Introduction to Parallel Architectures

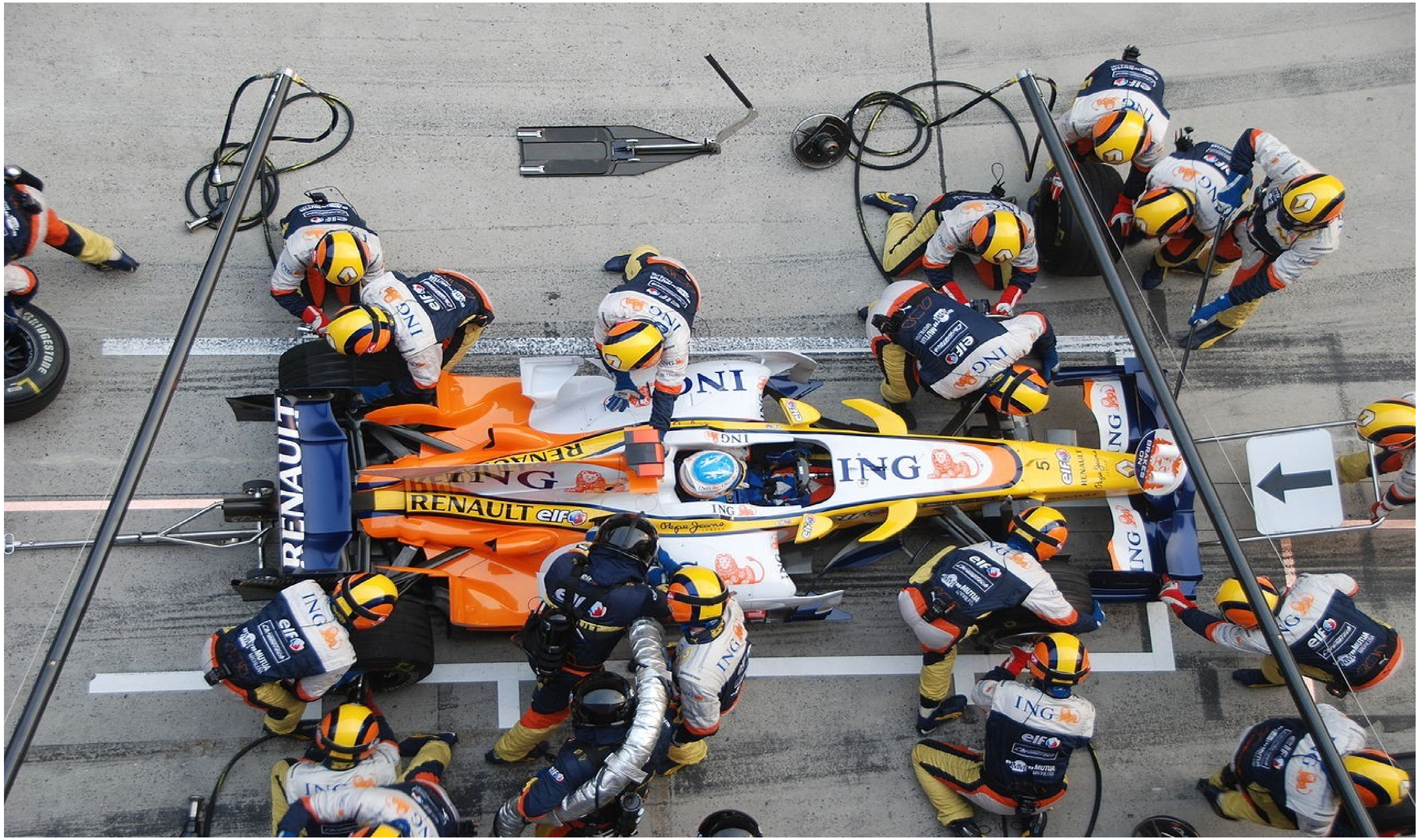
5 Hours

Topics covered:

1. Introduction to Parallel Processing
2. Parallel Computer Structures
3. Architectural Classification Schemes
4. GPU as Parallel Computers
5. Architecture of a Modern GPU
6. Need for Parallelism
7. Parallel Programming Languages and Models

❖ **Please click on the links in following slides to display the relevant figures and web pages**

Parallel Programming Analogy



Realistic Expectations

- Ex. – Your program takes 20 days to run
- 95% can be parallelized
- 5% cannot (serial)
- What is the fastest this code can run?
  - As many CPU's as you want!

$$\boxed{1\text{ day!}} \longrightarrow \boxed{(5/100)*20}$$

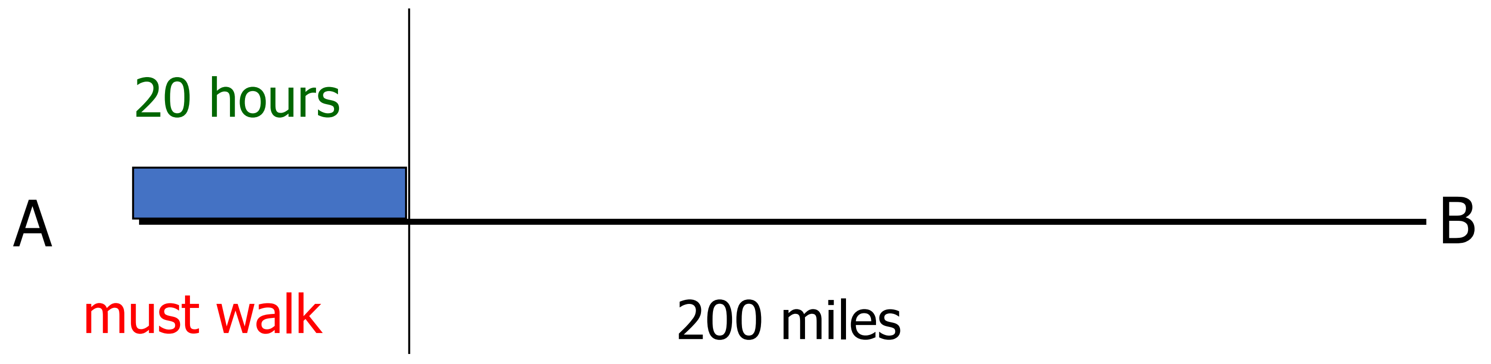
Amdahl's Law

The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used

Speedup (S)

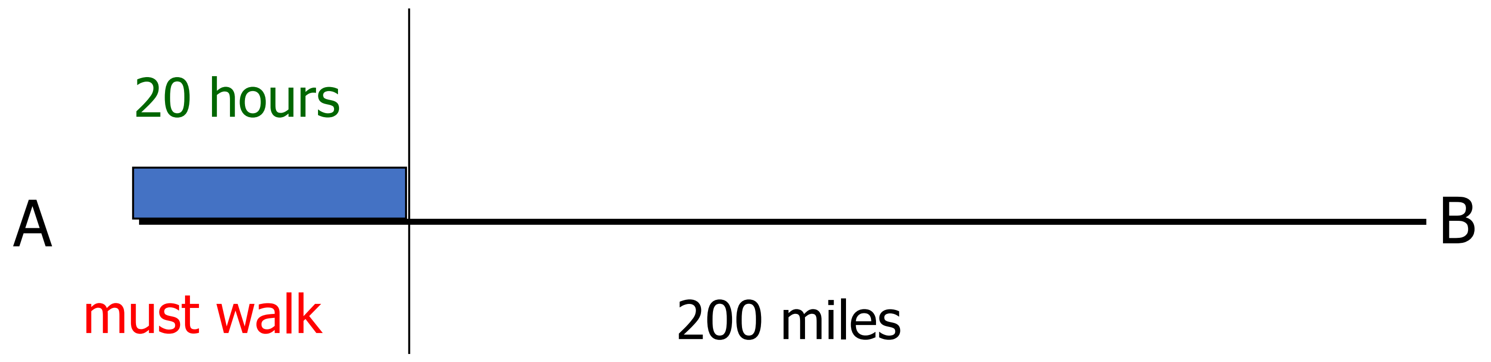
- S = Speed(new) / Speed(old)
- S = Work(new) / time(old)
- S = time(old) / time(new)
- S = time(before improvement) /
  time(after improvement)

Example



Walk 4 miles /hour
Bike 10 miles / hour
Car-1 50 miles / hour
Car-2 120 miles / hour
Car-3 600 miles /hour

Example



Walk 4 miles /hour	✉	50 + **20** = 70 hours	S = 1
Bike 10 miles / hour	✉	20 + **20** = 40 hours	S = 1.8
Car-1 50 miles / hour	✉	4 + **20** = 24 hours	S = 2.9
Car-2 120 miles / hour	✉	1.67 + **20** = 21.67 hours	S = 3.2
Car-3 600 miles /hour	✉	0.33 + **20** = 20.33 hours	S = 3.4

Introduction to Parallel Processing

The trends towards parallel processing can be seen from:

1. **Application point of view**
2. **Operating system point of view**

1. Application point of view

The computers are experiencing a trend of four ascending levels of sophistication:

- **Data Processing** 🖂 Data objects include numbers, characters, images, audio, video, multidimensional measures etc.
- **Information Processing** 🖂 Information item is a collection of data objects that are related by some syntactic structure or relation
- **Knowledge Processing** 🖂 Knowledge consists of information items plus some semantic meanings
- **Intelligence Processing** 🖂 Intelligence is derived from a collection of knowledge items.

Data-space

2. Operating system point of view

The computer systems have improved chronologically in four phases:

- Batch Processing
- Multiprogramming
- Time Sharing
- Multiprocessing(refe text book)

Parallel processing of information increases sharply by exploiting concurrent events

Concurrency ✉ implies parallelism, simultaneity, and pipelining

Parallelism✉ Parallel events may occur in multiple resources during the *same time interval*

Simultaneity✉ Simultaneous events may occur at the *same time instant*

Pipelining✉ Pipelined events may occur in *overlapped time spans*

Degree (level) of parallel processing

Job or Program level ✉The highest level of parallel processing is conducted among multiple jobs or programs through multiprogramming, time sharing and multiprocessing

Task or Procedure level ✉The next highest level of parallel processing is conducted among procedures or tasks (program segments)

Inter-instruction level ✉ The third level exploit concurrency among multiple instructions

Intra-instruction level ✉ The final last level exploit concurrency within each instruction

The highest level of parallelism is often conducted algorithmically and the lowest level is implemented by hardware means

Parallel Computer Structures

Parallel computers are those systems that emphasize parallel processing. Parallel computers are divided into three architectural configurations:

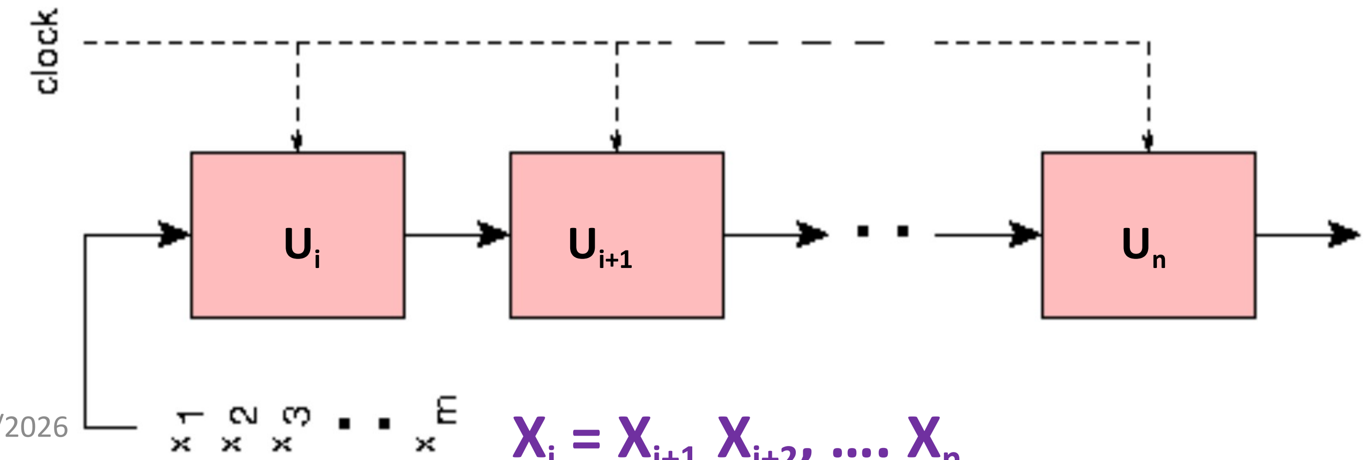
1. Pipeline computers ✉ Perform overlapped computations to exploit *temporal parallelism*

2. Array processors ✉ Use multiple synchronized arithmetic logic units to achieve *spatial parallelism*

3. Multiprocessor systems ✉ Use a set of interactive processors with shared resources(memory, etc.) to achieve *asynchronous parallelism*

Temporal parallelism

- *Temporal parallelism* or *pipelining* refers to the execution of a task as a ***'cascade' of sub-tasks***
- There exists one functional unit to carry out each sub-task
- All these successive units can work at the same time, in an overlapped fashion
- As data are processed by a given unit **U_i** , they are sent to the next unit **U_{i+1}** and the unit **U_i** restarts its processing on new data, analogously to the flow of work in a car production line. Each functional unit can be seen as a "specialized" processor in the sense that it always execute the same sub-task

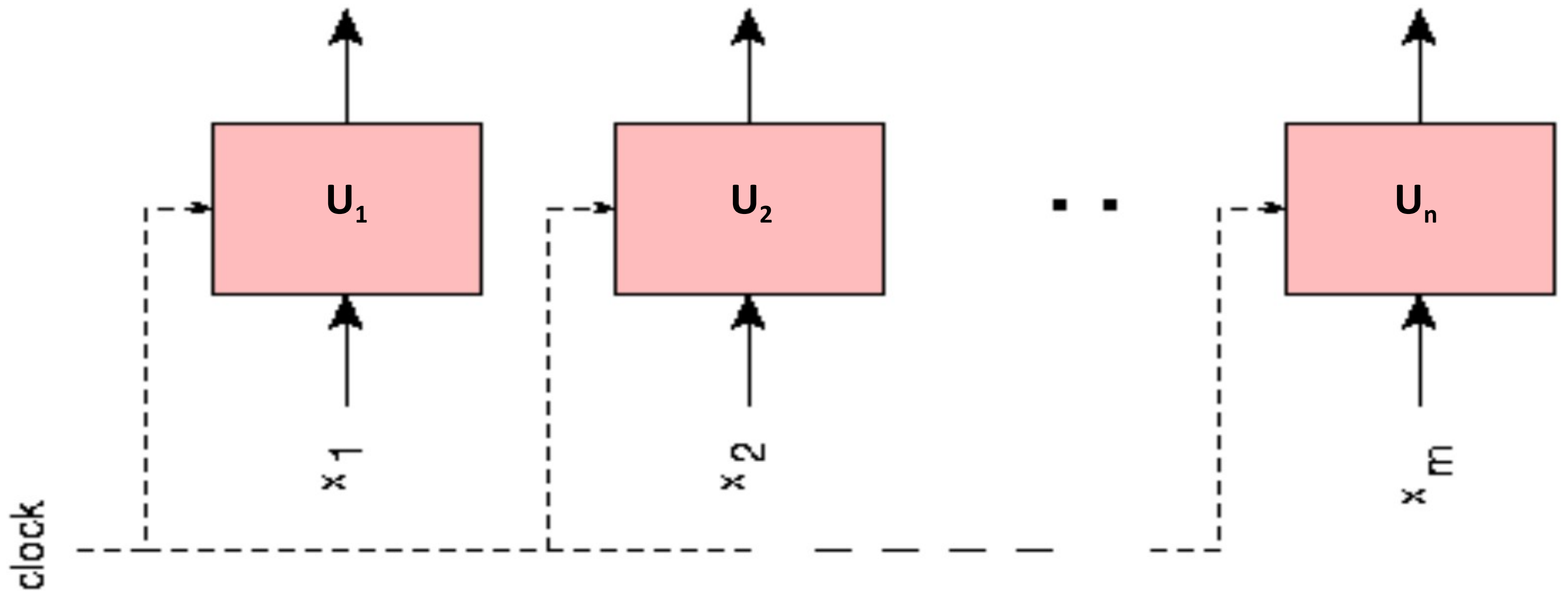


$X_i = X_{i+1}, X_{i+2}, \ldots. X_n$

Spatial parallelism

- *Spatial parallelism* refers to the ***simultaneous execution*** of tasks by ***several processing units***

- At a given instant, these units can be executing the *same task* (or instruction) or *different tasks*. The former case is called ***SIMD*** ***(Single Instruction stream, Multiple Data stream)***, whereas the latter is called ***MIMD*** ***(Multiple Instruction stream, Multiple Data stream)***



$X_i = X_{i+1}, X_{i+2}, \ldots X_n$

1. *Pipeline computers*

- Normally, the process of executing an instruction in a digital computer involves four major steps:

  1. **Instruction Fetch (IF)**✉ fetching instruction from the main memory

  2. **Instruction Decoding (ID)** ✉ identifying the operation to be performed

  3. **Operand Fetch (OF)** ✉ fetching the operands is needed in the execution

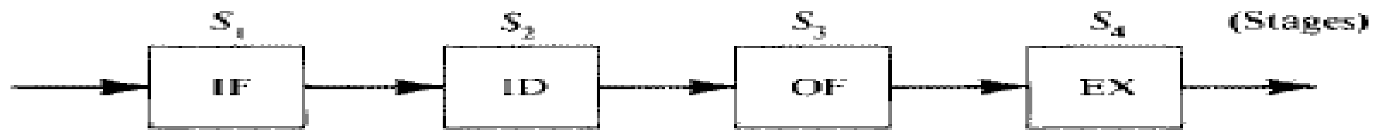
  4. **Execution (EX)** ✉ Execution of the decoded arithmetic-logic operation

- In a nonpipelined computer, *these four steps must be completed before the next instruction can be issued*

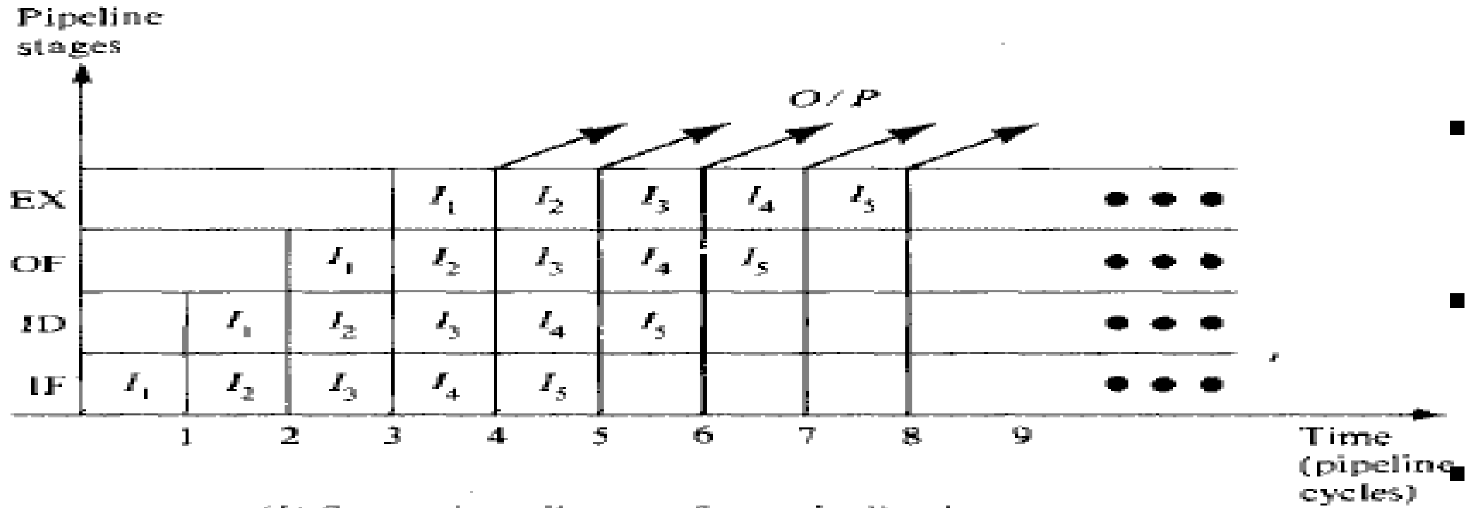
- In a pipelined computer, *successive instructions are executed in an overlapped fashion*. The Four pipeline stages, lF, ID, OF, and EX, are arranged into a linear cascade

- The two *space-time diagrams* in the next slide show the difference between overlapped instruction execution *(pipelining)* and sequentially nonoverlapped execution *(non-pipelining)*

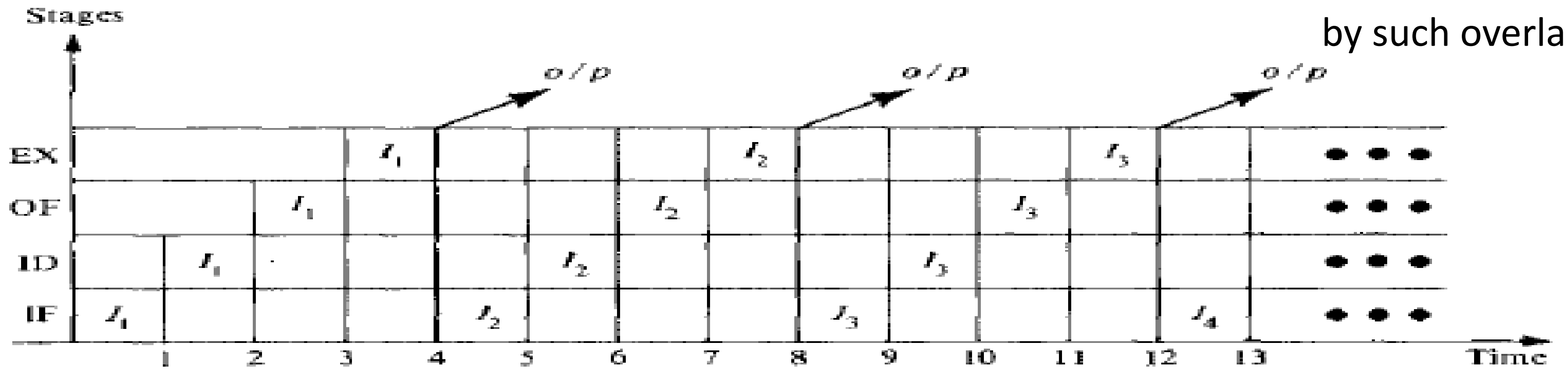
Pipeline computers



(*a*) A pipelined processor



(*b*) Space-time diagram for a pipelined processor



(*c*) Space-time diagram for a nonpipelined processor

- A pipeline cycle can be set equal to the delay of the *slowest stage*
- The flow of data (input operands, intermediate results, and output results) from stage to stage is triggered by a *common clock* of the pipeline
- The operation of all stages is *synchronized* under a common clock control
- *Interface latches* are used between adjacent segments to hold the intermediate results
- The instruction cycle has been effectively *reduced to one-fourth* of the original cycle time by such overlapped execution.

Pipeline computers (limitations)

- Due to the overlapped instruction and arithmetic execution, it is obvious that pipeline machines are *better tuned to perform the same operations repeatedly* through the pipeline.

- Whenever there is a change of operation, say from *add* to *multiply*, the arithmetic pipeline must be *drained and reconfigured*, which will cause extra time delays.

- Therefore, pipeline computers are more attractive for *vector processing*, where component operations may be repeated many times.

Pipeline computer

2. Array computers

- An *array processor* is a synchronous parallel computer with **multiple arithmetic logic units**, called *processing elements*(PE), that can operate in parallel in a lock- step fashion
- By replication of ALUs, we can achieve the *spatial parallelism*
- The PEs are *synchronized* to perform the same function at the same time
- *Scalar* and *control-type instructions* are directly executed in the *control unit* (CU)
- Each PE consists of an *ALU with registers* and a *local memory*

Array computer

- The PEs are inter connected by a *data-routing network*
- *Vector- instructions* are *broadcast* to the PEs for *distributed execution* over different component operands fetched directly from the local memories.
- *Instruction fetch* (from local memories or from the control memory) and *decode* is done by the control unit. The PEs are *passive devices* without instruction decoding capabilities.

3. Multiprocessor systems

- Research and development of multiprocessor systems are aimed at improving *throughput*, *reliability*, *flexibility*, and *availability*
- A basic multiprocessor organization is conceptually depicted in this figure **Multiprocessor systems**
- The system contains *two or more processors* of approximately *comparable capabilities*
- All processors *share access to* common sets of memory modules, I/O channels, and peripheral devices

Multiprocessor systems

- Most importantly, the entire system *must be controlled by a single integrated operating system* providing interactions between processors and their programs at various levels

- Besides the shared memories and I/O devices, each processor has its own *local memory*

- *Interprocessor communications* can be done through the *shared memories* or through an *interrupt network*

- The interconnection structure between the memories and processors can be implemented using:

  - Time-shared common bus
  - Crossbar switch network
  - Multiport memories

Architectural classification scheme

- In general, digital computers may be classified into four categories, according to the *multiplicity of instruction and data streams* (Flynn's classification)

- An *instruction stream* is a sequence of instructions as executed by the machine; a *data stream* is a sequence of data including input, partial, or temporary results called for by the instruction stream

- There are four categories of digital computers based on the multiplicity of instruction and data streams:
  1. Single instruction stream-single data stream (**SISD**)
  2. Single instruction stream-multiple data stream (**SIMD**)
  3. Multiple instruction stream-single date stream (**MISD**)
  4. Multiple instruction stream-multiple data stream (**MIMD**)

- Conceptually, only three types of system components are needed in the illustration. Both instructions and data are fetched from the ***memory modules***. Instructions are decoded by the ***control unit***, which sends the decoded instruction stream to the ***processor units*** for execution

- *Each instruction stream* is generated by an ***independent control unit***. *Multiple data streams* originate from the subsystem of ***shared memory modules***

1. SISD computer organization

SISD

- This organization represents most serial computers available today
- Instructions are executed sequentially but may be overlapped in their execution stages (pipelining)
- An SISD computer may have more than one functional unit in It. All the functional units are under the supervision of one control unit
- Applications:
  - Whatever we do with our personal computers today

2. SIMD computer organization

SIMD

- In this organization, there are *multiple processing elements* supervised by the same *control unit*
- All PEs receive the *same instruction broadcast* from the *control unit* but operate on *different data sets* from *distinct data streams*
- The SIMD model of parallel computing consists of two parts:
  - A front-end computer of the usual von Neumann style
  - And a processor array
- Applications:
  - Image processing
  - Matrix manipulations
  - Sorting

3. MISD computer organization

MISD

- In this organization, there are ***n*** processor units, each *receiving distinct instructions* operating over the *same data stream* and its derivatives

- The *results* (output) of one processor become the *input* (operands) of the next processor in the macropipe.

- This structure has received much less attention and has been challenged as impractical by some computer architects. No real embodiment of this class exists.

- Applications:
    - Space shuttle flight control computers
    - Robot vision

3. MIMD computer organization

MIMD

- Most *multiprocessor systems* and *multiple computer systems* can be classified in this category
- An *intrinsic MIMD computer* implies interactions among the ***n*** processors
- An intrinsic MIMD computer is *tightly coupled* if the degree of interactions among the processors is high. Otherwise, we consider them *loosely coupled*
- Most commercial MIMD computers are loosely coupled.
- Applications:
  - Computer-aided design/computer-aided manufacturing
  - Simulation, modeling
  - Communication switches

GPU as Parallel Computers

Since 2003, the semiconductor industry has settled on two main trajectories for designing microprocessor (MP):

1. The multi-core trajectory:

- It seeks to maintain the execution speed of sequential programs while moving into multiple cores.

- *Multicore trajectory* began as two-core processors, with the number of cores doubling every generation of MP.
  - ➢ Eg: Intel Corei7 MP which has 4 processor cores, each of which is an out-of-order, multiple instruction issue processor implementing the full x86 instruction set.
  - ➢ This MP supports *hyperthreading* with two *hardware threads* per core and is designed to maximize the exec speed of sequential programs.

2. The many-core (many-thread) trajectory:

- It focuses more on the *execution throughput* of parallel applications

- The many-cores began as a *large number of much smaller cores*, and, once again, the number of cores doubles with each generation
  - ➢ Eg: The NVIDIA GeForce GTX 280 graphics processing unit (GPU) with 240 cores, each of which is a heavily multithreaded

- Many-core processors, especially the GPUs, dominated the race of Floating-point (FP) performance. The ratio between many-core GPUs and multicore CPUs for peak floating-point calculation throughput is about 10 to 1.

GPU vs CPU FP performance

-

- This gap has motivated many app developers to *move the computationally intensive parts of their software to GPUs for execution*.

- When there is more work to do, there is more opportunity to divide the work among cooperating parallel workers.

Why there is such a large performance gap between many-core GPUs and general-purpose multicore CPUs?

- The answer lies in the differences in the fundamental design philosophies between the two types of processors

GPU vs CPU architecture

CPU design principle:

- The design of a CPU is optimized for sequential code performance

- It makes use of sophisticated control logic to allow instructions from a single thread of execution to execute in parallel

- The large cache memories are provided to reduce the instruction and data access latencies of large complex applications ✉ *latency-oriented design*

- Memory bandwidth limit the speed of applications by limiting the rate at which data can be delivered from the memory system to processors.

Why there is such a large performance gap between many-core GPUs and general-purpose multicore CPUs?

GPU design principle:

- The design philosophy of the GPUs is shaped by the fast growing video game industry, which requires the ability to perform a massive number of floating-point calculations per video frame

- GPU performs well by *executing massive numbers of threads*

- GPU hardware takes advantage of a large number of execution threads to find work to do when some of them are waiting for long-latency memory accesses

- Small cache memories are provided to help the bandwidth requirements of applications, so multiple threads that access the same memory data need not always access the DRAM ✉ *throughput-oriented design*

- Most apps will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs

Factors to be considered by the application developers to choose the processors for running their applications

Along with the performance, there are several other factors which an application developer has to look upon when selecting a processor.

1. Installation base:

The processors of choice must have a *very large presence in the marketplace*. The cost of software development is reduced significantly with a very large customer population

2. Practical form factors and easy accessibility:

The parallel software applications should be *easily accessible to customers* and should provide the *solutions to a large set of common customer needs*

3. Support for the IEEE floating-point standard:

The standard makes it possible to have *predictable results across processors* from different vendors

Factors to be considered by the application developers to choose the processors for running their applications

4. Ease of programming for accessing the GPU cores:

- Early GPU programmers had to use the *equivalent of graphics API functions* to access the GPU cores (tedious task)
- This practice was called general-purpose programming using GPU (**GPGPU**)
- These APIs *limit the kinds of applications* that one can actually write for these chips

5. Introduction of CUDA:

- NVIDIA released CUDA (Compute Unified Device Architecture) and devoted silicon area in GPU to facilitate the ease of parallel programming
- Along with the *change in software*, *additional hardware was added* to the chip
- CUDA programs *do not* go through the graphics interface at all. Instead, a new *general-purpose parallel programming interface* on the silicon chip serves the requests of CUDA programs
- All of the other *software layers were redone* so the programmers can use the familiar C/C++ programming tools

Architecture of a Modern GPU

[Modern GPU architecture]

- It is organized into an array of highly threaded *streaming multiprocessors* (SMs)

- In the figure, two SMs form a building block; however, the number of SMs in a building block can vary from one generation of CUDA GPUs to another generation

- Each SM has a number of *streaming processors* (SPs) that share control logic and instruction cache

- Each GPU currently comes with up to 4 GB of graphics double data rate (GDDR) DRAM, referred to as *global memory*

GDDR DRAMs

- The GDDR DRAMs differ from the system DRAMs on the CPU motherboard in that they are essentially the frame buffer memory that is used for graphics processing
- For graphics applications, GDDR DRARs hold *video images*, and *texture information* for *three-dimensional (3D) rendering*
- For normal computing they function as *very-high-bandwidth, off-chip memory*, though with somewhat more latency than typical system memory. For massively parallel applications, the higher bandwidth makes up for the longer latency

[CUDA-paper]

Need for Parallelism – Why do we need ??

- Many applications that we have today will continue to demand for increased speed in future
- When an application is suitable for parallel execution, a good implementation on a GPU can achieve more than *100 times speedup over sequential execution*
- Many applications of the future will be what we currently consider to be *supercomputing applications* (Superapps)
- The limitations of todays scientific experiments can be effectively addressed by incorporating a GPU enabled computational model to *simulate* the underlying activities. From the simulation we can measure even more details and test more hypotheses than with traditional instruments

Need for Parallelism – Why do we need ??

- In the future, new functionalities such as view synthesis and high-resolution display of low-resolution videos will demand that televisions have more computing power
- future versions of user interfaces will incorporate higher definition, three-dimensional perspectives, voice and computer vision based interfaces, requiring even more computing speed
- With increased computing speed, the future games can be based on dynamic simulation rather than pre-arranged scenes
- Applications handling bigdata will require exploiting parallelism in datasets

Parallel processing applications

- ❑ **Predictive Modeling and Simulations**
  - Weather forecasting
  - Oceanography and astrophysics
  - Socioeconomics and government use

- ❑ **Engineering Design and Automation**
  - Finite-element analysis
  - Computational aerodynamics
  - Artificial intelligence and automation
  - Remote sensing applications

- ❑ **Energy Resources Exploration**
  - Seismic exploration
  - Reservoir modeling
  - Plasma fusion power
  - Nuclear reactor safety

- ❑ **Medical, Military, and Basic Research**
  - Computer-aided diagnosis
  - Genetic engineering
  - Weapon research and defense
  - Basic Research Problems

Parallel Programming Languages and Models

Some of the most popular software frameworks/standards that allow general-purpose programming using GPU (GPGPU) to accelerate processing in applications are:

- **MPI (Message Passing Interface)**
- **OpenMP (Open Multi-Processing)**
- **CUDA (Compute Unified Device Architecture)**
- **OpenCL (Open Computing Language)**

❖ GPGPU is the utilization of a GPU (graphics processing unit), which would typically only handle computer graphics, to assist in performing tasks that are traditionally handled solely by the CPU (central processing unit)

Parallel Programming Languages and Models

MPI (most of the versions are open source)

- Used for *scalable cluster computing*
- The computing nodes in a cluster *do not share memory*
- All data sharing and interaction among the nodes must be done through *explicit message passing*
- MPI have been known to run successfully on cluster computing systems with more than 100,000 nodes
- The amount of effort required to port an application into MPI can be *extremely high due to lack of shared memory across computing nodes*

OpenMP (open source)

- Used for *shared-memory multiprocessor* systems
- An openMP implementation consists of a *compiler* and a *runtime*
- A programmer specifies ***directives*** (commands) and ***pragmas*** (hints) about a loop to the openMP compiler. With these directives and pragmas, openMP compiler **generate parallel code**. The runtime system supports the **execution of the parallel code** by managing parallel threads and resources
- In OpenMP, compilers do more of the *automation* in managing parallel execution

Parallel Programming Languages and Models

CUDA (NVIDIA proprietary)

- Aspects of CUDA are similar to both MPI and OpenMP in that the *programmer manages the parallel code constructs*
- CUDA provides *shared memory* for parallel execution in the GPU
- CUDA achieves much *higher scalability* with simple, low-overhead thread management and no cache coherence hardware requirements
- many super-applications fit well into the simple thread management model of CUDA and thus enjoy the scalability and performance

OpenCL (open source)

- Several major industry players, including Apple, Intel, AMD/ATI, and NVIDIA, have jointly developed a *standardized programming model* called OpenCL
- Similar to CUDA, the OpenCL programming model defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery
- The level of programming constructs in OpenCL is still at a *lower level* than CUDA and much more tedious to use
- Also, the speed achieved in an application expressed in OpenCL is still *much lower* than in CUDA on the platforms that support both
- OpenCL is a standardized programming model in that applications developed in OpenCL can run without modification on all processors that support the OpenCL language extensions and API
- In comparison to CUDA, OpenCL relies more on APIs and less on language extensions

Parallel Programming Languages and Models

CUDA vs OpenCL

CPU vs GPU