Parallel Processing - 01

Md. Biplob Hosen

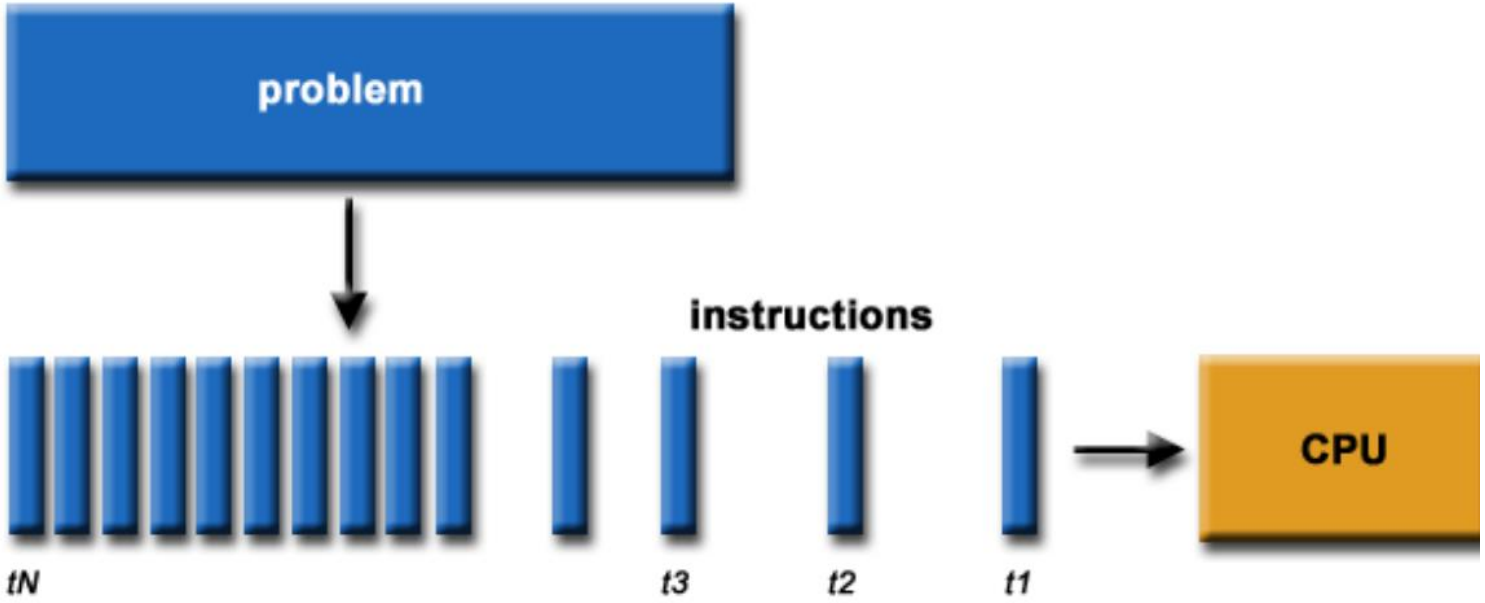
Lecturer, IIT-JU

Email: biplob.hosen@juniv.edu

What is Parallel Computing?

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

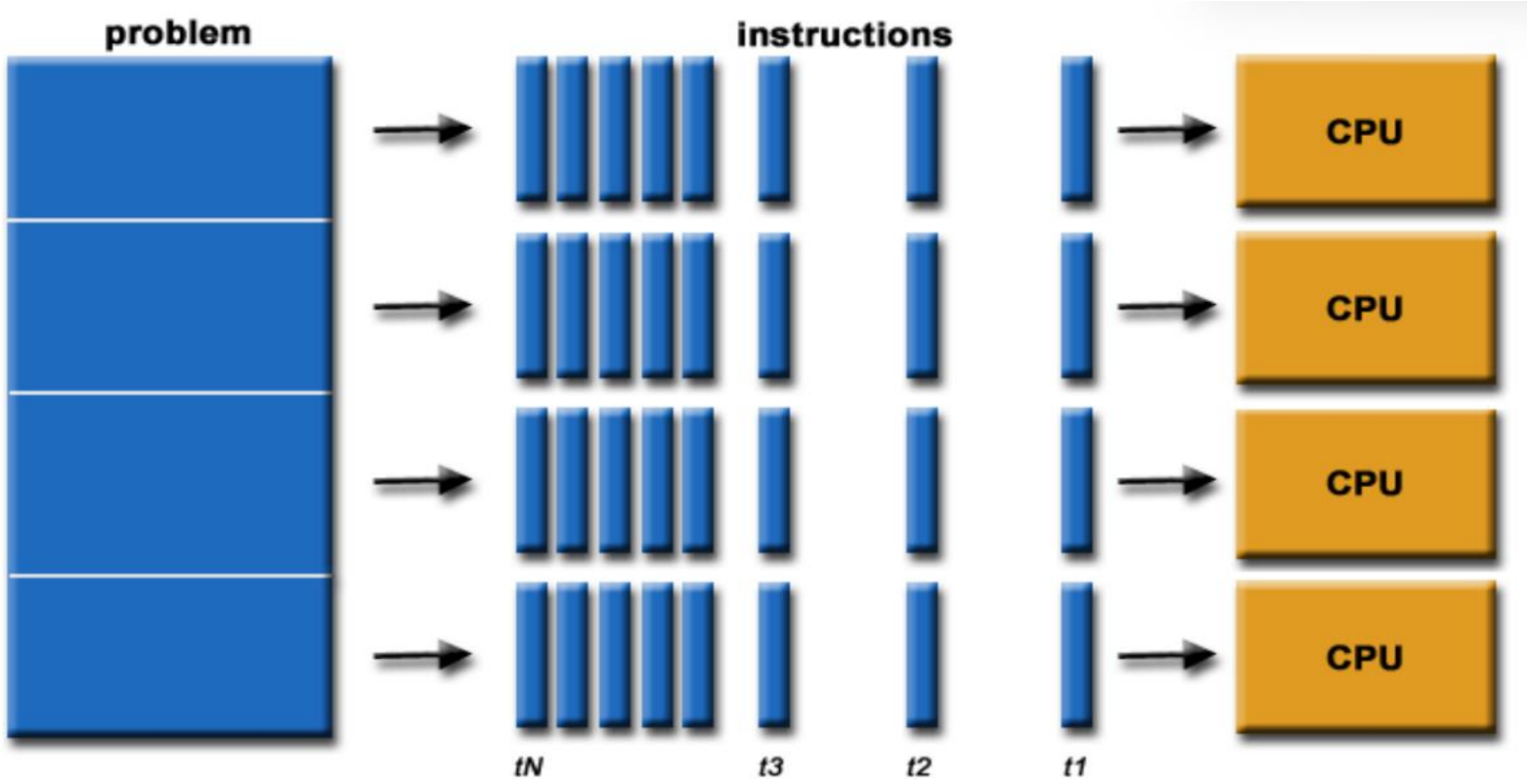
- To be run on a single computer having a single Central Processing Unit (CPU);
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.



What is Parallel Computing?

Parallel computation:

- 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.

Environment

- The compute resources can include:
  - A single computer with multiple processors;
  - An arbitrary number of computers connected by a network;
  - A combination of both.
- The computational problem usually demonstrates characteristics such as the ability to be:
  - Broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Solved in less time with multiple compute resources than with a single compute resource.

Why Use Parallel Computing?

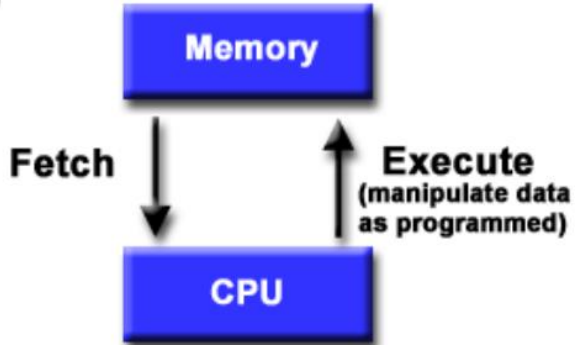
- The primary reasons for using parallel computing:
  - Save time - wall clock time.
  - Solve larger problems.
  - Provide concurrency (do multiple things at the same time).
- Other reasons might include:
  - Taking advantage of non-local resources - using available compute resources on a wide area 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 infinite memory resources. For large problems, using the memories of multiple computers may overcome this obstacle.

von Neumann Architecture

- For over 40 years, virtually all computers have followed a common machine model known as the von Neumann computer.
- 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:

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



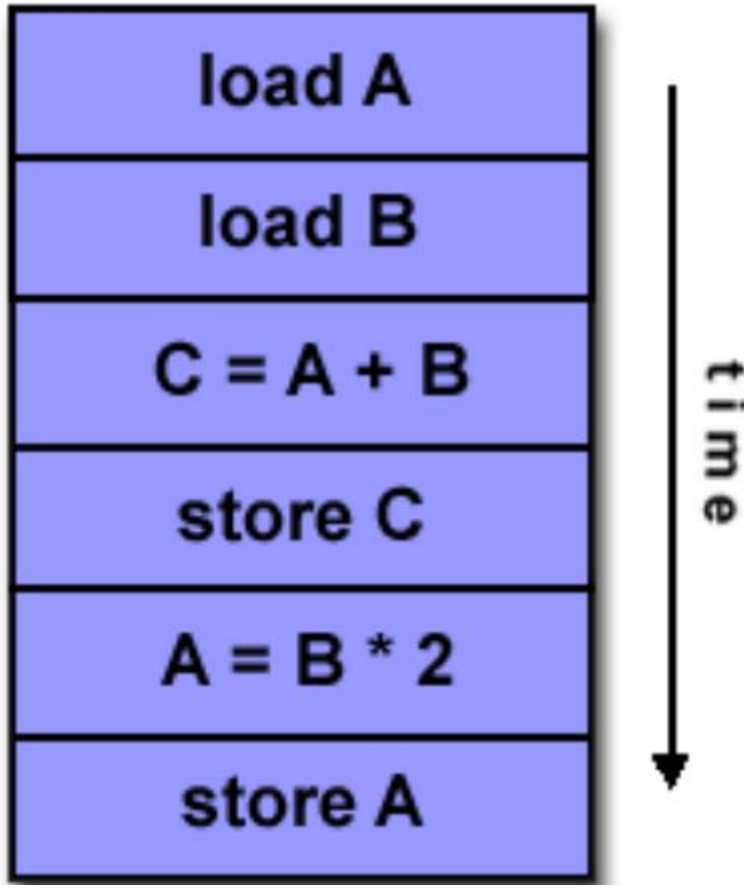
Flynn's Classical Taxonomy

- There are different ways to classify parallel computers.
- One of the more widely used classification in use 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.
- Here are 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

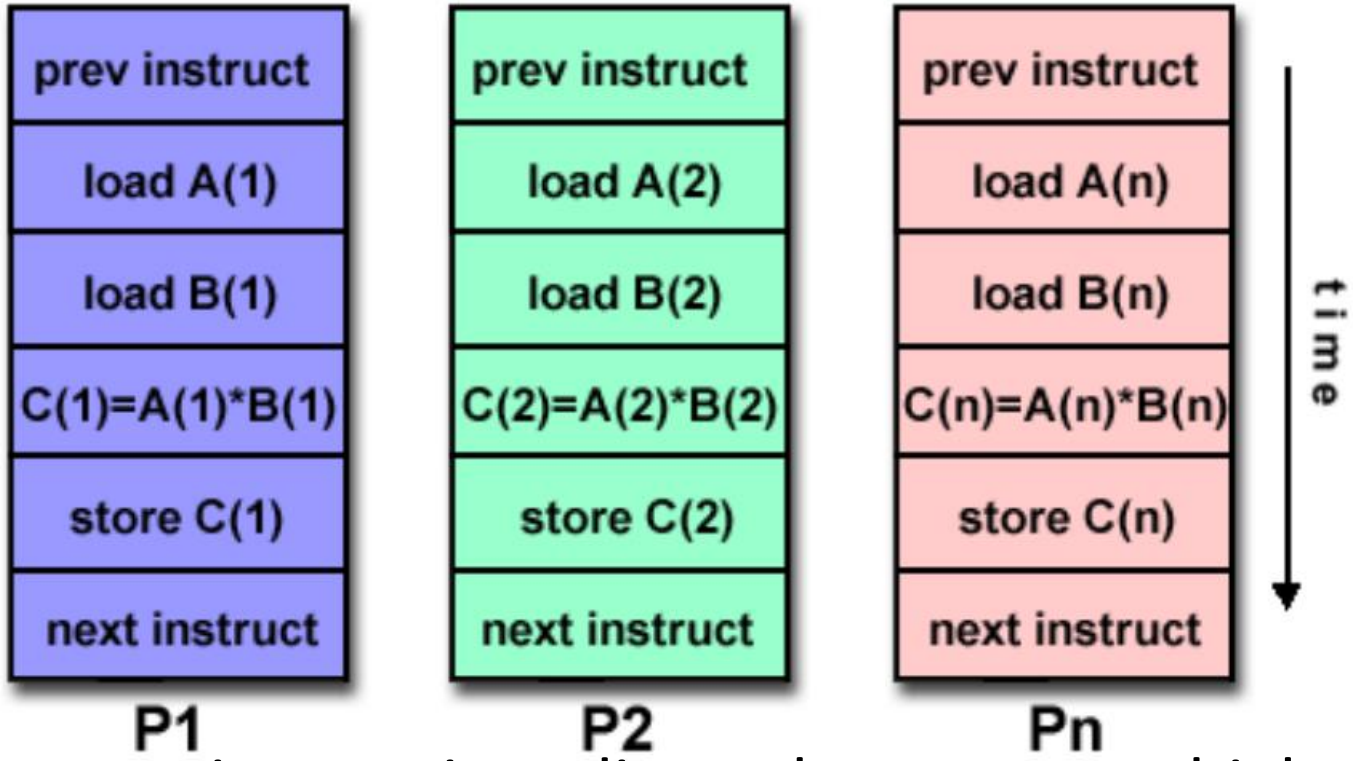
Single Instruction, Single Data (SISD)

- 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.
- Examples: Most PCs, single CPU workstations and mainframes.



Single Instruction, Multiple Data (SIMD)

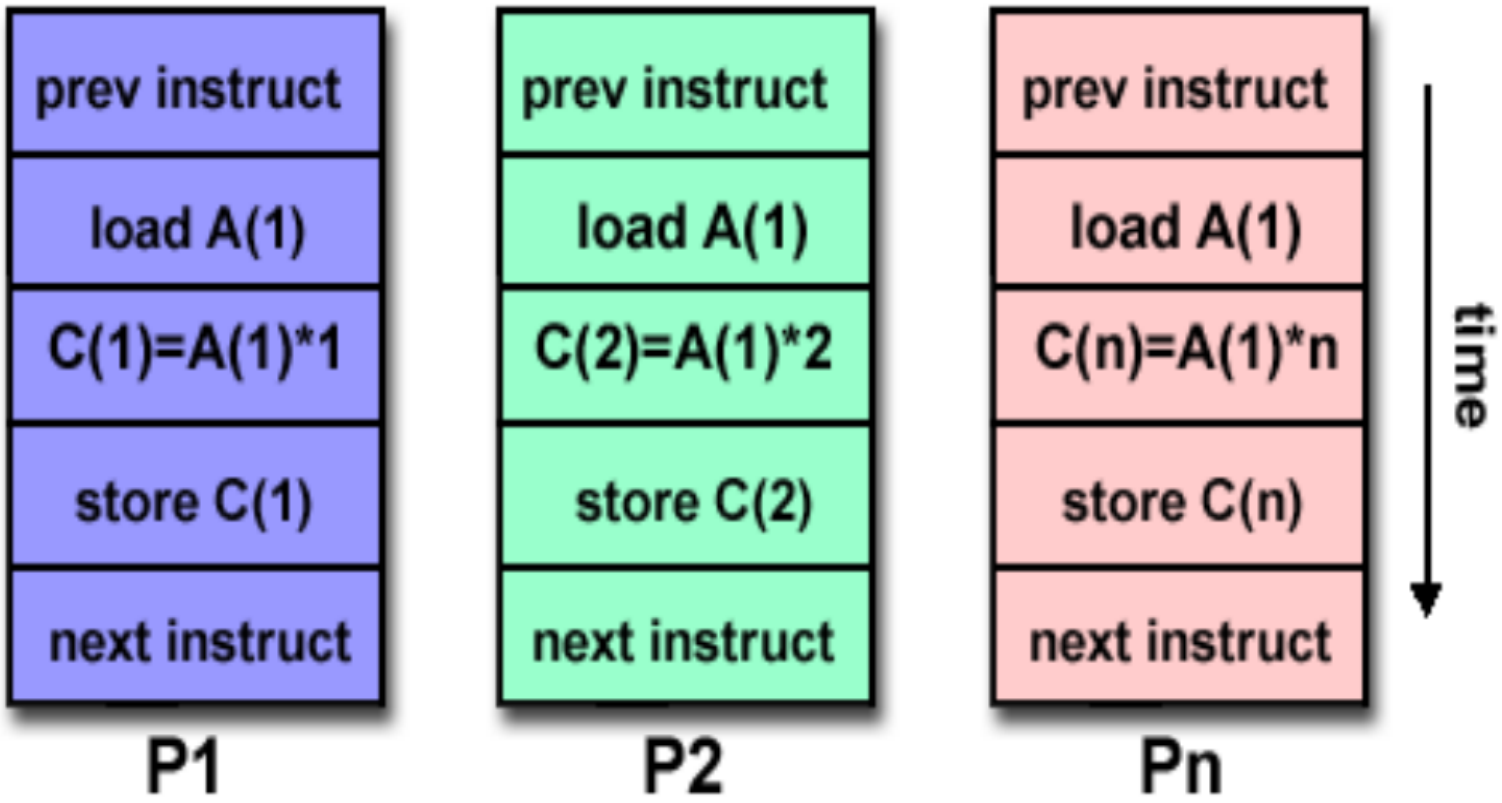
- A type of parallel computer.
- **Single instruction:** All processing units execute the same instruction at any given clock cycle.
- **Multiple data:** Each processing unit can operate on a different data element.



- 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.

Multiple Instruction, Single Data (MISD)

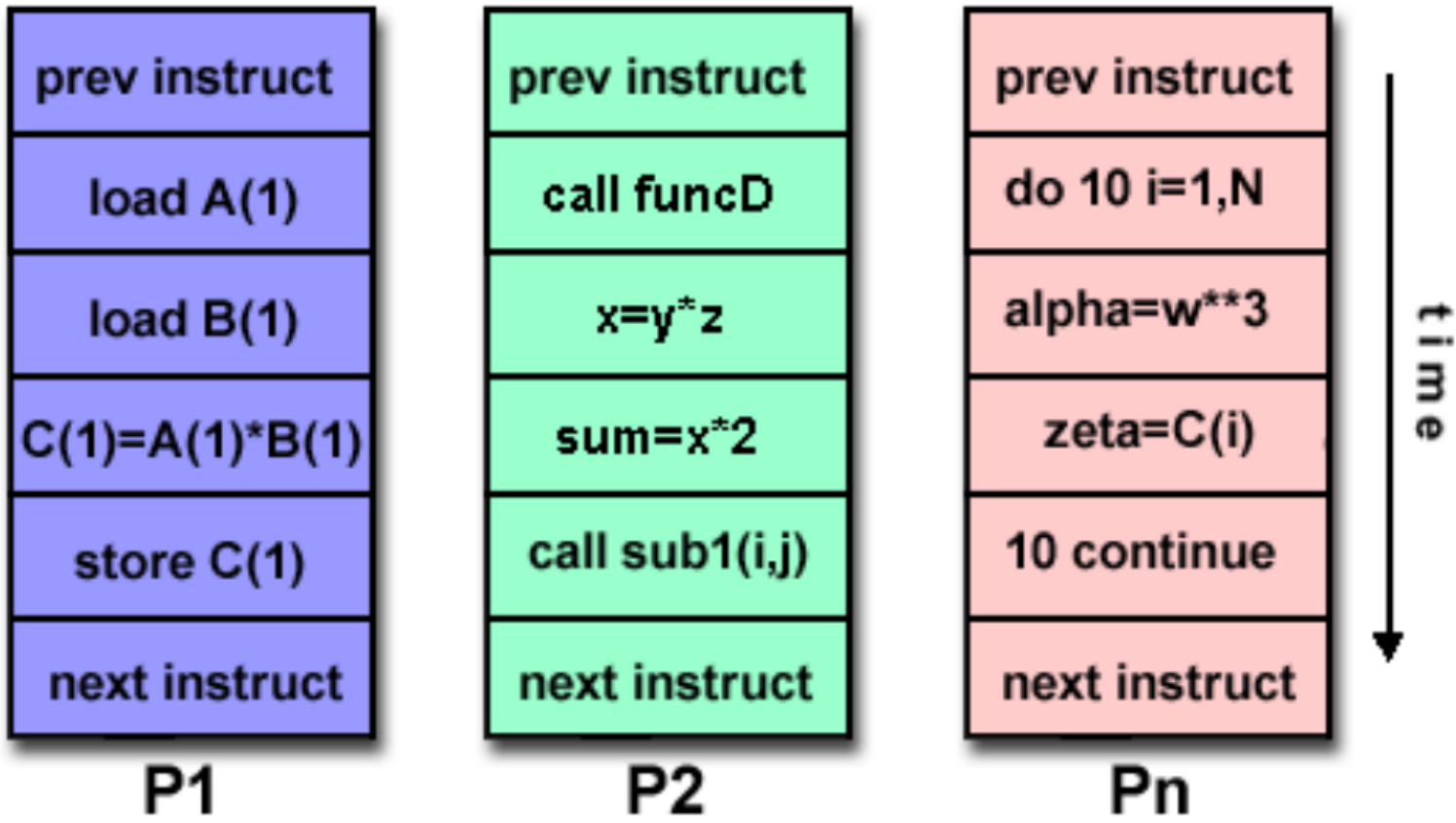
- A single data stream is fed into multiple processing units.
- Each processing unit operates on the data independently via independent instruction streams.



- Few actual examples of this class of parallel computer have ever existed.
- Some conceivable uses might be:
  - Multiple frequency filters operating on a single signal stream.
  - Multiple cryptography algorithms attempting to crack a single coded message.

Multiple Instruction, Multiple Data (MIMD)

- 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.
- Examples: most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers.

Amdahl's Law - Speedup

- Deals with the potential speedup of a program using multiple processors compared to a single processor.
- Consider a program running on a single processor such that a fraction (1 - f) of the execution time involves code that is inherently serial and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead.
- Let T be the total execution time of the program using a single processor.
- Then the speedup using a parallel processor with N processors that fully exploits the parallel portion of the program is as follows:

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on N parallel processors}}$$

Continue…

$$= \frac{T(1-f)+Tf}{T(1-f)+\frac{Tf}{n}} \qquad = \frac{1}{(1-f)+\frac{f}{N}}$$

Two important conclusions can be drawn:

1. When f is small, the use of parallel processors has little effect.
2. As N approaches infinity, speedup is bound by 1/(1 - f), so that there are diminishing returns for using more processors.

- Amdahl's law can be generalized to evaluate any design or technical improvement in a computer system.
- Consider any enhancement to a feature of a system that results in a speedup. The speedup can be expressed as:

$$\text{Speedup} = \frac{\text{Performance after enhancement}}{\text{Performance before enhancement}}$$

$$\text{Speedup} = \frac{\text{Execution time before enhancement}}{\text{Execution time after enhancement}}$$

Continue…

- Suppose that a feature of the system is used during execution a fraction of the time f, before enhancement. The speedup of the feature after enhancement is SU_f. Then the overall speedup of the system is:

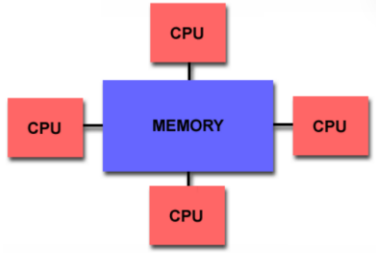
$$\text{Speedup} = \frac{1}{(1-f) + \frac{f}{SU_f}}$$

- Suppose that a task makes extensive use of floating-point operations, with 40% of the time is consumed by floating-point operations. With a new hardware design, the floating-point module is speeded up by a factor of K.
- **What is the overall speedup?**
- **What is the maximum speedup?**

$$\text{Speedup} = \frac{1}{0.6 + \frac{0.4}{k}}$$

Maximum Speedup = 1.67

Memory Organization



Shared Memory

- Shared memory parallel computers vary widely, but 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.
- Changes in a memory location effected by one processor are visible to all other processors.

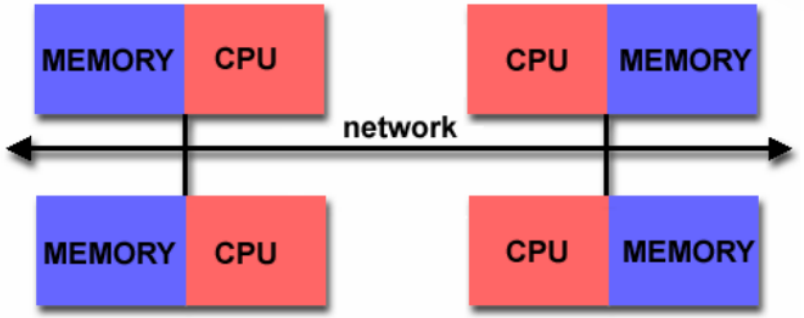
Advantages:

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

Disadvantages:

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

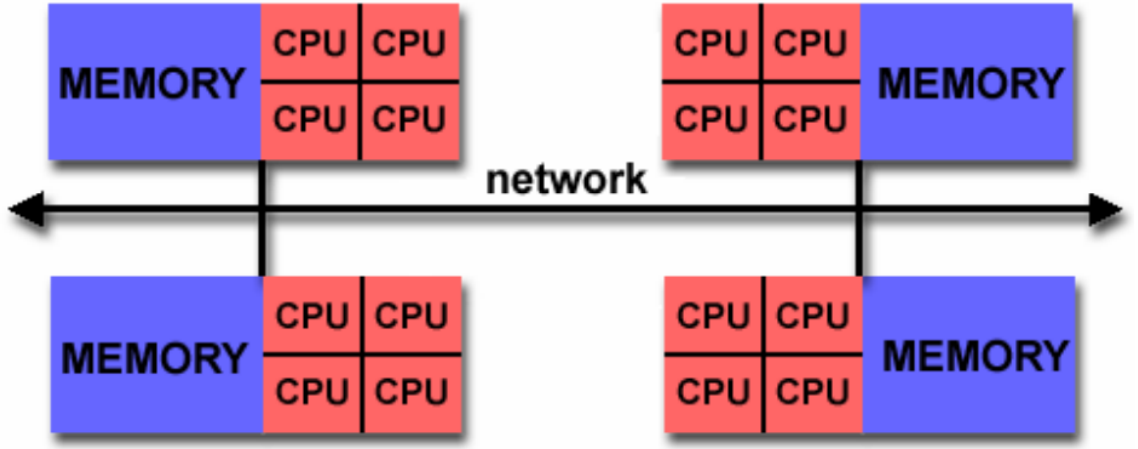
Memory Organization



Distributed Memory

- Distributed memory systems require a communication network to connect inter-processor memory.
- Processors have their own local memory. Memory addresses in one processor do not map to another processor, so there is no concept of global address space across all processors.
- 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.

Memory Organization



Hybrid Distributed Shared Memory

- 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 Symmetric Multiprocessor (SMP) machine.
- Processors on a given SMP can address that machine's memory as global.
- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP.
- Therefore, network communications are required to move data from one SMP to another.

Parallel Programming Models

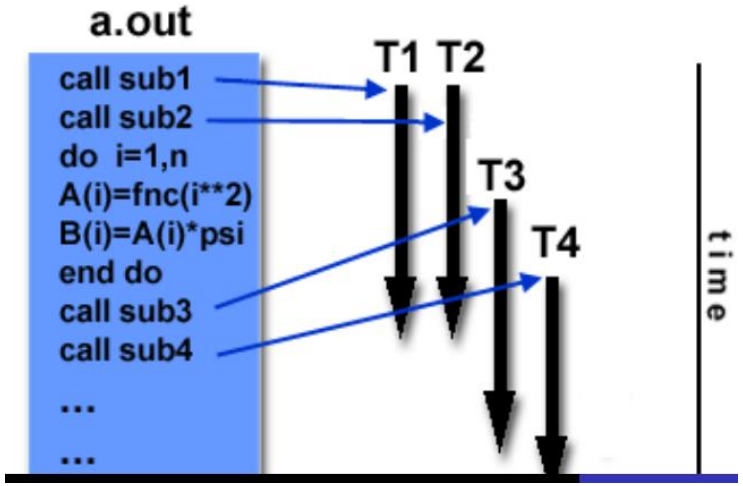
- There are several parallel programming models in common use:
  1. Shared Memory
  2. Threads
  3. Message Passing
  4. Data Parallel
  5. Hybrid
- These models are NOT specific to a particular type of machine or memory architecture. In fact, these models can (theoretically) be implemented on any underlying hardware.
- Which model to use is often a combination of what is available and personal choice. There is no "best" model, although there certainly are better implementations of some models over others.

Shared Memory Model

- In the shared-memory programming model, tasks share a common address space, which they read and write asynchronously.
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory.
- An advantage of this model from the programmer's point of view is that the notion of data "ownership" is lacking, so there is no need to specify explicitly the communication of data between tasks. 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.
- **Implementations:** On shared memory platforms, the native compilers translate user program variables into actual memory addresses, which are global.

Threads Model

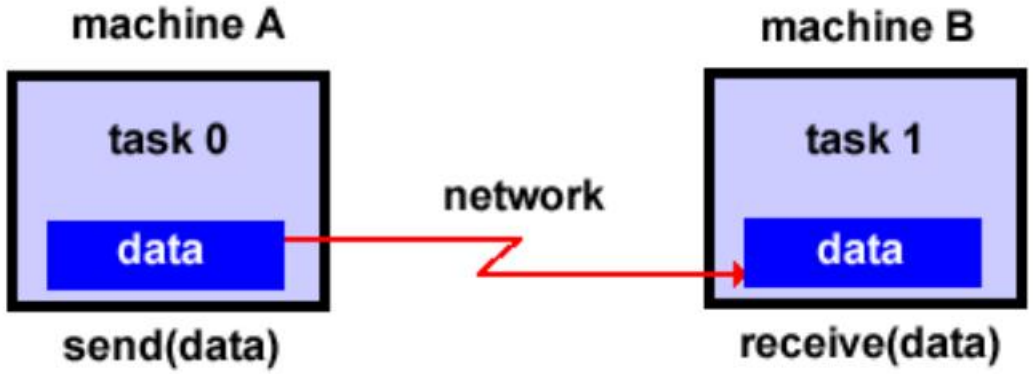
- In the threads model of parallel programming, a single process can have multiple, concurrent execution paths.
- An analogy to describe threads is the concept of a single program that includes a number of subroutines.
- The main program a.out is scheduled to run by the native OS. It performs some serial work, and then creates tasks (threads) that can be scheduled and run by the OS concurrently.
- Each thread has local data, but also, shares the entire resources of a.out.
- 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. This requires synchronization constructs to insure that more than one thread is not updating the same global address at any time.

Message Passing Model

- The message passing model has the following characteristics:
- A set of tasks that use their own local memory during computation.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.

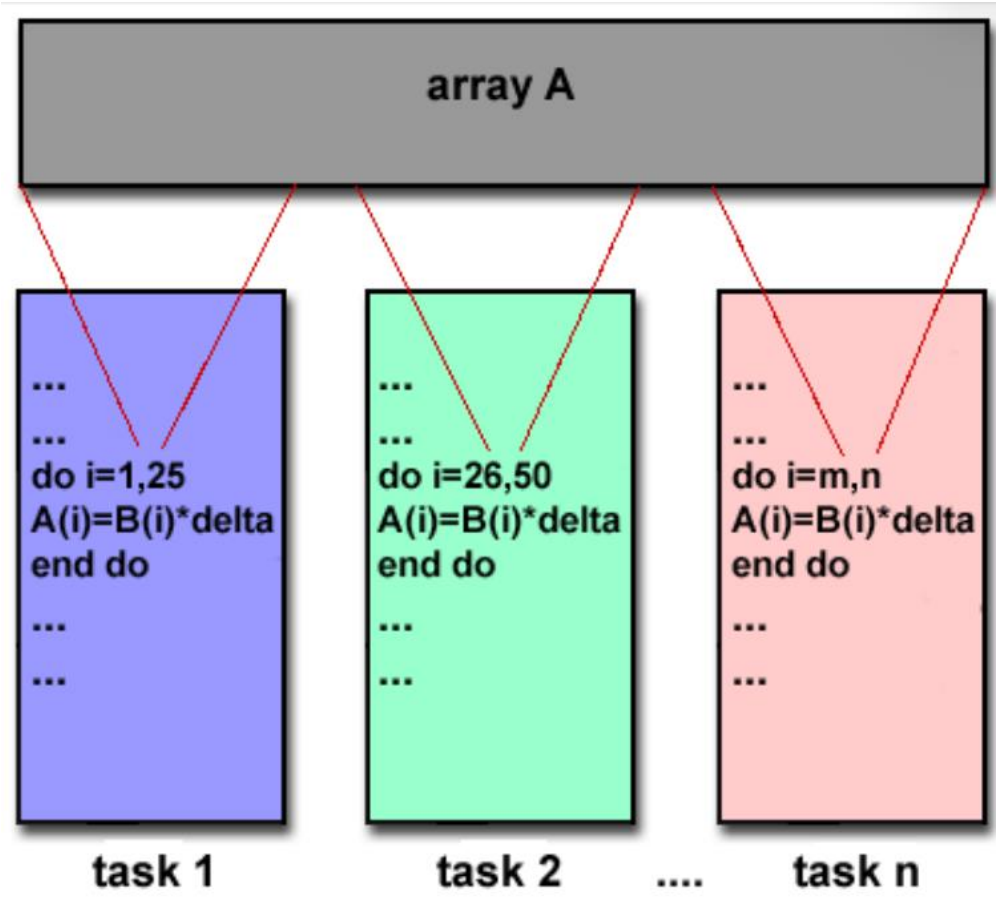


- From a programming perspective, message passing implementations commonly comprise a library of subroutines that are imbedded in source code. The programmer is responsible for determining all parallelism.

Data Parallel Model

The data parallel model demonstrates the following characteristics:

- Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
- A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
- Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".



Hybrid Model

- Two or more parallel programming models are combined:
- Currently, a common example of a hybrid model is the combination of the message passing model with either the threads model or the shared memory model. This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines.
- Another common example of a hybrid model is combining data parallel with message passing. Data parallel implementations on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer

Types of Parallelism

Data Parallelism

- Data Parallelism means concurrent execution of the same task on each multiple computing core.
- Let's take an example, summing the contents of an array of size N.
- For a single-core system, one thread would simply sum the elements [0] . . . [N − 1].
- For a dual-core system, however, thread A, running on core 0, could sum the elements [0] . . . [N/2 − 1] and while thread B, running on core 1, could sum the elements [N/2] . . . [N − 1].
- So the Two threads would be running in parallel on separate computing cores.

Types of Parallelism

Task Parallelism

- Task Parallelism means concurrent execution of the different task on multiple computing cores.
- Consider again our example above, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. Again, the threads are operating in parallel on separate computing cores, but each is performing a unique operation.

Types of Parallelism

Bit-level Parallelism

- Bit-level parallelism is a form of parallel computing which is based on increasing processor word size.
- In this type of parallelism, with increasing the word size reduces the number of instructions the processor must execute in order to perform an operation on variables whose sizes are greater than the length of the word.
- E.g., consider a case where an 8-bit processor must add two 16-bit integers.
- First the 8 lower-order bits from each integer were must added by processor, then add the 8 higher-order bits, and then two instructions to complete a single operation.
- A processor with 16- bit would be able to complete the operation with single instruction.

Types of Parallelism

Instruction-level Parallelism

- Instruction-level parallelism means the simultaneous execution of multiple instructions from a program.
- While pipelining is a form of ILP, we must exploit it to achieve parallel execution of the instructions in the instruction stream.
- Example

```
for (i=1; i<=100; i= i+1)
        y[i] = y[i] + x[i];
```

- This is a parallel loop. Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little opportunity for overlap.

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:

Automatic vs. Manual Parallelization

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.

Understand the Problem

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

Example of Parallelizable Problem:

- Calculate the potential energy for each of several thousand independent conformations of a molecule.
- When done, find the minimum energy conformation.
- This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable.

Example of non-Parallelizable Problem:

- Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula: $F(k + 2) = F(k + 1) + F(k)$
- This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones.

Hotspots & Bottlenecks

Hotspots: 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.

Bottlenecks: Identify bottlenecks in the program:

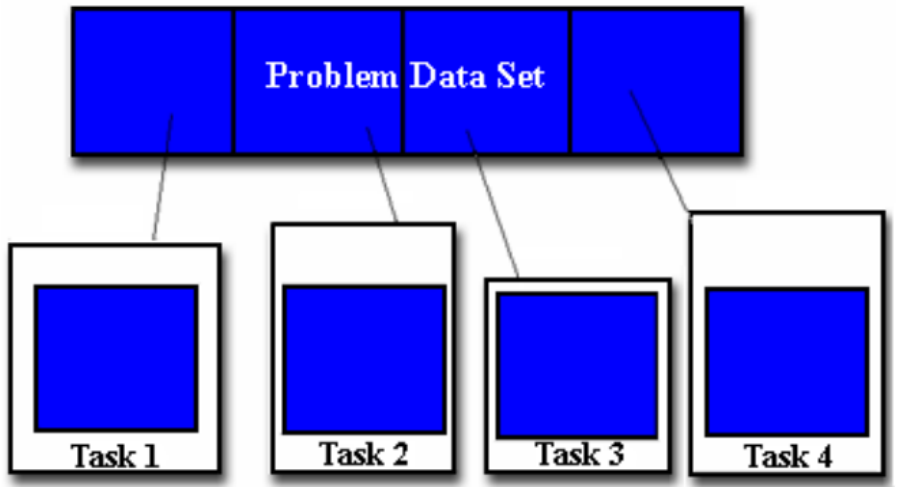
- Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred?
- For example, I/O is usually something that slows a program down.
- May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas.

Partitioning Problem

- 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.



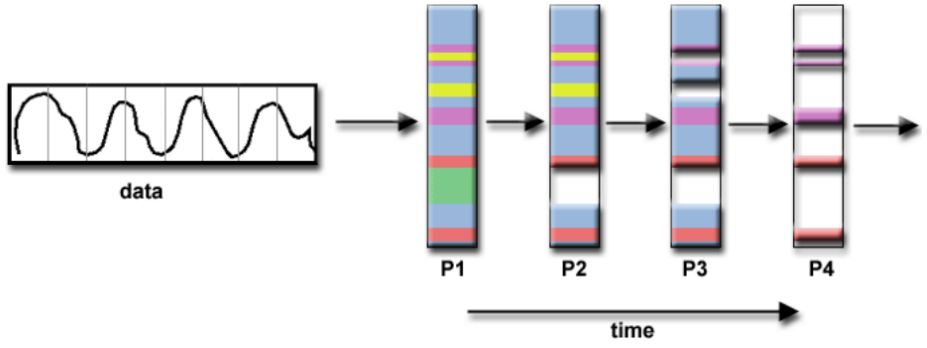
Partitioning Problem

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.

Example: 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.



Synchronization

Types of Synchronization

- Barrier
- Lock / semaphore
- Synchronous communication operations

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.

Synchronization

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.
- 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.

Thank You ☺