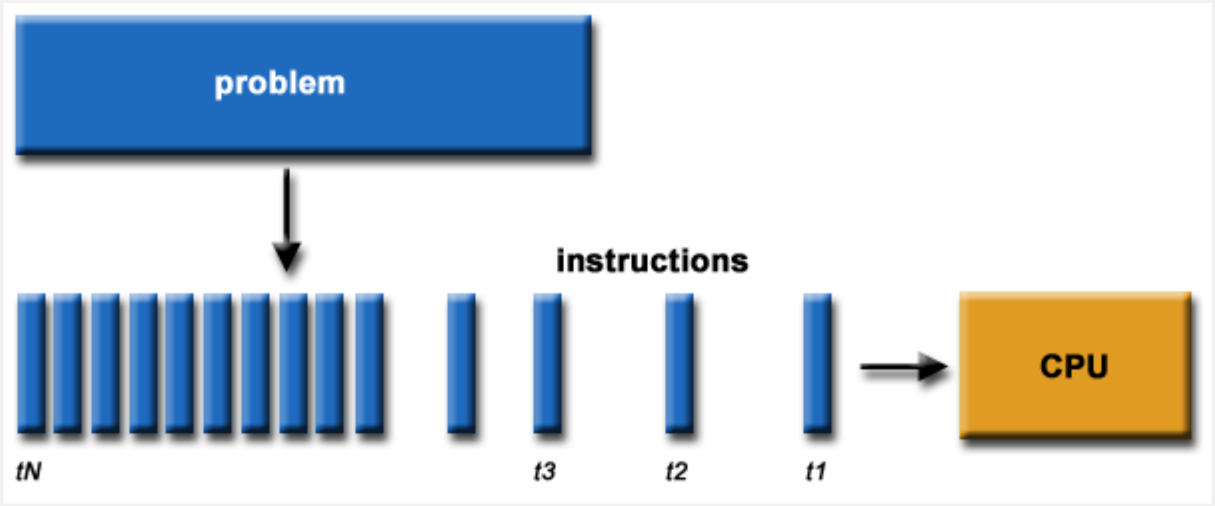
DDAM
INTRODUCTION TO PARALLEL COMPUTING

Docente: Patrizio Dazzi

Mail: patrizio.dazzi@isti.cnr.it

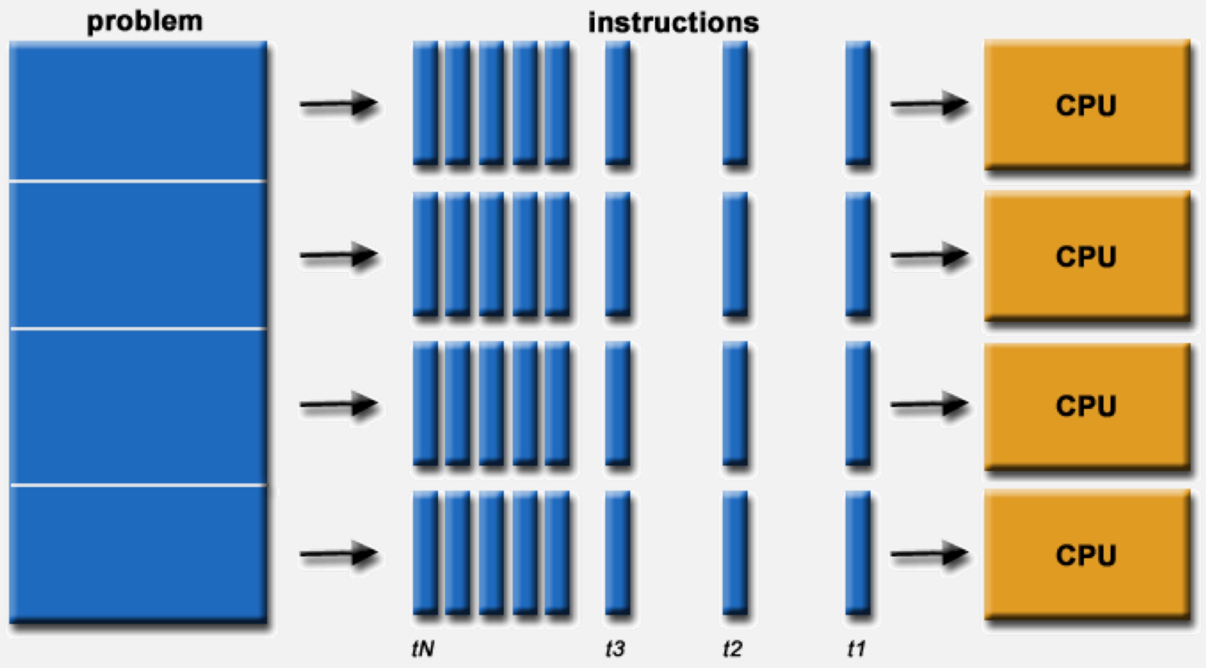
WHAT IS PARALLEL COMPUTING? (1)

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



WHAT IS PARALLEL COMPUTING? (2)

- 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



PARALLEL COMPUTING: RESOURCES

- The compute resources can include:
  - A single computer with multiple processors;
  - A single computer with (multiple) processor(s) and some specialized computer resources (GPU, FPGA …)
  - An arbitrary number of computers connected by a network;
  - A combination of both.

PARALLEL COMPUTING: THE COMPUTATIONAL PROBLEM

- 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.

WHY PARALLEL COMPUTING? (1)

- This is a legitime question! Parallel computing is complex on any aspect!

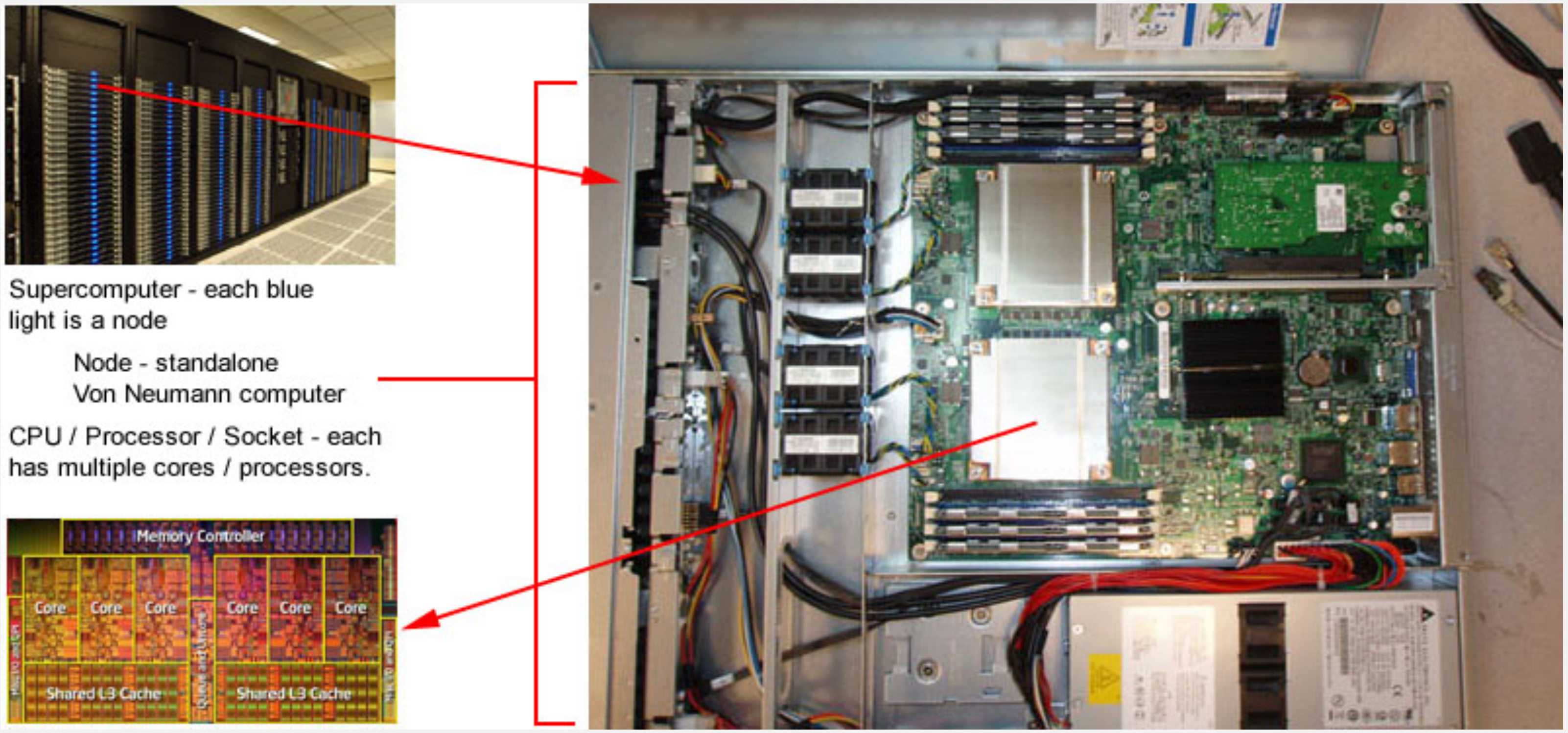
- The primary reasons for using parallel computing:
  - Save time - wall clock time
  - Solve larger problems
  - Provide concurrency (do multiple things at the same time)

WHY PARALLEL COMPUTING? (2)

- Other reasons might include:
  - Taking advantage of non-local resources - using available compute resources on a wide area network, or even the Internet when local compute resources are scarce.
  - Cost savings - using multiple "cheap" computing resources instead of paying for time on a supercomputer.
  - Overcoming memory constraints - single computers have very finite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

CONCEPTS AND TERMINOLOGY

CPU – SOCKET – PROCESSOR - CORE



Supercomputer - each blue light is a node

Node - standalone Von Neumann computer

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

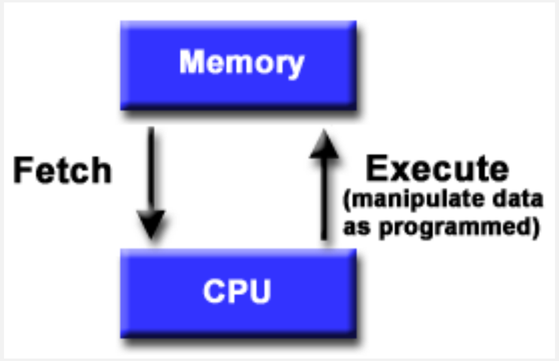
VON NEUMANN ARCHITECTURE

- For over 40 years, virtually all computers have followed a common machine model known as the von Neumann computer. Named after the Hungarian mathematician John von Neumann.

- A von Neumann computer uses the stored-program concept. The CPU executes a stored program that specifies a sequence of read and write operations on the memory.

BASIC DESIGN

- Basic design
  - Memory is used to store both program and data instructions
  - Program instructions are coded data which tell the computer to do something
  - Data is simply information to be used by the program
- A central processing unit (CPU) gets instructions and/or data from memory, decodes the instructions and then ***sequentially*** performs them.

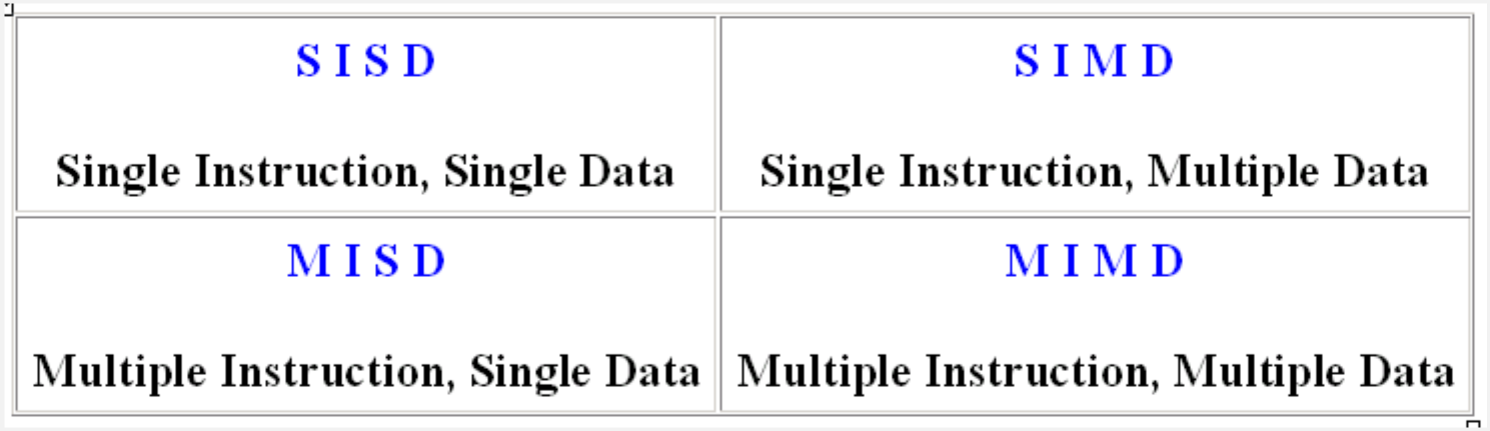


FLYNN'S CLASSICAL TAXONOMY

- There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- 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***.

FLYNN MATRIX

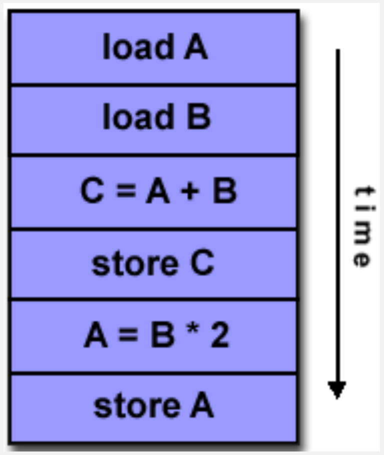
- The matrix below defines the 4 possible classifications according to Flynn



S I S D **Single Instruction, Single Data**	**S I M D** **Single Instruction, Multiple Data**
M I S D **Multiple Instruction, Single Data**	**M I M D** **Multiple Instruction, Multiple Data**

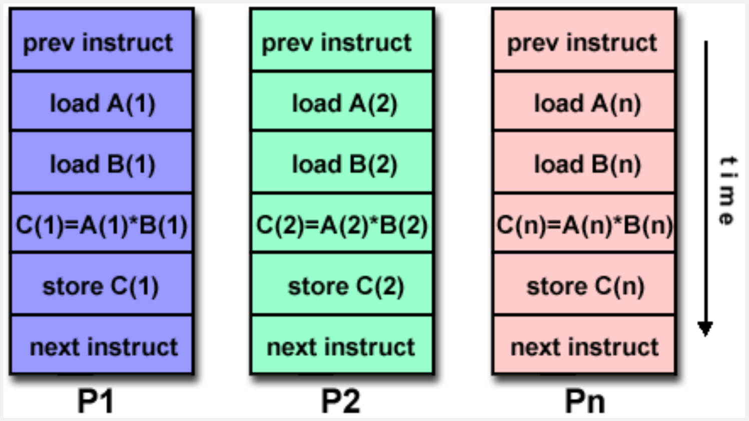
SINGLE INSTRUCTION, SINGLE DATA (SISD)

- A serial (non-parallel) computer
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer
- Examples: most PCs, single CPU workstations and mainframes



SINGLE INSTRUCTION, MULTIPLE DATA (SIMD)

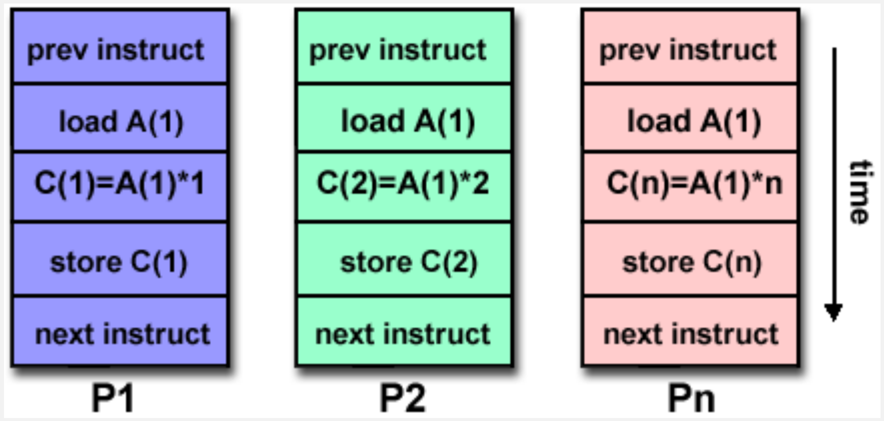
- A type of parallel computer
- Single instruction: All processing units execute the same instruction at any given clock cycle
- Multiple data: Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines



- Examples:
  - Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2
  - Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2, Hitachi S820

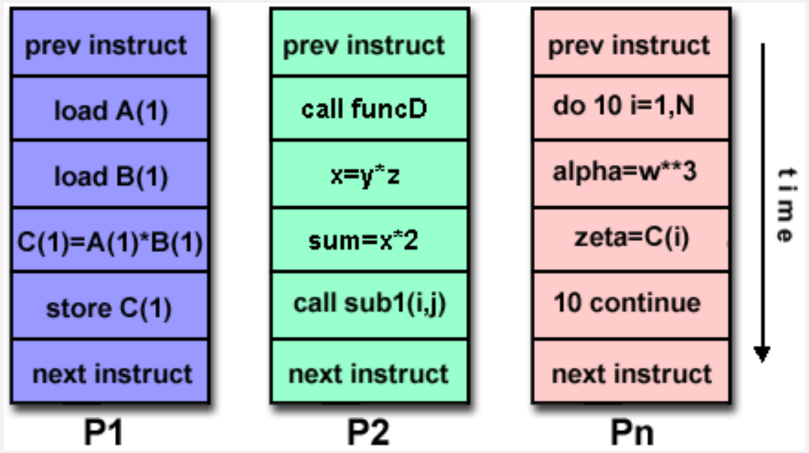
MULTIPLE INSTRUCTION, SINGLE DATA (MISD)

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.
- Few actual examples of this class of parallel computer have ever existed. 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.



MULTIPLE INSTRUCTION, MULTIPLE DATA (MIMD)

- 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 "grids" and multi-processor SMP computers - including some types of PCs.



SOME GENERAL PARALLEL TERMINOLOGY

- **Task**
  - A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor.
- **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.
- **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:

$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

  - One of the simplest and most widely used indicators for a parallel program's performance.

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

- **Scalability**
  - Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
    - Hardware - particularly memory-cpu bandwidths and network communications
    - Application algorithm
    - Parallel overhead related
    - Characteristics of your specific application and coding

LIMITS AND COSTS OF PARALLEL PROGRAMMING

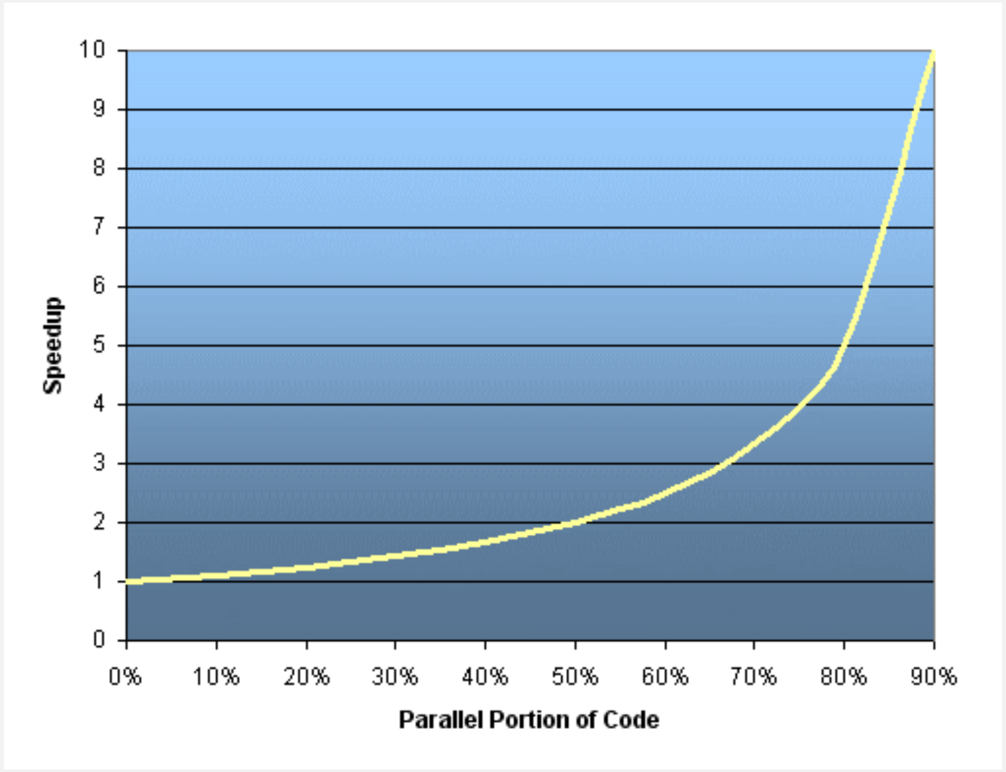
AMDAHL'S LAW (I)

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

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

- If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
- If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
- If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.

AMDAHL'S LAW (II)



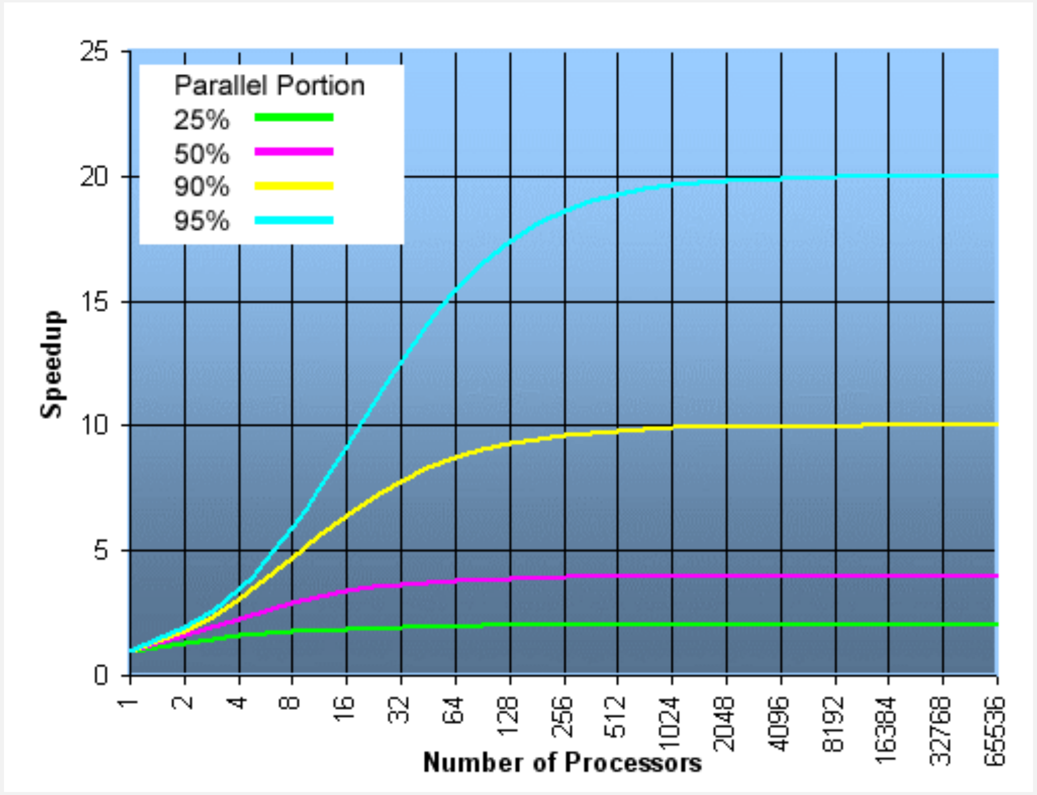
INTRODUCING THE NUMBER OF PROCESSORS (I)

Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

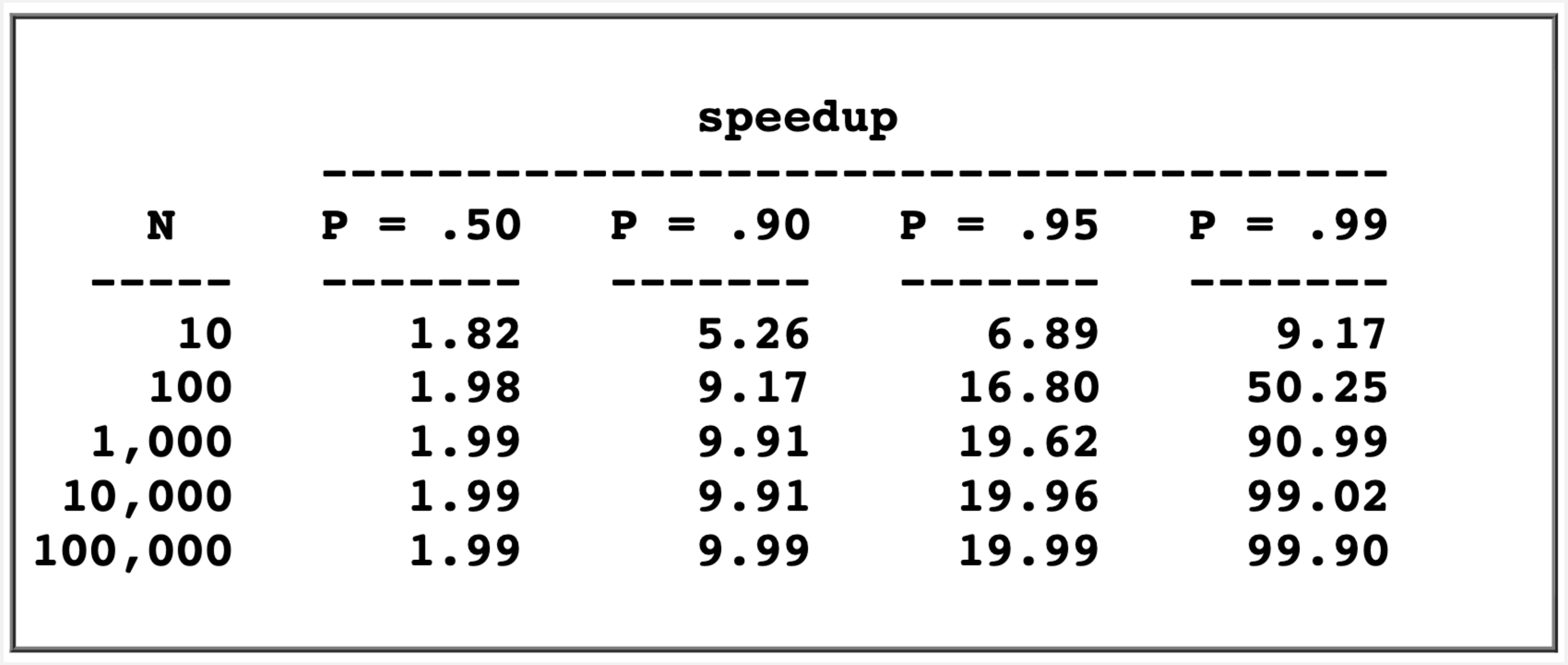
```
                           1
    speedup    =    ------------
                      P    +  S
                     ---
                      N
```

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

INTRODUCING THE NUMBER OF PROCESSORS (II)



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

DETERMINE THE RATIO IS NOT SO EASY

- Certain problems demonstrate increased performance by increasing the problem size.
- From

```
2D Grid Calculations     85 seconds   85%
Serial fraction          15 seconds   15%
```

- To

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

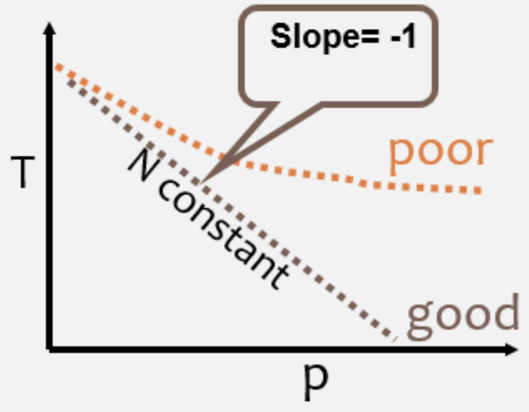
Problems that increase the percentage of parallel time with their size are more scalable than problems with a fixed percentage of parallel time.

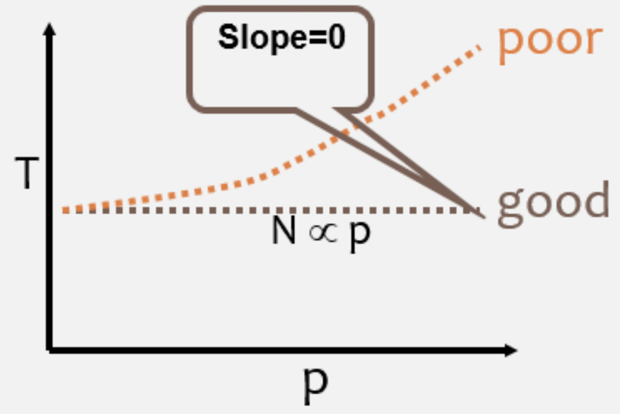
SCALABILITY (I)

- Two types of scaling based on time to solution: **strong scaling** and **weak scaling.**
- Strong scaling:
  - 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)

SCALABILITY (II)

- Weak scaling:
  - 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





ACHIEVE SCALABILITY (I)

- 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

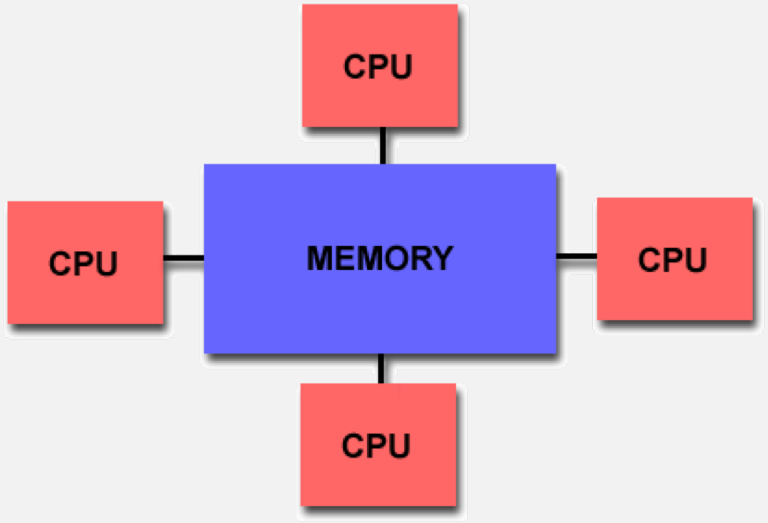
PARALLEL COMPUTER MEMORY ARCHITECTURES

MEMORY ARCHITECTURES

- Shared Memory
- Distributed Memory
- Hybrid Distributed-Shared Memory

SHARED MEMORY (I)

- 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***.

SHARED MEMORY (II)

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

SHARED MEMORY (III)

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

SHARED MEMORY (IV)

Advantages:

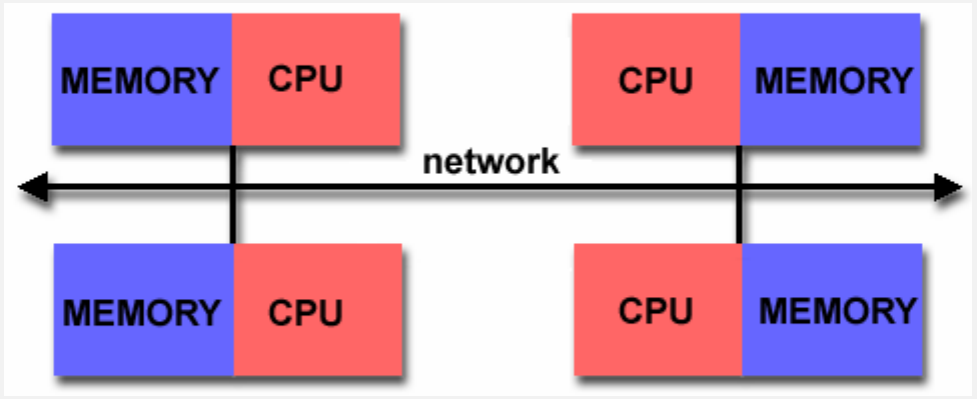
- Global address space provides a user-friendly programming perspective to memory
- Data sharing between tasks is both fast and uniform due to the proximity of memory to CPUs

Disadvantages:

- 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 (I)

- 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.



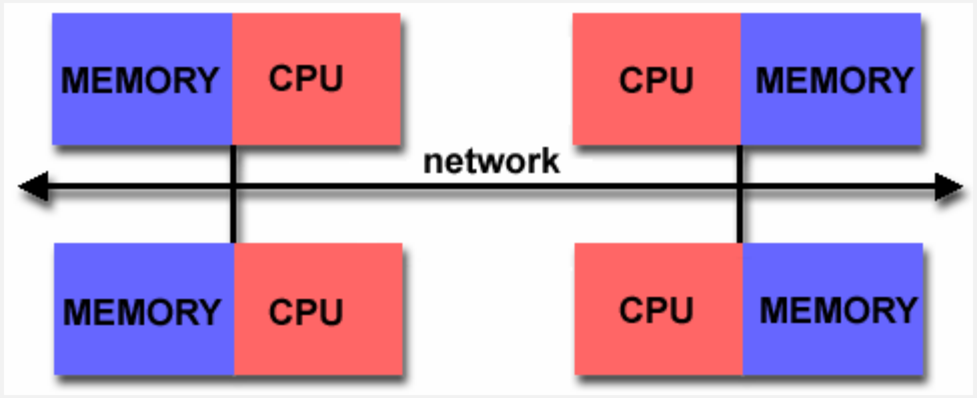
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.

DISTRIBUTED MEMORY (II)

- 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.

- 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.



DISTRIBUTED MEMORY (III)

Advantages:

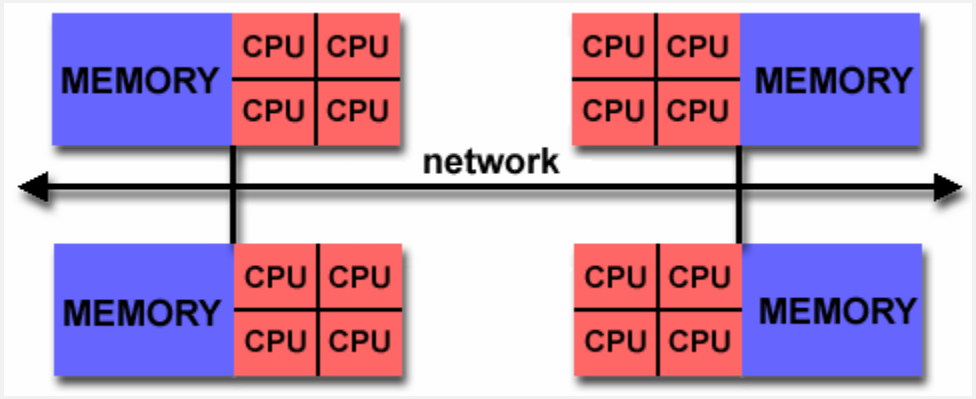
- Memory is scalable with the number of processors. Increase the number of processors and the size of memory increases proportionately.
- Each processor can rapidly access its own memory without interference and without the overhead incurred with trying to maintain global cache coherency.
- Cost effectiveness: 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 (I)

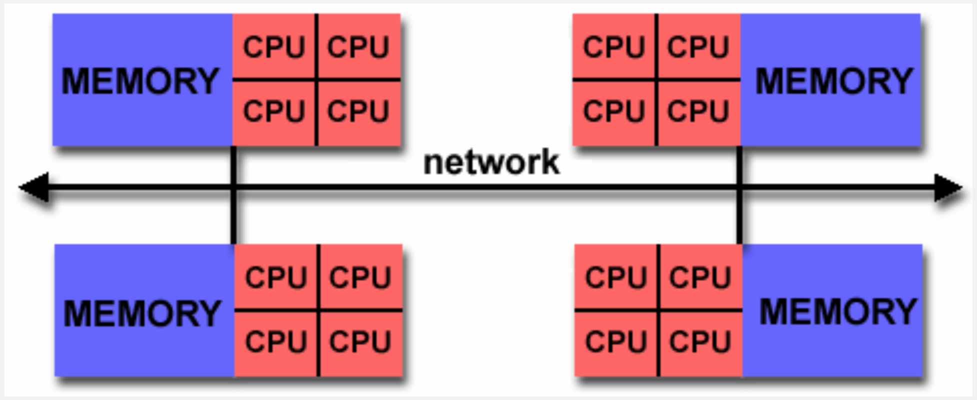
- The largest and fastest computers in the world today employ both shared and distributed memory architectures.



- The shared memory component is usually a **cache coherent SMP machine**. Processors on a given SMP can address that machine's memory as global.

HYBRID DISTRIBUTED-SHARED MEMORY (II)

- The distributed memory component enables the execution on multiple SMPs. Every SMP knows only about its own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.

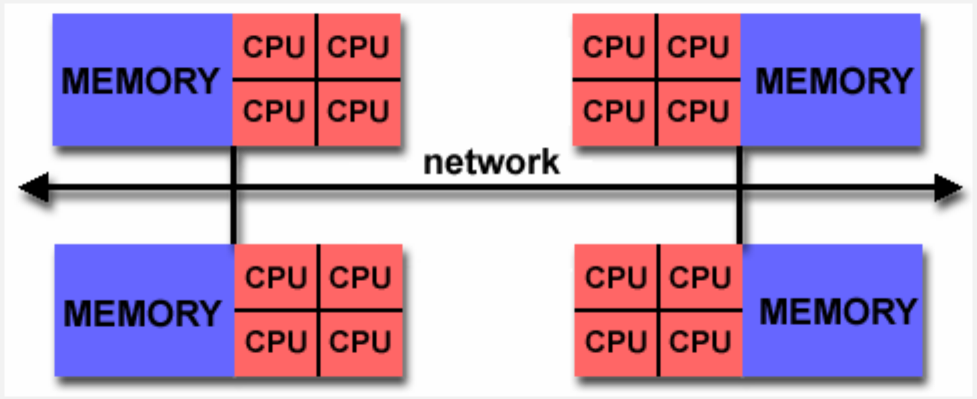


- Current trends seem to indicate that this type of memory architecture will continue to prevail and increase at the high end of computing for the foreseeable future.
  - Do you remember the story of "flexibility"?

HYBRID DISTRIBUTED-SHARED MEMORY (III)

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



DESIGNING PARALLEL PROGRAMS

UNDERSTAND THE PROBLEM AND THE PROGRAM

- 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 necessitates 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**.

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

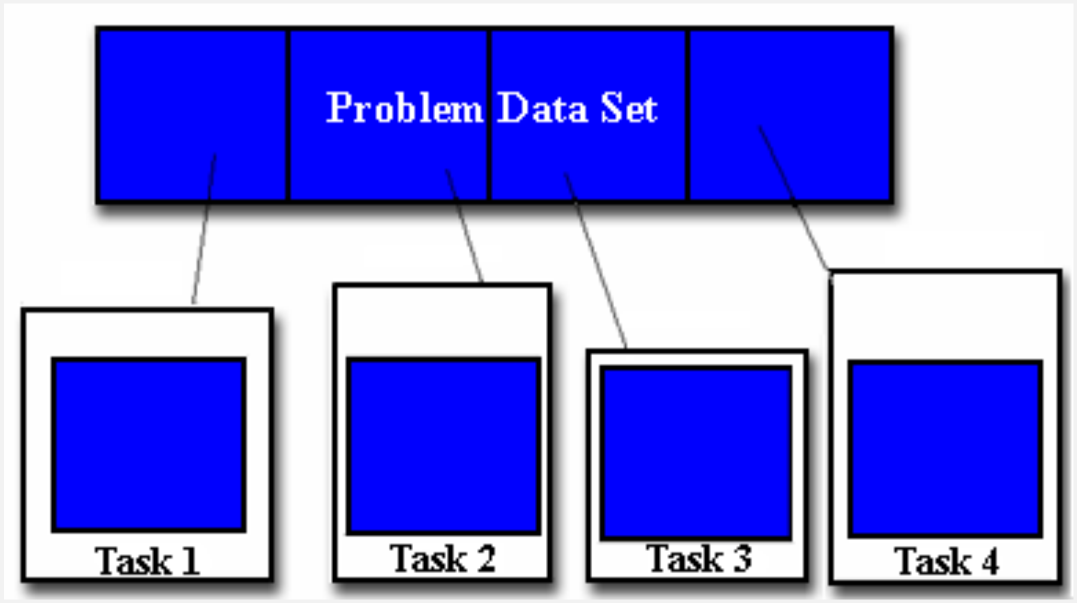
- Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
- May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

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***

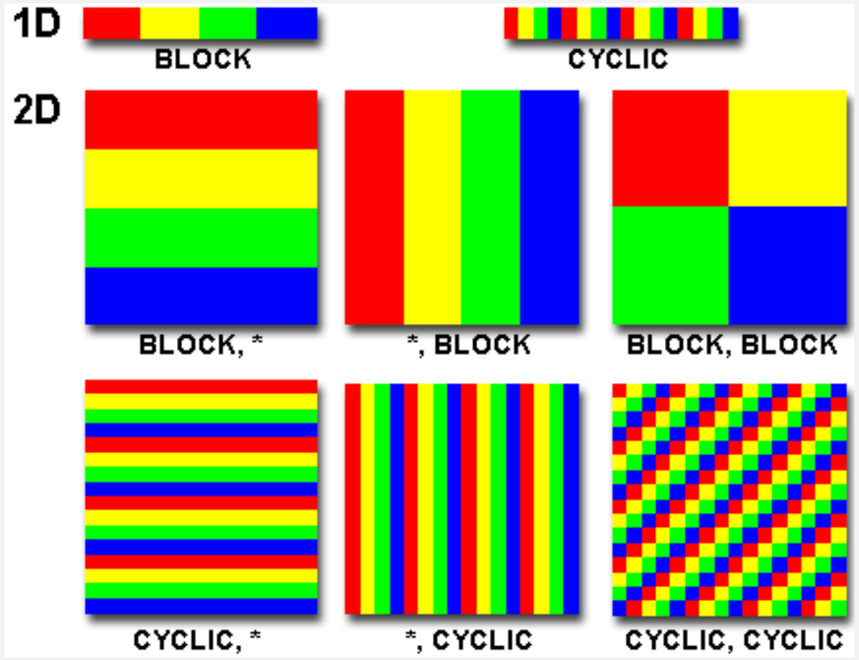
DOMAIN DECOMPOSITION

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



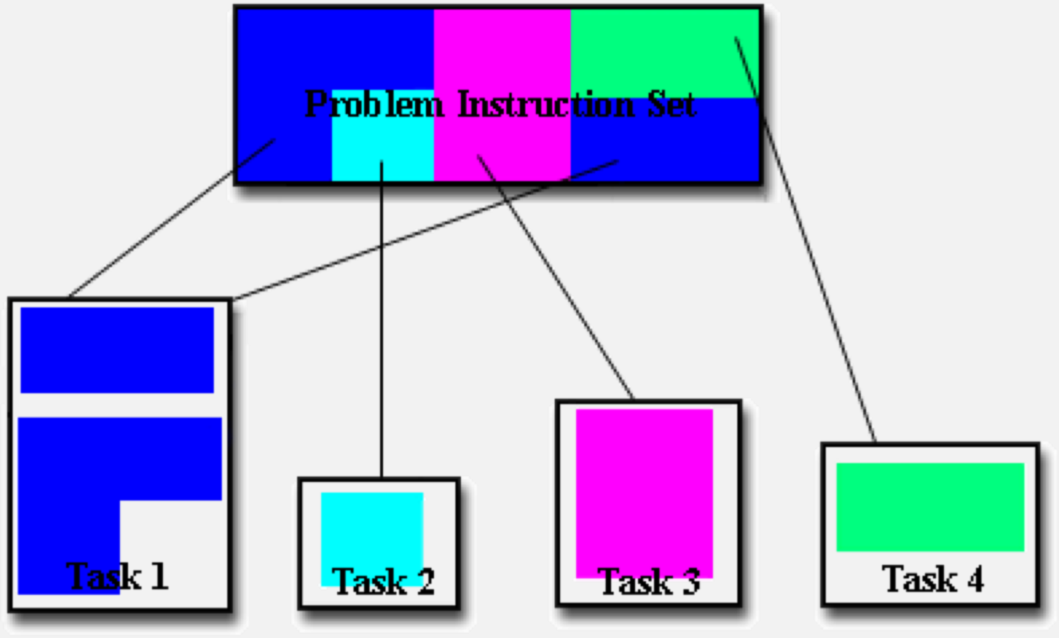
PARTITIONING DATA

- There are different ways to partition data



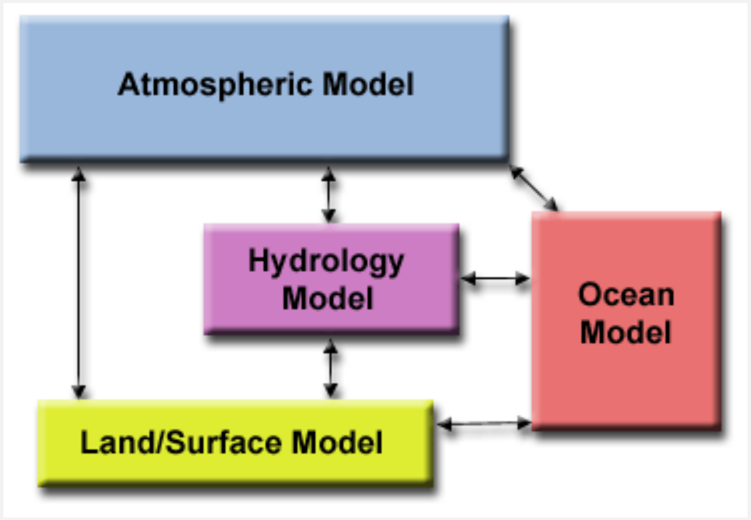
FUNCTIONAL DECOMPOSITION

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
- Functional decomposition lends itself well to problems that can be split into different tasks. For example
  - Ecosystem Modeling
  - Signal Processing
  - Climate Modeling



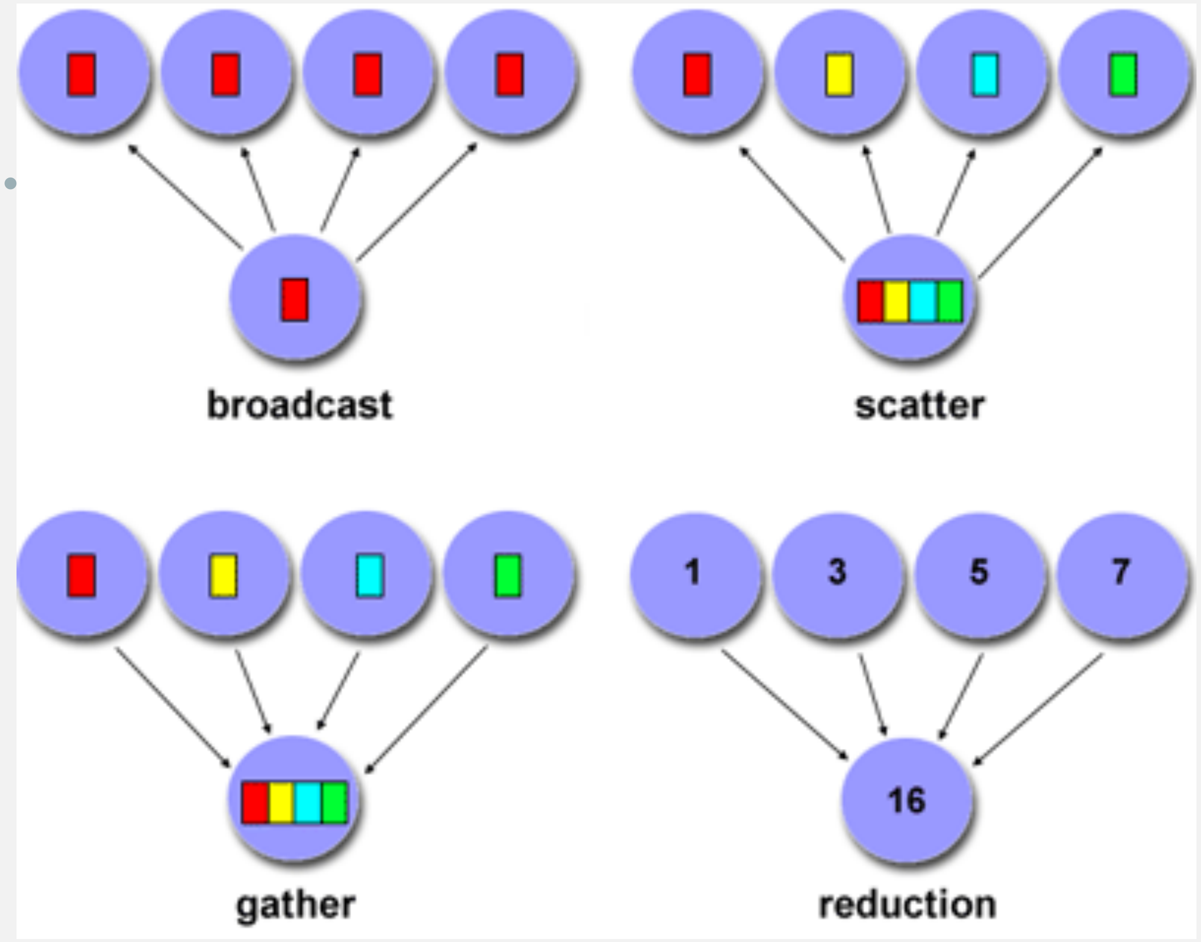
HYBRID DECOMPOSITION

- 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.



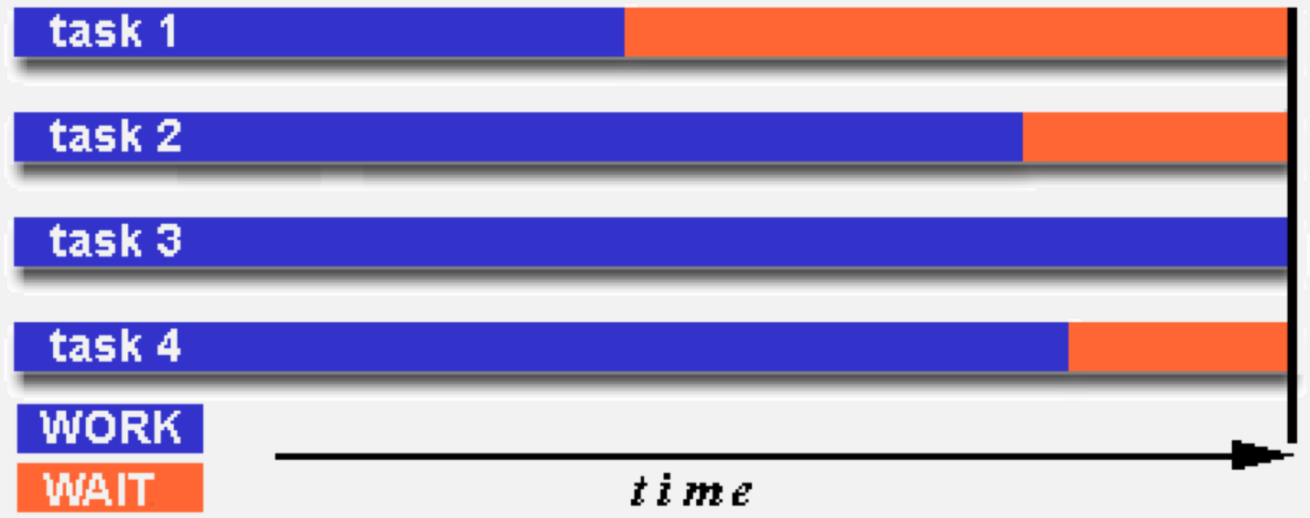
- Combining these two types of problem decomposition is common and natural.

COLLECTIVE COMMUNICATIONS



LOAD BALANCING

- Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



HOW TO ACHIEVE LOAD BALANCE? (1)

- **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.

HOW TO ACHIEVE LOAD BALANCE? (2)

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