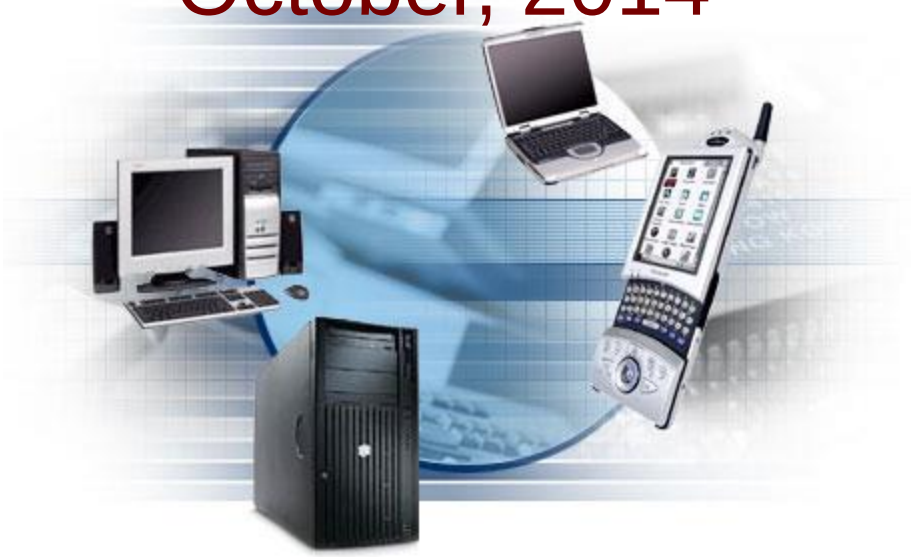
School of Parallel Programming & Parallel Architecture for HPC
ICTP
October, 2014



Intro to HPC Architecture

Instructor: Ekpe Okorafor

A little about me!

- PhD Computer Engineering – Texas A&M University
- Computer Science Faculty
  - Texas A&M University
  - University of Texas at Dallas
  - Addis Ababa University, Ethiopia
  - African University of Science & Technology, Abuja, Nigeria
- Big Data Academy, Accenture Digital (Analytics) – USA

Outline

- Overview
  - What is Parallel Computing?
  - Why Use Parallel Computing?
  - Who is Using Parallel Computing?
- Theoretical Background
- Concepts and Terminology
  - Von Neumann Computer Architecture
  - Flynn's Classical Taxonomy
  - Some General Parallel Terminology
  - Limits and Costs of Parallel Programming
- Parallel Computer Memory Architectures
  - Shared Memory
  - Distributed Memory
  - Hybrid Distributed-Shared Memory

Acknowledgements

- Blaise Barney
  - Lawrence Livermore National Laboratory, “Introduction to Parallel Computing”

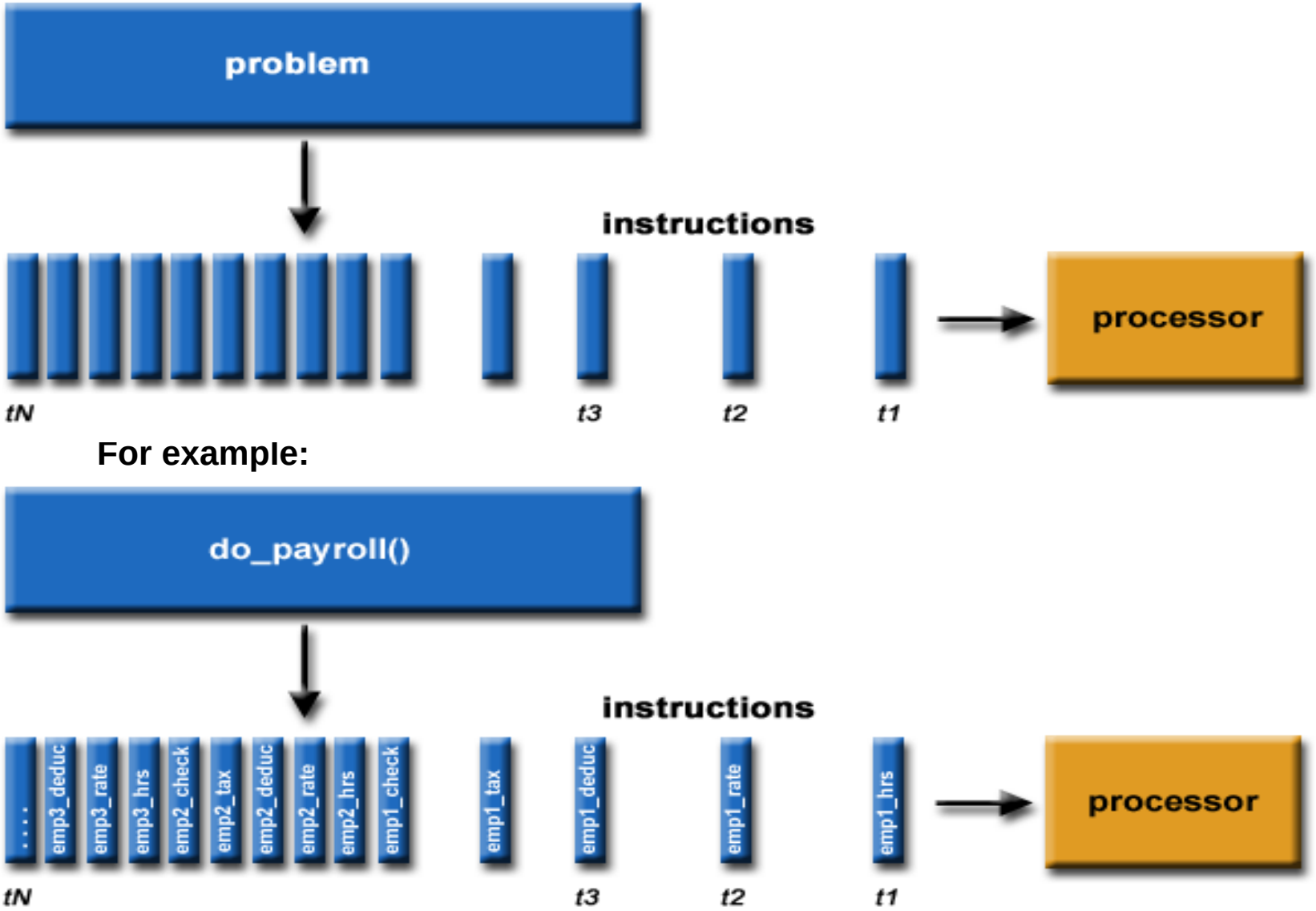
- Jim Demmel
  - Parallel Computing Lab at UC Berkeley, “Parallel Computing”

Overview

What is parallel computing?

- Traditionally, software has been written for serial computation:
  - A problem is broken into a discrete series of instructions
  - Instructions are executed sequentially one after another
  - Executed on a single processor
  - Only one instruction may execute at any moment in time

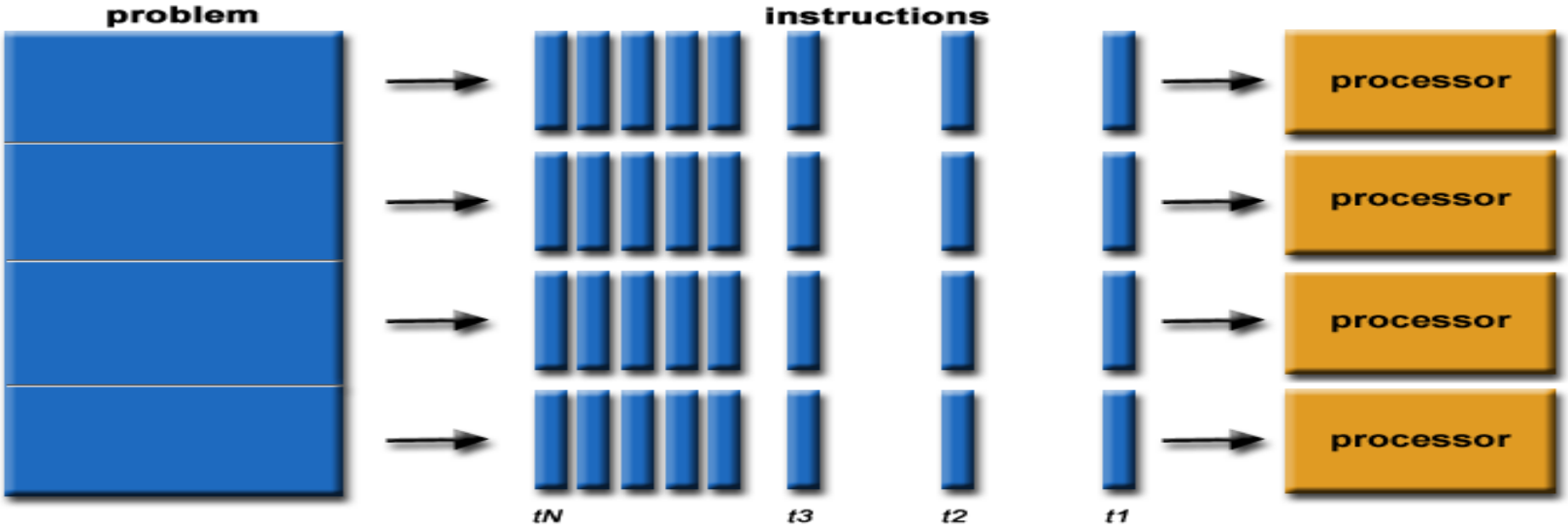
What is parallel computing?



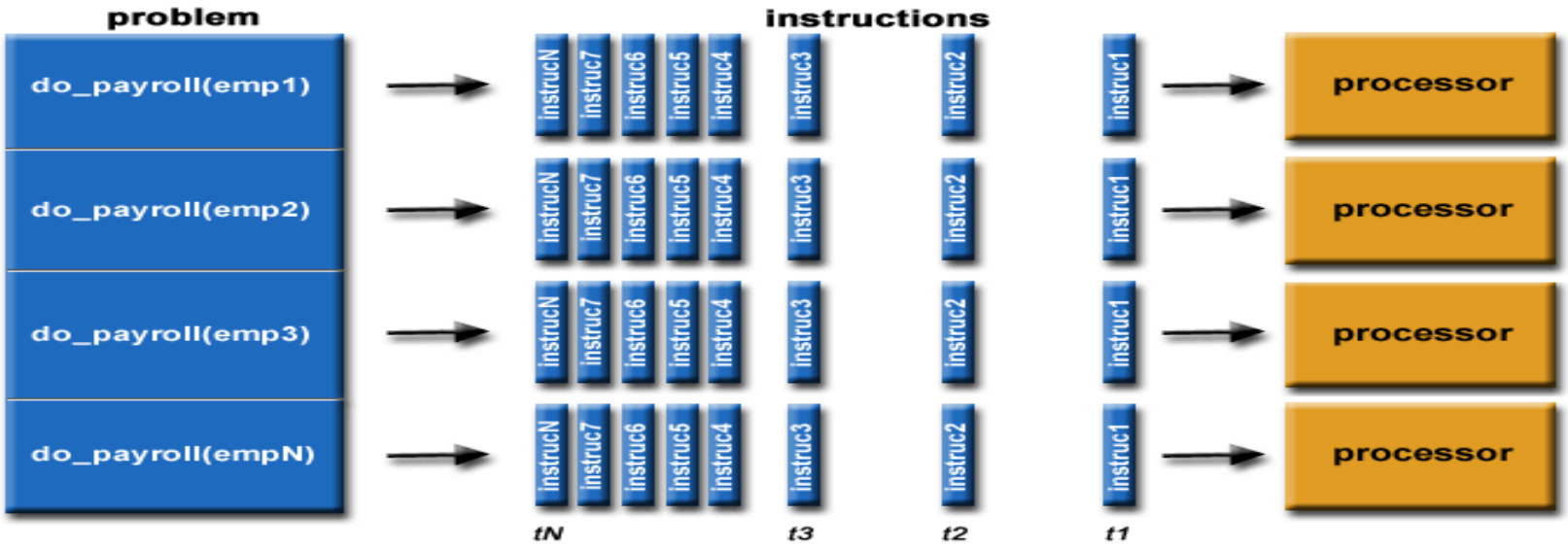
What is parallel computing?

- Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:
  - This is accomplished by breaking the problem into discrete parts so that each processing element can execute its part of the algorithm simultaneously with other processors

What is parallel computing?



For example



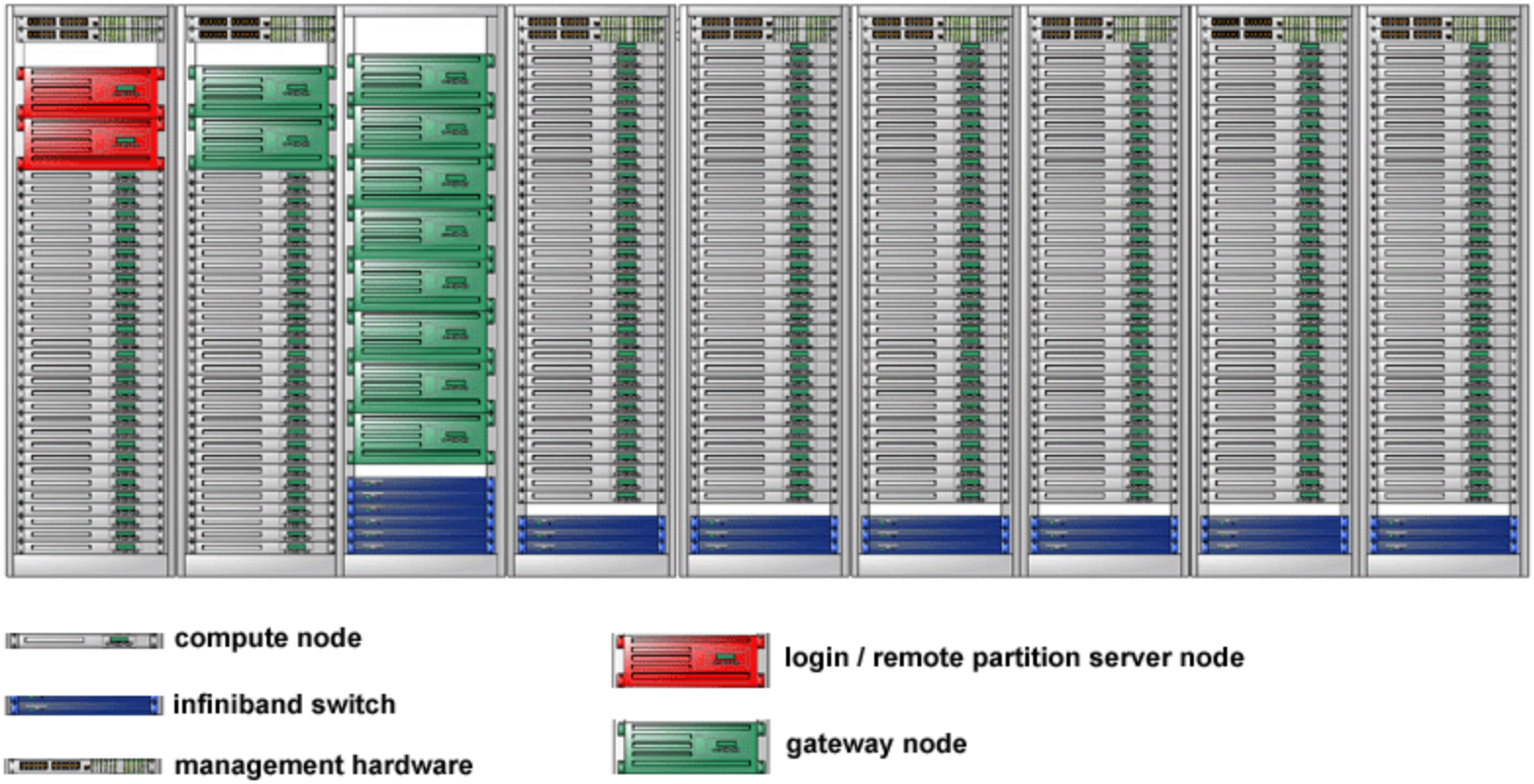
What is parallel computing?

- The computational problem should be able to:
  - 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
  - Several networked computers
  - A combination of both

What is parallel computing?

- LLNL parallel computer cluster:
  - Each compute node is a multi-processor parallel computer
  - Multiple compute nodes are networked together with an Infiniband network



Uses for parallel computing?

- 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

Uses for parallel computing?

Galaxy Formation

Planetary Movments

Climate Change

Rush Hour Traffic

Plate Tectonics

Weather

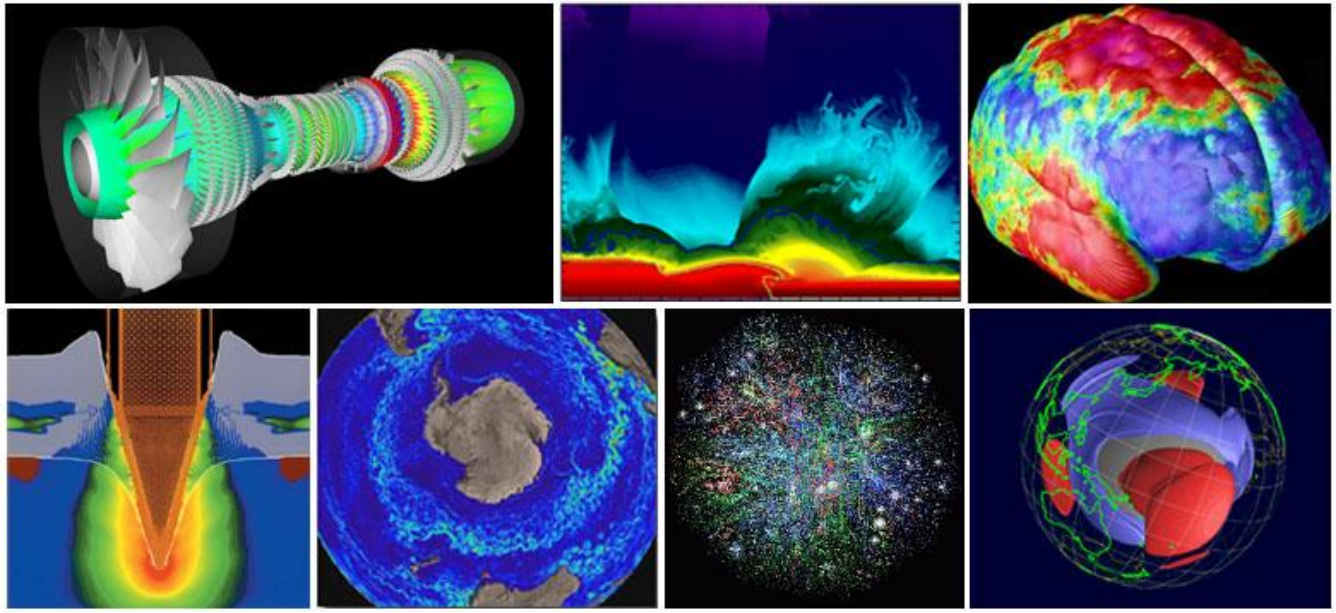
Auto Assembly

Jet Construction

Drive-thru Lunch

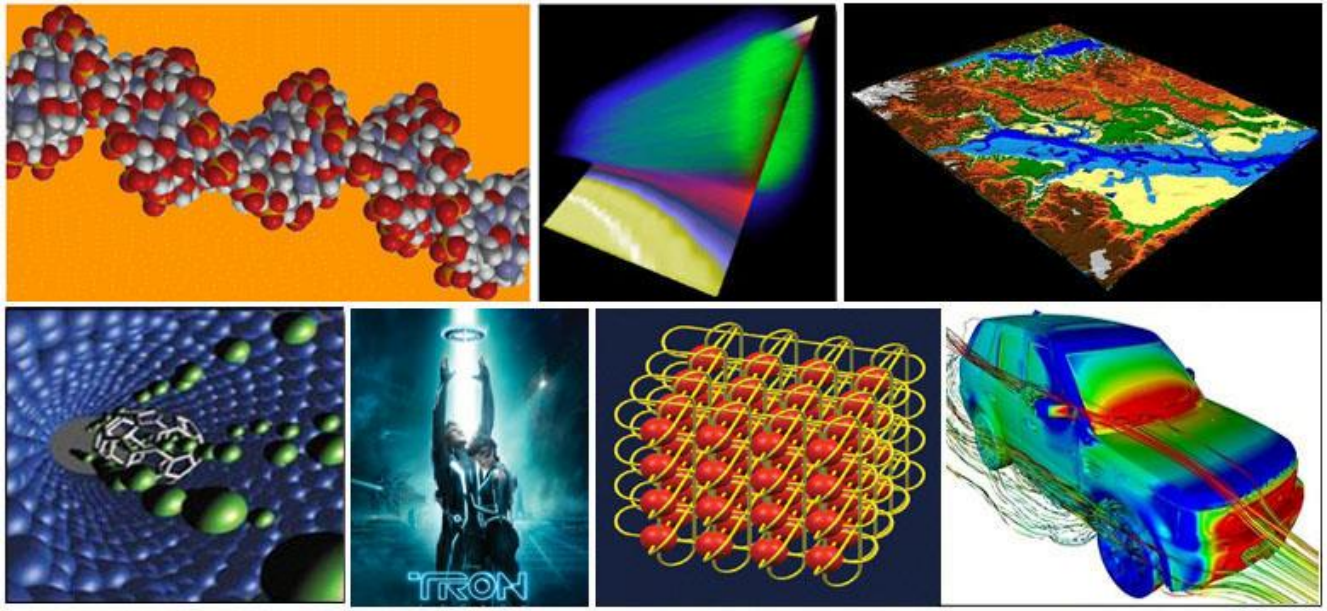
Uses for parallel computing

- **Science and Engineering**
  - Historically, parallel computing has been used to model difficult problems in many areas of science and engineering:
    - Atmosphere, Earth, Environment, Physics, Bioscience, Biotechnology, Genetics, Chemistry, Molecular Sciences, Geology, Seismology, Mechanical Engineering, Electrical Engineering, Circuit Design, Microelectronics, Computer Science, Mathematics, Defense, Weapons



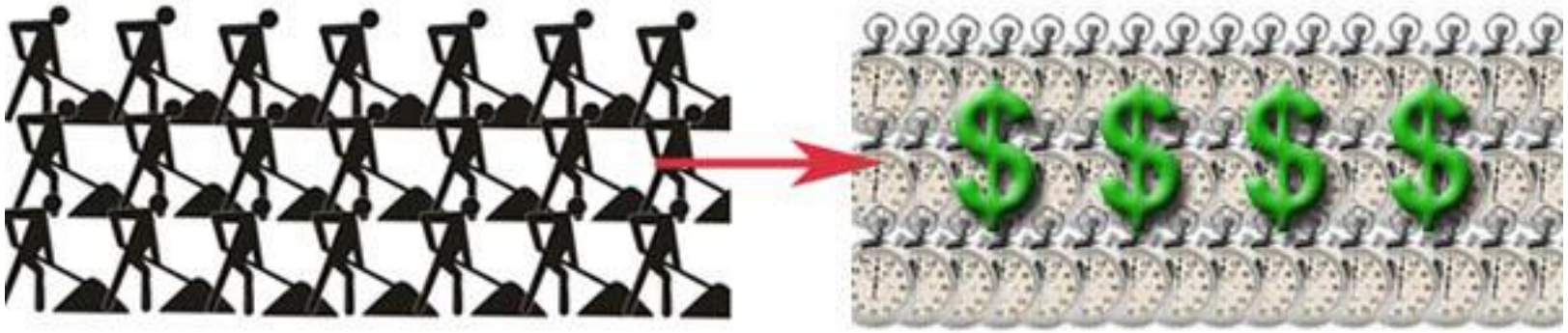
Uses for parallel computing

- **Industrial and Commercial**:
  - Today, commercial applications provide an equal or greater driving force in the development of faster computers.
    - Big Data, Analytics, Databases, Oil exploration, Web search engines, Medical imaging and diagnosis, Pharmaceutical design, Financial and economic modeling, Advanced graphics and virtual reality, Networked video and multi-media technologies, Collaborative work environments



Why use parallel computing?

- **Save time and/or money:**
  - In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings.
  - Parallel computers can be built from cheap, commodity components.



Why use parallel computing?

- **Solve larger / more complex problems:**
  - Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.
  - Example: "Grand Challenge Problems" (en.wikipedia.org/wiki/Grand_Challenge) requiring PetaFLOPS and PetaBytes of computing resources.
  - Example: Web search engines/databases processing millions of transactions per second

Why use parallel computing?

- **Provide concurrency:**
  - A single compute resource can only do one thing at a time. Multiple compute resources can do many things simultaneously.
  - Example: the Access Grid (www.accessgrid.org) provides a global collaboration network where people from around the world can meet and conduct work "virtually".

Why use parallel computing?

- **Use of non-local resources:**
  - Using compute resources on a wide area network, or even the Internet when local compute resources are scarce or insufficient.
  - Example: SETI@home (setiathome.berkeley.edu) over 1.3 million users, 3.4 million computers in nearly every country in the world. Source: www.boincsynergy.com/stats/ (June, 2013).
  - Example: Folding@home (folding.stanford.edu) uses over 320,000 computers globally (June, 2013)

Why use parallel computing?

- **Limits of serial computing:**
  - Transmission speeds
    - The speed of a serial computer is dependent upon how fast data can move through hardware
    - Absolute limits are the speed of light and the transmission of copper wire
    - Increasing speeds necessitate increasing proximity of processing elements
  - Limits of miniaturization
    - Processor technology is allowing an increasing number of transistors to be placed on a chip
    - However, even with molecular or atomic-level components, a limit will be reached on how small components can be

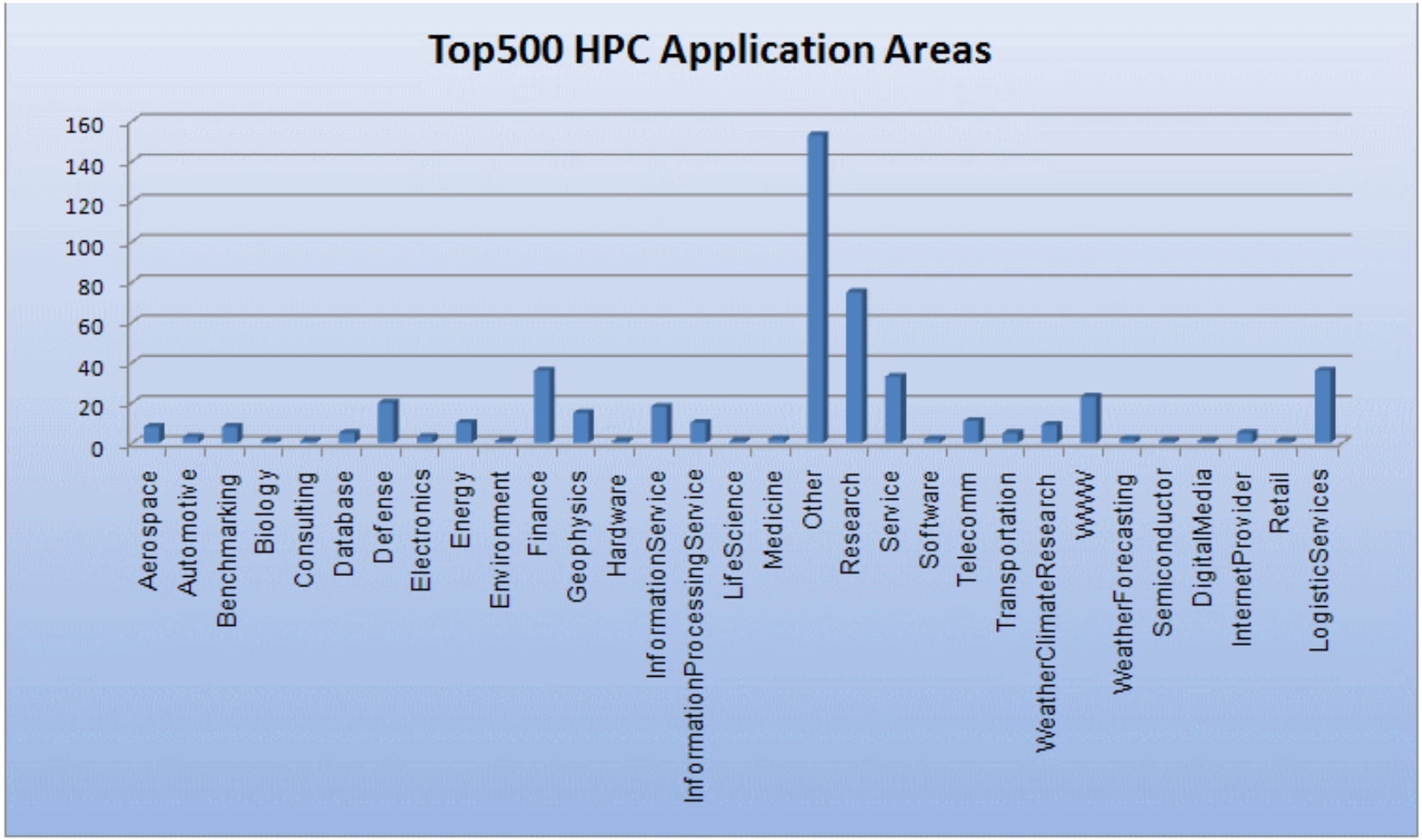
Why use parallel computing?

- **Limits of serial computing:**
  - Economic limitations
    - It is increasingly expensive to make a single processor faster
    - Using a larger number of moderately fast commodity processors to achieve the same or better performance is less expensive
  - Current computer architectures are increasingly relying upon hardware level parallelism to improve performance
    - Multiple execution units
    - Pipelined instructions
    - Multi-core

The future

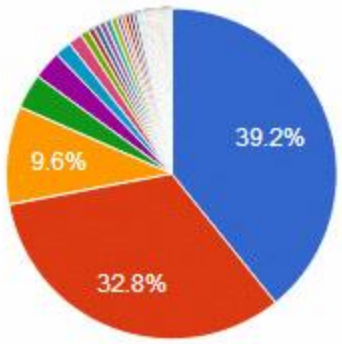
- Trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures clearly show that ***parallelism is the future of computing***.
- There has been a greater that 1000x increase in supercomputer performance, with no end currently in sight
- The race is already on for Exascale Computing!
  - 1 exaFlops = 10^{18}, floating point operations per second

Who is using parallel computing?



Who is using parallel computing?

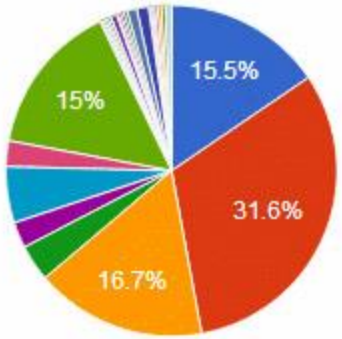
Vendors System Share



HP
IBM
Cray Inc.
SGI
Bull
Fujitsu
Dell

1/5

Vendors Performance Share



HP
IBM
Cray Inc.
SGI
Bull
Fujitsu
Dell

1/4

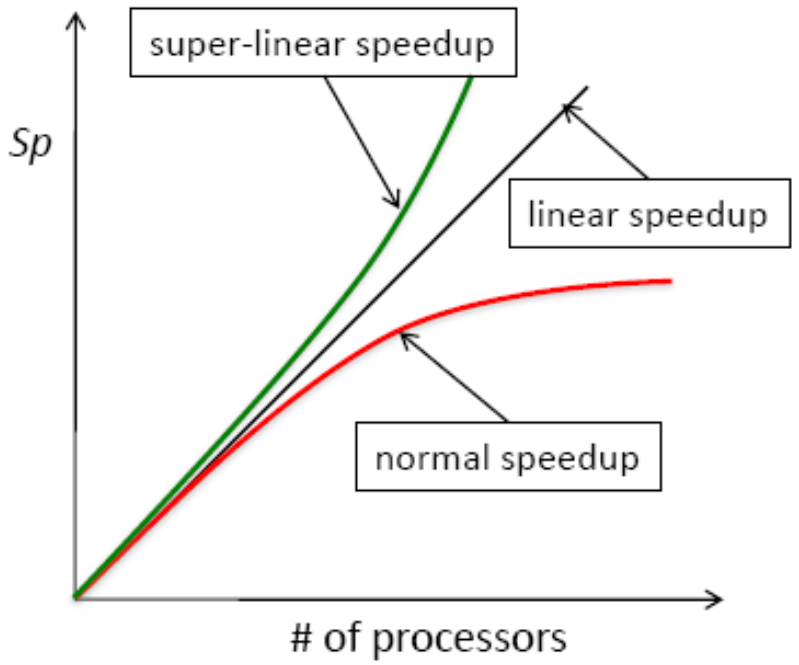
Vendors	Count	System Share (%)
HP	196	39.2
IBM	164	32.8
Cray Inc.	48	9.6
SGI	17	3.4
Bull	14	2.8
Fujitsu	8	1.6
Dell	7	1.4
NUDT	4	0.8
Supermicro	3	0.6
NEC	3	0.6
Megware	3	0.6
Hitachi	3	0.6
Oracle	3	0.6
Dawning	2	0.4
Itautec	2	0.4
RSC Group	2	0.4
Self-made	2	0.4
NRCPCET	2	0.4
Atipa	1	0.2
NEC/HP	1	0.2
Penguin Computing	1	0.2
Raytheon/Aspen Systems	1	0.2
Dell/Sun/IBM	1	0.2
Xenon Systems	1	0.2
Acer Group	1	0.2
HP/WIPRO	1	0.2
ClusterVision	1	0.2
Inspur	1	0.2
Clustervision/Supermicro	1	0.2
Netweb Technologies	1	0.2
IPE, Nvidia, Tyan	1	0.2
Adtech	1	0.2
Intel	1	0.2
T-Platforms	1	0.2
Hitachi/Fujitsu	1	0.2

Theoretical Background

Speedup & parallel efficiency

- Speedup: $S_p = \frac{T_s}{T_p}$
  - p = # of processors
  - Ts = execution time of the sequential algorithm
  - Tp = execution time of the parallel algorithm with p processors
  - $Sp = P$ (linear speedup: ideal)
- Parallel efficiency

$$E_p = \frac{S_p}{p} = \frac{T_s}{pT_p}$$



Limits of parallel computing

- Theoretical Upper Limits
  - Amdahl's Law
- Practical Limits
  - Load balancing
  - Non-computational sections
- Other Considerations
  - time to re-write code

Amdahl's law

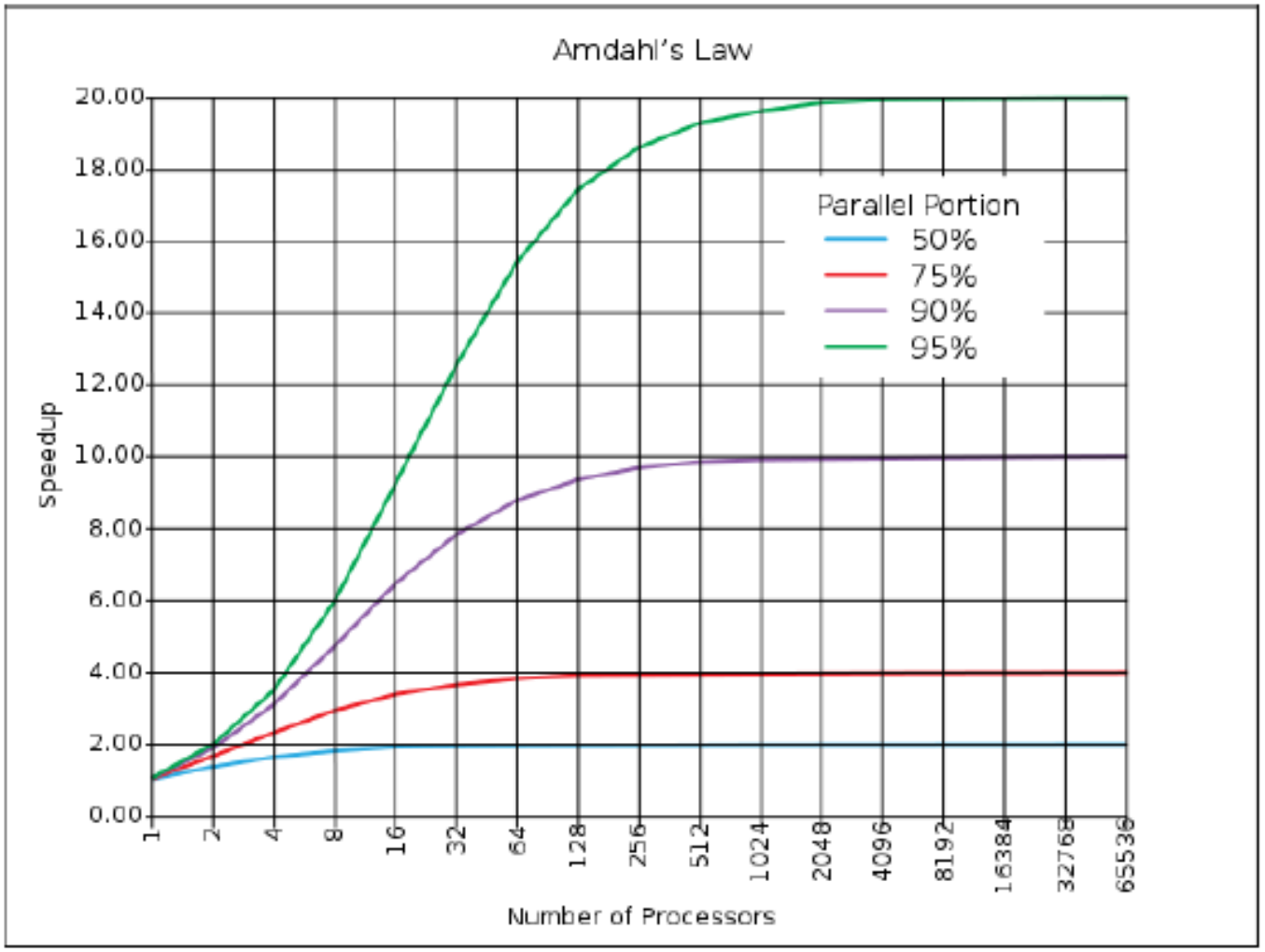
- All parallel programs contain:
  - parallel sections (we hope!)
  - serial sections (unfortunately)
- Serial sections limit the parallel effectiveness
- Amdahl's Law states this formally
  - Effect of multiple processors on speed up

$$S_P \le \frac{T_S}{T_P} = \frac{1}{f_s + \frac{f_p}{P}}$$

where

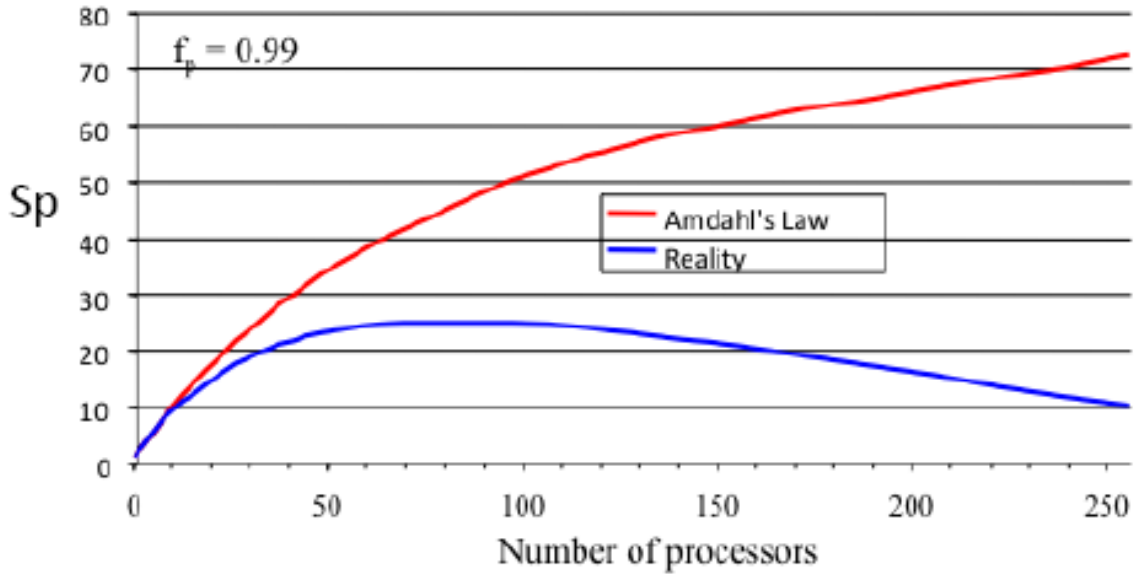
- f_s = serial fraction of code
- f_p = parallel fraction of code
- P = number of processors

Amdahl's Law


Amdahl's Law
Parallel Portion
50%
75%
90%
95%
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Practical limits: Amdahl's law vs. reality

- In reality, the situation is even worse than predicted by Amdahl's Law due to:
  - Load balancing (waiting)
  - Scheduling (shared processors or memory)
  - Cost of Communications
  - I/O



Gustafson's law

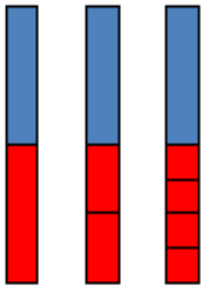
- Effect of multiple processors on run time of a problem with a *fixed amount of parallel work per processor.*

$$S(P) = P - \alpha \cdot (P - 1)$$

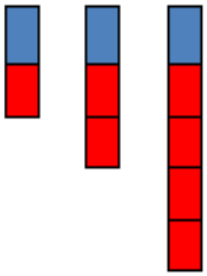
  - α is the fraction of non-parallelized code where the parallel work per processor is fixed (not the same as f_p from Amdahl's)
  - P is the number of processors

Comparison of Amdahl & Gustafson

$f_p = 0.5$



$\alpha = 0.5$



$$s \leq \frac{1}{f_s + f_p / N}$$

$$s(p) = p - \alpha(p-1)$$

$$s_2 \leq \frac{1}{0.5 + 0.5/2} = 1.3$$

$$s(2) = 2 - 0.5(2-1) = 1.5$$

$$s_4 \leq \frac{1}{0.5 + 0.5/4} = 1.6$$

$$s(4) = 4 - 0.5(4-1) = 2.5$$

Scaling: Strong Vs. Weak

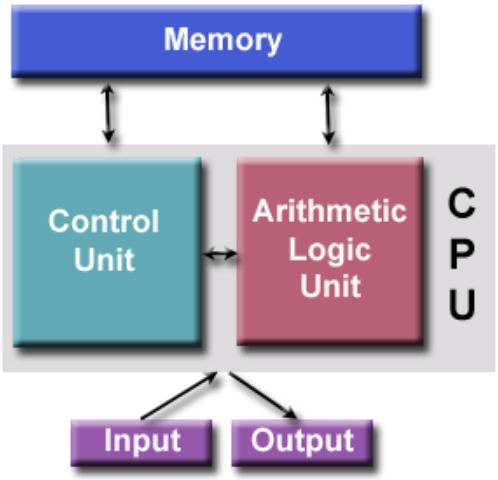
- We want to know how quickly we can complete analysis on a particular data set by increasing the PE count
  - Amdahl's Law
  - Known as "strong scaling"

- We want to know if we can analyze more data in approximately the same amount of time by increasing the PE count
  - Gustafson's Law
  - Known as "weak scaling"

Concepts & Terminology

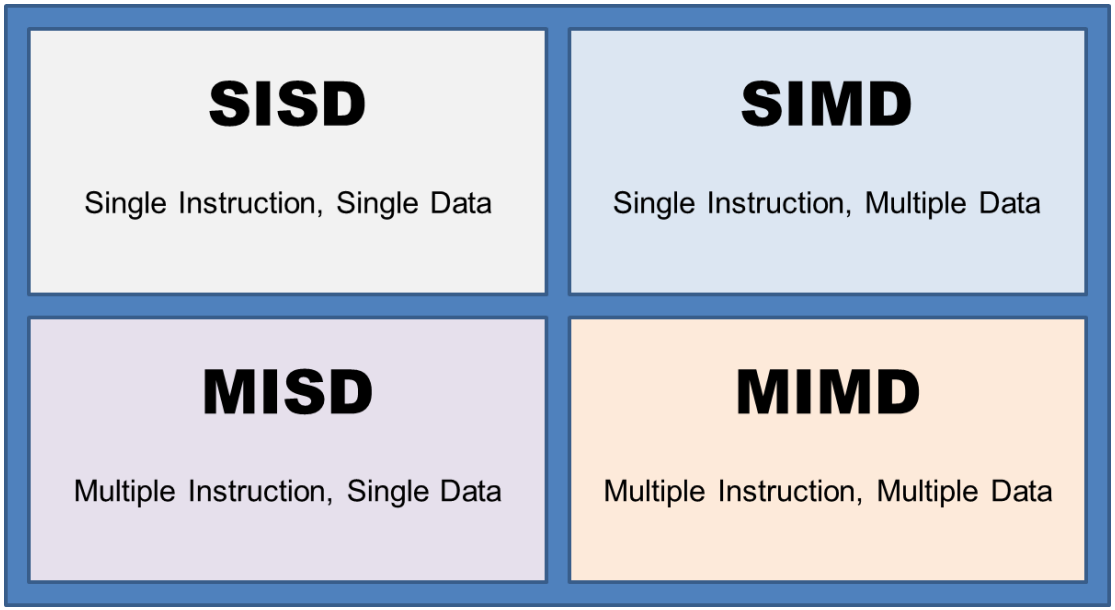
Von Neumann architecture

- Named after the Hungarian mathematician John von Neumann who first authored the general requirements for an electronic computer in his 1945 papers.
- Since then, virtually all computers have followed this basic design:
- Comprises of four main components:
  - Memory – used to store instructions and data
  - Control Unit – fetch, decode, coordinates
  - Arithmetic Logic Unit – basic arithmetic operations
  - Input/Output – interface to the human operator



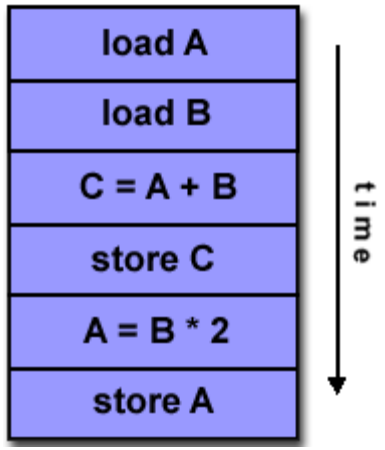
Flynn's classical taxonomy

- There are different ways to classify parallel computers
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
  - Based on the number of concurrent instructions and data streams available in the architecture



Flynn's classical taxonomy

- 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.



Flynn's classical taxonomy

- Single Instruction, Single Data (SISD):


UNIVAC1


IBM 360


CRAY1

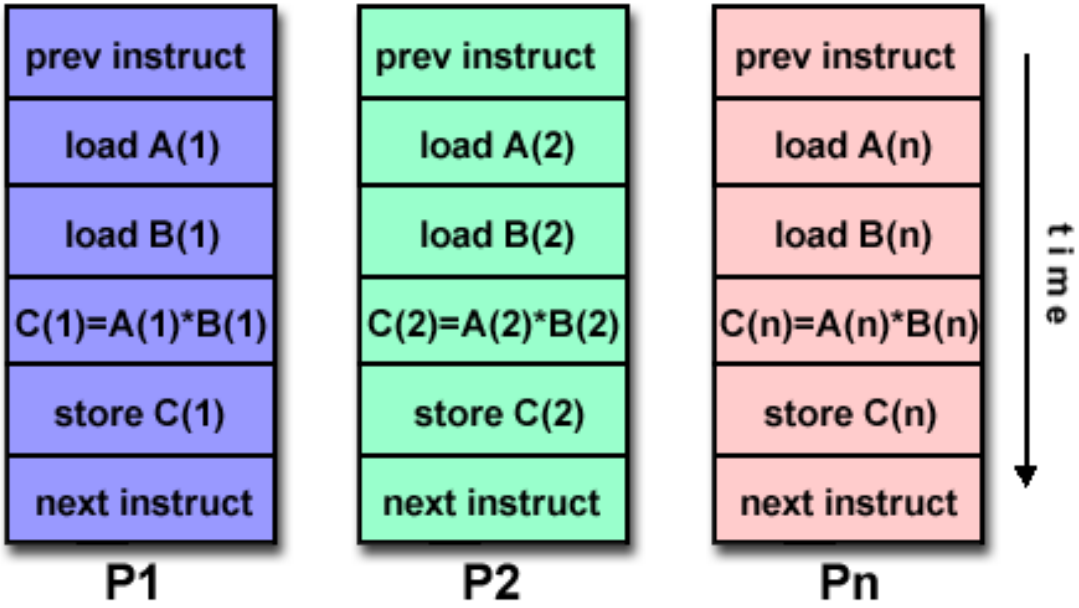

CDC 7600


PDP1


Dell Laptop

Flynn's classical taxonomy

- Single Instruction, Multiple Data (SIMD):
  - A type of parallel computer that exploits multiple data sets against a single instruction
    - **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

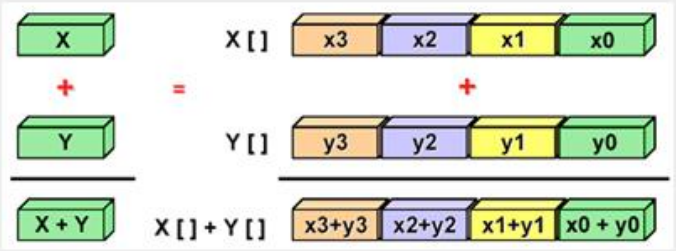


Flynn's classical taxonomy

- Single Instruction, Multiple Data (SIMD):
  - Two varieties: Processor Arrays and Vector Pipelines

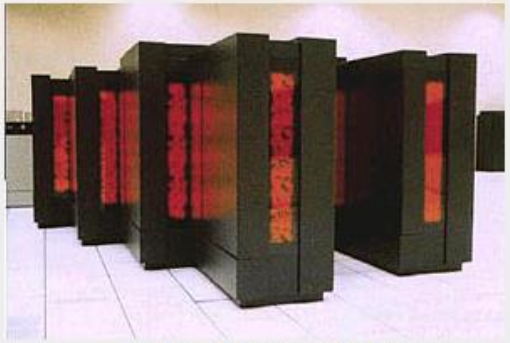
ILLIAC IV

MasPar

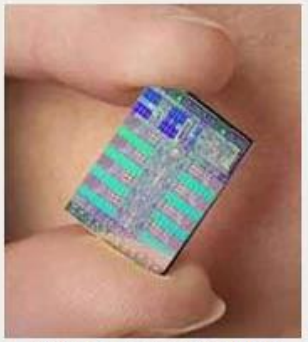


Cray X-MP

Cray Y-MP



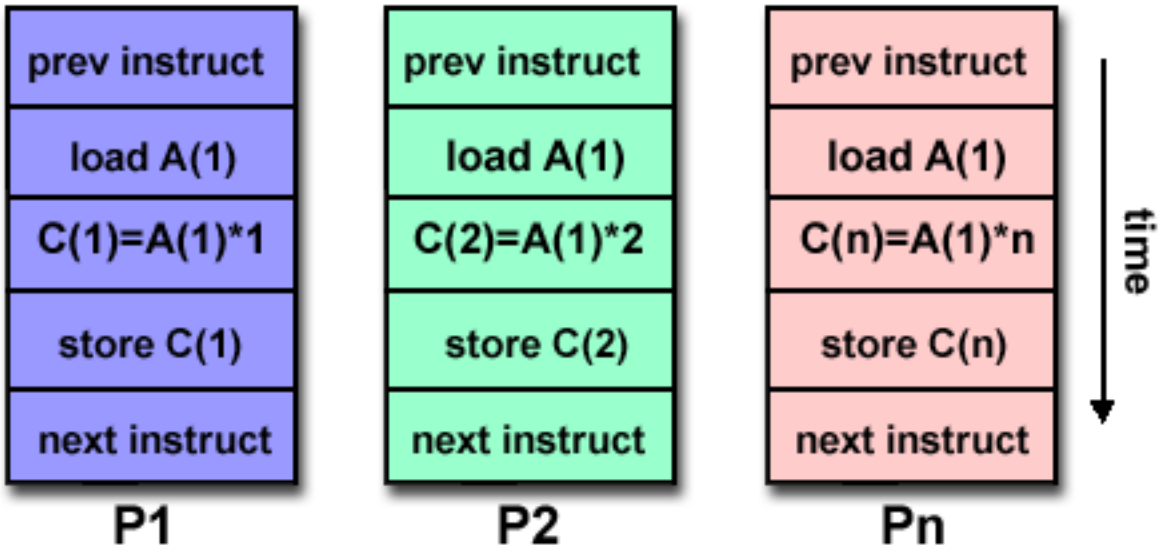
Thinking Machines CM-2



Cell Processor (GPU)

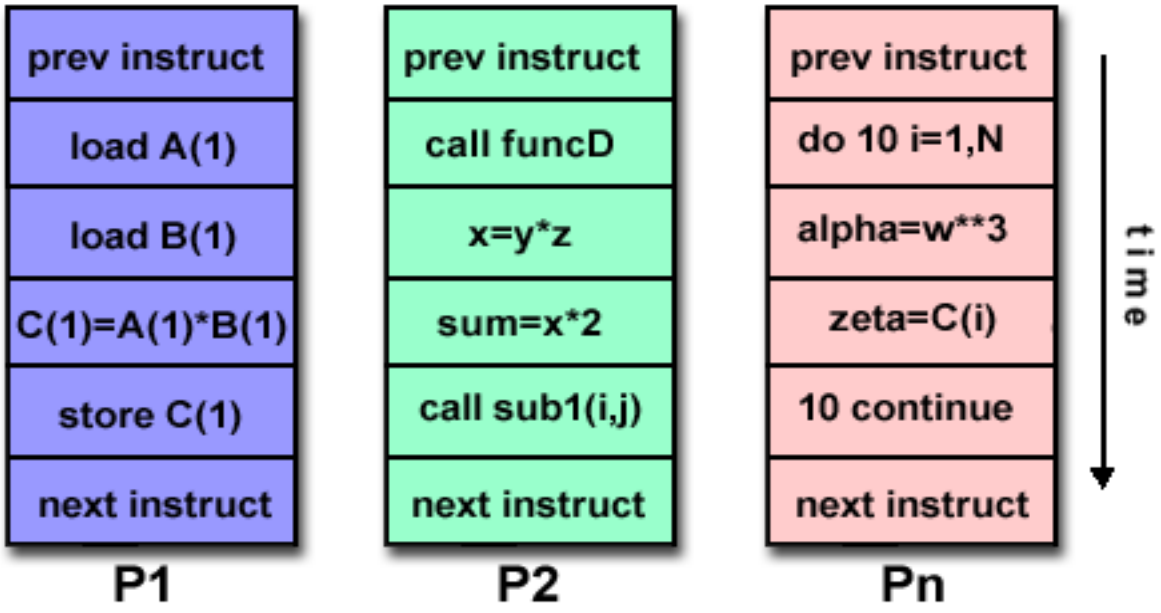
Flynn's classical taxonomy

- Multiple Instruction, Single Data (MISD):
  - **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.
  - Some conceivable uses might be:
    - multiple cryptography algorithms attempting to crack a single coded message.



Flynn's classical taxonomy

- Multiple Instruction, Mingle Data (MIMD):
  - Multiple autonomous processors simultaneously executing different instructions on different data
    - **Multiple Instruction:** Every processor may be executing a different instruction stream
    - **Multiple Data:** Every processor may be working with a different data stream



Flynn's classical taxonomy

- Most current supercomputers, networked parallel computer clusters, grids, clouds, multi-core PCs
- Many MIMD architectures also include SIMD execution sub-components


IBM POWER5


HP/Compaq Alphaserver


Intel IA32


AMD Opteron


Cray XT3


IBM BG/L

Some general parallel terminology

Like everything else, parallel computing has its own "jargon". Some of the more commonly used terms associated with parallel computing are listed below. Most of these will be discussed in more detail later.

- **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.
- **Parallel Task**
  - A task that can be executed by multiple processors safely (yields correct results)
- **Serial Execution**
  - Execution of a program sequentially, one statement at a time. In the simplest sense, this is what happens on a one processor machine. However, virtually all parallel tasks will have sections of a parallel program that must be executed serially.

Some general parallel terminology

- **Parallel Execution**
  - Execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time.
- **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.
- **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.

Some general parallel terminology

- **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.

Some general parallel terminology

- **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.

Some general parallel terminology

- **Parallel Overhead**
  - The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
    - Task start-up time
    - Synchronizations
    - Data communications
    - Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
    - Task termination time
- **Massively Parallel**
  - Refers to the hardware that comprises a given parallel system - having many processors. The meaning of many keeps increasing, but currently BG/L pushes this number to 6 digits.

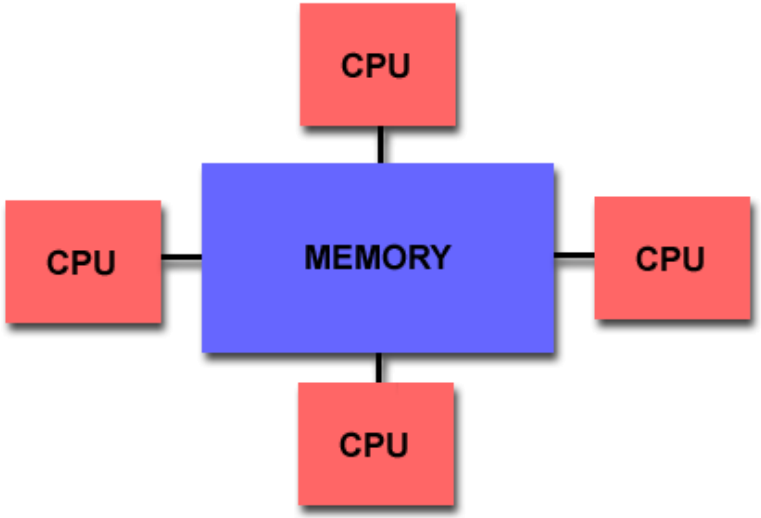
Some general parallel terminology

- **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 processors. Factors that contribute to scalability include:
    - Hardware - particularly memory-cpu bandwidths and network communications
    - Application algorithm
    - Parallel overhead related
    - Characteristics of your specific application and coding

Parallel Computer Memory Architectures

Shared memory

- All processors access all memory as global address space.
  - Multiple processors can operate independently but share the same memory resources.
  - Changes in a memory location effected by one processor are visible to all other processors.

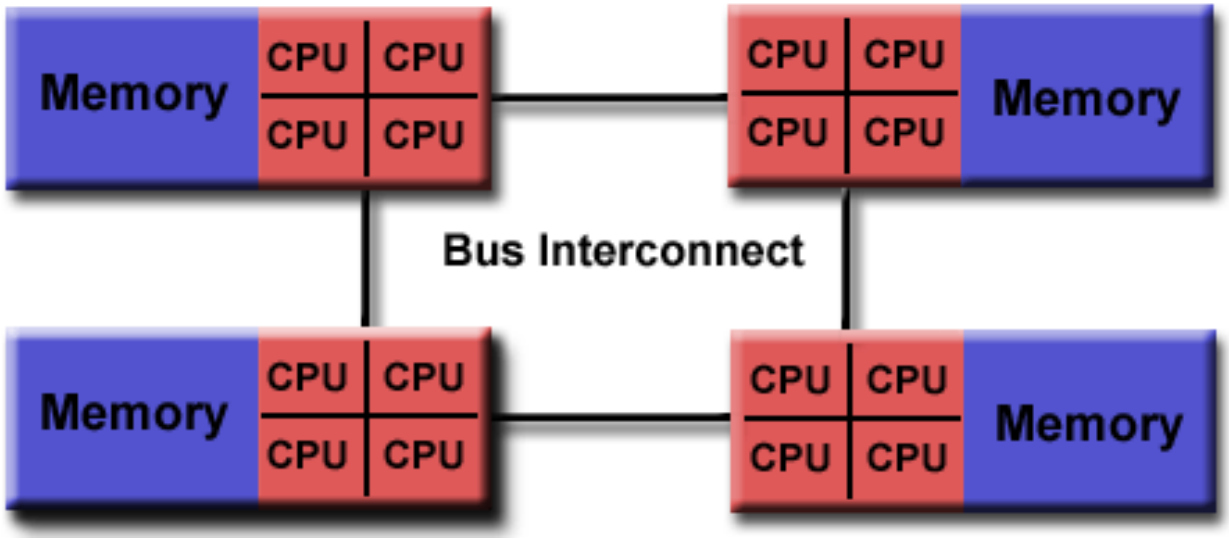


Shared memory:

- Shared memory machines can be divided into two main classes based upon memory access times: ***UMA*** and ***NUMA***.
  - 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.

Shared memory:

- 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



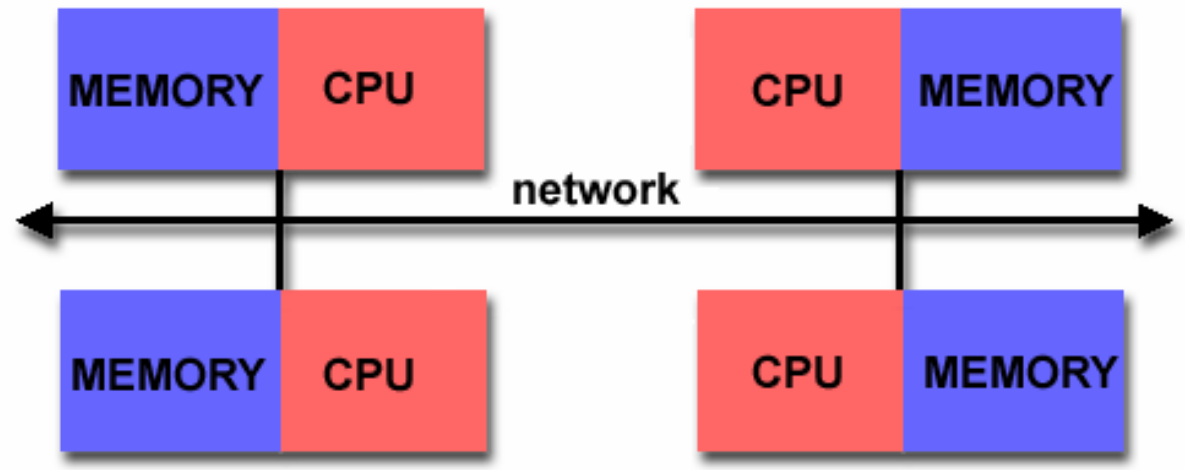
Shared memory: pro and con

- **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 insure "correct" access of global memory.

Distributed memory

- Each processor has a local memory
  - Changes to processor's local memory have no effect on the memory of other processors.
  - 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: pro and con

- **Advantages**
  - Memory is scalable with 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 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 (NUMA) times

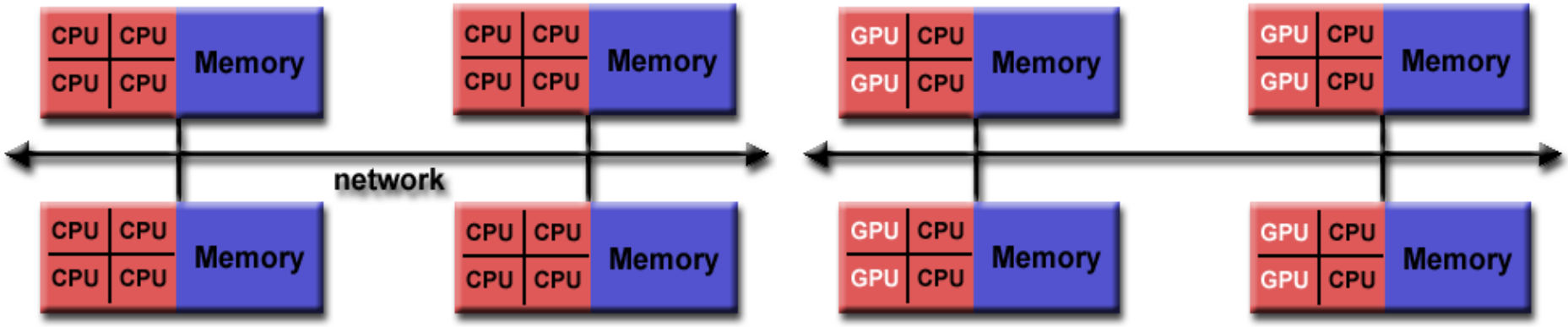
Hybrid distributed-shared memory

Summarizing a few of the key characteristics of shared and distributed memory machines

Comparison of Shared and Distributed Memory Architectures			
Architecture	CC-UMA	CC-NUMA	Distributed
Examples	SMPs Sun Vexx DEC/Compaq SGI Challenge IBM POWER3	Bull NovaScale SGI Origin Sequent HP Exemplar DEC/Compaq IBM POWER4 (MCM)	Cray T3E Maspar IBM SP2 IBM BlueGene
Communications	MPI Threads OpenMP shmem	MPI Threads OpenMP shmem	MPI
Scalability	to 10s of processors	to 100s of processors	to 1000s of processors
Draw Backs	Memory-CPU bandwidth	Memory-CPU bandwidth Non-uniform access times	System administration Programming is hard to develop and maintain
Software Availability	many 1000s ISVs	many 1000s ISVs	100s ISVs

Hybrid distributed-shared memory

- The largest and fastest computers in the world today employ both shared and distributed memory architectures.
  - The shared memory component is usually a cache coherent SMP machine or graphics processing units.
  - The distributed memory component is the networking of multiple SMPs or GPU machines.



Hybrid distributed-shared memory

- **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

Questions?