High Performance Computing for Science and Engineering

I

Lecturer: Petros Koumoutsakos

TA's: Christian Conti - Dmitry Alexeev - Panagiotis Chatzidoukas - Jana Lipkova - Jakob Progsch

1. September 25
   - Introduction
2. October 2
   - Performance + N-body Solvers
3. October 9th (CC)
   - N-Body + **CUDA**
4. October 16th (CC)
   - Linked Lists + CUDA
5. October 23rd
   - Diffusion with PDES + CUDA
6. October 30th
   - Roofline Models
7. November 6th
   - Stochastic Simulations + **OpenMP**
8. November 13th
   - Mesh<->Particle - OpenMP
9. November 20th (CC)
   - N**2 + Open MP
10. November 27th
    - N**2 + **MPI**
11. December 4th
    - PDE's + MPI
12. December 11th
    - PDEs (Linear Algebra) + MPI

December 18th - FINAL EXAM

GPUS

- Introduction to GPU
- Basics of **CUDA**
- Basic algorithms on GPU
- Optimization in **CUDA**

OpenMP - MPI

OpenMP environment

- Basic principles
- Work sharing
- Synchronization
- Threads
- Performance and optimization

MPI environment

- Point to point and collective communications
- Optimizations
- Topologies
- Memory to memory copies
- Parallel input-output

MPI and OpenMP - Deeper Cuts

ADVANCED MPI

- Derived types
- Point to point communication modes and comparison with collective communications
- Collecting computations via communications
- Problems of load balancing
- Placing of processes

ADVANCED Open/MP

- Fine synchronisation between 2 threads among « n » threads
- « Fine grain » approach: advantages and disadvantages for performance, extendability, ease of programming, etc...

HYBRID MPI/OpenMP

- Motivation
- Presentation of general concepts
- Hybrid programming model on SMP cluster
- Feedback, presentation of results
- Practical implementation
- Tools

4 "Dwarfs"

- N-Body Problem
- Stochastic Simulations
- Partial Differential Equations
- Mesh to Particle to Mesh Interpolations

Collaboration

NO PLAGIARISM (one incident = CLASS FAIL)

- Submitting as your own work the work or parts thereof of another person (whether it is from a book, the web, a program library or ANY other source) constitutes academic misconduct.

- If the source is clearly cited then it is OK, but then the homework credit will be adjusted to compensate for original work.

- You are encouraged to DISCUSS the homework problems with other colleagues

BOOKS

Introduction to High Performance Computing for Scientists and Engineers, Georg Hager and Gerhard Wellein, CRC Press, 2011

- **READING MATERIAL**
  - See Class Web Site
  - Introduction to Parallel Programming : http://www-users.cs.umn.edu/~karypis/parbook/
  - CUDA by Example, J. Sanders and E. Kandrot
  - Computer Organization and Design, D.H. Patterson, J.L. Hennessy

Links

http://www.cse-lab.ethz.ch/teaching.html

HPC tutorials at LLNL - Blaise Barney

https://computing.llnl.gov/?set=training&page=index

Hardware and Concepts

https://hpcforge.org/docman/view.php/67/134/EssentialHardware.pdf
https://hpcforge.org/docman/view.php/67/135/CoreConcepts.pdf

Parallel Program Design - Ian Foster

http://www.mcs.anl.gov/~itf/dbpp/

(see Class website for more)

http://www.cse-lab.ethz.ch/index.php/teaching/42-teaching/classes/577-hpcsei

D-DAY

- **GRADING** (subject to revision)

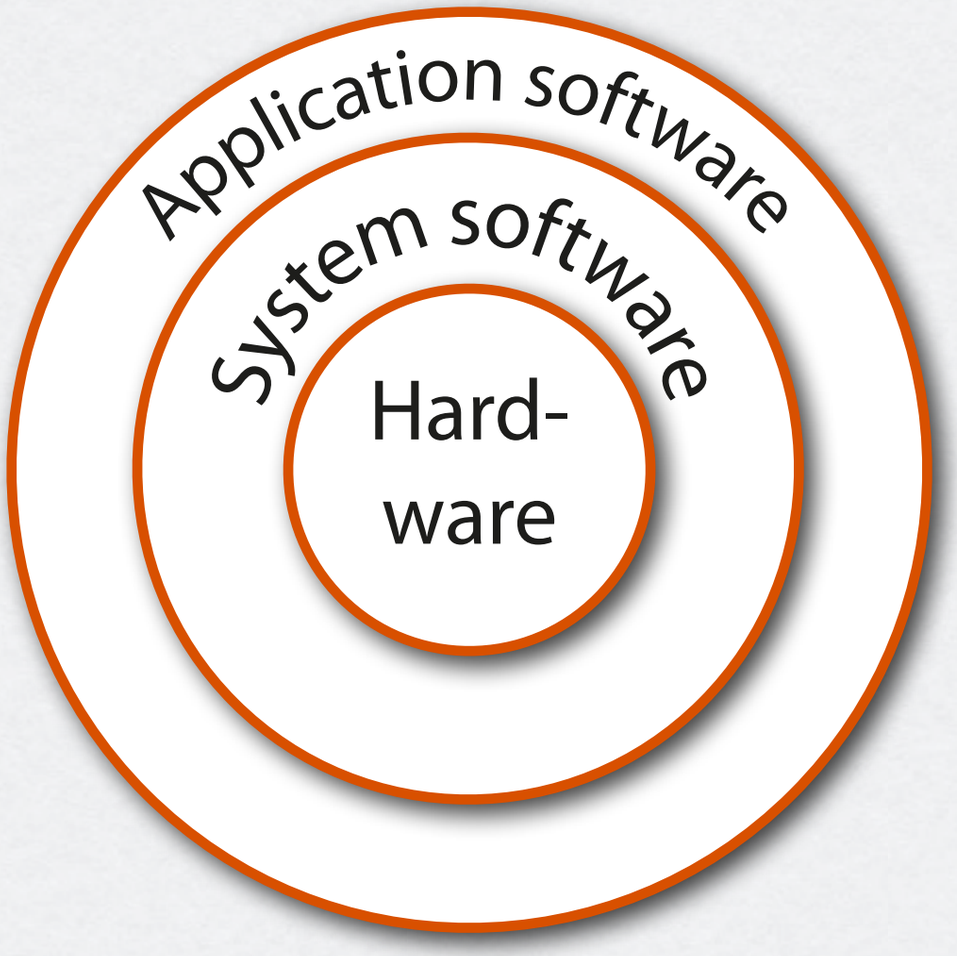
- FINAL EXAM

- YOU MUST DO THE 3 PROJECTS

Hardware and software

- The **hardware** can execute extremely simple, low level instructions.

Complex application —multiple levels→ simple instructions



System Software

- *Operating System:*
  - I/O operations
  - Storage and memory
  - Protection among concurrent applications
- *Compilers*
  - Translate from a high level language to hardware instructions

High level language to hardware

- Computer understands “on” and “off”
- The symbols for these states are 0 and 1 and we commonly use a language based on binary numbers.
  We use numbers for instructions and data.
- First programmers communicated with 0 and 1. Later programs were developed to translate from symbolic notation to binary. The first was called assembly.
  E. g. a programmer writes: `add A, B` ← Assembly language

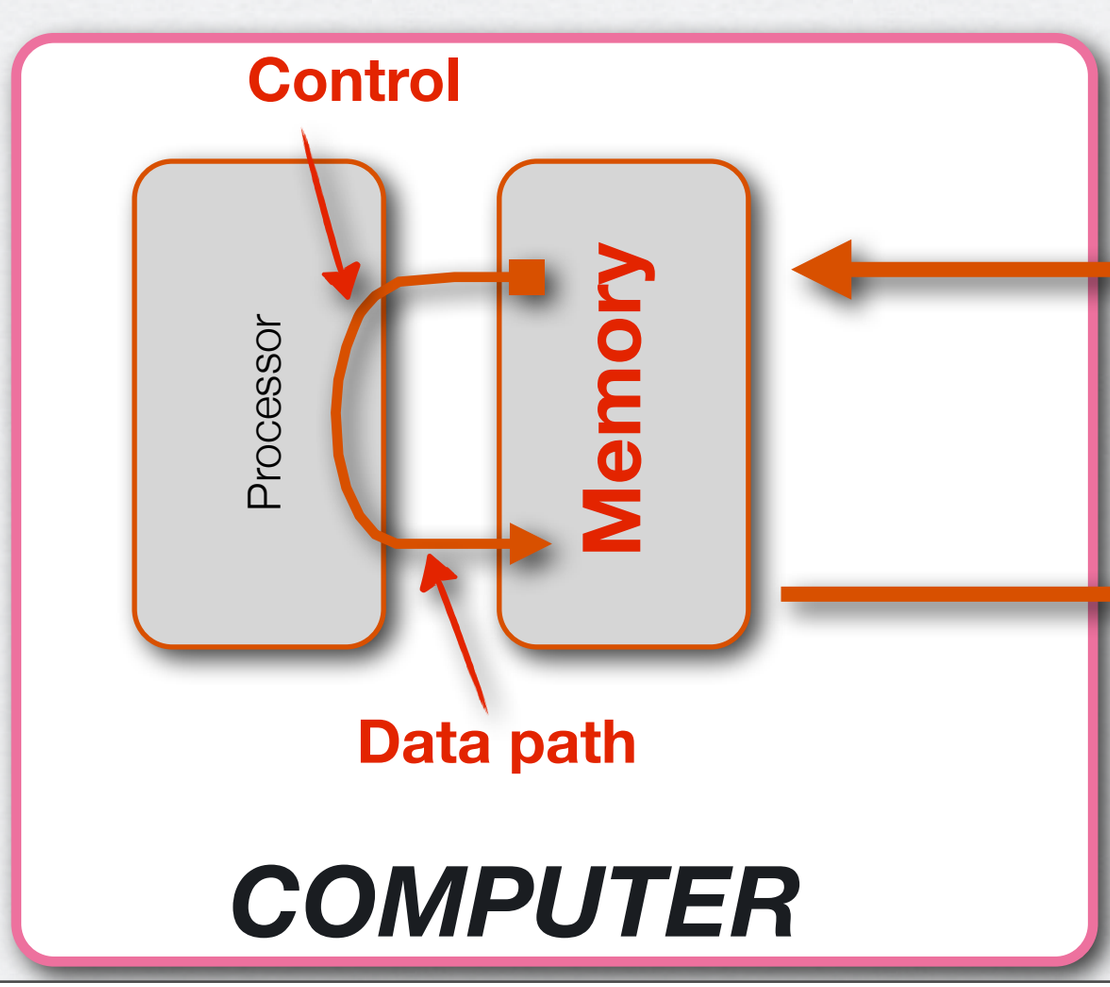
  assembly translates to: `1000110010100000` ← Machine language
- Advanced Languages are better than Assembly for:
  - programmer thinks in a more natural language
  - productivity of software development
  - portability

Computer: the BIG picture

SOFTWARE

Compiler

Performance evaluation



Input

Devices:

keyboard, mouse, etc.

Output

screen, printer, etc.

Computer: the BIG picture

Five classic components:

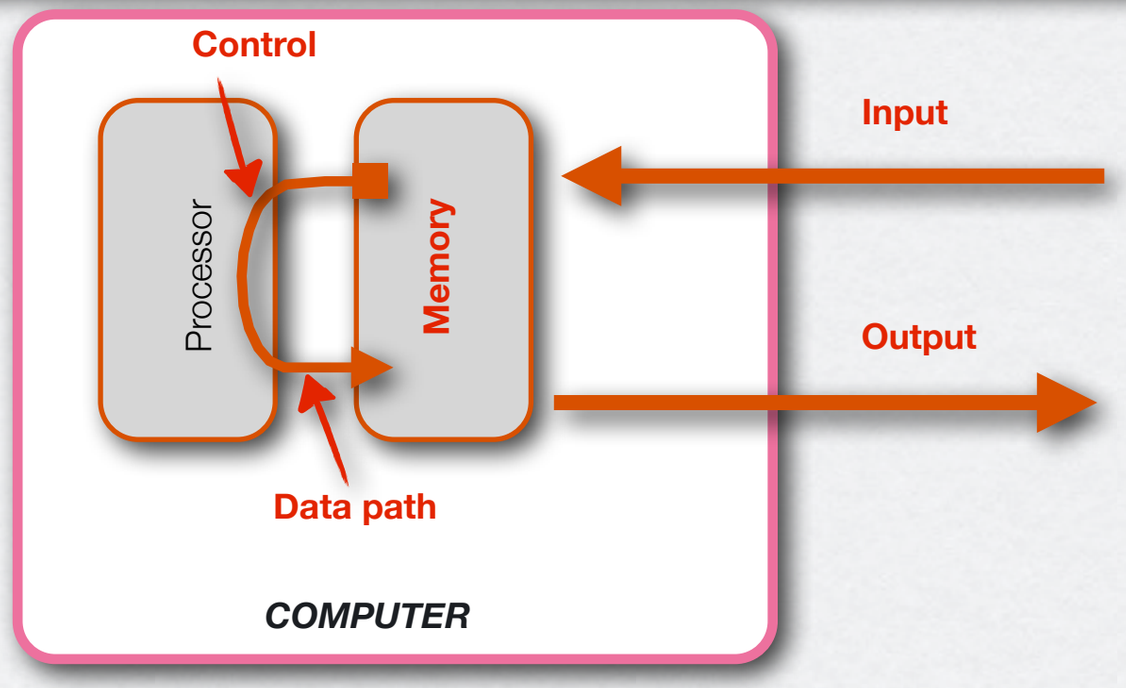
I.input

II.output

III.memory

IV.data path

V.control.



Memory: program and data storage → DRAM chips (some cost no matter what portion accessed)

Processor: (iv) data path + (v) control + cache memory

- (iv) data path: performs arithmetic operations
- (v) control: tells the data path memory and I/O what to do according to the program direction
- cache memory: buffer for DRAM memory

Communication between computers for: communicating information, resource sharing, nonlocal access

Performance

Response/Execution time: time between start and completion of a task

Throughput / Bandwidth: total amount of work done at a certain time

Performance and execution time:

$$Perf_x = \frac{1}{Exectime_x}$$

$$Perf_x > Perf_y \implies Exectime_y > Exectime_x$$

X is *n* times faster than Y to mean:

$$\frac{Perf_x}{Perf_y} = n = \frac{Exectime_y}{Exectime_x}$$

Time

Definitions of time

- **wall clock/response/elapsed:** total time including memory, CPU access, I/O, OS, everything
- **CPU execution time:** processor time spent on a task (system CPU/user CPU)
  - **user CPU time :** time spent in the program
  - **system CPU time :** time spent in the operating system performing tasks on behalf of the program.

CPU performance and its factors

$$\text{CPU exec. time} = \text{CPU}\ {}^{clock}_{cycles} \times \text{Clock cycle time}$$

$$= \frac{\text{CPU}\ {}^{clock}_{cycles}}{\text{Clock rate}}$$

Note: No reference to the number of instructions needed by the program

CPU performance and its factors

EXAMPLE

Program runs in 10s on a processor A which has a 2GHz clock rate. How to design a computer B to run this program in 6s?

Design consideration: Possible to increase in clock rate but then B will require 1.2 as many clock cycles as A for this program

$$\text{CPU}_A = \frac{\text{CPU clock cycles}_A}{\text{Clock rate}_\text{A}} = \frac{\text{CPU clock cycles}_A}{2\text{GHz}} = 10\text{s}$$

$$\implies \text{CPU clock cycles}_A = 2 * 10^{10}\text{cycles.}$$

$$\text{CPU}_B = \frac{1.2\text{CPU clock cycles}_A}{\text{Clock rate}_\text{B}} = 6\text{s} \qquad \implies \frac{1.2 * 2 * 10^{10}}{6} = \text{Clock rate}_\text{B} = 4\text{GHz}$$

CPU performance and its factors

Compiler generates instructions to execute

Computer executes these instructions

Execution time depends on number of instructions

Number of clock cycles to run a program:

$$\text{CPU clock cycles} = \text{Instructions for a program} \times \text{Average Clock Cycles Per Instruction (CPI)}$$

provides a way to compare different architectures

Note : Different arithmetic operations may have different numbers of instructions (e.g. x vs /)

CPU performance equation

$$\text{CPU time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock cycle time}$$

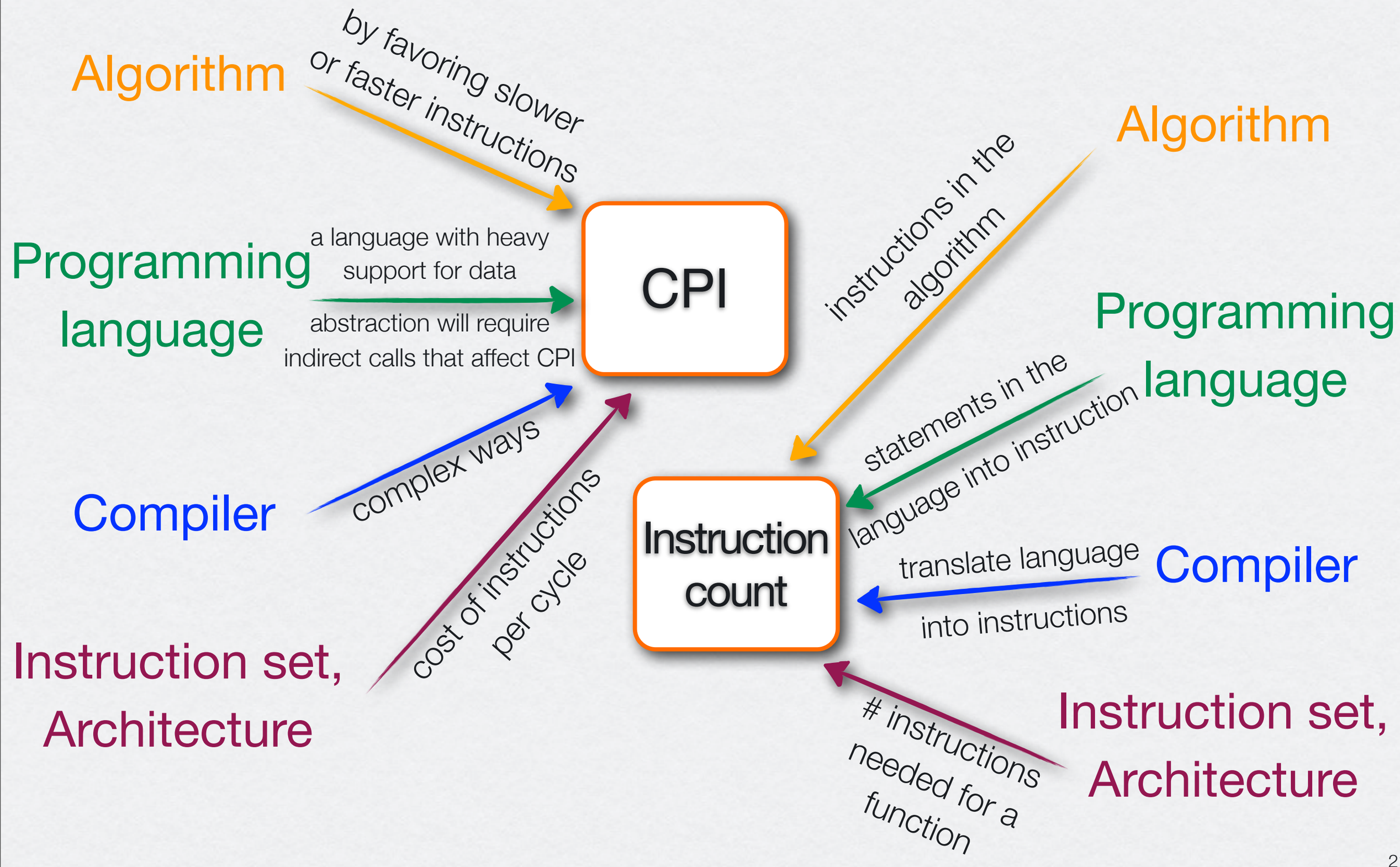
$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock rate}}$$

Includes the 3 factors that affect performance.

Performance is measured in time:

$$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycles}}$$

Program performance



The power wall

- Dominant technology for integrated circuits is called CMOS (Complementary Metal-Oxide Semiconductors)

- For CMOS the primary source of power dissipation is the so-called dynamic power - power consumed in switching

Power = Capacitive Load per Transistor x Voltage^2 x Frequency switched

Function of number of transistors connected to an output (fanout) and technology that determines the capacitance of both wires and transistors

This is a function of the Clock Rate

The power wall

- How can power grow by 30x while clock rates by 10^3x ?
  Reduce the voltage! (~15% per generation)

- Further lowering the voltage makes the transistors too leaky (like water faucets) that cannot shut off

- Even today 40% of consumption is due to leakage (**leakage current**, gate leakage)

- Alternative is to take that power and use it more efficiently or to cool the computers

After hitting the wall ⟶ MULTICORE

Multicore

- The sea change: since 2006 all microprocessor companies are shipping computers with multiple cores per chip
- New Moore: Double the number of cores per microprocessor per semi-conductor technology generation every two years

When do I need Parallelism ?

A program needs to be:

- correct
- solve an important problem
- provide a useful interface (to people and other programs)

OK
Sequential

- Fast
- Throughput
- UQ
- Optimization

Only
Parallel

Parallel computing challenges

- Reduce communication/synchronization overload
- Load balance
- Instructions: synchronization
- Arithmetics: associativity (instruction ordering matters)
- Advanced instructions: hardware and compilers
- Memory hierarchy: cache coherence (different caches with same values)
- I/O: redundant arrays of inexpensive disks (RAID: multiple resources for I/O - goes for fault tolerance as well)

Fallacies and pitfalls

Pitfall: Expecting the improvement of one aspect of a computer to increase overall performance by an amount proportional to the size of an improvement

Example: A program runs in 100s on a computer, with multiply operators responsible for 80s of this time. How much do I have to improve the speed of multiplication if I want to run 5 times faster?

$$\text{Exec. time after the improvement} = \frac{\text{Exec. time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected} =$$

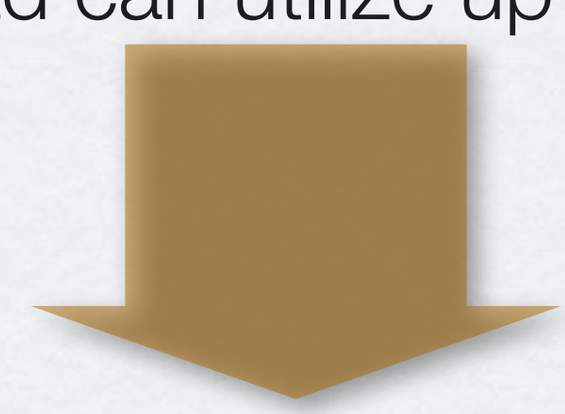
$$= \frac{80\text{s}}{n} + (100 - 80) = \frac{80}{n} + 20$$

Asking for the left side to be 20 requires $n = \infty$

Fallacies and pitfalls

Fallacy: Computers at low utilization use little power

Servers at 10% of load can utilize up to 2/3 of peak power



Towards Hardware for energy aware computing

Fallacies and pitfalls

Pitfall: Using a subset of performance equation as a performance metric

One alternative to Time for performance is MIPS (Million Instructions Per Second)

$$\text{MIPS} = \frac{\#\text{Instruction}}{\text{Execution time}} \times 10^{-6} = \frac{\#\text{Instructions}}{\frac{\#\text{Instructions} \times \text{CPI}}{\text{Clockrate}}} \times 10^{-6} = \frac{\text{Clock rate}}{\text{CPI}} \times 10^{-6}$$

Problem: instruction count may vary, MIPS varies between programs

$\Longrightarrow$ no single MIPS rating for computers

Execution time is the only valid and unimpeachable metric of performance

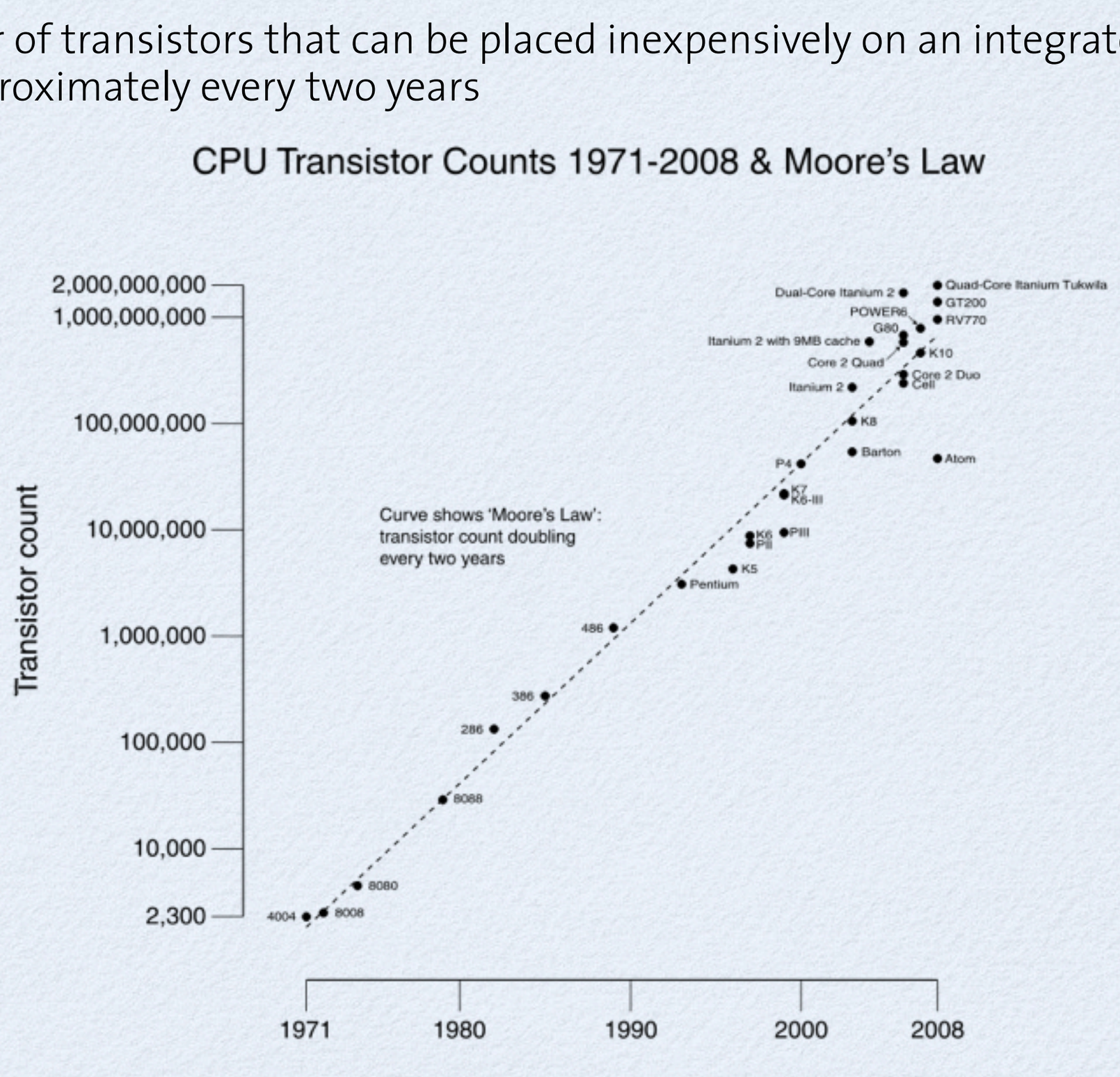
$$\text{Time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instructions}} \times \frac{\text{Seconds}}{\text{Clock cycles}}$$

Multi-Many Core = Parallel Computing

- Parallel computing = Exploit Concurrency of Computational Problems
- Concurrency exists in a computational problem when the problem can be decomposed into subproblems that can safely execute at the same time.(Mattson et al,in Patterns for parallel programming)

Moore's Law (still alive and well)

The number of transistors that can be placed inexpensively on an integrated circuit doubles approximately every two years



Single processor performance is no longer tracking Moore's Law

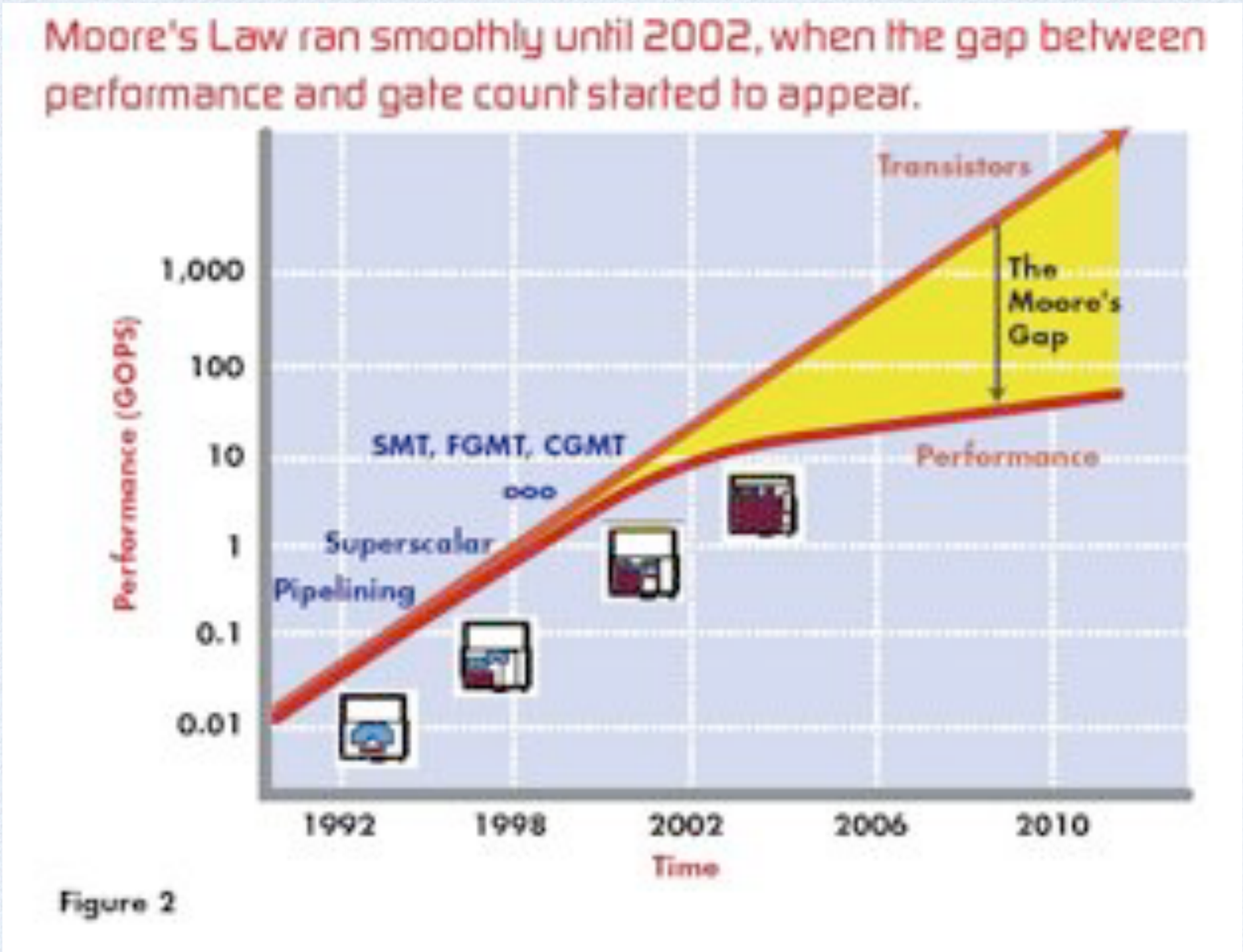


Figure 2

Credit : T. Schulthess

But we still have Moore's Law

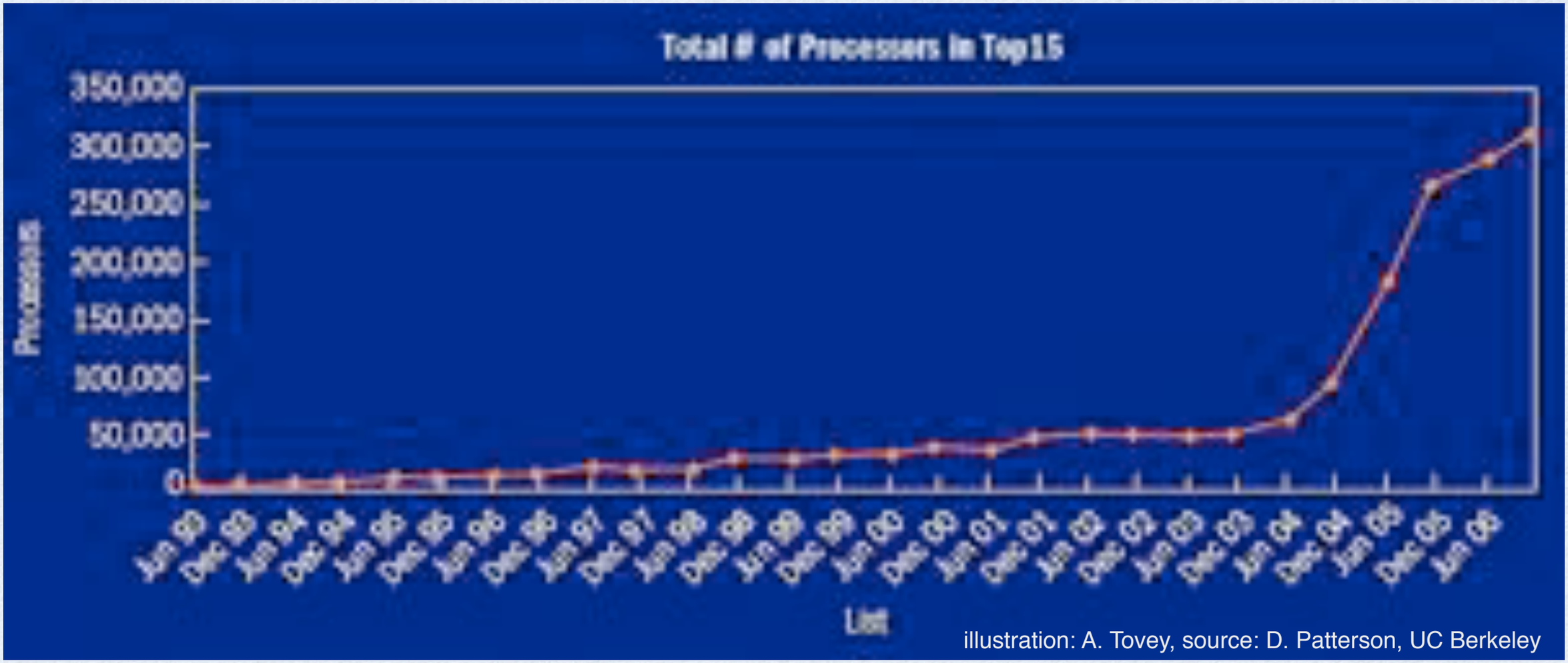


illustration: A. Tovey, source: D. Patterson, UC Berkeley

Performance Increase due to exploding number of processing cores

MultiCore Architectures

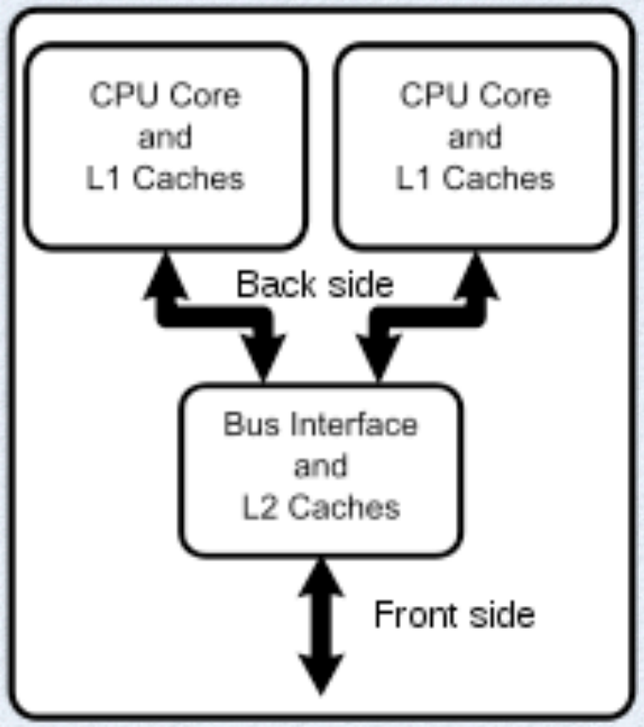
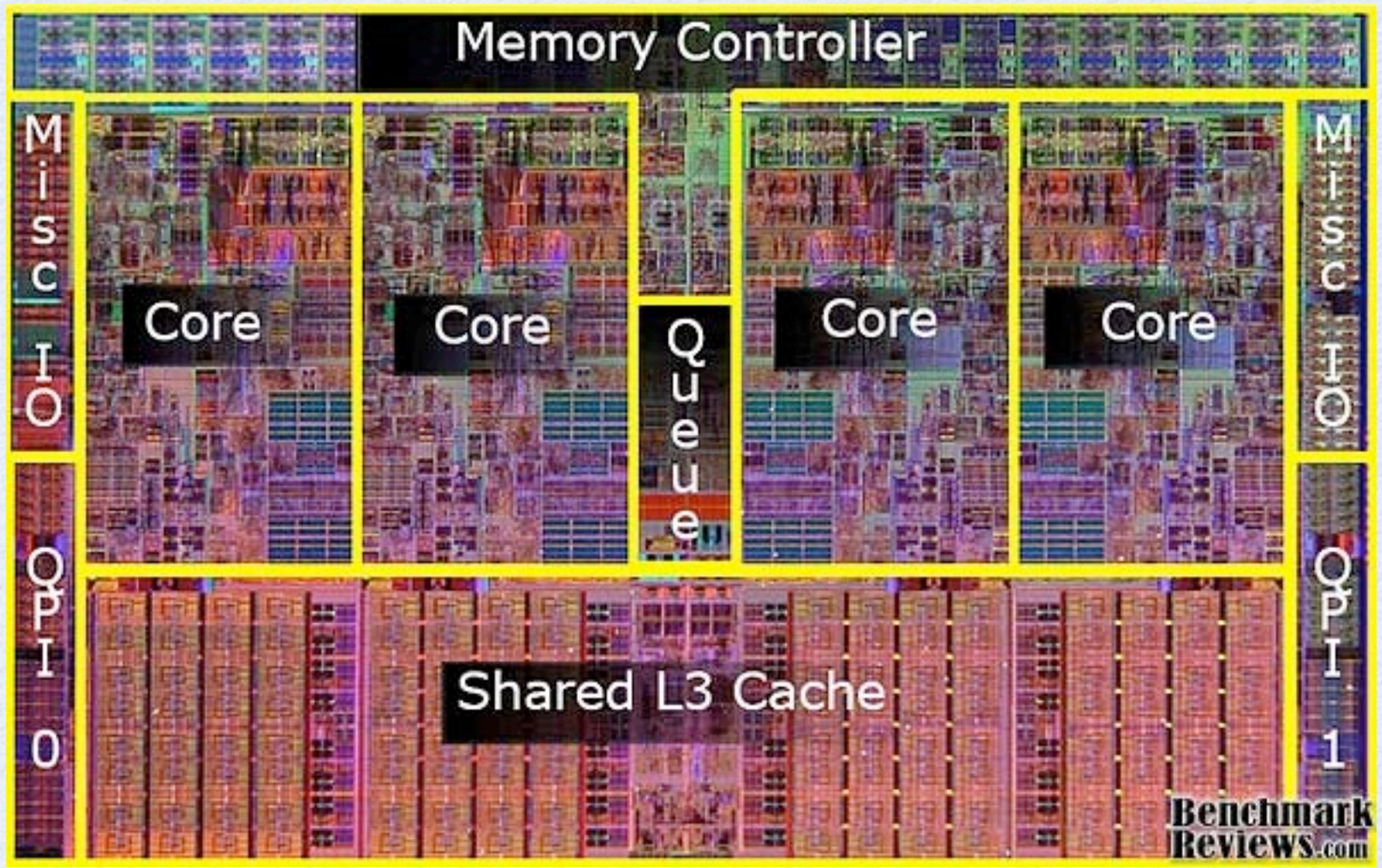


Diagram of a generic dual-core processor, with CPU-local level 1 caches, and a shared, on-die level 2 cache.



Intel QuickPath Architecture

A **multi-core processor** is a single computing component with two or more independent actual processors (called "cores"), which are the units that read and execute program instructions.[1] The data in the instruction tells the processor what to do. The instructions are very basic things like reading data from memory or sending data to the user display, but they are processed so rapidly that human perception experiences the results as the smooth operation of a program. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package. (SOURCE : Wikipedia)

Moore's Law Continues

- technology scaling (32 nm in 2010),
- improving transistor performance to increase frequency,
- increasing transistor integration capacity to realize complex architectures,
- reducing energy consumed per logic operation to keep power dissipation within limit.

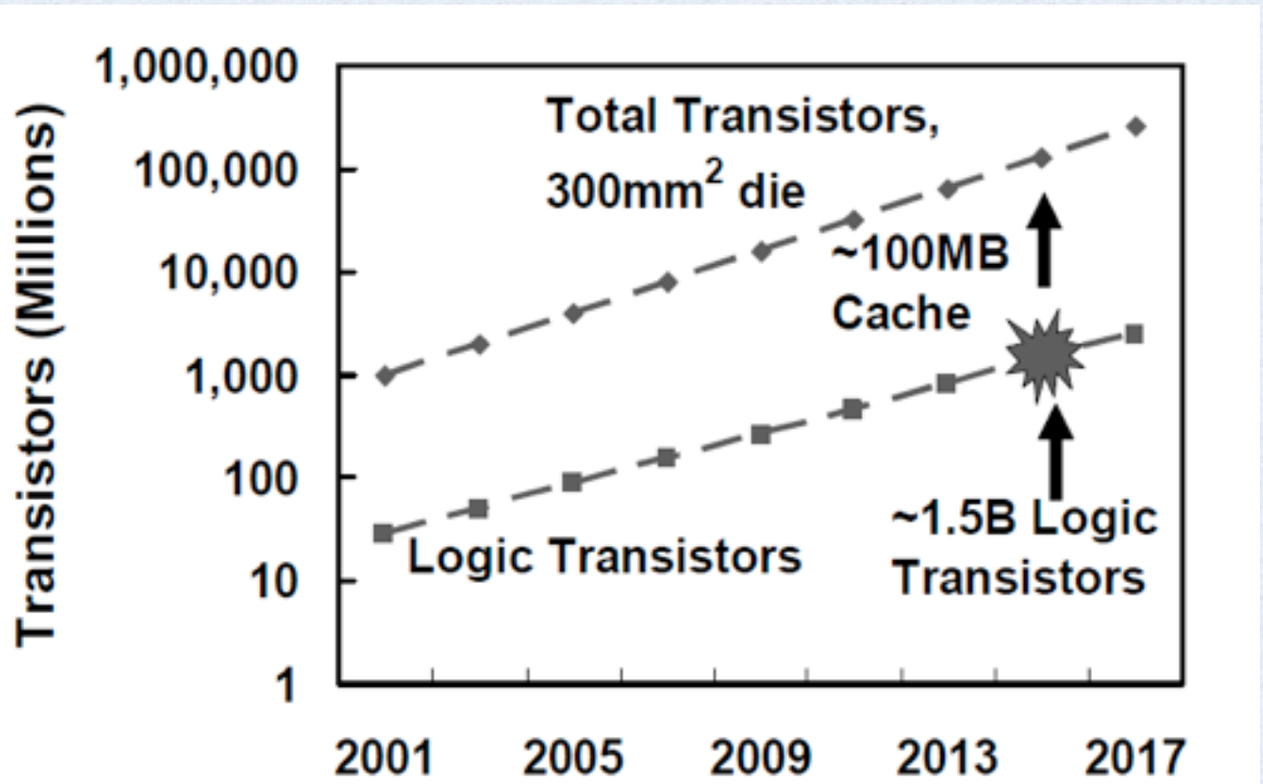


Figure 1: Transistor integration capacity

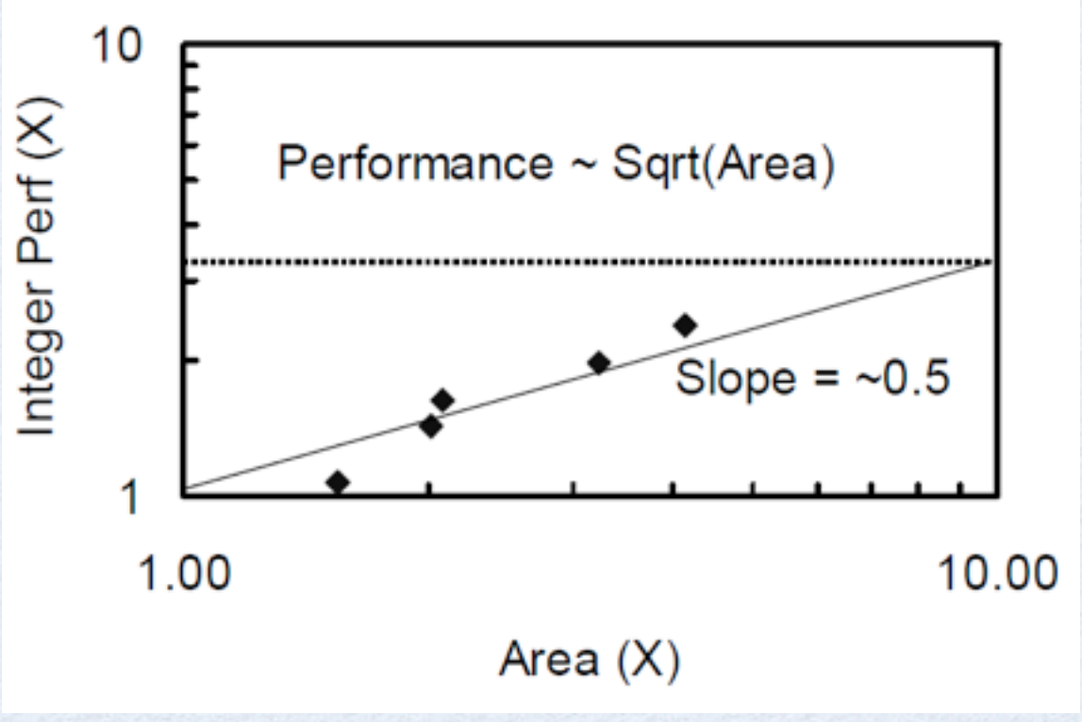
Shekhar Borkar, Thousand Core Chips - A Technology Perspective, in Intel Corp,Microprocessor Technology Lab, 2007, p. 1-4

MultiCore Architectures

Pollack's rule

- microprocessor **"performance increase due to microarchitecture advances is roughly proportional to [the] square root of [the] increase in complexity".** This contrasts with power consumption increase, which is roughly linearly proportional to the increase in complexity. **Complexity** in this context means processor logic, i.e its **area**.(source Wikipedia)

- if you double the logic in a processor core, then it delivers only 40%more performance
- A multi-core microarchitecture has potential to provide near linear performance improvement with complexity and power.
  - For example, two smaller processor cores, instead of a large monolithic processor core, can potentially provide 70-80% more performance, as compared to only 40% from a large monolithic core



More from Multicore

Multi-core processors have several other benefits:

- each processor core could be individually turned on or off, thereby saving power;
- each processor core can be run at its own optimized supply voltage and frequency;
- easier to load balance among processor cores to distribute heat across the die;
- can potentially produce lower die temperatures improving reliability and leakage.

7 Questions for Parallel Computing

Applications

1. What are the applications?
2. What are common kernels of the applications?

Hardware

3. What are the hardware building blocks?
4. How to connect them?

Programming Models

5. How to describe applications and kernels?
6. How to program the hardware?

Evaluation:

7. How to measure success?

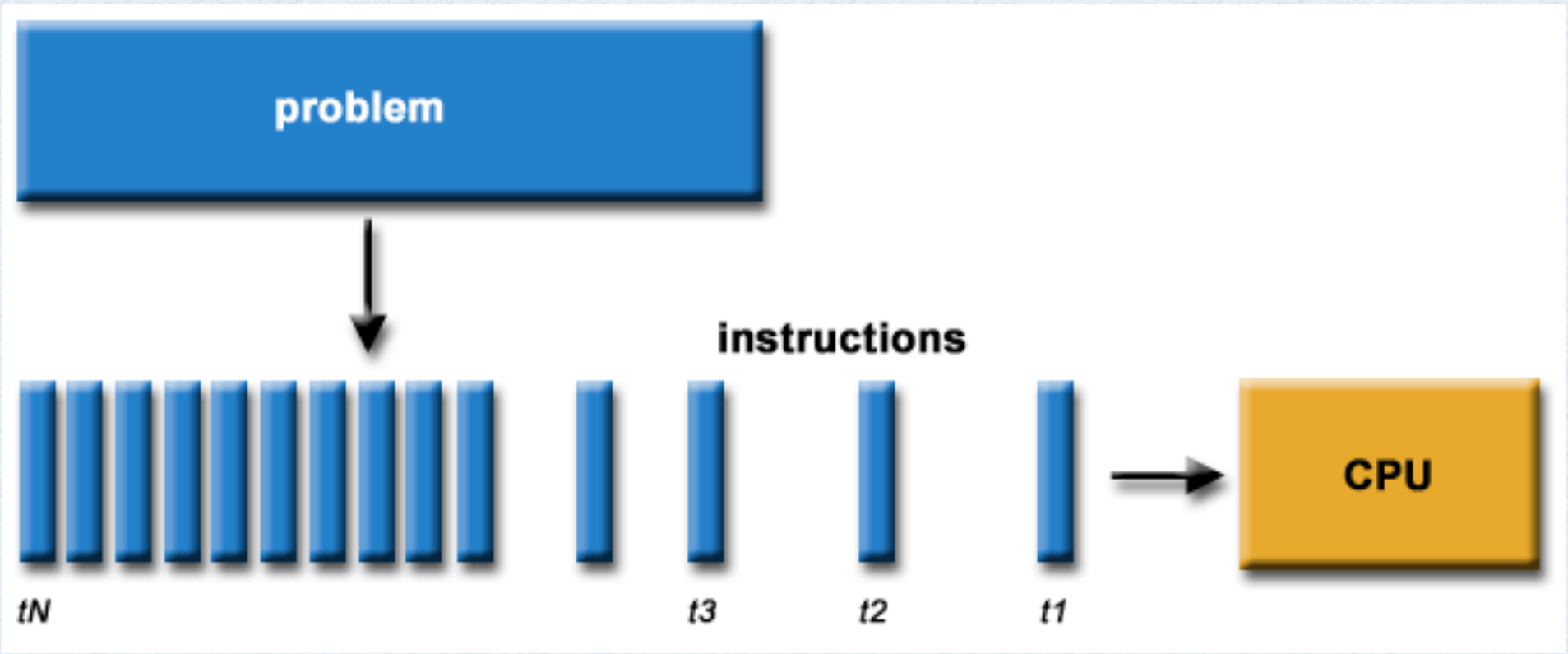
From Asanovic et.al., 2006, **The Landscape of Parallel Computing Research: A View from Berkeley**

CREDITS

- THE FOLLOWING NOTES ARE TAKEN FROM THE EXCELLENT WEB TUTORIAL OF **BLAISE BARNEY**
- https://computing.llnl.gov/tutorials/parallel_comp/

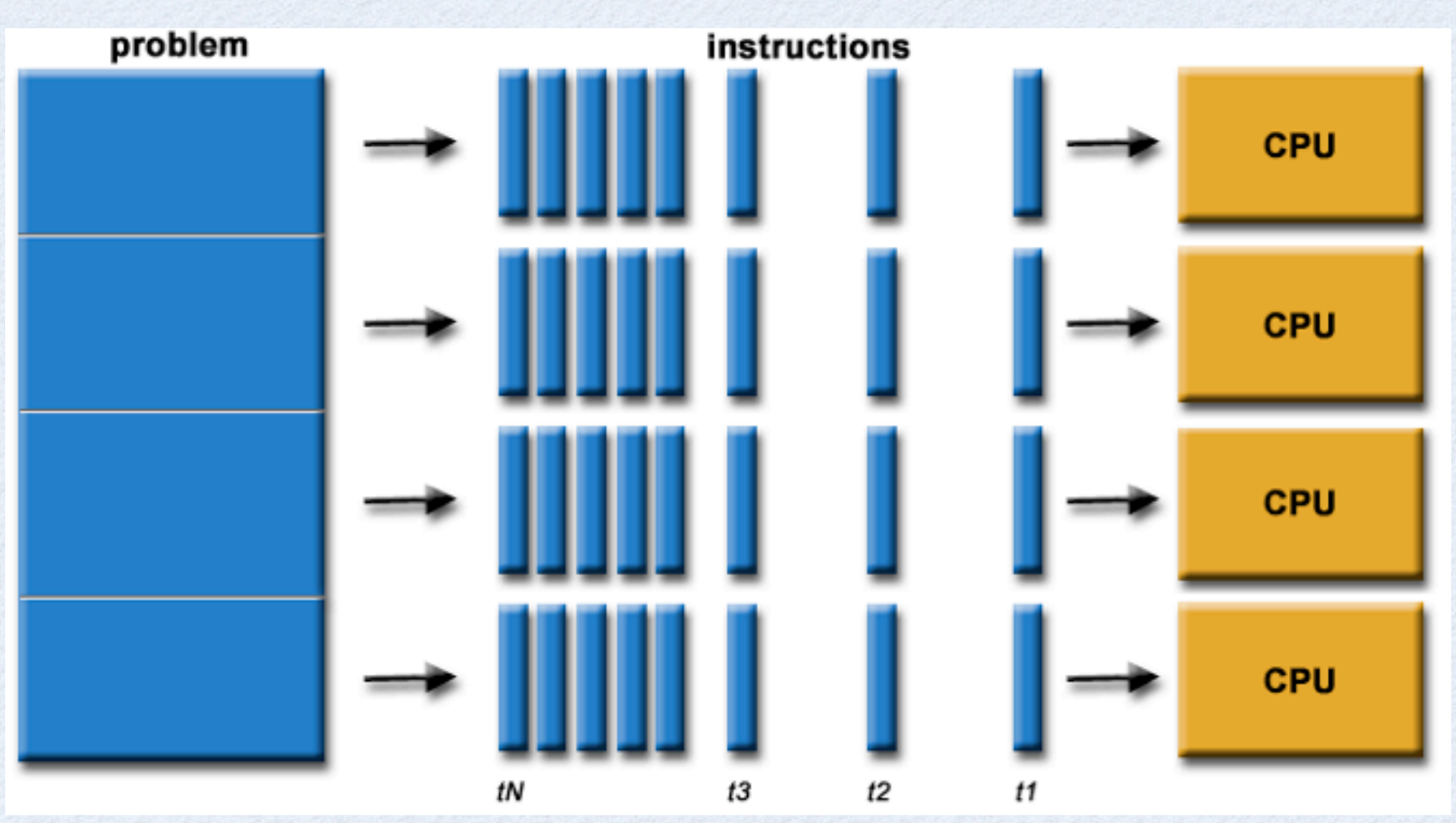
Serial Computing

- Serial Implementation of Programs for single CPUs
  - Program has series of Instructions
  - Executed one after the other
  - One instruction at a time



Parallel Computing

- Program is executed using Multiple Machines
- 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 machines (here : CPUs)



Parallel Computing : Machines and Problems

MACHINES

- Single computer with multiple processors;
- An arbitrary number of computers connected by a network;
- A combination of both.

PROBLEM

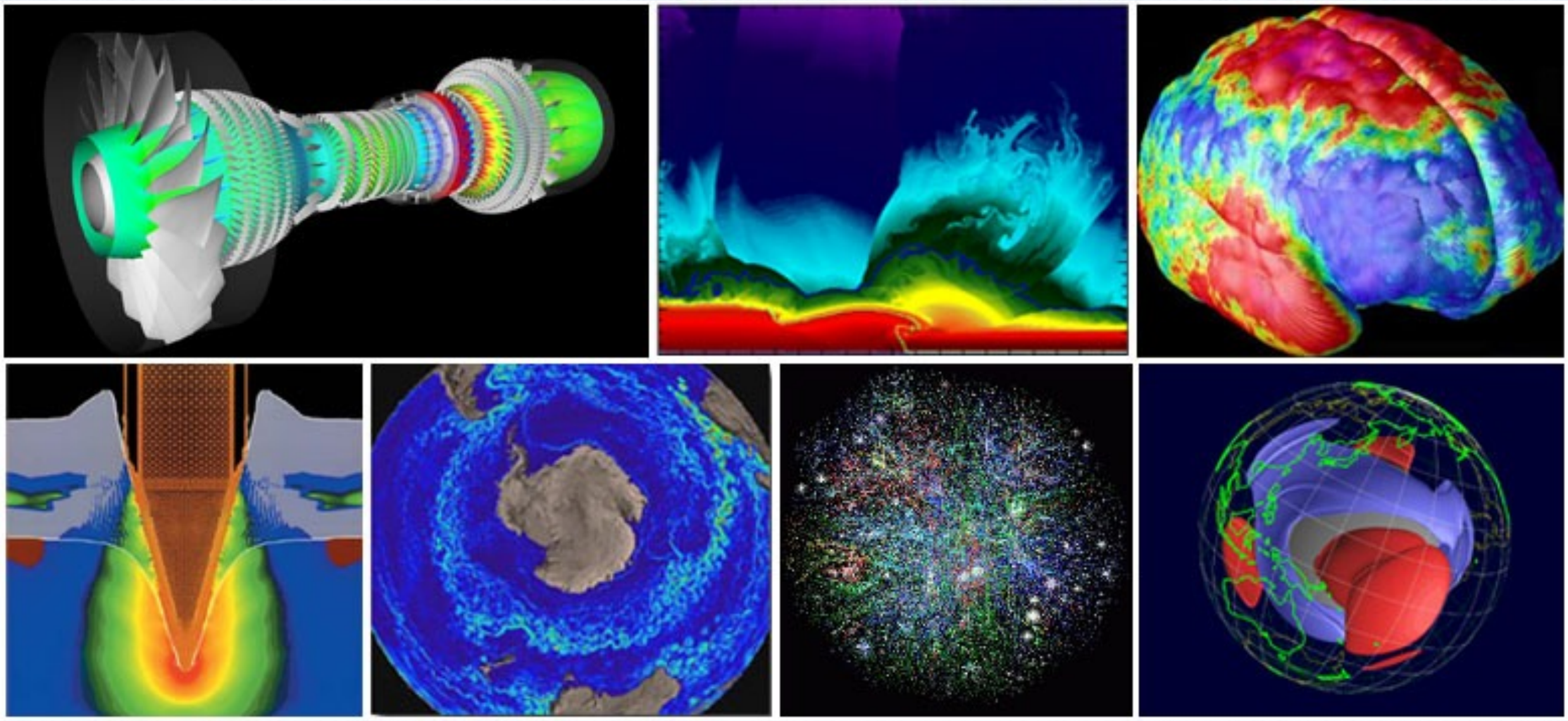
- Broken into distinct pieces of work solved in parallel;
- Execute multiple program instructions at any time;
- Solved in less time with multiple compute resources than with a single compute resource.

The world is parallel

Interrelated Events Happening Concurrently

PAST : Parallel Computing = High End Science

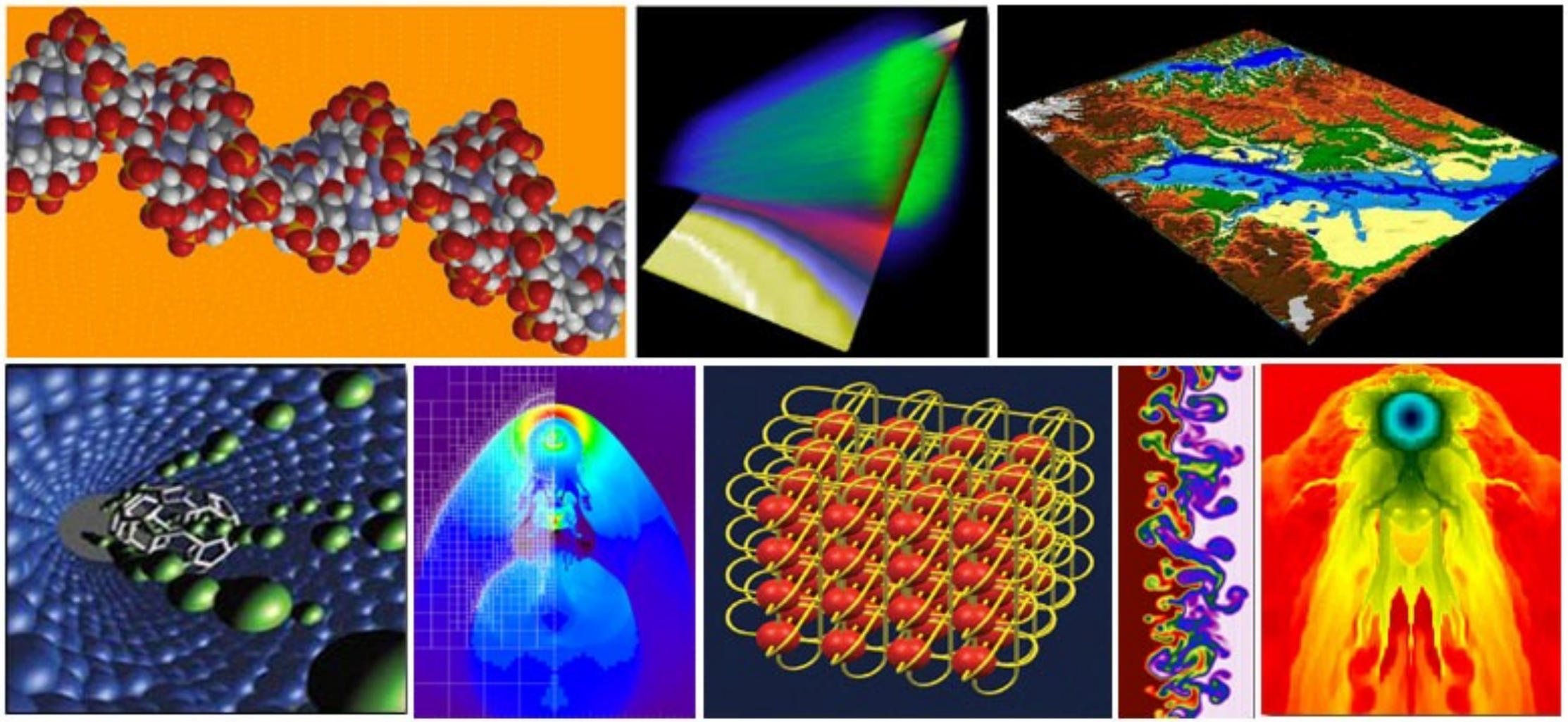
- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Science



- Geology, Seismology
- Mechanical Engineering - from prosthetics to spacecraft
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics

TODAY : Parallel Computing
= Everyday (industry, academia) Computing

- Pharmaceutical design
- Medical imaging and diagnosis
- Oil exploration
- Product Design
- Databases, data mining
- Web search engines, web based business services

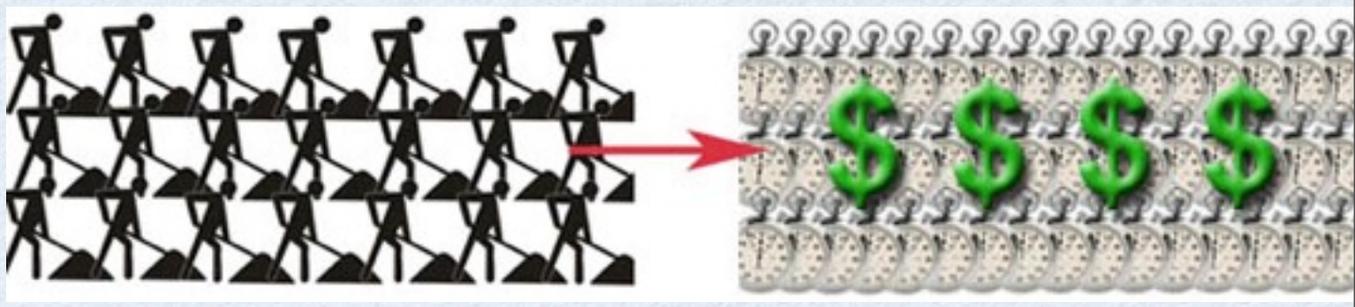


- Management of national and multi-national corporations
- Financial and economic modeling
- Advanced graphics and virtual reality(entertainment industry)
- Networked video and multi-media technologies
- Collaborative work environments

I. Why Parallel Computing ?

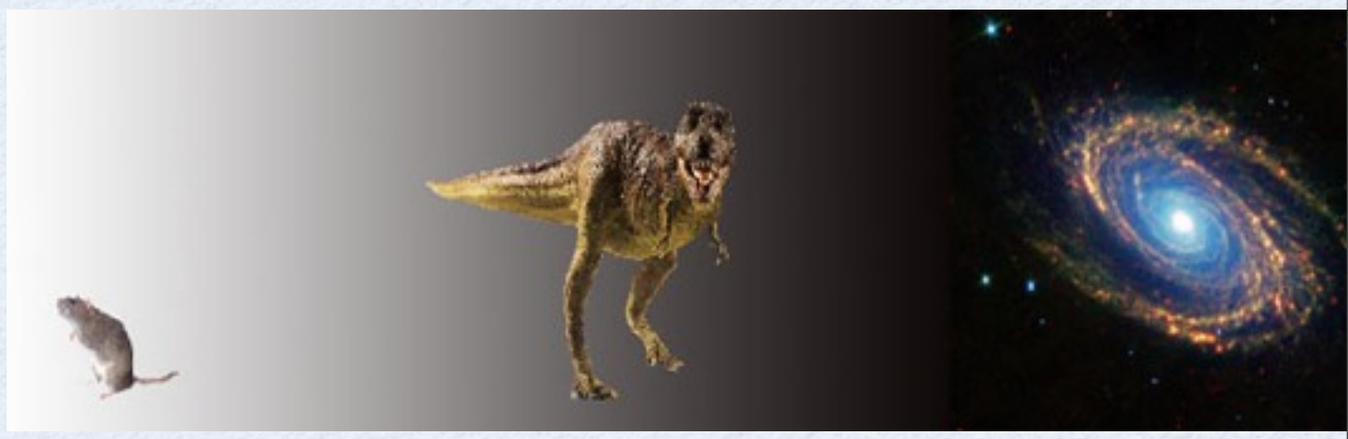
Save time and/or money:

Shorten Time to Completion - Parallel Computers built from commodity components



Solve Larger Problems

- "Grand Challenge" (en.wikipedia.org/wiki/Grand_Challenge) problems requiring PetaFLOPS and PetaBytes of computing resources.
- Web search engines/databases processing millions of transactions per second



Provide Concurrency

- A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.
- For example, the Access Grid (www.accessgrid.org) provides a global collaboration network where people from around the world can meet and conduct work "virtually".

Use Non-Local Resources

- SETI@home (setiathome.berkeley.edu) uses 2.9 million computers in 253 countries. (July 2011)
- Folding@home (folding.stanford.edu) uses over 450,000 cpus globally (July 2011)

II. Why Parallel Computing ?

Physical/Practical constraints for even faster serial computers:

- **Transmission speeds** - the speed of a serial computer is directly dependent upon how fast data can move through hardware. Absolute limits are the speed of light (30 cm/nanosecond) and the transmission limit of copper wire (9 cm/nanosecond). Increasing speeds necessitate increasing proximity of processing elements.

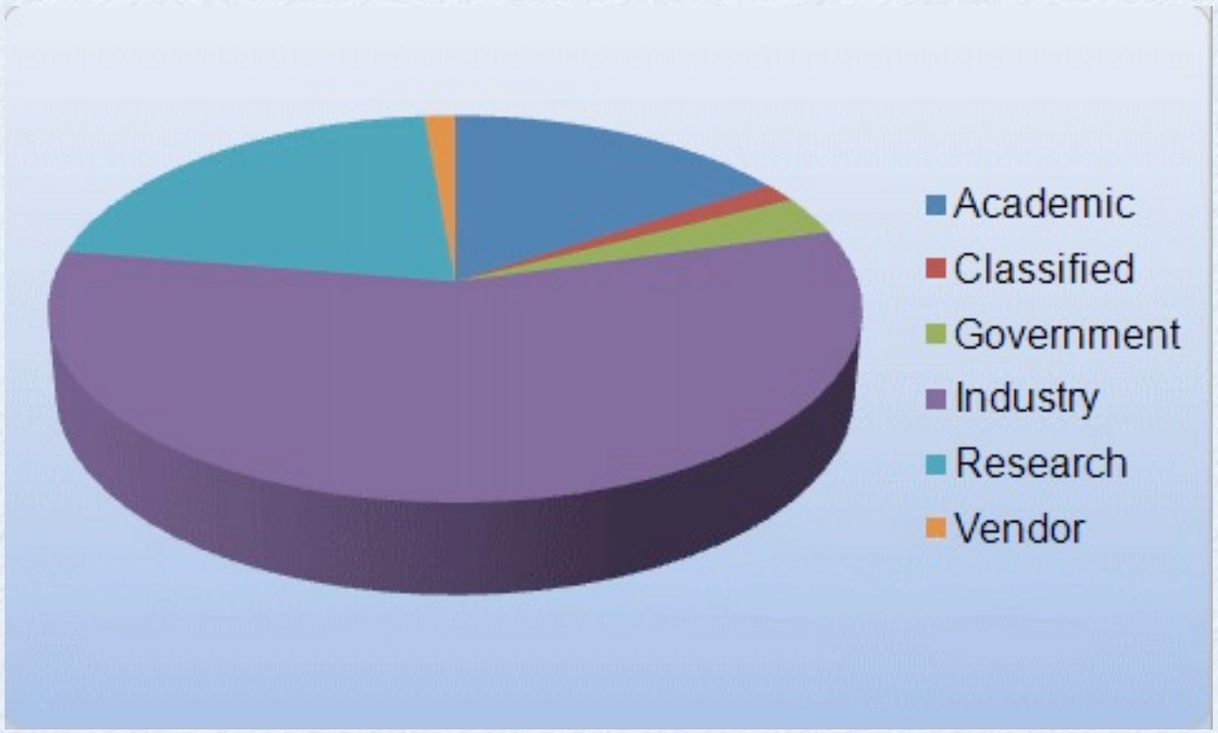
- **Limits to miniaturization** - processor technology is allowing an increasing number of transistors to be placed on a chip. However, even with molecular or atomic-level components, a limit will be reached on how small components can be.

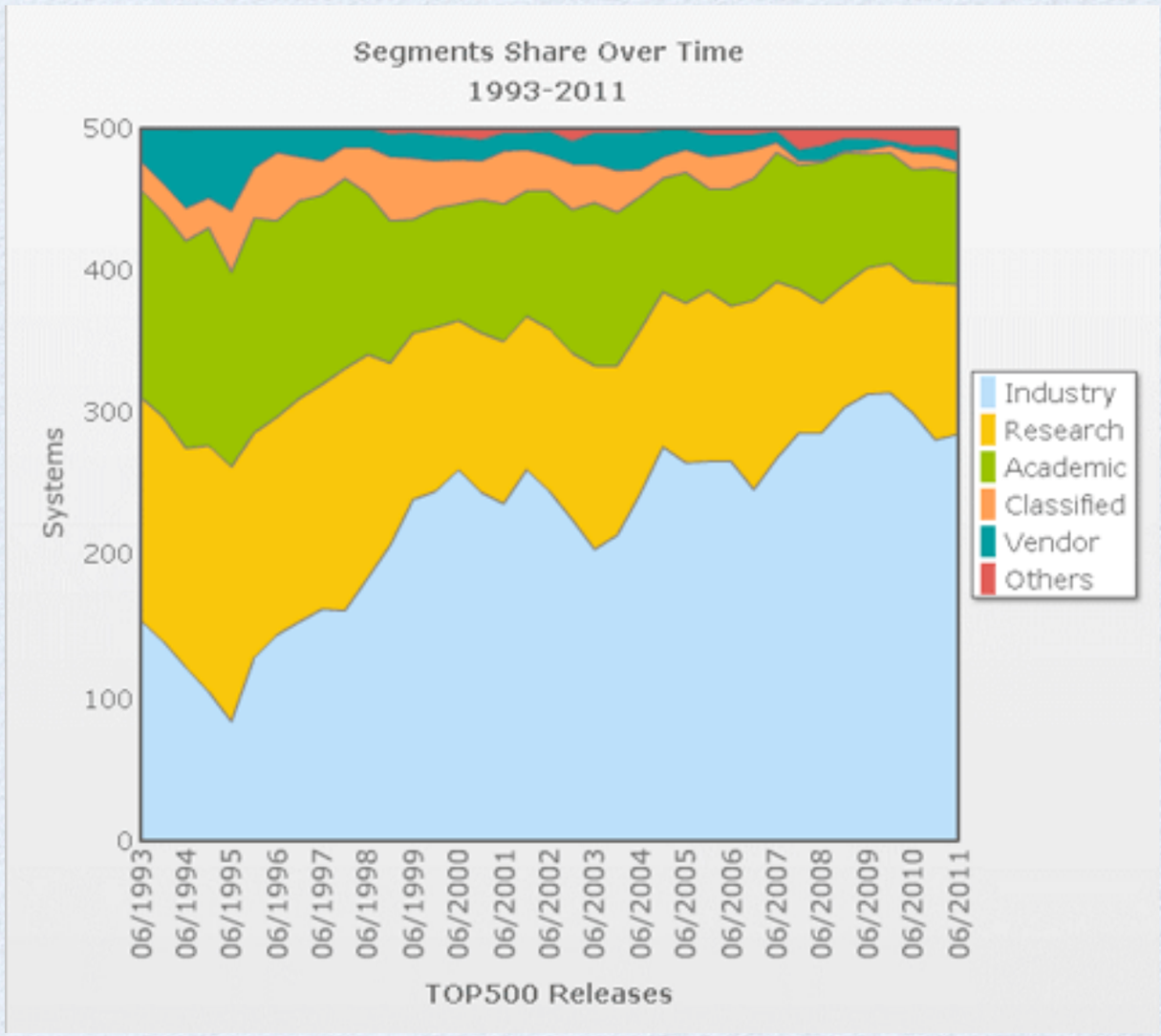
- **Economic limitations** - it is increasingly expensive to make a single processor faster. Using a larger number of moderately fast commodity processors to achieve the same (or better) performance is less expensive.

- **Energy Limits** - limits imposed by cooling needs for chips and supercomputers.

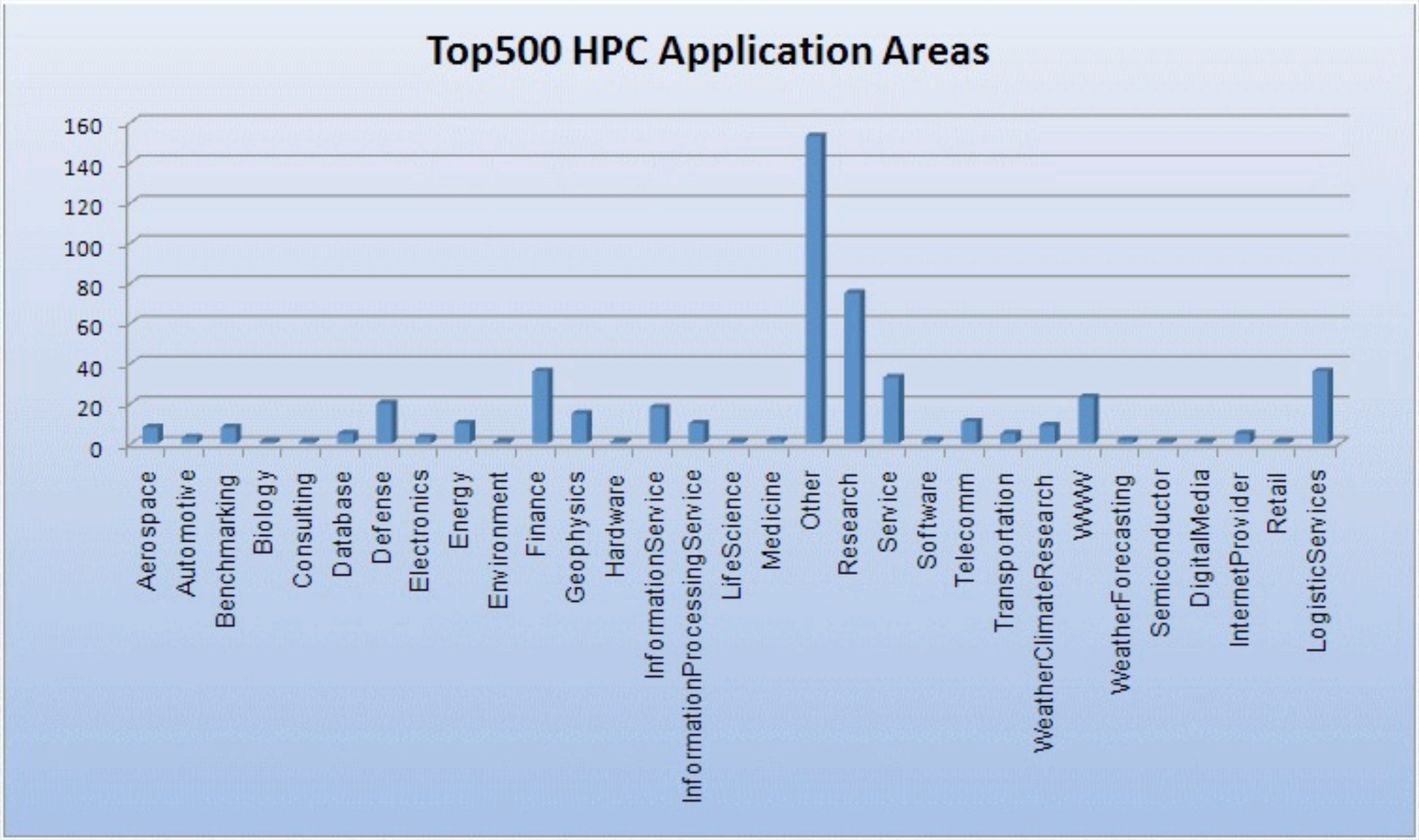
  - **Current computer architectures are increasingly relying upon hardware level parallelism to improve performance:**
  - **Multiple execution units**
  - **Pipelined instructions**
  - **Multi-core**

Who and What ? – top500.org

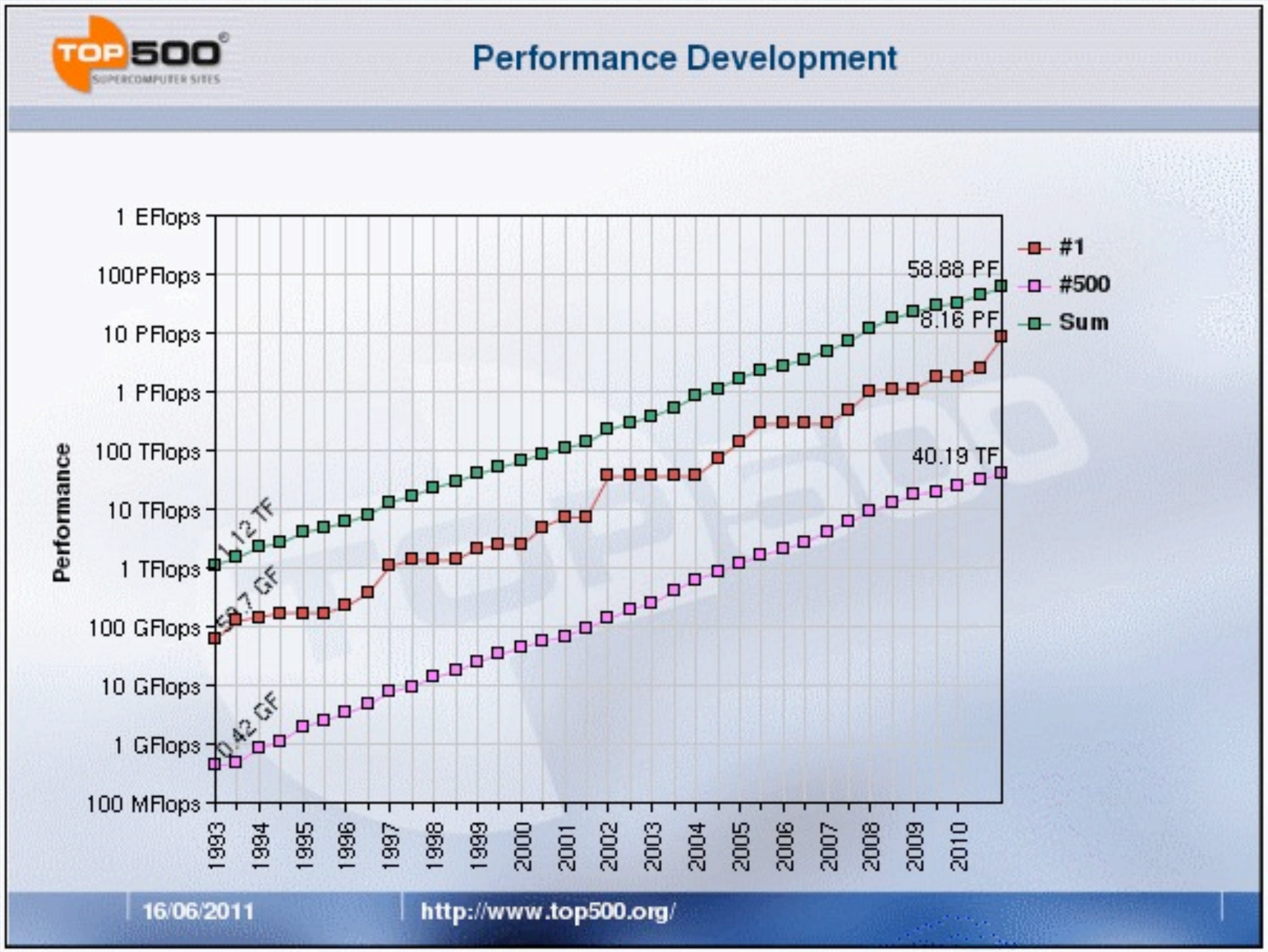




Top 500 Application Areas



Performance Development

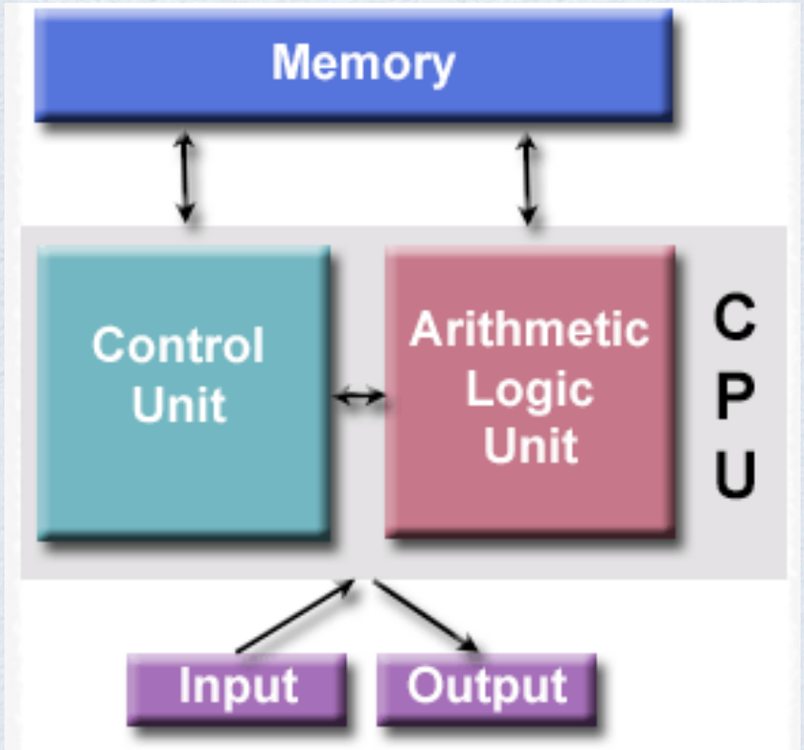


Concepts – Terminology

von Neumann Architecture

Four main components:

- Memory
- Control Unit
- Arithmetic Logic Unit
- Input/Output



- **Read/write, Random Access Memory** stores both program instructions and data
  - Program instructions are coded data which tell the computer to do something
  - Data is simply information to be used by the program
- **Control unit** fetches instructions/data from memory, decodes the instructions and then ***sequentially*** coordinates operations to accomplish the programmed task.
- **Arithmetic Unit** performs basic arithmetic operations
- **Input/Output** is the interface to the human operator

Many parallel computers follow this basic design, just multiplied in units. The basic, fundamental architecture remains the same.

Elements of Parallel Computers

- **Hardware**
  - Multiple Levels of Parallelism (ILP, DLP, TLP)
  - Multiple Memories
  - Interconnection Network

- **System Software**
  - Parallel Operating System
  - Programming Constructs to Orchestrate Concurrency

- **Application Software**
  - Parallel Algorithms

- ***GOALs***
  - **Achieve Speedup: Tp = Ts/p**
  - **Solve problems requiring a large amount of memory**

Parallel Computing Platforms

- **LOGICAL Organization**

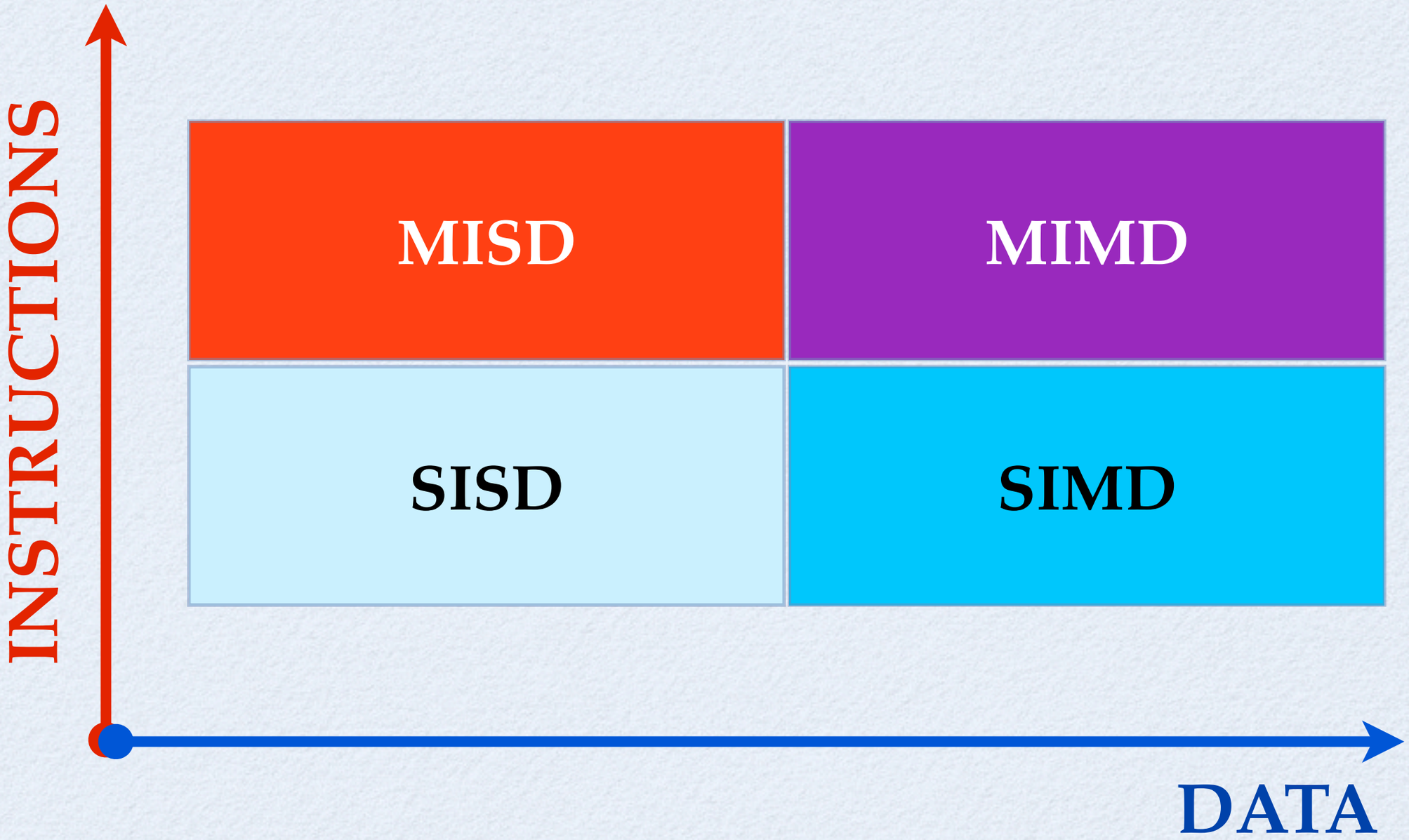
The user's view of the machine as it is being presented via its system software

- **PHYSICAL Organization**

The actual hardware architecture

NOTE : Physical Architecture is to a large extent independent of the Logical Architecture

Logical Organization Elements

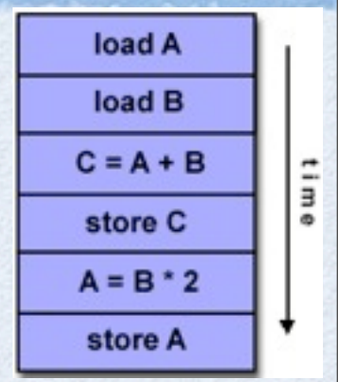


Flynn's Taxonomy (1966)

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 and even today, the most common type of computer



UNIVAC1

IBM360

CRAY I

CDC7600

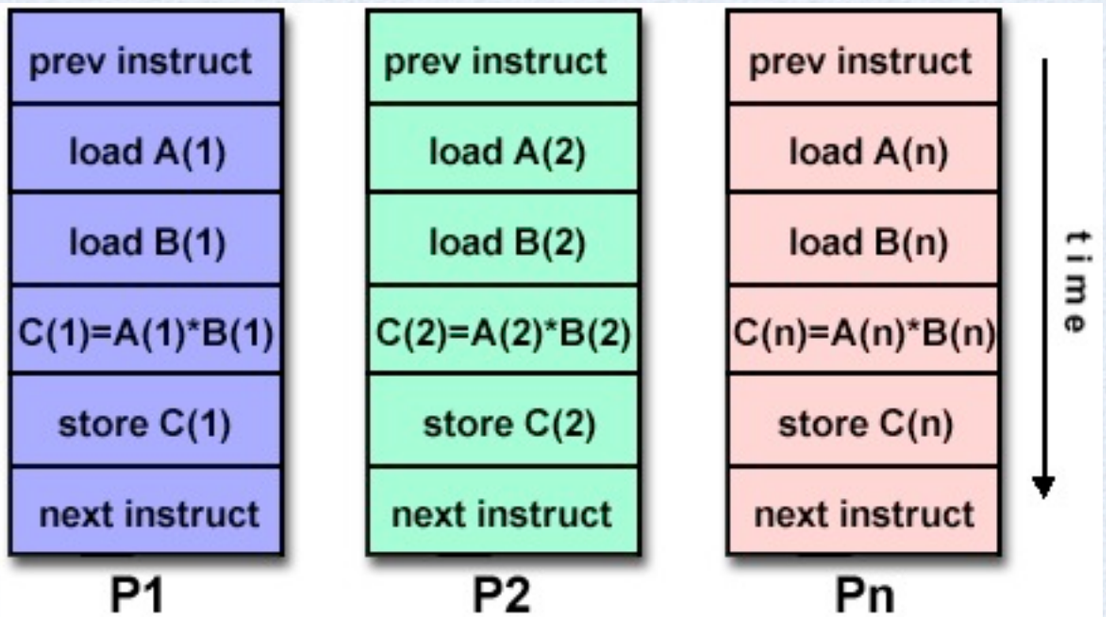
PDP1

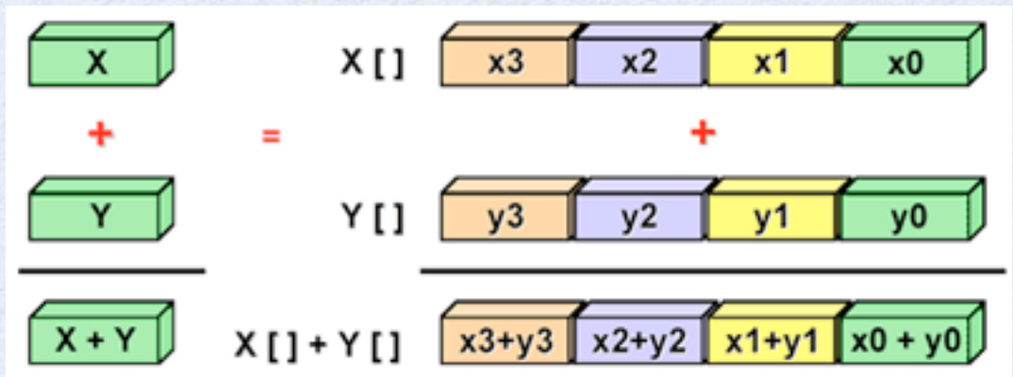
Dell Laptop

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: Connection Machine 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 GPUs employ SIMD instructions and execution units

SIMD Machines

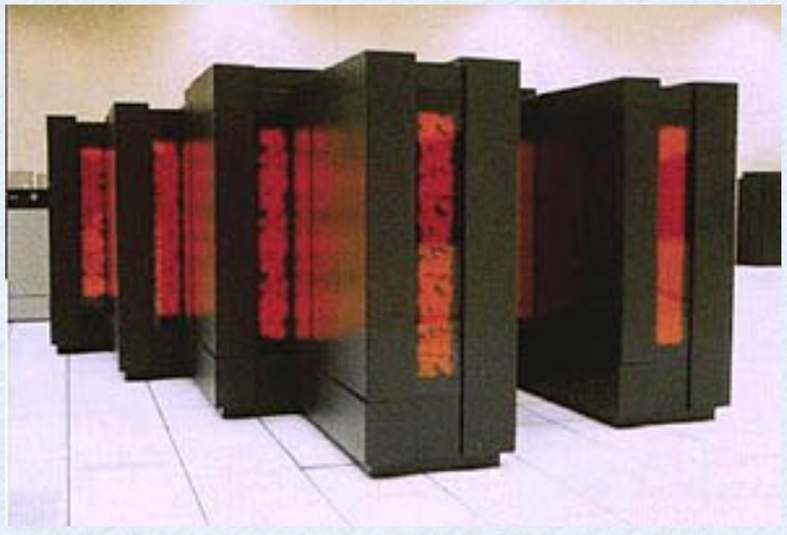
ILLIAC IV

MasPar

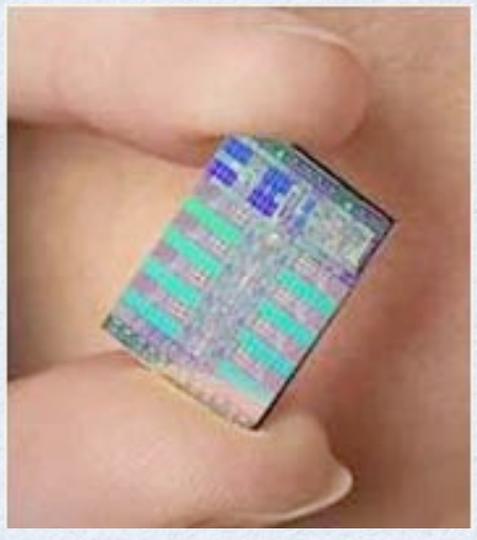
Cray X-MP

Cray Y-MP

Thinking Machines CM-2



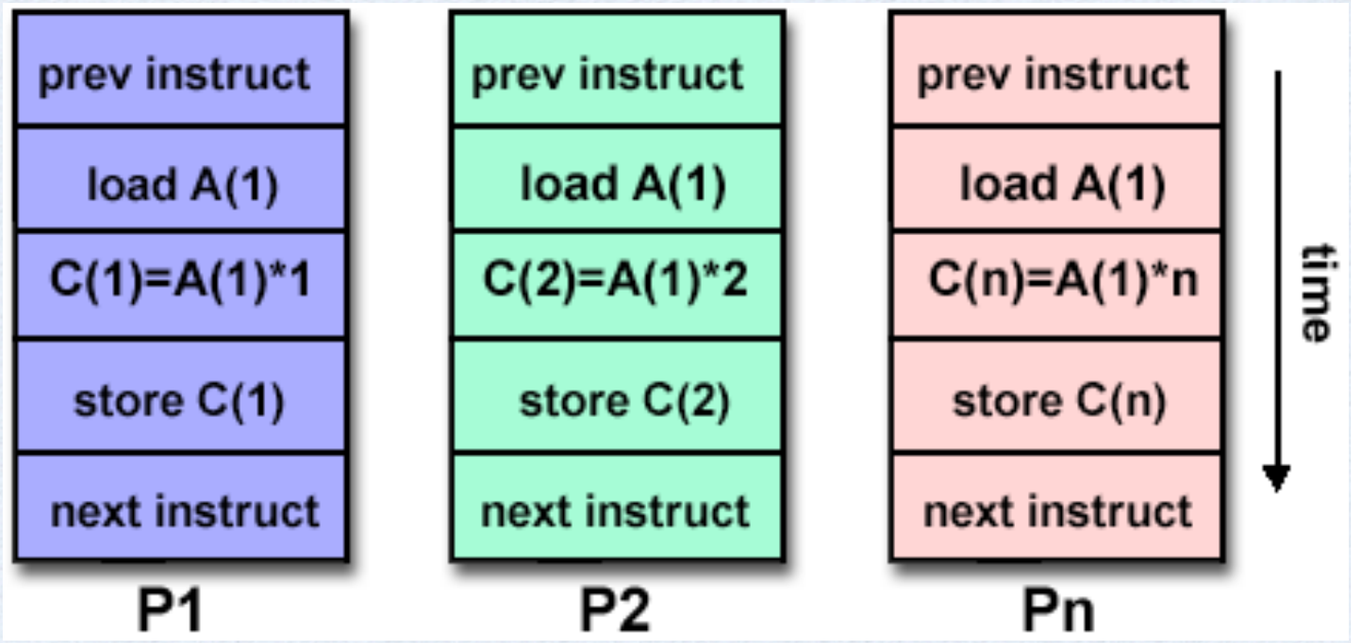
Cell Processor



MISD

Multiple Instruction: Each processing unit operates on the data independently via separate instruction streams.

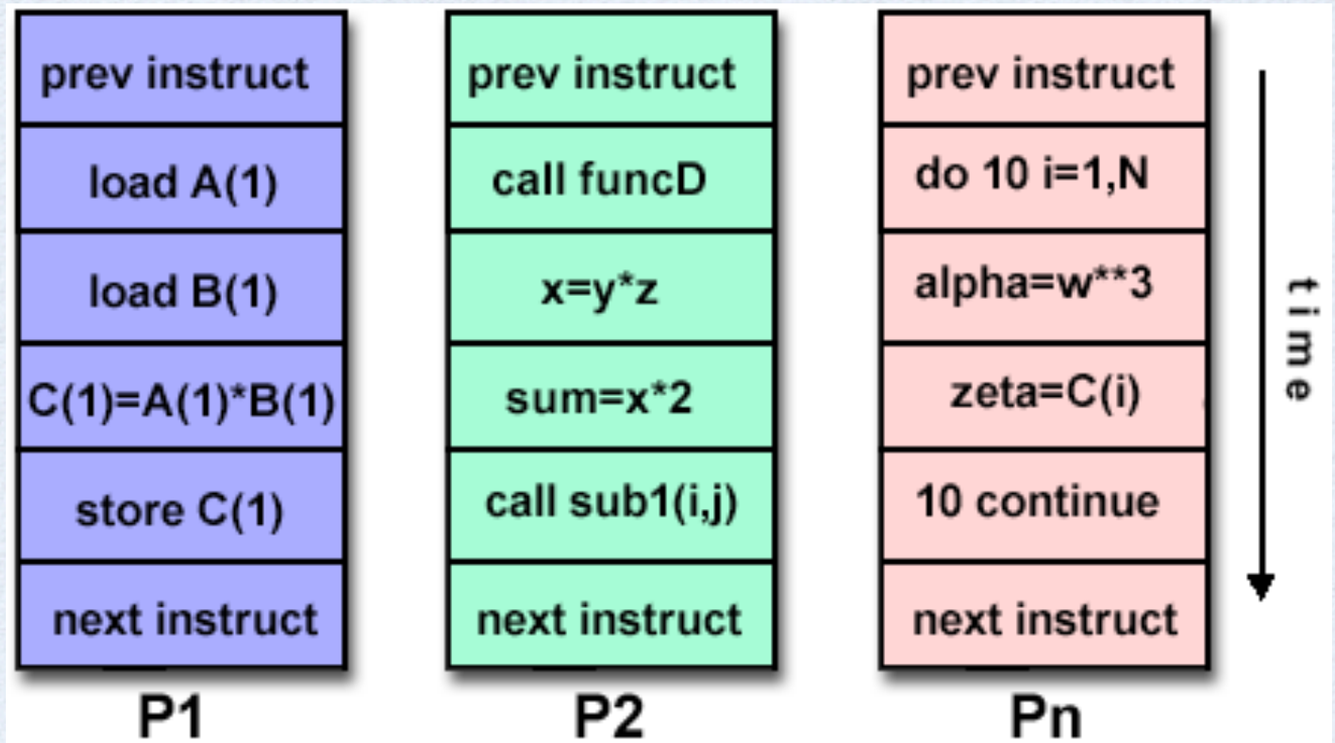
- **Single Data:** A single data stream is fed into multiple processing units.



- Few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.

MIMD

- **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

MIMD Machines

IBM POWER5

HP/Compaq Alphaserver

Intel IA32

AMD Opteron

Cray XT3

IBM BG/L

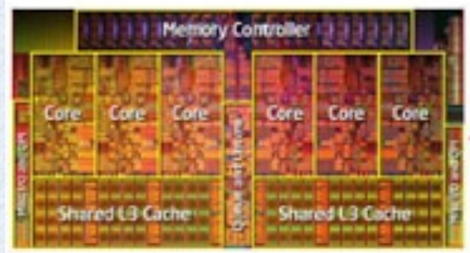
Terminology I

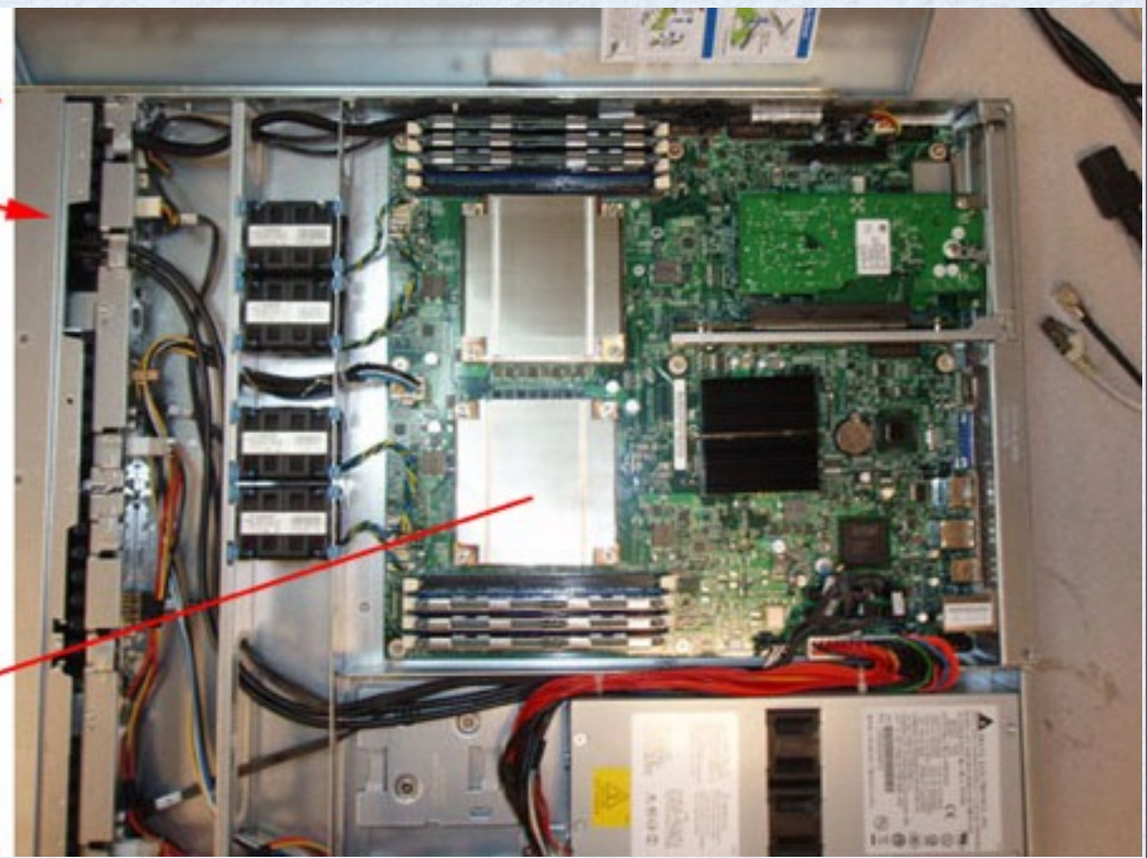
Node
A standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores. Nodes are networked together to comprise a supercomputer.

Supercomputer - each blue light is a node

Node - standalone Von Neumann computer

CPU / Processor / Socket - each has multiple cores / processors.





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.

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.

Terminology II

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)

Hardware architecture : multiple processors share a single address space and access to all resources; shared memory computing.

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.

Terminology III

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

The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead includes:

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

Terminology IV

Massively Parallel
Refers to the hardware that comprises a given parallel system - having many processors. The meaning of "many" keeps increasing, but currently, the largest parallel computers can be comprised of processors numbering in the hundreds of thousands.

Embarrassingly 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 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

- Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.

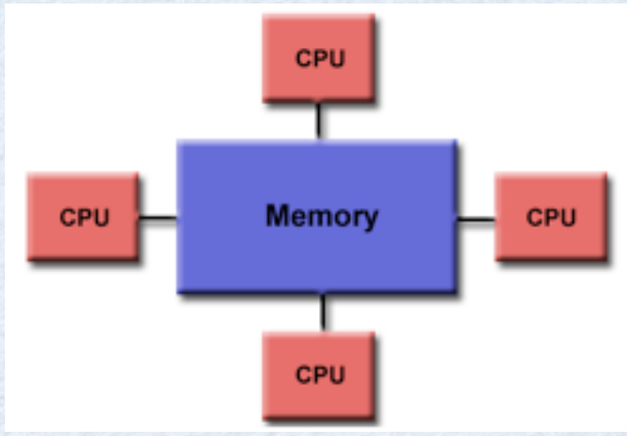
- Multiple processors can operate independently but share the same memory resources.

- Changes in a memory location effected by one processor are visible to all other processors.

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

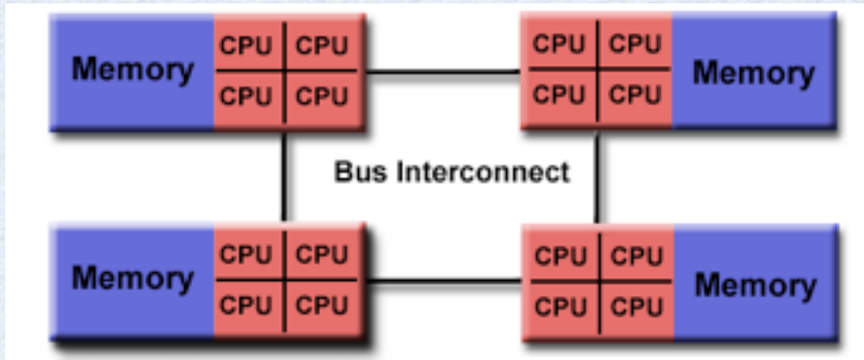
UMA vs NUMA

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.

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

UMA **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

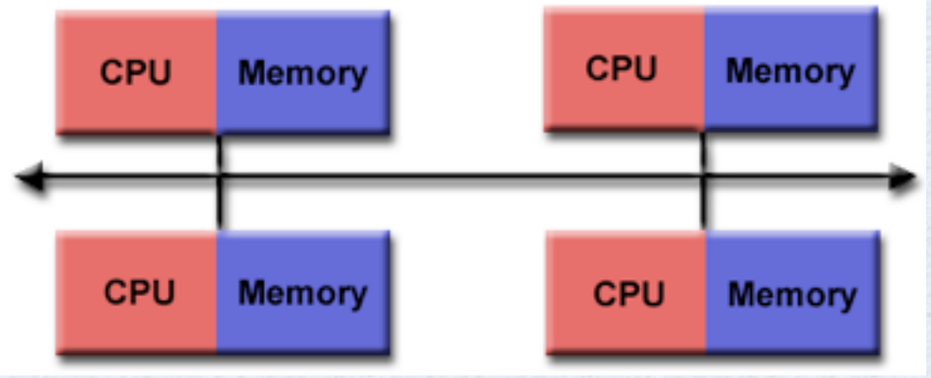
UMA **Disadvantages:**

- 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.
- Expense: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors.

Parallel Computer Memory Architectures : DISTRIBUTED MEMORY

Distributed memory systems also vary widely but share a common characteristic :
they 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 can be as simple as Ethernet.



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 cache coherency.
- Cost effectiveness: can use commodity, off-the-shelf processors and networking.

Disadvantages:

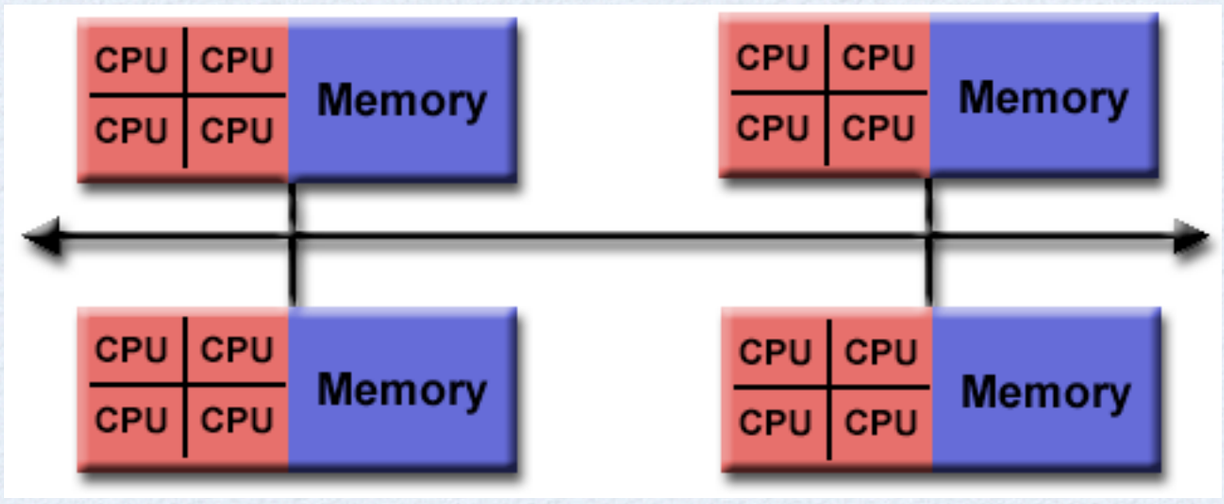
- The programmer is responsible for many of the details associated with data communication between processors.
- It may be difficult to map existing data structures, based on global memory, to this memory organization.
- Non-uniform memory access (NUMA) times

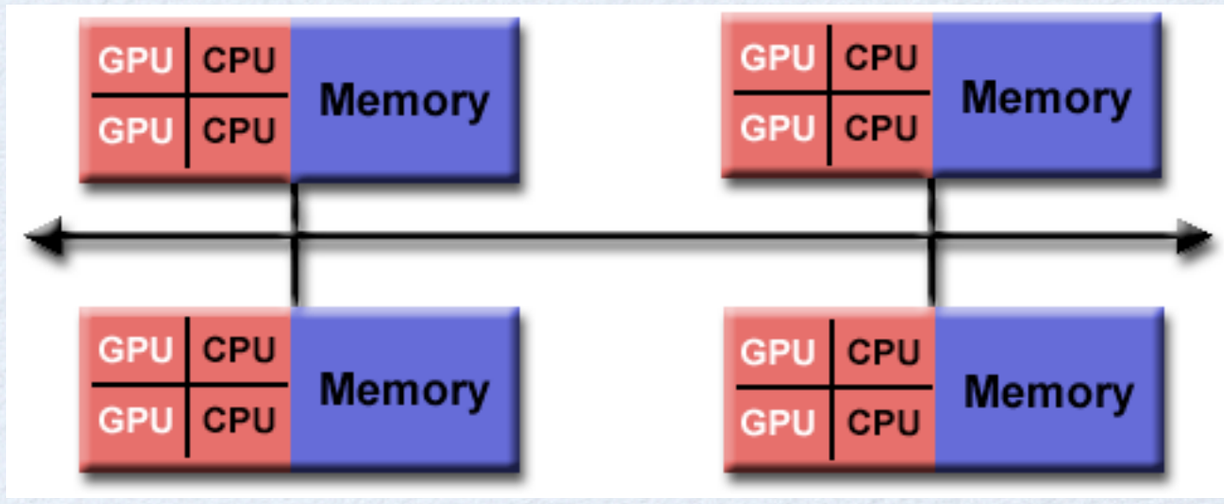
Parallel Computer Memory Architectures : HYBRID DISTRIBUTED/SHARED MEMORY

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

- The shared memory component can be a cache coherent SMP machine and/or graphics processing units (GPU).
- The distributed memory component is the networking of multiple SMP/GPU machines, which know only about their own memory - not the memory on another machine. Network communications required to move data between SMP/GPUs

- 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.

Programming Models

PROGRAMMING MODELS : OVERVIEW

There are several parallel programming models in common use:

1. **Shared Memory (without threads)**
2. **Threads**
3. **Distributed Memory / Message Passing**
4. **Data Parallel**
5. **Hybrid**
6. **Single Program Multiple Data (SPMD)**
7. **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.

- **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.

1.SHARED MEMORY MODEL (no threads)

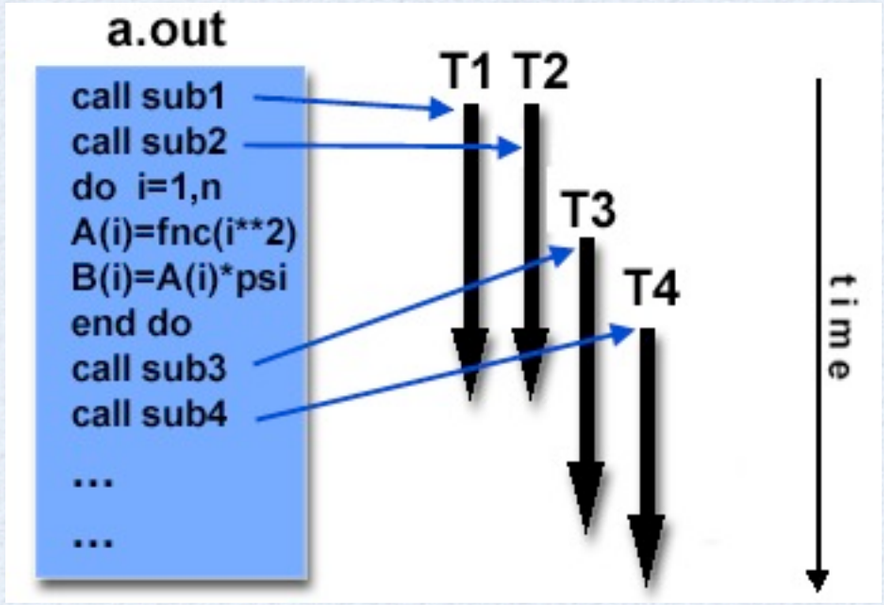
- Tasks share a common address space, which they read and write to asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. 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 processor that works on it conserves memory accesses, cache refreshes and bus traffic that occurs when multiple processors use the same data.
    - Unfortunately, controlling data locality is hard to understand and beyond the control of the average user.

Implementations:

- Native compilers and/or hardware translate user program variables into actual memory addresses, which are global. On stand-alone SMP machines, this is straightforward.
- On distributed shared memory machines, such as the SGI Origin, memory is physically distributed across a network of machines, but made global through specialized hardware and software.

2.SHARED MEMORY MODEL (with threads)

- In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
- Perhaps the most simple analogy that can be used to describe threads is the concept of a single program that includes a number of subroutines:
    - 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.
    - 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. 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.



SHARED MEMORY MODEL (with threads)

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 all parallelism.

- 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***.

OpenMP

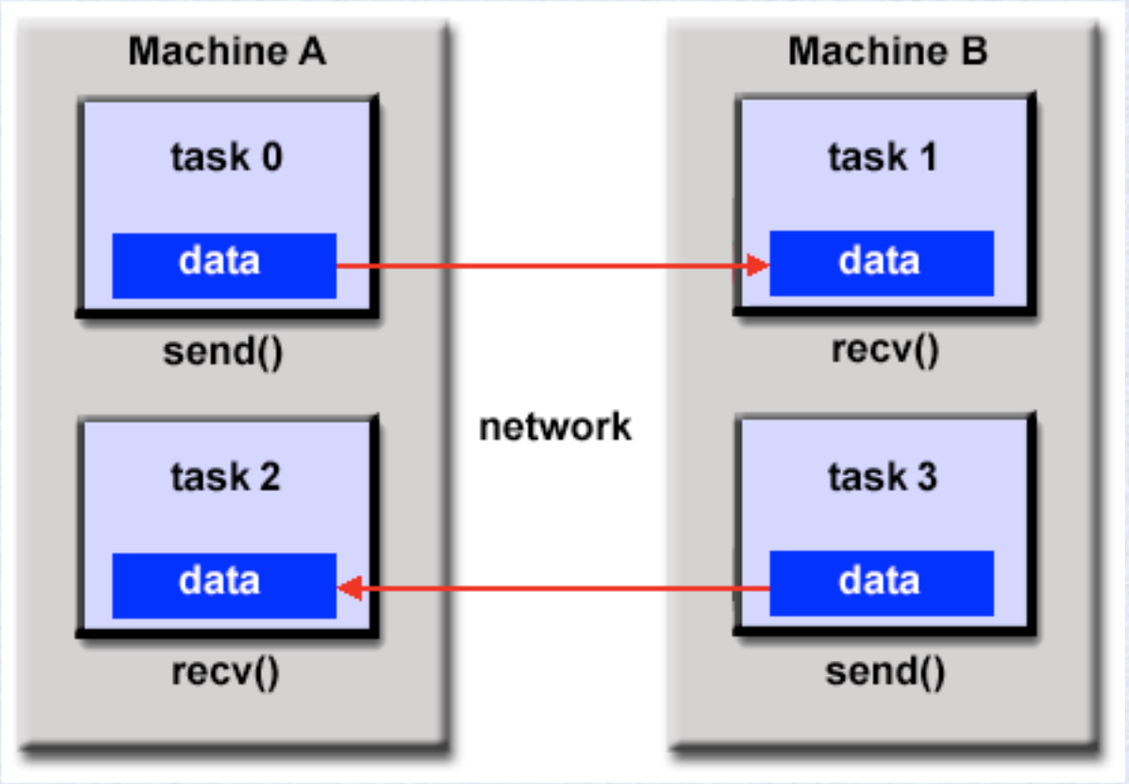
- Compiler directive based; can use serial code
- Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.
- Portable / multi-platform, including Unix and Windows NT platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism"

OpenMP tutorial: computing.llnl.gov/tutorials/openMP

3. DISTRIBUTED MEMORY/ MESSAGE PASSING

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/ MESSAGE PASSING

Implementations:

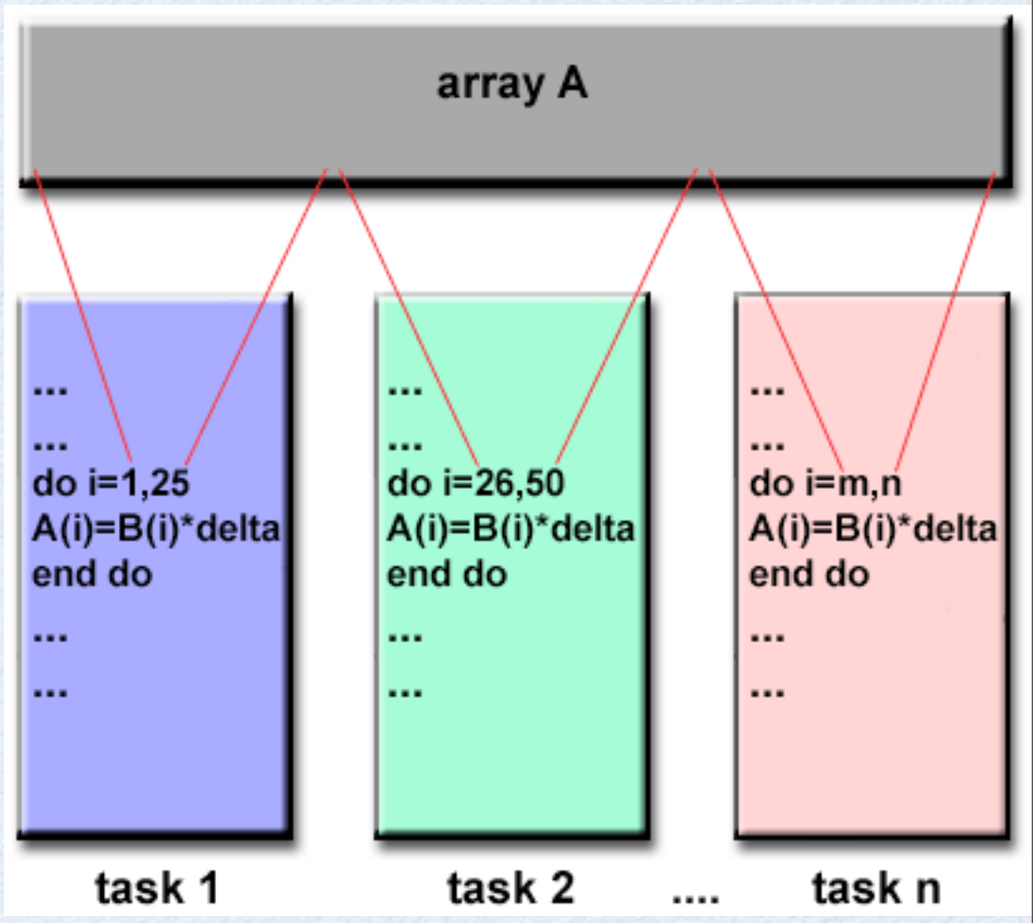
- From a programming perspective, message passing implementations usually comprise a library of subroutines. Calls to these subroutines are imbedded in source code. The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available 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. Both MPI specifications are available on the web at http://www-unix.mcs.anl.gov/mpi/.
- MPI is now 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 both MPI1 and MPI2.

More Information:

- MPI tutorial: computing.llnl.gov/tutorials/mpi

4. DATA PARALLEL MODEL -> GPUs

- This model has the following characteristics:
  - 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 data structure is split up and resides as "chunks" in the local memory of each task.

DATA PARALLEL MODEL

Implementations:

- Programming with the data parallel model is usually accomplished by writing a program with data parallel constructs. The constructs can be calls to a data parallel subroutine library or, compiler directives recognized by a data parallel compiler.
- **Fortran 90 and 95 (F90, F95):** ISO/ANSI standard extensions to Fortran 77.
  - Contains everything that is in Fortran 77
  - New source code format; additions to character set
  - Additions to program structure and commands
  - Variable additions - methods and arguments
  - Pointers and dynamic memory allocation added
  - Array processing (arrays treated as objects) added
  - Recursive and new intrinsic functions added
  - Many other new features
- Implementations are available for most common parallel platforms.

- **High Performance Fortran (HPF):** Extensions to Fortran 90 to support data parallel programming.
  - Contains everything in Fortran 90
  - Directives to tell compiler how to distribute data added
  - Assertions that can improve optimization of generated code added
  - Data parallel constructs added (now part of Fortran 95)
- HPF compilers were relatively common in the 1990s, but are no longer commonly implemented.

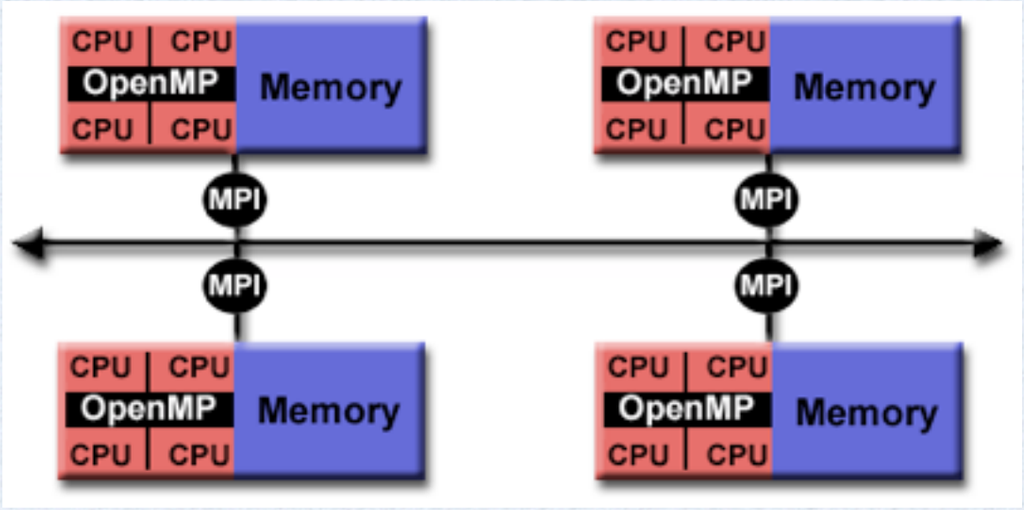
- **Compiler Directives:** Allow the programmer to specify the distribution and alignment of data. Fortran implementations are available for most common parallel platforms.
- Distributed memory implementations of this model usually require the compiler to produce object code with calls to a message passing library (MPI) for data distribution. All message passing is done invisibly to the programmer.

5. HYBRID MODEL

- **A hybrid model combines more than one of the previously described programming models.**

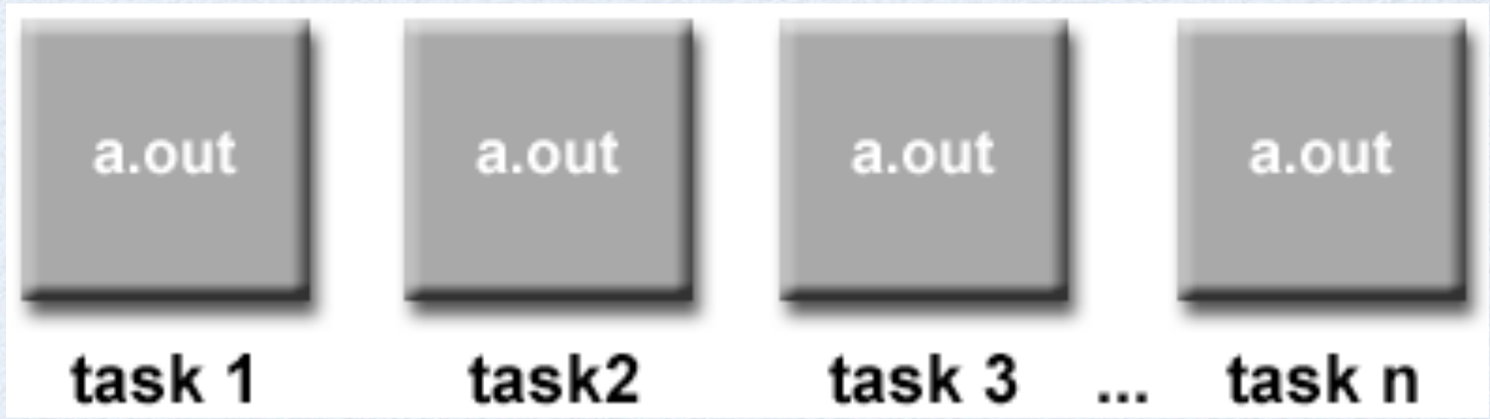
- Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with the threads model (OpenMP).
  - Threads perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI

- This hybrid model lends itself well to the increasingly common hardware environment of clustered multi/many-core machines.
- Another similar and increasingly popular example of a hybrid model is using MPI with GPU (Graphics Processing Unit) programming.
  - GPUs perform computationally intensive kernels using local, on-node data
  - Communications between processes on different nodes occurs over the network using MPI



6.SPMD

Single Program Multiple Data (SPMD):

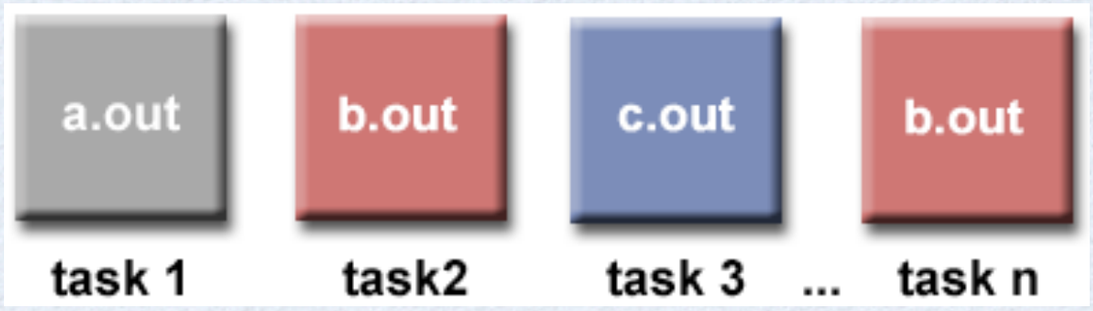


- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.

- 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.

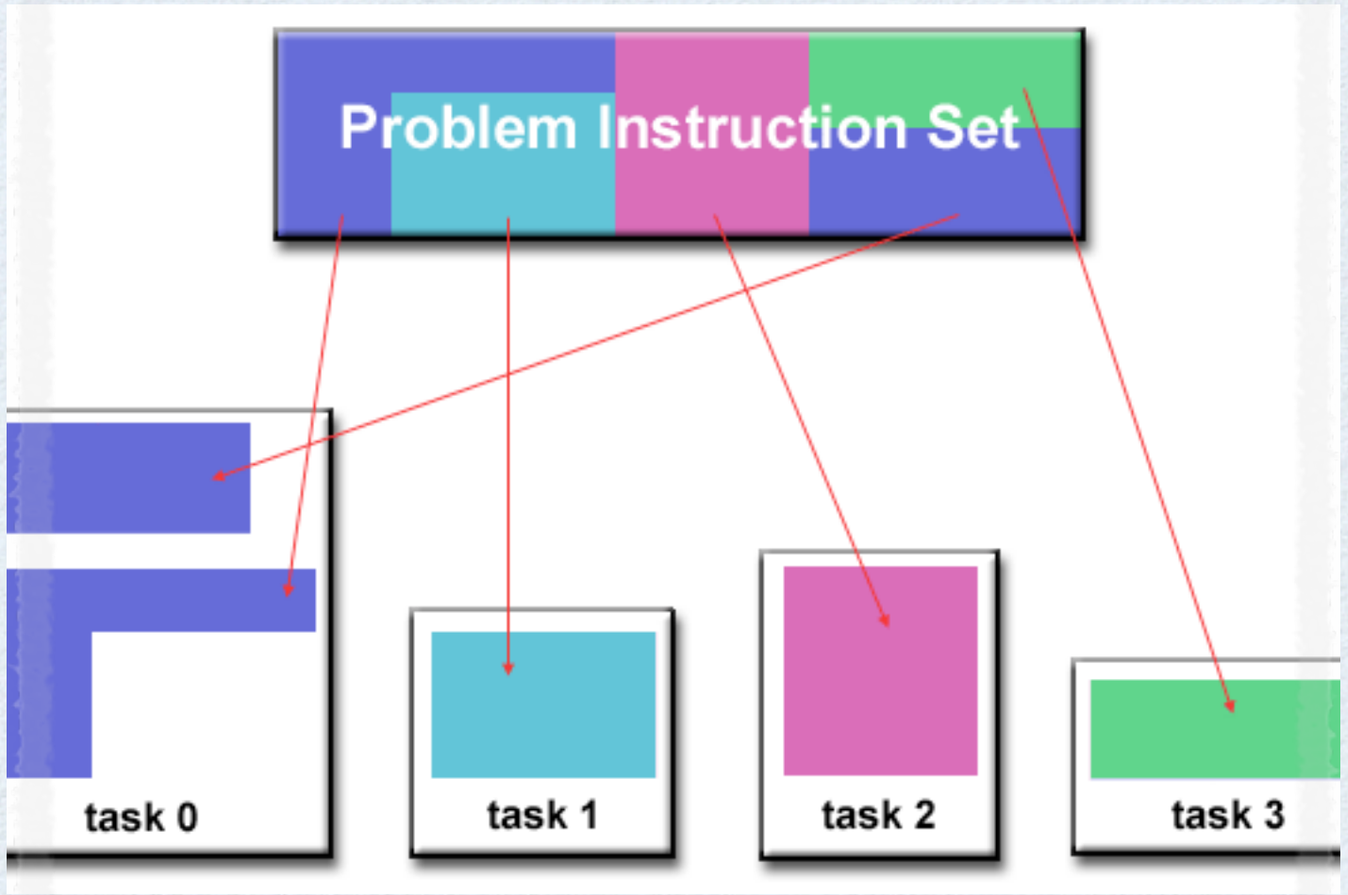
7.MPMD

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 Partioning).



Designing Parallel Programs

Automatic vs. Manual Parallelization

- 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) loops are the most frequent target for automatic parallelization.
  - **Programmer Directed**
    - Using "compiler directives" or 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.
- 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 analysis suggests there are inhibitors or the code is too complex

THE REST OF THE NOTES REFER TO MANUAL PARALLELIZATION

Parallel Program Design : Understand the Problem I

- The first step in developing parallel software is to first understand the problem that you wish to solve in parallel.
- Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.

  - Example of Parallelizable Problem:
  - **Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.**
  - This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

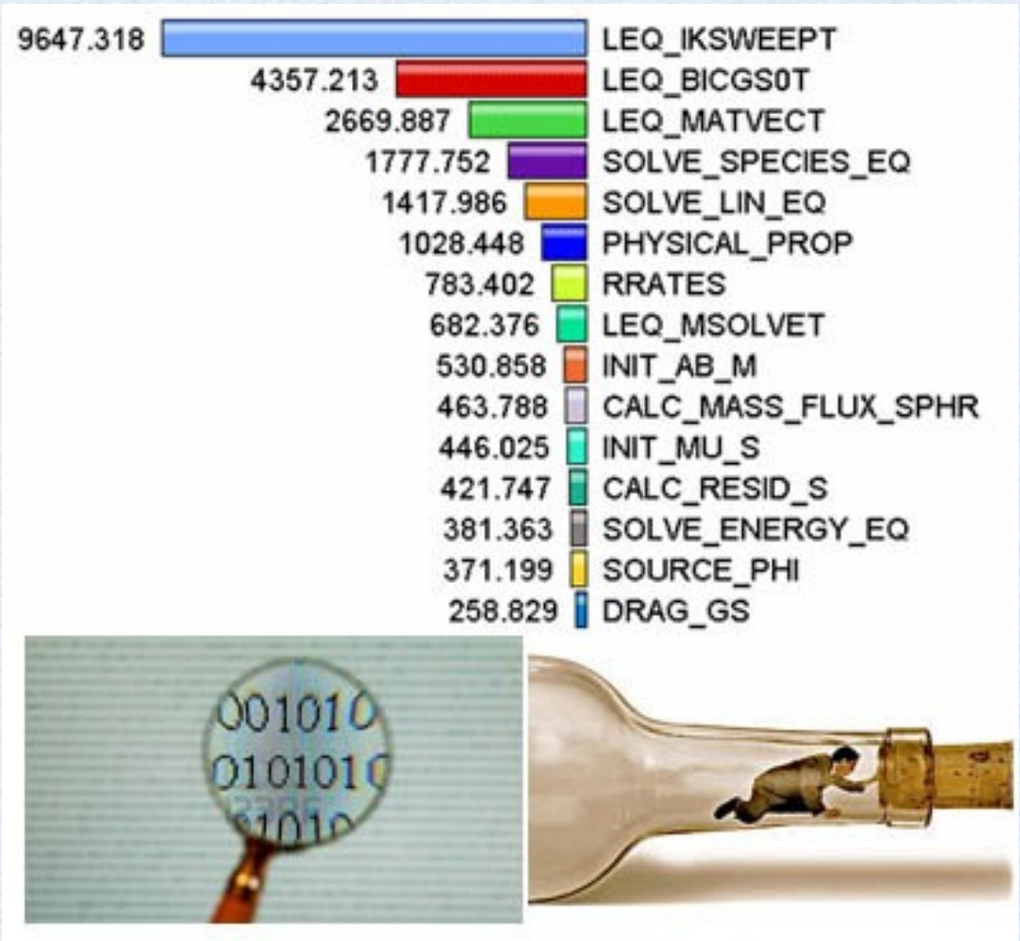
  - Example of a Non-parallelizable Problem:
  - **Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula: F(n) = F(n-1) + F(n-2)**
  - This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the F(n) value uses those of both F(n-1) and F(n-2). These three terms cannot be calculated independently and therefore, not in parallel.

Parallel Program Design : Understand the Problem II

- Identify the program's ***hotspots***:
  - Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
  - Profilers and performance analysis tools can help here
  - Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

- Identify ***bottlenecks*** in the program
  - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
  - May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

- Identify inhibitors to parallelism. One common class of inhibitor is ***data dependence,*** as demonstrated by the Fibonacci sequence above.
- Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

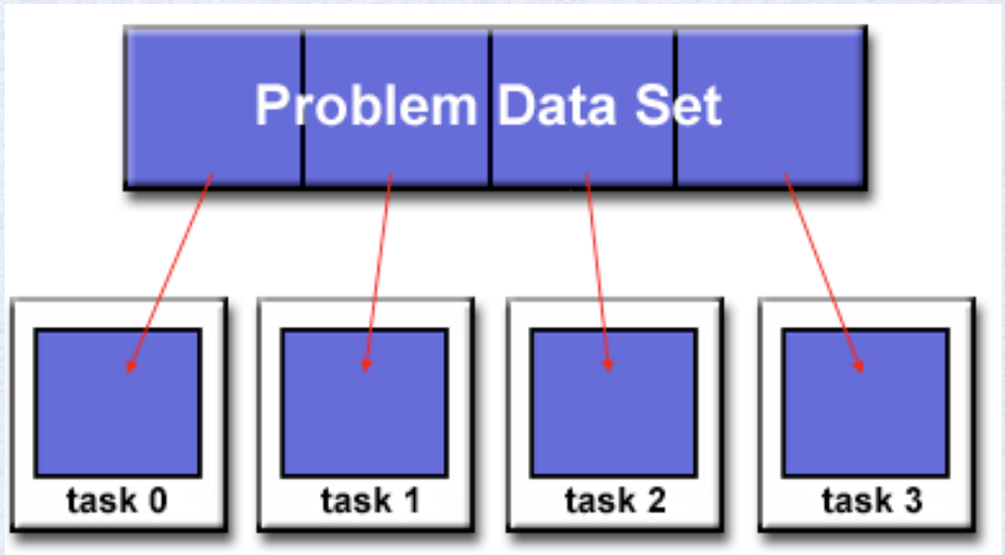


Decomposition/Partitioning

- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as **decomposition or partitioning.**
- Two basic ways to partition work among parallel tasks: ***domain decomposition*** and ***functional decomposition***.

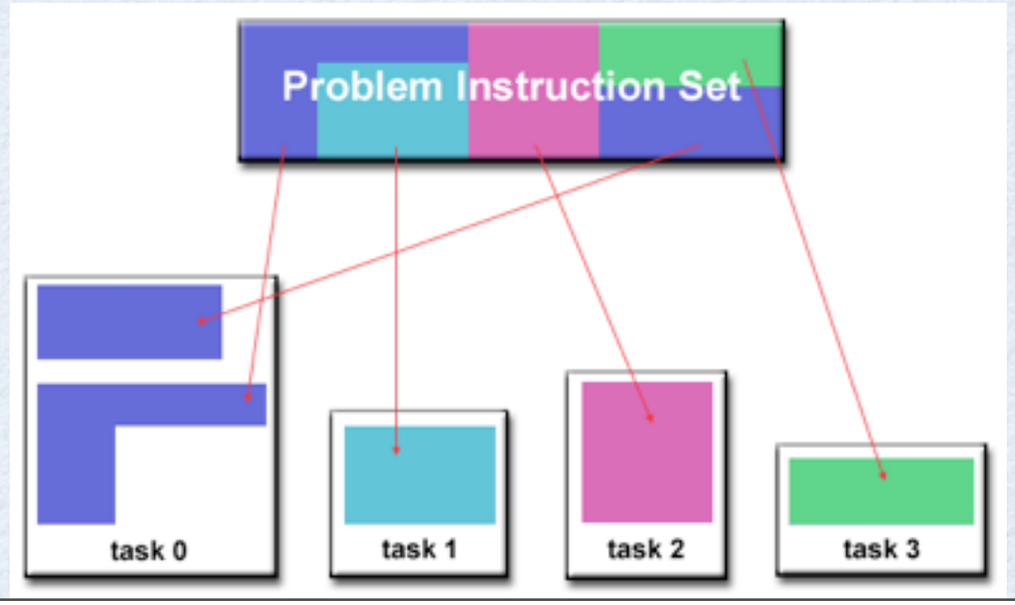
Domain Decomposition:

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.



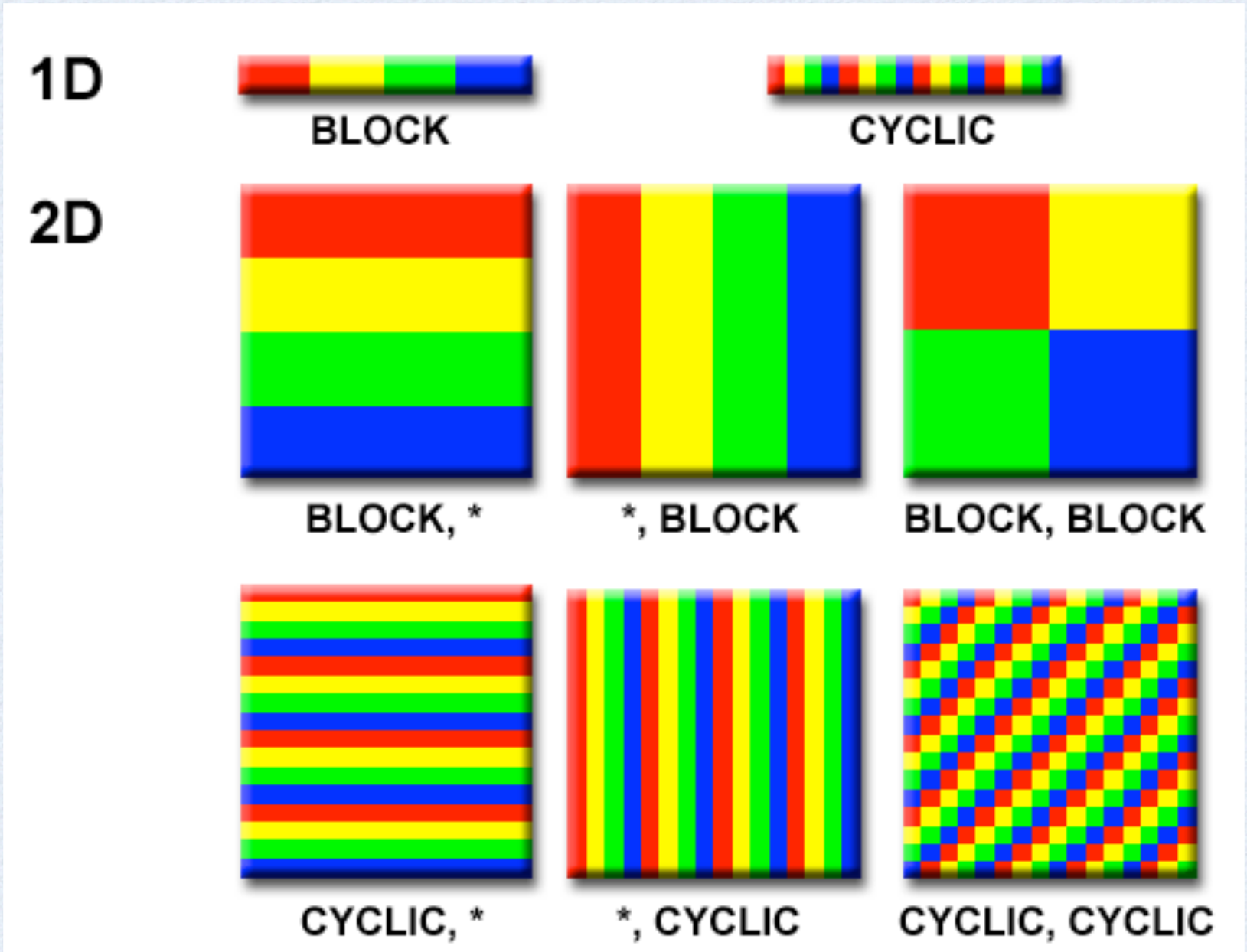
Functional Decomposition:

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.



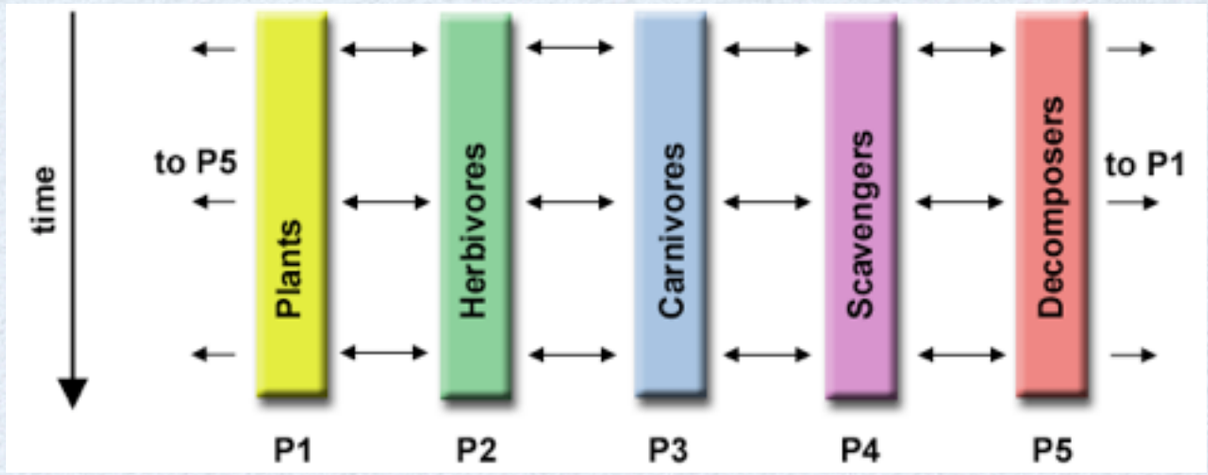
Domain Decomposition

There are different ways to partition data:

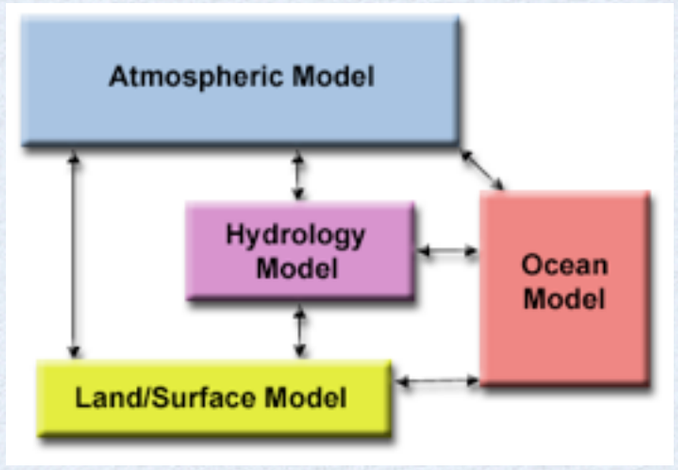


Functional Decomposition

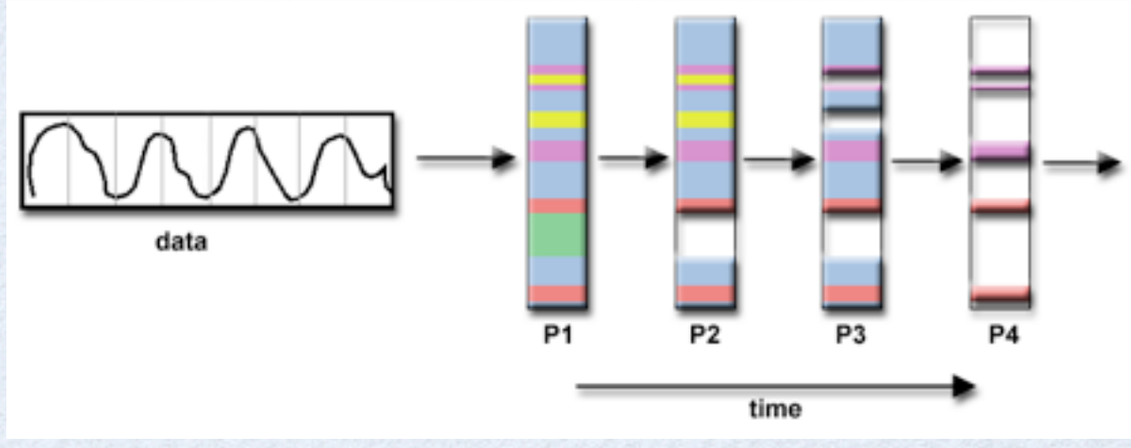
Ecosystem Modeling
Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time step.



Climate Modeling
Each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.



Signal Processing
An audio signal data set is passed through four distinct computational filters. Each filter is a separate process. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy.



COMMUNICATION

Who Needs Communications?

- The need for communications between tasks depends upon your problem:

- **You DON'T need communications**
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. For example, imagine an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.
  - These types of problems are often called ***embarrassingly parallel*** because they are so straight-forward. Very little inter-task communication is required.

- **You DO need communications**
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other. For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.

COMMUNICATION – FACTORS I

- **Cost of communications**
  - Inter-task communication virtually always implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
  - Competing communication traffic can saturate the available network bandwidth, aggravating performance

- **Latency vs. Bandwidth**
  - ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Expressed as microseconds.
  - ***bandwidth*** is the amount of data that can be communicated per unit of time. Expressed as Mb/sec or Gb/sec.
  - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

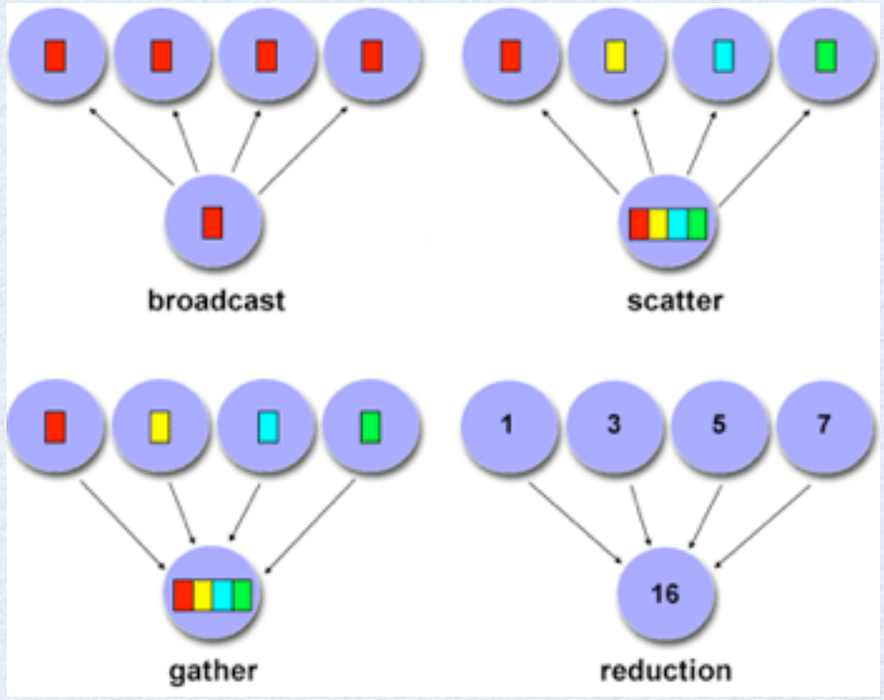
- **Visibility of communications**
  - With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
  - With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not be able to know how inter-task communications are accomplished.

- **Synchronous vs. asynchronous communications**
  - Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
  - Synchronous communications are often referred to as ***blocking*** as other work must wait until they have completed.
  - Asynchronous communications allow tasks to transfer data independently from one another. E.g. task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 receives the data doesn't matter.
  - Asynchronous communications are often referred to as ***non-blocking*** communications since other work can be done while the communications are taking place.
  - Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

COMMUNICATION – FACTORS II

Scope of communications

- Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scopings described below can be implemented synchronously or asynchronously.
- ***Point-to-point*** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
- ***Collective*** - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variations (there are more):



Efficiency of communications

- Very often, the programmer will have a choice with regard to factors that can affect communications performance. Only a few are mentioned here.
- Which implementation for a given model should be used? Using the Message Passing Model as an example, one MPI implementation may be faster on a given hardware platform than another.
- What type of communication operations should be used? As mentioned previously, asynchronous communication operations can improve overall program performance.
- Network media - some platforms may offer more than one network for communications. Which one is best?

this is only a partial list of things to consider!!!

SYNCHRONIZATION

Types of Synchronization:

- **Barrier**
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized.
  - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.

- **Lock / semaphore**
  - Can involve any number of tasks
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking

- **Synchronous communication operations**
  - Involves only those tasks executing a communication operation
  - When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication. For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.
  - Discussed previously in the Communications section.

DATA DEPENDENCIES

A ***dependence*** exists between program statements when the order of statement execution affects the results of the program.
A ***data dependence*** results from multiple use of the same location(s) in storage by different tasks.
Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.

EXAMPLE : Loop carried data dependence

```
DO 500 J = MYSTART,MYEND
   A(J) = A(J-1) * 2.0
500 CONTINUE
```

The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.

If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:

- Distributed memory architecture - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation
- Shared memory architecture - task 2 must read A(J-1) after task 1 updates it

EXAMPLE : Loop Independent data dependence

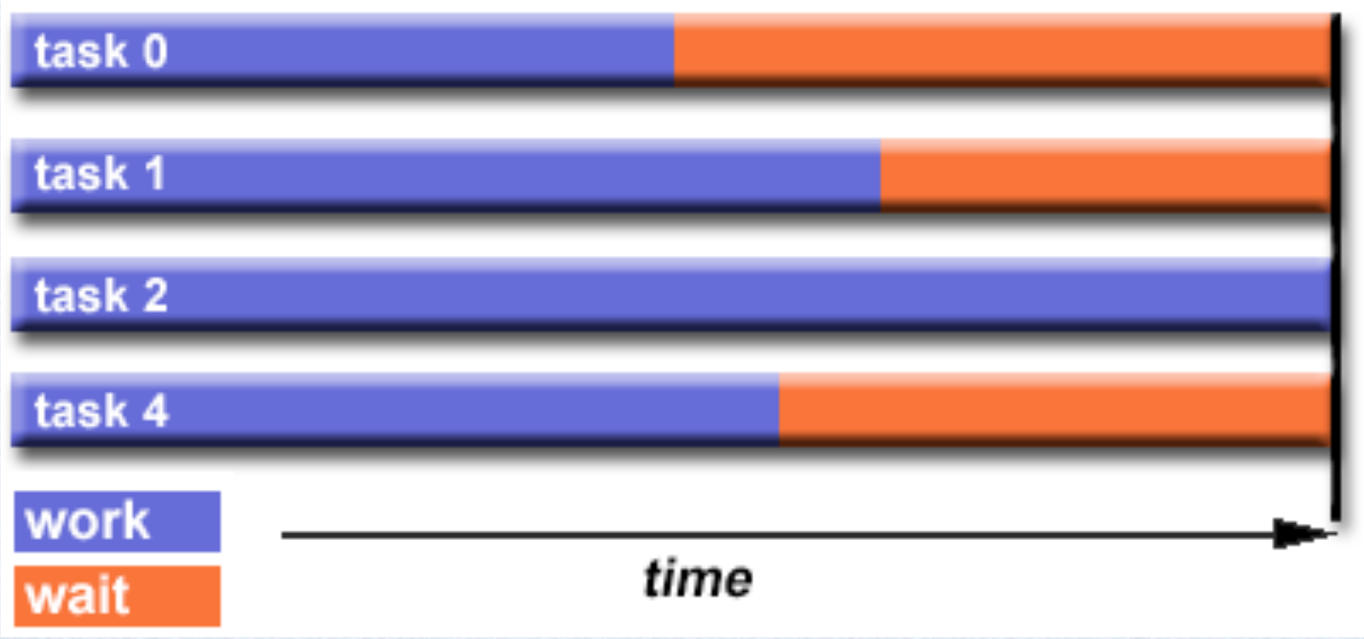
```
task 1          task 2
------          ------

X = 2           X = 4
  .               .
  .               .
Y = X**2        Y = X**3
```

- As with the previous example, parallelism is inhibited. The value of Y is dependent on:
  - Distributed memory architecture - if or when the value of X is communicated between the tasks.
  - Shared memory architecture - which task last stores the value of X.
- Although all data dependencies are important to identify when designing parallel programs, loop carried dependencies are particularly important since loops are possibly the most common target of parallelization efforts.

How to Handle Data Dependencies:

- Distributed memory architectures - communicate required data at synchronization points.
- Shared memory architectures -synchronize read/write operations between tasks.

LOAD BALANCING



- Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time. It can be considered a minimization of task idle time.

- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.

(how to achieve) LOAD BALANCING

- **Equally partition the work each task receives**
  - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
  - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  - If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances.

- **Use dynamic work assignment**
  - Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
    - Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".
    - Adaptive grid methods - some tasks may need to refine their mesh while others don't.
    - *N*-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.
  - When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get a new piece of work.
  - It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

GRANULARITY

Computation / Communication Ratio:

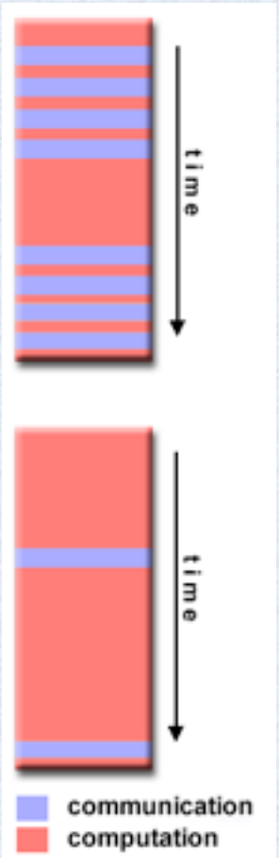
- In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
- Periods of computation are typically separated from periods of communication by synchronization events.

Fine-grain Parallelism:

- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

Coarse-grain Parallelism:

- Relatively large amounts of computational work are done between communication/ synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently



Which is Best?

- The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.

I/O

The Bad News:

- I/O operations are generally regarded as inhibitors to parallelism
- Parallel I/O systems may be immature or not available for all platforms
- In an environment where all tasks see the same file space, write operations can result in file overwriting
- Read operations can be affected by the file server's ability to handle multiple read requests at the same time
- I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks and even crash file servers.

The Good News:

- Parallel file systems are available. For example:
  - GPFS: General Parallel File System for AIX (IBM)
  - Lustre: for Linux clusters (Oracle)
  - PVFS/PVFS2: Parallel Virtual File System for Linux clusters (Clemson/Argonne/Ohio State/others)
  - PanFS: Panasas ActiveScale File System for Linux clusters (Panasas, Inc.)
  - HP SFS: HP StorageWorks Scalable File Share. Lustre based parallel file system (Global File System for Linux) product from HP

- The parallel I/O programming interface specification for MPI has been available since 1996 as part of MPI-2. Vendor and "free" implementations are now commonly available.

I/O Pointers

A few pointers:

- Rule #1: Reduce overall I/O as much as possible
- If you have access to a parallel file system, investigate using it.
- Writing large chunks of data rather than small packets is usually significantly more efficient.
- Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks. For example, Task 1 could read an input file and then communicate required data to other tasks. Likewise, Task 1 could perform write operation after receiving required data from all other tasks.
- Use local, on-node file space for I/O if possible. For example, each node may have /tmp filespace which can used. This is usually much more efficient than performing I/O over the network to one's home directory.

LIMITS AND COSTS

Amdahl's Law:

- **Amdahl's Law** states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

$$\text{speedup} = \frac{1}{1 - P}$$

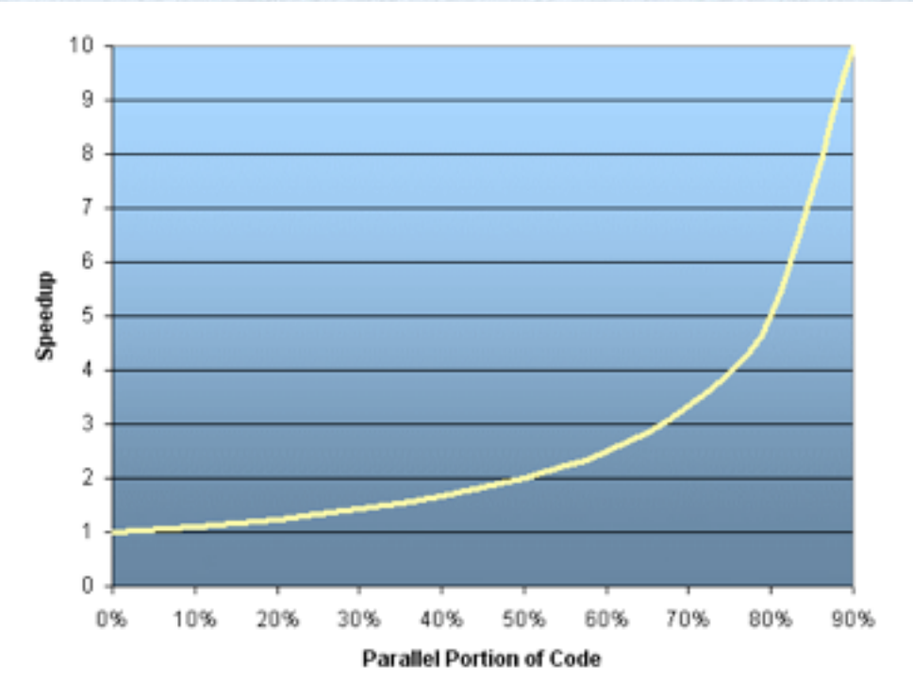
- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
- If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.
- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

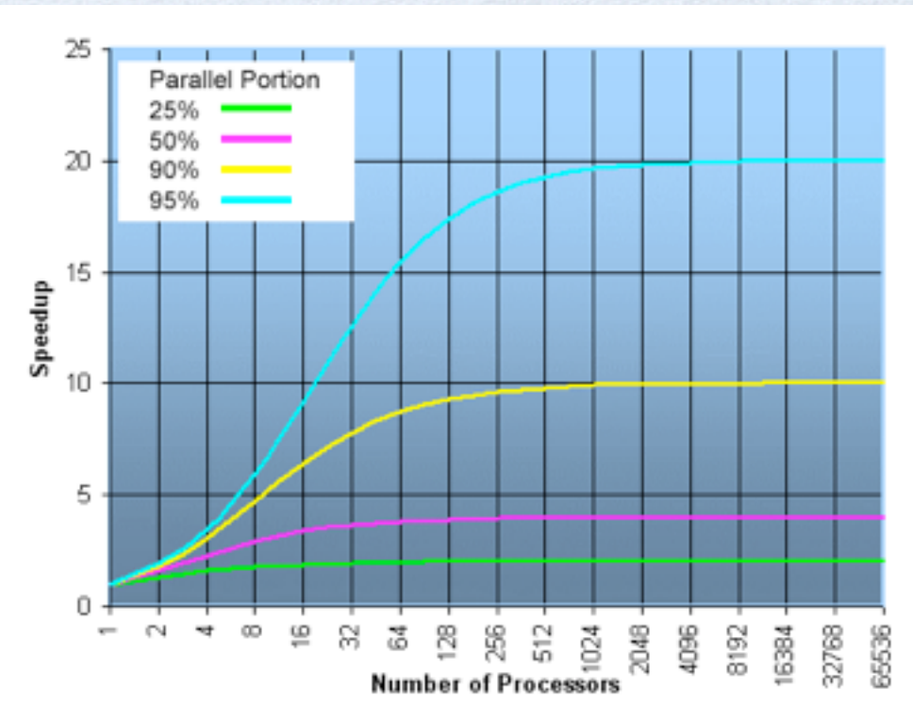
$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

where P = parallel fraction, N = number of processors and S = 1-P = serial fraction.

- It soon becomes obvious that there are limits to the scalability of parallelism. For example:

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02
100000	1.99	9.99	99.90





SCALABILITY

However, certain problems demonstrate **increased performance by increasing the problem size.** For example:

- **2D Grid Calculations 85 seconds 85%**
- **Serial fraction 15 seconds 15%**

We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:

- **2D Grid Calculations 680 seconds 97.84%**
- **Serial fraction 15 seconds 2.16%**

Problems that increase the percentage of parallel time with their size are more ***scalable*** than problems with a fixed percentage of parallel time.

Scalability:

- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more machines is rarely the answer.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. Most parallel solutions demonstrate this characteristic at some point.
- Hardware factors play a significant role in scalability. Examples:
  - Memory-cpu bus bandwidth on an SMP machine
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed

- Parallel support libraries and subsystems software can limit scalability independent of your application.