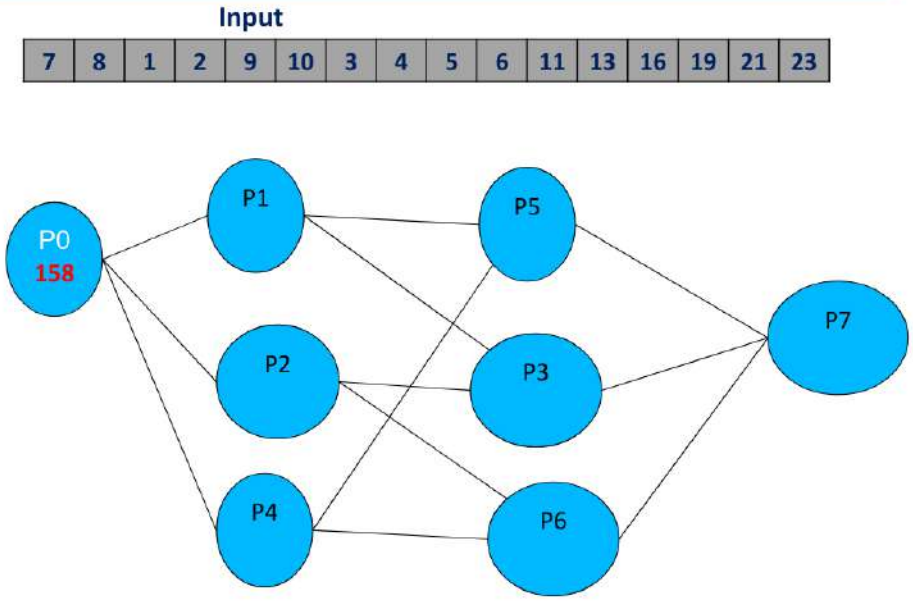
Microprocessor & Computer Architecture (μpCA)

Summation (Hypercube SIMD)



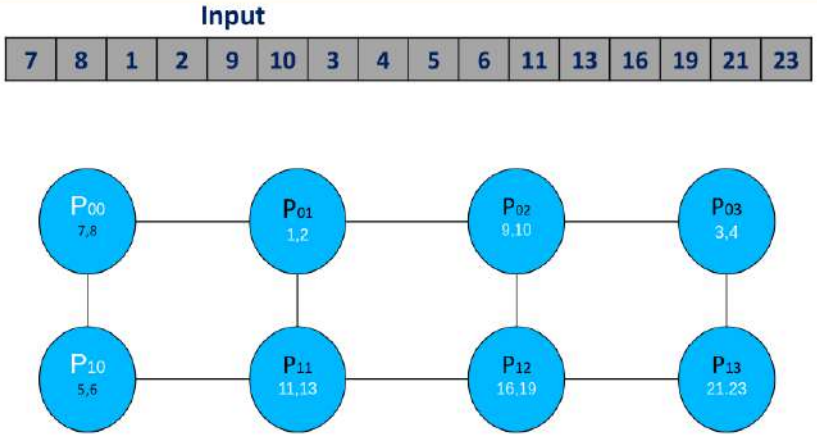
Microprocessor & Computer Architecture (μpCA)

Summation (Hypercube SIMD)



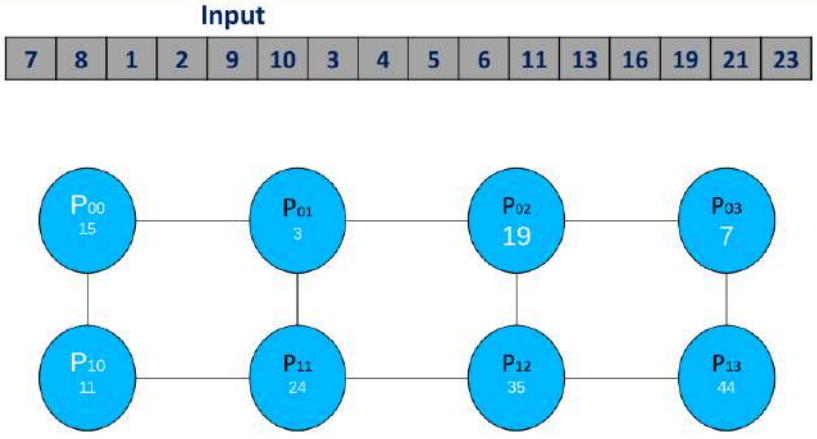
Microprocessor & Computer Architecture (μpCA)

Summation (MESH-SIMD)



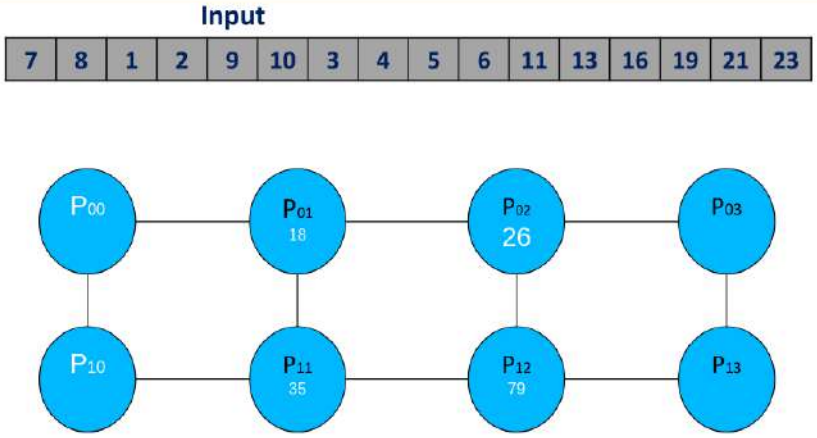
Microprocessor & Computer Architecture (μpCA)

Summation (Hypercube SIMD)



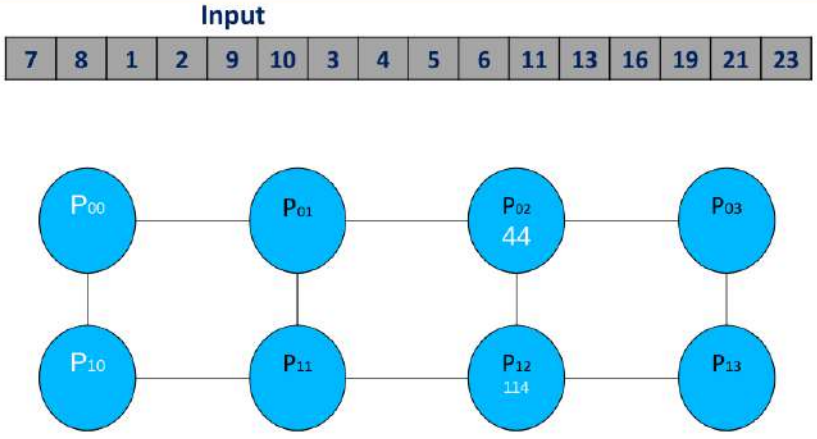
Microprocessor & Computer Architecture (μpCA)

Summation (Hypercube SIMD)



Microprocessor & Computer Architecture (μpCA)

Summation (Hypercube SIMD)



Parallel Computing: Amdahl's Law

Amdahl's Law :

Named after computer scientist Gene Amdahl

Often used in parallel computing to predict the theoretical speedup when using multiple processors.

States that the maximum speedup possible in parallelizing an algorithm is limited by the sequential portion of the code.

Amdahl's law can be formulated in the following way:

Speedup = TS / TN

Where TS - proportion of execution time to be computed sequentially.
TN - proportion of execution time of the code to be executed in parallel.

Microprocessor & Computer Architecture (µpCA)

Parallel Computing: Amdahl's Law – Numerical Examples

Speedup = TS/TN

= TS/(1-F)TS+(F/N)TS

= **1/(1-F)+(F/N)**

Where, F – Fraction time or Proportion of parallelization code
N - Number of Processors

Parallel Computing: Amdahl's Law – Numerical Examples

1. Suppose you want to perform summation of 1000 numbers. What is the speedup if the fraction of sequential computation is 60% that has 25 processors.

Solution:

Speedup = 1/(1-F)+(F/N)

= 1 /(1-0.4)+(0.4/25)

= **1.623**

Microprocessor & Computer Architecture (μpCA)

Parallel Computing: Amdahl's Law – Numerical Examples

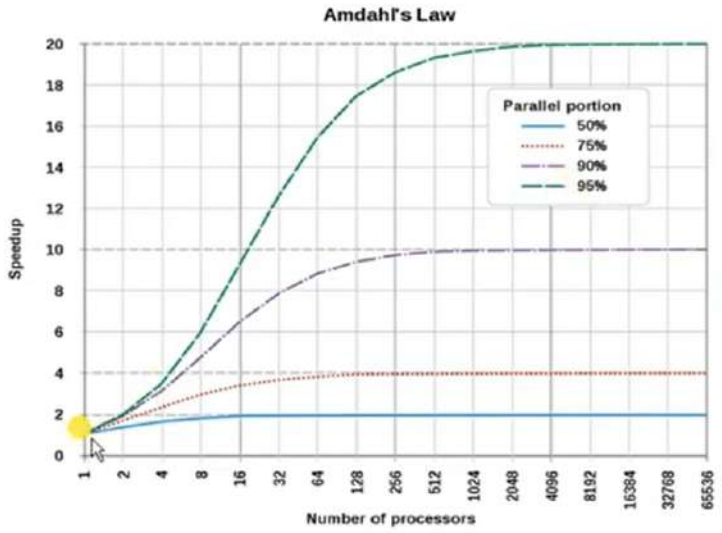
2. Suppose you want to perform summation of 1000 numbers. What is the speedup if the fraction of parallel computation is 90% that has 100 processors.

Solution:

Speedup = 1/(1-F)+(F/N)

= 1 /(1-0.9)+(0.9/100)

=**9.1743**



Parallel Computing: Amdahl's Law – Numerical Examples

Suppose you want to perform two sums:

1. Sum of 10 scalar variables and
2. A matrix sum of a pair of 2D arrays, with dimensions 10 x 10.

What is the speedup you get with 10 vs 100 processors?

Also, calculate the speedups assuming the matrices grow 100 x 100.

Solution:

1. Sum of 10 scalar variables- **sum = sum0+sum1+sum2+……+sum9**
2. A matrix sum of a pair of 2D arrays, with dimensions 10 x 10.

   Sum0= a[0][j]+b[0][j] - on processor 1/ CPU1

   Sum1= a[1][j]+b[1][j] - on processor 2/ CPU2 and so on…

   ……

   Sum9= a[9][j]+b[9][j]

Total of 110 opérations

100/110= **0.909** Parallelizable for 100 processors

10/110= 0.091 parallelizable or (1-909=**0.091**) for 10 processors.

On 10 processor	On 100 processor
Speedup = 1/(.091 + .909/10) = 1/0.1819 = 5.5	**Speedup = 1/(.091 + .909/100) = 1/0.10009 = 10.0**

Parallel Computing: Amdahl's Law – Limitations / Disadvantages

- Amdahl's law is a fatal limit to the usefulness of parallelism.
- Do not consider Size of the Program.
- Ignores communication cost

Gustafson's Law

Gustafon's Law: The proportion of the computations that are sequential, normally decreases as the problem size increases.

Note: Gustafson's law is a "observed phenomena" and not a theorem

- Rather than assuming that the problem size is fixed, assume that the parallel execution time is fixed.
- As problem size is increases, increase the parallel processing units (N).
- Gustafson, makes the case that the serial section of the code does not increase with the problem size.

A Driving Metaphor – Amdahl's Law and Gustafson's Law

Amdahl's Law approximately suggests:

" Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph. No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city. Since it has already taken you 1 hour and you only have a distance of 60 miles total; going infinitely fast you would only achieve 60 mph. "

- **Here Fixed Distance is Code size.**
- **Amdahl's law assumes code size to be constant.**

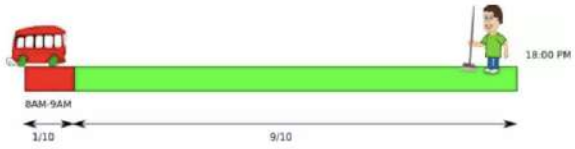
Gustafson's Law approximately states:

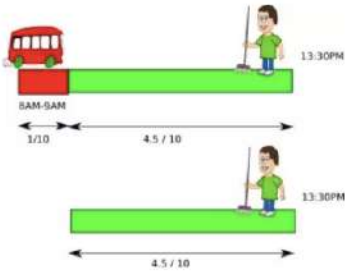
"Suppose a car has already been traveling for some time at less than 90mph. Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled. For example, if the car spent one hour at 30 mph, it could achieve this by driving at 120 mph for two additional hours, or at 150 mph for an hour, and so on."

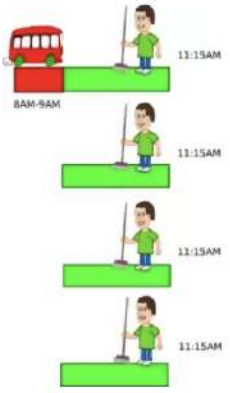
Here, it is assumed that the Distance may vary, as the distance is more, chances of increased speedup is more

Microprocessor & Computer Architecture (µpCA)

Metaphor – Amdahl's Law







Metaphor – Gustafson's Law

Time Calculation = 2-1*(1/10)= 1.9 times speed of 1 person

Time Calculation = 4-3*(1/10)= 3.7 times speed of 1 person

Microprocessor & Computer Architecture (µpCA)

Parallel Computing: Gustafson's Law

Gustafson's Law :

Named after computer scientist John L. Gustafson

Gives the speedup in the execution time of a task that theoretically gains from parallel computing.

Addresses based on the limitations of Amdahl's Law (Fixed problem size).

Gustafson's Law can be formulated as

Scaled Speedup = N-(N-1)*S

Where N – Number of processors
S – Percentage of serial section.

Comparision of Amdahl's Law vs Gustafson's Law

Consider a code that can be parallelized. The proportion of serial computation is 5% with 20 processors.
Compare the performance with respect to both the laws.

Gustafon's Law

Scaled Speed-up = 20 + (1-20)*0.05
=20-0.95
=19.05

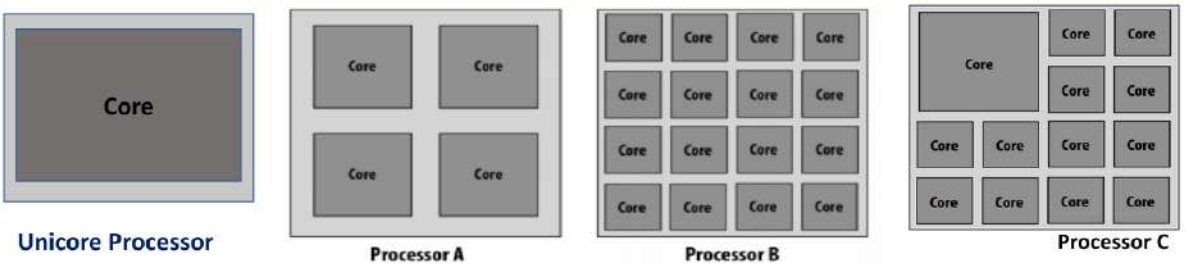
Amdahl's Law

Speed-up = 1/0.05 + (0.95/20)
=10.25

Limitations Of Single Core

- **The Power Wall**
  - o Limit on the scaling of clock speeds.
  - o Ability to handle on-chip heat has reached a physical limit.
- **The Memory Wall**
  - o Need for bigger cache sizes.
  - o Memory access latency still not in line with processor speeds
- **The ILP Wall**
  - o Identifying Implicit Parallelism within the threads is limited in many Application
  - o Dependency
  - o Hardware Restrictions such as, Instruction Window Size (How many instructions can be fetched at a time).

Resource vs Power



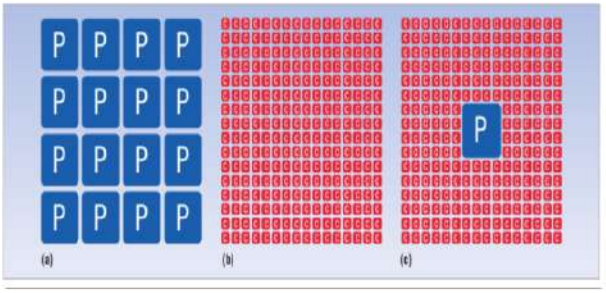
P = Total processing resources used to make Uni-core processor (e.g., transistors on a chip).
R= Resources dedicated to each processing core.

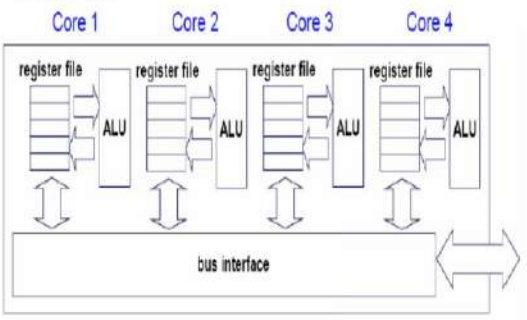
Let N=16

- Unicore Processor is made up of ***P=R=16*** resources.
- Each core in Processor A is made of ***R= 4*** resources.
- Each core in Processor B is made of ***R=1*** Resources.
- 1 Core of Processor C is made of ***R=4*** Resources and other 12 cores are made up of ***R=1*** Resources

Homogeneous : Multiple Core Architecture

Identical processor cores support same Instruction set Architecture (ISA)

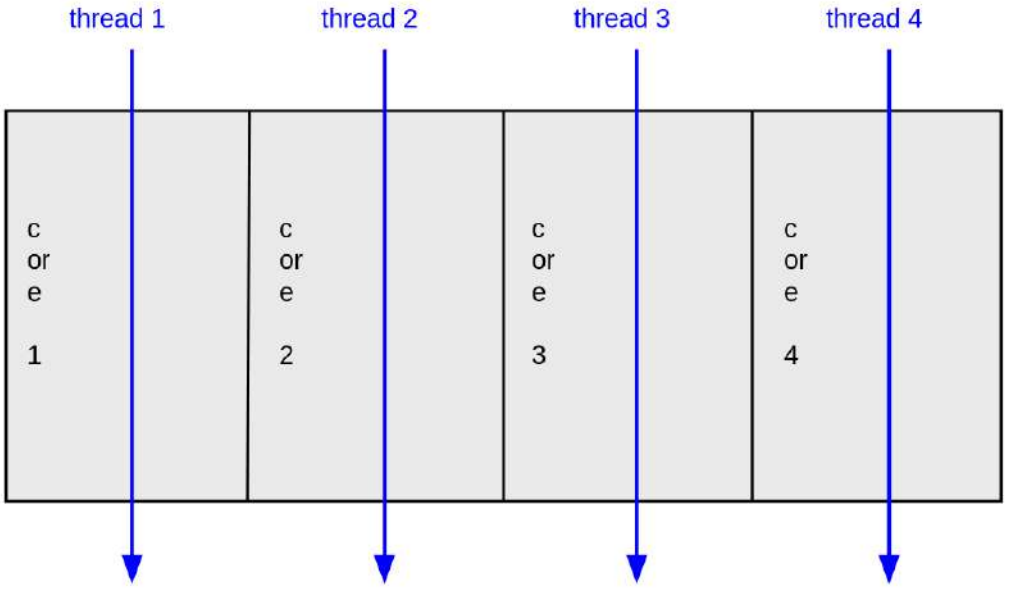




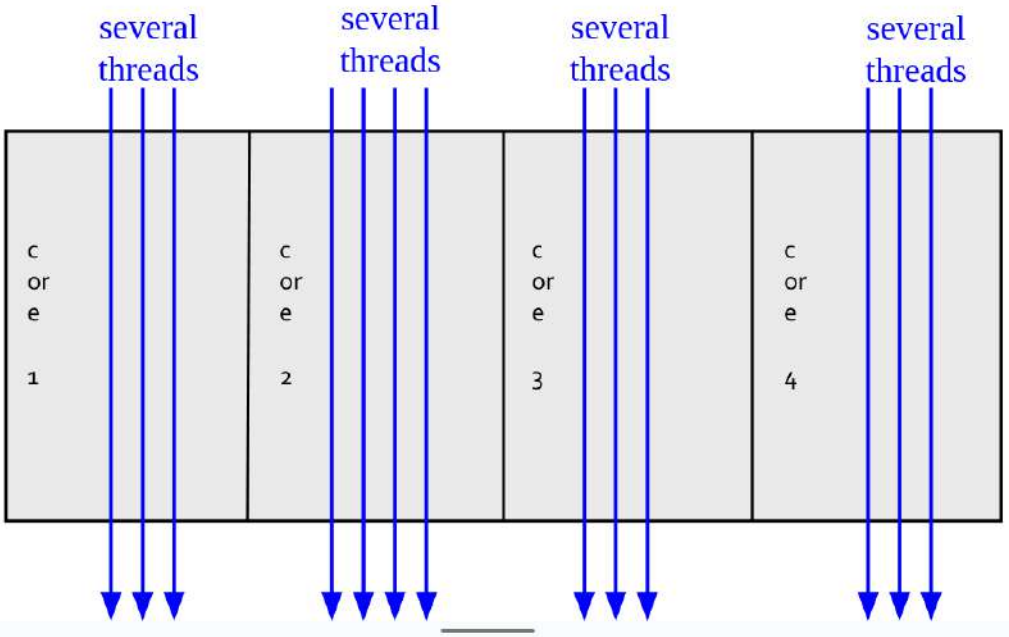
Multi-core CPU chip

For example – MPC8641, Intel Core Duo

Multi-core Architectures



The cores run in parallel (like on a uniprocessor)



Amdahl's Law for Multicore Processor

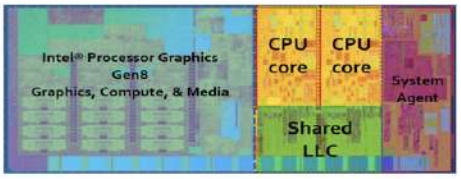
$$\text{Speedup} = \frac{1}{\frac{1 - F}{\text{Perf}(R)} + \frac{F * R}{\text{Perf}(R)*N}}$$

Note: This slide is only to realize that Speed UP measurement is different for Multicore processor over Multi Processor.

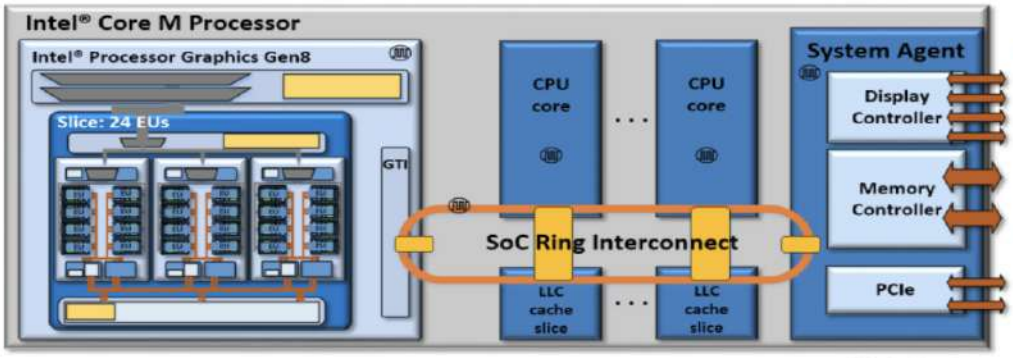
This need lot of discussion which will ***not be done*** as part of course (MPCA UE20CSE252).

Heterogeneous : Multiple Core Architecture

Non-identical processor cores, support different Instruction Set Architecture (ISA).



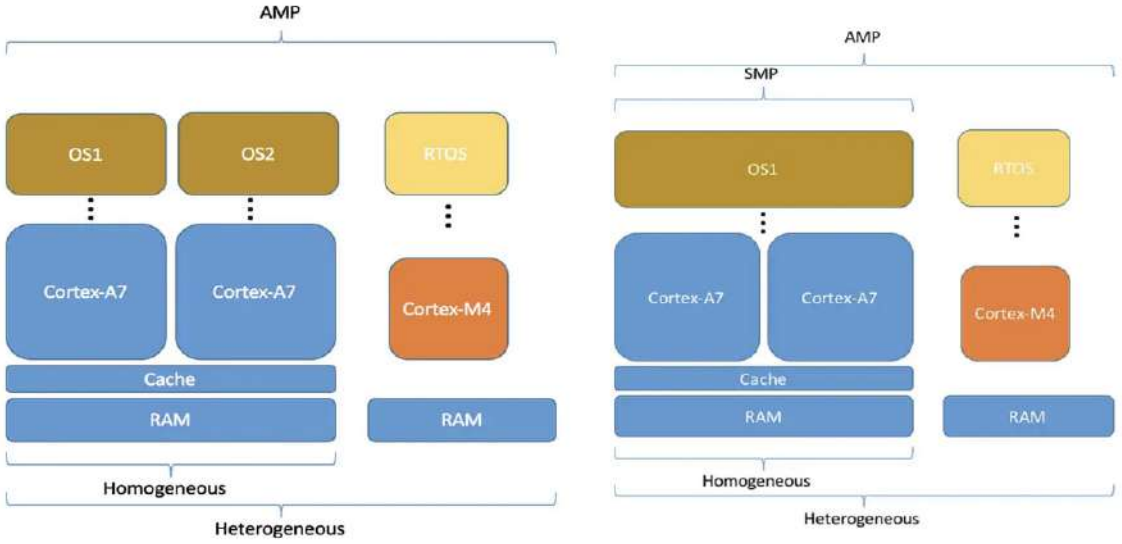
An Intel® Core™ M Processor SoC



Why?

- Most Efficient Processors are Heterogeneous .
- Power Efficient (Green Computing)

Heterogeneous ARM Processor



Super Computers are Heterogeneous!

https://www.top500.org/lists/top500/2020/11/

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646

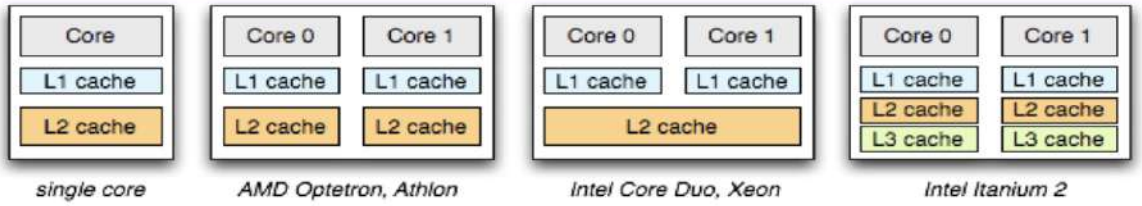
Role of User

- Programmers must use threads or processes.
- Spread the workload across multiple cores.
- Write parallel algorithms.
- OS will map threads/processes to cores

Role of Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores.
- Most major OS support multi-core today

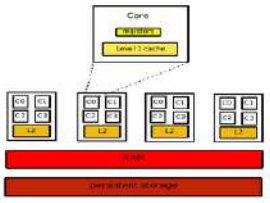
Role of Memory



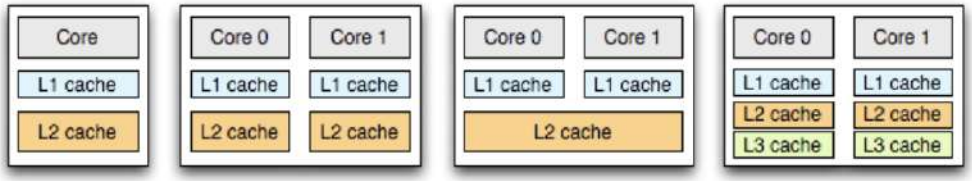
single core *AMD Optetron, Athlon* *Intel Core Duo, Xeon* *Intel Itanium 2*

Need to place Cache, on-Chip to Reduce bandwidth and increase Latency
Occupy Processor space and Power

Memory Contention: Communication and Computation uses same memory bandwidth.



Role of Memory



single core *AMD Optetron, Athlon* *Intel Core Duo, Xeon* *Intel Itanium 2*

Cache Coherence: A program running on multiple processors will normally have copies of the same data in several caches. The protocols to maintain coherence for multiple processors are called **cache coherence protocols. (Write Policies)**

False Sharing in the shared Cache: If two or more processors are writing data to different portions of the same cache line, then a lot of cache and bus traffic might result for effectively invalidating or updating every cached copy of the old line on other processors. This is called *"false sharing"* or also *"CPU cache line interference"*.

False Sharing

Problem: Updating the Array element by 2

5	4	6	3	12	15	7	9	11	1

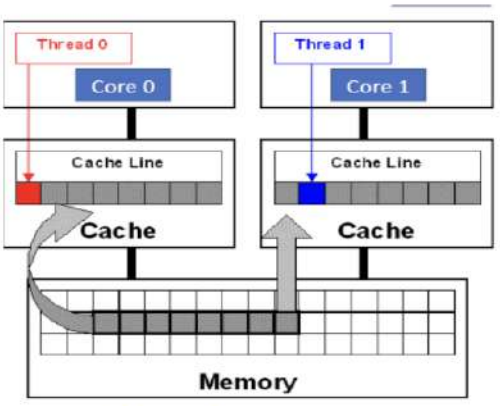
Parallel Solution: Split the array int0 two half to create 2 Threads, Thread 0 and Thread 1

However, two copy of entire array is copied into cache of each core.

False Sharing:

- Core0 update 1st half of the array.
- Core1 update 2nd half of the array.
- The cache line which contain the array is Invalidated, because they are in the same cache line, which lead to unnecessary cache update to maintain cache coherence.

False sharing will have serious effect on performance☹



Cache Coherence False Sharing Reference 3

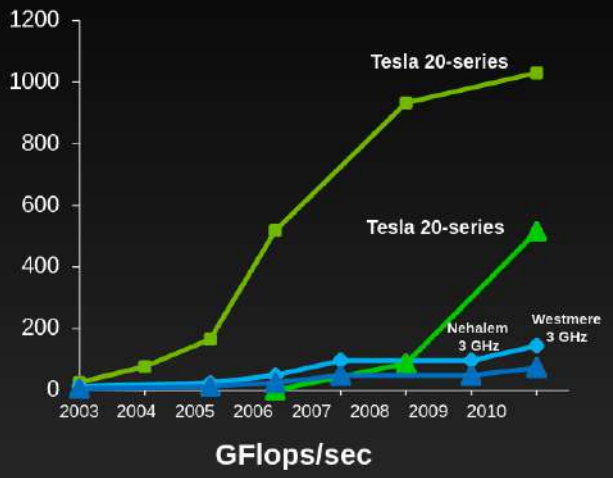
Limitations Of Multi Core Processors

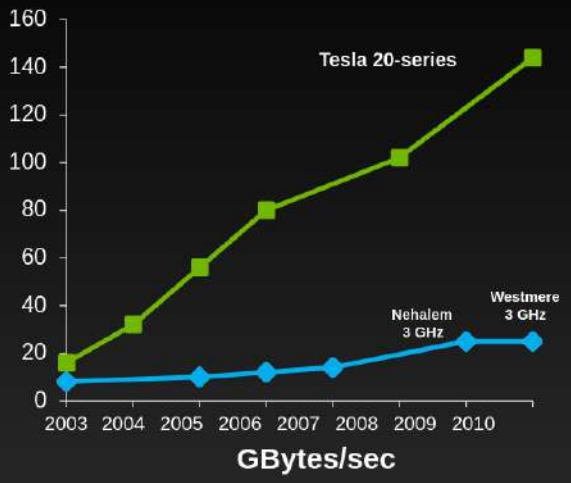
- Expensive than a solitary center processor.
- Speed isn't twice that of the typical processor.
- The presentation of the multicore processor relies on how the client utilizes the PC.
- They burn-through greater power.
- These processors become hot while accomplishing more work.
- Decomposition, Synchronization is done for all the tasks assigned and running on a multicore processor.

Note: In today's scenario, the compute power of Multicore processor is not sufficient.

So, **What Next ???**

Why GPU Computing?

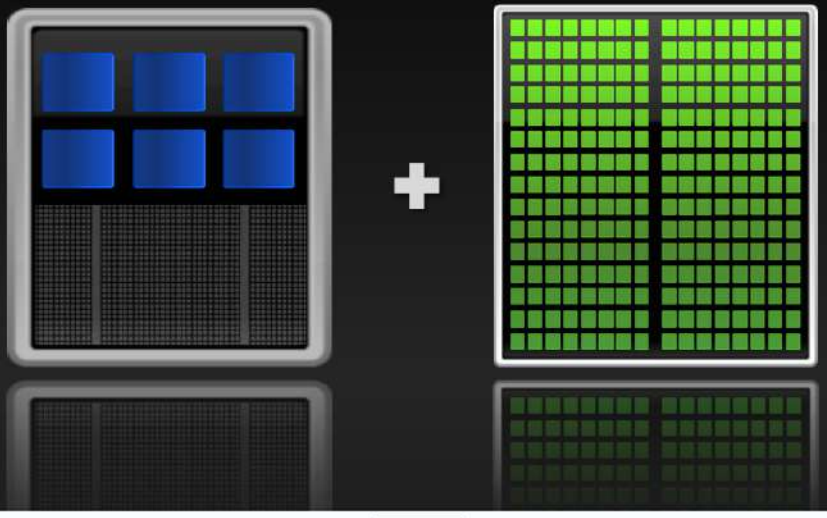




Single Precision: NVIDIA GPU
Double Precision: NVIDIA
Single Precision: x86 CPU
Double Precision: x86 CPU
NVIDIA GPU
X86 CPU
ECC off

Add GPUs: Accelerate Science Applications

CPU GPU



Speed v. Throughput

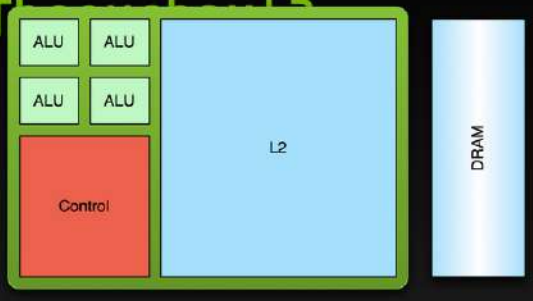
Speed

Throughput

Which is better depends on your needs…

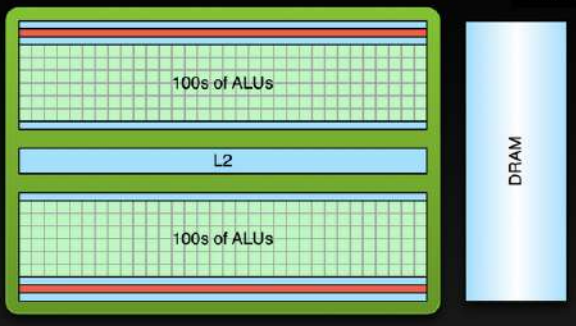
*Images from Wikimedia Commons via Creative Commons

Low Latency or High Throughput?



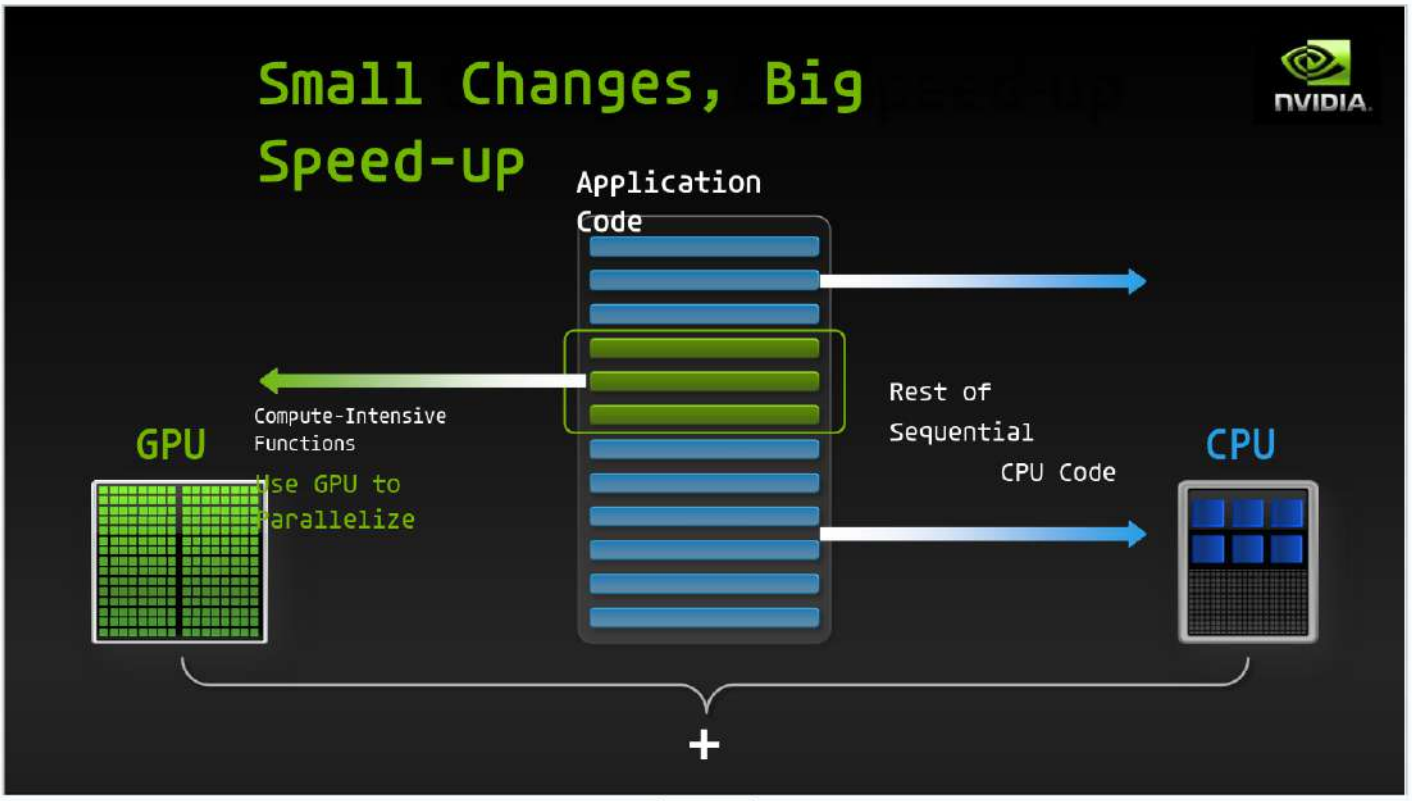
CPU

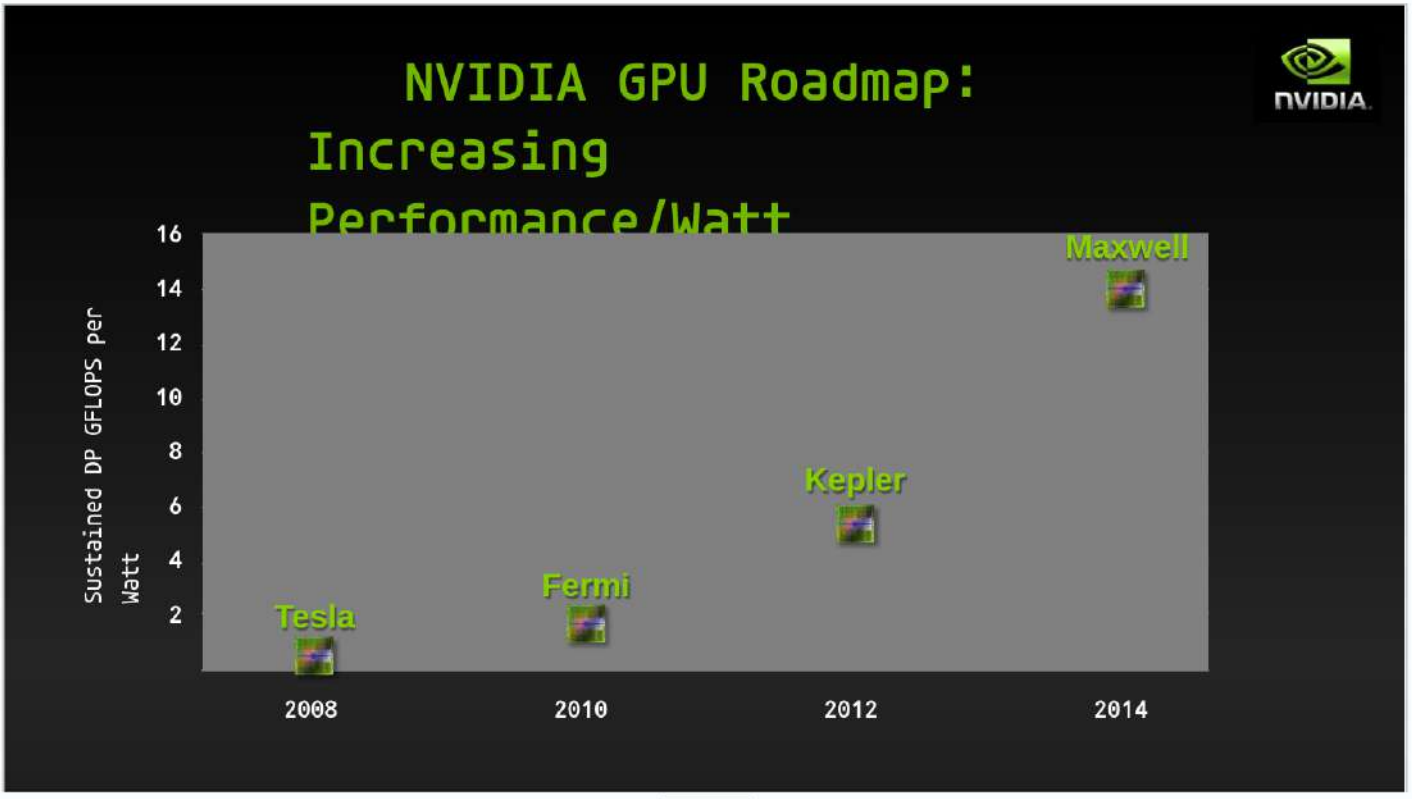
- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution



GPU

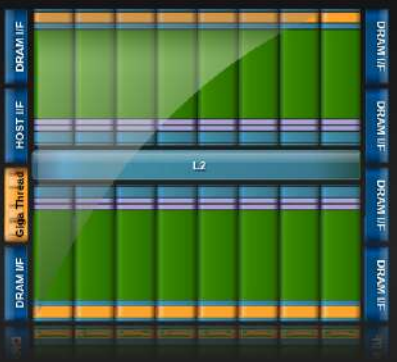
- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation


Small Changes, Big Speed-up
NVIDIA
Application Code
GPU
Compute-Intensive Functions
Use GPU to Parallelize
Rest of Sequential CPU Code
CPU
+


NVIDIA GPU Roadmap: Increasing Performance/Watt
NVIDIA
Sustained DP GFLOPS per Watt
16
14
12
10
8
6
4
2
Tesla
Fermi
Kepler
Maxwell
2008
2010
2012
2014

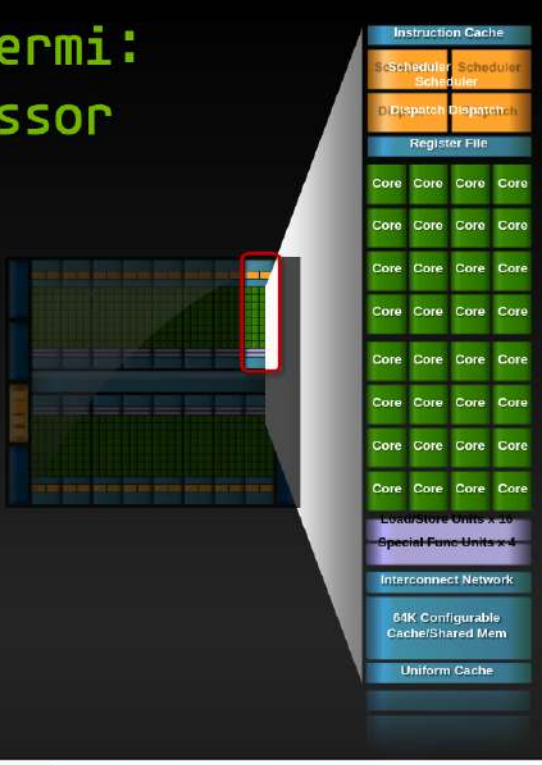
GPU Architecture: Two Main Components

- Global memory
  - Analogous to RAM in a CPU server Accessible by both GPU and CPU Currently up to 6 GB
  - Bandwidth currently up to 177 GB/s for Quadro and Tesla products
  - ECC on/off option for Quadro and Tesla products
- Streaming Multiprocessors (SMs) Perform the actual computations Each SM has its own:
  - Control units, registers, execution pipelines, caches

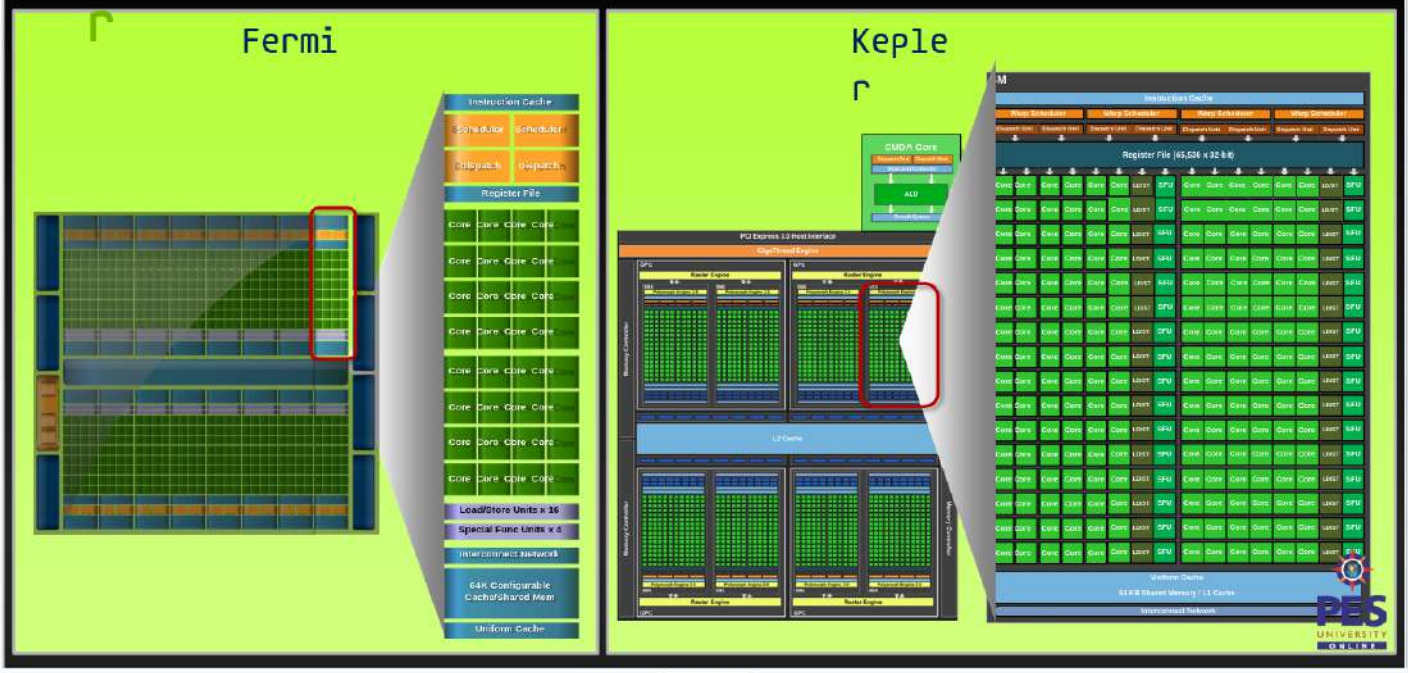


GPU Architecture – Fermi: Streaming Multiprocessor (SM)

- 32 CUDA Cores per SM
  - 32 fp32 ops/clock
  - 16 fp64 ops/clock
  - 32 int32 ops/clock
- 2 warp schedulers
  - Up to 1536 threads concurrently
- 4 special-function units
- 64KB shared mem + L1 cache
- 32K 32-bit registers



Kepler



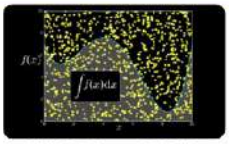
3 Ways to Accelerate Applications

Applications

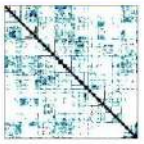
Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image

CULA tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and

NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

Thrust

C++ STL Features for CUDA

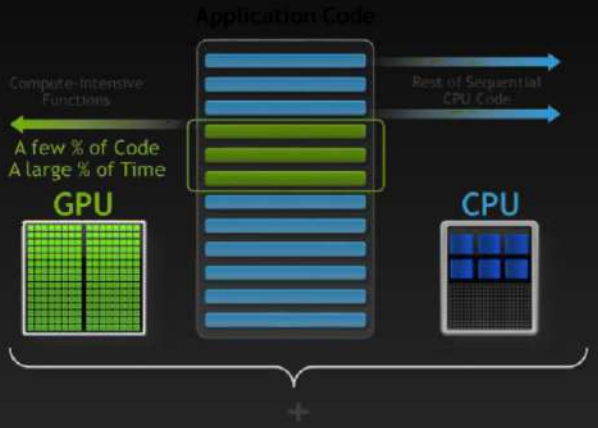
PES UNIVERSITY ONLINE

Parallelize

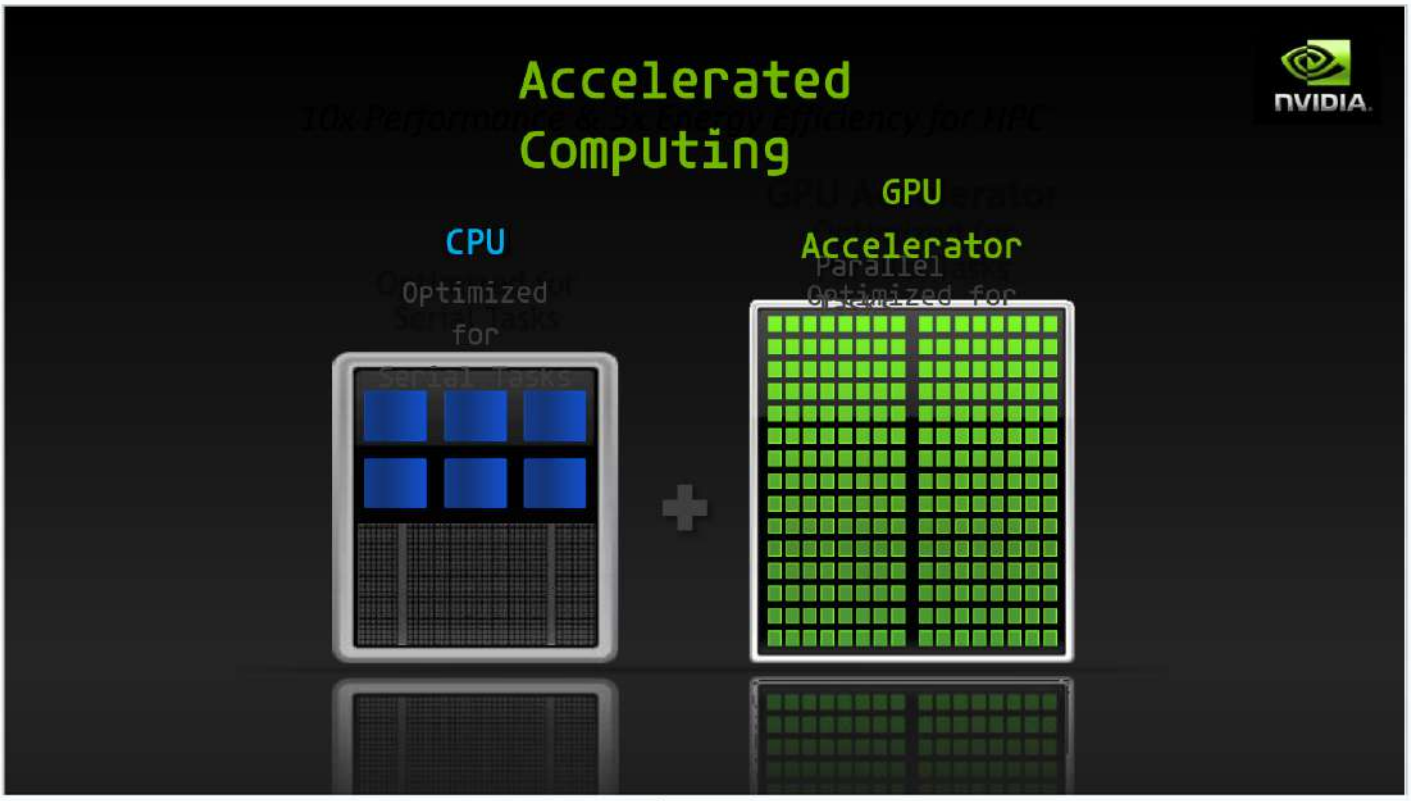
Insert OpenACC directives around important loops

Enable OpenACC in the compiler

Run on a parallel platform

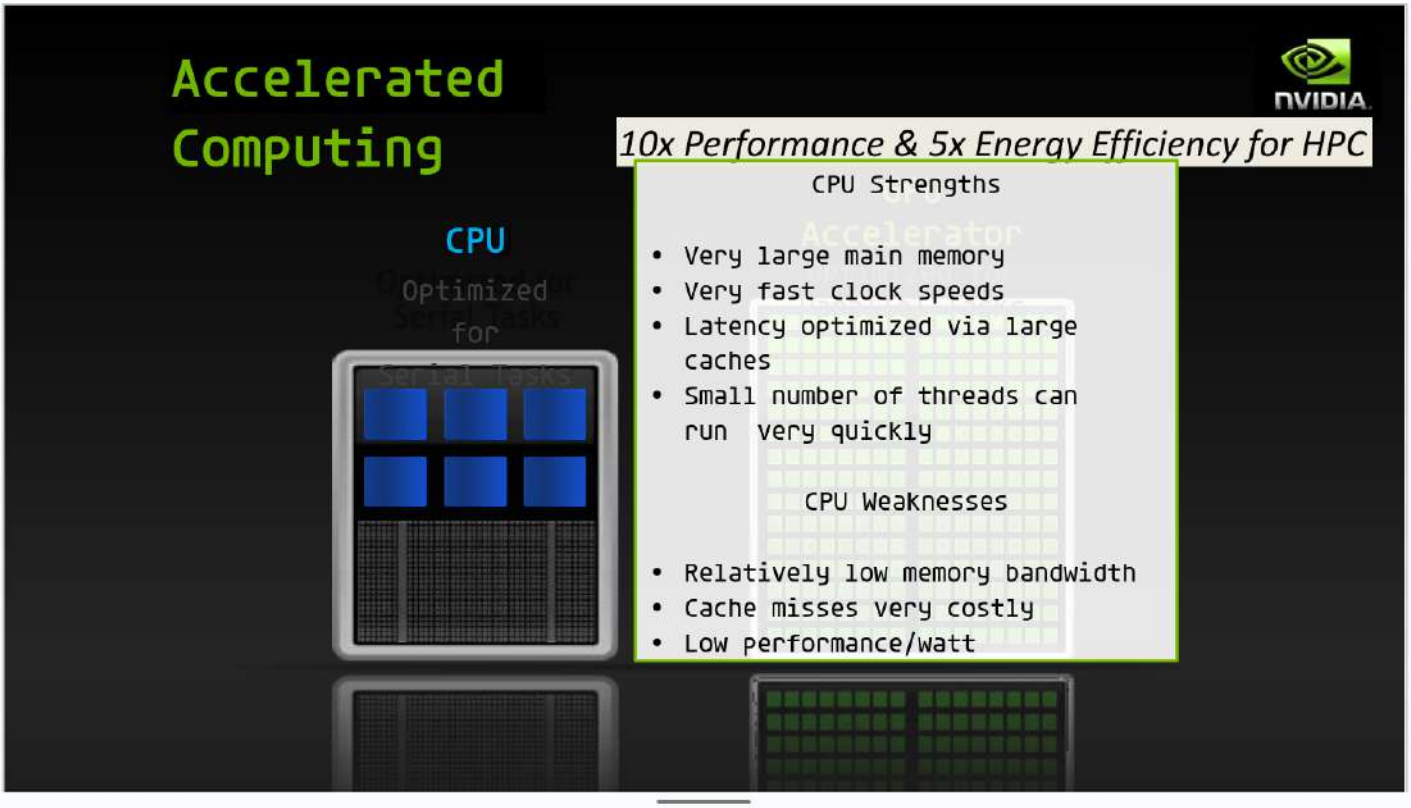


Accelerated Computing



Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC



CPU Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

CPU Weaknesses

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt

Accelerated Computing

10x Performance & 5x Energy Efficiency for HPC

GPU Strengths

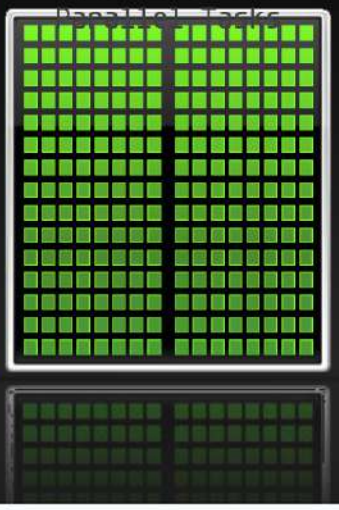
- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

GPU Weaknesses

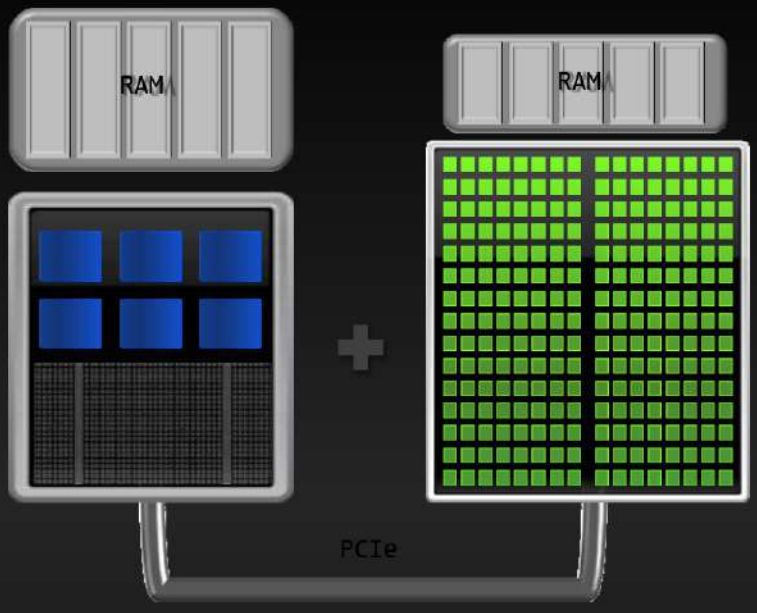
- Relatively low memory capacity
- Low per-thread performance

GPU Accelerator

Optimized for Parallel Tasks



Accelerator Nodes



CPU and GPU communicate via PCIe

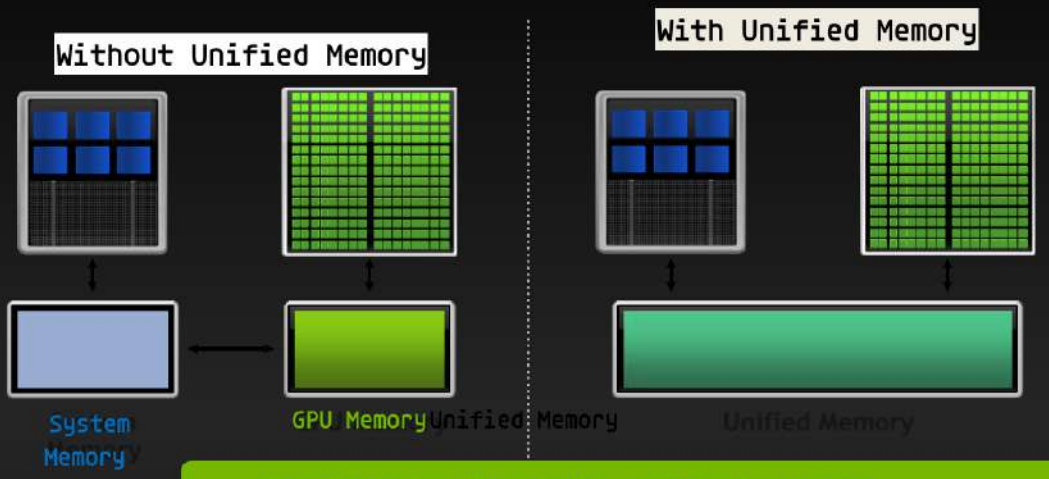
- Data must be copied between these memories over PCIe
- PCIe Bandwidth is much lower than either memories

Obtaining high performance on GPU nodes often requires reducing PCIe copies to a minimum

CUDA Unified Memory

Simplified Developer Effort

Sometimes referred to as "managed memory."



New "Pascal" GPUs handle Unified Memory in hardware.

OpenACC Parallel Directive

Generates parallelism

```
#pragma acc parallel
{
        When encountering the parallel directive, the
          compiler will generate 1 or more parallel
              gangs, which execute redundantly.
}
```