PARALLEL PROGRAMMING PARADIGMS

Dr. Emmanuel S. Pilli

Why Parallelism

- Faster, of course
  - Finish the work earlier
    - Same work in less time
  - Do more work
    - More work in the same time

How to parallelize

- Break down the computational part into small pieces
- Assign the small jobs to the parallel running processes
- May become complicated when the small piece of jobs depend upon others

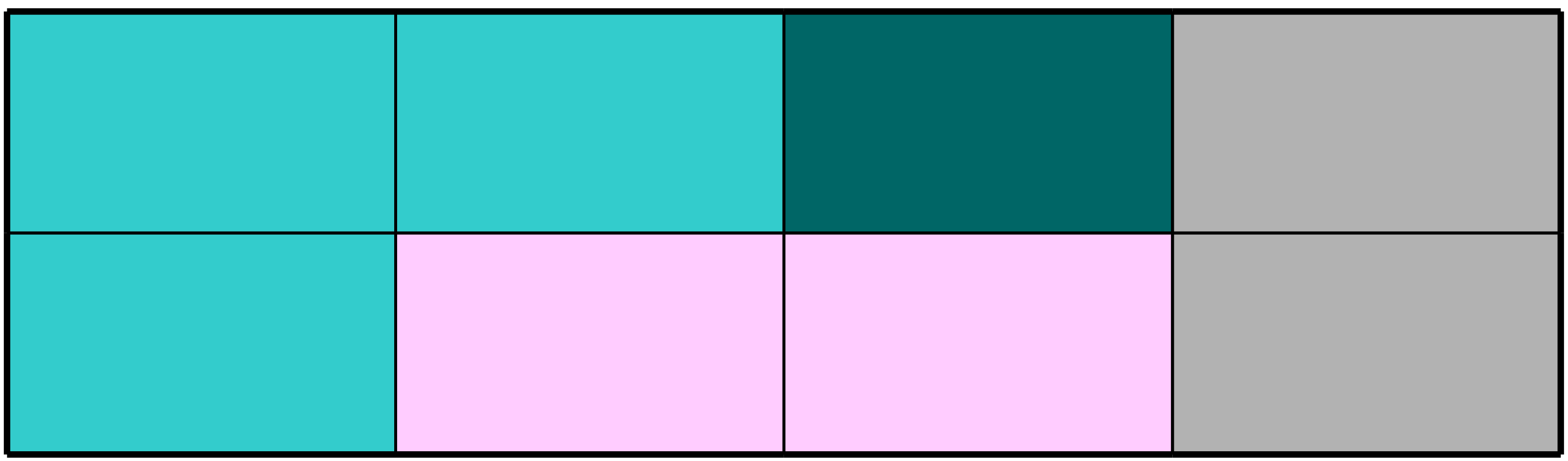
Writing a Parallel Program

- If you are starting with an existing serial program, debug the serial code completely
- Identify which parts of the program can be executed concurrently:
  - Requires a thorough understanding of the algorithm
  - Exploit any parallelism which may exist
  - May require restructuring of the program and/or algorithm. May require an entirely new algorithm.
- Decompose the program:
  - Task Parallelism
  - Data Parallelism
  - Combination of both

Task (Functional) Parallelism

- Decomposing the problem into different processes which can be distributed to multiple processors for simultaneous execution
- Good to use when there is not static structure or fixed determination of number of calculations to be performed

Task (Functional) Parallelism



The Problem

Machine 1

Machine 2

Machine 3

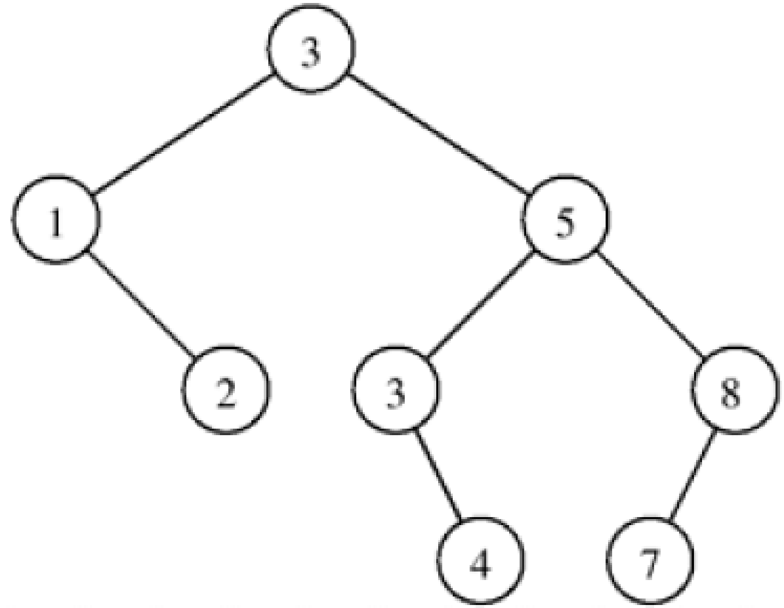
Machine 4

Task Parallelism

- The computations in any parallel algorithm can be viewed as a task dependency graph
- The interrelationships among the tasks are utilized to promote locality or to reduce interaction costs
- This is employed to solve problems where the amount of data associated with the tasks is large relative to the amount of computation associated with them

Task Parallelism

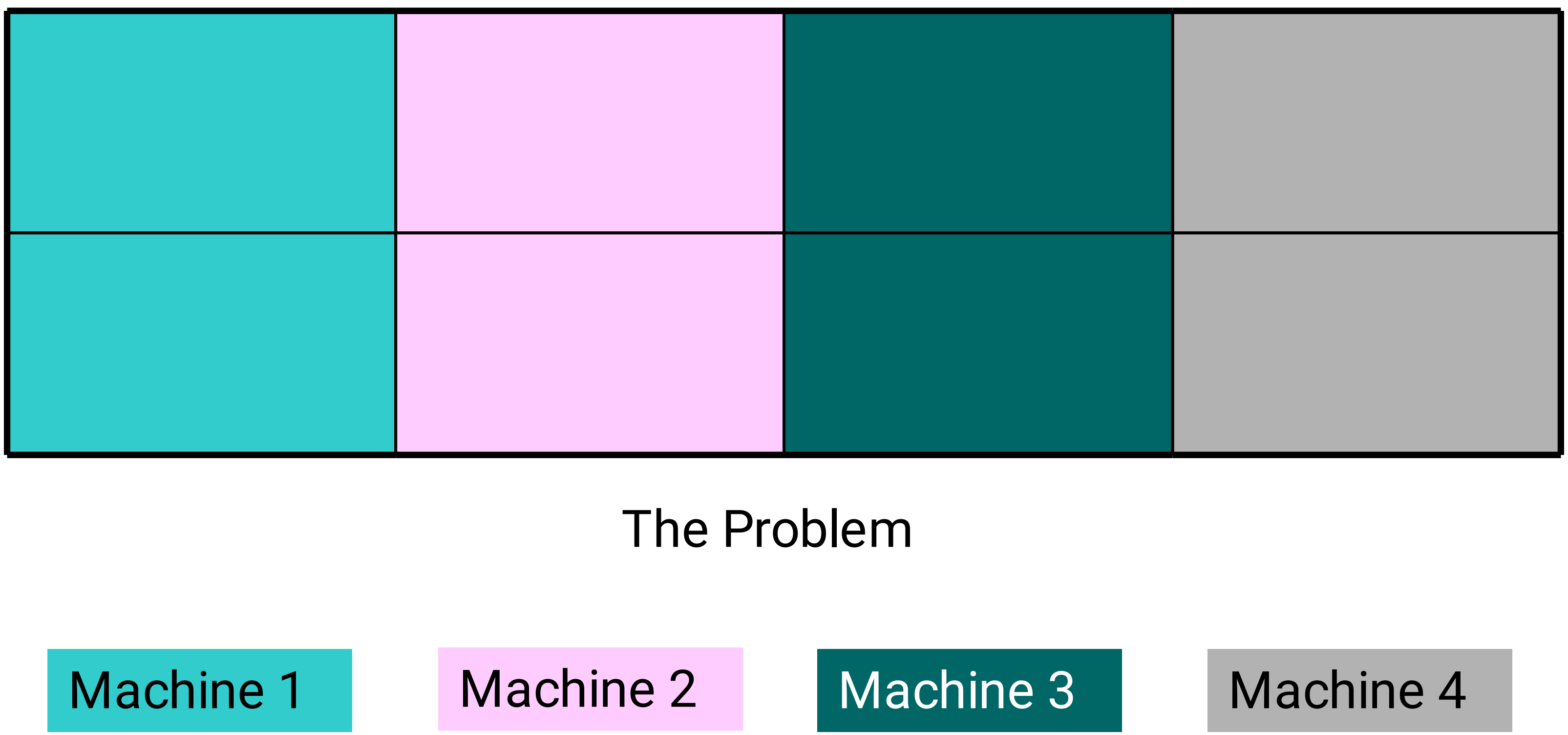
- Tasks are mapped statically to help optimize the cost of data movement among tasks
- This type of parallelism that is naturally expressed by independent tasks in a task-dependency graph is called ***task parallelism***.



Data (Domain) Parallelism

- Partitioning the problem's data domain and distributing portions to multiple processors for simultaneous execution
- Good to use for problems where:
  - data is static
  - dynamic data structure tied to single entity where entity can be subset
  - domain is fixed but computation within various regions of the domain is dynamic

Data Parallelism



Data Parallelism

- In the *data-parallel model,* the tasks are statically or semi-statically mapped onto processes and each task performs similar operations on different data.
- This type of parallelism that is a result of identical operations being applied concurrently on different data items is called ***data parallelism***.
- The work may be done in phases and the data operated upon in different phases may be different.

Data Parallelism

- Data-parallel computation phases are interspersed with interactions to synchronize the tasks or to get fresh data to the tasks.
- The decomposition of the problem into tasks is usually based on data partitioning because a uniform partitioning of data followed by a static mapping is sufficient to guarantee load balance.

Data Parallelism

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

(a)

$$\begin{aligned}
&\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\
&\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
&\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\
&\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{aligned}$$

(b)

Data vs Task Parallelism

- Data parallel computation:
  - Perform the same operation on different items of data at the same time; the parallelism grows with the size of the data.
- Task parallel computation:
  - Perform distinct computations -- or tasks -- at the same time. If the number of tasks is fixed, the parallelism is not scalable.

Data vs Task Parallelism

- FOR each CPU in parallel computing environment
  - Retrieve next task from task queue
  - Create a thread and provide it with the retrieved task
  - Start the created thread
- END FOR

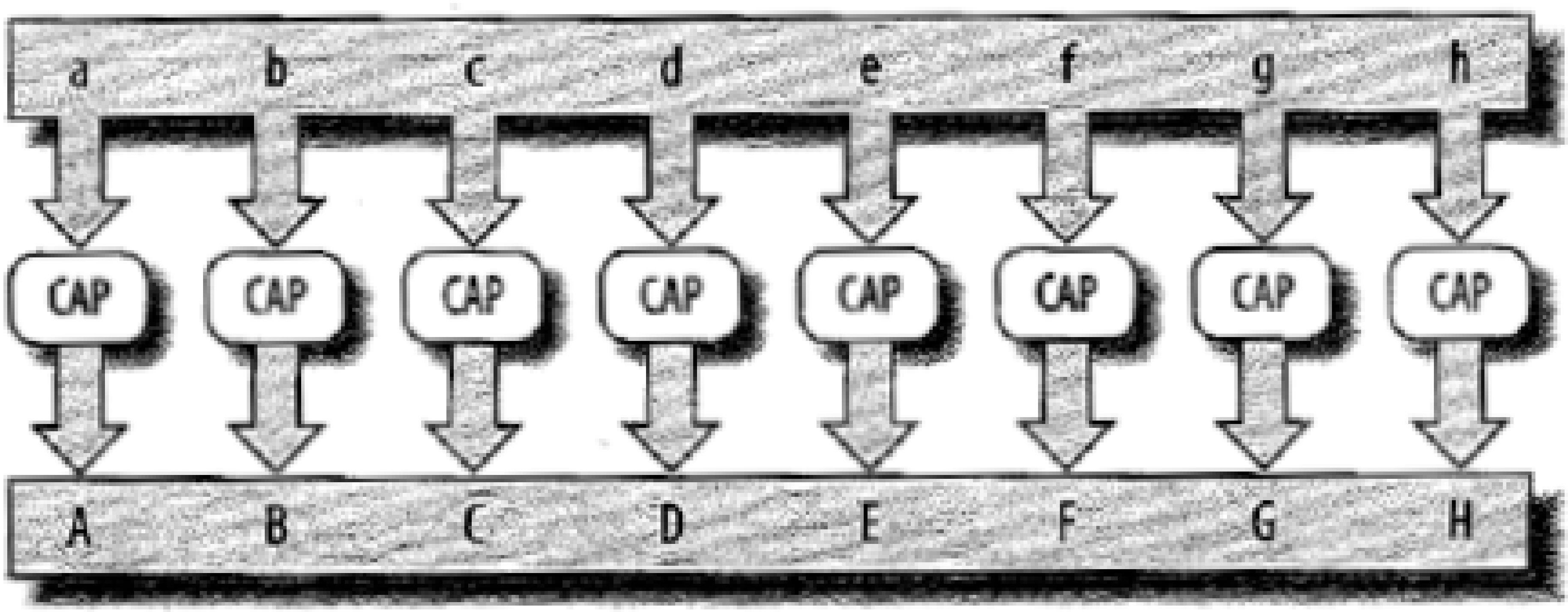
Data vs Task Parallelism

- lower_limit = 0
- upper_limit = 0
- FOR each CPU in parallel computing environment
  - lower_limit = upper_limit + 1
  - upper_limit = upper_limit + round(d.length/ no_of_cpus)
  - Create a thread and provide it with lower_limit and upper_limit data array indexes
  - Start the created thread
- END FOR

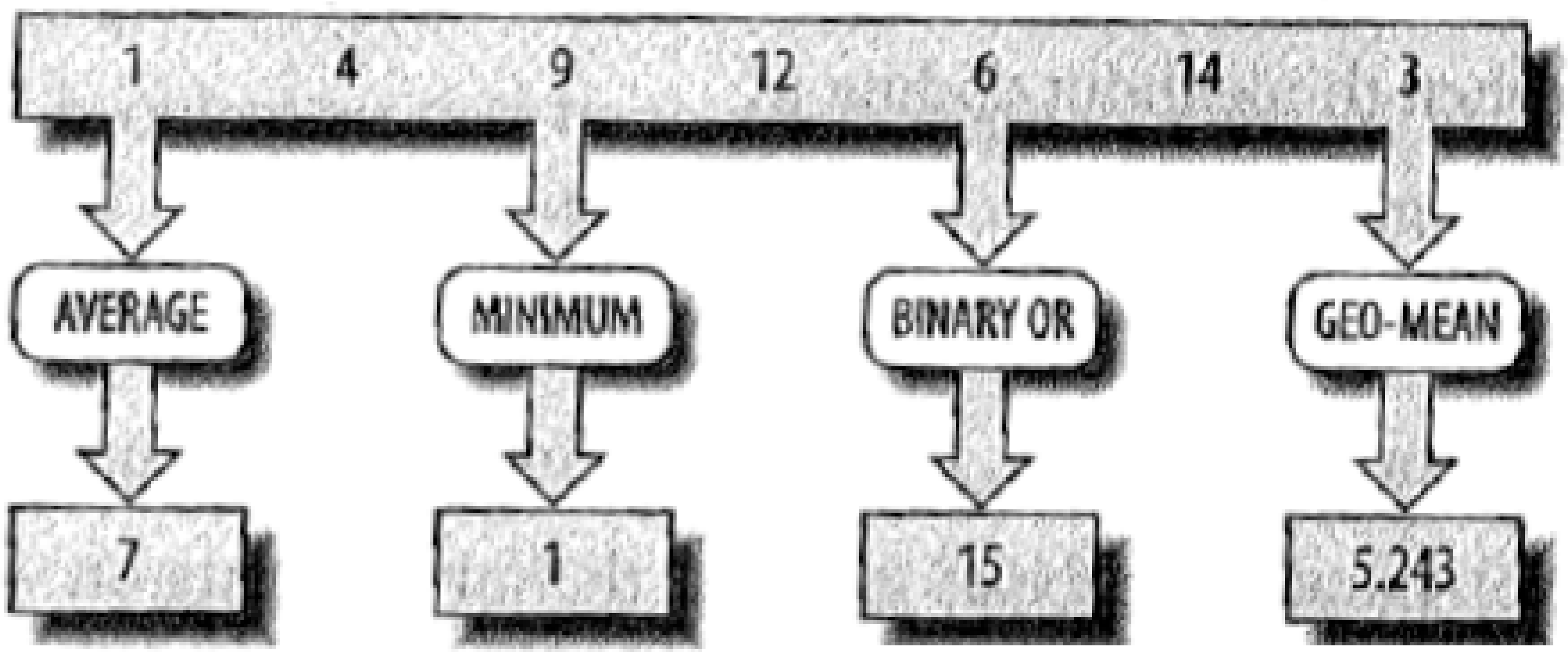
Data vs Task Parallelism

- Data parallelism:
  - The same task run on different data in parallel
- Task parallelism:
  - Different tasks running on the same data
- Hybrid data/task parallelism:
  - A parallel pipeline of tasks, each of which might be data parallel

Data or Task Parallel ??



Data or Task Parallel ??



Data vs Task Parallelism

Data Parallelism	Task Parallelism
Same operations are performed on different subsets of same data structure.	Different operations are performed on the same or different data in parallel to fully utilize the resources.
Synchronous computation	Asynchronous computation
Speed up is more as there is only one execution thread operating on all sets of data.	Speed up is less as each processor will execute a different thread or process on the same or different set of data.

When Data / Task Parallelism

- **Choose the data-parallel threading model for compute-intensive loops, that is, where the same, independent operation is performed repeatedly.**
  - Data parallelism implies that the same independent operation is applied repeatedly to different data.
- **Choose the task-parallel model when independent threads can readily service separate functions.**
  - Task-level concurrency calls for independent work encapsulated in functions to be mapped to individual threads, which execute asynchronously.

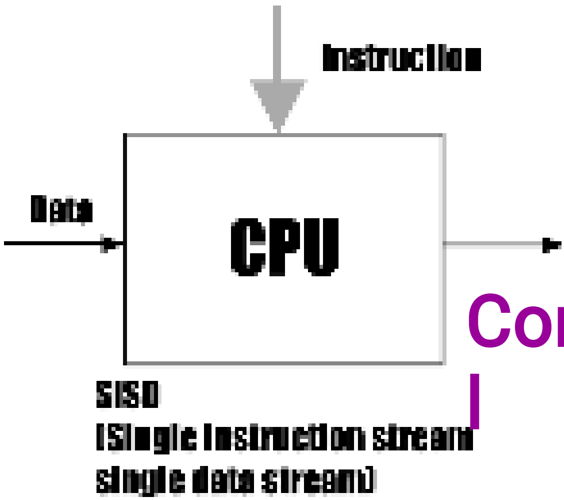
Elements of a Parallel Computer

- Hardware
  - Multiple Processors
  - Multiple Memories
  - Interconnection Network
- System Software
  - Parallel Operating System
  - Programming Constructs to Express/Orchestrate Concurrency
- Application Software
  - Parallel Algorithms

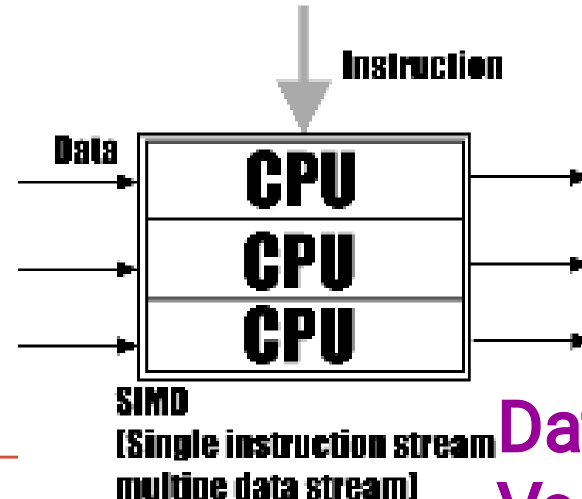
Parallel Computing Platform

- Logical Organization
  - The user's view of the machine as it is being presented via its system software
- Physical Organization
  - The actual hardware architecture
- Physical Architecture is to a large extent independent of the Logical Architecture

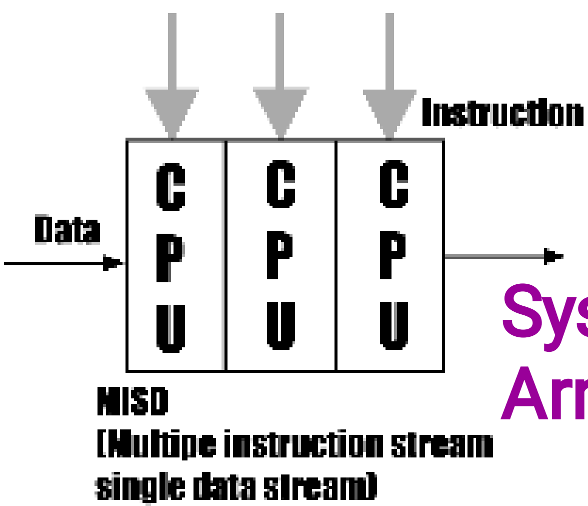
Flynn's Classification



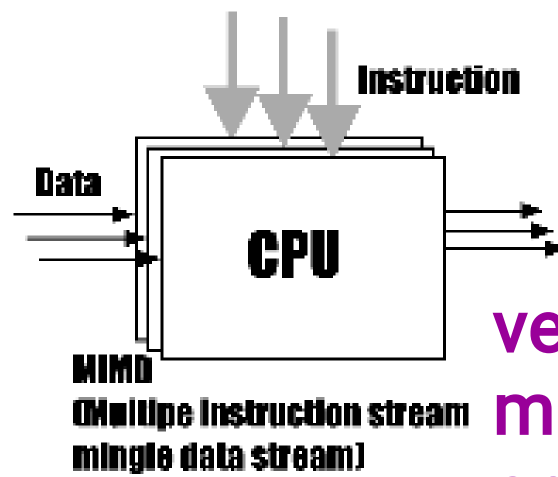
Conventional



Data Parallel,
Vector Computing

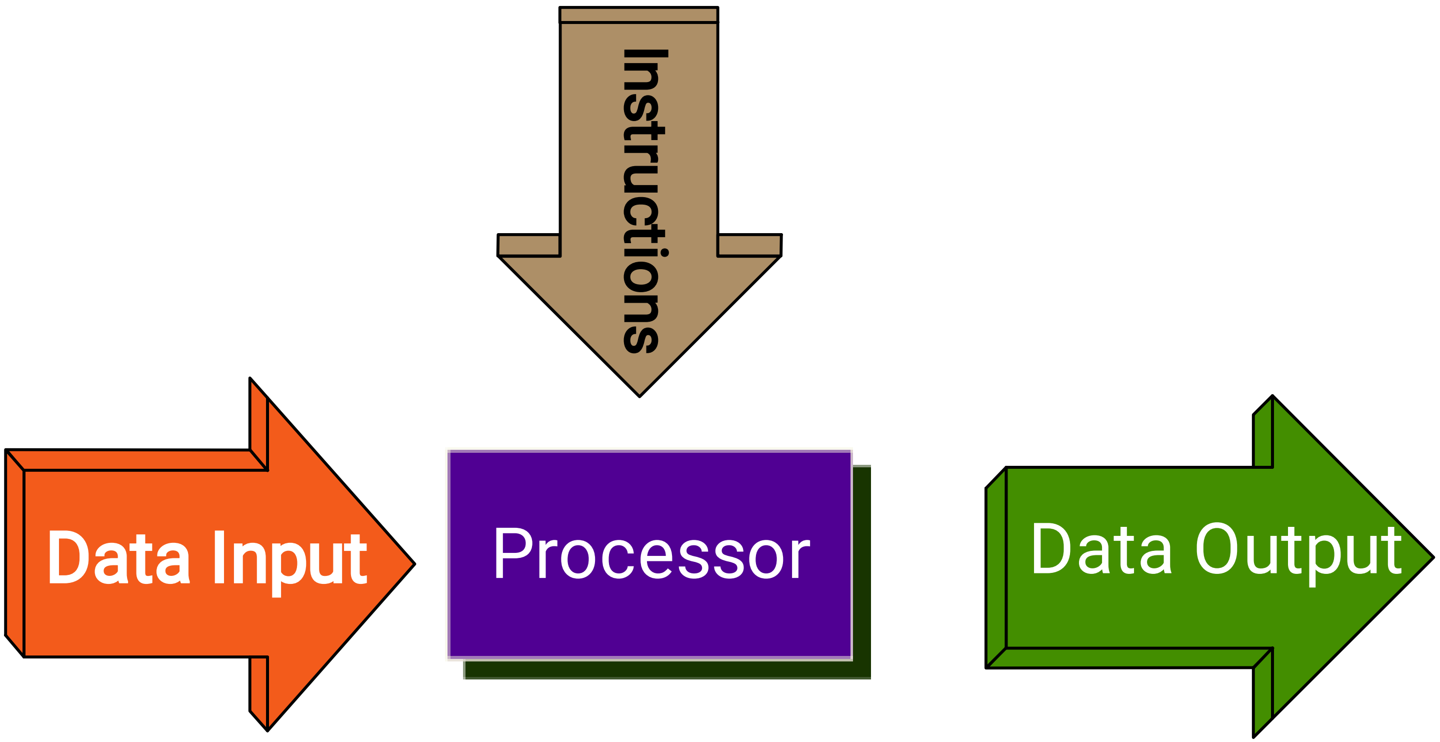


Systolic
Arrays

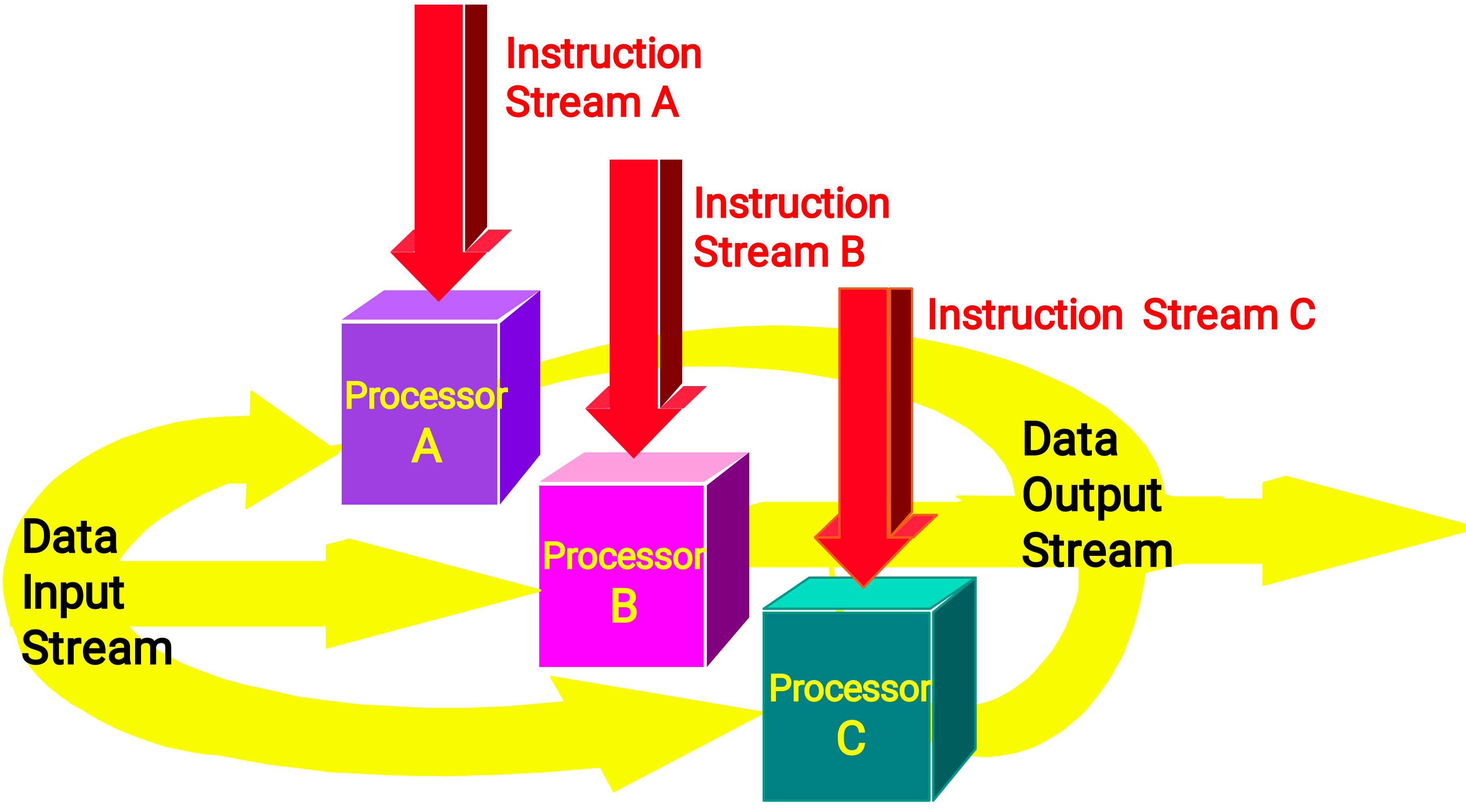


very general,
multiple
approaches

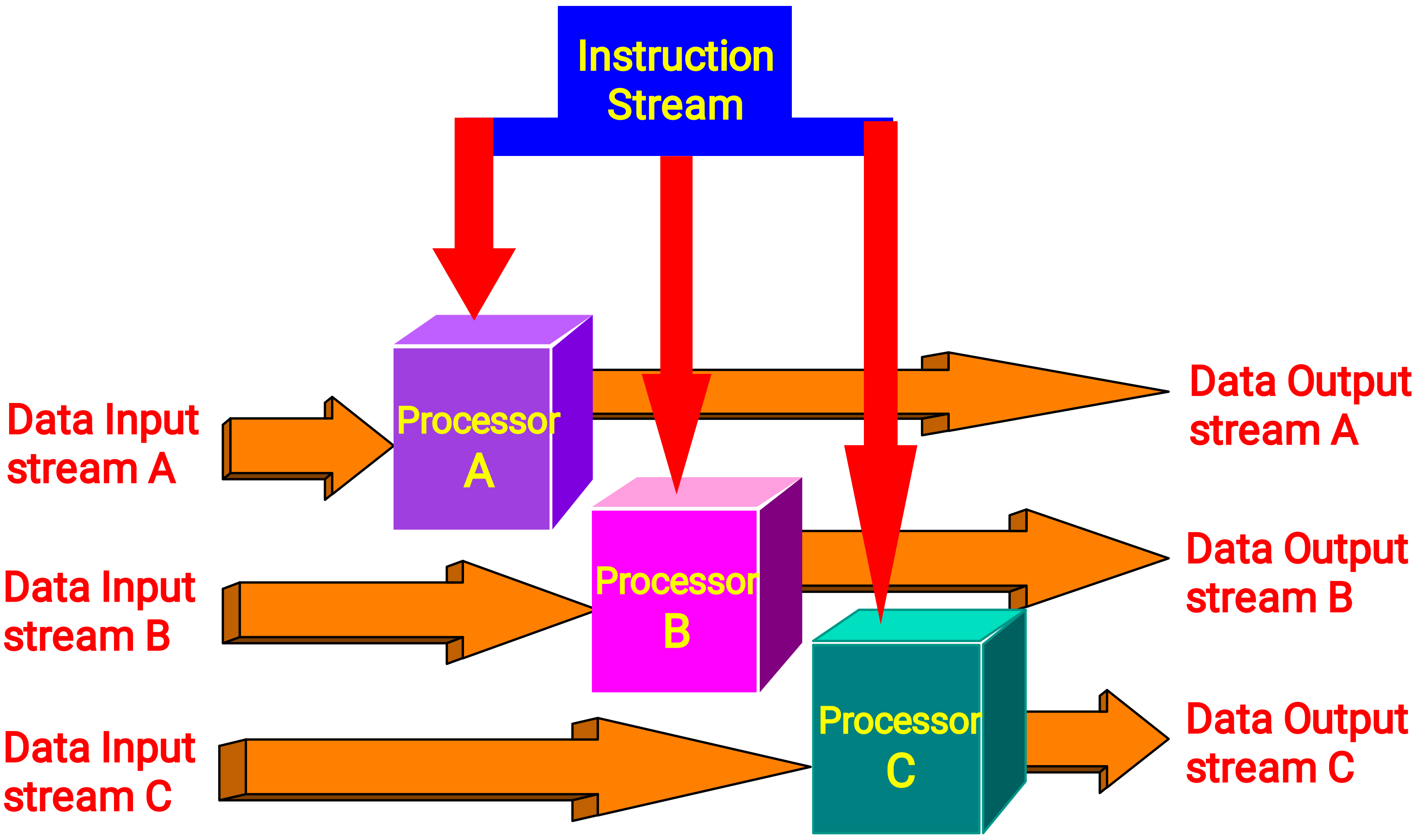
Single Instruction, Single Data (SISD)



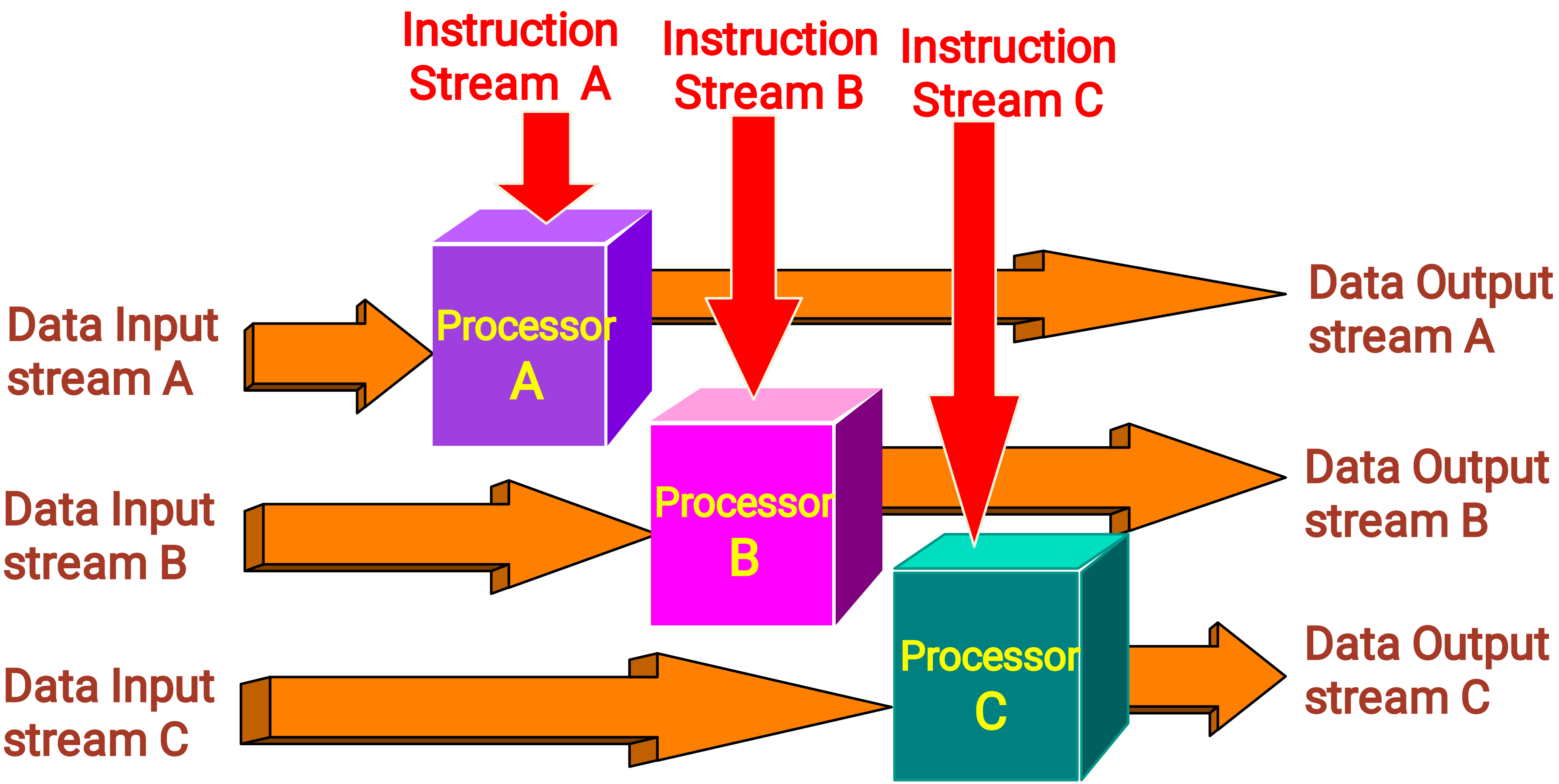
Multiple Instruction, Single Data (MISD)



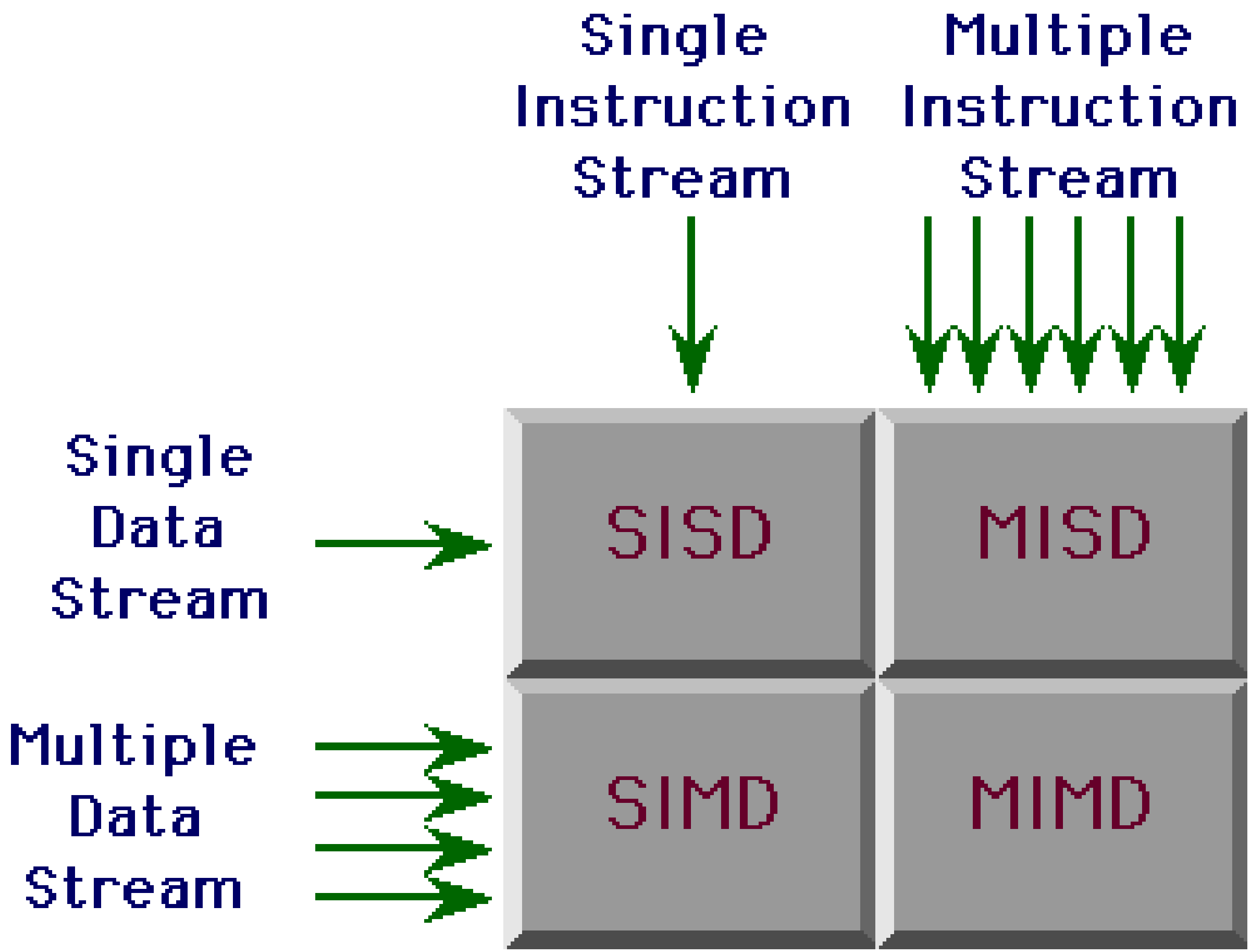
Single Instruction, Multiple Data (SIMD)



Multiple Instruction, Multiple Data (MIMD)



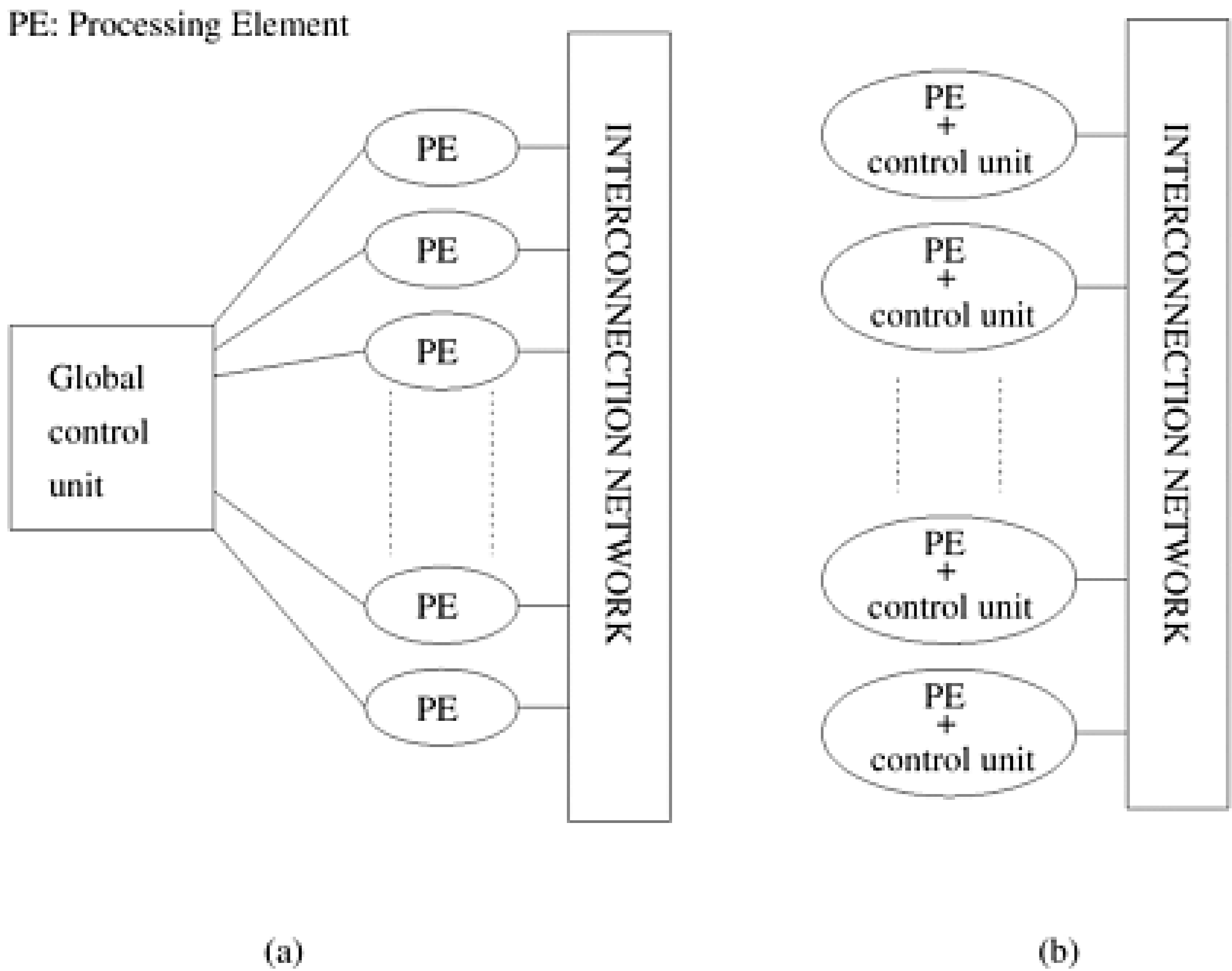
Flynn's Classification



Control Structure of Parallel Programs

- If there is a single control unit that dispatches the same instruction to various processors (that work on different data), the model is referred to as **single instruction stream, multiple data stream (SIMD)**.
- If each processor has its own control unit, each processor can execute different instructions on different data items. This model is called **multiple instruction stream, multiple data stream (MIMD)**.

SIMD and MIMD Processors



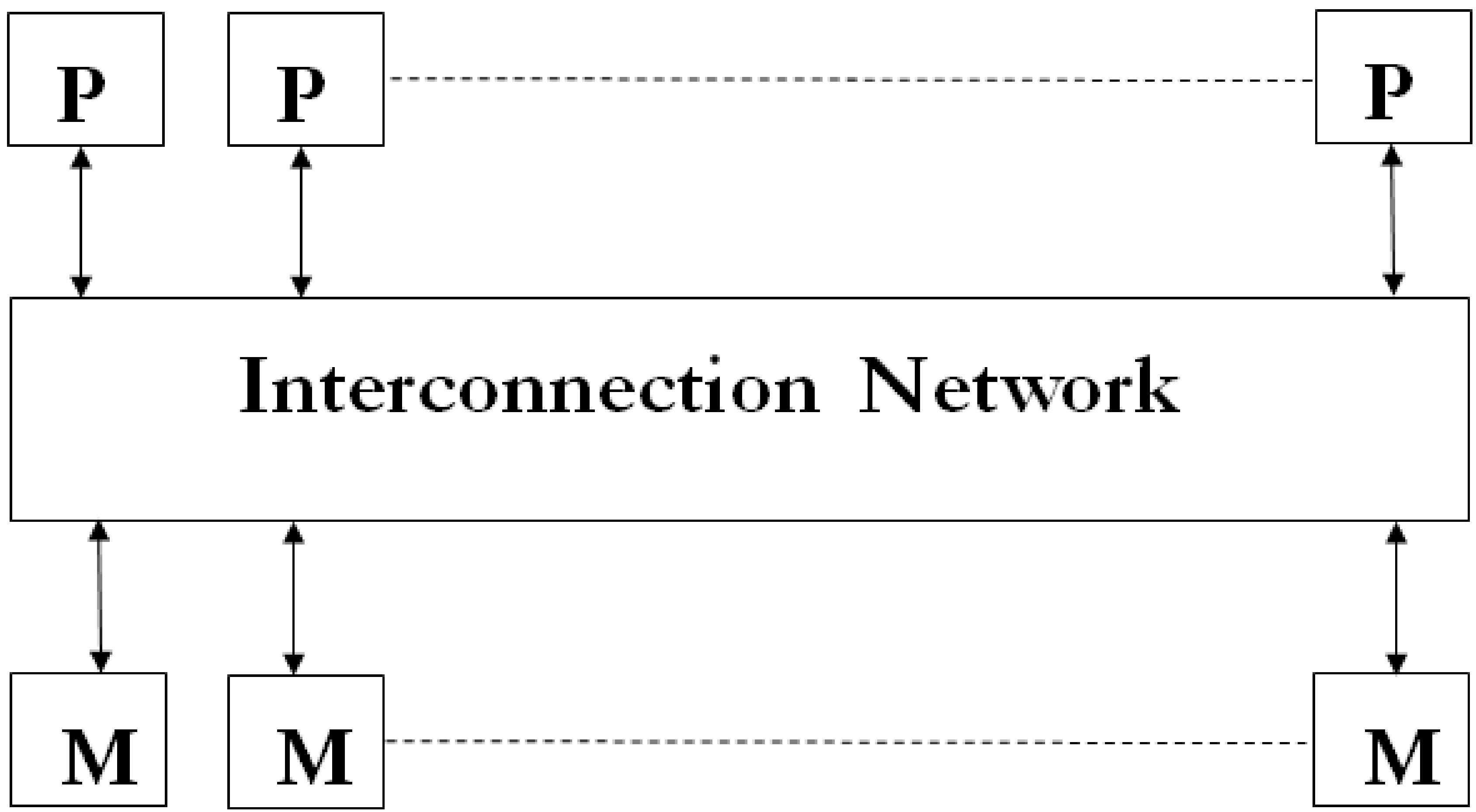
Communication Model of Parallel Platforms

- There are two primary forms of data exchange between parallel tasks - accessing a shared data space and exchanging messages.
- Platforms that provide a shared data space are called **shared-address-space** machines or **multiprocessors**.
- Platforms that support messaging are also called **message passing** platforms or **multicomputers**.

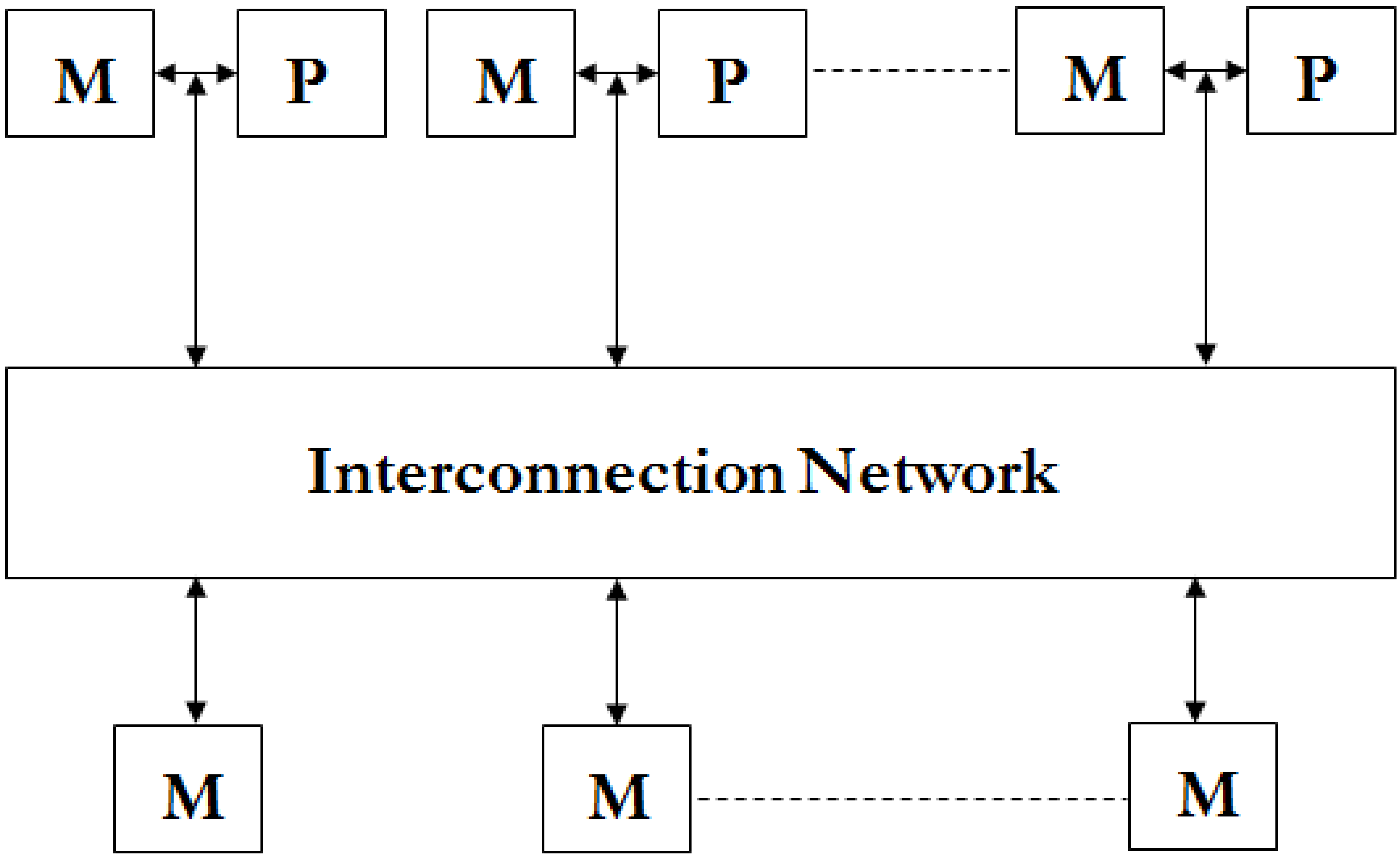
Shared-Address-Space Platforms

- Part (or all) of the memory is accessible to all processors.
- Processors interact by modifying data objects stored in this shared-address-space.
- If the time taken by a processor to access any memory word in the system (global or local) is
  - identical, then the platform is classified as a **uniform memory access (UMA)**,
  - not identical, then its classified as **non-uniform memory access (NUMA)** machine.

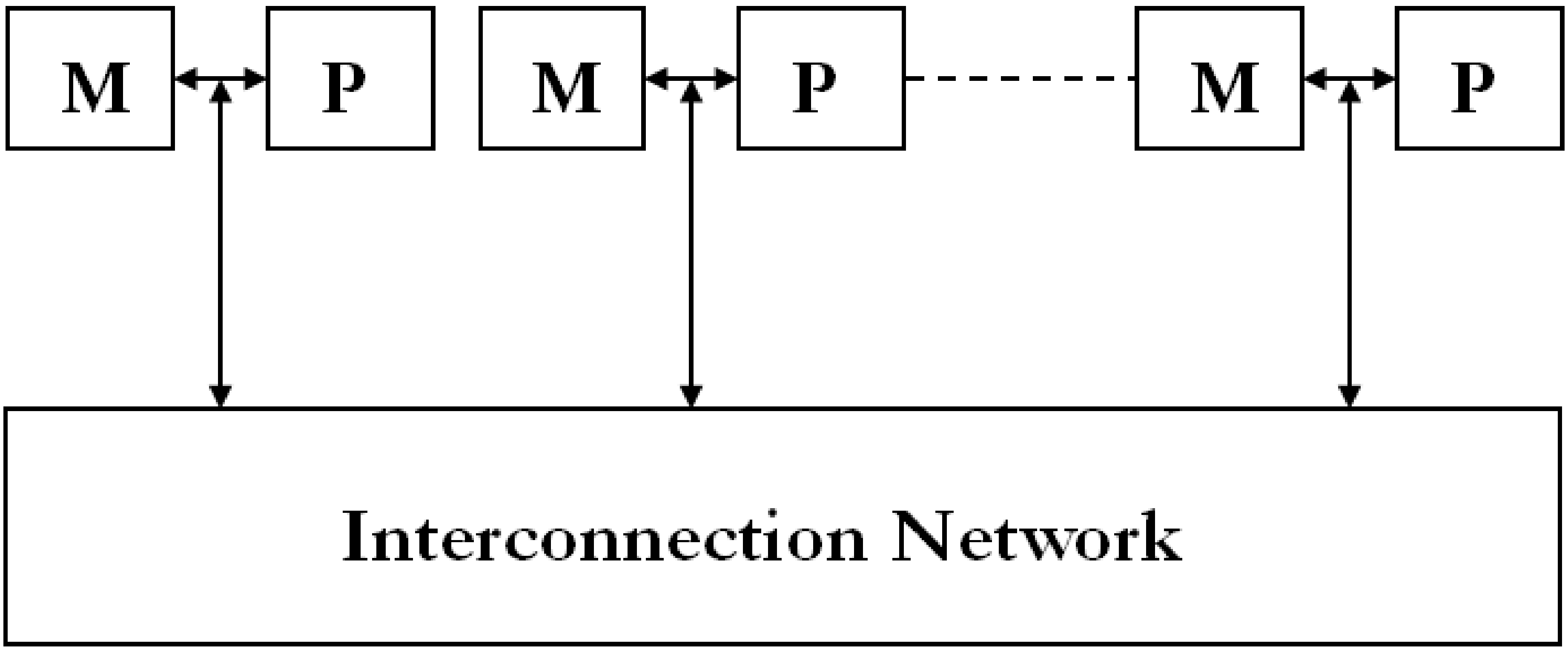
Uniform Memory Access (UMA)



Non-Uniform Memory Access (NUMA) with Local and Global Memories



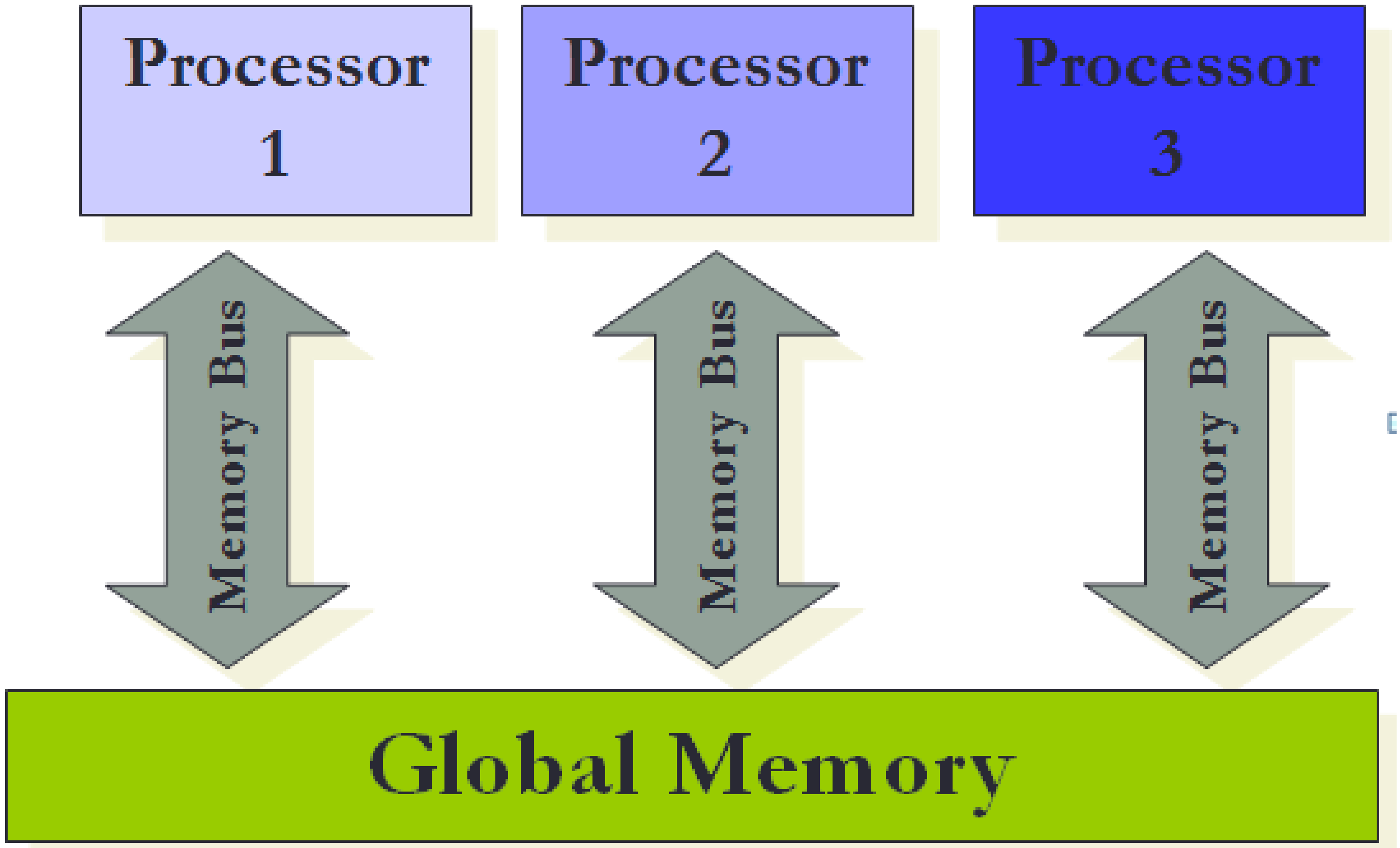
Non-Uniform Memory Access (NUMA) with Local Memory only



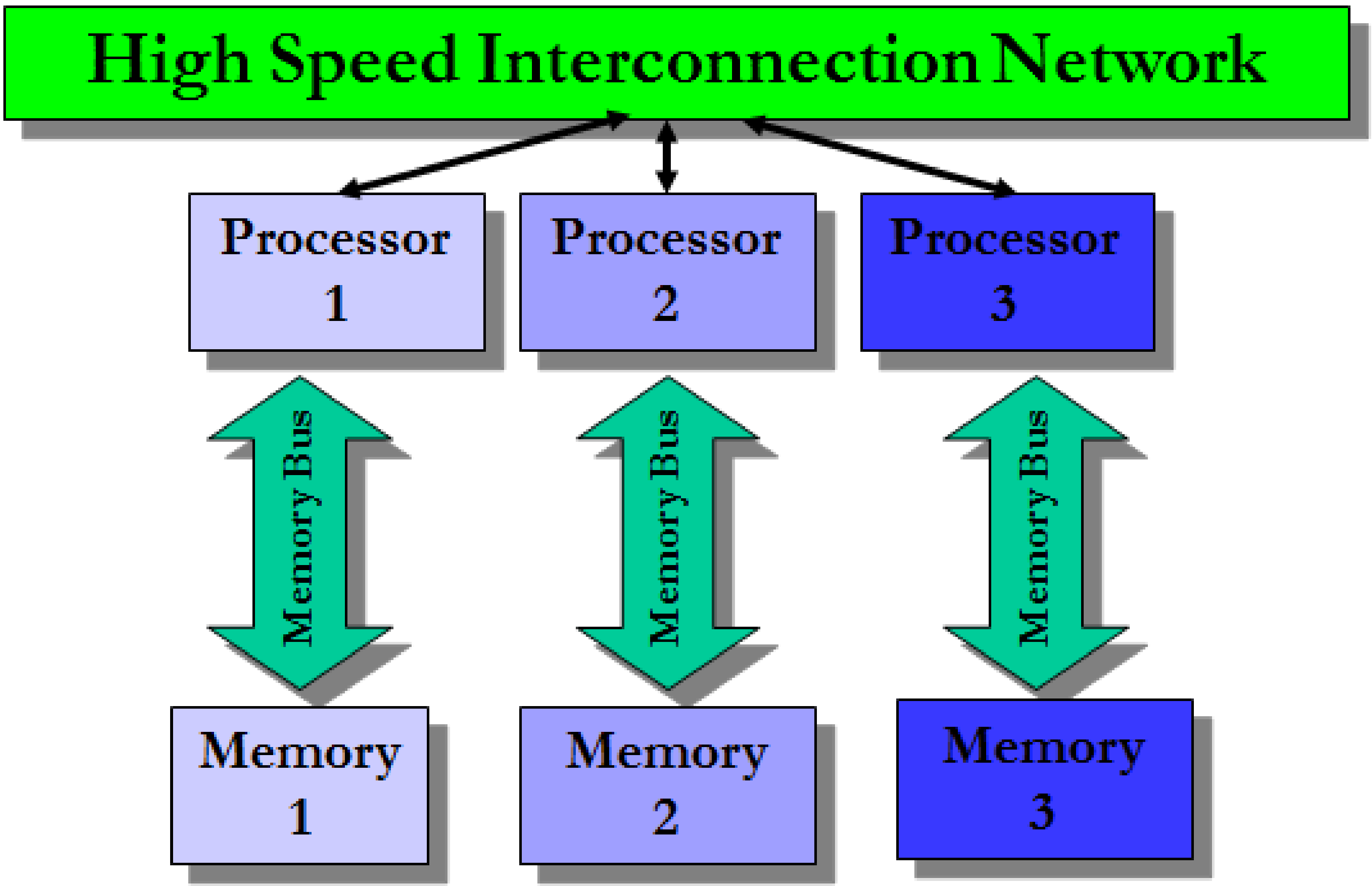
Shared-Address-Space versus Shared Memory

- We refer to '**shared address space**' as a programming abstraction and to '**shared memory**' as a physical machine attribute.
  - **Shared Memory** – memory is physically shared among various processors (Each processor has equal access to any memory segment) … UMA?
  - **Distributed Memory** – different segments of memory are physically associated with different processing elements … NUMA?
- Both models can provide a logical view of shared address space platform.

Shared Memory Architecture



Distributed Memory Architecture



Message-Passing Platforms

- These platforms comprise of a set of processors and their own (exclusive) memory.
- Instances of such a view come naturally from clustered workstations and non-shared-address-space multicomputers.
- Interactions between processes running on different nodes must be accomplished using messages

Message-Passing Platforms

- The exchange of messages is used to transfer data, work and to synchronize actions among the processes
- These platforms are programmed using (variants of) **send** and **receive** primitives.
- Libraries such as MPI and PVM provide such primitives.

Message Passing vs Shared Memory

- Message passing requires little hardware support, other than a network.
- Shared memory platforms can easily emulate message passing.
- The reverse is more difficult to do (in an efficient manner).

Message Passing vs Shared Memory

Aspect	Shared Memory	Message Passing
Communication	Implicit	Explicit
Synchronization	Explicit	Implicit
Hardware Support	Typically Required	Required as number of processors becomes large
Development Effort	Lower	Higher
Communication Granularity	Finer	Coarser