**INSTRUCTOR HANDOUTS**

Computing: Computing is any goal-oriented activity requiring, benefiting from, or creating computing machinery. It includes the study and experimentation of algorithmic processes and development of both hardware and software. It has scientific, engineering, mathematical, technological and social aspects.

# History of Computing

All the generation

* Vacuumed Tube
* Transistor
* Integrated Circuits
* Silicon Chips
* AI

**Moors Law**

Moore's Law refers to Gordon Moore's perception that the number of transistors on a microchip

doubles every two years, though the cost of computers is halved. Moore's Law states that we can

expect the speed and capability of our computers to increase every couple of years, and we will

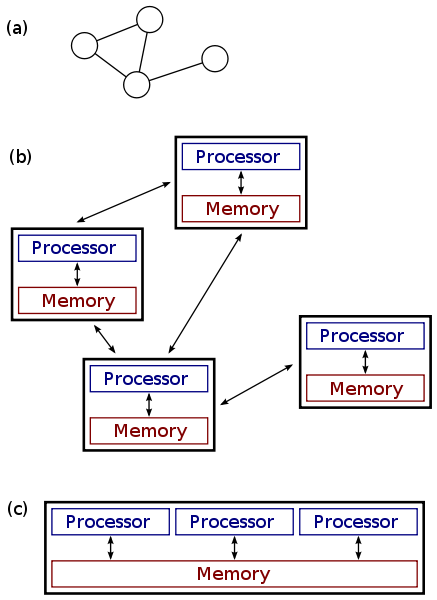
pay less for them

* **Fundamentals Of Parallel & Distributed Computing**

While both distributed computing and parallel systems are widely available these days, the main difference between these two is that a parallel computing system consists of multiple processors that communicate with each other using a shared memory, whereas a distributed computing system contains multiple processors connected by a communication network.

* **Computing Types**
  + Traditional Computing Environment
  + Cloud Computing
  + Grid Computing
  + Distributed Computing
  + Cluster Computing
  + Personal Computing
  + Time-Sharing Computing
  + Client-Server Computing
  + Peer-to-Peer Computing
  + Mobile Computing
* Pros & Cons of Parallel & Distributed Computing

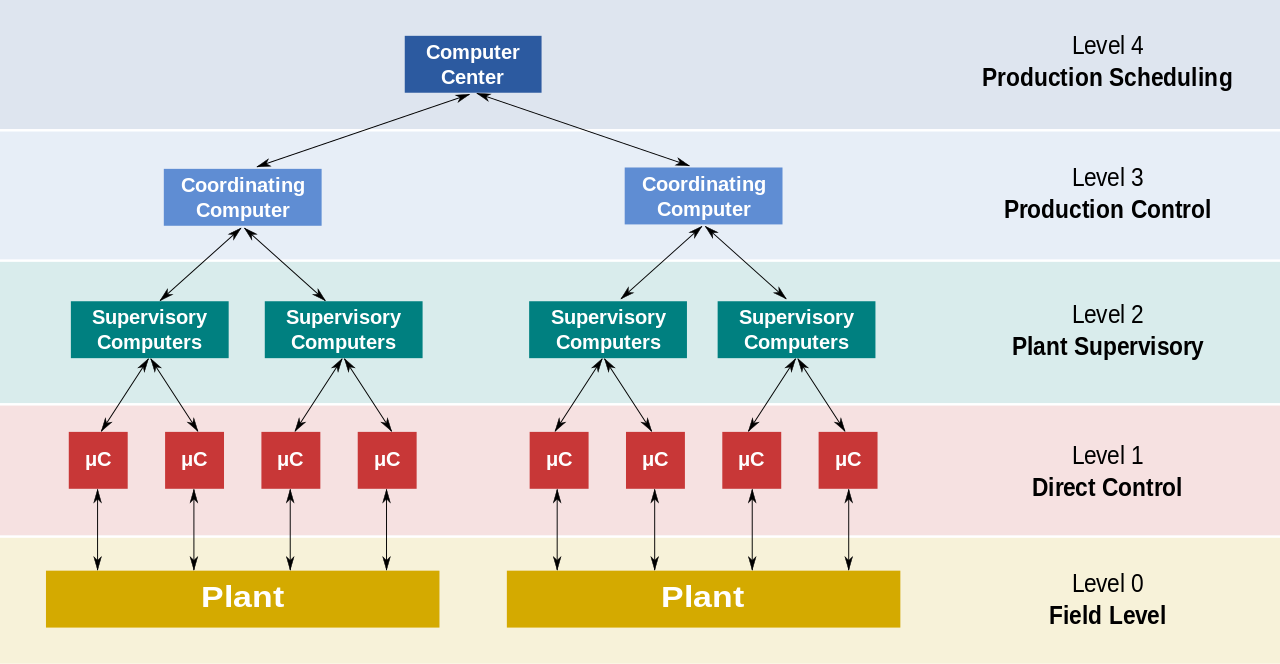
Parallel computing is also called **parallel processing**. There are multiple processors in parallel computing. Each of them performs the computations assigned to them. In other words, in parallel computing, multiple calculations are performed simultaneously. The systems that support parallel computing can have a shared [memory](https://pediaa.com/difference-between-register-and-main-memory/) or distributed memory. In shared memory systems, all the processors share the memory. In distributed memory systems, memory is divided among the processors.



There are multiple advantages to parallel computing. As there are multiple processors working simultaneously, it increases the [CPU](https://pediaa.com/difference-between-cpu-and-core/#CPU) utilization and improves the performance. Moreover, failure in one processor does not affect the functionality of other processors. Therefore, parallel computing provides reliability. On the other hand, increasing processors is costly. Furthermore, if one processor requires instructions of another, the processor might cause latency.

## **What is Distributed Computing**

Distributed computing divides a single task between multiple computers. Each computer can communicate with others via the network. All computers work together to achieve a common goal. Thus, they all work as a single entity. A computer in the distributed system is a node while a collection of nodes is a cluster.



There are multiple advantages of using distributed computing. It allows scalability and makes it easier to share resources easily. It also helps to perform computation tasks efficiently. On the other hand, it is difficult to develop distributed systems. Moreover, there can be network issues.

## **Difference between Parallel and Distributed Computing**

### **Definition**

Parallel computing is a type of computation in which many calculations or execution of processes are carried out simultaneously. Whereas, a distributed system is a system whose components are located on different networked computers which communicate and coordinate their actions by passing messages to one another. Thus, this is the fundamental difference between parallel and distributed computing.

### **Number of computers**

The number of computers involved is a difference between parallel and distributed computing. Parallel computing occurs in a single computer whereas distributed computing involves multiple computers.

### **Functionality**

In parallel computing, multiple processors execute multiple tasks at the same time. However, in distributed computing, multiple computers perform tasks at the same time. Hence, this is another difference between parallel and distributed computing.

### **Memory**

Moreover, memory is a major difference between parallel and distributed computing. In parallel computing, the computer can have a shared memory or distributed memory. In distributed computing, each computer has its own memory.

### **Communication**

Also, one other difference between parallel and distributed computing is the method of communication. In parallel computing, the processors communicate with each other using a bus. In distributed computing, computers communicate with each other via the network.

### **Usage**

Parallel computing helps to increase the performance of the system. In contrast, distributed computing allows scalability, sharing resources and helps to perform computation tasks efficiently. So, this is also a difference between parallel and distributed computing.

## **What is a Computing Environment?**

While solving a problem, a computer uses multiple devices. During which the devices can be arranged in a number of ways to work together towards the solution. As a result, the devices and components constitute a computing environment.

Moreover, a computing environment organizes components to exchange information. Hence, it uses various devices to process and solve different problems.

Additionally, a single computing environment includes multiple computers, computational devices, software, and networks. Therefore, these components collaborate to support processing and sharing information to solve various tasks and problems.

### Features of a Robust Computing Environment

* Firstly, a computing environment has the ability to dynamically choose machines to run tasks.
* Moreover, it also has the feature to migrate processes from one environment to another.
* Additionally, it offers support for fault tolerance.
* Further, multiple tasks may be mapped to multiple devices and components for faster delivery.
* Consequently, communication between tasks occurs during primitives. Hence, it is an ongoing process mapped to the communication tools in the environment.
* It certainly uses channel-based systems to monitor, redirect, and shift connections during tasks.
* Moreover, it also uses the components to achieve multiple or conflicting goals.
* Furthermore, it also schedules the work on idle machines to maximize utilization.
* Additionally, it also executes tasks on the best available platforms. As a result, selecting devices that are well-equipped to run the tasks at hand.

## Types of Computing Environments Businesses should learn about:

### Traditional Computing

Traditional Computing is a process of using physical data centers for various data assets. As a result, it also runs complete networking systems for day-to-day operations.

However, access to data, software, and storage is limited to users and devices. Hence, it only allows access to authorized devices that connect to the official network.

Therefore, it limits the users to only access the data from the system that stores it.

Moreover, users processes and systems processes provide services to a user. As a result, it mansges the tasks frequently for optimization of the computer time.

For example, Windows is created, while a user is running another task on the computer. Therefore, the environment is allowing users to simultaneously execute different tasks.

### **Cloud Computing**

Cloud Computing is the combination of configurable components. Moreover, components like system resources and advanced services help deliver tasks using internet connections.

Further, it runs tasks on third-party servers and enables the ability to access data from multiple locations. It also provides a cost-efficient solution and is more user-friendly.

Above all, it offers more storage space, servers, and computing power to help the apps run efficiently and smoothly. Moreover, it only requires fast, eligible, and stable internet connections to execute tasks.

### **Grid Computing**

Grid Computing is a process where computers and devices from various locations work on a single problem. Further, in this system clusters jointly execute given tasks. As a result, it applies resources from multiple computers and nodes.

Therefore, it is a type of computing environment that utilizes several and scattered resources. Hence, these resources provide a functioning environment for executing a single task.

### **Distributed Computing**

Distributed Computing takes place when multiple computers and devices connect using a common network but are separated physically. As a result, a single task is performed by various functional units of different and distributed nodes and units.

Simultaneously, different programs of an application run on separate nodes. Therefore, communication takes place between different nodes of a system over the network to execute the task.

### **Cluster Computing**

In this type of computing environment, clusters execute tasks. Cluster Computing allows clusters to work as a set of loosely or tightly connected computers.

Consequently, it is viewed as a single system and executes tasks parallelly. Hence, it is also similar to a parallel type of computing environment.

As a result, the cluster computing environment prefers cluster-aware applications.

### **Personal Computing**

A Personal Computing Environment includes a single machine. Moreover, it incorporates complete programs on a computer and performs it.

For example, machines like laptops, mobiles, printers, etc are a part of the Personal Computing Environment. As a result, this type of computing environment is for single users to run tasks at home or offices.

### **Time-Sharing Computing**

A Time-Sharing Computing Environment enables multiple users to share a system concurrently. Furthermore, it allows various time slots for various users and processes. Hence, the processor switches rapidly and changes users according to their slots.

For example, Windows 95 and its later versions, Unix, IOS, Linux OS all run on the time-sharing computing environment.

### **Client-Server Computing**

Client-Server Computing is a type of environment that incorporates two machines. Therefore, it includes a client machine and a server machine. Sometimes, the same machine serves as the client and the server.

Subsequently, a client requests a resource or service and a server provides the same. Moreover, a server provides a resource or service to multiple clients simultaneously. Hence, the communication takes place using a computer network.

**Categorization of Client-Server Computing Environment is into two types:**

* Computer Server: It provides the interface to the clients. Hence, it helps communicate requests to execute tasks.  
  Meanwhile, the server performs the task and responds with the outcome.
* File-Server: The environment provides a file-system interface. Therefore, allowing clients to create, update, read, and delete files.

### **Peer-to-Peer Computing**

Peer-to-Peer Computing is a type of environment similar to a Distributed type of Computing Environment. That is to say, there are no differences between clients and servers in this type of computing environment.

P2P provides an advantage over traditional client-server environments. That is to say, it provides services using several nodes throughout the network.

### **Mobile Computing**

Mobile Computing refers to the type of environment that runs tasks on smartphones and tablets. Hence, it is computing on portable and lightweight devices.

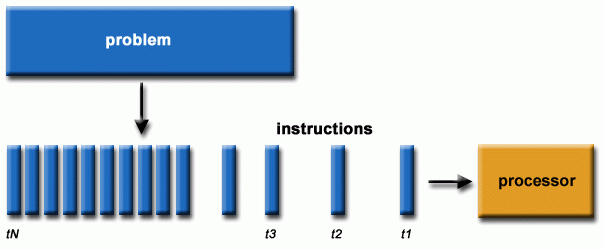
Although, compared to other devices, mobile systems lack screen size, memory capacity, and other traditional functionalities. However, it does provide remote access to multiple services.

Today, mobile computing environments consist of multiple functions. Hence, it offers services as good as any other traditional device. Moreover, the two main operating systems that dominate this market are Apple iOS and Google Android.

#### **Serial Computing**

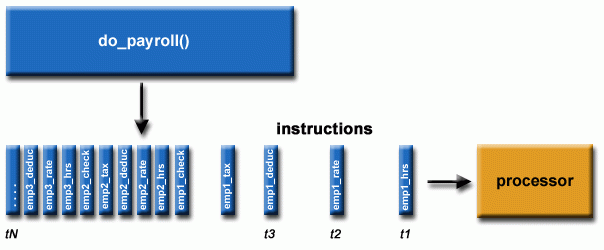
Traditionally, software has been written for **serial** computation:

* A problem is broken into a discrete series of instructions
* Instructions are executed sequentially one after another
* Executed on a single processor
* Only one instruction may execute at any moment in time



Serial computing generic example

**For example:**

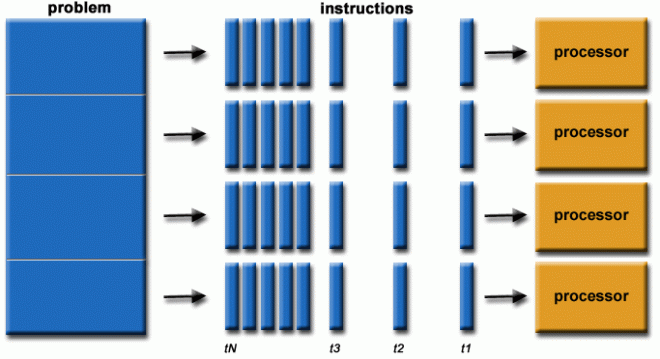


Serial computing example of processing payroll

#### **Parallel Computing**

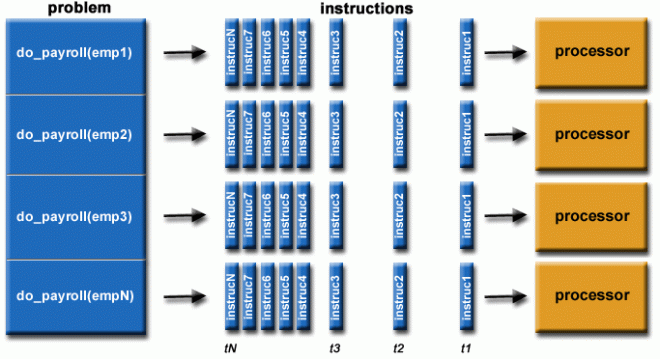
In the simplest sense, **parallel computing** is the simultaneous use of multiple compute resources to solve a computational problem:

* 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 processors
* An overall control/coordination mechanism is employed



Parallel computing generic example

**For example:**

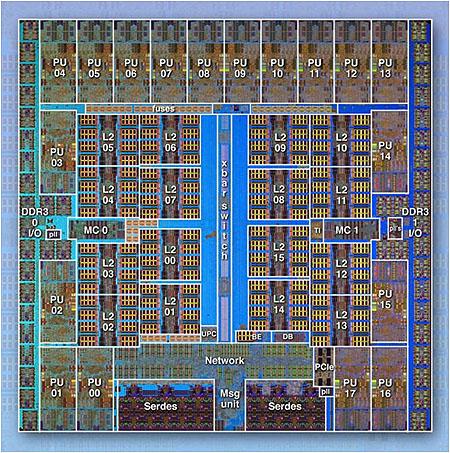


Parallel computing example of processing payroll

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

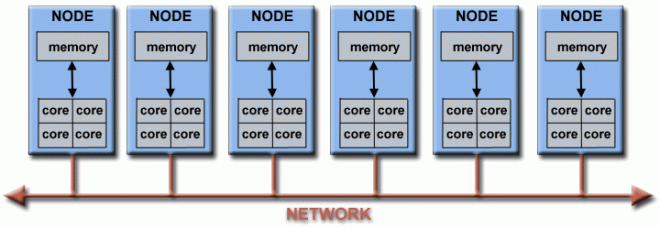
#### **Parallel Computers**

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



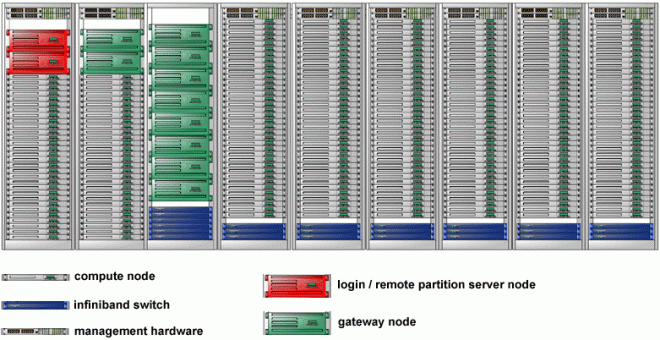
IBM BG/Q Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

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



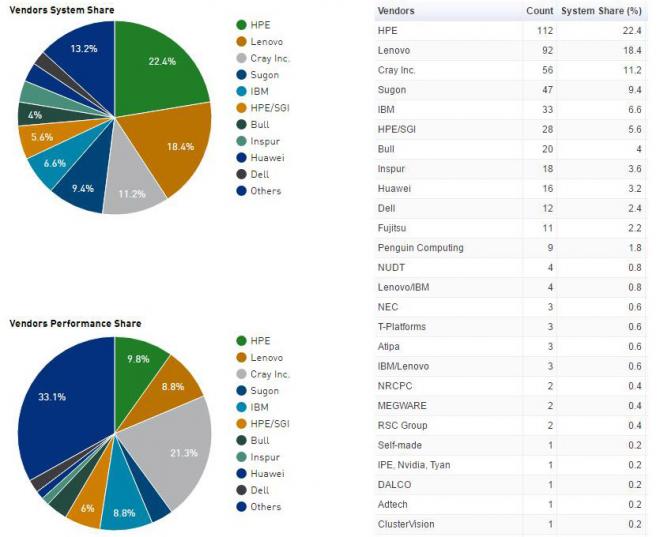
Network connections

* For example, the schematic below shows a typical LLNL parallel computer cluster:
  + Each compute node is a multi-processor parallel computer in itself
  + Multiple compute nodes are networked together with an Infiniband network
  + Special purpose nodes, also multi-processor, are used for other purposes



Example of typical parallel computer cluster

* The majority of the world's large parallel computers (supercomputers) are clusters of hardware produced by a handful of (mostly) well known vendors.



Source: [*Top500.org*](https://www.top500.org/)

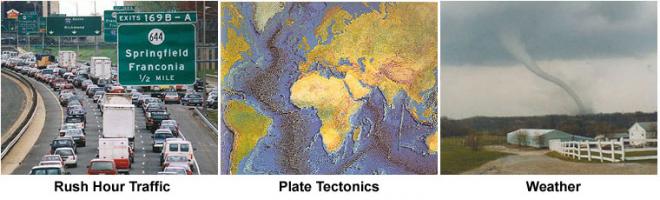
### **Why Use Parallel Computing?**

#### **The Real World Is Massively Complex**

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



Real world phenomena can be simulated with parallel computing



Real world phenomena can be simulated with parallel computing

#### **Main Reasons for Using Parallel Programming**

##### **SAVE TIME AND/OR MONEY**

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



Working in parallel shortens completion time

##### **SOLVE LARGER / MORE COMPLEX PROBLEMS**

* Many problems are so large and/or complex that it is impractical or impossible to solve them using a serial program, especially given limited computer memory.
* Example: "Grand Challenge Problems" ([en.wikipedia.org/wiki/Grand\_Challenge](https://en.wikipedia.org/wiki/Grand_Challenges)) requiring petaflops and petabytes of computing resources.
* Example: Web search engines/databases processing millions of transactions every second



Parallel computing can solve increasingly complex problems

##### **PROVIDE CONCURRENCY**

* A single compute resource can only do one thing at a time. Multiple compute resources can do many things simultaneously.
* Example: Collaborative Networks provide a global venue where people from around the world can meet and conduct work "virtually."



Collaborative networks

##### **TAKE ADVANTAGE OF NON-LOCAL RESOURCES**

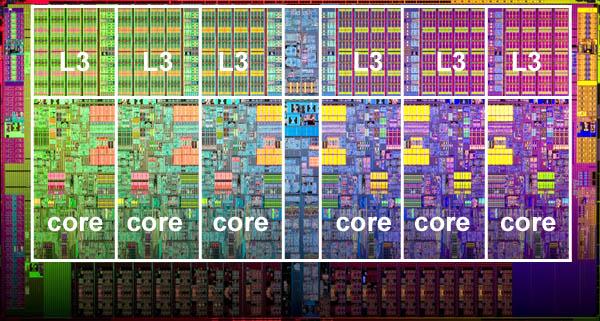
* Using compute resources on a wide area network, or even the Internet when local compute resources are scarce or insufficient.
* Example: SETI@home ([setiathome.berkeley.edu](http://setiathome.berkeley.edu/)) has over 1.7 million users in nearly every country in the world (May, 2018).



SETI has a large worldwide user base

##### **MAKE BETTER USE OF UNDERLYING PARALLEL HARDWARE**

* Modern computers, even laptops, are parallel in architecture with multiple processors/cores.
* Parallel software is specifically intended for parallel hardware with multiple cores, threads, etc.
* In most cases, serial programs run on modern computers "waste" potential computing power.

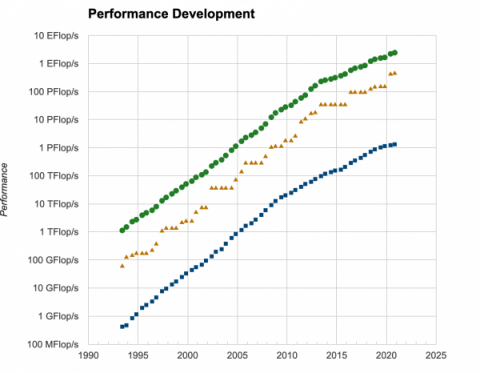


Parallel computing cores

#### **The Future**

* During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that **parallelism is the future of computing**.
* In this same time period, there has been a greater than **500,000x** increase in supercomputer performance, with no end currently in sight.
* **The race is already on for Exascale Computing - we are entering Exascale era**
  + Exaflop = 1018 calculations per second
  + US DOE Exascale Computing Project: [https://www.exascaleproject.org](https://www.exascaleproject.org/)

Image



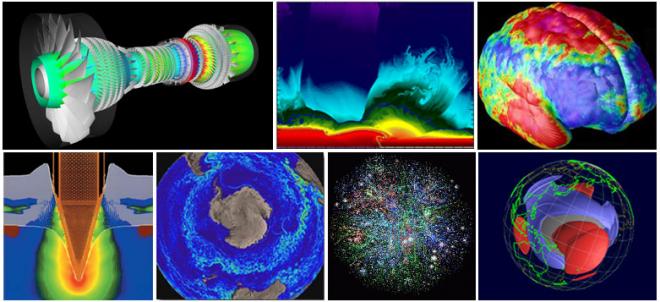
Source: [*Top500.org*](https://www.top500.org/)

### **Who Is Using Parallel Computing?**

#### **Science and Engineering**

Historically, parallel computing has been considered to be "the high end of computing," and has been used to model difficult problems in many areas of 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
* Defense, Weapons

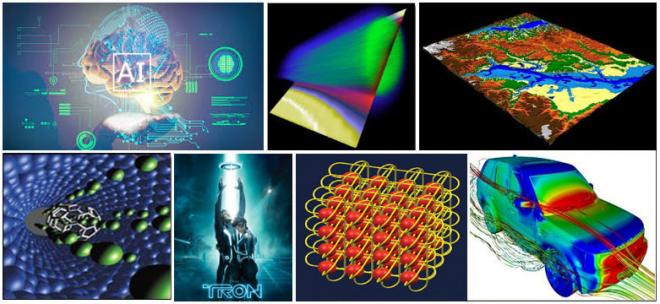


Parallel computing is key to simulating a range of complex physical phenomena

#### **Industrial and Commercial**

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. For example:

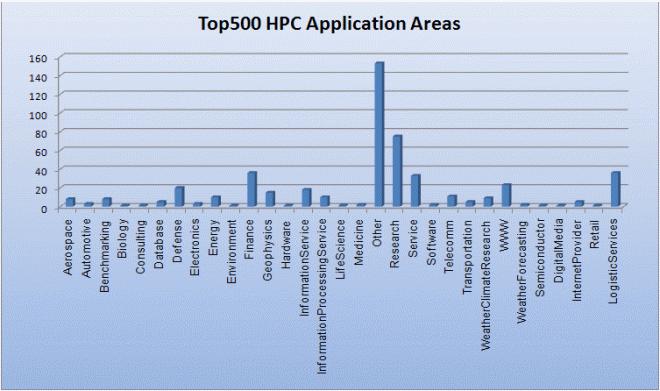
* "Big Data," databases, data mining
* Artificial Intelligence (AI)
* Oil exploration
* Web search engines, web based business services
* Medical imaging and diagnosis
* Pharmaceutical design
* Financial and economic modeling
* Management of national and multi-national corporations
* Advanced graphics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



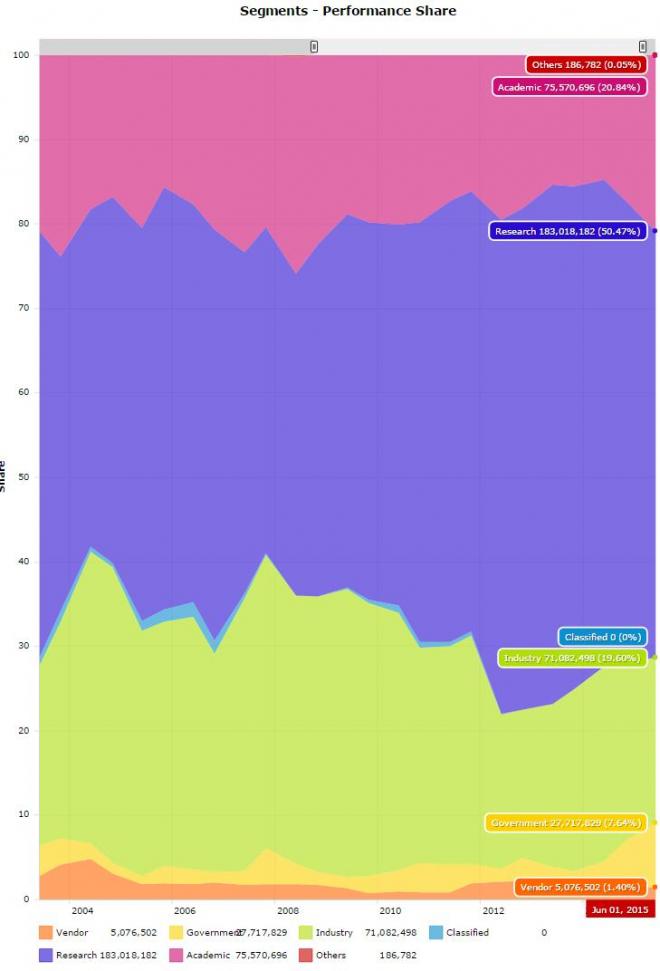
Parallel computing is used in many commercial applications

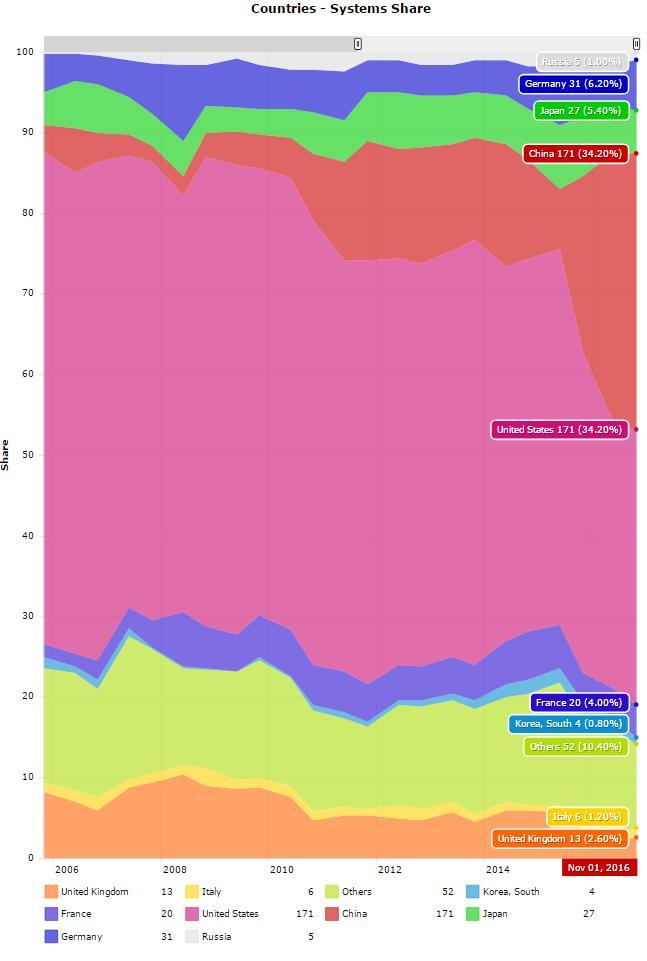
#### **Global Applications**

* Parallel computing is now being used extensively around the world, in a wide variety of applications.



Source: [*Top500.org*](https://www.top500.org/)





Source: [*Top500.org*](https://www.top500.org/)

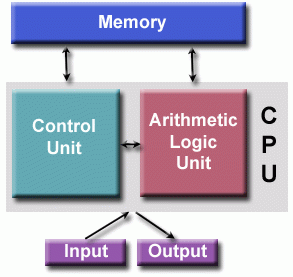
## **Concepts and Terminology**

### **von Neumann Computer Architecture**



John von Neumann circa 1940s(Source: LANL archives)

* Named after the Hungarian mathematician John von Neumann who first authored the general requirements for an electronic computer in his 1945 papers.
* Also known as "stored-program computer" - both program instructions and data are kept in electronic memory. Differs from earlier computers which were programmed through "hard wiring".
* Since then, virtually all computers have followed this basic design:



Basic computing architecture

* Comprised of four main components:

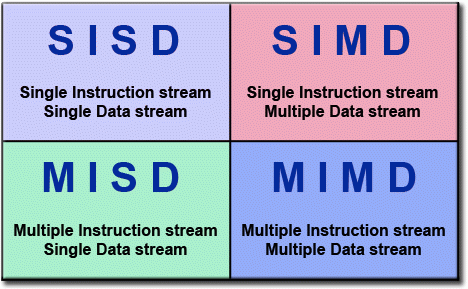
1. Memory
2. Control Unit
3. Arithmetic Logic Unit
4. Input/Output

* 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.
* Arithmetic Unit performs basic arithmetic operations
* Input/Output is the interface to the human operator

Parallel computers still follow this basic design, just multiplied in units. The basic, fundamental architecture remains the same. More info on his other remarkable accomplishments: <http://en.wikipedia.org/wiki/John_von_Neumann>

### **Flynn's Classical Taxonomy**

* There are a number of [different ways](https://hpc.llnl.gov/sites/default/files/parallelClassifications_0.pdf) to classify parallel computers. Examples are available in the references.
* One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
* Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of**Instruction Stream** and ***Data Stream***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.
* The matrix below defines the 4 possible classifications according to Flynn:



Flynn's taxonomy

#### **Single Instruction, Single Data (SISD)**

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

|  |  |  |
| --- | --- | --- |
| Image  SISD diagram |  | Image  SISD diagram |
| IBM 360Image  UNIVAC1 | Image | Image  CRAY1 |
| PDP 1Image  CDC 7600 | Image | Image  Dell Laptop |

#### **Single Instruction, Multiple Data (SIMD)**

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

|  |  |  |
| --- | --- | --- |
| SIMD diagramImage  SIMD diagram | Image | Image  SIMD diagram |
| Mas ParImage  ILLIAC IVCell Processor (GPU) | Image | Image |
| Cray Y-MPImage  Cray X-MP |  | Image  Thinking Machines CM-2 |

#### **Multiple Instruction, Single Data (MISD)**

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

|  |  |
| --- | --- |
| Image  MISD diagramMISD diagram | Image |

#### **Multiple Instruction, Multiple Data (MIMD)**

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

|  |  |  |  |
| --- | --- | --- | --- |
| Image  MIMD diagram |  | Image  MIMD diagram | |
| Image  IBM POWER5 | Image  HP/Compaq Alphaserver | Image  Intel IA32 |  |
| Image  AMD Opteron | Image  Cray XT3 | Image  IBM BG/L |  |

### **General Parallel Computing Terminology**

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

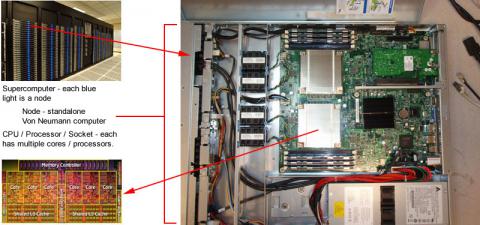
##### **CPU**

Contemporary CPUs consist of one or more cores - a distinct execution unit with its own instruction stream. Cores with a CPU may be organized into one or more sockets - each socket with its own distinct memory . When a CPU consists of two or more sockets, usually hardware infrastructure supports memory sharing across sockets.

##### **Node**

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

Image



Nodes in a supercomputer

##### **Task**

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

##### **Pipelining**

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

##### **Shared Memory**

Describes a computer architecture where all processors have direct access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

##### **Symmetric Multi-Processor (SMP)**

Shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources - memory, disk, etc.

##### **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.

##### **Synchronization**

The coordination of parallel tasks in real time, very often associated with communications.

Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

##### **Computational Granularity**

In parallel computing, granularity is a quantitative or 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**

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

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

##### **Massively Parallel**

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

##### **Embarrassingly (IDEALY) Parallel**

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

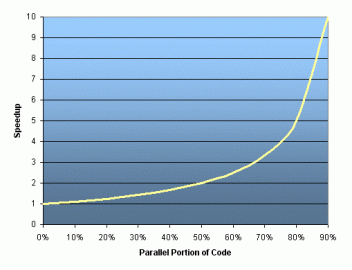
##### **Scalability**

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

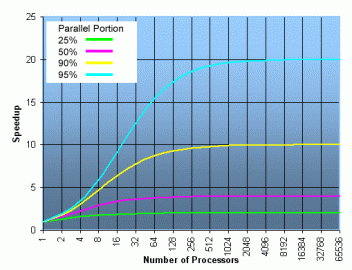
* Hardware - particularly memory-cpu bandwidths and network communication properties
* Application algorithm
* Parallel overhead related
* Characteristics of your specific application

### **Potential Benefits, Limits and Costs of Parallel Programming**

#### **Amdahl's Law**



Amdahl's law



Speedup when introducing more processors

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

1

speedup = --------

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:

1

speedup = ------------

P + S

---

N

* 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. For example:

speedup

-------------------------------------

N P = .50 P = .90 P = .95 P = .99

----- ------- ------- ------- -------

10 1.82 5.26 6.89 9.17

100 1.98 9.17 16.80 50.25

1,000 1.99 9.91 19.62 90.99

10,000 1.99 9.91 19.96 99.02

100,000 1.99 9.99 19.99 99.90

* **"Famous" quote:** You can spend a lifetime getting 95% of your code to be parallel, and never achieve better than 20x speedup no matter how many processors you throw at it!
* However, certain problems demonstrate increased performance by increasing the problem size. For example:

2D Grid Calculations

Parallel fraction 85 seconds 85%

Serial fraction 15 seconds 15%

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

2D Grid Calculations

Parallel fraction 680 seconds 97.84%

Serial fraction 15 seconds 2.16%

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

#### **Complexity**

* In general, parallel applications are  more complex than corresponding serial applications. Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.
* The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle:
  + Design
  + Coding
  + Debugging
  + Tuning
  + Maintenance
* Adhering to "good" software development practices is essential when developing  parallel applications.

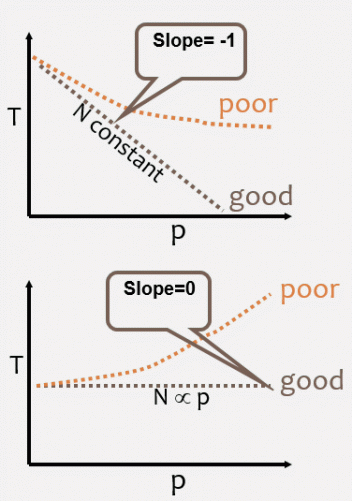
#### **Portability**

* Thanks to standardization in several APIs, such as MPI, OpenMP and POSIX threads, portability issues with parallel programs are not as serious as in years past. However...
* All of the usual portability issues associated with serial programs apply to parallel programs. For example, if you use vendor "enhancements" to Fortran, C or C++, portability will be a problem.
* Even though standards exist for several APIs, implementations will differ in a number of details, sometimes to the point of requiring code modifications in order to effect portability.
* Operating systems can play a key role in code portability issues.
* Hardware architectures are characteristically highly variable and can affect portability.

#### **Resource Requirements**

* The primary intent of parallel programming is to decrease execution wall clock time, however in order to accomplish this, more CPU time is required. For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time.
* The amount of memory required can be greater for parallel codes than serial codes, due to the need to replicate data and for overheads associated with parallel support libraries and subsystems.
* For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation. The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.

#### **Scalability**



Strong and weak scaling

* Two types of scaling based on time to solution: strong scaling and weak scaling.
* **Strong scaling (Amdahl):**
  + The total problem size stays fixed as more processors are added.
  + Goal is to run the same problem size faster
  + Perfect scaling means problem is solved in 1/P time (compared to serial)
* **Weak scaling (Gustafson):**
  + The problem size per processor stays fixed as more processors are added. The total problem size is proportional to the number of processors used.
  + Goal is to run larger problem in same amount of time
  + Perfect scaling means problem Px runs in same time as single processor run
* The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more processors is rarely the answer.
* The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. This is a common situation with many parallel applications.
* Hardware factors play a significant role in scalability. Examples:
  + Memory-cpu bus bandwidth on an SMP machine
  + Communications network bandwidth
  + Amount of memory available on any given machine or set of machines
  + Processor clock speed
* Parallel support libraries and subsystems software can limit scalability independent of your application.

## **Parallel Computer Memory Architectures**

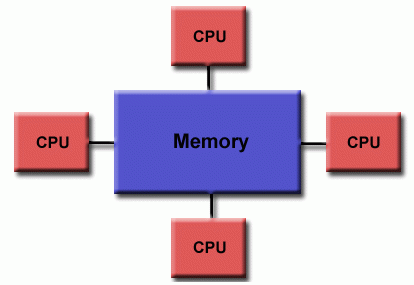
### **Shared Memory**

#### **General Characteristics**

* Shared memory parallel computers vary widely, but generally have in common the ability for all processors to access all memory as global address space.
* Multiple processors can operate independently but share the same memory resources.
* Changes in a memory location effected by one processor are visible to all other processors.
* Historically, shared memory machines have been classified as ***UMA*** and ***NUMA***, based upon memory access times.

#### **Uniform Memory Access (UMA)**

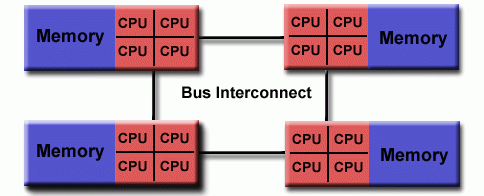
* Most commonly represented today by ***Symmetric Multiprocessor (SMP)*** machines
* Identical processors
* Equal access and access times to memory
* Sometimes called CC-UMA - Cache Coherent UMA. Cache coherent means if one processor updates a location in shared memory, all the other processors know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



Uniform memory access

#### **Non-Uniform Memory Access (NUMA)**

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



Non-uniform memory access

#### **Advantages**

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

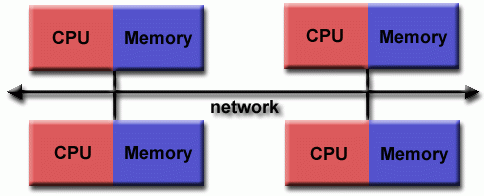
#### **Disadvantages**

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

### **Distributed Memory**

#### **General Characteristics**

* Like shared memory systems, distributed memory systems vary widely but share a common characteristic. Distributed memory systems require a communication network to connect inter-processor memory.
* Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
* Because each processor has its own local memory, it operates independently. Changes it makes to its local memory have no effect on the memory of other processors. Hence, the concept of cache coherency does not apply.
* When a processor needs access to data in another processor, it is usually the task of the programmer to explicitly define how and when data is communicated. Synchronization between tasks is likewise the programmer's responsibility.
* The network "fabric" used for data transfer varies widely, though it can be as simple as Ethernet.



Distributed memory

#### **Advantages**

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

#### **Disadvantages**

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

### **Hybrid Distributed-Shared Memory**

#### **General Characteristics**

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

|  |  |
| --- | --- |
| Image  Diagram of hybrid distributed-shared memoryDiagram of hybrid distributed-shared memory | Image |

* The shared memory component can be a shared memory machine and/or graphics processing units (GPU).
* The distributed memory component is the networking of multiple shared memory/GPU machines, which know only about their own memory - not the memory on another machine. Therefore, network communications are required to move data from one machine to another.
* Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.

#### **Advantages and Disadvantages**

* Whatever is common to both shared and distributed memory architectures.
* Increased scalability is an important advantage
* Increased programmer complexity is an important disadvantage

## **Parallel Programming Models**

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

##### **SHARED memory model on a DISTRIBUTED memory machine**

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

|  |  |
| --- | --- |
| Image  Shared memory model abstractionKSR approach networked machines | Image |

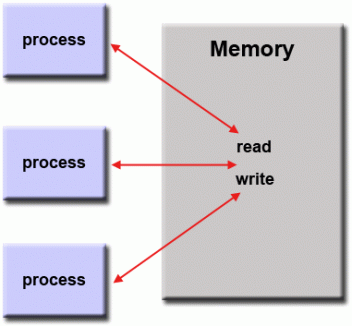
##### **DISTRIBUTED memory model on a SHARED memory machine**

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

|  |  |
| --- | --- |
| Image  Distributed memory model abstractionSGI Origin 2000 | Image |

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

### **Shared Memory Model (without threads)**



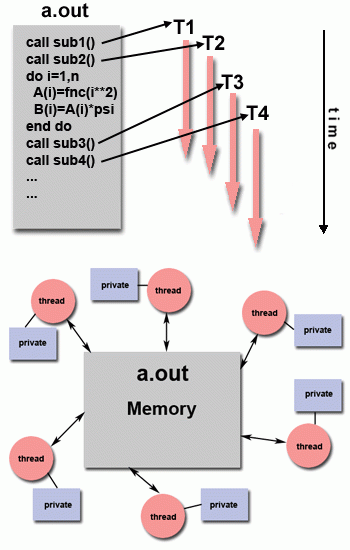
Shared memory model

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

**Implementations**

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

### **Threads Model**



Threads model

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

**Implementations**

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

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

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

##### **POSIX Threads**

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

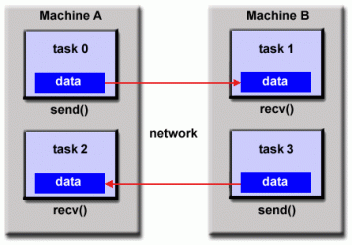
##### **OpenMP**

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

#### **More Information**

* POSIX Threads tutorial: [hpc.llnl.gov/sites/default/files/2019.08.21.TAU\_.pdf](https://hpc.llnl.gov/sites/default/files/2019.08.21.TAU_.pdf)
* OpenMP tutorial: [hpc-tutorials.llnl.gov/openmp/](https://hpc-tutorials.llnl.gov/openmp/)

### **Distributed Memory / Message Passing Model**



Distributed memory model

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

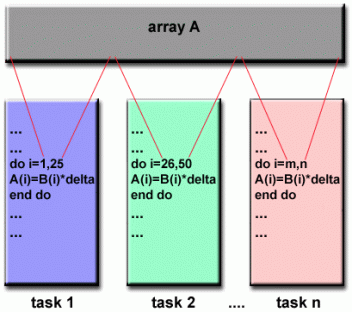
**Implementations:**

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

#### **More Information**

* MPI tutorial: [hpc-tutorials.llnl.gov/mpi/](https://hpc-tutorials.llnl.gov/mpi/)

### **Data Parallel Model**



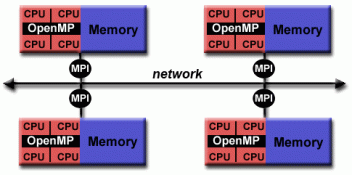
Data parallel model

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

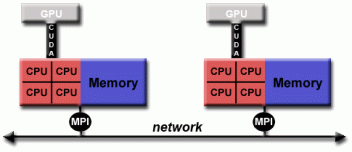
**Implementations:**

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

### **Hybrid Model**



Hybrid model with MPI and OpenMP

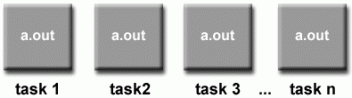


Hybrid model with MPI and CUDA

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

### **SPMD and MPMD**

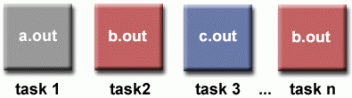
#### **Single Program Multiple Data (SPMD)**



SPMD model

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

#### **Multiple Program Multiple Data (MPMD)**



MPMD model

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

## **Designing Parallel Programs**

### **Automatic vs. Manual Parallelization**

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

##### **Fully Automatic**

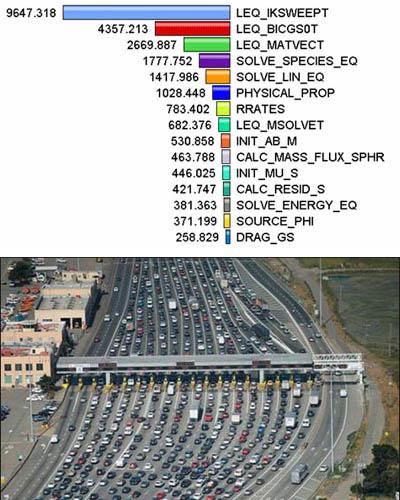
* The compiler analyzes the source code and identifies opportunities for parallelism.
* The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
* Loops (do, for) are the most frequent target for automatic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

##### **Programmer Directed**

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

### **Understand the Problem and the Program**

Programs = algorithms + data + (hardware)



Determine whether the problem can actually be parallelized

* Undoubtedly, the first step in developing parallel software is to first understand the problem that you wish to solve in parallel. If you are starting with a serial program, this means understanding the existing code also.
* Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.
  + Example of an easy-to-parallelize problem:

**Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.**

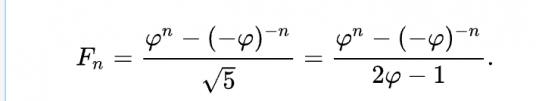
This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

* Example of a problem and algorithm with little-to-no parallelism:

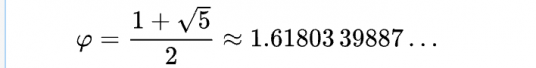
**Calculation of the first 10,000 members of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula:**  
**F(n) = F(n-1) + F(n-2)**

The calculation of the F(n) value uses those of both F(n-1) and F(n-2), which must be computed first.

An example of a parallel algorithm for solving this problem (using Binet's formula):



where



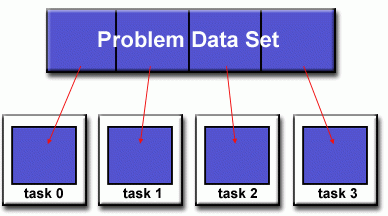
* Identify the program's **hotspots**:
  + Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
  + Profilers and performance analysis tools can help here
  + Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.
* Identify ***bottlenecks*** in the program:
  + Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
  + May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas
* Identify inhibitors to parallelism. One common class of inhibitor is data dependence, as demonstrated by the Fibonacci sequence above.
* Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.
* Take advantage of optimized third party parallel software and highly optimized math libraries available from leading vendors (IBM's ESSL, Intel's MKL, AMD's AMCL, etc.).

### **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**.

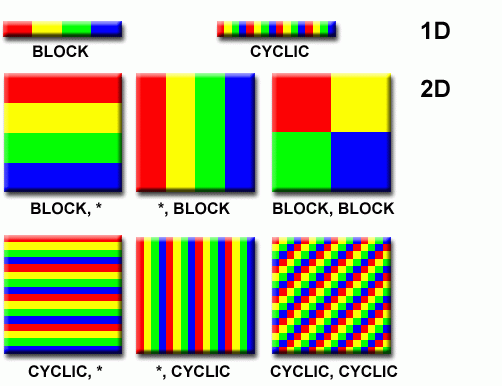
#### **Domain Decomposition**

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



Domain decomposition

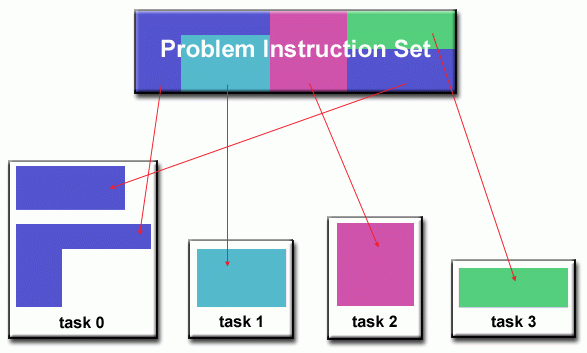
There are different ways to partition data:



Partitioning examples

#### **Functional Decomposition**

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

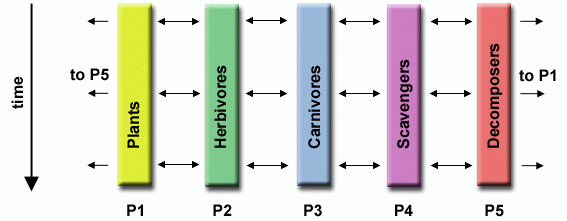


Functional decomposition

Functional decomposition lends itself well to problems that can be split into different tasks. For example:

##### **Ecosystem Modeling**

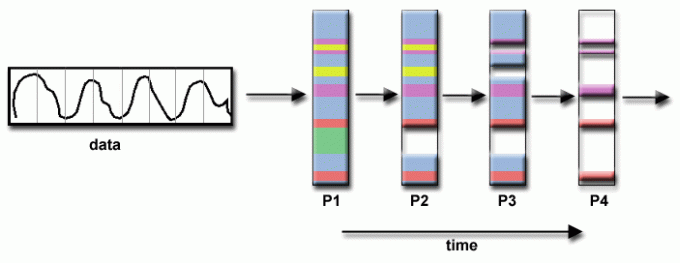
Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time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



Ecosystem modeling

##### **Signal Processing**

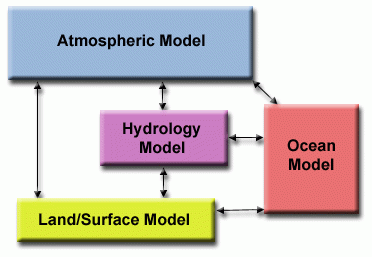
An audio signal data set is passed through four distinct computational filters. Each filter is a separate process. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth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



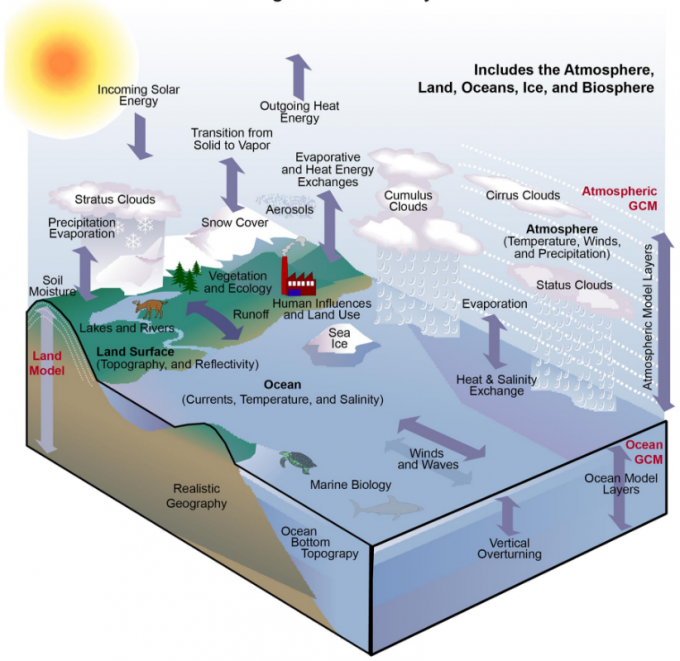
Signal processing

##### **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.



Climate modeling



Complex relationships between climate and atmospheric modeling components

* Combining these two types of problem decomposition is common and natural.

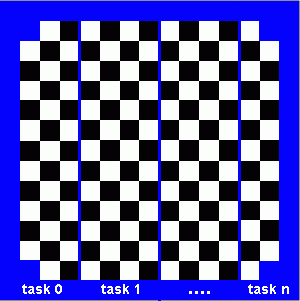
### **Communications**

#### **Who Needs Communications?**

The need for communications between tasks depends upon your problem:

##### **You DON'T need communications**

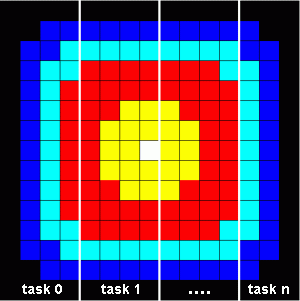
* Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data. These types of problems are often called **embarrassingly parallel** - little or no communications are required.
* For example, imagine an image processing operation where every pixel in 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.



Parallel processing without need for much communication

##### **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 2-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.

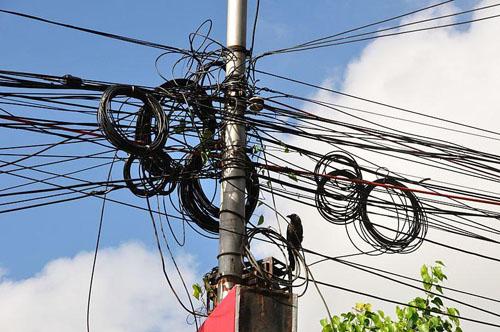


Parallel processing that needs a lot of communication

#### **Factors to Consider**

There are a number of important factors to consider when designing your program's inter-task communications:

##### **Communication overhead**



Communication is complex

* 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**

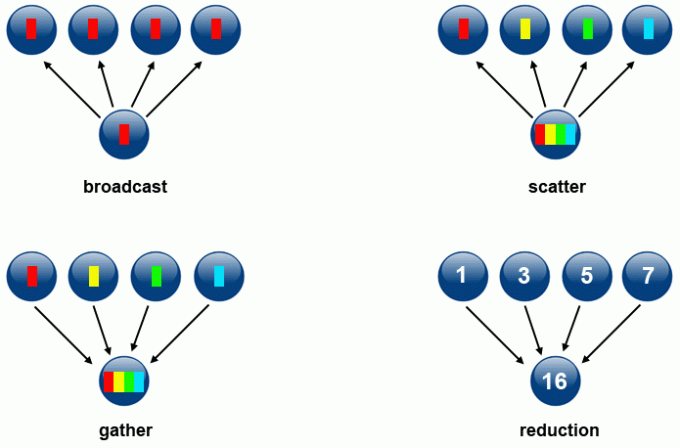
* With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
* With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even 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.

##### **Scope of communications**

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

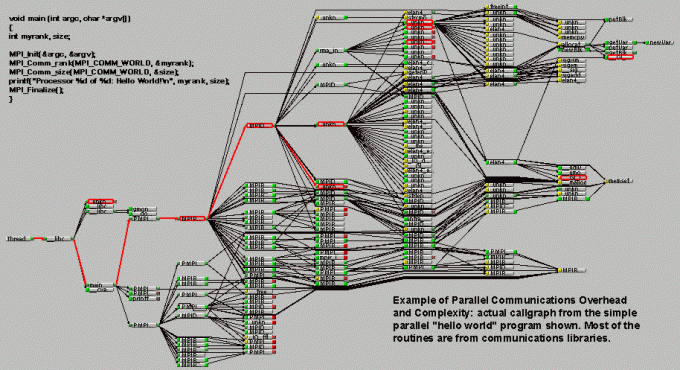


Different types of communication scope

##### **Efficiency of communications**

* Oftentimes, the programmer has choices that can affect communications performance. Only a few are mentioned here.
* Which implementation for a given model should be used? Using the Message Passing Model as an example, one MPI implementation may be faster on a given hardware platform than another.
* What type of communication operations should be used? As mentioned previously, asynchronous communication operations can improve overall program performance.
* Network fabric—different platforms use different networks. Some networks perform better than others. Choosing a platform with a faster network may be an option.

##### **Overhead and complexity**



Example of parallel communications overhead and complexity

Finally, realize that this is only a partial list of things to consider!

### **Synchronization**

* Managing the sequence of work and the tasks performing it is a critical design consideration for most parallel programs.
* Can be a significant factor in program performance (or lack of it)
* Often requires "serialization" of segments of the program.



Synchronization of tasks in parallel processing is an important design consideration

#### **Types of Synchronization**

##### **Barrier**

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

##### **Lock / semaphore**

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

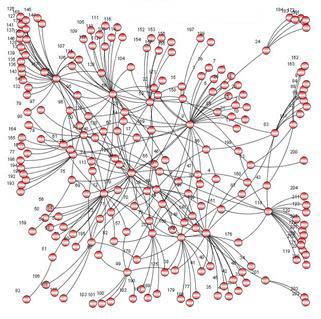
##### **Synchronous communication operations**

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

### **Data Dependencies**

#### **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.



Visualization of dependencies

#### **Examples**

##### **Loop carried data dependence**

DO J = MYSTART,MYEND

A(J) = A(J-1) \* 2.0

END DO

* The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.
* If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:
  + Distributed memory architecture - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation
  + Shared memory architecture - task 2 must read A(J-1) after task 1 updates it

##### **Loop independent data dependence**

task 1 task 2

------ ------

X = 2 X = 4

. .

. .

Y = X\*\*2 Y = X\*\*3

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

#### **How to Handle Data Dependencies**

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

### **Load Balancing**

* Load balancing refers to the practice of distributing approximately equal amounts of 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.

|  |  |
| --- | --- |
| Image  Barrier synchronization point diagram | Image  Traffic as an example of load imbalance |

#### **How to Achieve Load Balance**

##### **Equally partition the work each task receives**

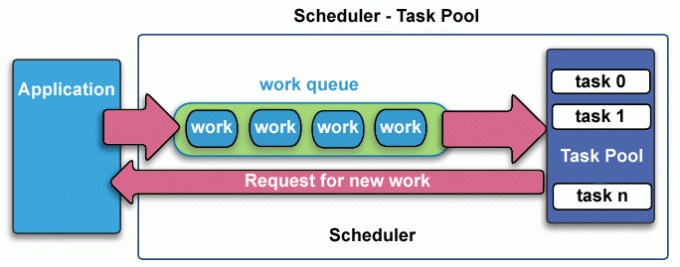
* For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
* For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
* If a heterogeneous mix of machines with varying performance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

##### **Use dynamic work assignment**

* Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:

|  |  |  |
| --- | --- | --- |
| Image  Sparse arrays  Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros." | Image  Adaptive grid methods  Adaptive grid methods - some tasks may need to refine their mesh while others don't. | Image  N-body simulation  N-body simulations - particles may migrate across task domains requiring more work for some tasks. |

* When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a **scheduler-task pool**approach. As each task finishes its work, it receives a new piece from the work queue.



Scheduler-task pool

* Ultimately, it may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.