Introduction to high performance scientific computing
Parallel computing

MAM5 - INUM, Polytech Nice Sophia

Stéphane Lanteri
Stephane.Lanteri@inria.fr

Nachos project-team
Inria Sophia Antipolis - Méditerranée research center, France

January 2017

Outline

Outline

Preamble

This introductory lecture is for a major part

extracted from the tutorial of

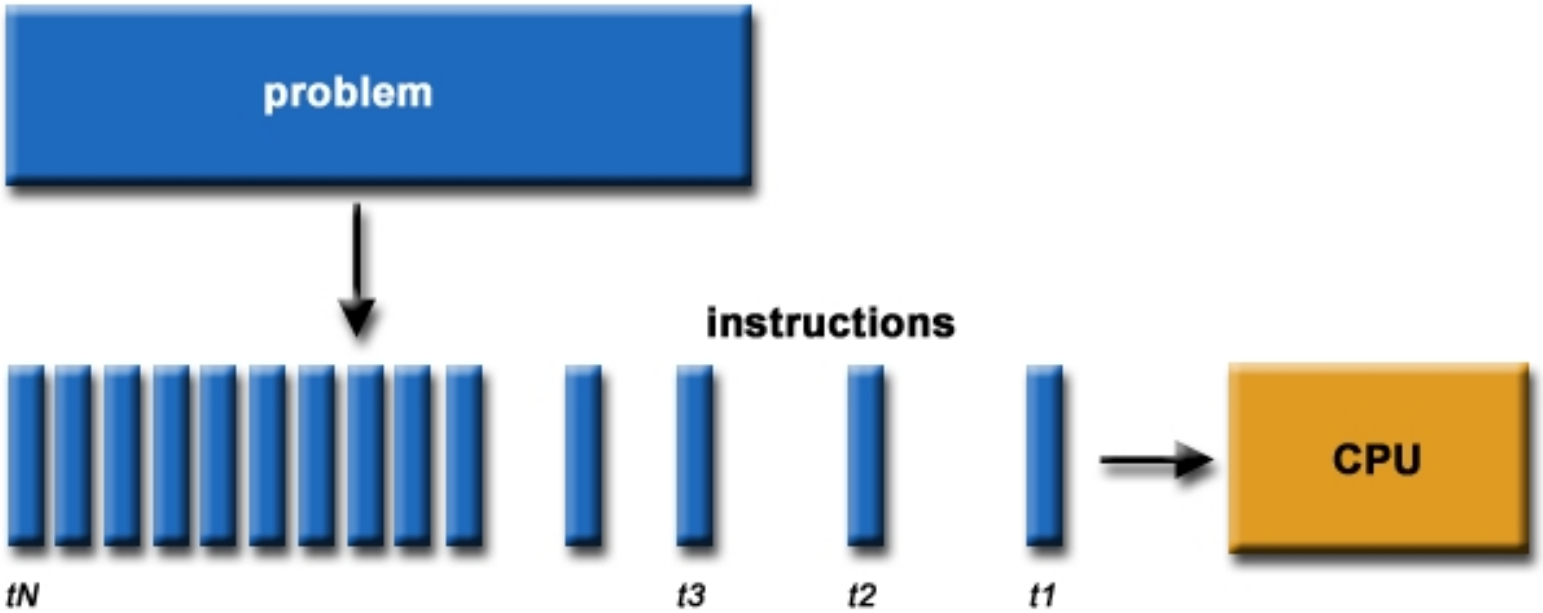
Blaise Barney, Lawrence Livermore National Laboratory

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

Outline

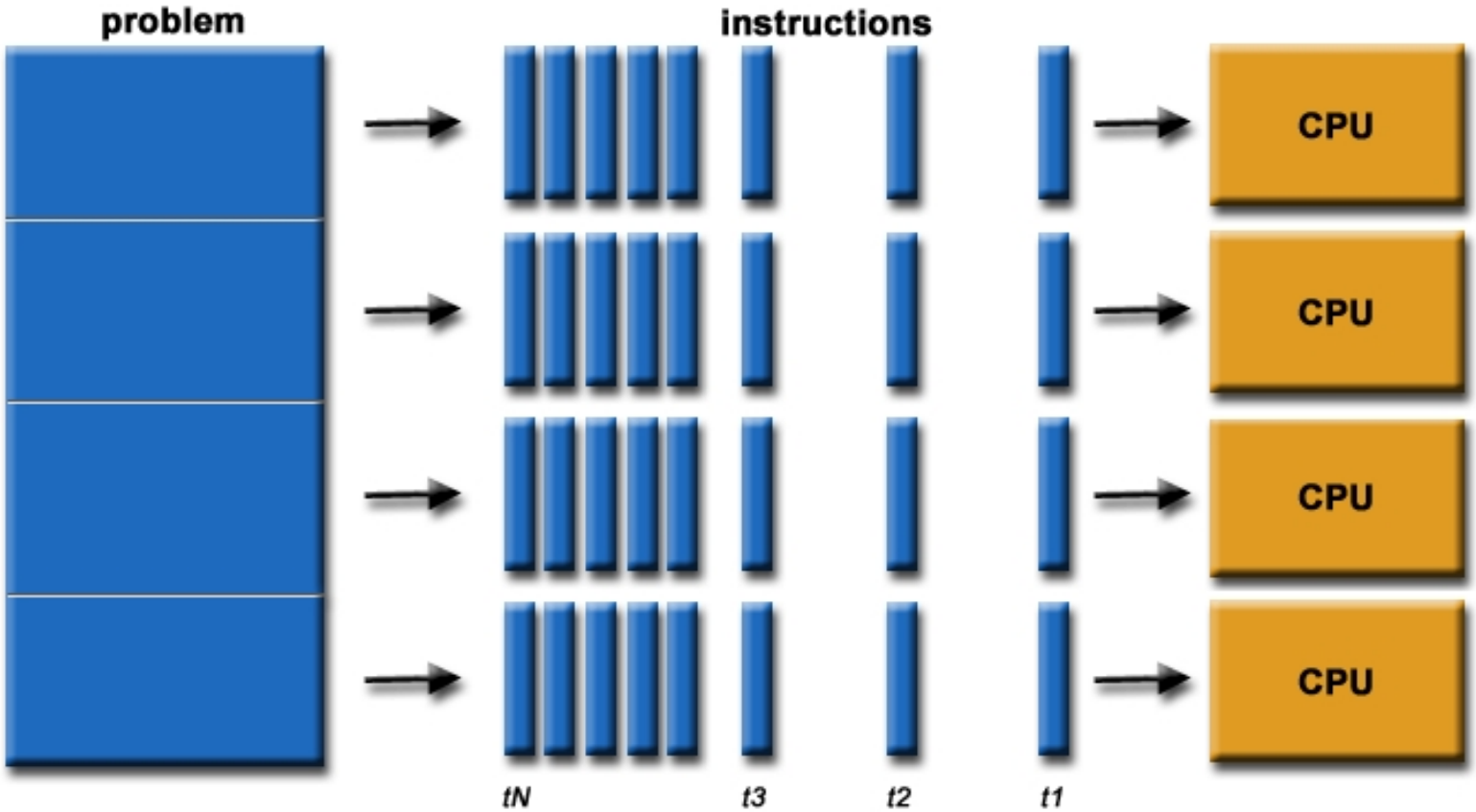
What is parallel computing ?

- Traditionally, software has been written for serial computation:
  - a problem is run on a single computer having a single Central Processing Unit (CPU),
  - it is 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 ?

- 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.



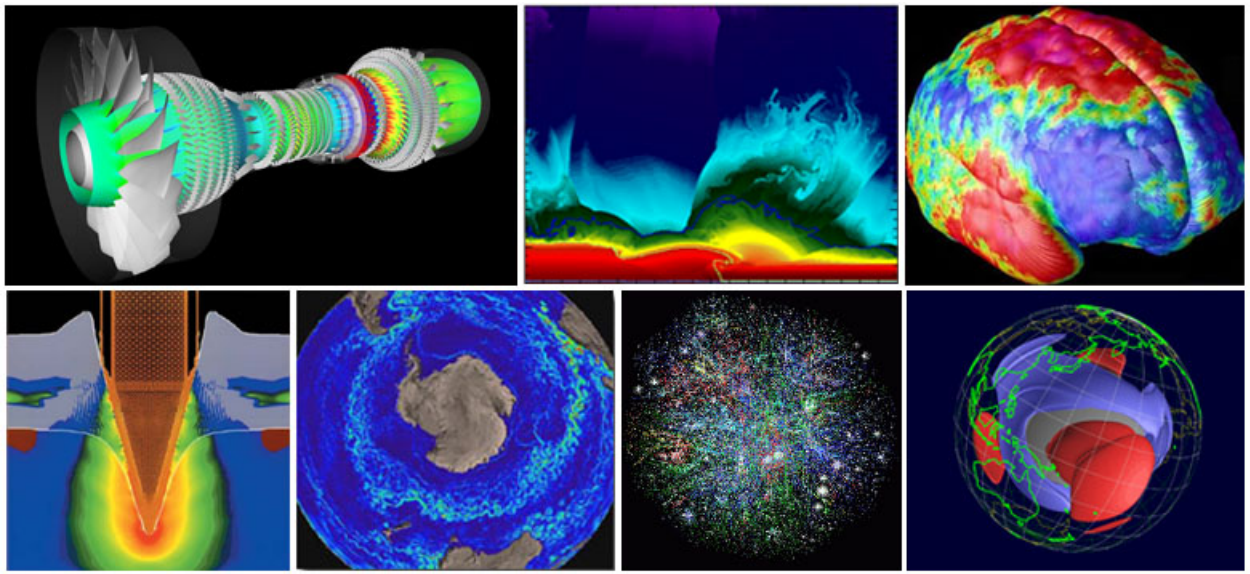
What is parallel computing ?

- 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 problem usually demonstrates characteristics such as the ability to be:
  - broken apart into discrete pieces of work that can be solved simultaneously,
  - execute multiple program instructions at any moment in time,
  - solved in less time with multiple compute resources than with a single compute resource.

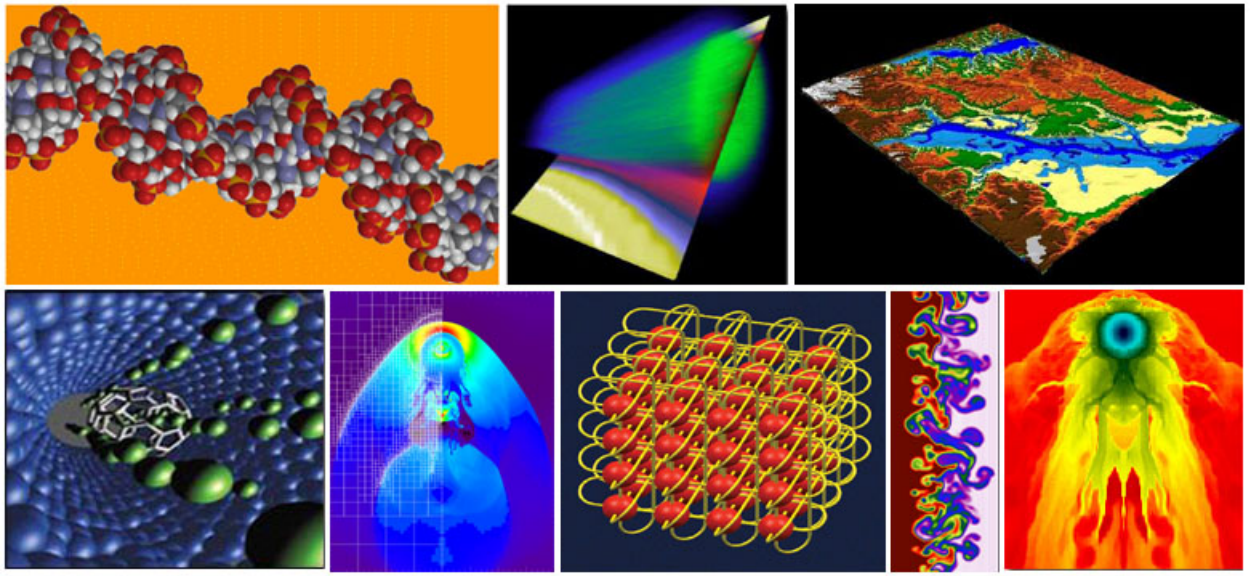
Use of parallel computing

- Historically, parallel computing has been considered to be the high end of computing, and has been used to model difficult scientific and engineering problems found in the real world
  - 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



Use of parallel computing

- Today, commercial applications provide an equal or greater driving force in the development of faster computers
- These applications require the processing of large amounts of data in sophisticated ways
  - Databases, data mining
  - Web search engines, web based business services
  - Medical imaging and diagnosis
  - Financial and economic modeling
  - Advanced graphics and virtual reality, particularly in the entertainment industry
  - Networked video and multi-media technologies
  - Collaborative work environments



Why use of parallel computing ?

- Main Reasons:
  - Save time and/or money - in theory, throwing more resources at a task will shorten its time to completion, with potential cost savings, and parallel clusters can be built from cheap, commodity components
  - Solve larger problems - many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory
  - Provide concurrency:

    a single compute resource can only do one thing at a time,

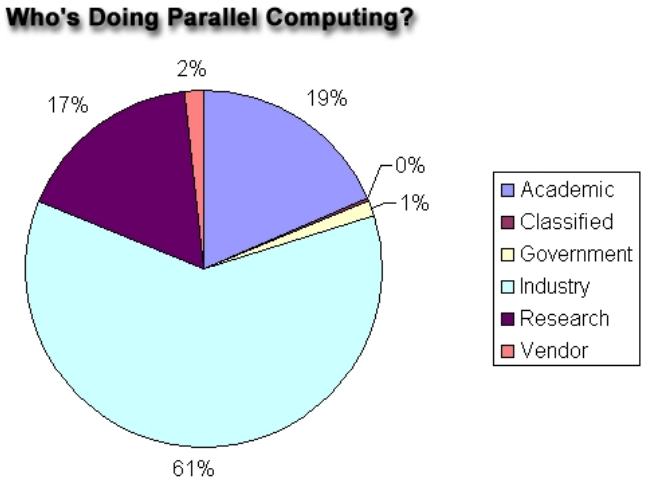
    multiple computing resources can be doing many things simultaneously
  - Example: the Access Grid provides a global collaboration network where people from around the world can meet and conduct work virtually

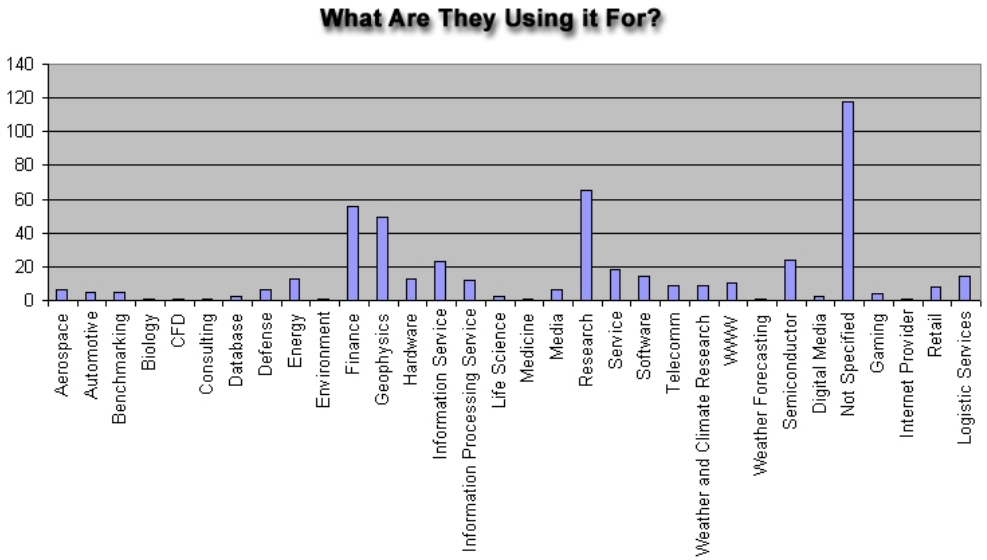
Why use of parallel computing ?

- Limits to serial computing - both physical and practical reasons pose significant constraints to simply building ever 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.
- Current computer architectures are increasingly relying upon hardware level parallelism to improve performance: multiple execution units, pipelined instructions, multi-core

Who and what ?

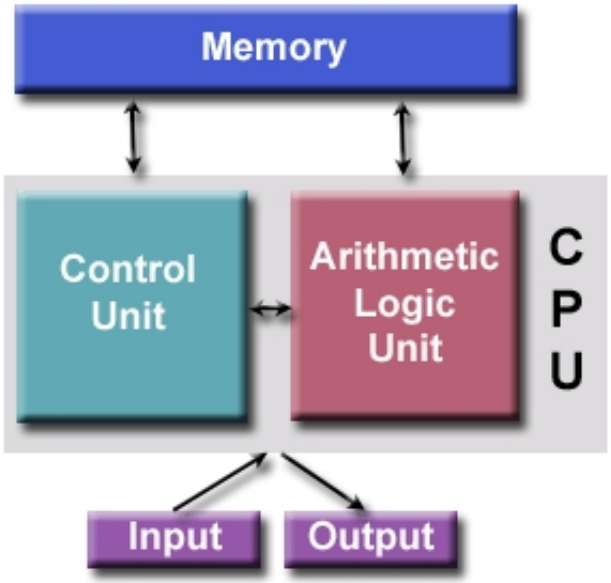
- Top500.org provides statistics on parallel computing users



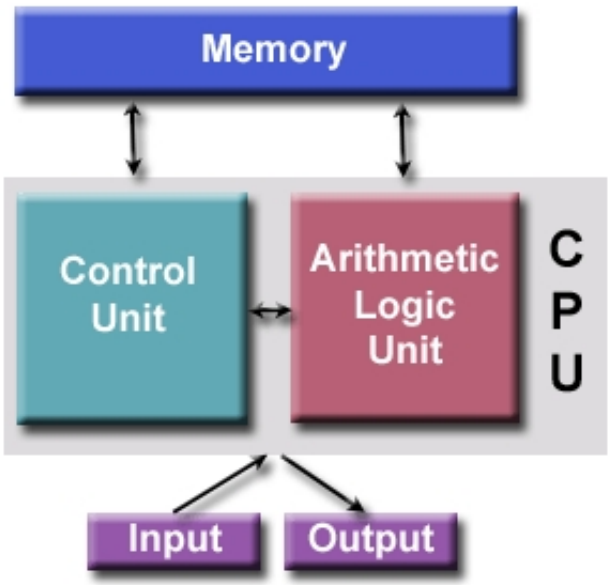


Outline

- Named after the Hungarian mathematician John von Neumann who first authored the general requirements for an electronic computer in his 1945 papers
- Since then, virtually all computers have followed this basic design, which differed from earlier computers programmed through hard wiring

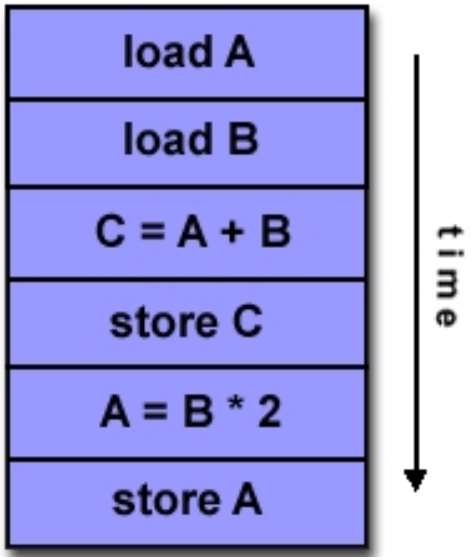


- Comprised of four main components: memory, control unit, arithmetic logic unit, input/output
- Read/write, random access memory is used to store both program instructions and data:
  - program instructions are coded data which tell the computer to do something,
  - data is simply information to be used by the program.
- Control unit fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task
- Aritmetic unit performs basic arithmetic operations
- Input/output is the interface to the human operator



- There are different ways to classify parallel computers
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of instruction and data
- Each of these dimensions can have only one of two possible states: single or multiple
- The matrix below defines the 4 possible classifications according to Flynn:
  - SISD: Single Instruction Single Data
  - SIMD: Single Instruction Multiple Data
  - MISD: Multiple Instruction Single Data
  - MIMD: Multiple Instruction Multiple Data

- 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



CDC 7600

CRAY 1

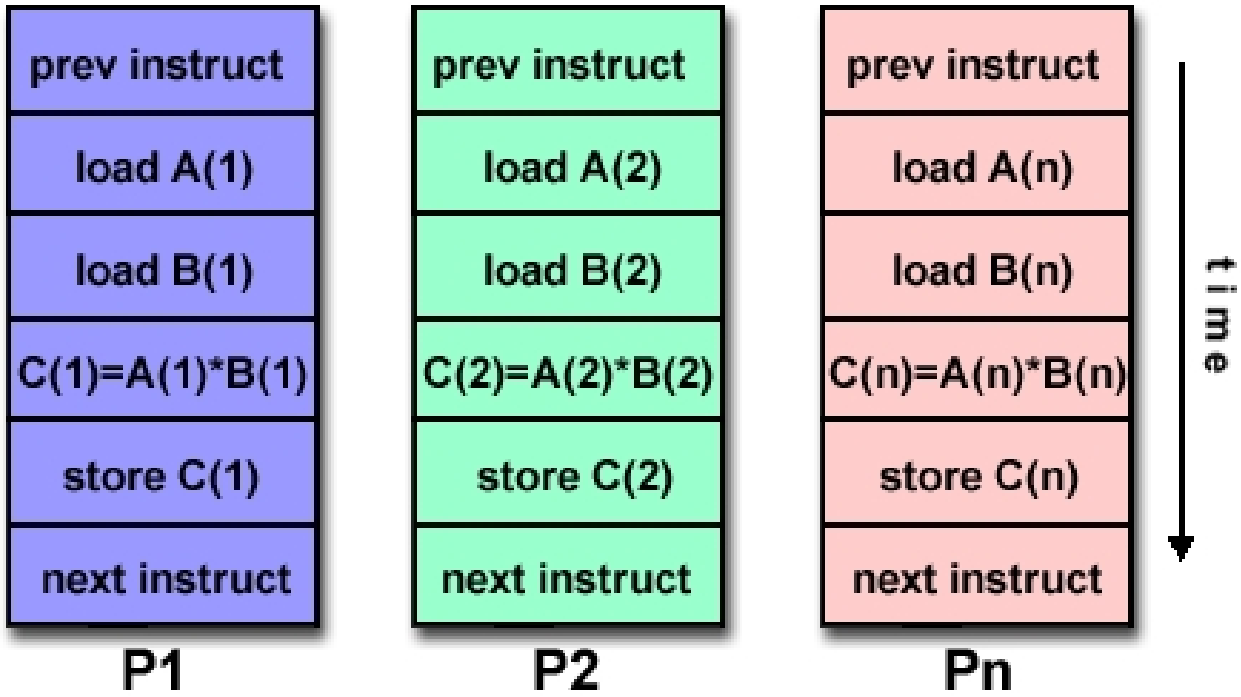
PDP 1

IBM 360

UNIVAC 1

DELL laptop

- 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
  - Processor arrays: Connection Machine CM-2, MasPar MP-1 and MP-2, ILLIAC IV
  - Vector Pipelines: IBM 9000, Cray X-MP, Y-MP and C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
- Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units

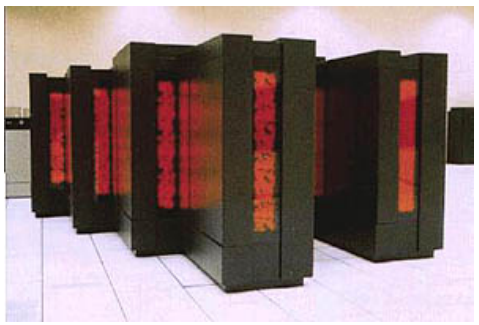

prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
load A(2)
load B(2)
C(2)=A(2)*B(2)
store C(2)
next instruct
P2
prev instruct
load A(n)
load B(n)
C(n)=A(n)*B(n)
store C(n)
next instruct
Pn
time

MasPar

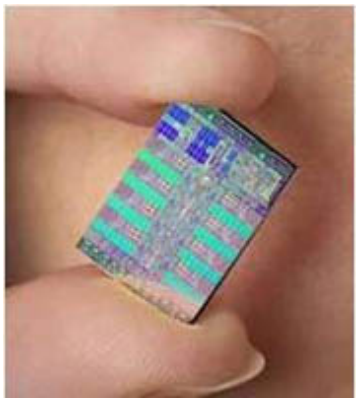
Cray X-MP

Cray Y-MP

ILLIAC IV

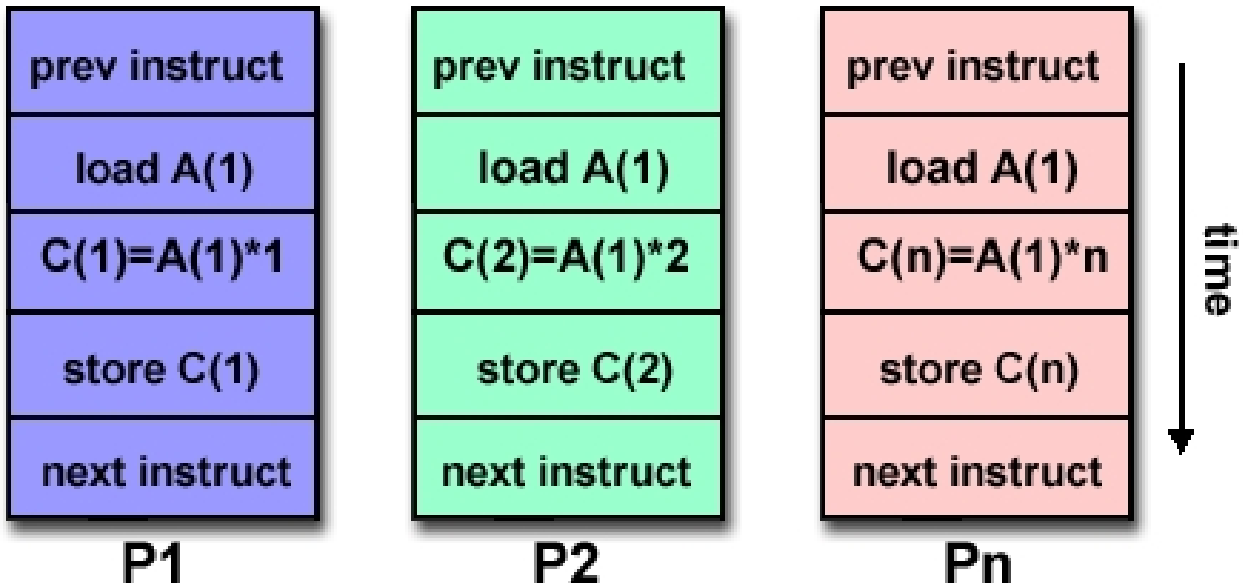


CM-2

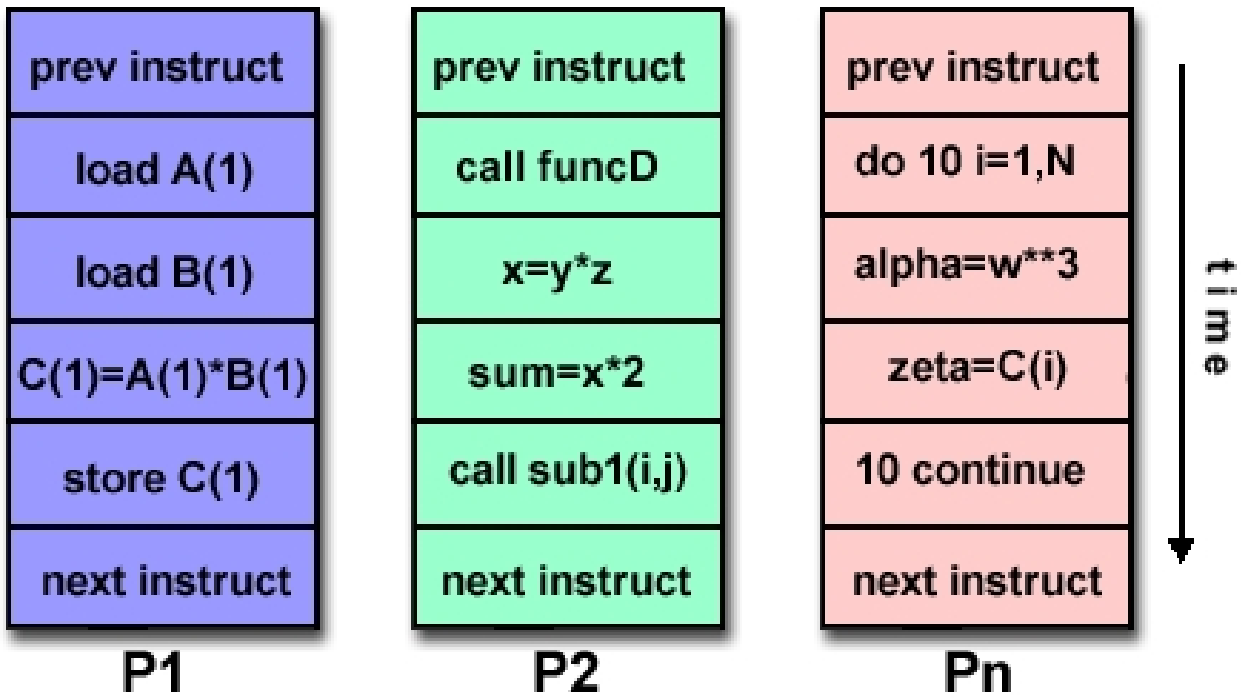


Cell Processor (GPU)

- 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 (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.



- Currently, the most common type of parallel computer
- 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 and grids, multi-processor SMP computers, multi-core PCs
- Note: many MIMD architectures also include SIMD execution sub-components


prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
call funcD
x=y*z
sum=x*2
call sub1(i,j)
next instruct
P2
prev instruct
do 10 i=1,N
alpha=w**3
zeta=C(i)
10 continue
next instruct
Pn
time

HP/Compaq Alphaserver

Intel IA32 cluster

IBM POWER5

AMD Opteron cluster

Cray XT3

IBM BG/L

- Task
  - A logically discrete section of computational work
  - A task is typically a program or program-like set of instructions that is executed by a processor
- Parallel task: a task that can be executed by multiple processors safely (yields correct results)
- Serial execution
  - Execution of a program sequentially, one statement at a time
  - In the simplest sense, this is what happens on a one processor machine
  - However, virtually all parallel tasks will have sections of a parallel program that must be executed serially
- Parallel execution: execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time
- Shared memory
  - From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory
  - In a programming sense, it describes a model where parallel tasks all have the same picture of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists

- Task
  - A logically discrete section of computational work
  - A task is typically a program or program-like set of instructions that is executed by a processor
- Parallel task: a task that can be executed by multiple processors safely (yields correct results)
- Serial execution
  - Execution of a program sequentially, one statement at a time
  - In the simplest sense, this is what happens on a one processor machine
  - However, virtually all parallel tasks will have sections of a parallel program that must be executed serially
- Parallel execution: execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time
- Shared memory
  - From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory
  - In a programming sense, it describes a model where parallel tasks all have the same picture of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists

- Task
  - A logically discrete section of computational work
  - A task is typically a program or program-like set of instructions that is executed by a processor
- Parallel task: a task that can be executed by multiple processors safely (yields correct results)
- Serial execution
  - Execution of a program sequentially, one statement at a time
  - In the simplest sense, this is what happens on a one processor machine
  - However, virtually all parallel tasks will have sections of a parallel program that must be executed serially
- Parallel execution: execution of a program by more than one task, with each task being able to execute the same or different statement at the same moment in time
- Shared memory
  - From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory
  - In a programming sense, it describes a model where parallel tasks all have the same picture of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists

- Symmetric Multi-Processor (SMP)
  - Hardware architecture where multiple processors share a single address space and access to all resources
- Distributed memory
  - In hardware, refers to network based memory access for physical memory that is not common
  - As a programming model, tasks can only logically see local machine memory and must use communications to access memory on other machines where other tasks are executing
- Communications
  - Parallel tasks typically need to exchange data
  - There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed

- Symmetric Multi-Processor (SMP)
  - Hardware architecture where multiple processors share a single address space and access to all resources
- Distributed memory
  - In hardware, refers to network based memory access for physical memory that is not common
  - As a programming model, tasks can only logically see local machine memory and must use communications to access memory on other machines where other tasks are executing
- Communications
  - Parallel tasks typically need to exchange data
  - There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed

- Symmetric Multi-Processor (SMP)
  - Hardware architecture where multiple processors share a single address space and access to all resources
- Distributed memory
  - In hardware, refers to network based memory access for physical memory that is not common
  - As a programming model, tasks can only logically see local machine memory and must use communications to access memory on other machines where other tasks are executing
- Communications
  - Parallel tasks typically need to exchange data
  - There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed

- Synchronization
  - The coordination of parallel tasks in real time, very often associated with communications
  - Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point
  - Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase
- Granularity
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication
  - Coarse: relatively large amounts of computational work are done between communication events
  - Fine: relatively small amounts of computational work are done between communication events
- Observed speedup
  - Observed speedup of a code which has been parallelized, defined as: wall-clock time of serial execution/wall-clock time of parallel execution
  - One of the simplest and most widely used indicators for a parallel program's performance

- Synchronization
  - The coordination of parallel tasks in real time, very often associated with communications
  - Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point
  - Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase
- Granularity
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication
  - Coarse: relatively large amounts of computational work are done between communication events
  - Fine: relatively small amounts of computational work are done between communication events
- Observed speedup
  - Observed speedup of a code which has been parallelized, defined as: wall-clock time of serial execution/wall-clock time of parallel execution
  - One of the simplest and most widely used indicators for a parallel program's performance

- Synchronization
  - The coordination of parallel tasks in real time, very often associated with communications
  - Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point
  - Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase
- Granularity
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication
  - Coarse: relatively large amounts of computational work are done between communication events
  - Fine: relatively small amounts of computational work are done between communication events
- Observed speedup
  - Observed speedup of a code which has been parallelized, defined as: wall-clock time of serial execution/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 can include factors such as task start-up time, synchronizations, data communications, software overhead imposed by parallel compilers, libraries, tools, operating system, etc. and task termination time
- Massively parallel
  - Refers to the hardware that comprises a given parallel system having many processors
  - The meaning of many keeps increasing, but currently, 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

- Parallel overhead
  - The amount of time required to coordinate parallel tasks, as opposed to doing useful work
  - Parallel overhead can include factors such as task start-up time, synchronizations, data communications, software overhead imposed by parallel compilers, libraries, tools, operating system, etc. and task termination time
- Massively parallel
  - Refers to the hardware that comprises a given parallel system having many processors
  - The meaning of many keeps increasing, but currently, 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

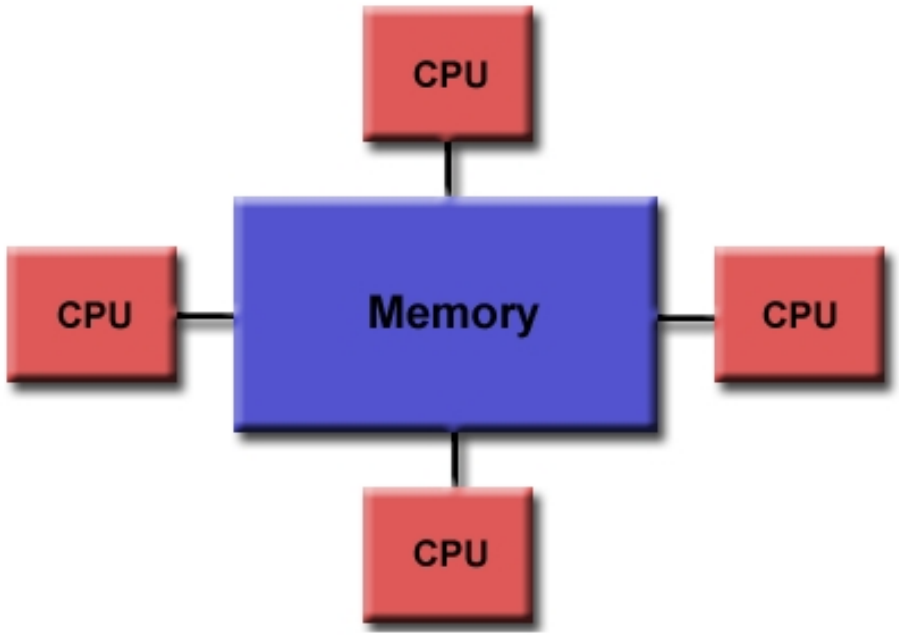
- Scalability
  - Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors
  - Factors that contribute to scalability include hardware (particularly memory-cpu bandwidths and network communications), application algorithm, parallel overhead related issues and characteristics of the specific application and coding
- Multi-core processors
  - Multiple processors (cores) on a single chip
- Cluster computing
  - Use of a combination of commodity units (processors, networks or SMPs) to build a parallel system
- Supercomputing/high performance computing
  - Use of large machines to solve large problems

- 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 issues and characteristics of the specific application and coding
- Multi-core processors
  - Multiple processors (cores) on a single chip
- Cluster computing
  - Use of a combination of commodity units (processors, networks or SMPs) to build a parallel system
- Supercomputing/high performance computing
  - Use of large machines to solve large problems

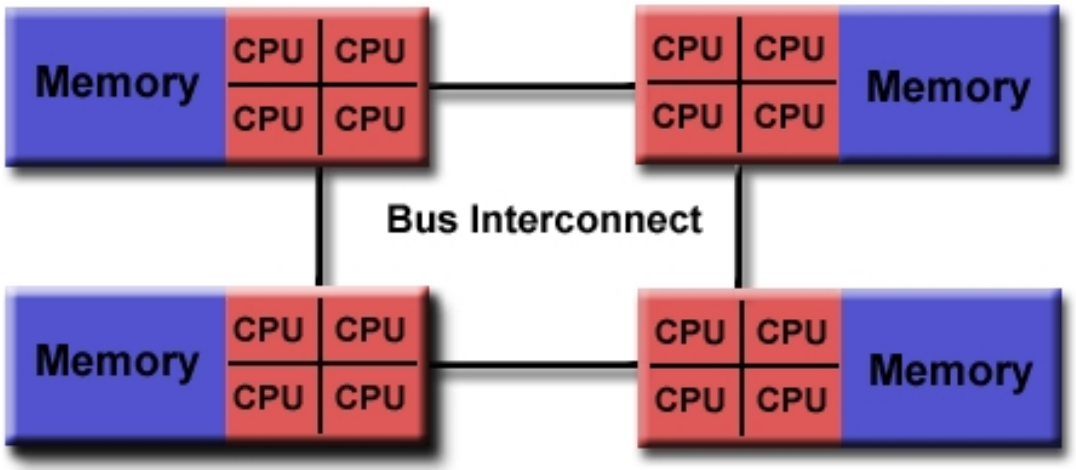
Outline

- General characteristics
  - Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space
  - Multiple processors can operate independently but share the same memory resources
  - Changes in a memory location effected by one processor are visible to all other processors
  - Shared memory machines can be divided into two main classes based upon memory access times: UMA and NUMA

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

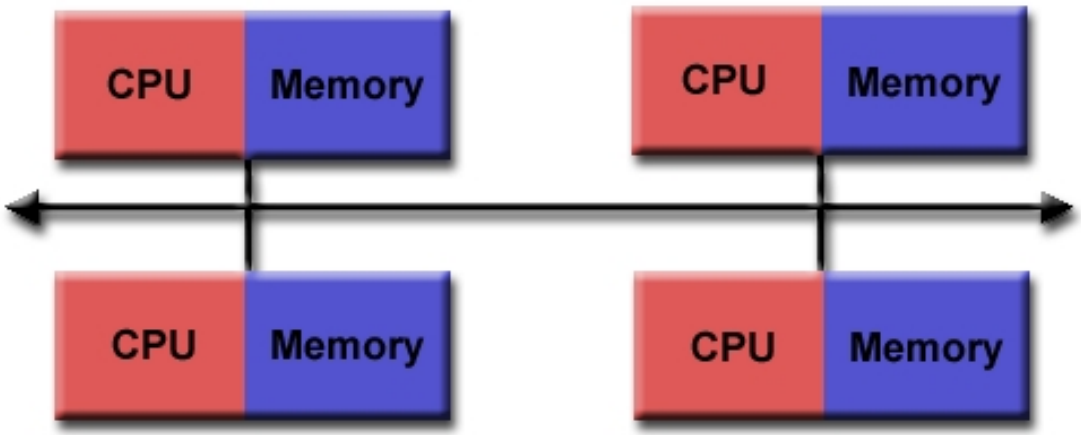


- Non-Uniform Memory Access (NUMA)
  - Often made by physically linking two or more SMPs
  - One SMP can directly access memory of another SMP
  - Not all processors have equal access time to all memories
  - Memory access across link is slower
  - If cache coherency is maintained, then may also be called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



- Advantages
  - Global address space provides a user-friendly programming perspective to memory
  - Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs
- Disadvantages
  - Primary disadvantage is the lack of scalability between memory and CPUs
    - Adding more CPUs can geometrically increases traffic on the shared memory-CPU path, and for cache coherent systems, geometrically increase traffic associated with cache/memory management
  - Programmer responsibility for synchronization constructs that insure correct access of global memory
  - Expensive: it becomes increasingly difficult and expensive to design and produce shared memory machines with ever increasing numbers of processors

- General characteristics
  - Like shared memory systems, distributed memory systems vary widely but share a common characteristic
  - Distributed memory systems require a communication network to connect inter-processor memory

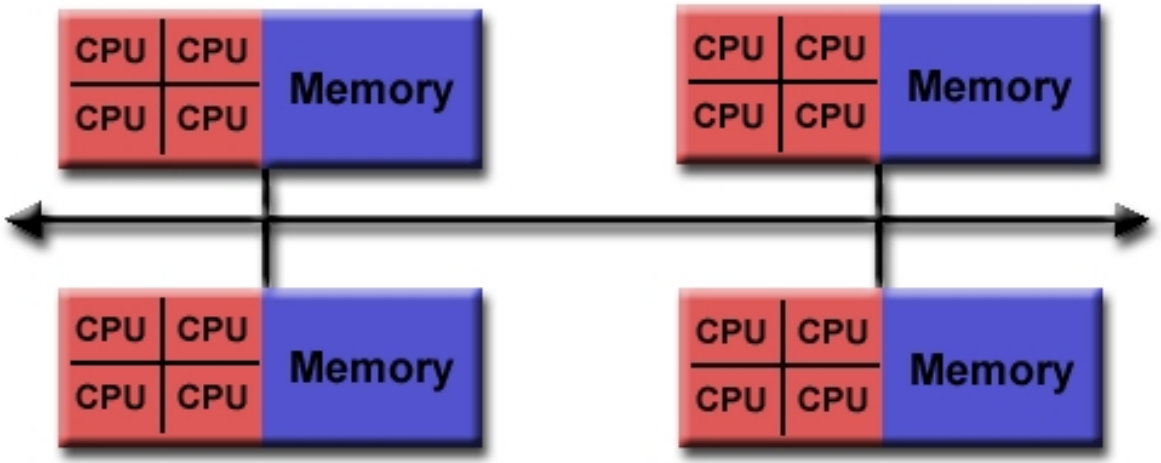


- General characteristics
  - Processors have their own local memory
    - Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors
    - Because each processor has its own local memory, it operates independently
    - Changes 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 number of processors i.e. 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

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



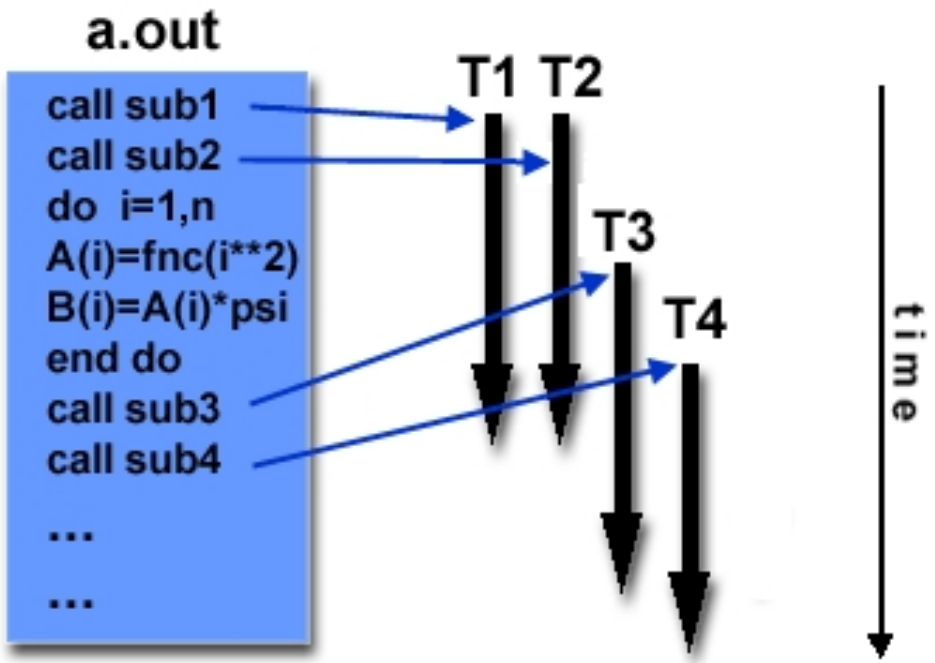
Outline

- There are several parallel programming models in common use:
  - shared memory,
  - threads,
  - message passing,
  - data parallel,
  - hybrid.
- Parallel programming models exist as an abstraction above hardware and memory architectures
- 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 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

- In the shared-memory programming model, tasks share a common address space, which they read and write 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, and 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
  - On shared memory platforms, the native compilers translate user program variables into actual memory addresses, which are global
  - No common distributed memory platform implementations currently exist

- 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, while 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 insure that more than one thread is not updating the same global address at any time
  - Threads can come and go, but a.out remains present to provide the necessary shared resources until the application has completed

- Threads are commonly associated with shared memory architectures and operating systems

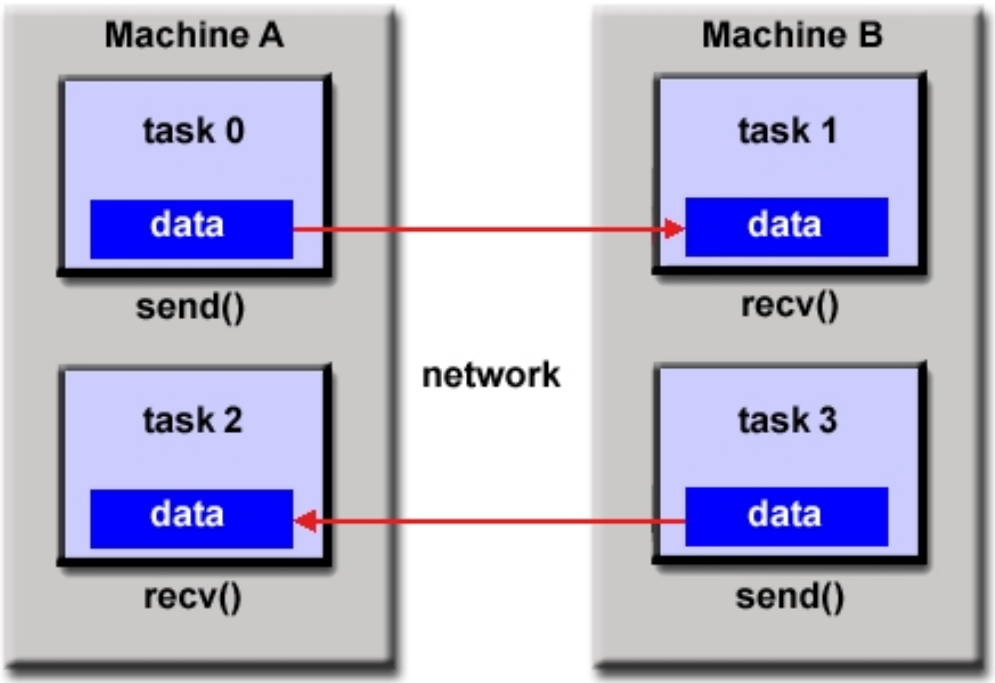


- From a programming perspective, threads implementations commonly comprise:
  - a library of subroutines that are called from within parallel source code,
  - a set of compiler directives imbedded in either serial or parallel source code.
- In both cases, the programmer is responsible for determining 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

- POSIX threads
  - Library based; requires parallel coding
  - Specified by the IEEE POSIX 1003.1c standard (1995)
  - C Language only
  - Commonly referred to as Pthreads
  - Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations
  - Very explicit parallelism; requires significant programmer attention to detail
  - POSIX threads tutorial: computing.llnl.gov/tutorials/pthreads
- 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

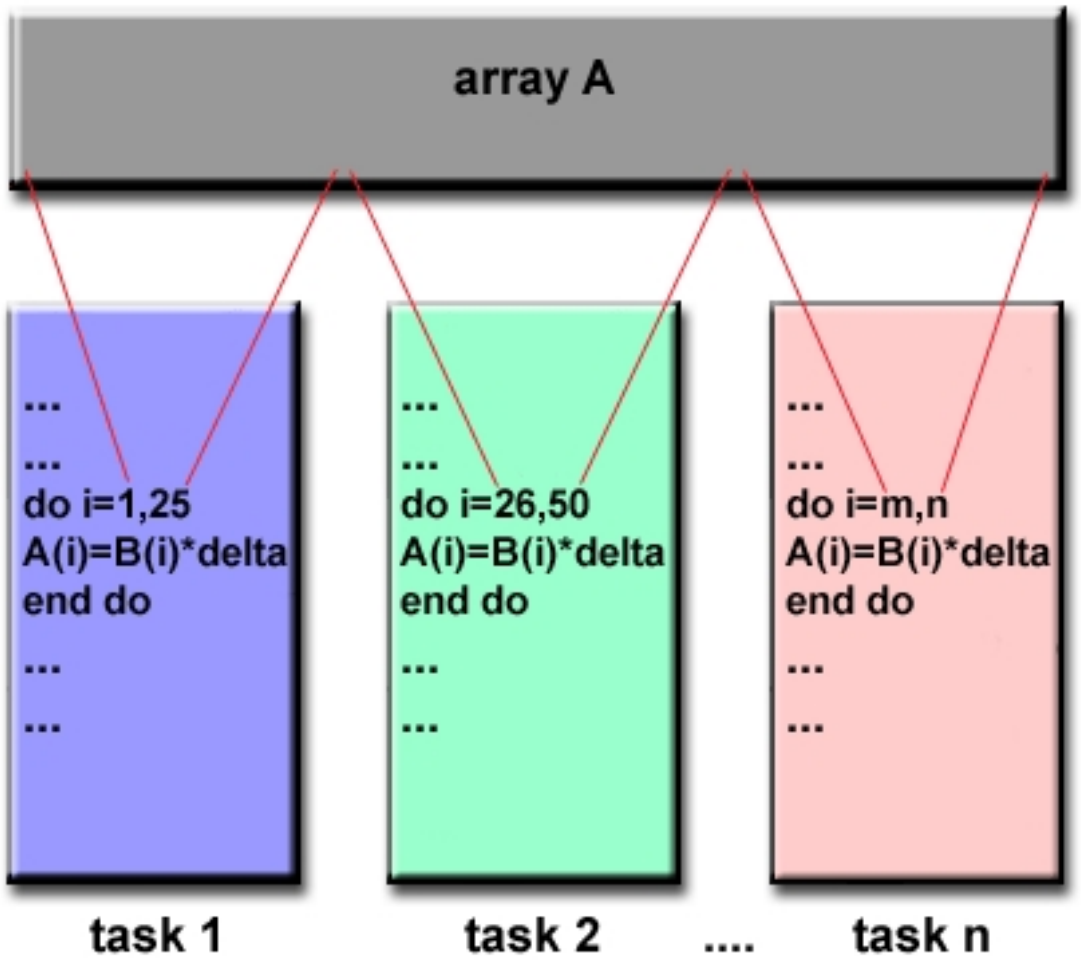
- POSIX threads
  - Library based; requires parallel coding
  - Specified by the IEEE POSIX 1003.1c standard (1995)
  - C Language only
  - Commonly referred to as Pthreads
  - Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations
  - Very explicit parallelism; requires significant programmer attention to detail
  - POSIX threads tutorial: computing.llnl.gov/tutorials/pthreads
- 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

- The message passing 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 as well 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).



- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in source code, and 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
- For shared memory architectures, MPI implementations usually don't use a network for task communications and instead use shared memory (memory copies) for performance reasons
- MPI tutorial: computing.llnl.gov/tutorials/mpi

- The data parallel model demonstrates 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.
- 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


array A
...
...
do i=1,25
A(i)=B(i)*delta
end do
...
...
...
...
do i=26,50
A(i)=B(i)*delta
end do
...
...
...
...
do i=m,n
A(i)=B(i)*delta
end do
...
...
task 1
task 2
....
task n

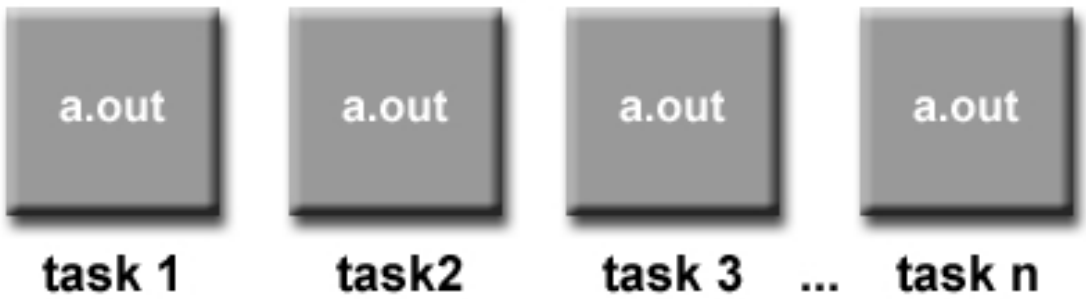
- Programming with the data parallel model is usually accomplished by writing a program with data parallel constructs
- The constructs can be calls to a data parallel subroutine library or, compiler directives recognized by a data parallel compiler
- Fortran 90 and 95 (F90, F95) - ISO/ANSI standard extensions 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)
  - Implementations are available for most common parallel platforms
- 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 have the compiler convert the program into standard code with calls to a message passing library (MPI usually) to distribute the data to all the processes; all message passing is done invisibly to the programmer

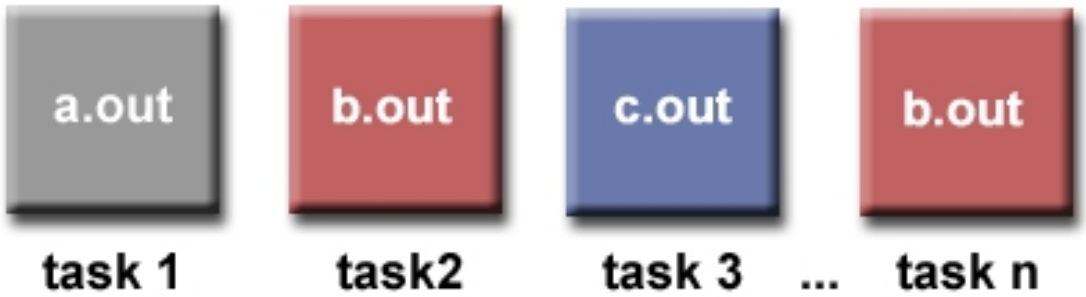
- Other parallel programming models besides those previously mentioned certainly exist, and will continue to evolve along with the ever changing world of computer hardware and software

- Hybrid
  - In this model, any two or more parallel programming models are combined
  - Currently, a common example of a hybrid model is the combination of the message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP)
  - Another common example of a hybrid model is combining data parallel with message passing (data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer)

- 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
  - A single program is executed by all tasks simultaneously
  - At any moment in time, tasks can be executing the same or different instructions within the same program
  - 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)
  - All tasks may use different data



- 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
  - MPMD applications typically have multiple executable object files (programs)
  - While the application is being run in parallel, each task can be executing the same or different program as other tasks
  - All tasks may use different data



Outline

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

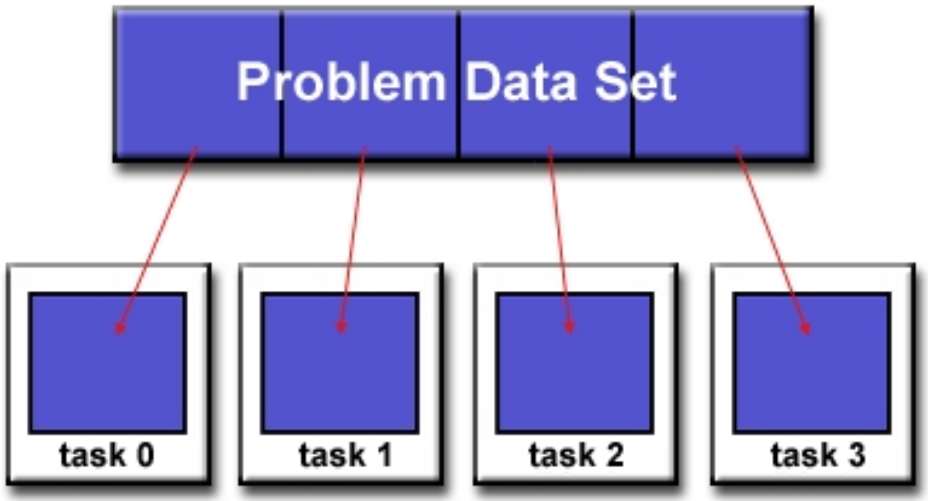
- A parallelizing compiler generally works in two different ways
- Fully automatic:
  - the compiler analyzes the source code and identifies opportunities for parallelism,
  - the analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance,
  - loops (do, for) are the most frequent target for automatic parallelization.
- Programmer directed:
  - using compiler directives or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code,
  - may be able to be used in conjunction with some degree of automatic parallelization also.

- 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.

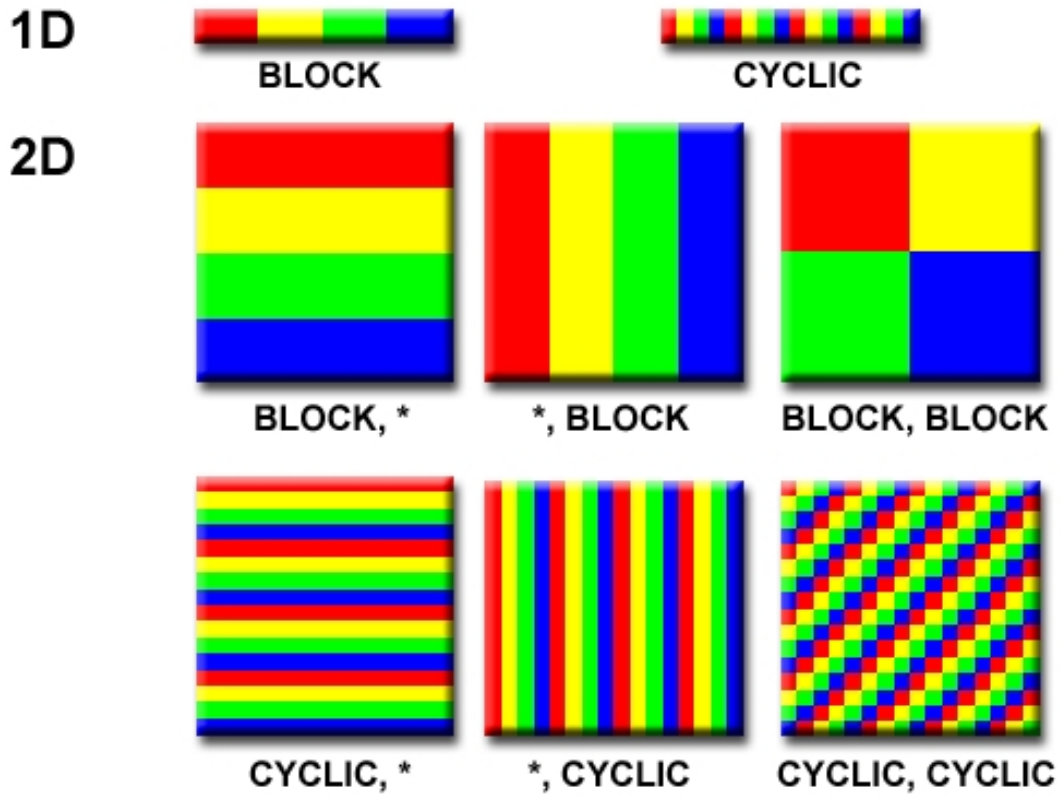
- Partitioning
  - One of the first steps in designing a parallel program is to break the problem into discrete chunks of work that can be distributed to multiple tasks - this is known as decomposition or partitioning
  - There are two basic ways to partition computational work among parallel tasks: domain decomposition and functional decomposition
  - Domain decomposition

    In this type of partitioning, the data associated with a problem is decomposed

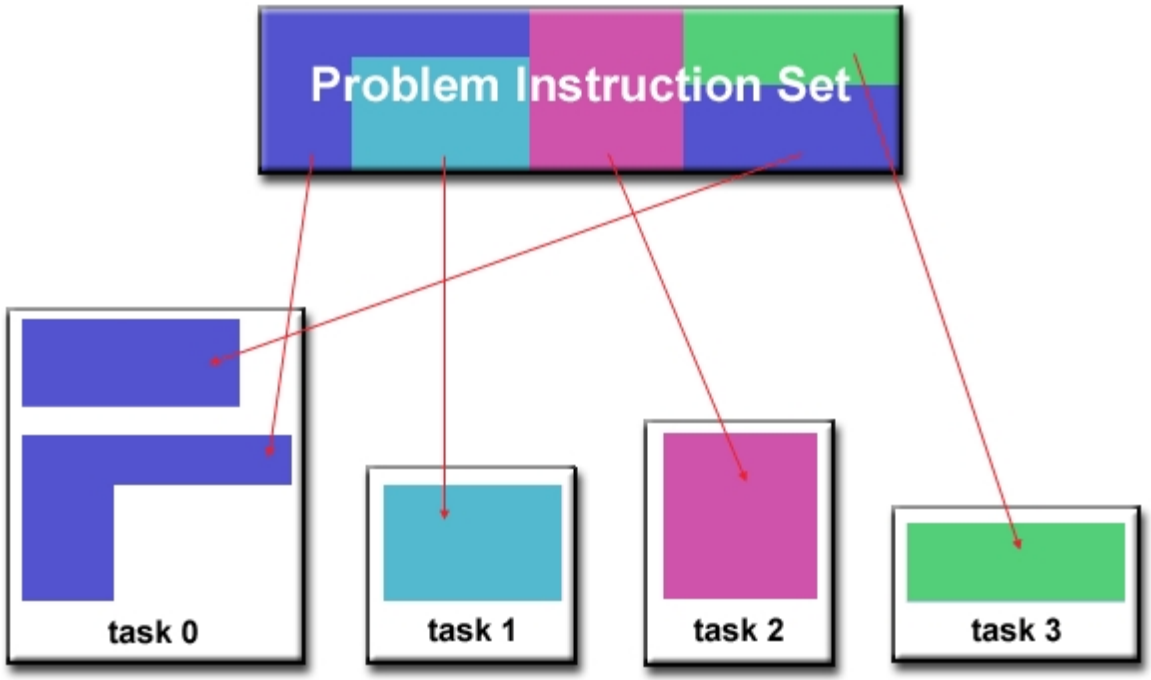
    Each parallel task then works on a portion of of the data



- Partitioning: domain decomposition
  - There are different ways to partition data



- Partitioning: 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 and each task then performs a portion of the overall work

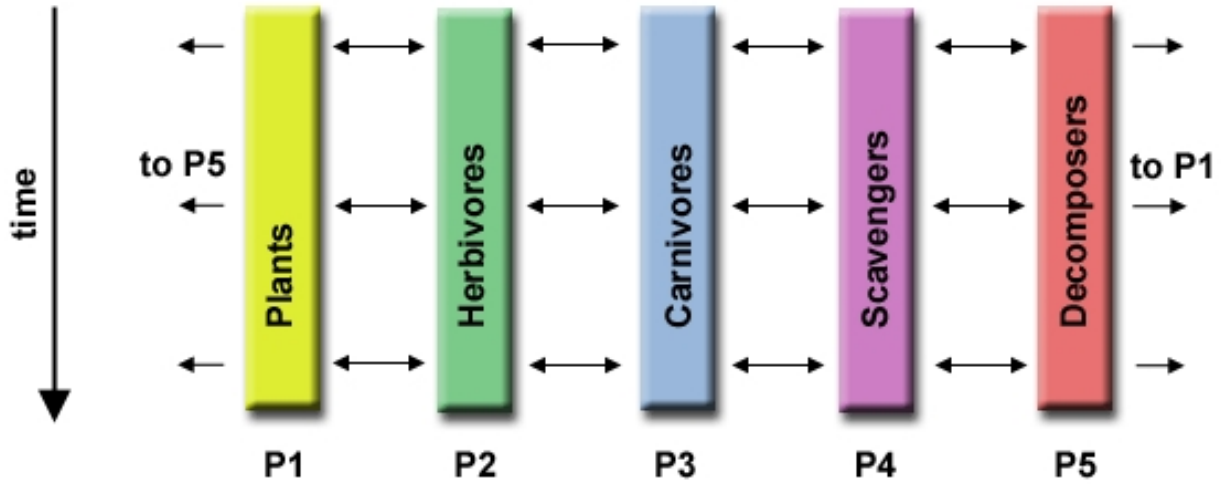


- Partitioning: functional decomposition
  - Functional decomposition lends itself well to problems that can be split into different tasks
  - Ecosystem modeling

    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.



- Partitioning: functional decomposition
  - Functional decomposition lends itself well to problems that can be split into different tasks
  - Signal processing

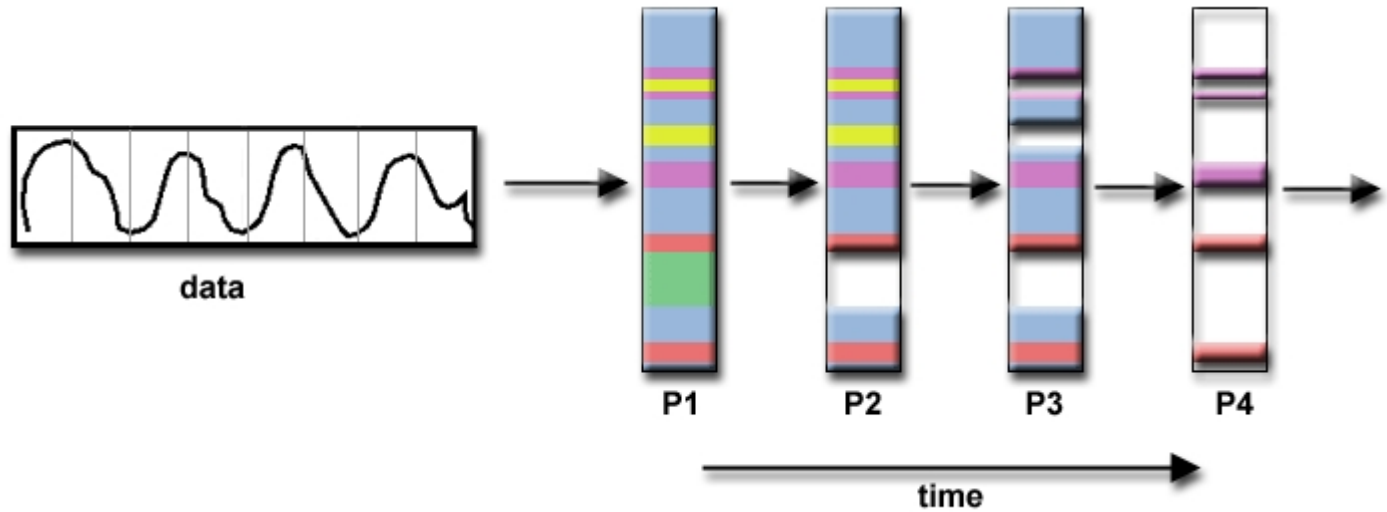
    An audio signal data set is passed through four distinct computational filters.

    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.

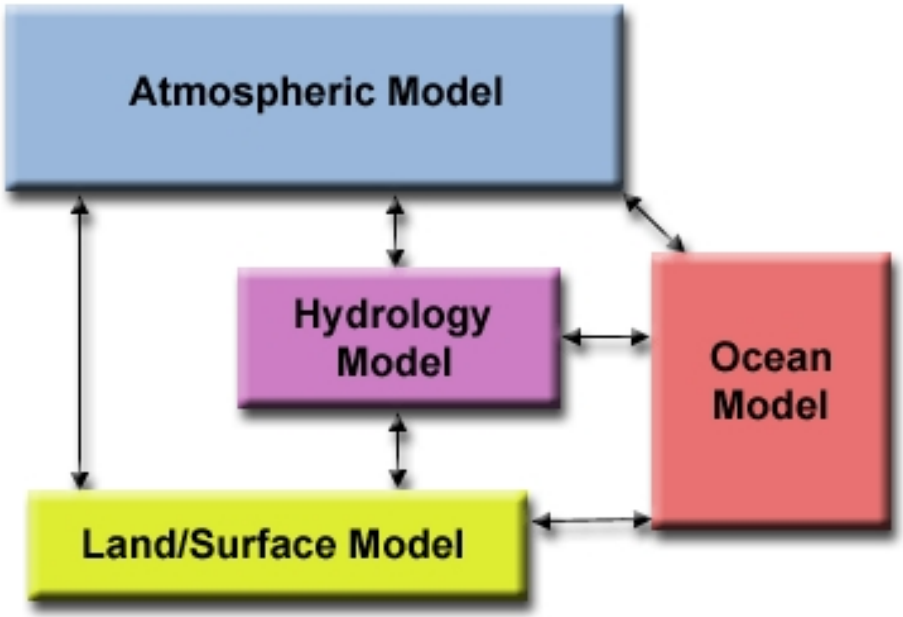
    By the time the fourth segment of data is in the first filter, all four tasks are busy.



- Partitioning: functional decomposition
  - Functional decomposition lends itself well to problems that can be split into different tasks
  - 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.



- 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.

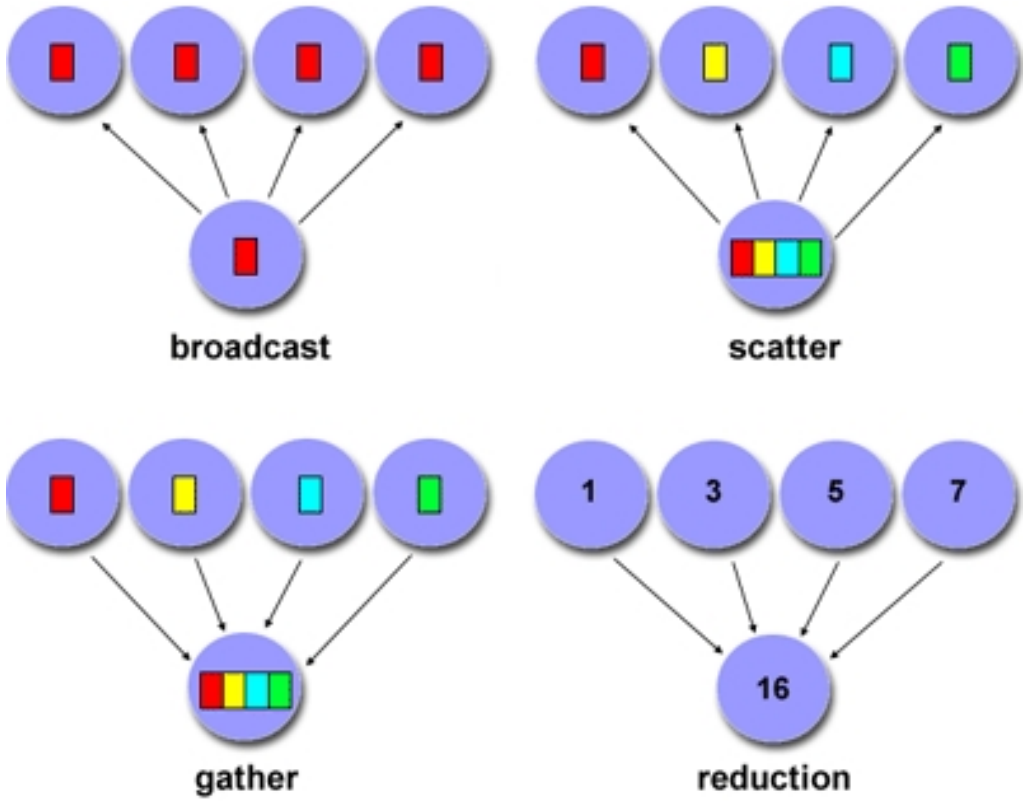
- Cost of communications
  - Inter-task communication virtually always implies overhead
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data
  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time waiting instead of doing work
  - Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems
- Latency vs. bandwidth
  - Latency is the time it takes to send a minimal (0 byte) message from point A to point B (commonly expressed as microseconds)
  - Bandwidth is the amount of data that can be communicated per unit of time (commonly expressed as megabytes/sec or gigabytes/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

- 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, further aggravating performance problems
- Latency vs. bandwidth
  - Latency is the time it takes to send a minimal (0 byte) message from point A to point B (commonly expressed as microseconds)
  - Bandwidth is the amount of data that can be communicated per unit of time (commonly expressed as megabytes/sec or gigabytes/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 even be able to know exactly how inter-task communications are being 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 communications since other work must wait until the communications have completed
  - Asynchronous communications allow tasks to transfer data independently from one another (for example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work; when task 2 actually 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

- 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 even be able to know exactly how inter-task communications are being 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 communications since other work must wait until the communications have completed
  - Asynchronous communications allow tasks to transfer data independently from one another (for example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work; when task 2 actually 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

- Scope of communications
  - Knowing which tasks must communicate with each other is critical during the design stage of a parallel code
  - 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



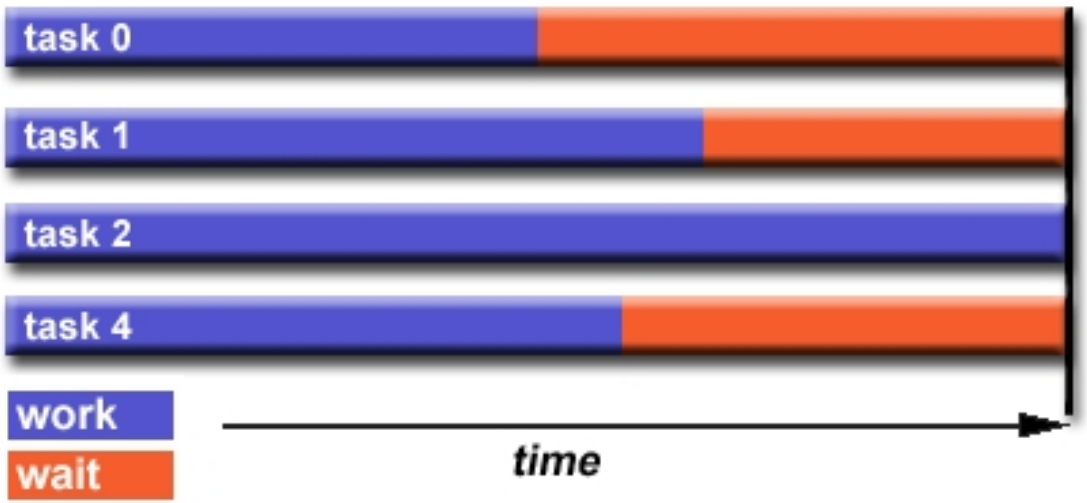
- Barrier
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier, and then stops, or blocks
  - When the last task reaches the barrier, all tasks are synchronized
- Lock/semaphore
  - Can involve any number of tasks
  - Typically used to serialize (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 and 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
- 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)

- Barrier
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier, and then stops, or blocks
  - When the last task reaches the barrier, all tasks are synchronized
- 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 and 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
- 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)

- Barrier
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier, and then stops, or blocks
  - When the last task reaches the barrier, all tasks are synchronized
- 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 and 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
- 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)

- Definition
  - 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
- 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 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 balance ?
  - 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 and adjust work accordingly.

  - Use dynamic work assignment

    Certain classes of problems result in load imbalances even if data is evenly distributed among tasks e.g. 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), etc.

    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.

- How to achieve load balance ?
  - 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 and adjust work accordingly.

  - Use dynamic work assignment

    Certain classes of problems result in load imbalances even if data is evenly distributed among tasks e.g. 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), etc.

    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.

- 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.

- 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.

- 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

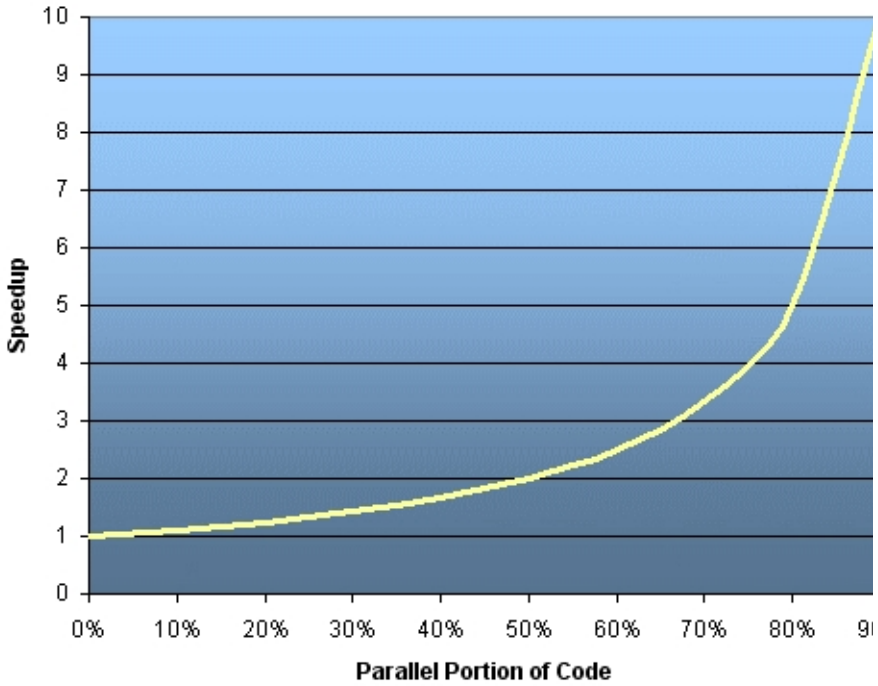
- Parallel speedup : $\frac{T_s}{T_p}$
- Amdahl's law
  - Amdahl's law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

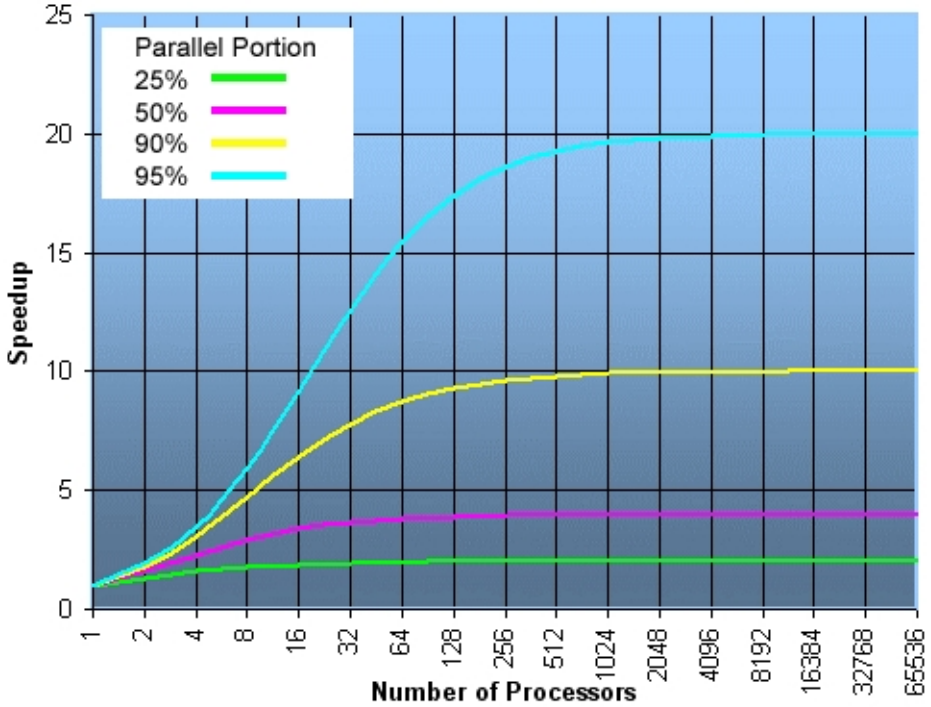
$$\text{Parallel speedup} = \frac{1}{1-P}.$$

  - 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:

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

    where P = parallel fraction, N = number of processors and S = serial fraction.
  - It soon becomes obvious that there are limits to the scalability of parallelism


Speedup
Parallel Portion of Code
0
1
2
3
4
5
6
7
8
9
10
0%
10%
20%
30%
40%
50%
60%
70%
80%


Parallel Portion
25%
50%
90%
95%
Speedup
Number of Processors
0
5
10
15
20
25
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536