Introduction to High Performance Computing

Unit-I

Syed Rameem Zahra
(Assistant Professor)
Department of CSE, NSUT

What & Why

- What is high performance computing (HPC)?
  - The use of the most efficient algorithms on computers capable of the highest performance to solve the most demanding problems.
- Why HPC?
  - Numerical simulation to predict the behaviour of physical systems.
  - High performance graphics—particularly visualization, and animation.
  - Big data analytics for strategic decision making.
  - Synthesis of molecules for designing medicines.

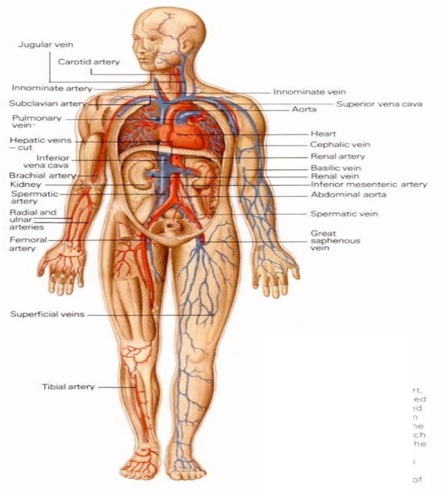
HPC Examples: Blood Flow in Human Vascular Network

- ❑ Cardiovascular disease accounts for about 50% of deaths in western world;
- ❑ Formation of arterial disease strongly correlated to blood flow patterns;

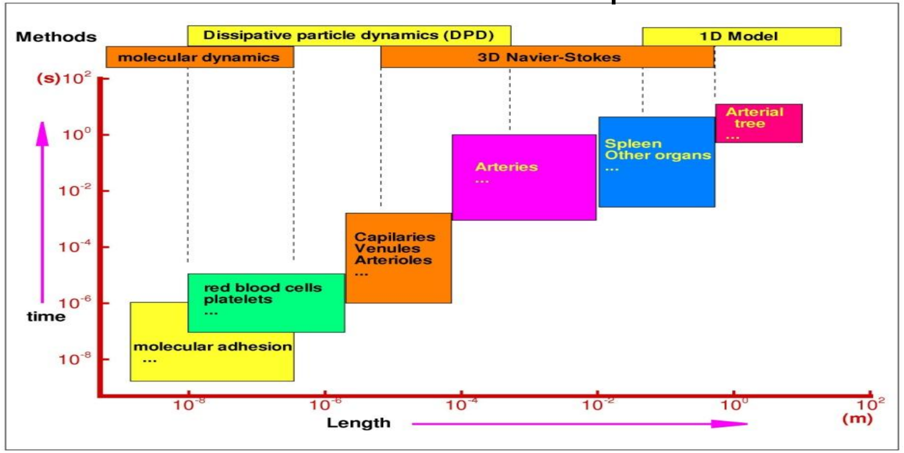
In one minute, the heart pumps the entire blood supply of 5 quarts through 60,000 miles of vessels, that is a quarter of the distance between the moon and the earth

Computational challenges:

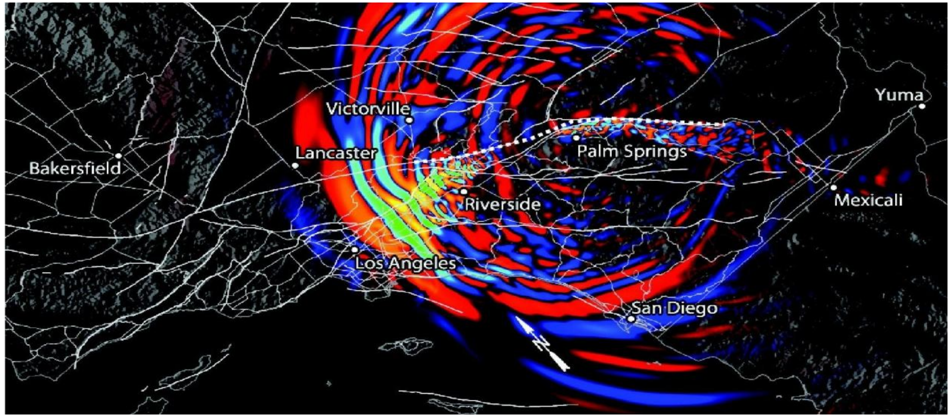
Enormous problem size



Blood flow involves multiple scales

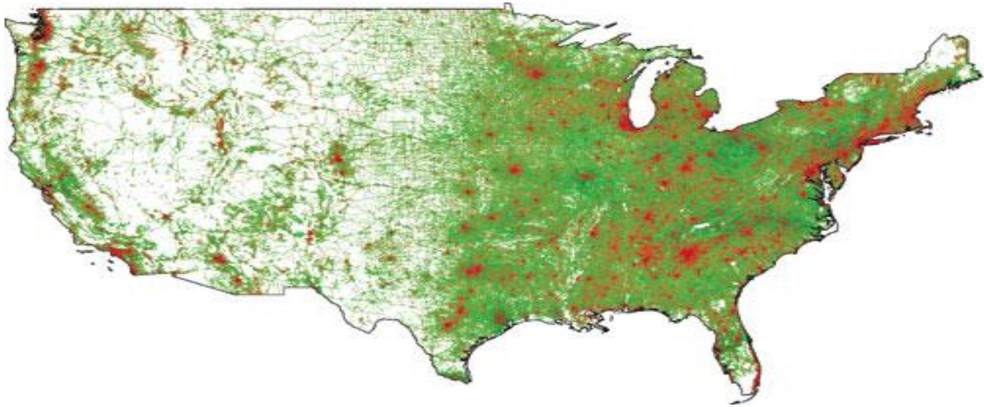


HPC Examples



Earthquake simulation

Surface velocity 75 sec after earthquake



Flu pandemic simulation

300 million people tracked

Density of infected population, **45 days** after breakout

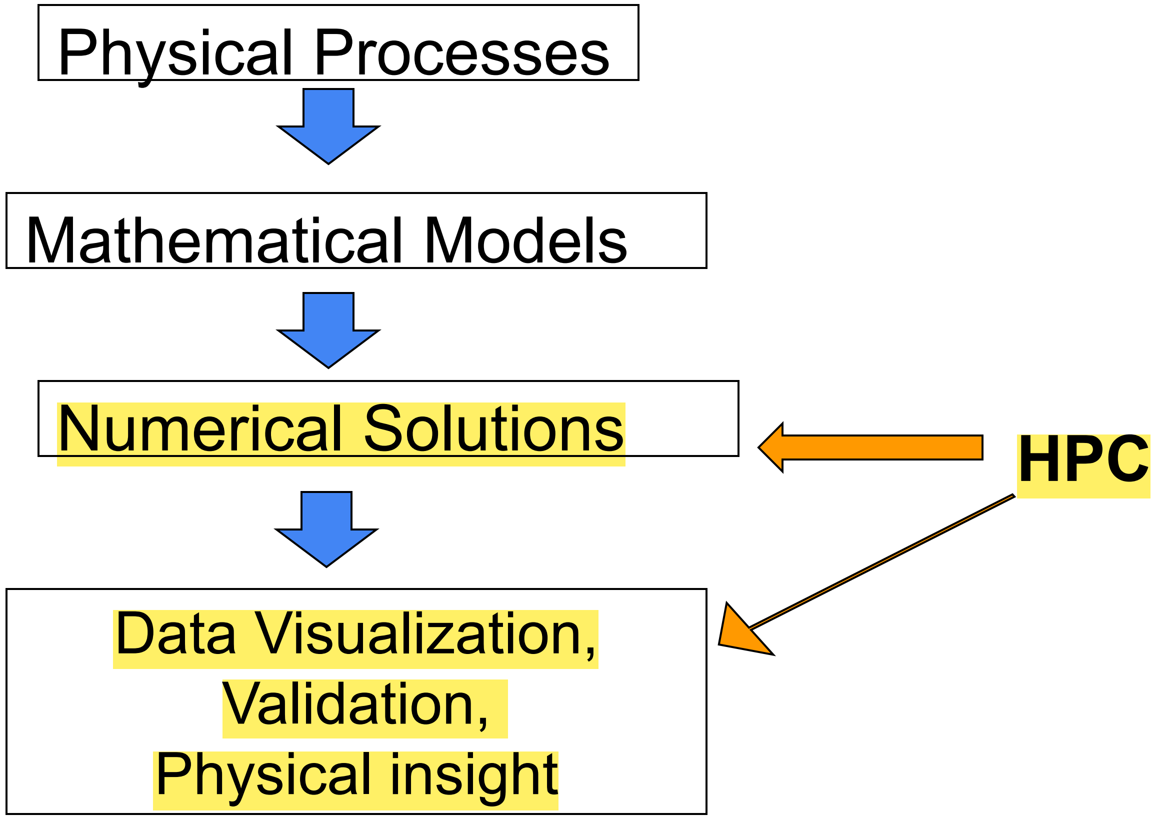
How HPC fits into Scientific Computing

Air flow around an airplane

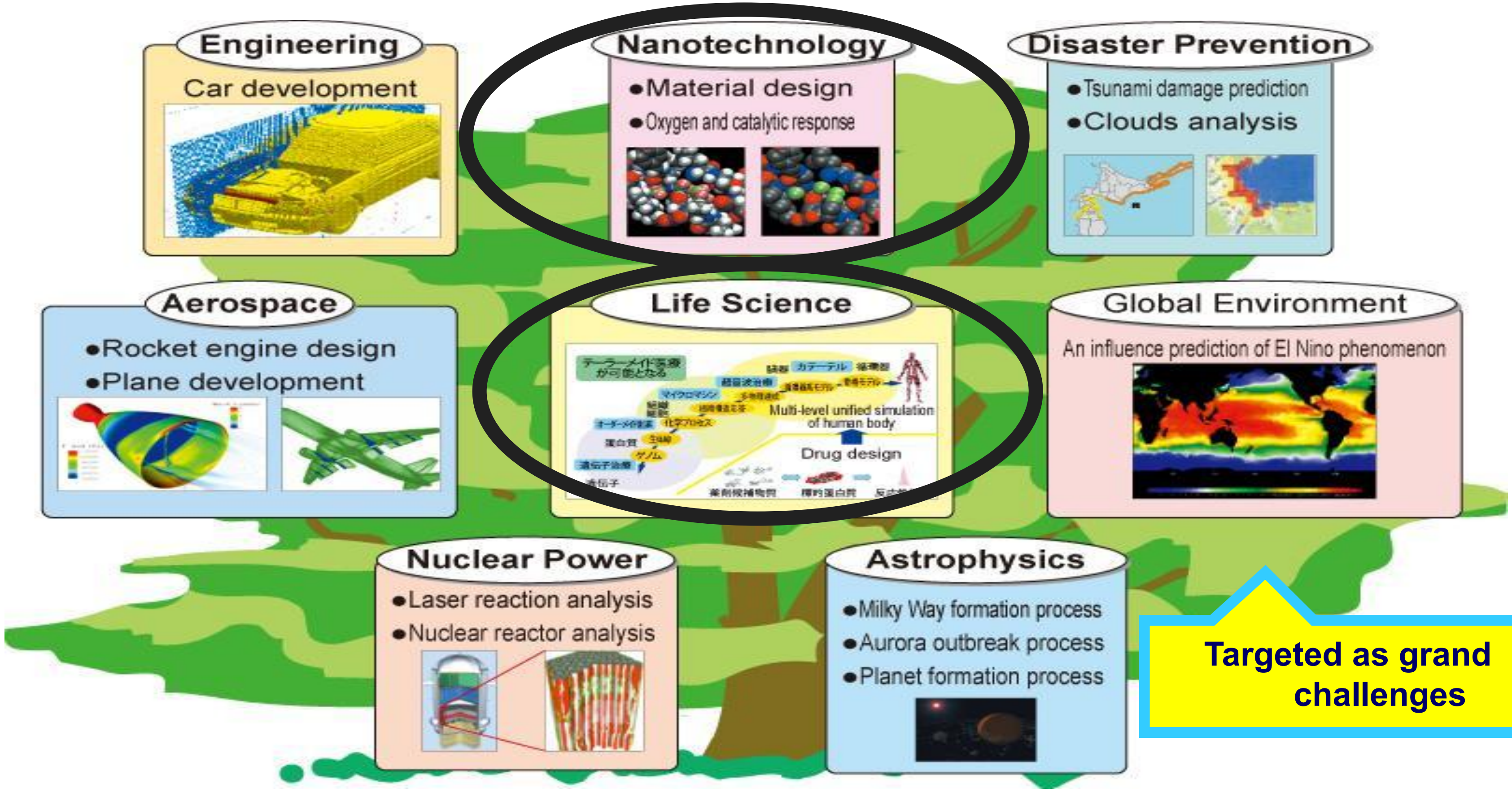
Navier-stokes equations

Algorithms, BCs, solvers, Application codes, supercomputers

Viz software



Major Applications of Next Generation Supercomputer



Targeted as grand challenges

Performance Metrics

- FLOPS, or FLOP/S: **FL**oating-point **O**perations **P**er **S**econd
  - **MFLOPS:** MegaFLOPS, 10^6 flops
  - **GFLOPS:** GigaFLOPS, 10^9 flops, home PC
  - **TFLOPS:** TeraGLOPS, 10^12 flops, present-day supercomputers (www.top500.org)
  - **PFLOPS:** PetaFLOPS, 10^15 flops, by 2011
  - **EFLOPS:** ExaFLOPS, 10^18 flops, by 2020
  - **MIPS=**Mega Instructions per Second = MegaHertz (if only 1IPS)

    Note: von Neumann computer -- 0.00083 MIPS

Performance Metrics

- **Theoretical peak performance (R_theor):** maximum FLOPS a machine can reach in theory.
  - **Clock_rate * no_cpus * no_FPU/CPU**
  - *3GHz, 2 cpus, 1 FPU/CPU, then R_theor = 3x10^9 * 2 = 6 GFLOPS*
- **Real performance (R_real):** FLOPS for specific operations, e.g. vector multiplication
- **Sustained performance (R_sustained):** performance on an application, e.g. CFD

R_sustained << R_real << R_theor

Not uncommon
R_sustained < 10%R_theor

Computer Performance

- CPU operates on data. If no data, CPU has to wait; performance degrades.
  - typical workstation: 3.2GHz CPU, Memory 667MHz. Memory 5 times slower.
  - **Moore's law:** CPU speed doubles every 18 months
  - Memory speed increases much much slower;
- Fast CPU requires sufficiently *fast* memory.
- Rule of thumb: Memory size in GB=R_theor in GFLOPS
  - 1CPU cycle (1 FLOPS) handles 1 byte of data
  - 1MFLOPS needs 1MB of data/memory
  - 1GFLOPS needs 1GB of data/memory

Many "tricks" designed for performance improvement targets the memory

CPU Performance

- Computer time is measured in terms of CPU cycles
  - Minimum time to execute 1 instruction is 1 CPU cycle
- Time to execute a given program:

$$T = n_c \times t_c = n_i \times \frac{n_c}{n_i} \times t_c = n_i \times CPI \times t_c$$

n_c: total number of CPU cycles

n_i: total number of instructions

CPI = n_c/n_i, average **c**ycles **p**er **i**nstruction

t_c: cycle time, for 1GHz, t_c = 1/(10^9Hz) = 10^(-9)sec = 1ns

To Make a Program/Computer Faster…

- **Reduce cycle time t_c:**
  - Increase clock frequency; however, there is a physical limit
  - In 1ns, light travels 30cm
- **Reduce number of instructions n_i:**
  - More efficient algorithms
  - Better compilers
- **Reduce CPI --** The key is parallelism.
  - Instruction-level parallelism. Pipelining technology
  - Internal parallelism, multiple functional units; superscalar processors; multi-core processors
  - External parallelism, multiple CPUs, parallel machine

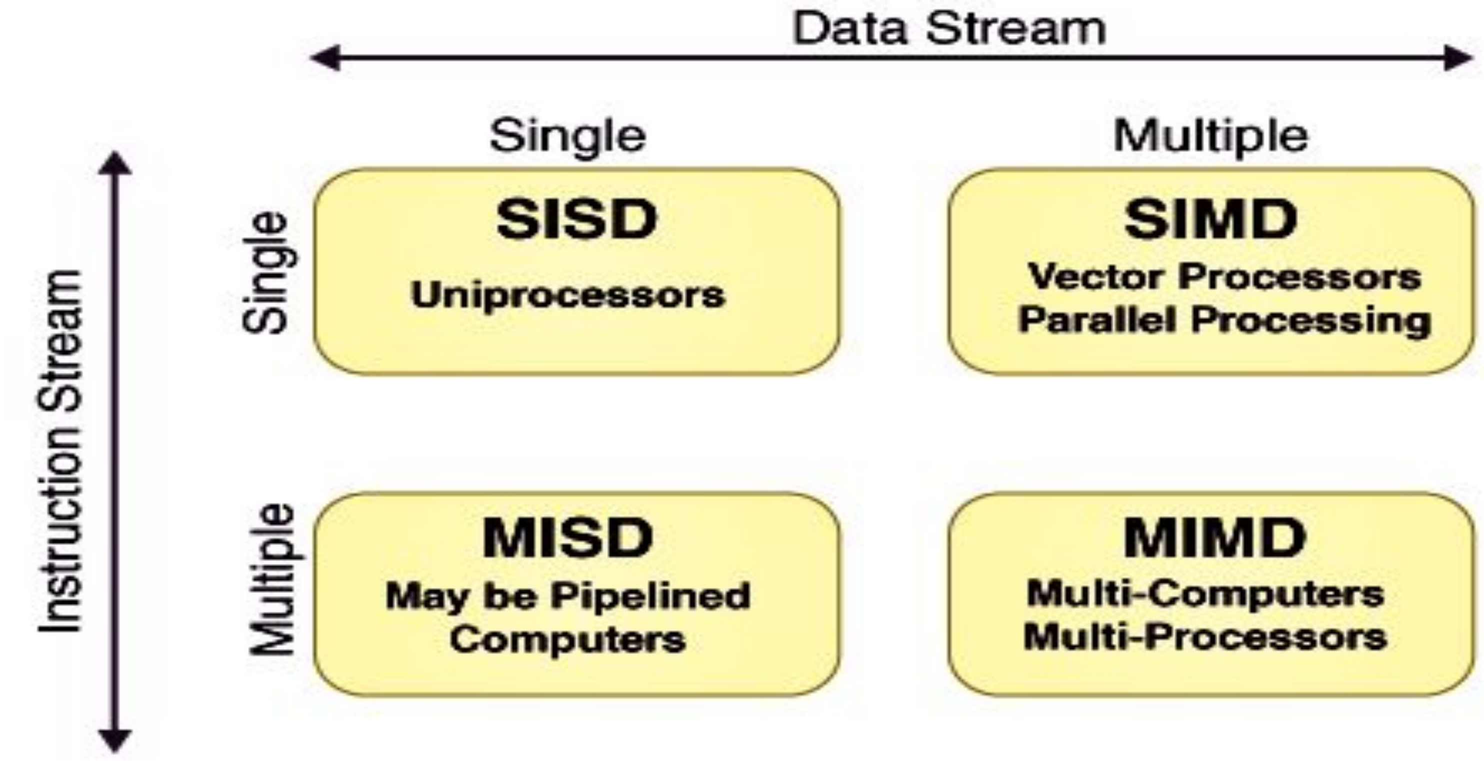
To Make a Program/Computer Faster…

- **use parallelism in a single processor computer**
  - Overlap execution of a number of instructions by pipelining, or by using multiple functional units, or multiple processor “cores”.
  - Overlap operation of different units of a computer.
  - Increase the speed of arithmetic logic unit by exploiting data and/or temporal parallelism.
- **use parallelism in the problem to solve it on a parallel computer.**
  - Use number of interconnected computers to work cooperatively to solve the problem.

Flynn’s Classification of Parallel Architectures

- M.J. Flynn offered a classification for a computer system’s organisation based on the number of instructions as well as data items that are changed at the same time.
- An **instruction stream** is a collection of instructions read from memory. A **data stream** is the result of the actions done on the data in the processor.
- The term ‘**stream**’ refers to the flow of data or instructions.
- ***Parallel processing can happen in the data stream, the instruction stream, or both.***

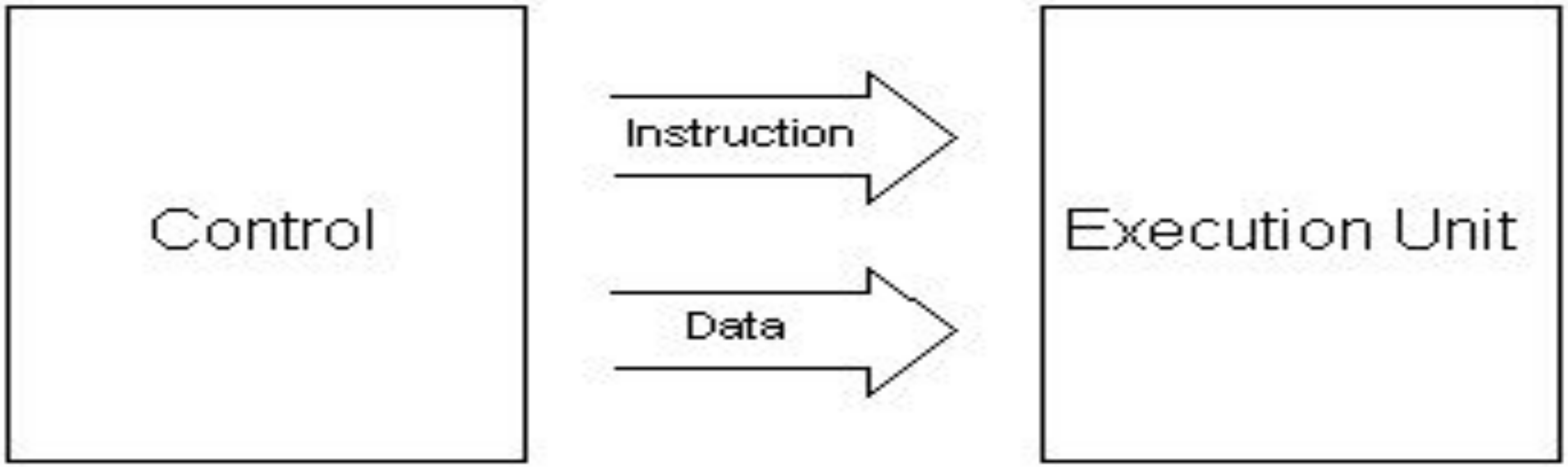
Flynn's Classification of Parallel Architectures



SISD

- They are also called scalar processor i.e., one instruction at a time and each instruction have only one set of operands.
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle.
- Characteristics:
  - Single data: only one data stream is being used as input during any one clock cycle.
  - Deterministic execution.
  - Instructions are executed sequentially.
  - This is the oldest and until recently, the most prevalent form of computer.
- Example: most PCs,single CPU workstations and mainframes.

SISD



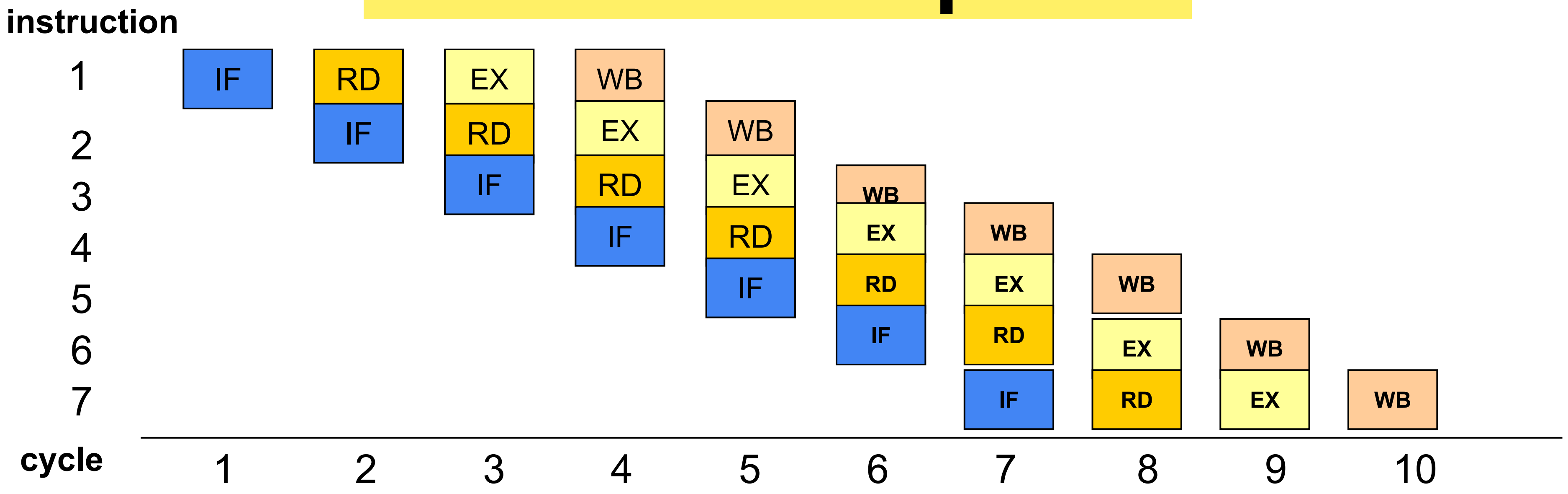
SISD Bottleneck

- Level of Parallelism is low
  - Data dependency
  - Control dependency
- Limitation improvements
  - Pipeline
  - Super scalar
  - Super-pipeline scalar

Pipelining

- Overlapping execution of multiple instructions
  - 1 instruction per cycle
- Sub-divide instruction into multiple stages; Processor handles different stages of adjacent instructions simultaneously
- Suppose 4 stages in instruction:
  - Instruction fetch and decode (IF)
  - Read data (RD)
  - Execute (EX)
  - Write-back results (WB)

Instruction Pipeline



Depth of pipeline: number of stages in an instruction

After the pipeline is full, 1 result per cycle!, in pipelined system **CPI = (n+depth-1)/n**

With pipeline, 7 instructions take 10 cycles. If no pipeline, 7 instructions take 28 cycles

Inhibitors of Pipelining

- ❑ Dependencies between instructions interrupts pipelining, degrading performance
  - ❖ Control dependence.
  - ❖ Data dependence.
  - ❖ Structural dependency
- ❑ They are Pipeline hazards

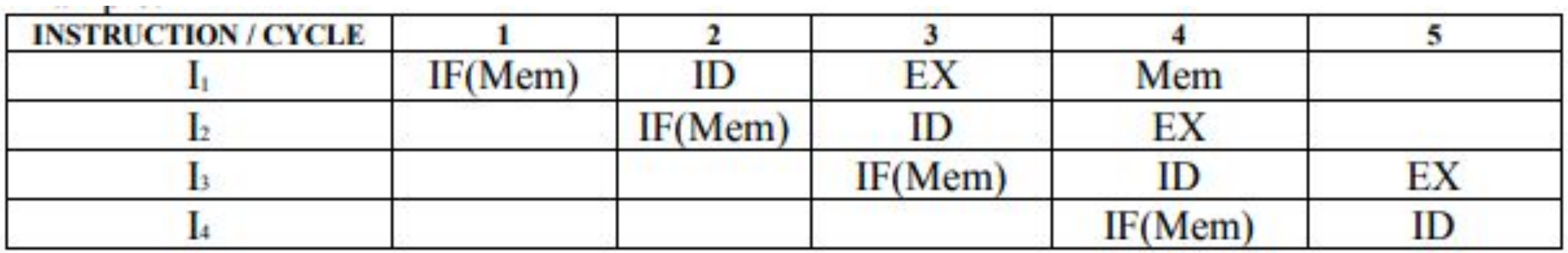
Pipeline hazards

- These are delays in pipeline execution of instructions due to non-ideal conditions.
  - Non-ideal conditions include:
  - Available resources in a processor are limited.
  - Successive instructions are not independent of one another. The result generated by an instruction may be required by the next instruction.
  - All programs have branches and loops. Execution of a program is thus not in a "straight line". An ideal pipeline assumes a continuous flow of tasks.
- Delays due to resource constraints is known as **structural hazard**.
- Delays due to data dependency between instructions is known as **data hazard**.
- Delays due to branch instructions or control dependency in a program is known as **control hazard**.

Structural Dependency

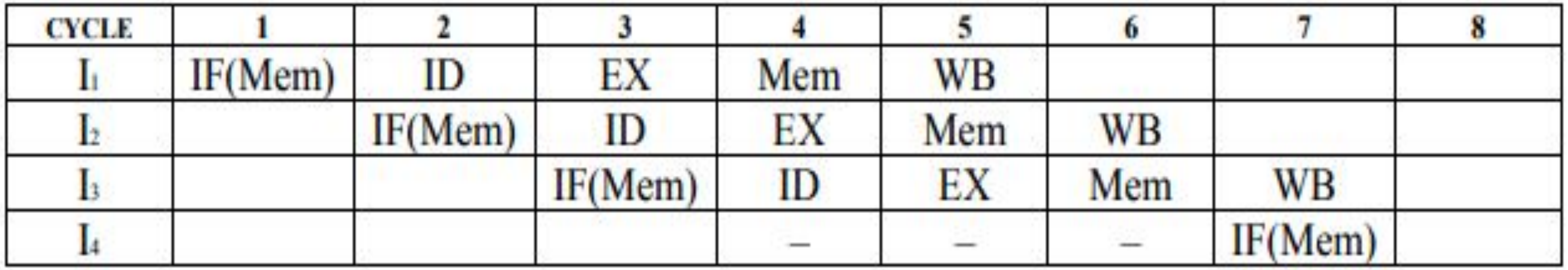
- This dependency arises due to the resource conflict in the pipeline.
- A resource conflict is a situation when more than one instruction tries to access the same resource in the same cycle.
- A resource can be a register, memory, or ALU.
- **Solution for structural dependency:**
    - To minimize structural dependency stalls in the pipeline, we use a hardware mechanism called Renaming.
    - **Renaming:** According to renaming, we divide the memory into two independent modules used to store the instruction and data separately called Code memory (CM) and Data memory (DM) respectively.
    - CM will contain all the instructions and DM will contain all the operands that are required for the instructions.

Structural Dependency: Example



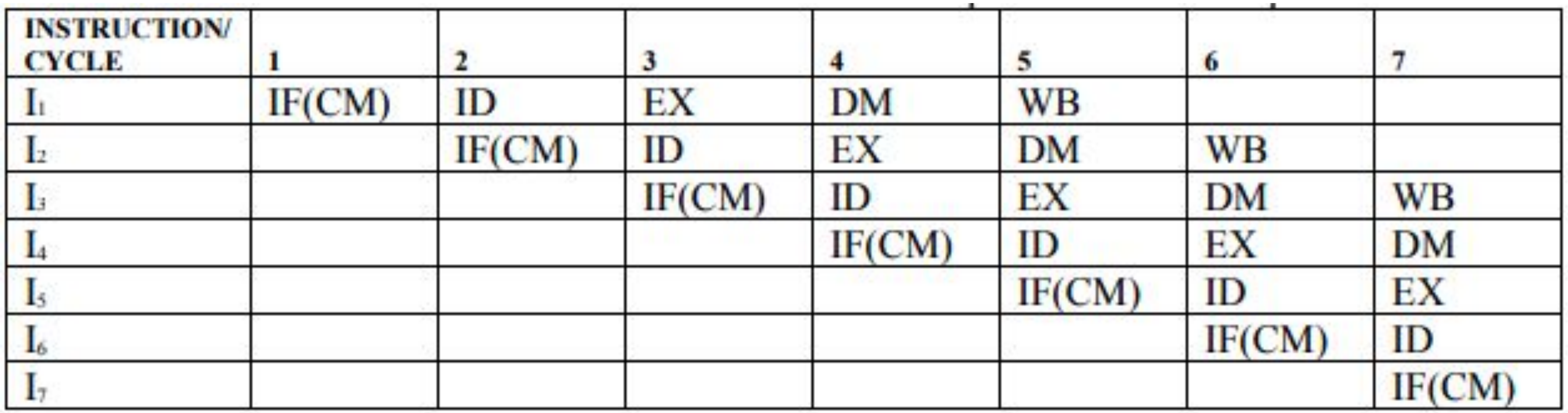
INSTRUCTION / CYCLE	1	2	3	4	5
I_1	IF(Mem)	ID	EX	Mem	
I_2		IF(Mem)	ID	EX	
I_3			IF(Mem)	ID	EX
I_4				IF(Mem)	ID

- In the above scenario, in cycle 4, instructions I1 and I4 are trying to access same resource (Memory) which introduces a resource conflict.
- To avoid this problem, we have to keep the instruction on wait until the required resource (memory in our case) becomes available. This wait will introduce stalls in the pipeline as shown below:



CYCLE	1	2	3	4	5	6	7	8
I_1	IF(Mem)	ID	EX	Mem	WB			
I_2		IF(Mem)	ID	EX	Mem	WB		
I_3			IF(Mem)	ID	EX	Mem	WB	
I_4				–	–	–	IF(Mem)	

Structural Dependency Solution: Example



INSTRUCTION/ CYCLE	1	2	3	4	5	6	7
I_1	IF(CM)	ID	EX	DM	WB		
I_2		IF(CM)	ID	EX	DM	WB	
I_3			IF(CM)	ID	EX	DM	WB
I_4				IF(CM)	ID	EX	DM
I_5					IF(CM)	ID	EX
I_6						IF(CM)	ID
I_7							IF(CM)

Control Dependence

- **Branching:** when an instruction occurs after an conditional branch; so it is unknown whether that instruction will be executed beforehand
  - Loop: `for(i=0;i<n;i++)…; do…enddo`
  - Jump: goto …
  - Condition: if…else…

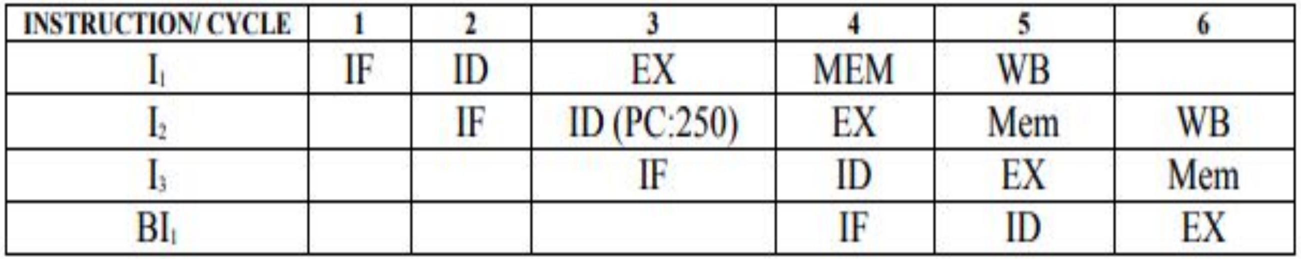
```
if(x>y) n=5;
```

Branching in programs interrupts pipeline degrades performance

Avoid excessive branching!

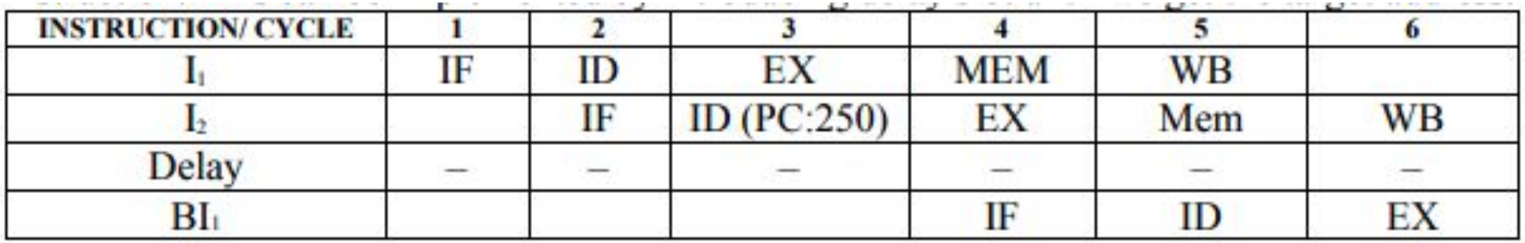
Control Dependence: Example

- Consider the following sequence of instructions in the program:
    - 100: I1
    - 101: I2 (JMP 250)
    - 102: I3
    - .
    - .
    - 250: BI1



INSTRUCTION/ CYCLE	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
I_3			IF	ID	EX	Mem
BI_1				IF	ID	EX

- Expected output: I1 -> I2 -> BI1
- To eliminate this problem we need to stop the Instruction fetch until we get target address of branch instruction. This can be implemented by introducing delay slot until we get the target address.



INSTRUCTION/ CYCLE	1	2	3	4	5	6
I_1	IF	ID	EX	MEM	WB	
I_2		IF	ID (PC:250)	EX	Mem	WB
Delay	–	–	–	–	–	–
BI_1				IF	ID	EX

Control Dependence

- **Solution for Control dependency:**
- **Branch Prediction** is the method through which stalls due to control dependency can be eliminated.
- In this at 1st stage prediction is done about which branch will be taken.
- For branch prediction Branch penalty is zero.
- Branch penalty: The number of stalls introduced during the branch operations in the pipelined processor is known as branch penalty
- Total number of stalls introduced in the pipeline due to branch instructions = Branch frequency * Branch Penalty

Data Dependence

- when an instruction depends on data from a previous instruction

```
x = 3*j;
y = x+5.0; // depends on previous instruction
```

Data Hazards

- Data hazards occur when instructions that exhibit data dependence, modify data in different stages of a pipeline. Hazard cause delays in the pipeline.
- There are mainly three types of data hazards:
  - **RAW** (Read after Write) [**Flow/True data dependency**]
  - **WAR** (Write after Read) [**Anti-Data dependency**]
  - **WAW** (Write after Write) [**Output data dependency**]

Examples:

1) RAW hazard occurs when instruction J tries to read data before instruction I writes it.

Eg:
I: R2 <- R1 + R3
J: R4 <- R2 + R3

2) WAR hazard occurs when instruction J tries to write data before instruction I reads it.

Eg:
I: R2 <- R1 + R3
J: R3 <- R4 + R5

3) WAW hazard occurs when instruction J tries to write output before instruction I writes it.

Eg:
I: R2 <- R1 + R3
J: R2 <- R4 + R5

WAR and WAW hazards occur during the out-of-order execution of the instructions.

Data Dependence: Solution

- To minimize data dependency stalls in the pipeline, operand forwarding is used.
- **Operand Forwarding:** In operand forwarding, we use the interface registers present between the stages to hold intermediate output so that dependent instruction can access new value from the interface register directly

Types of pipeline

- **1. Uniform delay pipeline:**
- In this type of pipeline, all the stages will take same time to complete an operation.
- In uniform delay pipeline, ***Cycle Time (Tp) = Stage Delay***
- If buffers are included between the stages then,
- Cycle Time (Tp) = Stage Delay + Buffer Delay
- **2. Non-Uniform delay pipeline:**
- In this type of pipeline, different stages take different time to complete an operation.
- In this type of pipeline, ***Cycle Time (Tp) = Maximum (Stage Delay)***
- For example, if there are 4 stages with delays, 1 ns, 2 ns, 3 ns, and 4 ns, then Tp = Maximum (1 ns, 2 ns, 3 ns, 4 ns) = 4 ns
- If buffers are included between the stages, Tp = Maximum (Stage delay + Buffer delay)

Performance of pipeline with stalls

Speed Up (S) = $\text{Performance}_{\text{pipeline}}$ / $\text{Performance}_{\text{non-pipeline}}$

=> S = $\text{Average Execution Time}_{\text{non-pipeline}}$ / $\text{Average Execution Time}_{\text{pipeline}}$

=> S = $\text{CPI}_{\text{non-pipeline}}$ * $\text{Cycle Time}_{\text{non-pipeline}}$ / $\text{CPI}_{\text{pipeline}}$ * $\text{Cycle Time}_{\text{pipeline}}$

Ideal CPI of the pipelined processor is ‘1’. But due to stalls, it becomes greater than ‘1’.

=> S = $\text{CPI}_{\text{non-pipeline}}$ * $\text{Cycle Time}_{\text{non-pipeline}}$ / (1 + Number of stalls per Instruction) * $\text{Cycle Time}_{\text{pipeline}}$

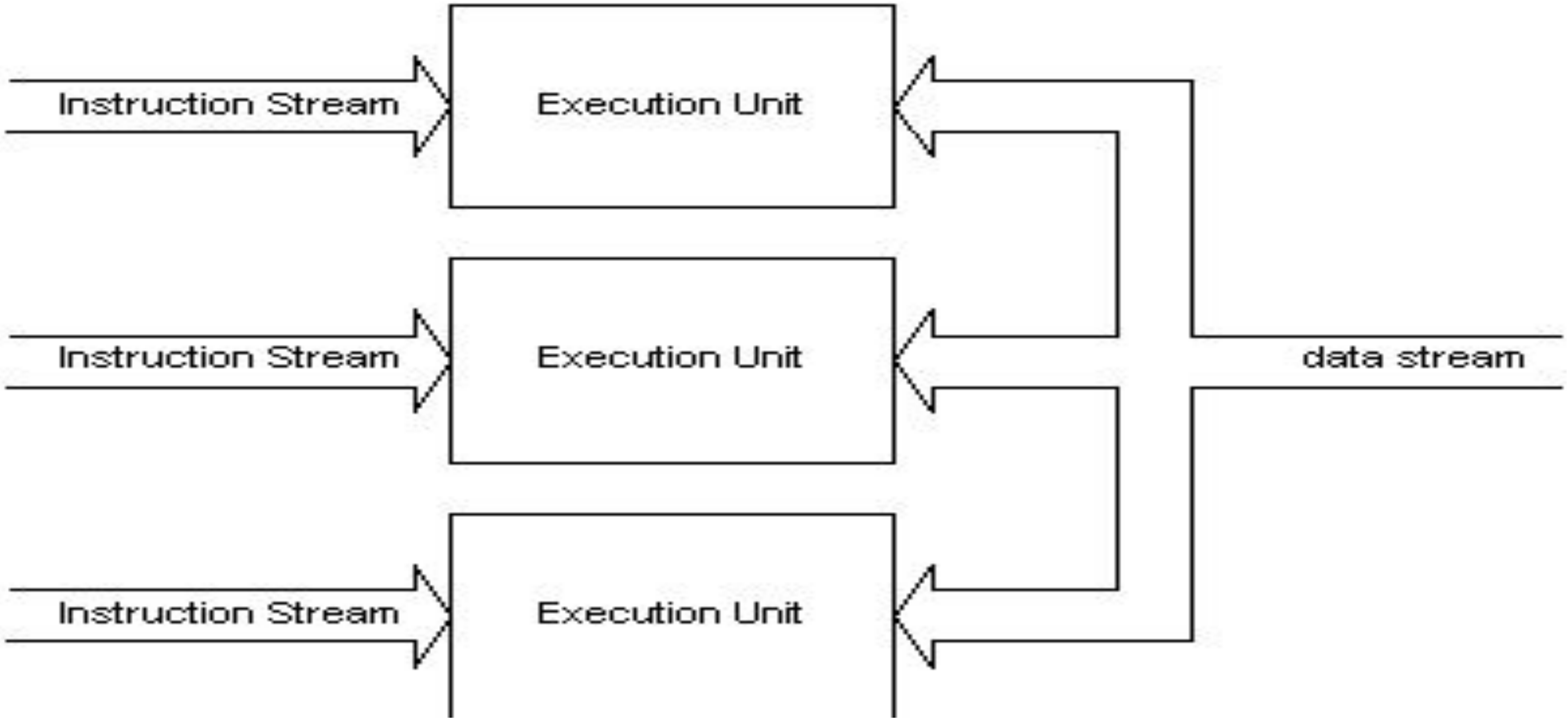
As $\text{Cycle Time}_{\text{non-pipeline}}$ = $\text{Cycle Time}_{\text{pipeline}}$,

Speed Up (S) = $\text{CPI}_{\text{non-pipeline}}$ / (1 + Number of stalls per instruction)

MISD

- Each processing unit operates on the data independently through independent instruction streams as shown in following figure a single data stream is forwarded to different processing unit which are connected to different control unit and execute instruction given to it by control unit to which it is attached.
- Thus, in these computers same data flow through a linear array of processors executing different instruction streams.
- This architecture is also known as systolic arrays for pipelined execution of specific instructions.

MISD


Instruction Stream
Execution Unit
Instruction Stream
Execution Unit
data stream
Instruction Stream
Execution Unit

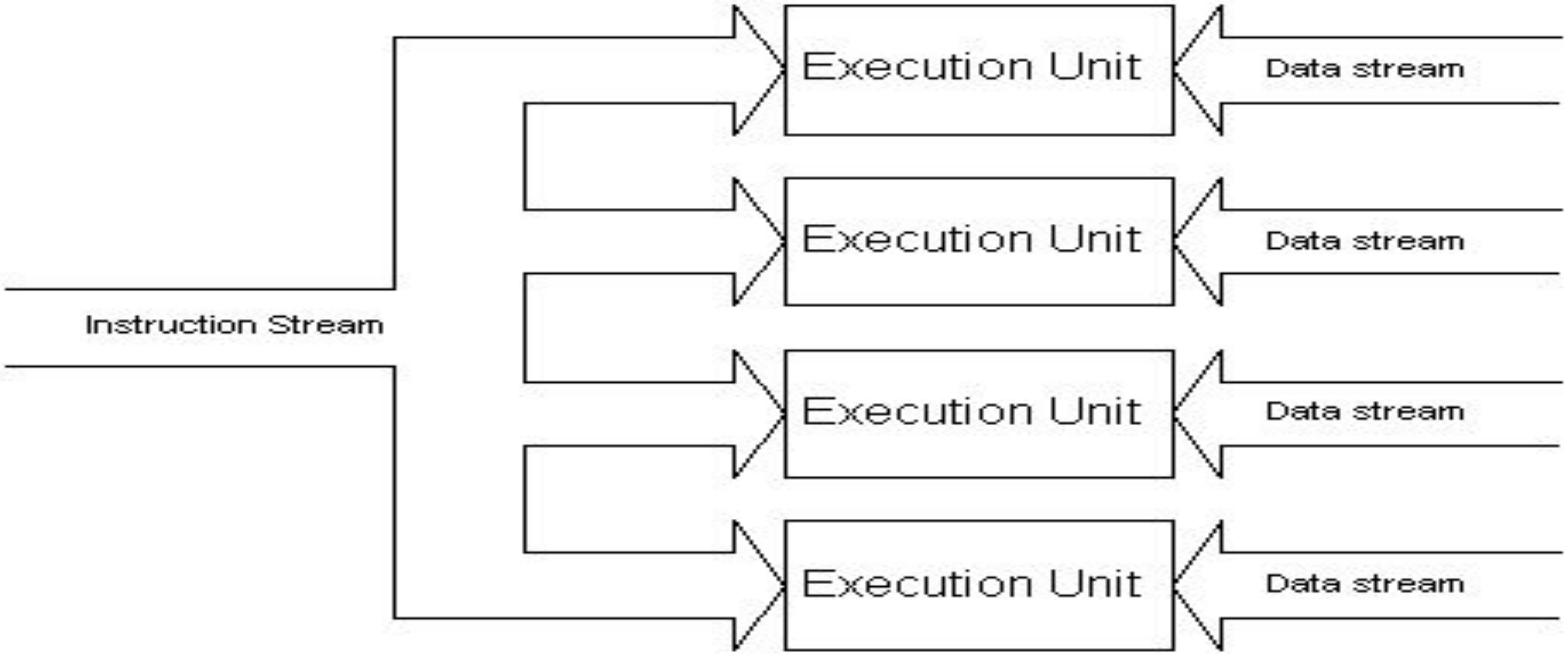
MISD bottleneck

- Low level of parallelism
- High synchronizations
- High bandwidth required
- CISC bottleneck
- High complexity

SIMD

- This is a type of parallel computer.
- Single instruction: All processing units execute the same instruction issued by the control unit at any given clock cycle where there are multiple processors executing instruction given by one control unit.
- Multiple data: Each processing unit can operate on a different data element, the processor are connected to shared memory or interconnection network providing multiple data to processing unit.
- 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. Thus, single instruction is executed by different processing unit on different set of data.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing and vector computation.
- Synchronous (lockstep) and deterministic execution

SIMD


Instruction Stream
Execution Unit
Data stream
Execution Unit
Data stream
Execution Unit
Data stream
Execution Unit
Data stream

SIMD

- A wide variety of applications can be solved by parallel algorithms with SIMD
  - only problems that can be divided into sub problems, all of those can be solved simultaneously by the same set of instructions
  - This algorithms are typical easy to implement

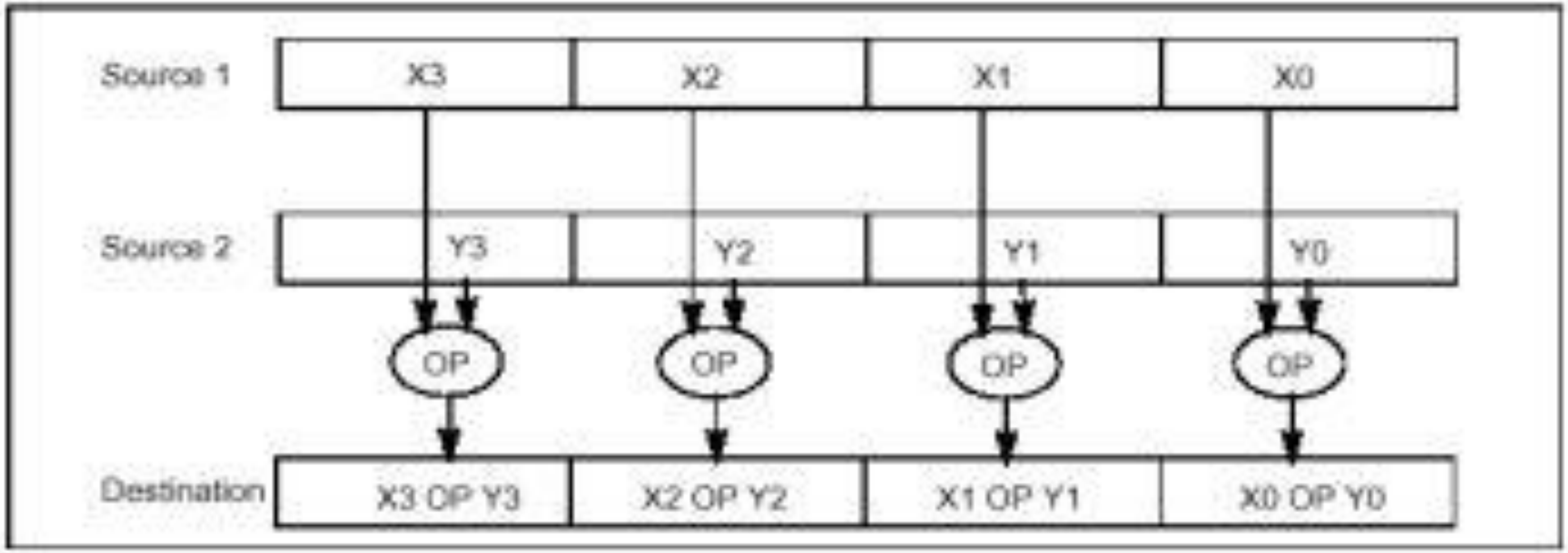
SIMD

- Example of
  - Ordinarily desktop and business applications
    - Word processor, database , OS and many more
  - Multimedia applications
    - 2D and 3D image processing, Game and etc
  - Scientific applications
    - CAD, Simulations

Example of CPU with SIMD ext

- Intel P4 & AMD Althon, x86 CPU
  - 8 x 128 bits SIMD registers
- G5 Vector CPU with SIMD extension
  - 32 x 128 bits registers
- Playstation II
  - 2 vector units with SIMD extension

SIMD operations



SIMD

- SIMD instructions supports
  - Load and store
  - Integer
  - Floating point
  - Logical and Arithmetic instructions
  - Additional instruction (optional)
    - Cache instructions to support different locality for different type of application characteristic

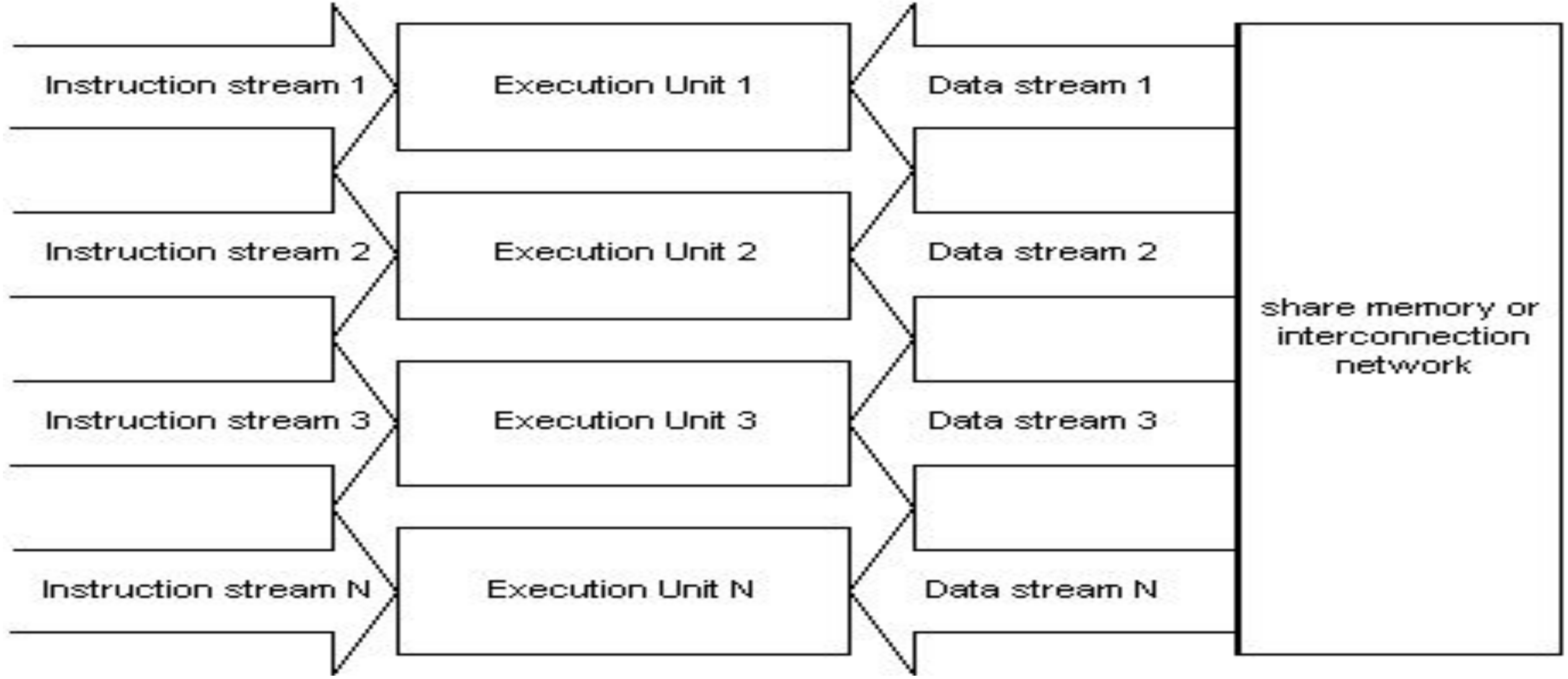
MIMD

- Multiple Instructions: Every Processor may be executing a different instruction stream.
- Multiple Data: every processor may be working with a different data stream, multiple data stream is provided by shared memory.
- Can be categorized as loosely coupled or tightly coupled depending on sharing of data and control.
- Execution can be synchronous or asynchronous, deterministic or non- deterministic.
- Examples: most current supercomputers, networked parallel computer " grids" and multiprocessor SMP computers - including some types of PCs.

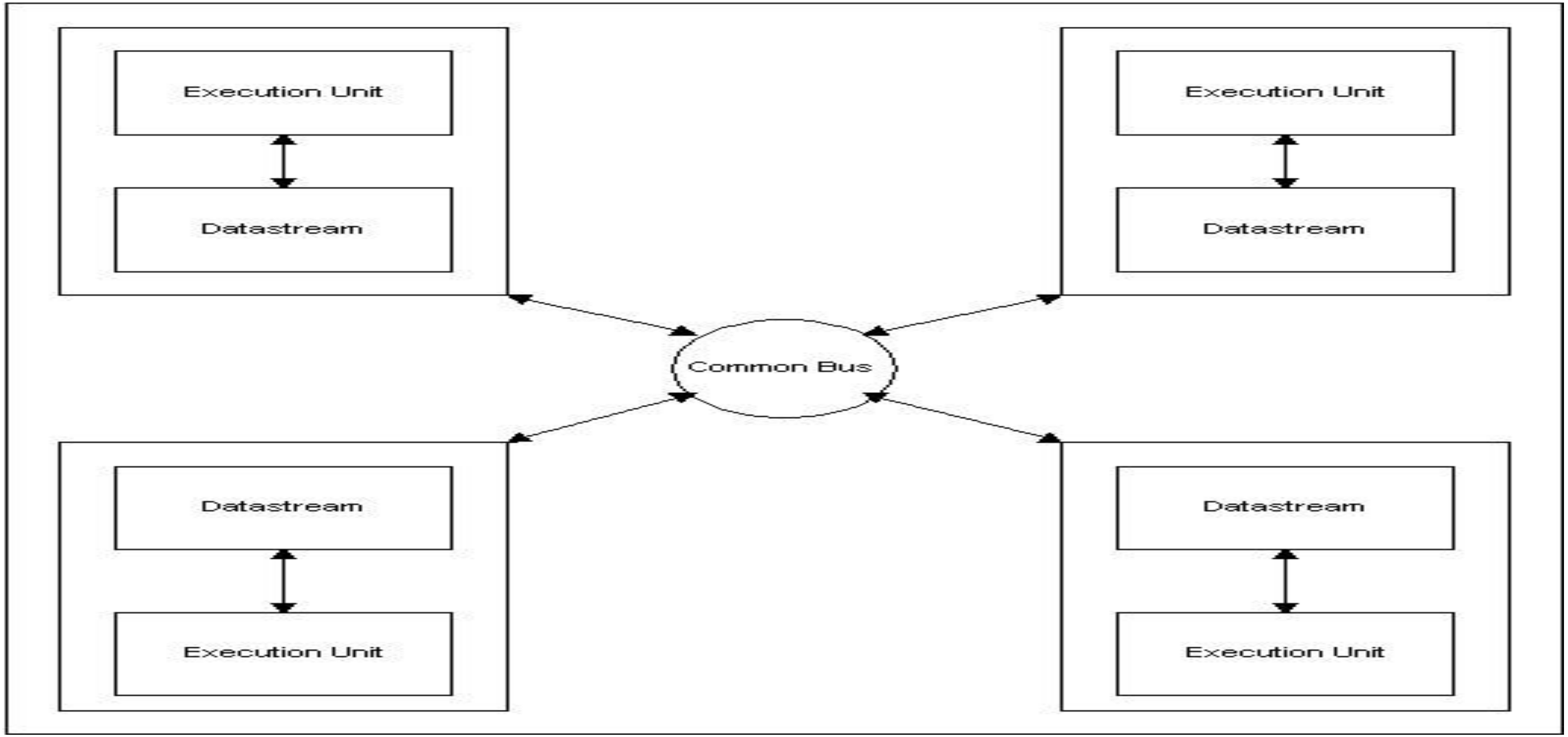
MIMD

- Requires
  - Synchronization
  - Inter-process communications
  - Parallel algorithms
  - Those algorithms are difficult to design, analyze and implement

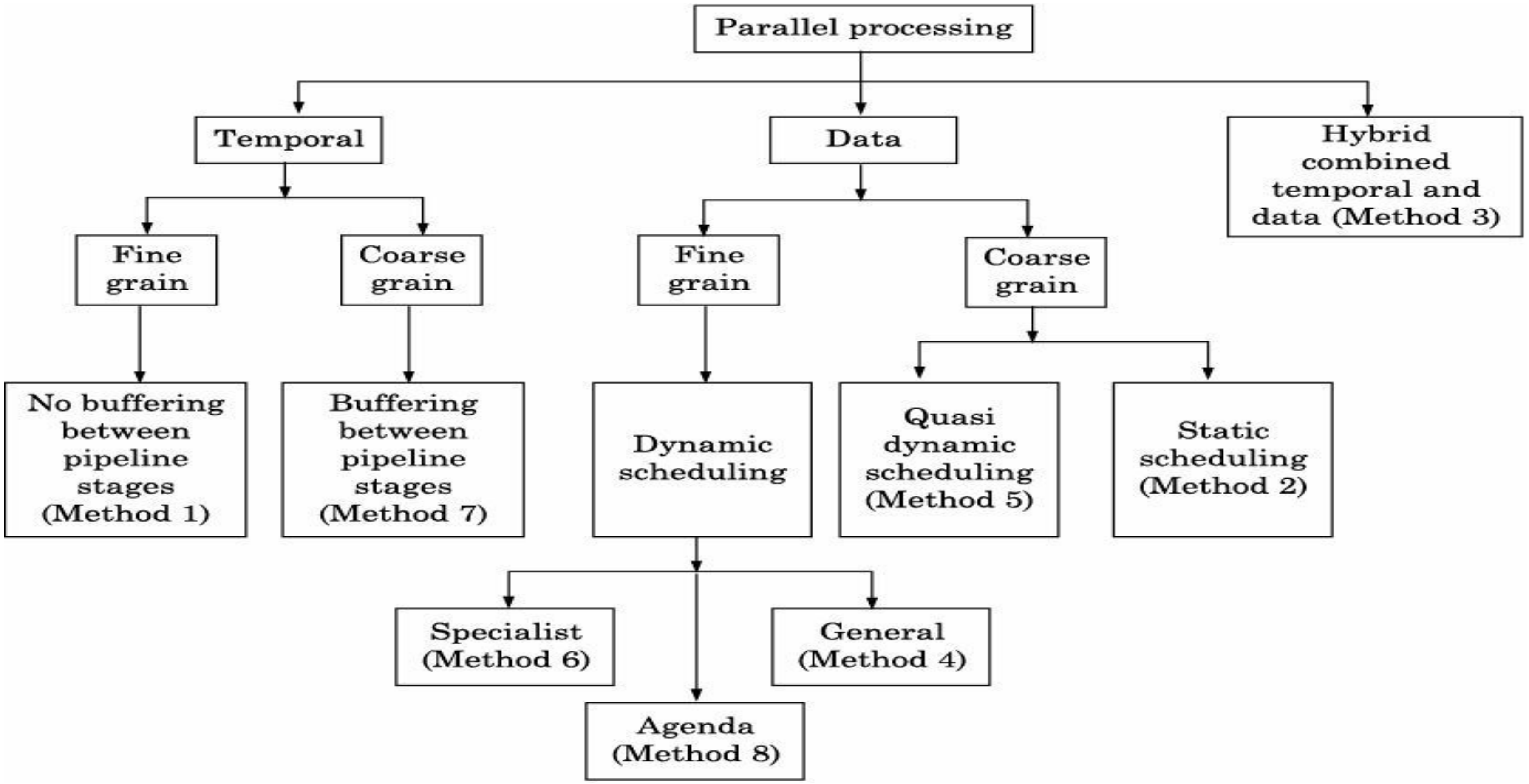
MIMD


Instruction stream 1
Execution Unit 1
Data stream 1
Instruction stream 2
Execution Unit 2
Data stream 2
share memory or interconnection network
Instruction stream 3
Execution Unit 3
Data stream 3
Instruction stream N
Execution Unit N
Data stream N

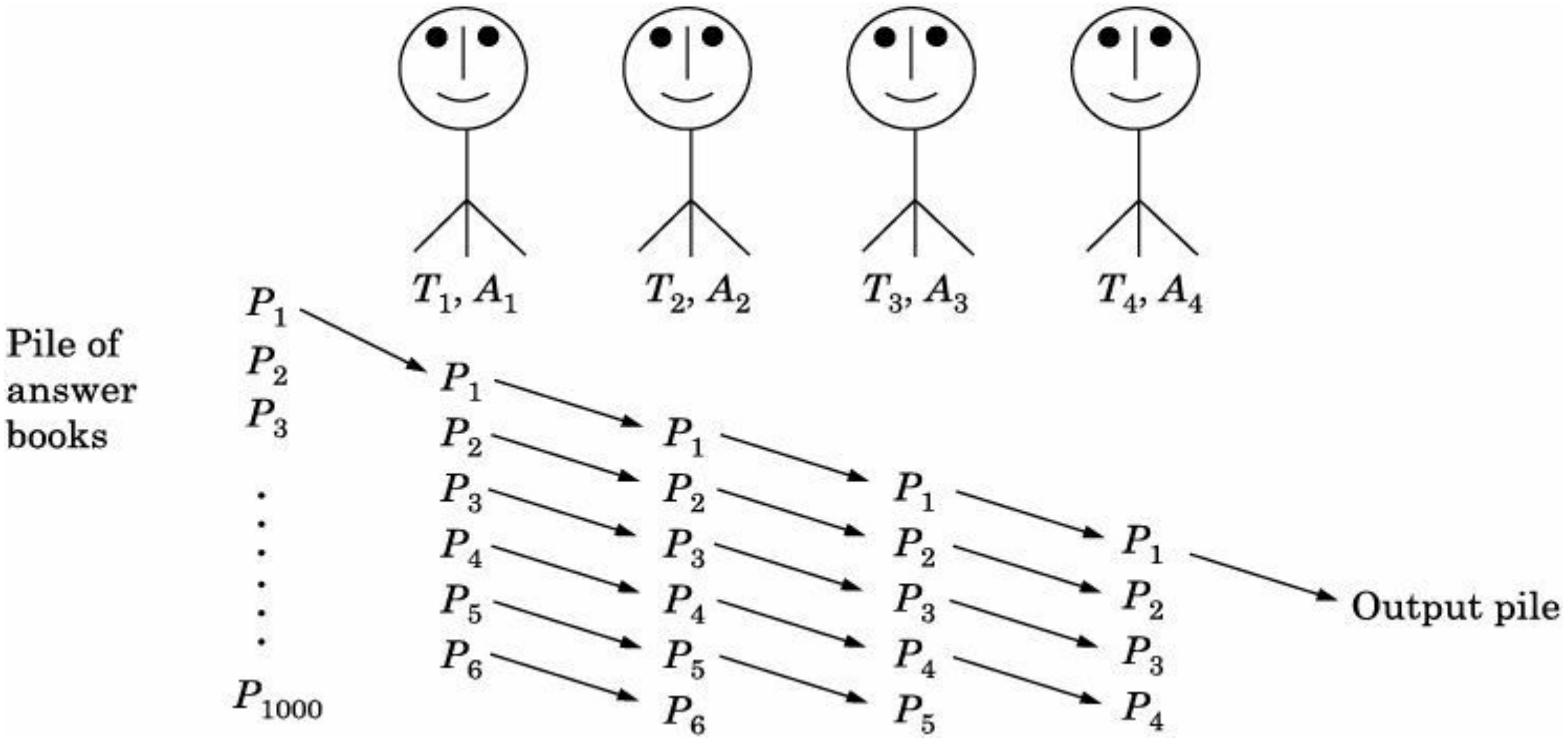
MIMD


Execution Unit
Datastream
Execution Unit
Datastream
Common Bus
Datastream
Execution Unit
Datastream
Execution Unit

Kinds of parallelism


Parallel processing
Temporal
Data
Hybrid combined temporal and data (Method 3)
Fine grain
Coarse grain
Fine grain
Coarse grain
No buffering between pipeline stages (Method 1)
Buffering between pipeline stages (Method 7)
Dynamic scheduling
Quasi dynamic scheduling (Method 5)
Static scheduling (Method 2)
Specialist (Method 6)
General (Method 4)
Agenda (Method 8)

Temporal Parallelism



Temporal Parallelism

- This method of parallel processing is appropriate if:
  - The jobs to be carried out are identical.
  - A job can be divided into many independent tasks (i.e., each task can be done independent of other tasks) and each can be performed by a different teacher.
  - The time taken for each task is same.
  - The time taken to send a job from one teacher to the next is negligible compared to the time needed to do a task.
  - The number of tasks is much smaller as compared to the total number of jobs to be done.

Temporal Parallelism

- Let the number of jobs = n
- Let the time to do a job = p
- Let each job be divisible into k tasks and let each task be done by a different individual.
- Let the time for doing each task = p/k.
- Time to complete n jobs with no pipeline processing = np.
- Time to complete n jobs with a pipeline organization of k individual

$$= p + (n - 1)\frac{p}{k} = p\frac{(k + n - 1)}{k}$$

- Speedup due to pipeline processing

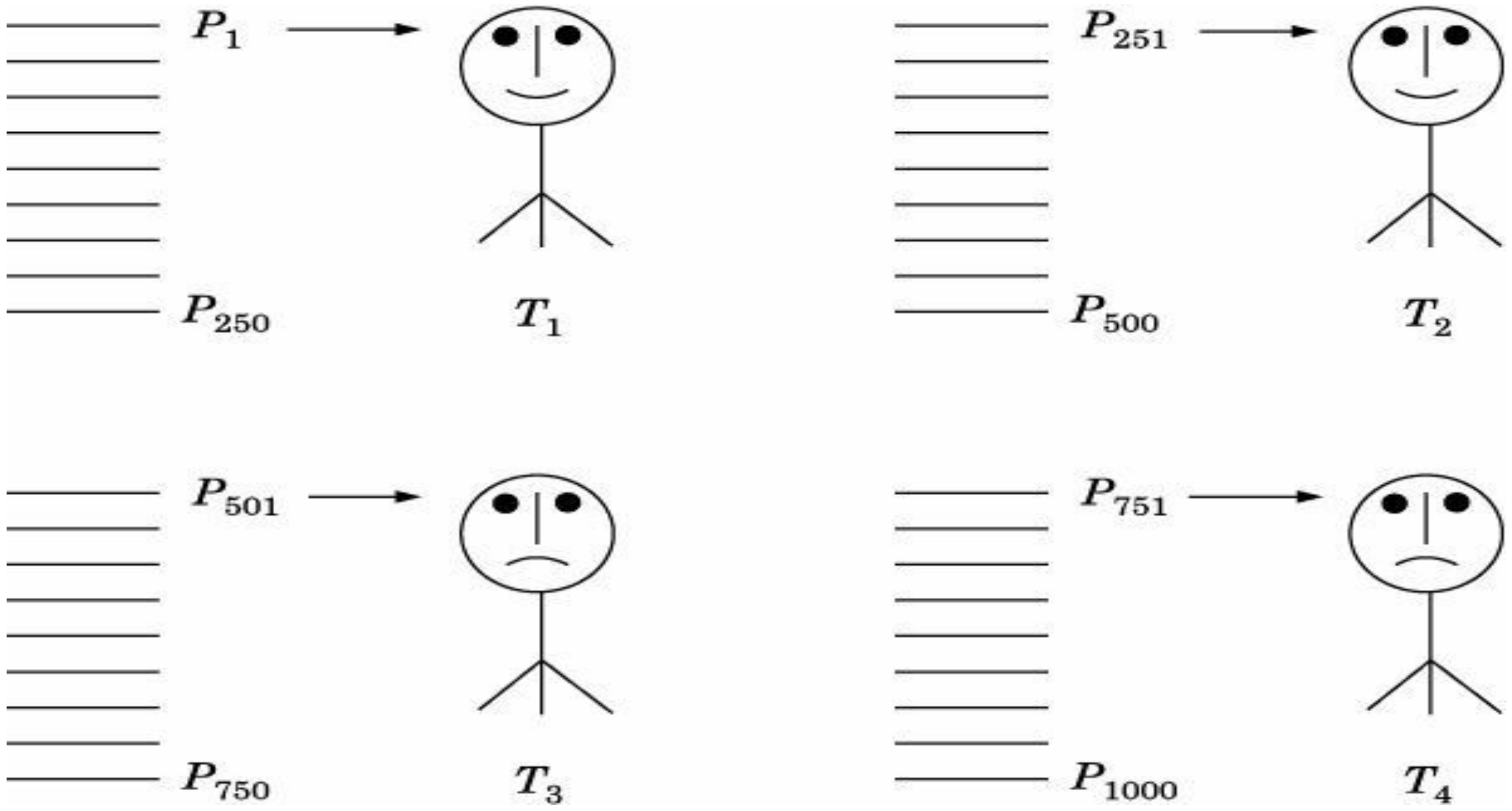
$$= \frac{np}{p(k + n - 1)/k} = \frac{k}{1 + \frac{k - 1}{n}}$$

Problems with Temporal Parallelism

- Synchronization
- Bubbles in pipeline
- Fault tolerance
- Inter-task communication
- Scalability

- In spite of these problems, this method is a very effective technique of using parallelism as it is easy to perceive the possibility of using temporal parallelism in many jobs.
- Pipelining is used extensively in processor design. It was the main technique used by vector supercomputers such as CRAY to attain their high speed.

Data Parallelism



Data Parallelism

- Let the time to distribute the jobs to k individuals be **kq**. Observe that this time is proportional to the number of individuals.
- The time to complete n jobs by a single individuals = **np**
- The time to complete n jobs by k individuals = **kq + np/k**
- Speedup due to parallel processing =

$$\frac{np}{kq + \frac{np}{k}} = \frac{knp}{k^2 q + np} = \frac{k}{1 + (k^2 q / np)}$$

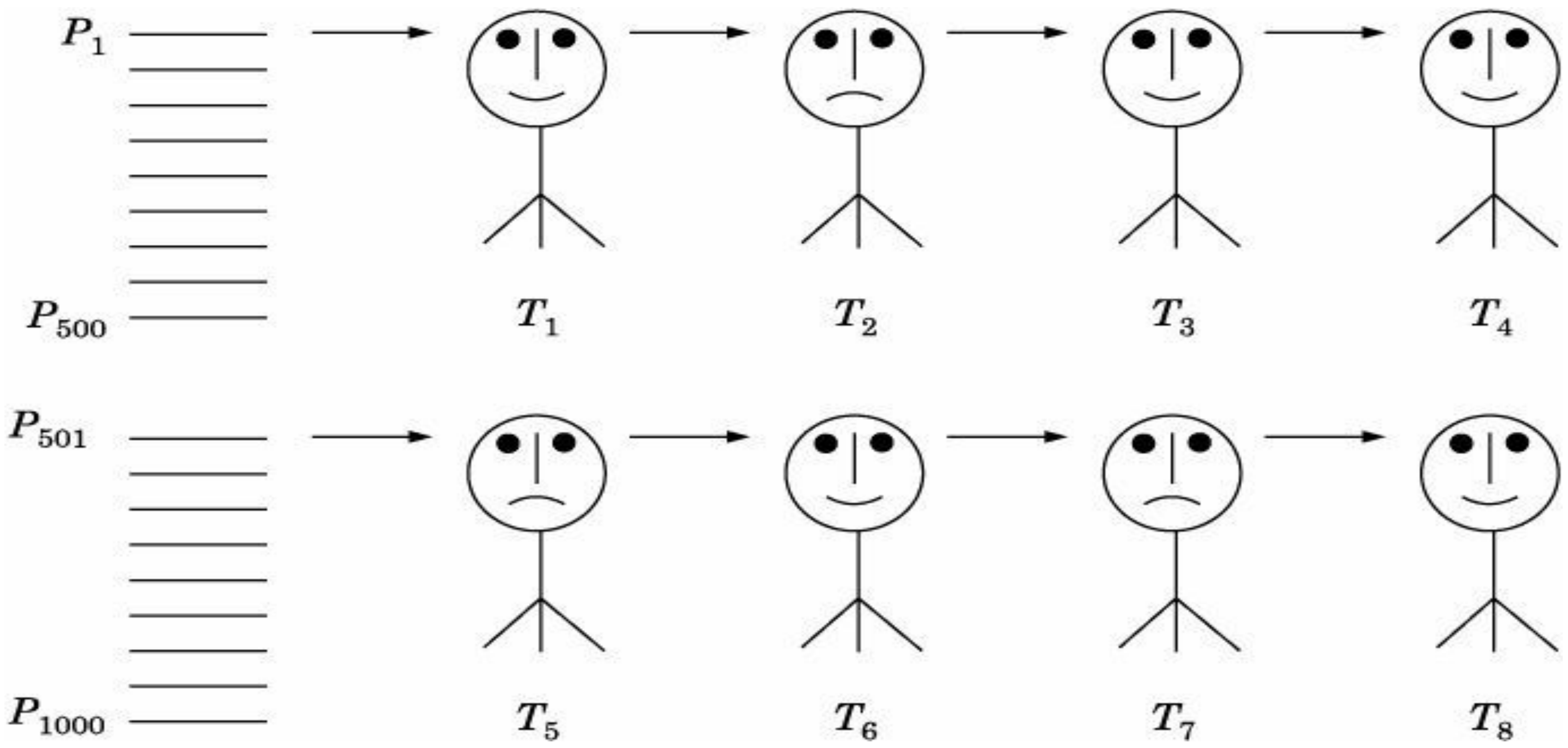
Data Parallelism

- The main **advantages** of this method are:
    - There is **no synchronization** required between teachers. Each teacher can correct papers independently at his own pace.
    - The problem of **bubbles is absent**. If a question is unanswered in a paper it only reduces the time to correct that paper.
    - This method is **more fault tolerant**. One of the teachers can take a coffee break without affecting the work of other teachers.
    - There is **no communication required** between teachers as each teacher works independently. Thus, there is no inter-task communication delay.

Data Parallelism

- The main **disadvantages** of this method are:
    - The assignment of jobs to each teacher is pre-decided. This is called a static assignment. Thus, if an individual is slow then the completion time of the total job will be slowed down.
    - We must be able to divide the set of jobs into subsets of mutually independent jobs. Each subset should take the same time to complete.
    - Each individual must be capable of correcting answers to all questions. This is to be contrasted with pipelining in which each individual specialized in correcting the answer to only one question.
    - The time taken to divide a set of jobs into equal subsets of jobs should be small. Further, the number of subsets should be small as compared to the number of jobs.

Mixed Parallelism: Parallel Pipeline Processing



Mixed Parallelism: Parallel Pipeline Processing

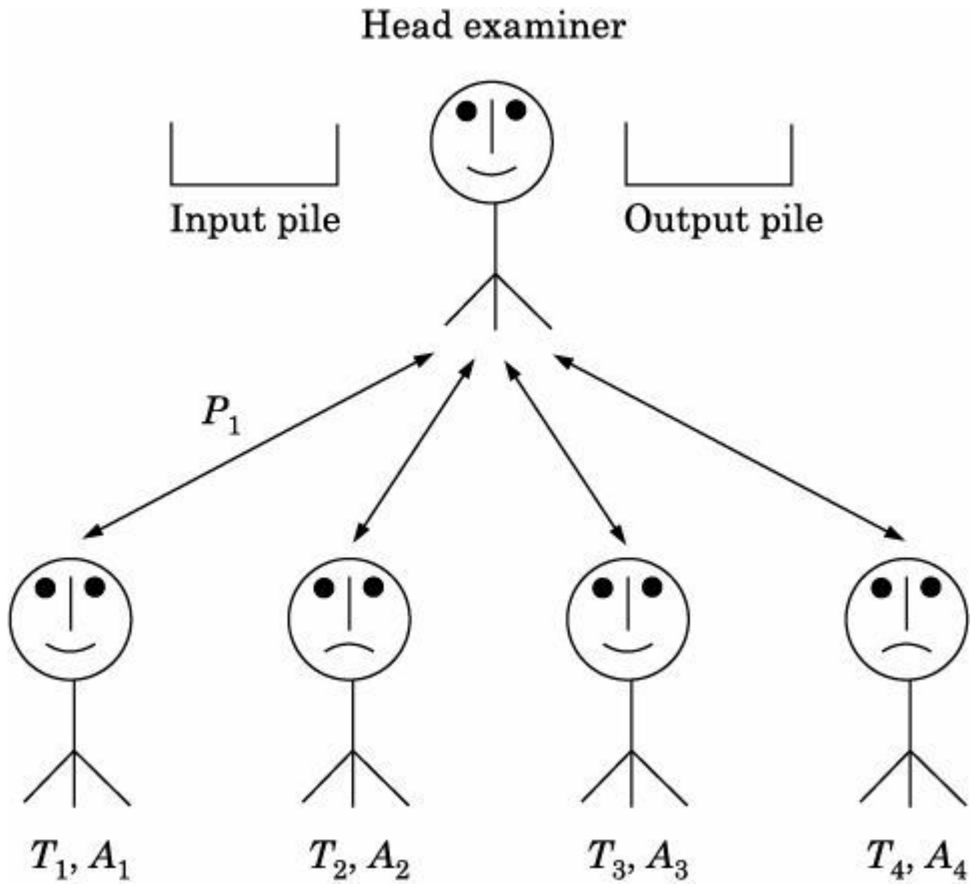
- Even though this method reduces the time to complete the set of jobs, it also has the disadvantages of both temporal parallelism and to some extent that of data parallelism.
- The method is effective only if the number of jobs given to each pipeline is much larger than the number of stages in the pipeline.
- Multiple pipeline processing was used in supercomputers such as Cray and NEC-SX as this method is very efficient for numerical computing in which a number of long vectors and large matrices are used as data and could be processed simultaneously.

Other Parallelisms

- **Data Parallelism with Dynamic Assignment**
    - Here a head examiner gives one answer paper to each teacher and keeps the rest with him. All teachers simultaneously correct the paper given to them. A teacher who completes correction goes to the head examiner for another paper which is given to him for correction. If a second teacher completes correction at the same time, then he queues up in front of the head examiner and waits for his turn to get an answer paper. The procedure is repeated till all the answer papers are corrected.
- **Data Parallelism with Quasi-dynamic Scheduling**
    - giving each teacher unequal sets of answer papers to correct.

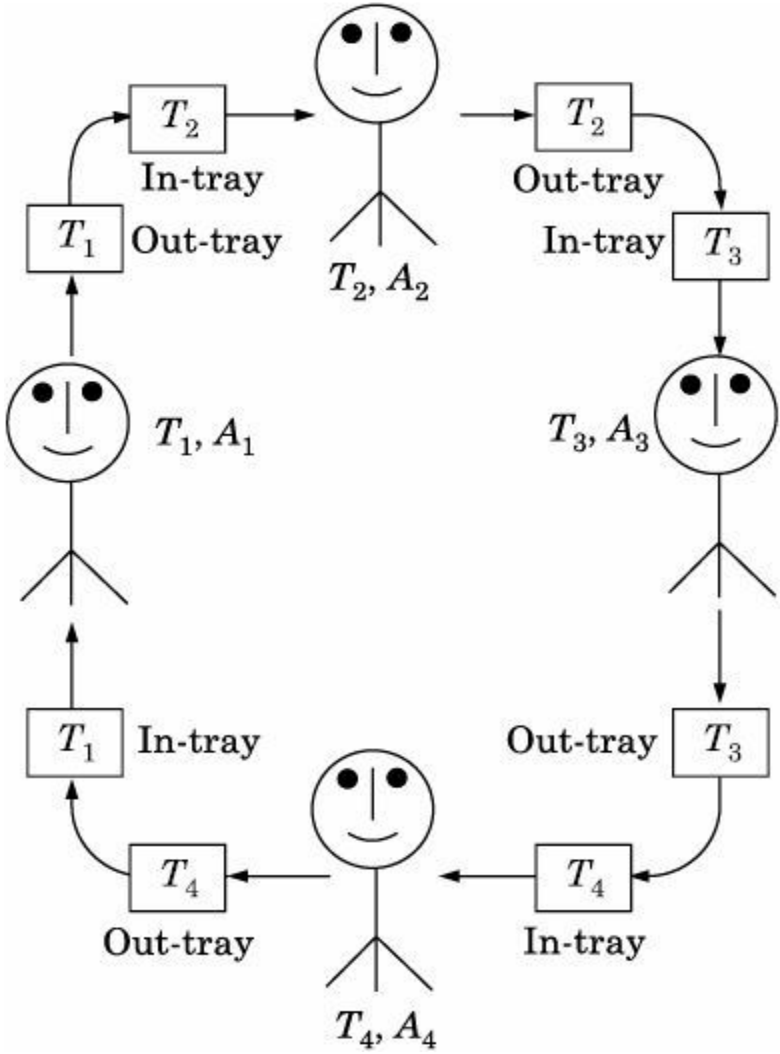
Other Parallelisms

- Specialist Data Parallelism



Other Parallelisms

- Coarse Grained Specialist Temporal Parallel Processing



Detecting Parallelism: Bernstein's Conditions

- In 1966, Bernstein revealed a set of conditions based on which two processes can execute in parallel.
- We define the input set **I_i**, of a process P_i, as the set of all input variables needed to execute the process (aka **read set**).
- Similarly, the output set **O_i** consists of all output variables generated after execution of the process P_i (aka **write set**).
- Input variables are essentially operands which can be fetched from memory or registers, and output variables are the results to be stored in working registers or memory locations.
- Now, consider two processes P_1 and P_2 with their input sets I_1 and I_2 and output sets O_1 and O_2, respectively. These two processes can execute in parallel and are denoted $P_1 \| P_2$ if they are independent and hence create deterministic results.

Detecting Parallelism: Bernstein’s Conditions

- Formally, these conditions are stated as follows:

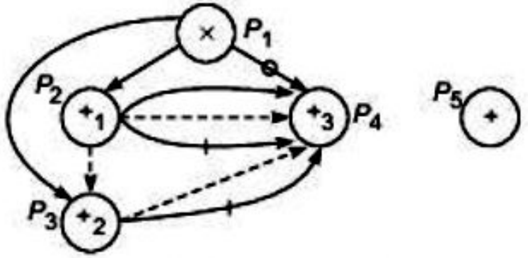
$$\left.\begin{aligned} I_1 \cap O_2 &= \phi \\ I_2 \cap O_1 &= \phi \\ O_1 \cap O_2 &= \phi \end{aligned}\right\}$$

- These three conditions are known as Bernstein’s conditions.
- In terms of data dependencies, Bernstein's conditions simply imply that two processes can execute in parallel if they are **flow-independent, anti-independent, and output-independent**.

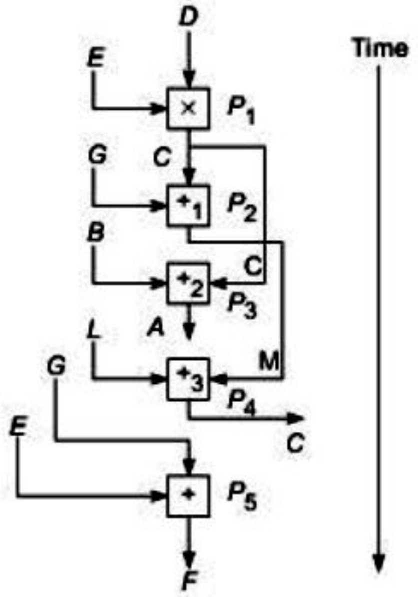
Detecting Parallelism: Example

- Consider the simple casein which each process is a single HLL statement. We want to detect the parallelism embedded in the following five statements labeled PI, P2, P3, P4, and P5 in program order

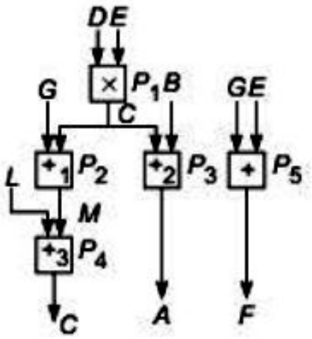
$$\left.\begin{aligned}
P_1 &: C = D \times E \\
P_2 &: M = G + C \\
P_3 &: A = B + C \\
P_4 &: C = L + M \\
P_5 &: F = G + E
\end{aligned}\right\}$$



(a) A dependence graph showing both data dependence (solid arrows) and resource dependence (dashed arrows)



(b) Sequential execution in five steps, assuming one step per statement (no pipelining)



(c) Parallel execution in three steps, assuming two adders are available per step

Processor Types

- **Vector processor:**
  - Cray X1/T90; NEC SX#; Japan Earth Simulator; Early Cray machines; Japan Life Simulator (hybrid)
- **Scalar processor:**
  - **CISC:** Complex Instruction Set Computer
    - Intel 80x86 (IA32)
  - **RISC:** Reduced Instruction Set Computer
    - Sun SPARC, IBM Power #, SGI MIPS

CISC Processor

- CISC
  - Complex instructions; Large number of instructions; Can complete more complicated functions at instruction level
  - Instruction actually invokes microcode. Microcodes are small programs in processor memory
  - Slower; Many instructions access memory; varying instruction length; allow no pipelining;

RISC Processor

- No microcode
- Simple instructions; Fewer instructions; Fast
- Only load and store instructions access memory
- Common instruction word length
- Allows pipelining

Almost all present-day high performance computers use RISC processors

Locality of References

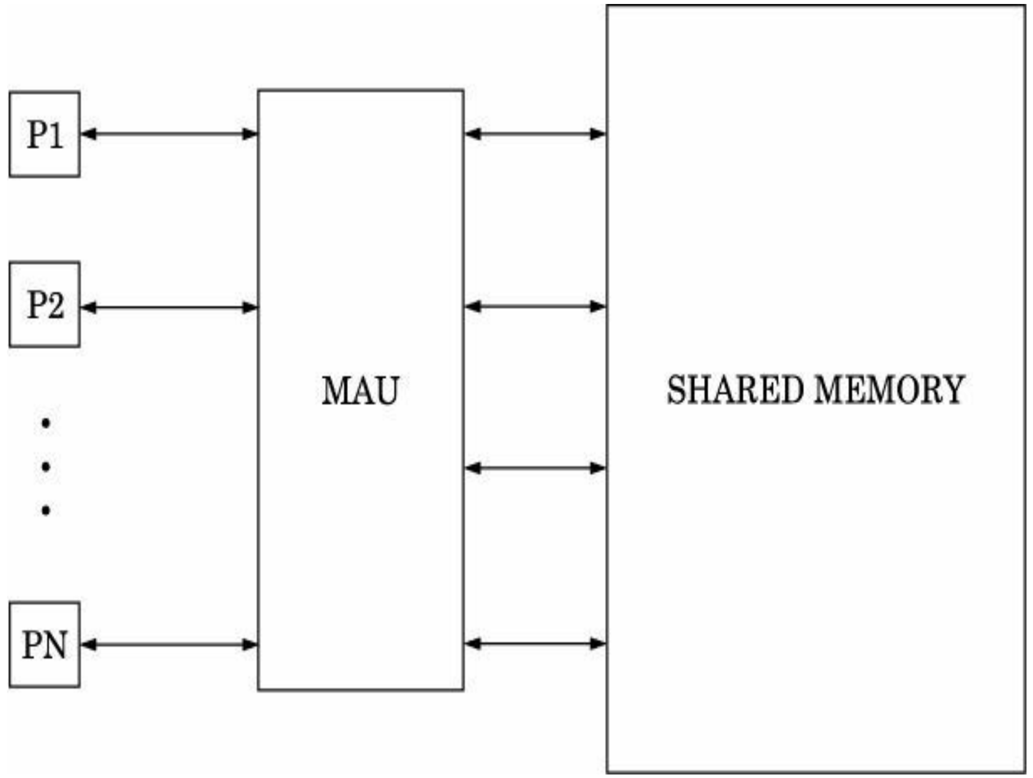
- **Spatial/Temporal locality**
  - If processor executes an instruction at time **t**, it is likely to execute an adjacent/next instruction at **(t+delta_t)**;
  - If processor accesses a memory location/data item **x** at time **t**, it is likely to access an adjacent memory location/data item **(x+delta_x)** at **(t+delta_t)**;

Pipelining, Caching and many other techniques all based on the locality of references

The Parallel Random Access Machine (PRAM)

The PRAM is one of the popular models for designing parallel algorithms, consists of the following:

- A set of N(P1, P2, ..., PN) identical processors. In principle, N is unbounded.
- A memory with M locations which is shared by all the N processors. Again, in principle, M is unbounded.
- An MAU which allows the processors to access the shared memory.



The Parallel Random Access Machine (PRAM)

- The PRAM model can be subdivided into 4 categories based on the way simultaneous memory accesses are handled.
    - Exclusive Read Exclusive Write (**EREW**) PRAM
        - every access to a memory location (read or write) has to be exclusive.
    - Concurrent Read Exclusive Write (**CREW**) PRAM
        - Only write operations to a memory location are exclusive.
    - Exclusive Read Concurrent Write (**ERCW**) PRAM
        - Multiple processors can concurrently write into the same memory location.
    - Concurrent Read Concurrent Write (**CRCW**) PRAM
        - Allows both multiple read and multiple write operations to a memory location.

The Parallel Random Access Machine (PRAM)

- There are many methods to implement the PRAM model, but the most prominent ones are:
    - Shared memory model
    - Message passing model
    - Data parallel model