# Advanced Scientific Computing

Fall 2010

# Topics

Fundamentals:

- Parallel computers (application oriented view)
- Parallel and distributed numerical computation
  - MPI: message-passing library specification
  - PETSc: Portable, Extensible Toolkit for Scientific Computation
- Numerical iterative techniques for solving large sparse systems
- Software design, analysis, implementation, performance evaluation, …

Details:

☺Your course project:

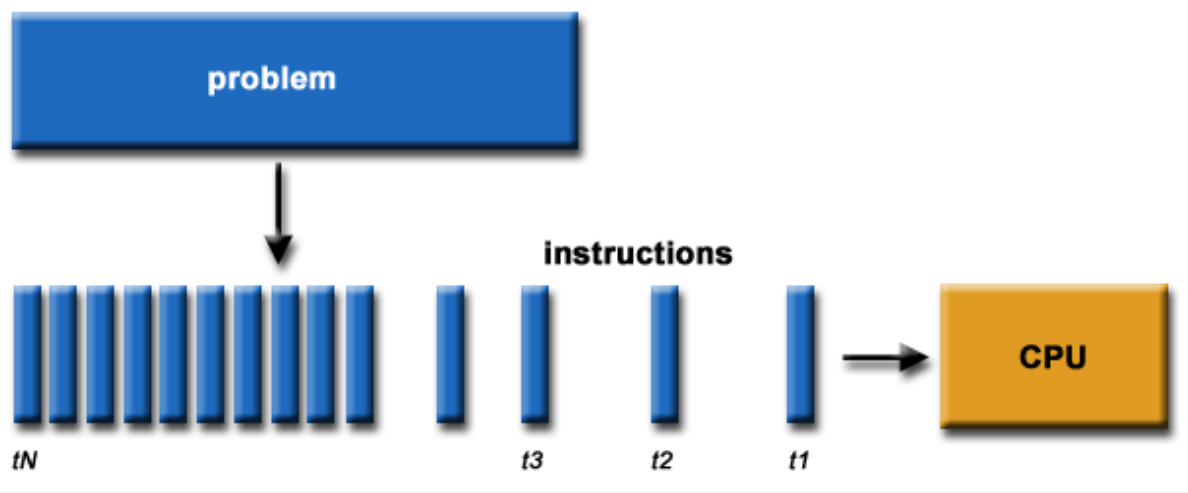
discussion, programming, documentation, presentation, and more…

# Overview of Parallel Computing

# What is Parallel Computing?

**Serial Computations**

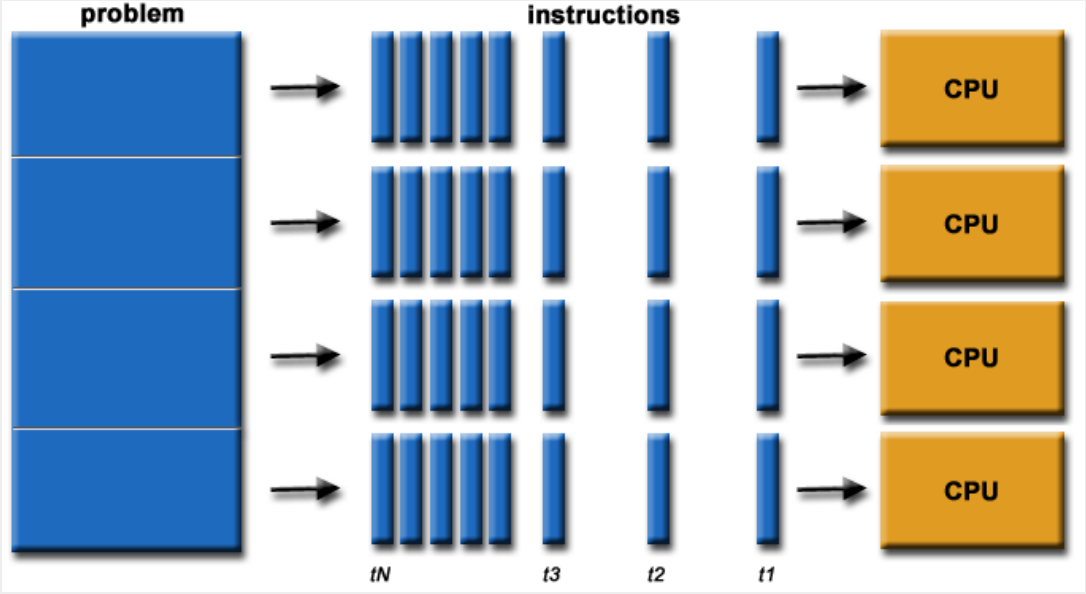
- Traditionally, software has been written for ***serial*** computation:
  - To be run on a single computer having a single Central Processing Unit (CPU);
  - A problem is broken into a discrete series of instructions.
  - Instructions are executed one after another.
  - Only one instruction may execute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



# What is Parallel Computing?

**Parallel Computations**

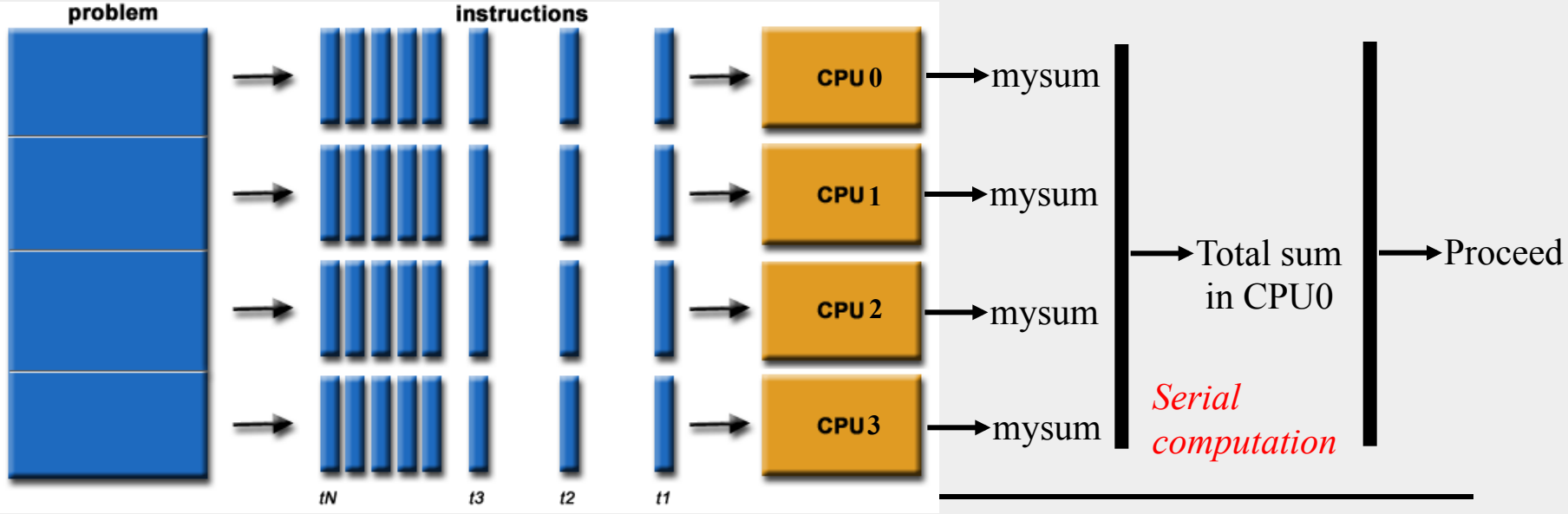
- In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:
  - To be run using multiple CPUs
  - A problem is broken into discrete parts that can be solved concurrently
  - Each part is further broken down to a series of instructions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



# Simple Example – summation of numbers

```
sum = 0
do i = 1,1000
   sum = sum + a(i)
enddo
```

```
mysum = 0
do i = mystart,myend
   mysum = mysum + a(i)
enddo
If(CPU 0)then
   do i = 1,4
      sum = sum + mysum
   enddo
endif
```



# Requirements

- The compute resources can include:
  - A single computer with multiple processors;
  - An arbitrary number of computers connected by a network;
  - A combination of both.
- The computational problems can be parallelized:
  - Broken apart into discrete pieces of work that can be solved simultaneously;
  - …
- Some problems are not non-parallelizable
  - calculation of the Fibonacci series

    (1,1,2,3,5,8,13,21,...) by F(n) = F(n-1) + F(n-2)

# Parallel Computers

- ~25 years ago
  - $1*10^6$ Floating Point Ops/sec (Mflop/s)
    - Scalar based
- 1993
  - $58.7*10^9$ Floating Point Ops/sec (Gflop/s)
    - Vector & shared memory computing
- June 2008
  - 478.2 TFlop/s ($478.2*10^{12}$ Floating Point Ops/sec)
  - Highly parallel, distributed processing, message passing, network based
  - Data decomposition, communication/computation
- June 2010
  - 1.759 Pflop/s ($1.759*10^{15}$ Floating Point Ops/sec )
  - Many more levels memory hierarchy, combination/grids &HPC
  - More adaptive, Latency Tolerant and Bandwidth aware, fault tolerant, extended precision, …

## TOP 500 Machines: http://www.top500.org

**Peak performance benchmark is not enough ☹**

Blue Waters (2011):
Enabling research and education with sustained petascale computing

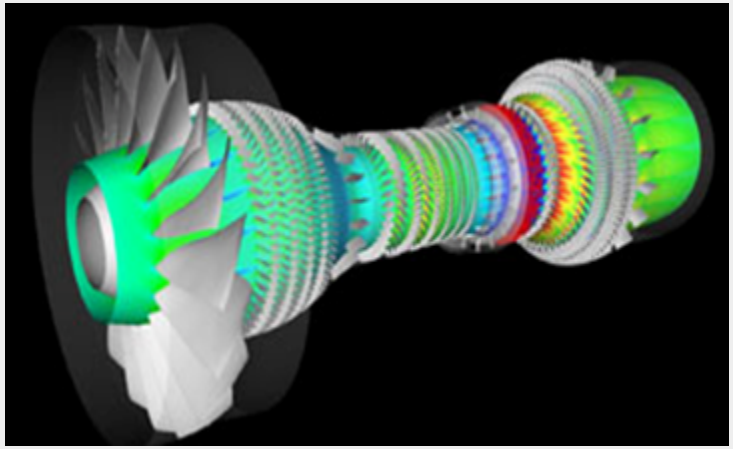
- Blue Waters is expected to be one of the fastest supercomputers in the world available for open scientific research when it comes online in 2011 at the University of Illinois. An 8 year project with an overall cost of over $500M, it will be the first system of its kind to sustain one petaflop performance on a range of science and engineering applications.

- a joint effort of UIUC, NCSA, IBM, and the Great Lakes Consortium

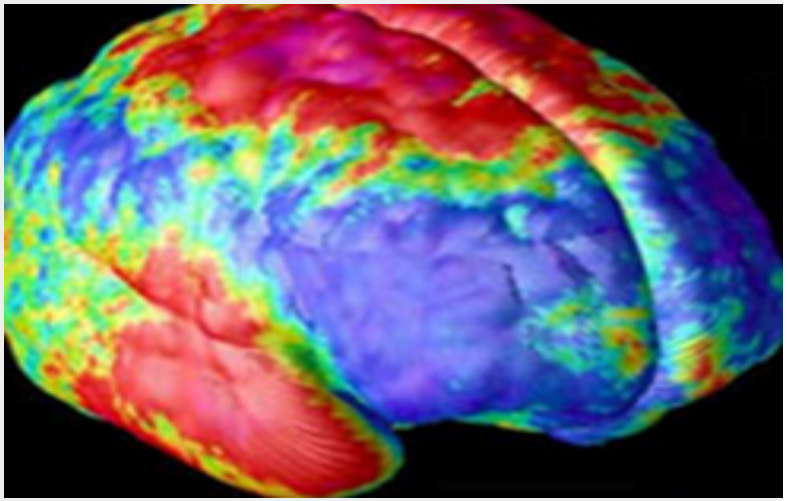
**Sustained petaflop performance on applications!**

# Applications (Science and Engineering)

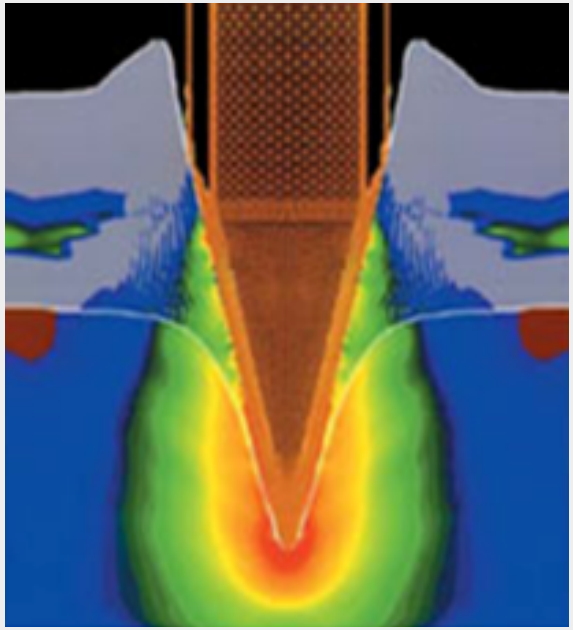
- Historically used for large scale problems in Science and Engineering
- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Geology, Seismology
- Mechanical Engineering - from prosthetics to spacecraft
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics



Turbo machinery (Gas turbine/compressor)

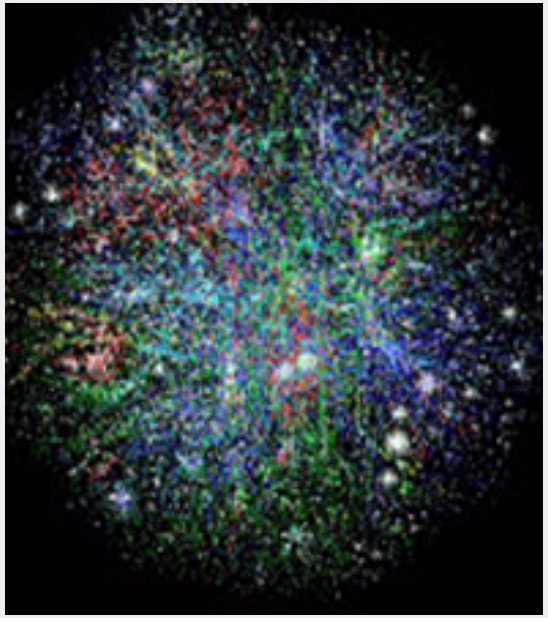


Biology application



Drilling application

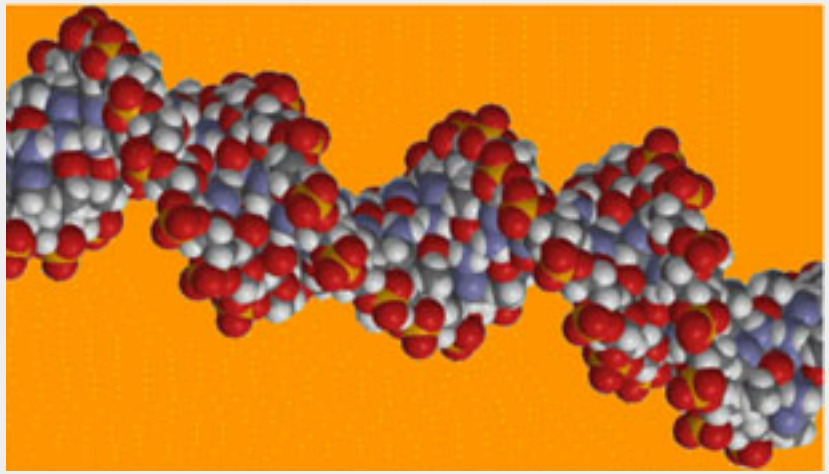

Transportation & traffic application



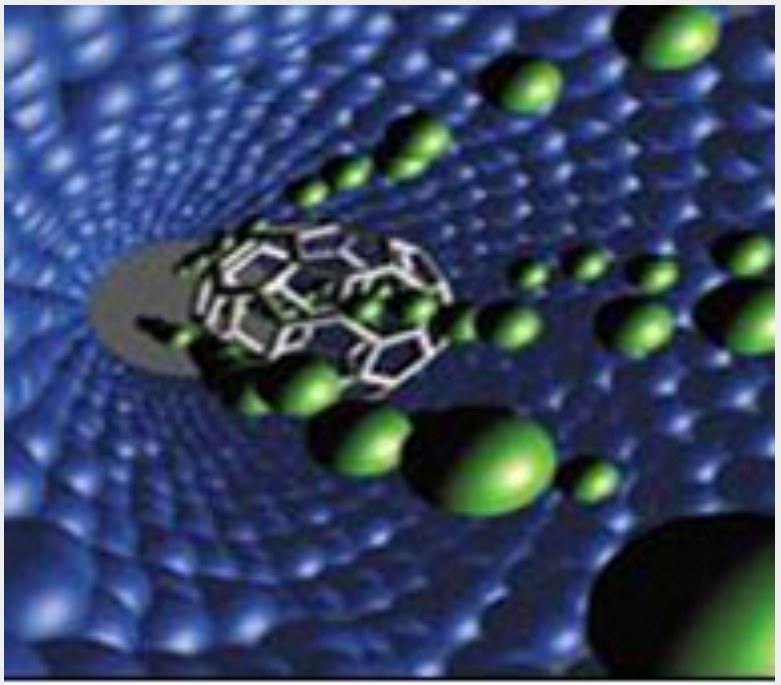
Astrophysics application

# Other applications (industry driven)

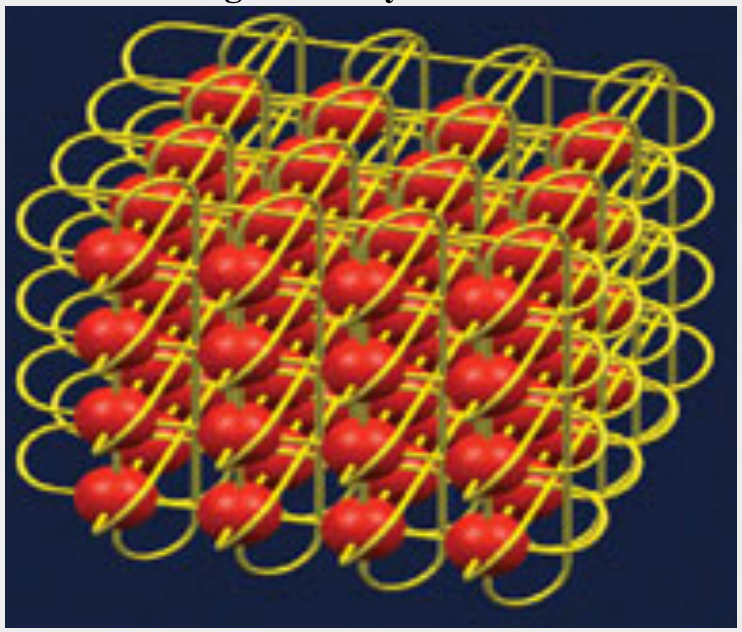
- Commercial applications also provide a driving force in the parallel computing. These applications require the processing of large amounts of data
- Databases, data mining
- Oil exploration
- Web search engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Management of national and multi-national corporations
- Financial and economic modeling
- Advanced graphics and virtual reality, particularly in the entertainment industry
- Networked video and multi-media technologies



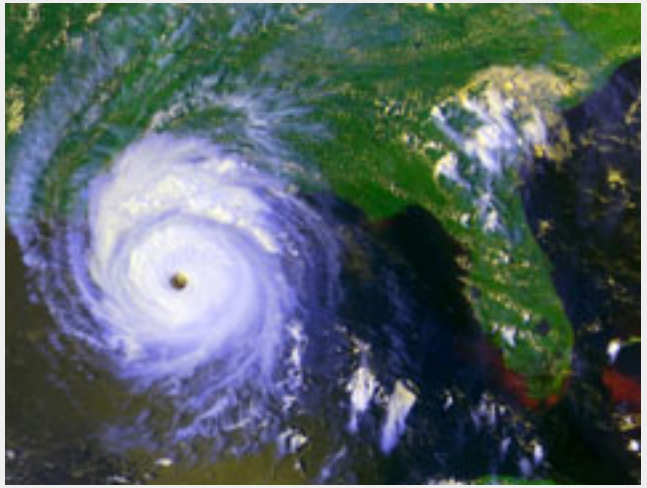
Drug discovery



Advanced Graphics



New materials



Weather modeling

# Goal of Parallel Computing

- Solve larger problems faster
- Often larger is more important than faster
- P-fold speedups not as important!

## Challenge of Parallel Computing:

Coordinate, control, and monitor the computation

# Aspects of Parallel Computing

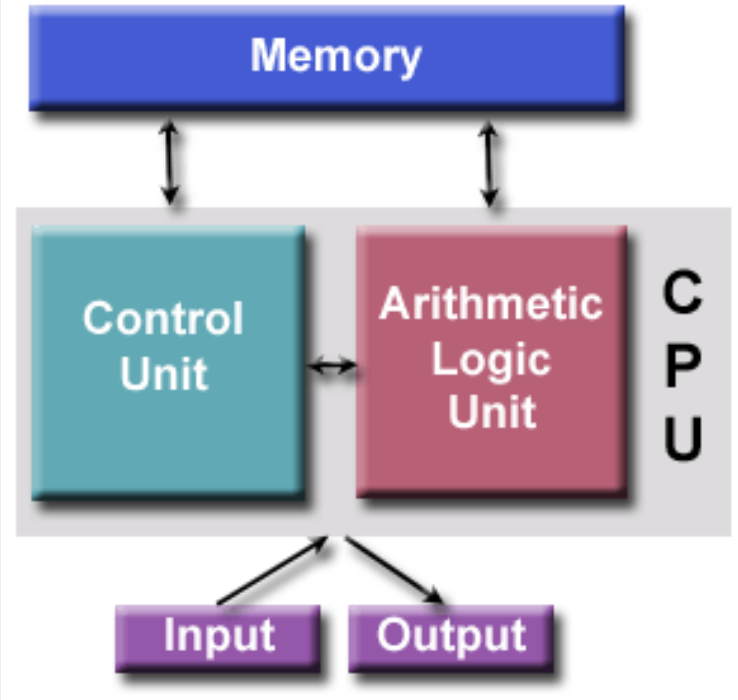
- Architectures:
  - Processors and memories connected together
  - Memory hierarchy
- Software:
  - Operating systems, compilers
  - Libraries
  - Tools – debuggers, performance analysis
- Algorithms and Programming:
  - Solve large scale problems on parallel computers

# Architecture

- Basic components of any architecture:
  - Processors and memory (processing units)
  - Interconnect
- Logic classification based on:
  - Control mechanism (Flynn's Taxonomy)
    - SISD (Single Instruction Single Datastream)
    - SIMD (Single Instruction Multiple Datastream)
    - MISD (Multiple Instruction Single Datastream)
    - MIMD (Multiple Instruction Multiple Datastream)
  - Address space organization
    - Shared Address Space
    - Distributed Address Space

# Concepts and Terminology

- Basic computer comprised of
  - Memory
  - CPU
  - Input/Output
- memory is used to store both program instructions and data
- 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



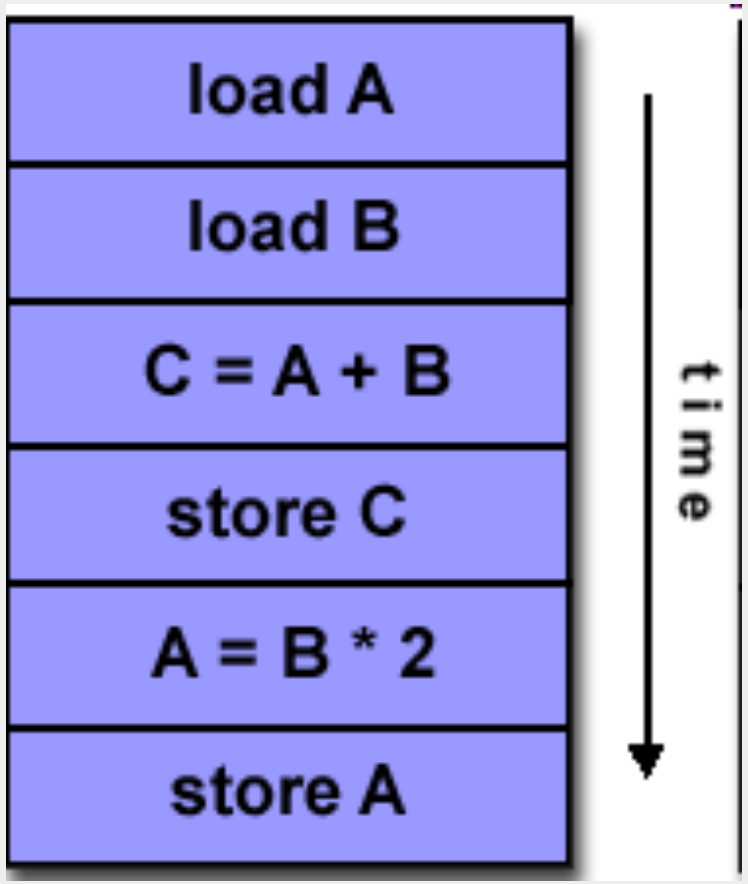
# Classification of Parallel Computers

- distinguishes multi-processor computer architectures according to ***Instruction*** and ***Data*** *(called* ***Flynn's Classical Taxonomy)***.

| | |
|---|---|
| **S I S D**<br>**Single Instruction, Single Data** | **S I M D**<br>**Single Instruction, Multiple Data** |
| **M I S D**<br>**Multiple Instruction, Single Data** | **M I M D**<br>**Multiple Instruction, Multiple Data** |

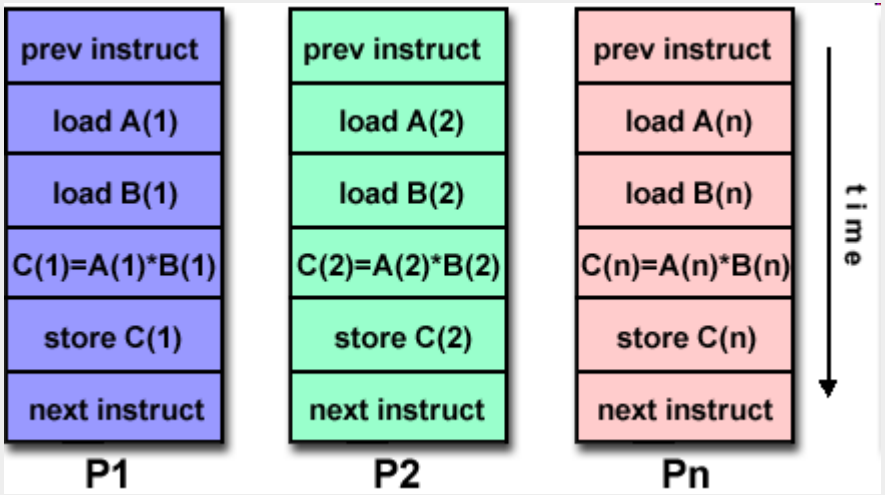
# Single Instruction, Single Data (SISD)

- 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 and even today, the most common type of computer
- Examples: older generation mainframes, minicomputers and workstations; most modern day PCs.



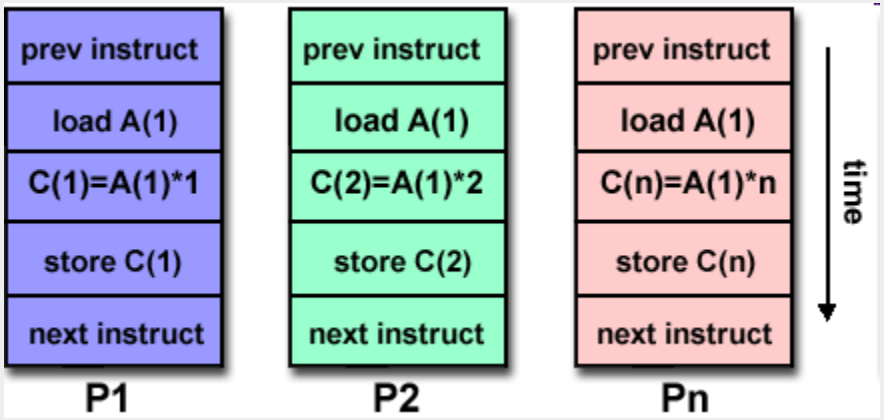
# Single Instruction, Multiple Data (SIMD)

- 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
- Graphics processor units (GPUs) employ SIMD



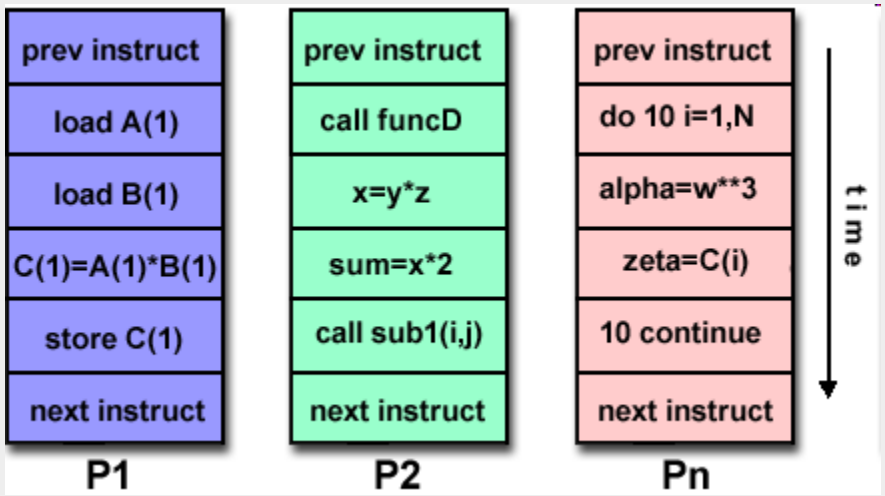
# Multiple Instruction, Single Data (MISD)

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



# Multiple Instruction, Multiple Data (MIMD)

- Most modern computers fall into this category.
- 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
- Examples: most current supercomputers, networked parallel computer clusters, multi-processor SMP computers, multi-core PCs.

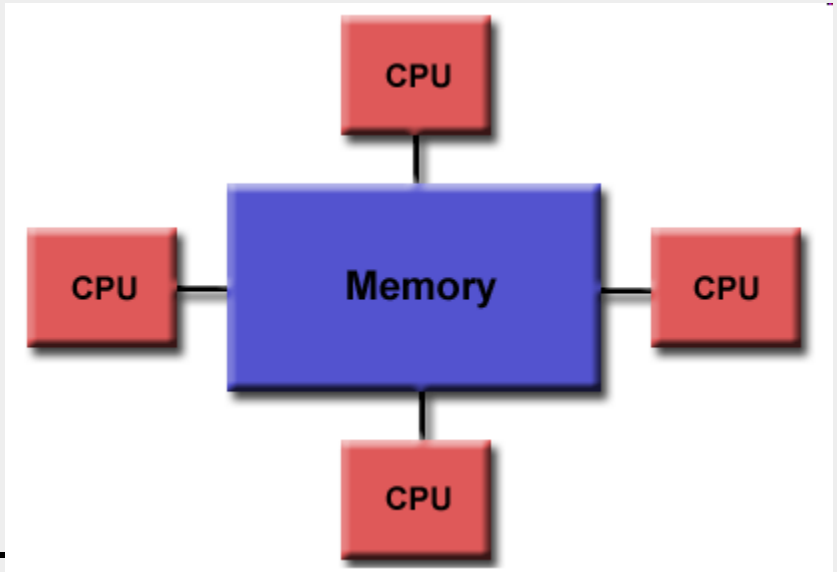


# Common terminology

- **Task**
  - A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor (eg. Loop, function, subroutine etc).
- **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.
- **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.

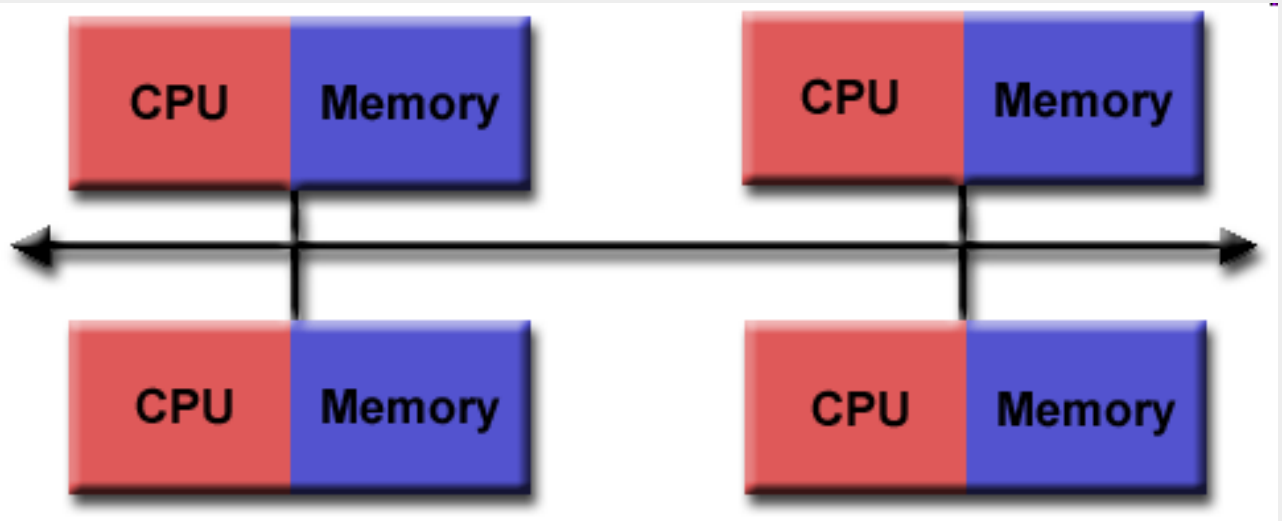
# Common terminology (cont'd)

- **Shared Memory**
  - Hardware sense - computer architecture where all processors have direct (usually bus based) access to common physical memory.
  - Programming sense - 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.



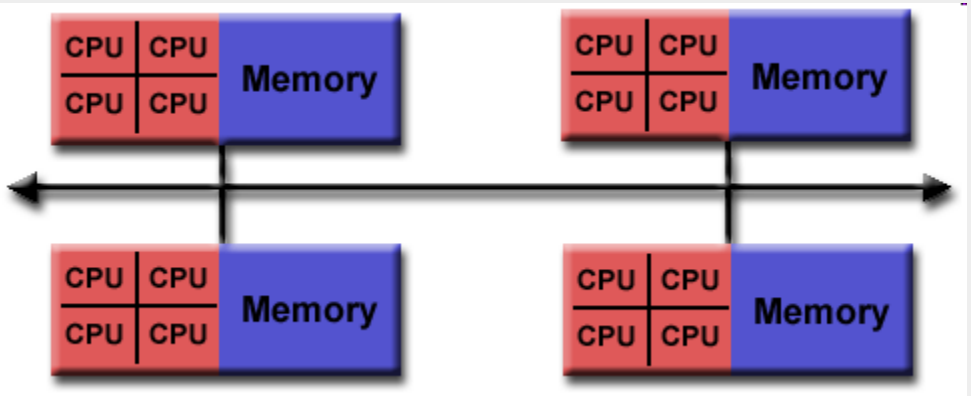
# Common terminology (cont'd)

- **Distributed Memory**
  - Hardware sense - refers to network based memory access for physical memory that is not common.
  - Programming model sense - tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

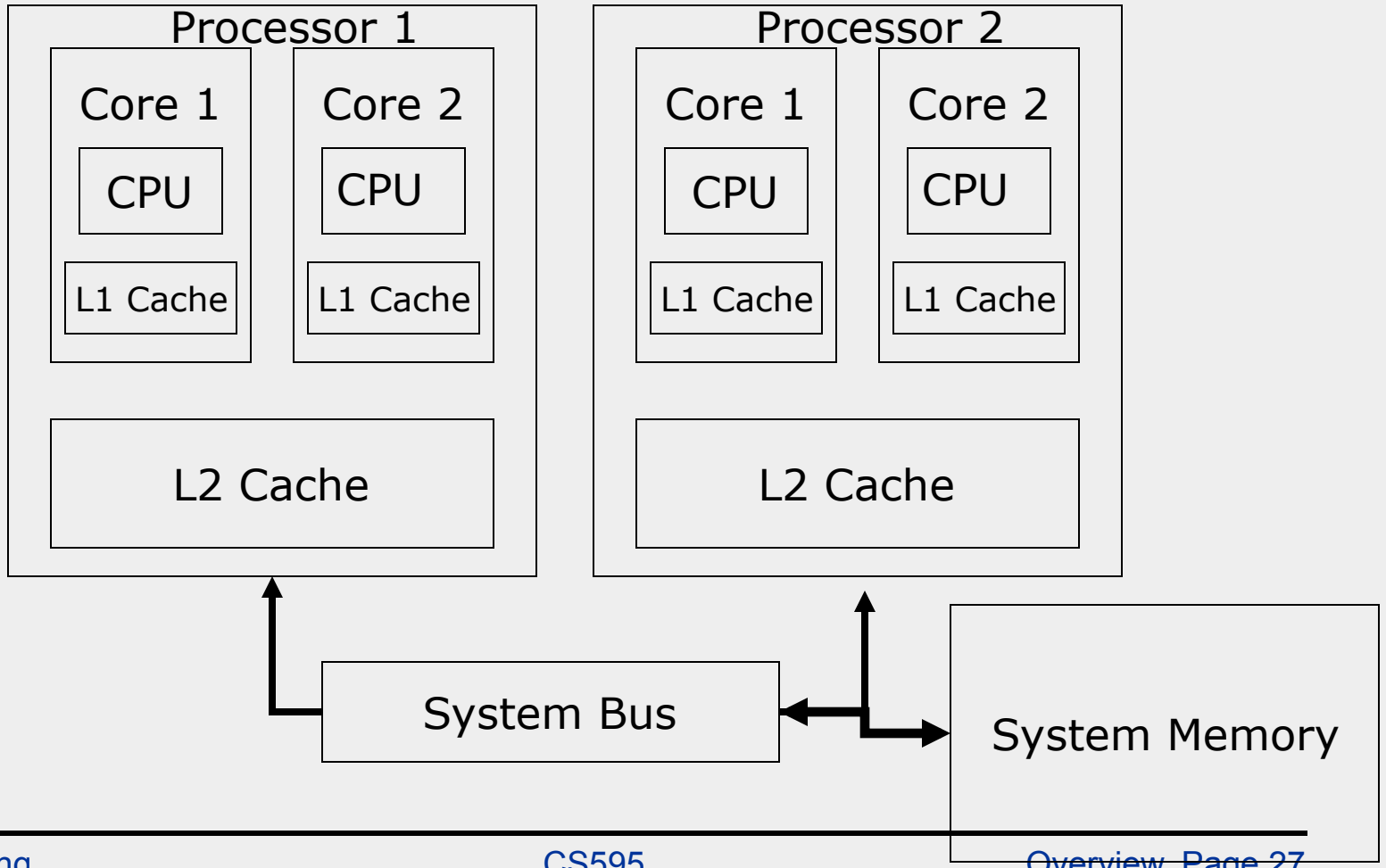


# Hybrid Distributed-Shared Memory

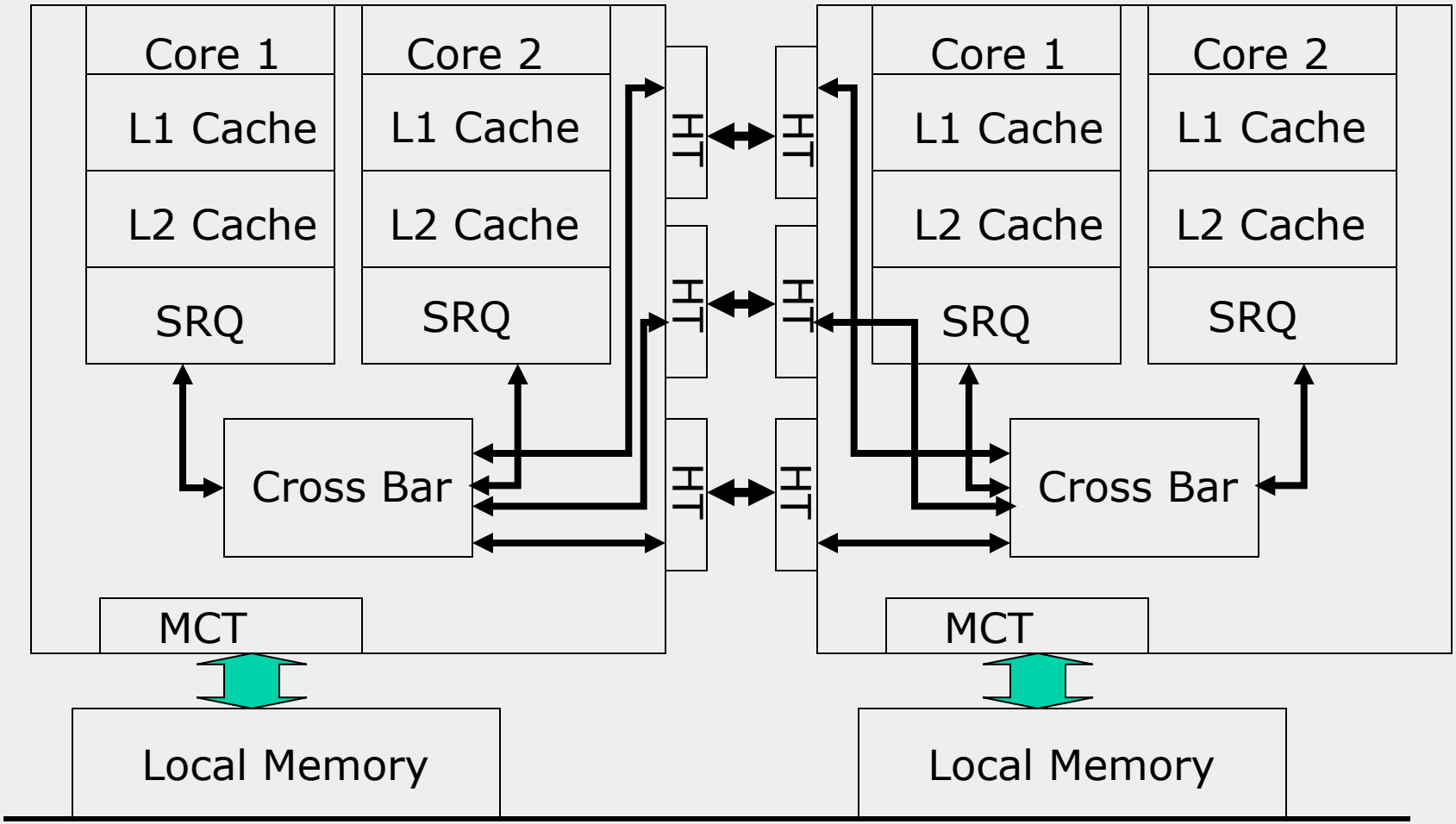
- The largest and fastest computers in the world today employ both shared and distributed memory architectures
- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.



# Intel Xeon Memory Architecture

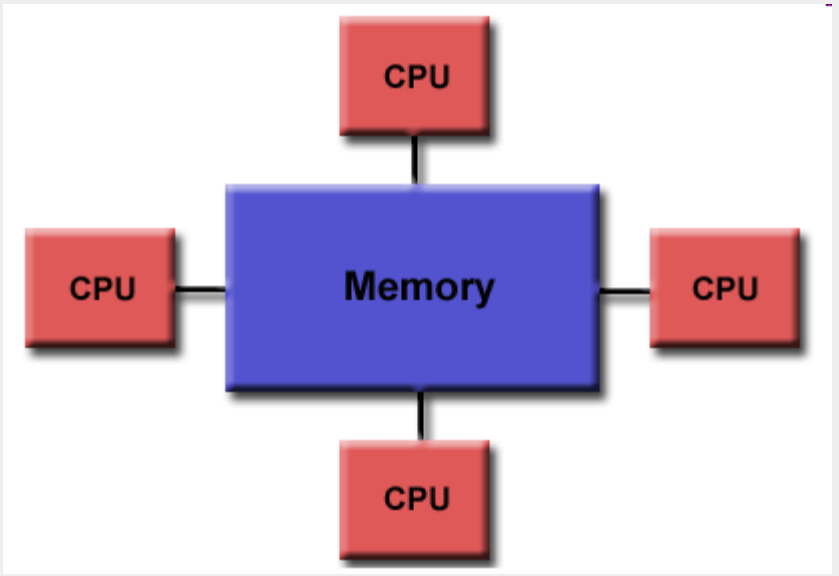


# AMD Opteron Memory Architecture



# Parallel Computer Memory Architectures

- **Shared Memory**
- Have 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.

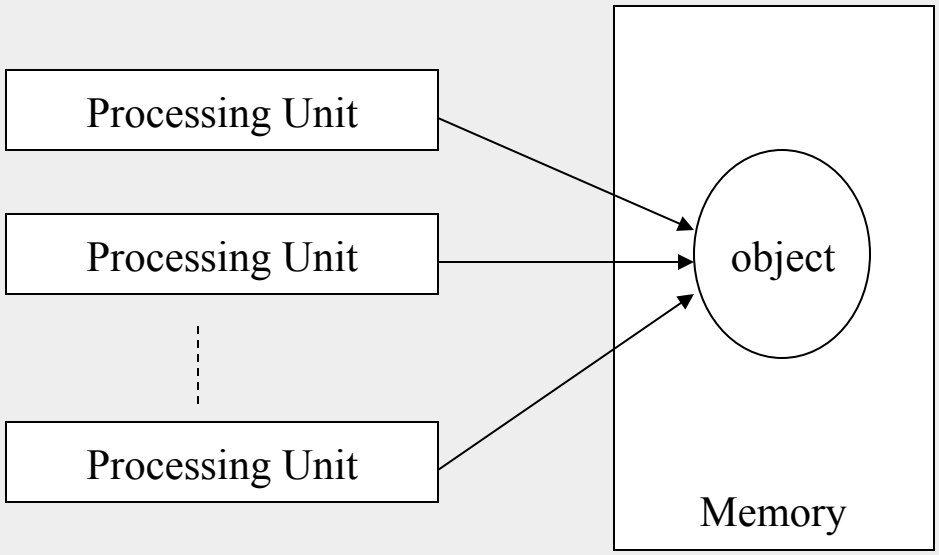


# Shared Memory Architecture

- 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 responsible for synchronization constructs that ensure "correct" access of global memory.
  - Expense: it becomes increasingly difficult and expensive to produce shared memory machines with increasing number of processors (chip design standpoint).

# Shared Address Space

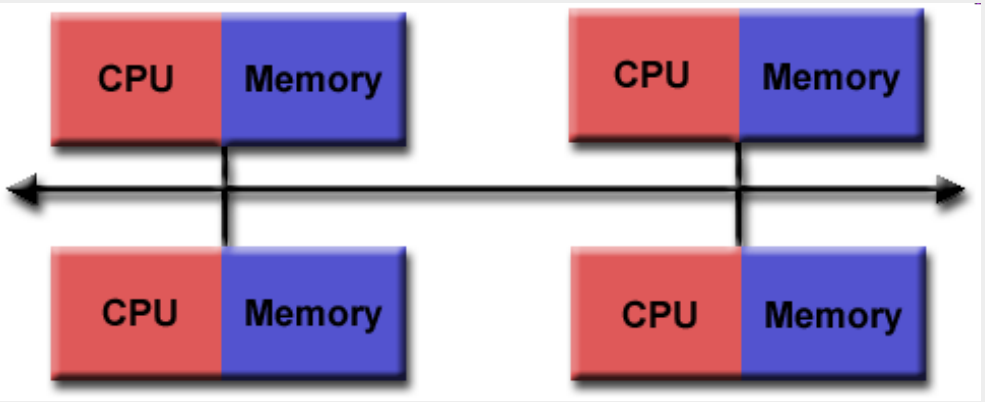
- Shared address space:
  - Processors can directly access all the data in the system



# Distributed Memory

- Distributed memory systems require a communication network to connect inter-processor memory.

- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors
- Each CPU 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.



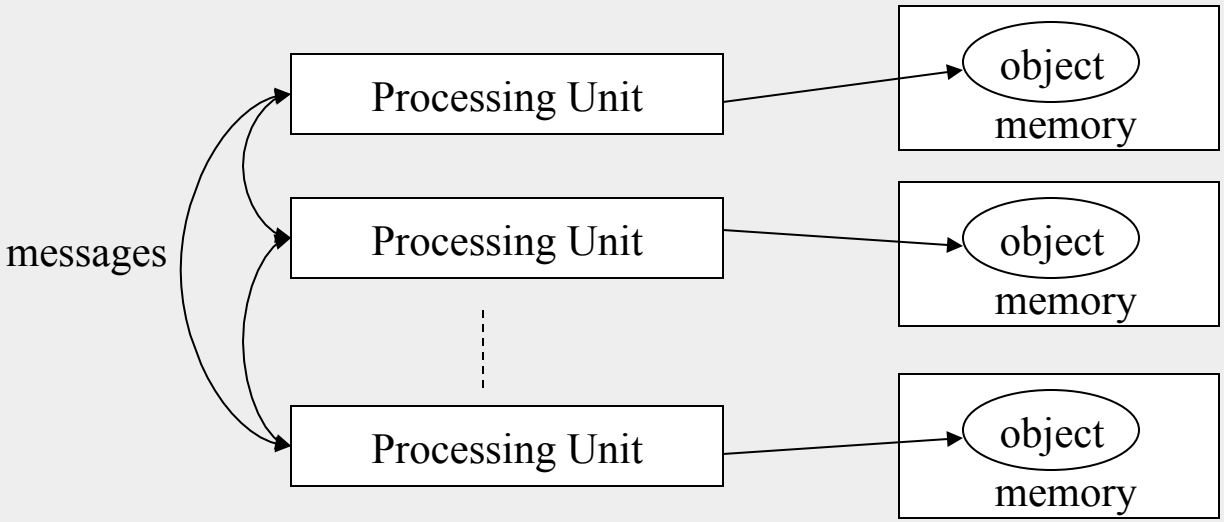

- Getting access to data in another processor is programmer's responsibility. So is synchronization between tasks.

# Distributed Memory

- 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.

# Private Address Space

- Distributed address space:
  - "Shared nothing:" each processor has a private memory
  - Processors can directly access only local data



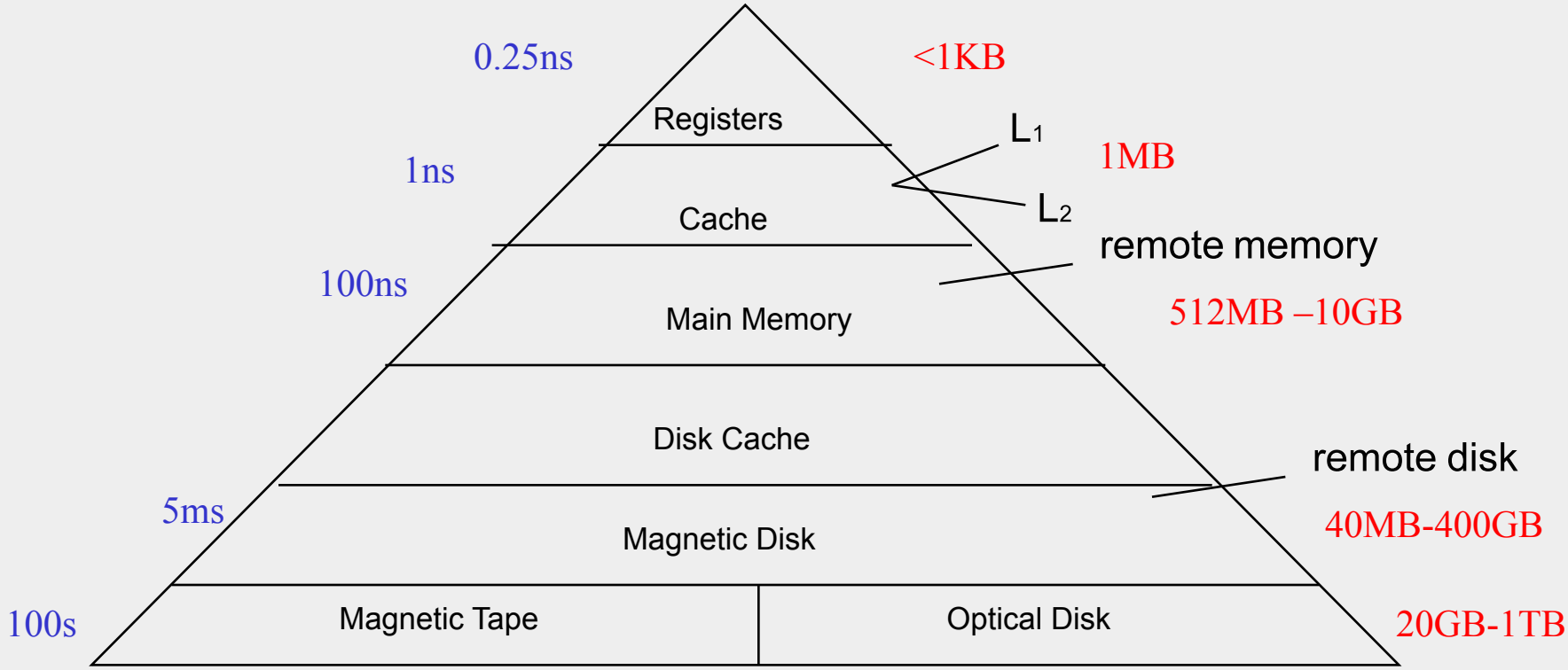
# Common terminology (cont'd)

- **Communications**
  - Data exchange between parallel tasks
    - Accomplished through shared memory bus or over a network
- **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.

# Aspects of Parallel Computing

- Architectures:
  - Processors and memories connected together
  - Memory hierarchy
- Software:
  - Operating systems
  - Compiler
  - Libraries
  - Tools – debuggers, performance analysis
- Algorithms and Programming:
  - Solve large scale problems on parallel computers

# Memory Hierarchy




Contemporary memory hierarchy

# Aspects of Parallel Computing

- Architectures:
  - Processors and memories connected together
  - Memory hierarchy
- Software:
  - Operating systems
  - Compiler
  - Libraries
  - Tools – debuggers, performance analysis
- Algorithms and Programming:
  - Solve large scale problems on parallel computers

# Operating Systems

- Need to support tasks similar to serial OS like Unix
  - Memory and process management, file systems,security
- Additional support needed:
  - Job scheduling: time shared, space sharing
  - Parallel programming support: message passing, synchronization

# Compilers

- Automatic parallelization
- Implicit parallel programming
  - Vector processing
  - Instruction-level parallelism
  - …
- Explicit parallel programming

# Libraries

- Make using parallel machines easier
- Library implementations are usually done by skilled and experienced programmers working closely with machine designers resulting in high levels of performance
- Library routines can be used as building blocks for complex applications
- Usually cover certain specialized application domains

- Examples: PETSc

  http://www.mcs.anl.gov/petsc
- Distributed environment: MPI

  http://www.mcs.anl.gov/mpi

# Tools

- Essential due to degree of complexity in implementation
- Examples:
  - Performance analyzers
    - Help in identifying bottlenecks
    - Can identify relative importance of different parts of program with respect to possible performance gains
  - Debugger:
    - Need to capture the state of multiple processes
    - Bugs commonly caused by synchronization errors are difficult to capture
  - Source control management:
    - Mercurial (http://mercurial.selenic.com/)

# Aspects of Parallel Computing

- Architectures:
  - Processors and memories connected together
- Software:
  - Operating systems
  - Compiler
  - Libraries
  - Tools – debuggers, performance analysis
- Algorithms and Programming:
  - Solve large scale problems on parallel computers

# Parallel Programming Models

- Users must choose a proper parallel programming model to develop their applications on a particular platform.

A parallel programming model

- defines how the programmer creates and coordinates parallel tasks
- is a set of software technologies to express parallel algorithms
  - match applications with the underlying parallel systems.
  - encompasses the applications, programming languages, compilers, libraries, communication systems, parallel I/O, ….

Commonly used parallel programming models:

  - shared memory, threads, message passing, data parallel, hybrid

# Steps to be taken in parallel programming

- Understand the **application**
  - determine whether or not the problem can be parallelized. For example, calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula: F(k + 2) = F(k + 1) + F(k) is not parallelizable

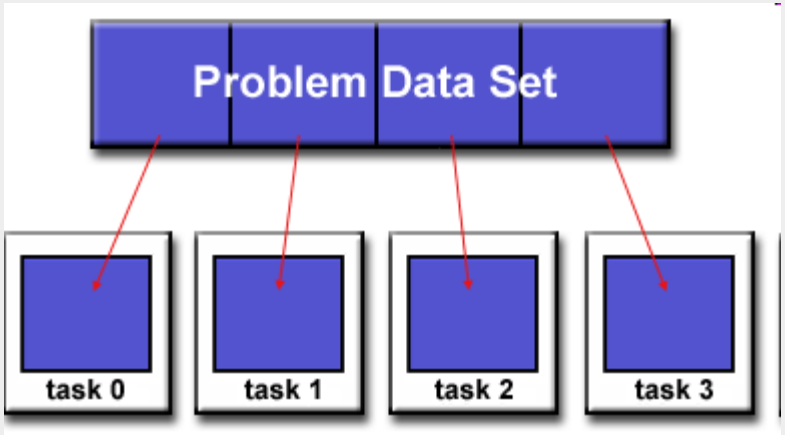
- Identify the program's ***hotspots***
  - Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
  - Profilers and performance analysis tools can help here
  - Focus on parallelizing the hotspots and ignore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

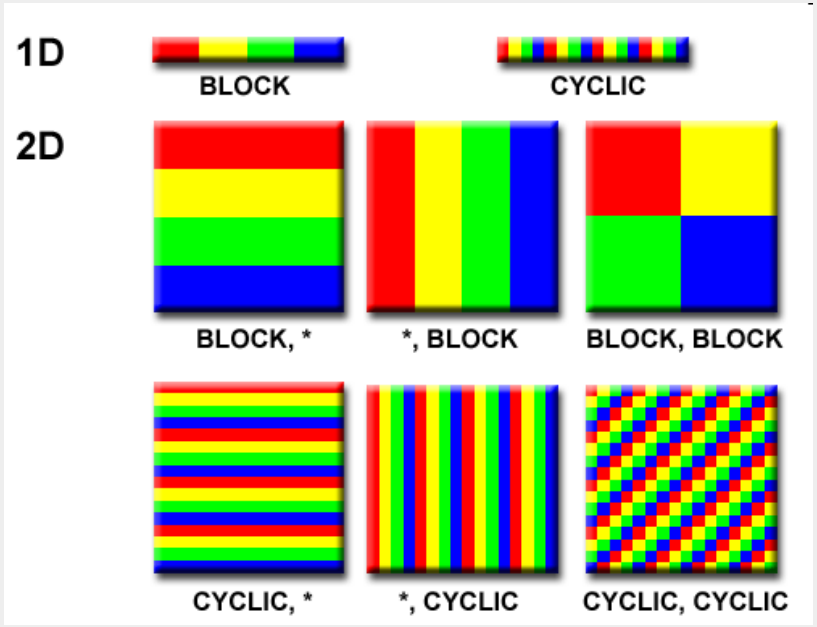
# Steps to be taken in parallel programming

- 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 of the program that account for little CPU usage. Identify inhibitors to parallelism. One common class of inhibitor is *data dependence*
- Identify ***inhibitors*** to parallelism. One common class of inhibitor is *data dependence ....eg:* **A(J) = A(J-1) * 2.0**
- Investigate other algorithms if possible.
  - iterative methods for solving large sparse linear equations Ax = b are more amenable to parallelization. The same is not generally true for direct solvers.

# Considerations in Parallel Programming

- **Partitioning**
  - breaking the problem into discrete "chunks" of work that can be distributed to multiple tasks.
  - Domain decomposition most commonly used. Data associated with a problem is decomposed



# Considerations … (cont'd)

- Load Balancing
  - Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time

- Communications between processors
  - Embarrassingly parallel:

    Little effort is required to separate the problem into parallel tasks (eg., Image processing)

  - Data dependency:

    Most problems require tasks to share data with each other. Changes to neighboring data has a direct effect on that task's data (eg. T(i) = (T(i-1) + T(i+1))/2)

  - Communication cost, Latency vs. Bandwidth, Synchronization,…

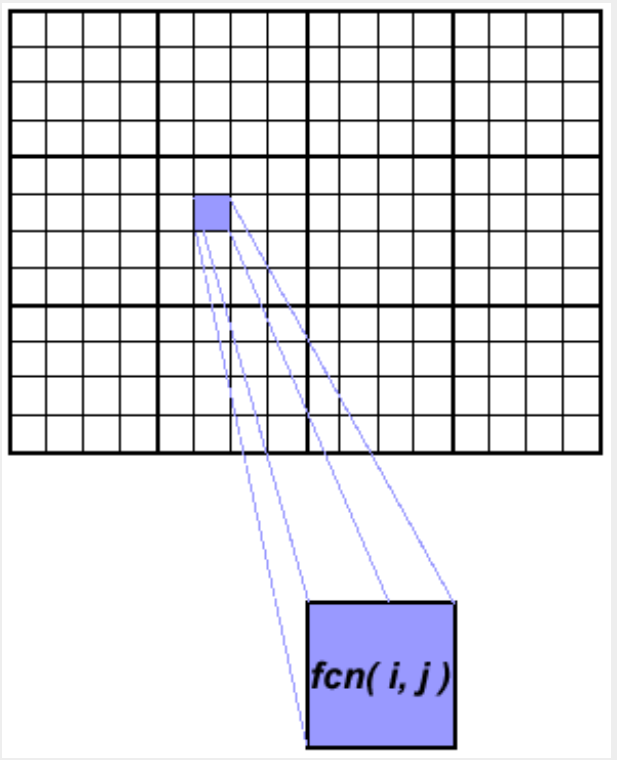
# Considerations … (cont'd)

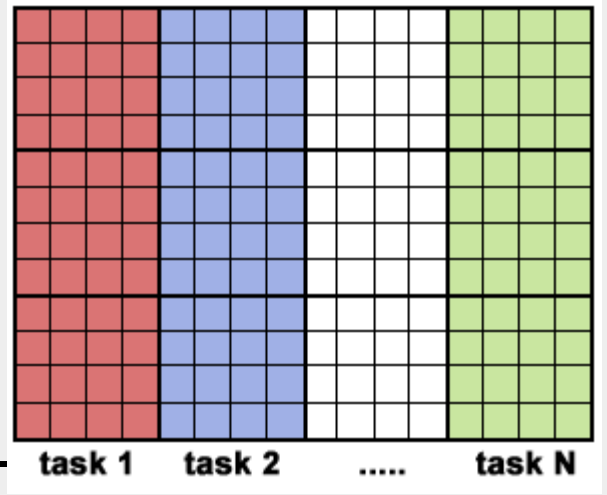
- Speedup and Scalability
  - Affected by algorithm, hardware (memory CPU bus bandwidth, communication network, memory available/CPU etc)
  - Limits: Amdahl's law (potential speed up dependent on fraction of code that can be parallelized)
- Performance Analysis and Tuning
  - debugging, monitoring and analyzing parallel program execution is significantly more of a challenge than for serial programs.
  - Various tools available
- Portability
  - Affected by hardware architecture of the platform, implementation of APIs, operating systems etc.
- I/O
  - Inhibitors to parallelism, parallel file systems available

## Parallel Example – (Array processing)

- 2-dimensional array element
- Each element independent of others

```
do j = 1,n
  do i = 1,n
     a(i,j) = fcn(i,j)
  end do
end do
```



## Parallel Example – (Array processing Implementation)

- Pseudo code shown below

  - Master process initializes array, sends info to worker processes and receives results.
  - Worker process receives info, performs its share of computation and sends results to master.
  - Using the Fortran storage scheme, perform block distribution of the array.

# Implementation

```
find out if I am MASTER or WORKER
    If I am MASTER
        initialize the array
        send each WORKER info on part of array it owns
        send each WORKER its portion of initial array
        receive from each WORKER results
    else if I am WORKER
        receive from MASTER info on part of array I own
        receive from MASTER my portion of initial array #
        calculate my portion of array
                do j = my first column,my last column
                do i = 1,n
                        a(i,j) = fcn(i,j)
                end do
                end do
    send MASTER results
    endif
```

Homework 1
http://blackboard.iit.edu/