PARALLEL & DISTRIBUTED PROCESSING
CSE-313

Md. Masudur Rahman, Dept. of CSE, UGV

PARALLEL & DISTRIBUTED PROCESSING

Course Code:	CSE-313	Credits:	03
		CIE Marks:	90
Exam Hours:	03	SEE Marks:	60

Course Learning Outcome (CLOs): After Completing this course successfully, the student will be able to…

CLO	Description
CLO 1	Demonstrate an understanding of parallel and distributed computing concepts, including their necessity, advantages, and architectural design.
CLO 2	Apply knowledge of Flynn’s taxonomy to classify parallel and distributed systems and their applications.
CLO 3	Analyze and solve computational problems using parallel computing techniques to improve performance and efficiency.
CLO 4	Develop programs using distributed computing models such as MapReduce for handling large-scale data processing.
CLO 5	Design and evaluate distributed databases with a focus on fragmentation, replication, and allocation strategies.
CLO 6	Assess the performance, fault tolerance, and scalability of distributed systems and suggest optimizations.
CLO 7	Implement synchronization techniques to manage shared resources and ensure consistency in distributed environments.
CLO 8	Explore the role of pipelining and instruction-level parallelism in improving the performance of parallel processors.

SUMMARY OF COURSE CONTENT

Sl.	Course Content	HRs	CLOs
1	Introduction to Parallel and Distributed Computing	4	CLO 1, CLO 2
2	Flynn’s Taxonomy and Parallel Architectures	3	CLO 2, CLO 3
3	Parallel Programming Models and Techniques	6	CLO 3, CLO 4
4	Distributed Systems: Concepts and Architectures	5	CLO 1, CLO 5
5	Synchronization and Resource Management	4	CLO 6, CLO 7
6	Pipelining and Instruction-Level Parallelism	3	CLO 8
7	MapReduce and Big Data Processing	4	CLO 4, CLO 6
8	Distributed Databases: Fragmentation, Replication, and Allocation	5	CLO 5, CLO 6
9	Fault Tolerance and Scalability in Distributed Systems	4	CLO 6
10	Practical Applications and Case Studies	2	CLO 4, CLO 6, CLO 7

- **Recommended Books:**

1. Parallel Programming: Techniques and Applications Using Networked Workstations and GPUs by Michael J. Quinn.
2. Distributed Systems: Principles and Paradigms by Andrew S. Tanenbaum and Maarten Van Steen.
3. Introduction to Parallel Computing by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar

ASSESSMENT PATTERN

CIE- Continuous Internal Evaluation (90 Marks)

Bloom's Category Marks (out of 90)	Tests (45)	Assignments (15)	Quizzes (15)	Attendance (15)
Remember	5	03		
Understand	5	04	05	
Apply	15	05	05	
Analyze	10			
Evaluate	5	03	05	
Create	5			

SEE- Semester End Examination (60 Marks)

Bloom's Category	Test
Remember	7
Understand	7
Apply	20
Analyze	15
Evaluate	6
Create	5

COURSE PLAN

Week No	Topics	Teaching Learning Strategy(s)	Assessment Strategy(s)	Alignment to CLO
1	Introduction to Parallel and Distributed Computing	Lectures, Class Discussions	Quiz, Participation	CLO 1
2	Flynn’s Taxonomy and Parallel Architectures	Lectures, Visual Diagrams	Assignment on Classification	CLO 2
3	Parallel Programming Concepts and Models	Interactive Coding Sessions, Demonstrations	Lab Exercises	CLO 3
4	Distributed System Architectures	Case Studies, Group Discussions	Midterm Quiz	CLO 1, CLO 5
5	Synchronization Techniques in Parallel Systems	Hands-On Labs, Collaborative Problem Solving	Lab Assessment	CLO 7
6	Pipelining and Instruction-Level Parallelism	Lectures, Simulations	Problem-Solving Assignment	CLO 8
7	MapReduce Framework Basics	Tutorials, Hands-On Exercises	Programming Assignment	CLO 4
8	Big Data and Distributed Processing Applications	Case Studies, Practical Labs	Lab Exercise	CLO 4, CLO 6
9	Distributed Databases: Fragmentation and Allocation	Lectures, Class Exercises	Quiz on Fragmentation Concepts	CLO 5
10	Data Replication and Fault Tolerance	Group Activities, Problem Solving	Case Study Report	CLO 6
11	Performance Optimization in Parallel Systems	Interactive Problem Solving, Discussions	Midterm Exam	CLO 6, CLO 8
12	Distributed Programming Techniques	Tutorials, Hands-On Practice	Programming Lab	CLO 4, CLO 5
13	Resource Management in Distributed Systems	Lectures, Interactive Discussions	Quiz	CLO 6, CLO 7
14	Scalability and Load Balancing in Distributed Systems	Case Studies, Simulations	Assignment on Scalability Strategies	CLO 6
15	Advanced MapReduce and Fault Tolerance Mechanisms	Lab Sessions, Research Presentations	Project Report	CLO 4, CLO 6
16	Applications of Parallel and Distributed Systems	Practical Applications, Guest Lectures	Final Project Presentation	CLO 4, CLO 6, CLO 7
17	Course Review and Final Exam	Q&A, Practice Tests	Final Exam	All CLOs

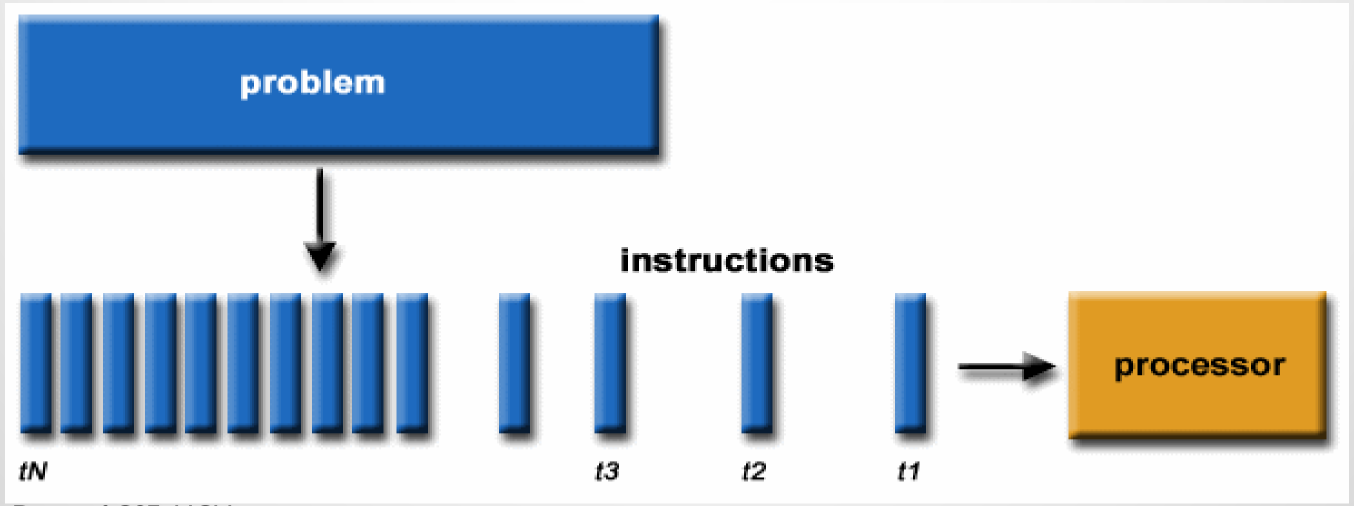
WEEK 1
SLIDES 6-25

PARALLEL PROCESSING

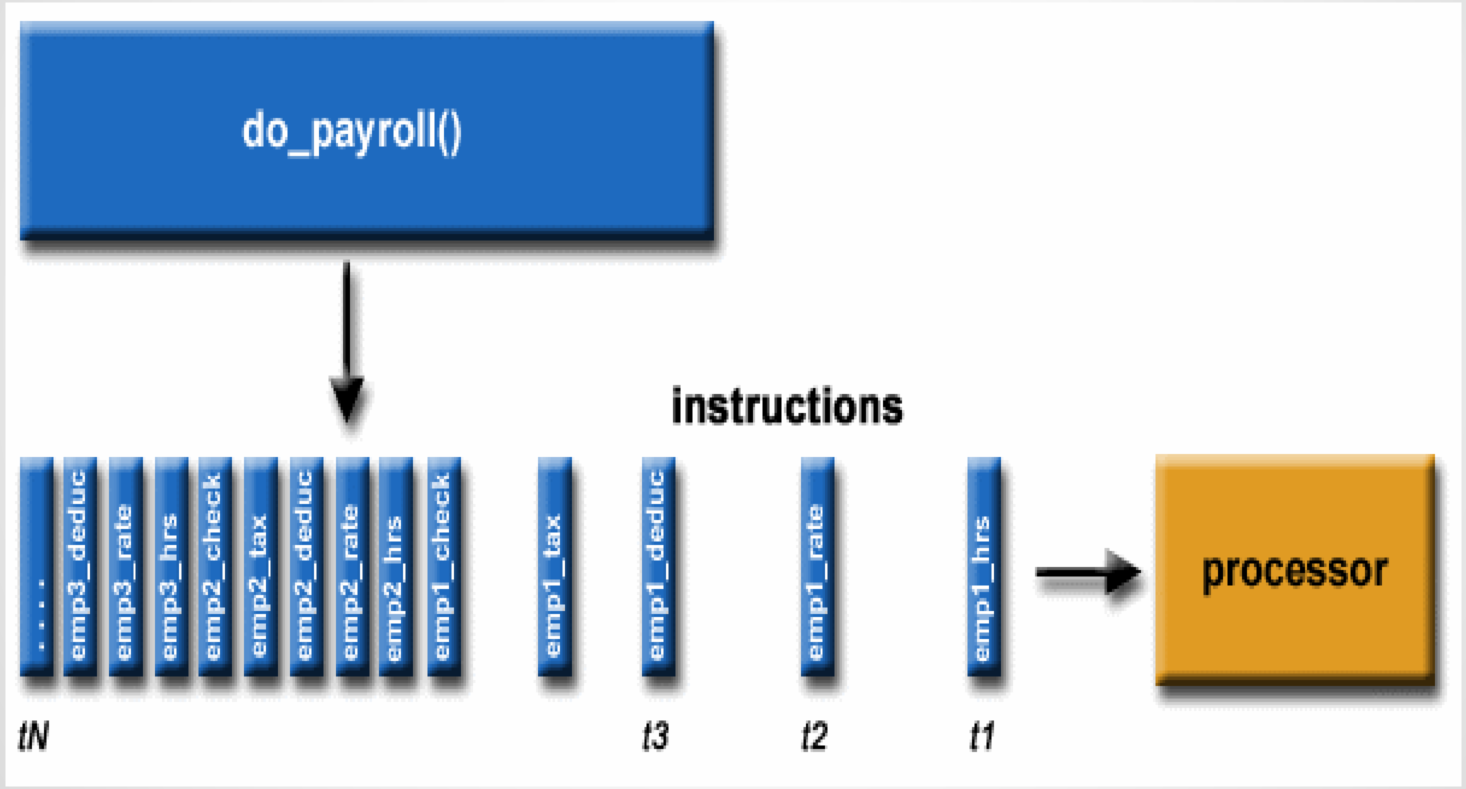
What is Serial Computing?

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

- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time

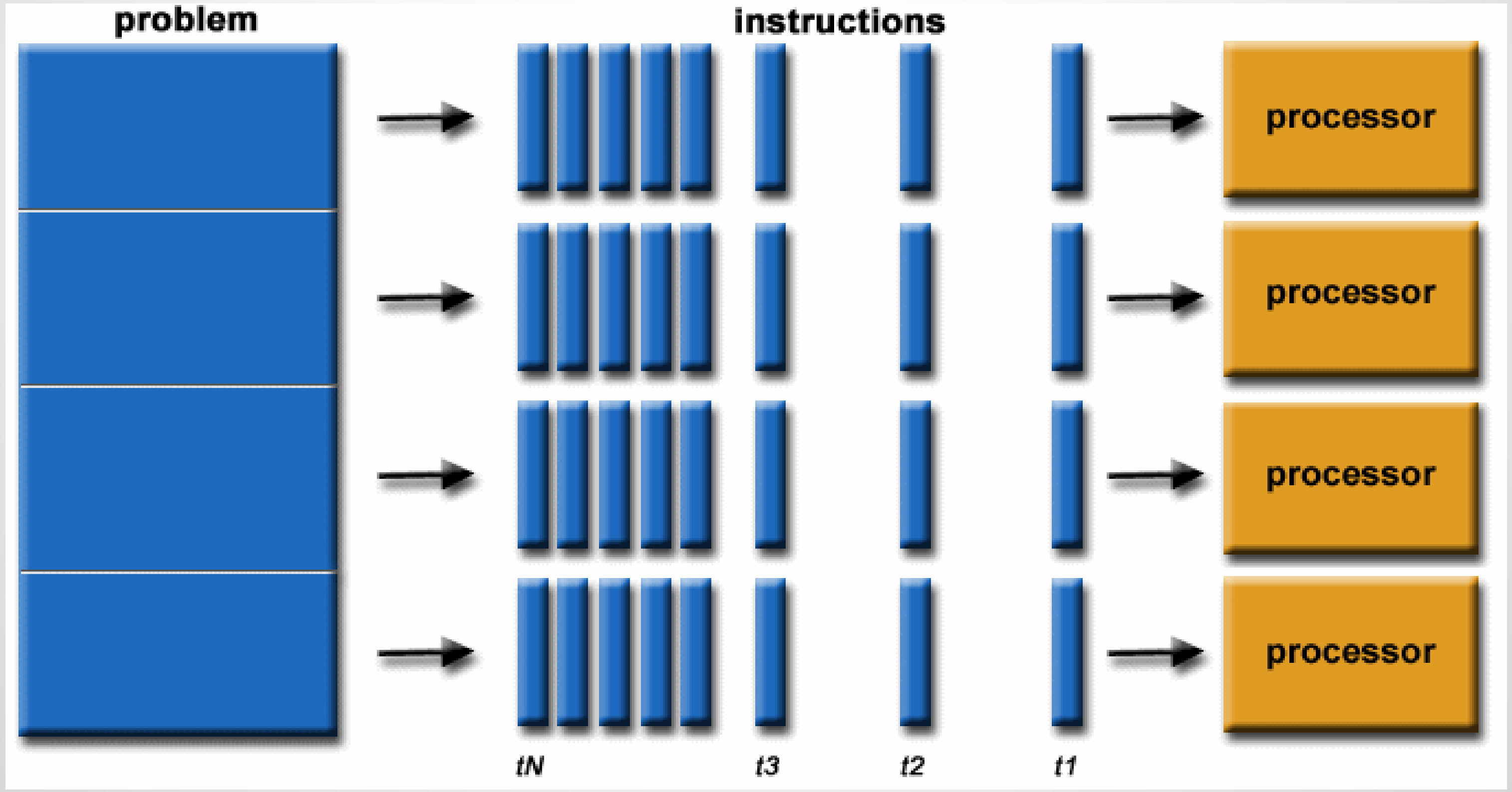


Example of Serial Computing



What is Parallel Computing?

In the simplest sense, ***parallel computing*** is the simultaneous use of multiple compute resources to solve a computational problem:



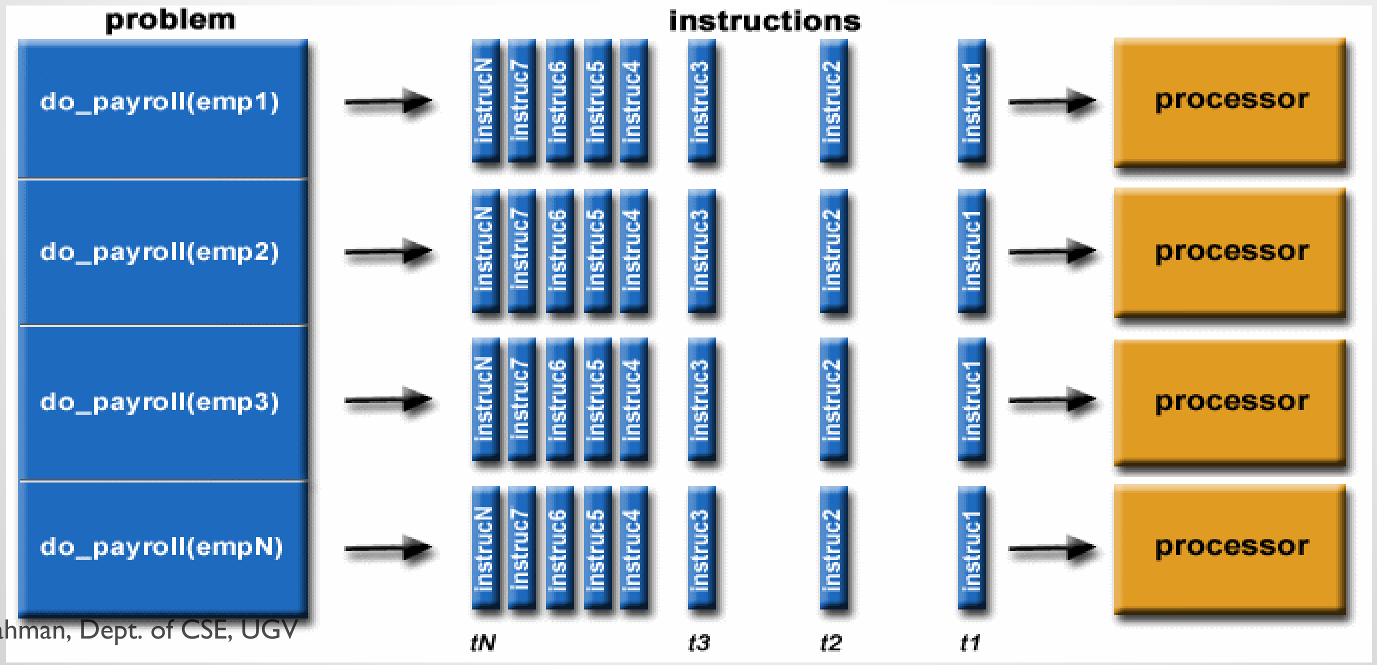
Parallel Computing

❖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 processors

❖An overall control/coordination mechanism is employed



Parallel Computing

- The computational problem should be able to:
  - Be broken apart into discrete pieces of work that can be solved simultaneously;
  - Execute multiple program instructions at any moment in time;
  - Be solved in less time with multiple compute resources than with a single compute resource.
- The compute resources are typically:
  - A single computer with multiple processors/cores
  - An arbitrary number of such computers connected by a network

Parallel Computers:

Virtually all stand-alone computers today are parallel from a hardware perspective:

- Multiple functional units (L1 cache, L2 cache, branch, decode, floating-point, graphics processing (GPU), etc.)
- Multiple execution units/cores
- Multiple hardware threads

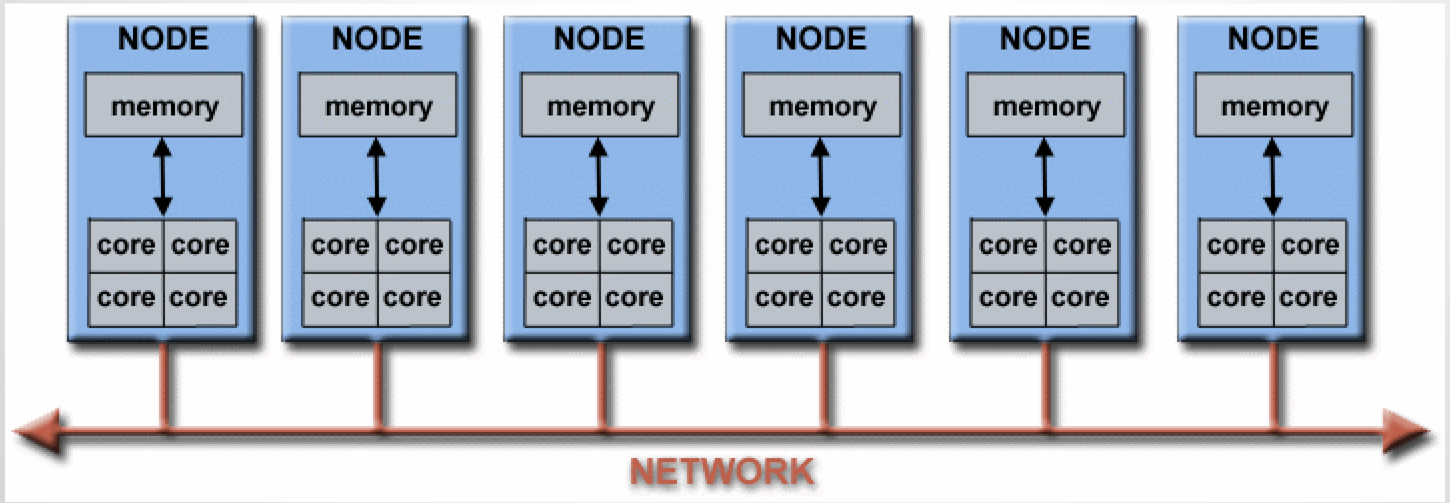
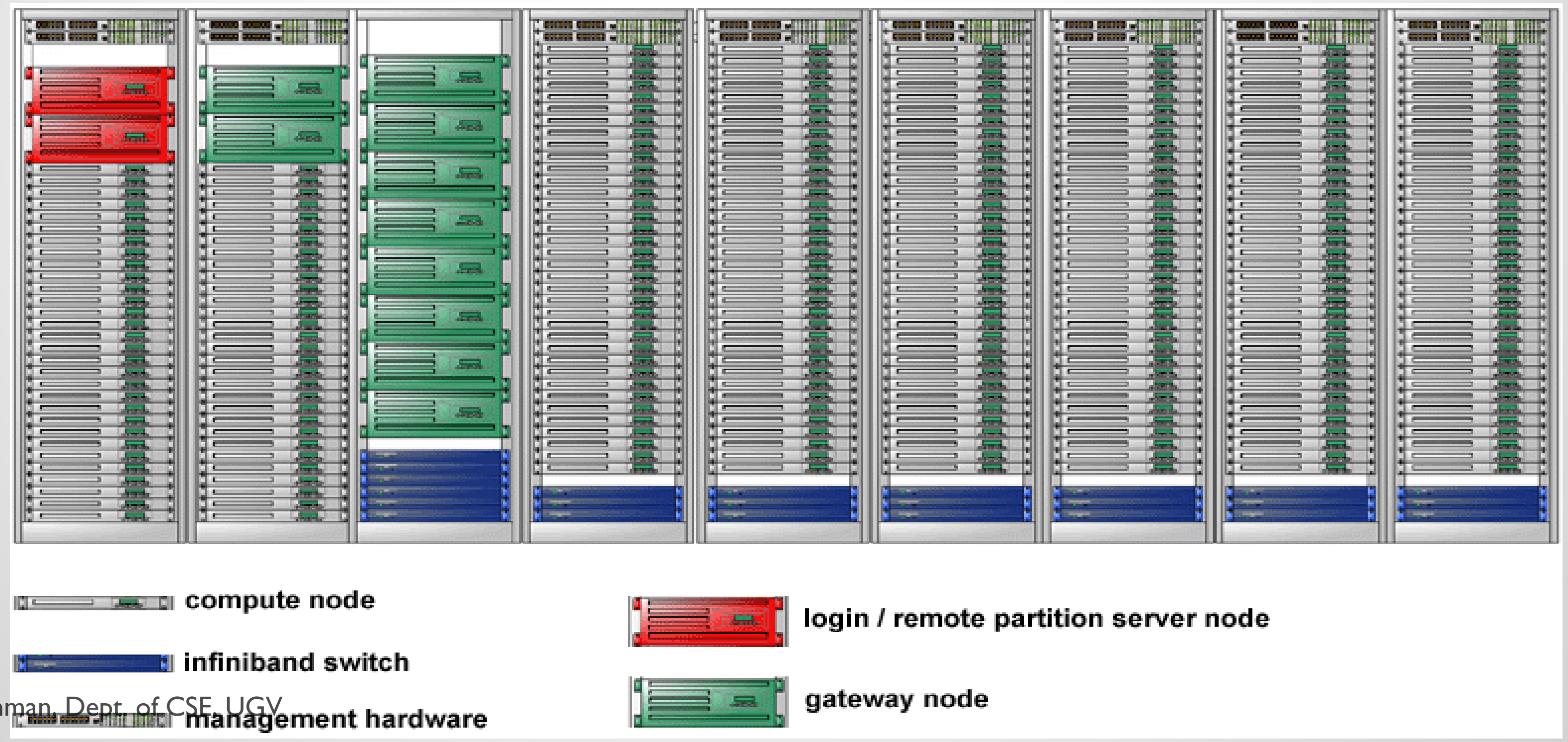


Fig: Networks connect multiple stand-alone computers (nodes) to make larger parallel computer clusters

A typical LLNL parallel computer cluster

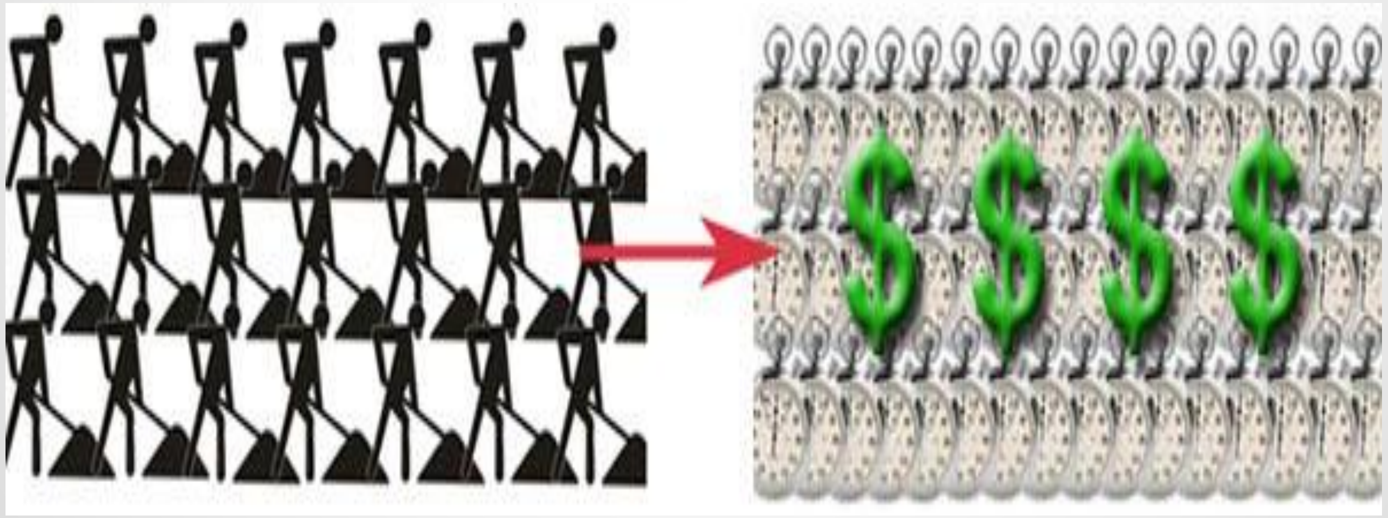
- Each compute node is a multi-processor parallel computer in itself
- Multiple compute nodes are networked together with an InfiniBand (IB) network
- Special purpose nodes, also multi-processor, are used for other purposes



Why Use Parallel Computing?

SAVE TIME AND/OR MONEY :

- In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings.
- Parallel computers can be built from cheap, commodity components.



SOLVE LARGER / MORE COMPLEX PROBLEMS:

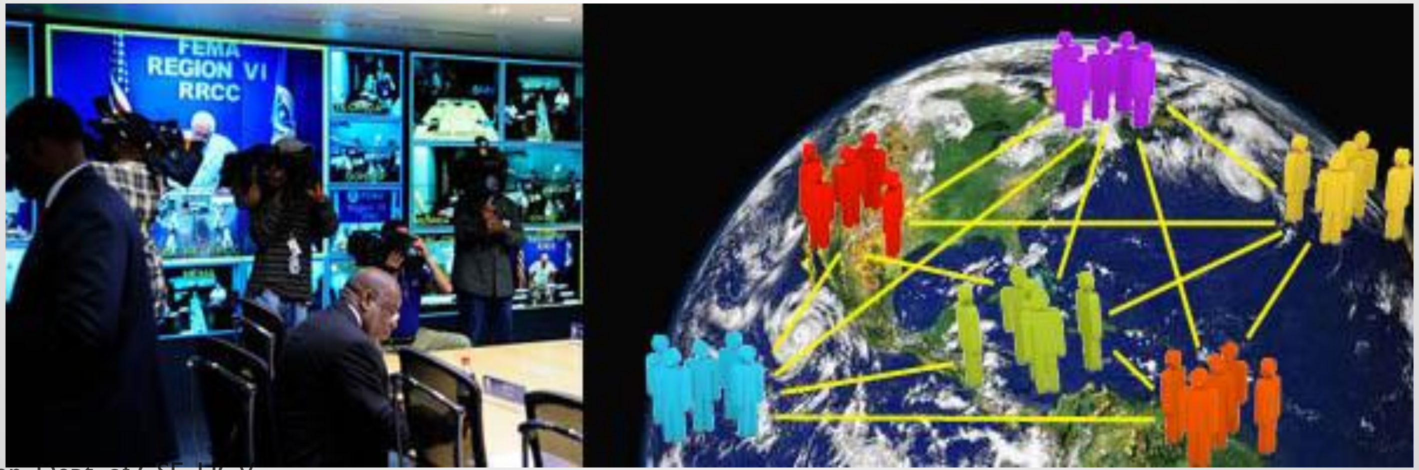
- Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.
- Example: Web search engines/databases processing millions of transactions every second

Why Use Parallel Computing?

PROVIDE CONCURRENCY:

✓A single compute resource can only do one thing at a time. Multiple compute resources can do many things simultaneously.

✓Example: Collaborative Networks provide a global venue where people from around the world can meet and conduct work "virtually".

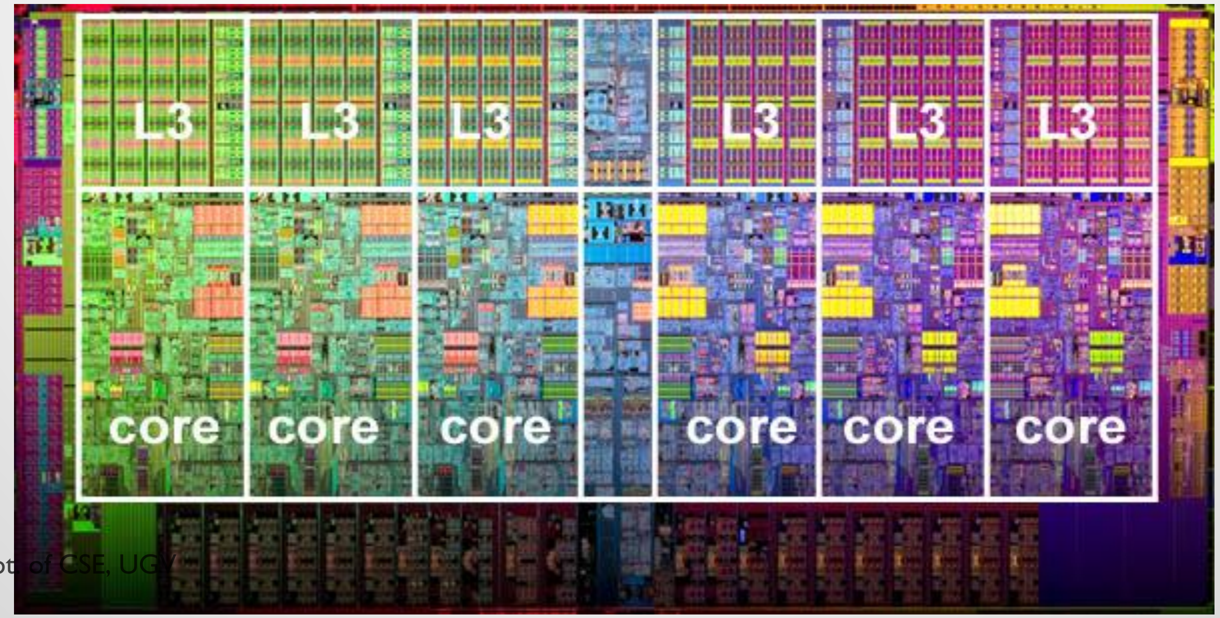


TAKE ADVANTAGE OF NON-LOCAL RESOURCES:

Using compute resources on a wide area network, or even the Internet when local compute resources are scarce or insufficient.

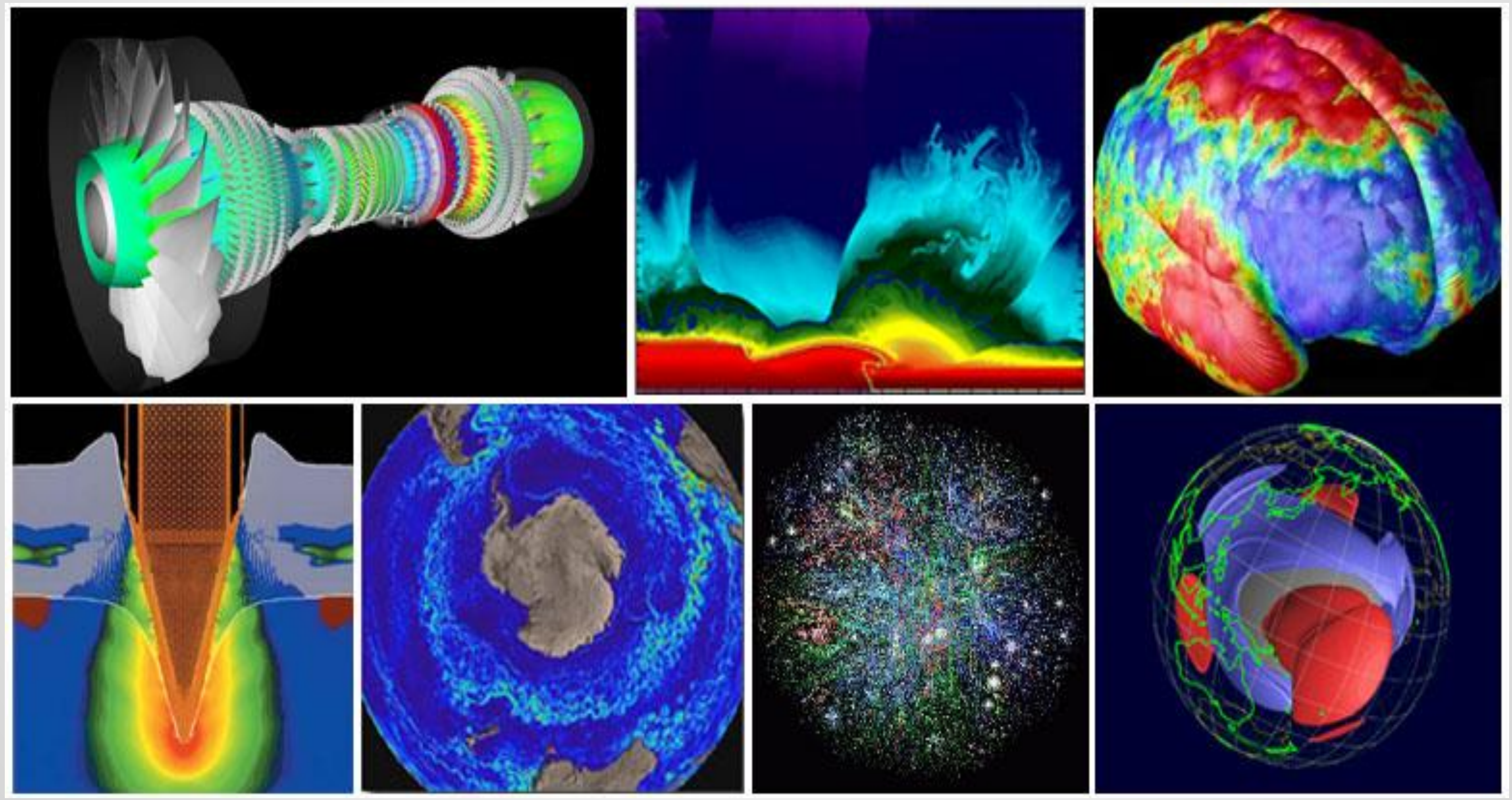
MAKE BETTER USE OF UNDERLYING PARALLEL HARDWARE:

- Modern computers, even laptops, are parallel in architecture with multiple processors/cores.
- Parallel software is specifically intended for parallel hardware with multiple cores, threads, etc.
- In most cases, serial programs run on modern computers "waste" potential computing power.



Who is Using Parallel Computing?

Historically, parallel computing has been considered to be "the high end of computing", and has been used to model difficult problems in many areas of **science and engineering**:



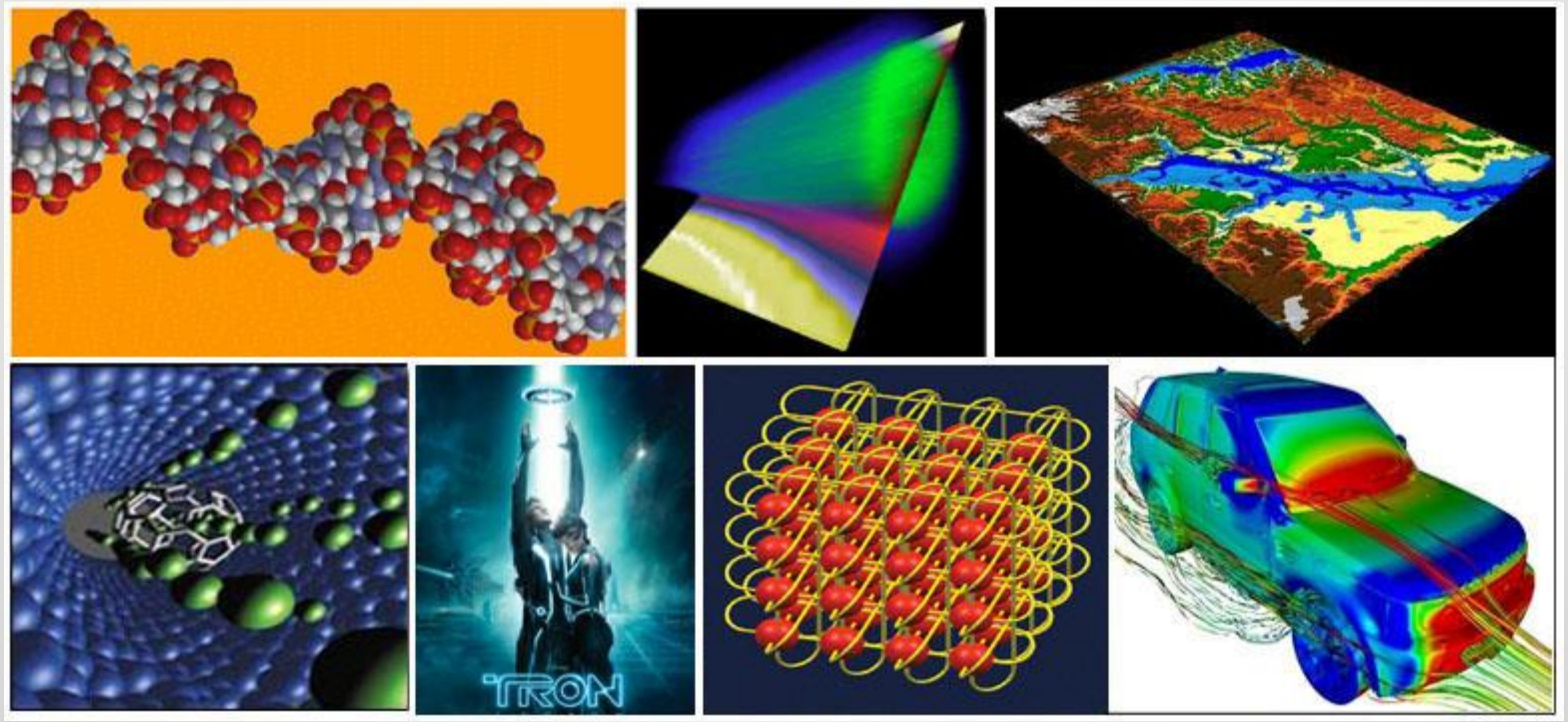
Who is Using Parallel Computing?

Science and Engineering:

- Atmosphere, Earth, Environment
- Physics - applied, nuclear, particle, condensed matter, high pressure, fusion, photonics
- Bioscience, Biotechnology, Genetics
- Chemistry, Molecular Sciences
- Defense, Geology
- Mechanical Engineering - from prosthetics to spacecraft
- Electrical Engineering, Circuit Design, Microelectronics
- Computer Science, Mathematics

Who is Using Parallel Computing?

Industrial and Commercial:
Today, commercial applications provide an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. For example:



Who is Using Parallel Computing?

Industrial and Commercial:

- "Big Data", databases, data mining
- Oil exploration
- Web search engines, web based business services
- Medical imaging and diagnosis
- Pharmaceutical design
- Financial and economic modeling
- Management of national and multi-national corporations
- Advanced graphics and virtual reality, particularly in the entertainment industry
- Networked video and multi-media technologies
- Collaborative work environments

Open Questions

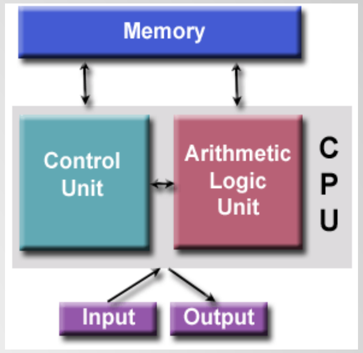
http://www.avego.com/blog/wp-content/uploads/2013/06/Your-Questions-Answered.jpg

THANK YOU

WEEK 2
SLIDES 26-37

Parallel Design and Programming

Von Neumann Architecture



•Comprised of four main components:

–Memory

–Control Unit

–Arithmetic Logic Unit

–Input / Output

•Read/write, random access memory is used to store both program instructions and data

–Program instructions are coded data which tell the computer to do something

–Data is simply information to be used by the program

•Control unit fetches instructions/data from memory, decodes the instructions and then ***sequentially*** coordinates operations to accomplish the programmed task.

•Aritmetic Unit performs basic arithmetic operations

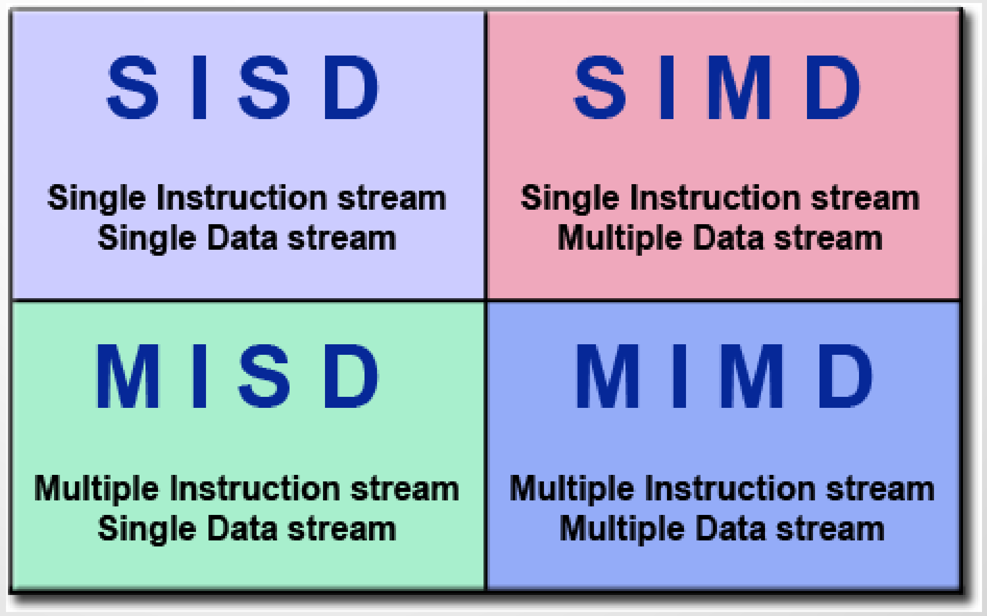
•Input/Output is the interface to the human operator

Parallel computers still follow this basic design, just multiplied in units. The basic, fundamental architecture remains the same.

Flynn's Classical Taxonomy

Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of ***Instruction Stream*** and ***Data Stream***. Each of these dimensions can have only one of two possible states: ***Single*** or ***Multiple***.

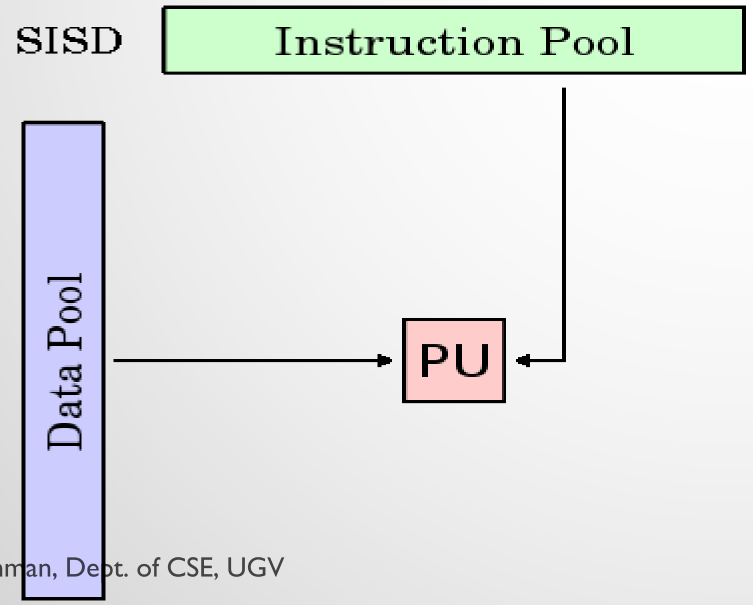
•The matrix below defines the 4 possible classifications according to Flynn:

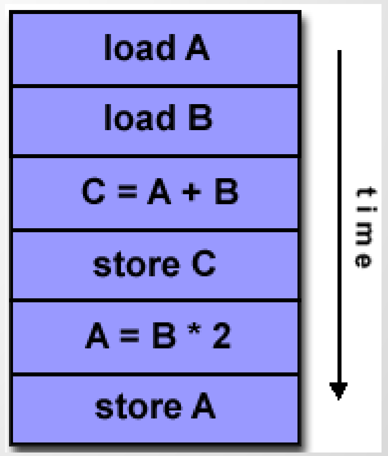


S I S D Single Instruction stream Single Data stream	**S I M D** Single Instruction stream Multiple Data stream
M I S D Multiple Instruction stream Single Data stream	**M I M D** Multiple Instruction stream Multiple Data stream

Single Instruction, Single Data (SISD):

1. A serial (non-parallel) computer
2. **Single Instruction:** Only one instruction stream is being acted on by the CPU during any one clock cycle
3. **Single Data:** Only one data stream is being used as input during any one clock cycle
4. Deterministic execution
5. This is the oldest type of computer
6. Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs.

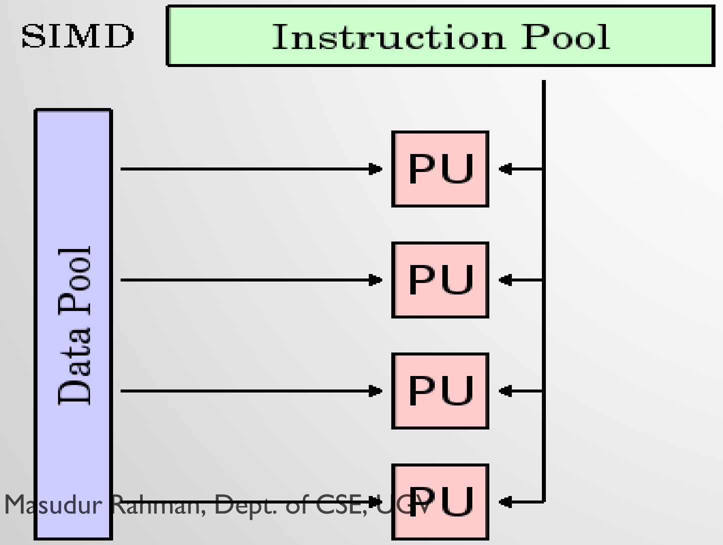


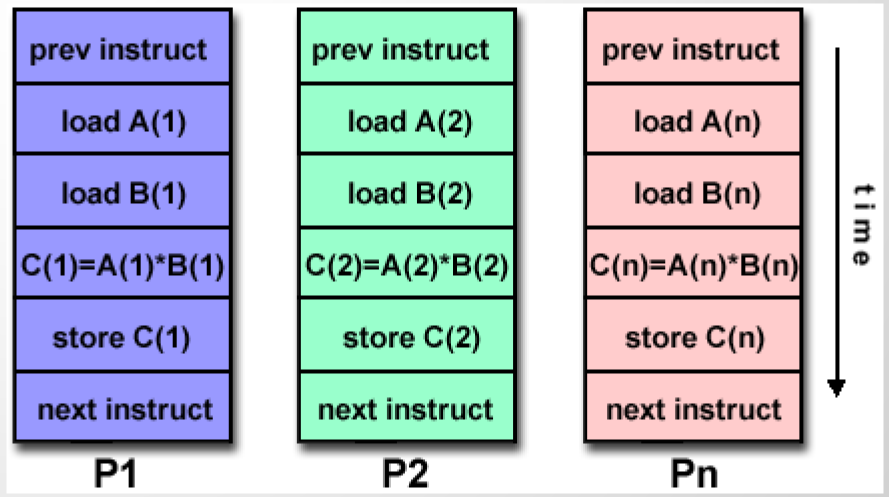


Single Instruction, Multiple Data (SIMD):

•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
•Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
•Synchronous (lockstep) and deterministic execution
•Two varieties: Processor Arrays and Vector Pipelines

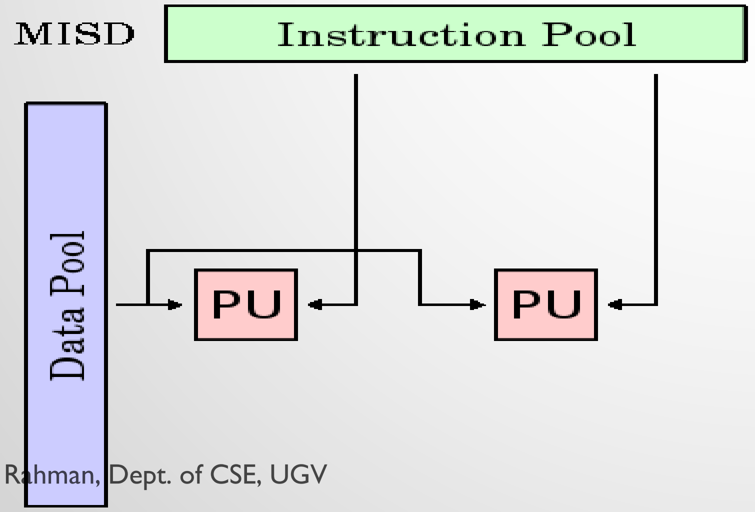
•Most modern computers, particularly those with graphics processor units (GPUs) employ SIMD instructions and execution units.

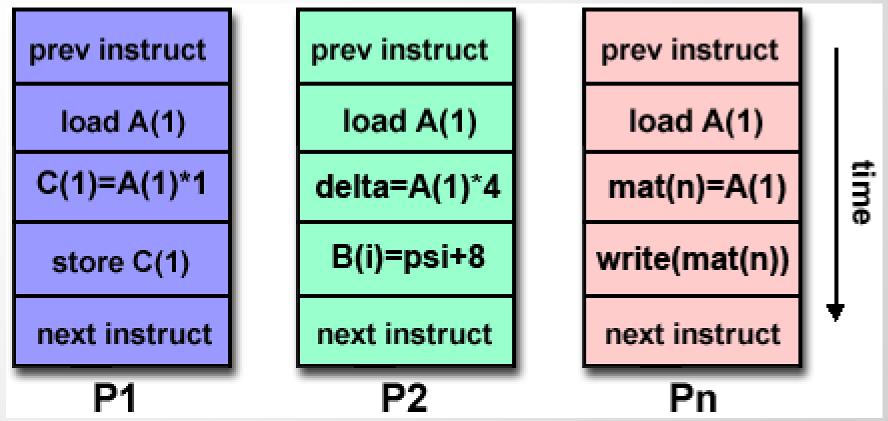




Multiple Instruction, Single Data (MISD)

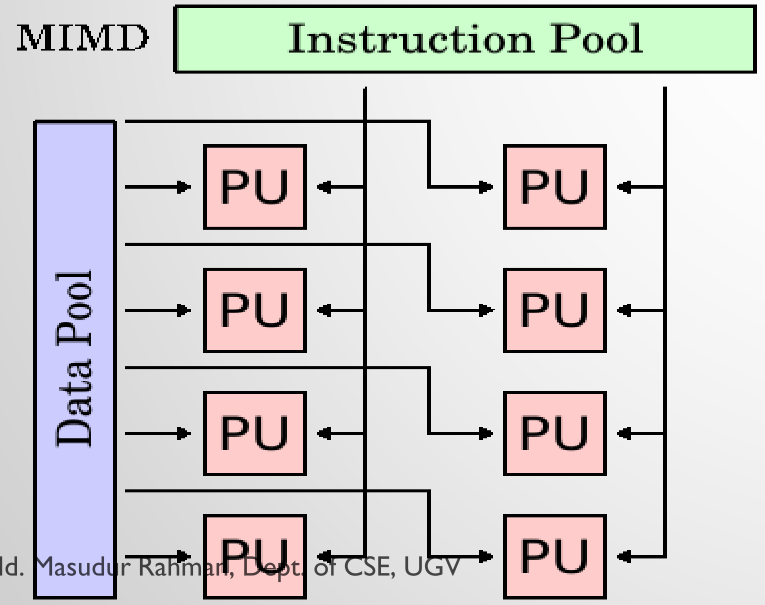
- A type of parallel computer
- **Multiple Instruction:** Each processing unit operates on the data independently via separate instruction streams.
- **Single Data:** A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever existed.
- Some conceivable uses might be:
  - multiple cryptography algorithms attempting to crack a single coded message.

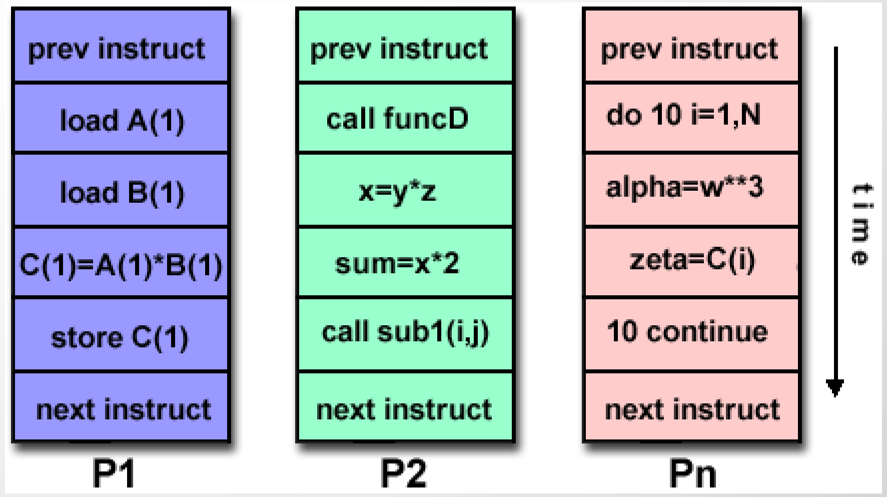




Multiple Instruction, Multiple Data (MIMD):

•A type of parallel computer
•**Multiple Instruction:** Every processor may be executing a different instruction stream
•**Multiple Data:** Every processor may be working with a different data stream
•Execution can be synchronous or asynchronous, deterministic or non-deterministic
•Currently, the most common type of parallel computer - most modern supercomputers fall into this category.
•Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
•Note: many MIMD architectures also include SIMD execution sub-components





Some General Parallel Terminology

- **Supercomputing / High Performance Computing (HPC) :** Using the world's fastest and largest computers to solve large problems.
- **Node :** A standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc. Nodes are networked together to comprise a supercomputer.
- **CPU / Socket / Processor / Core :**
  - ❑CPU (Central Processing Unit) was a singular execution component for a computer.
  - ❑Multiple CPUs were incorporated into a node.
  - ❑Individual CPUs were subdivided into multiple "cores", each being a unique execution unit.
  - ❑CPUs with multiple cores are sometimes called "sockets" - vendor dependent. The result is a node with multiple CPUs, each containing multiple cores.

•**Task :** A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.

•**Pipelining :** Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.

•**Shared Memory :** From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

•**Distributed Memory :** In hardware, refers to network based memory access for physical memory that is not common. As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing

•**Symmetric Multi-Processor (SMP) :** Shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources.

- **Synchronization :** The coordination of parallel tasks in real time, very often associated with communications. Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.
- **Massively Parallel :** Refers to the hardware that comprises a given parallel system - having many processing elements. The meaning of "many" keeps increasing, but currently, the largest parallel computers are comprised of processing elements numbering in the hundreds of thousands to millions.
- **Embarrassingly Parallel :** Solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks.
- **Scalability :** Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources. Factors that contribute to scalability include:
    - Hardware - particularly memory-cpu bandwidths and network communication properties
    - Application algorithm
    - Parallel overhead related
    - Characteristics of your specific application

Costs of Parallel Programming

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

1 speedup
= -------- 1 -
P

•If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
•If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
•If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.
•Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$\textbf{speedup} = \frac{1}{\frac{P}{N} + S}$$

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

WEEK 3
SLIDES 38-50

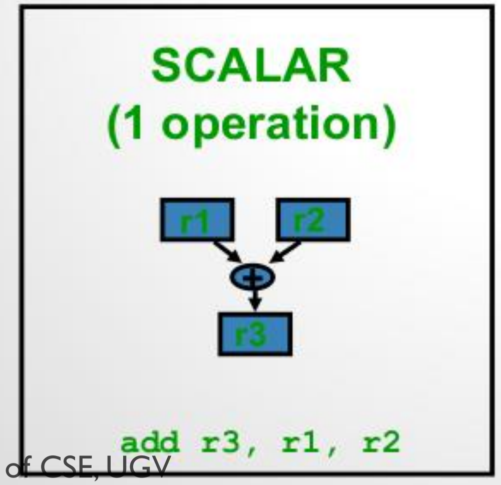
PARALLEL COMPUTER MODEL

OUTLINE

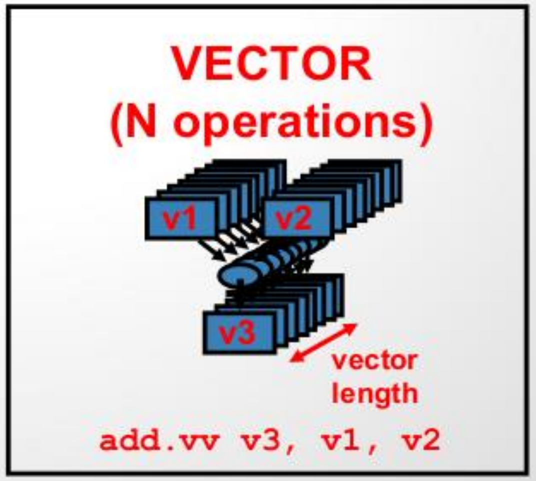
- Multivector Computer
  - Description
  - Advantages
  - Architecture
    - Vector supercomputers
    - Memory-to-memory
    - Register-to-register
- SIMD Computer

DESCRIPTION OF VECTOR PROCESSORS

- CPU that implements an instruction set that operates on 1-D arrays, called *vectors*
- Vectors contain multiple data elements
- Number of data elements per vector is typically referred to as the *vector length*
- Both instructions and data are pipelined to reduce decoding time

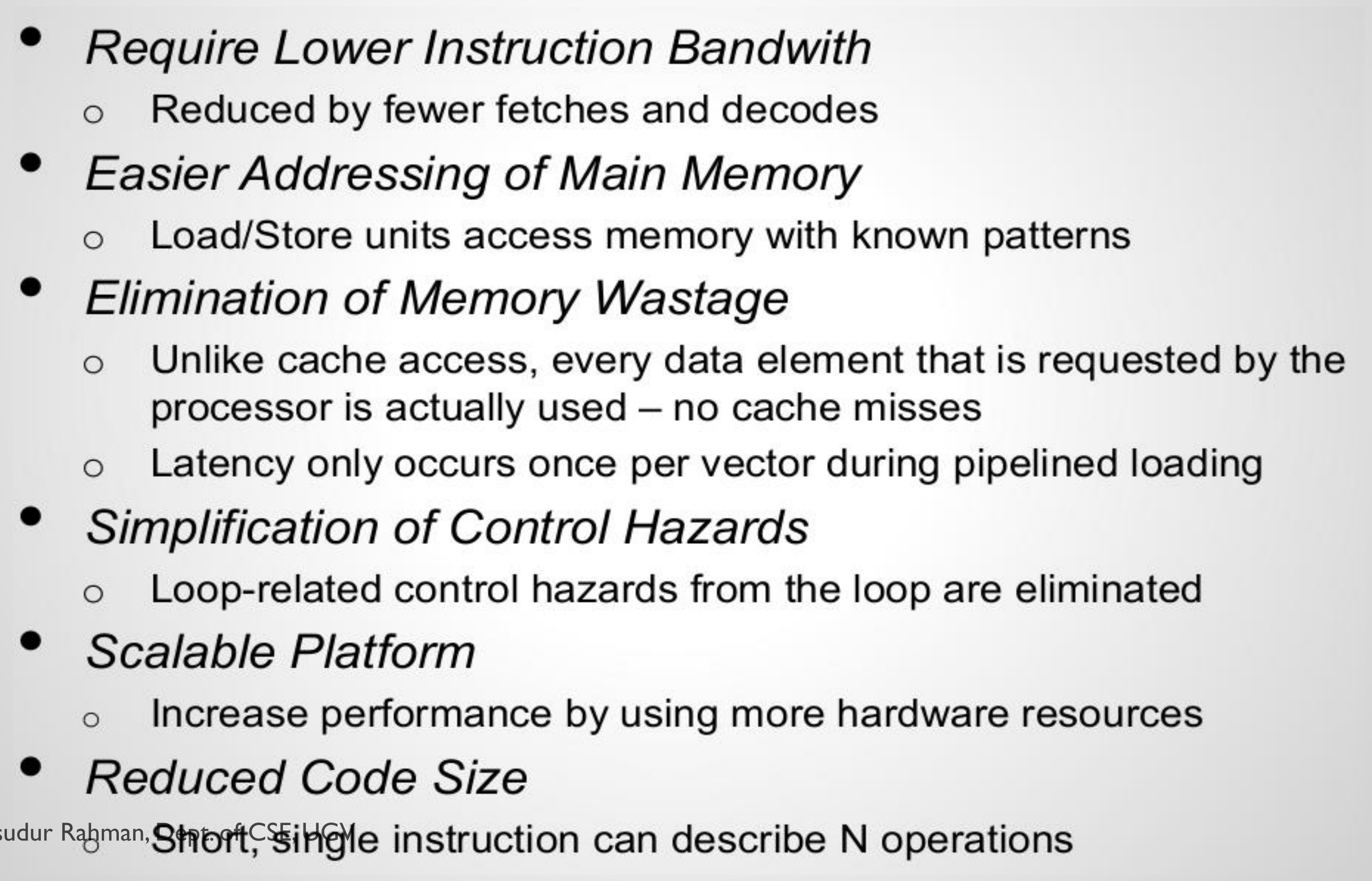


```
add r3, r1, r2
```



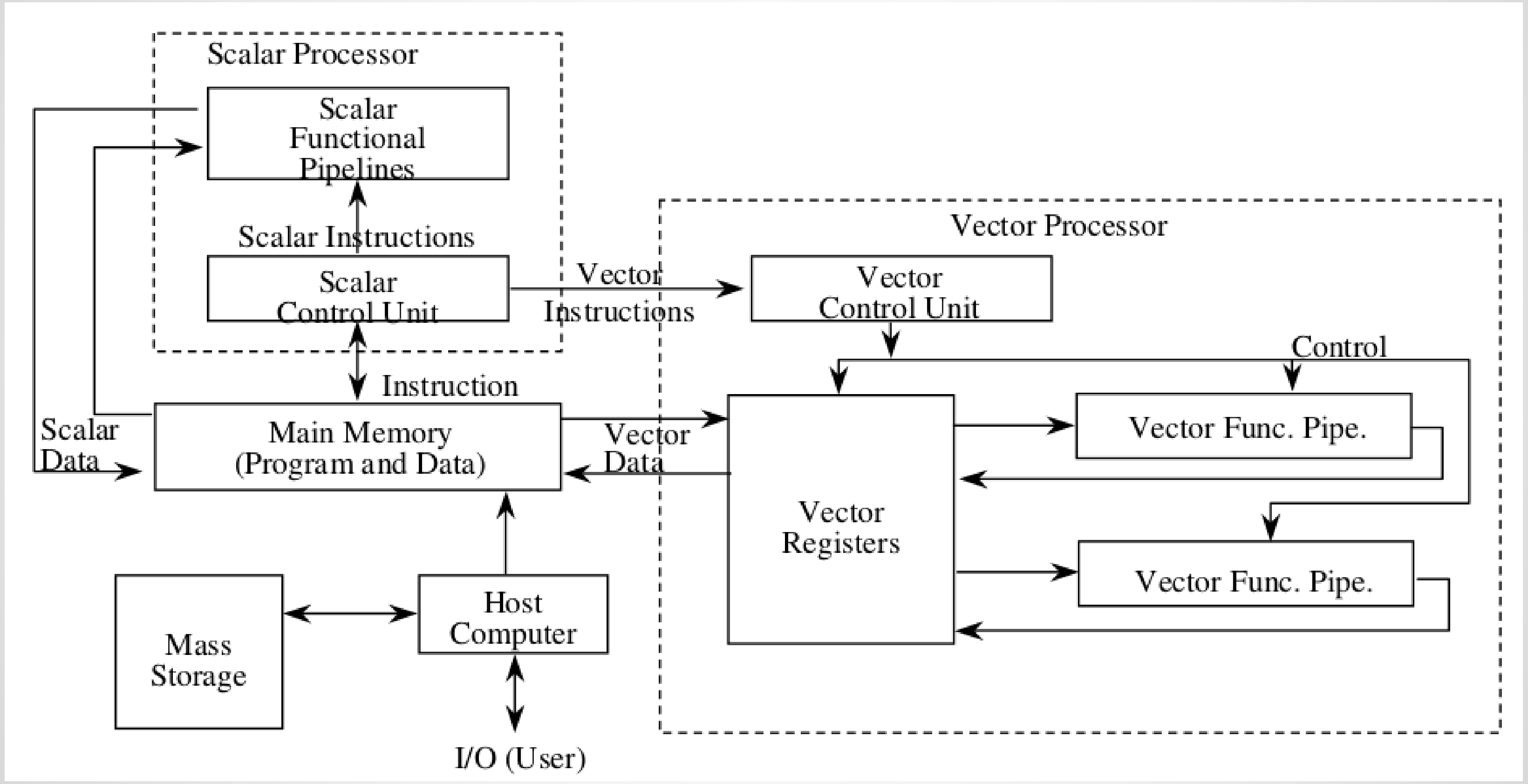
```
add.vv v3, v1, v2
```

ADVANTAGES OF VECTOR PROCESSORS



- *Require Lower Instruction Bandwith*
  - Reduced by fewer fetches and decodes
- *Easier Addressing of Main Memory*
  - Load/Store units access memory with known patterns
- *Elimination of Memory Wastage*
  - Unlike cache access, every data element that is requested by the processor is actually used – no cache misses
  - Latency only occurs once per vector during pipelined loading
- *Simplification of Control Hazards*
  - Loop-related control hazards from the loop are eliminated
- *Scalable Platform*
  - Increase performance by using more hardware resources
- *Reduced Code Size*
  - Short, single instruction can describe N operations

ARCHITECTURE OF A VECTOR SUPERCOMPUTERS

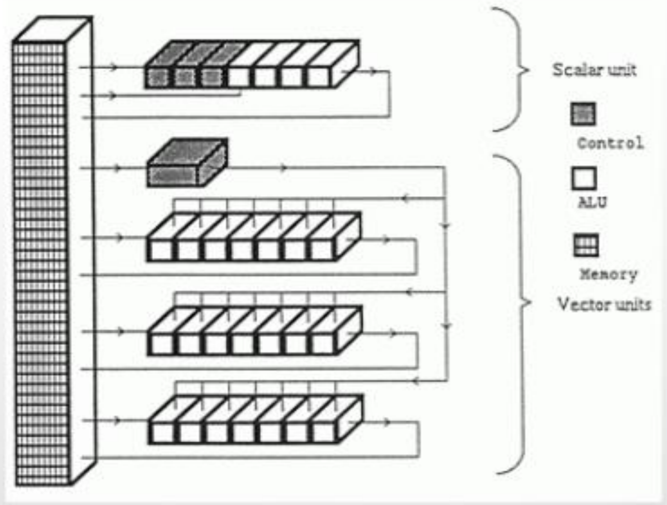


ARCHITECTURE OF A VECTOR SUPERCOMPUTERS(CONT)

- Often build on top of a scalar processor
- Vector processor is attached to the scalar processor as an optional feature
- Program and data are first loaded into the main memory through a host computer
- All instructions are first decoded by the **scalar control unit**. If the decoded instruction is a scalar operation or a program control then directly executed by the scalar processor using the scalar functional pipelines
- If the instruction is decoded as a vector operation then sent to the vector control unit(VCU).VCU supervise the flow of vector data between the main memory and vector functional pipelines
- Two pipeline vector supercomputer models
  - Register-to-register
  - Memory-to-memory

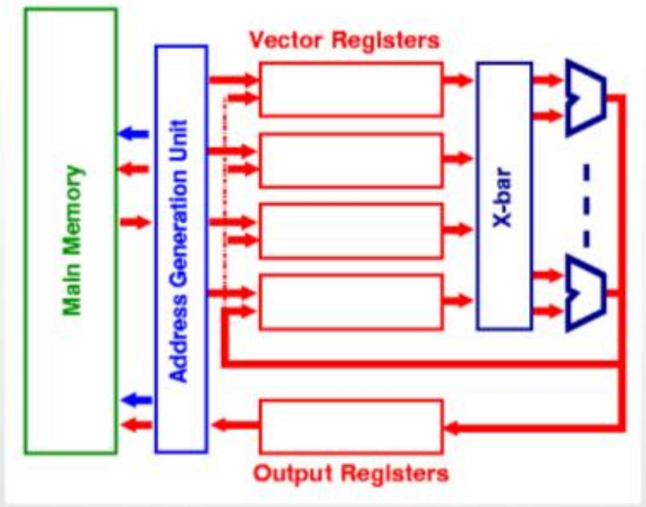
VECTOR PROCESSOR ARCHITECTURES

- *Memory-to-Memory Architecture* (Traditional)
  - For all vector operation, operands are fetched directly from main memory, then routed to the functional unit
  - Results are written back to main memory
  - Includes early vector machines through mid 1980s:
    - Advanced Scientific Computer (TI), Cyber 200 & ETA-10
  - Major reason for demise was due to *large startup time*



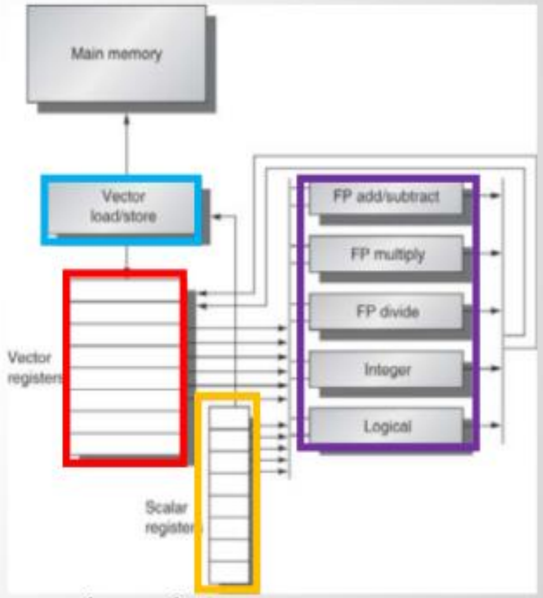
VECTOR PROCESSOR ARCHITECTURES (CONT)

- *Register-to-Register Architecture* (Modern)
  - All vector operations occur between vector registers
  - If necessary, operands are fetched from main memory into a set of vector registers (load-store unit)
  - Includes all vector machines since the late 1980s:
    - Convex, Cray, Fujitsu, Hitachi, NEC
  - SIMD processors are based on this architecture



COMPONENTS OF VECTOR PROCESSORS

- *Vector Registers*
  - Typically 8-32 vector registers with 64 - 128 64-bit elements
  - Each contains a vector of double-precision numbers
  - Register size determines the maximum vector length
  - Each includes at least 2 read and 1 write ports
- *Vector Functional Units (FUs)*
  - Fully pipelined, new operation every cycle
  - Performs arithmetic and logic operations
  - Typically 4-8 different units
- *Vector Load-Store Units (LSUs)*
  - Moves vectors between memory and registers
- *Scalar Registers*
  - Single elements for interconnecting FUs, LSUs, and registers



SIMD SUPERCOMPUTERS

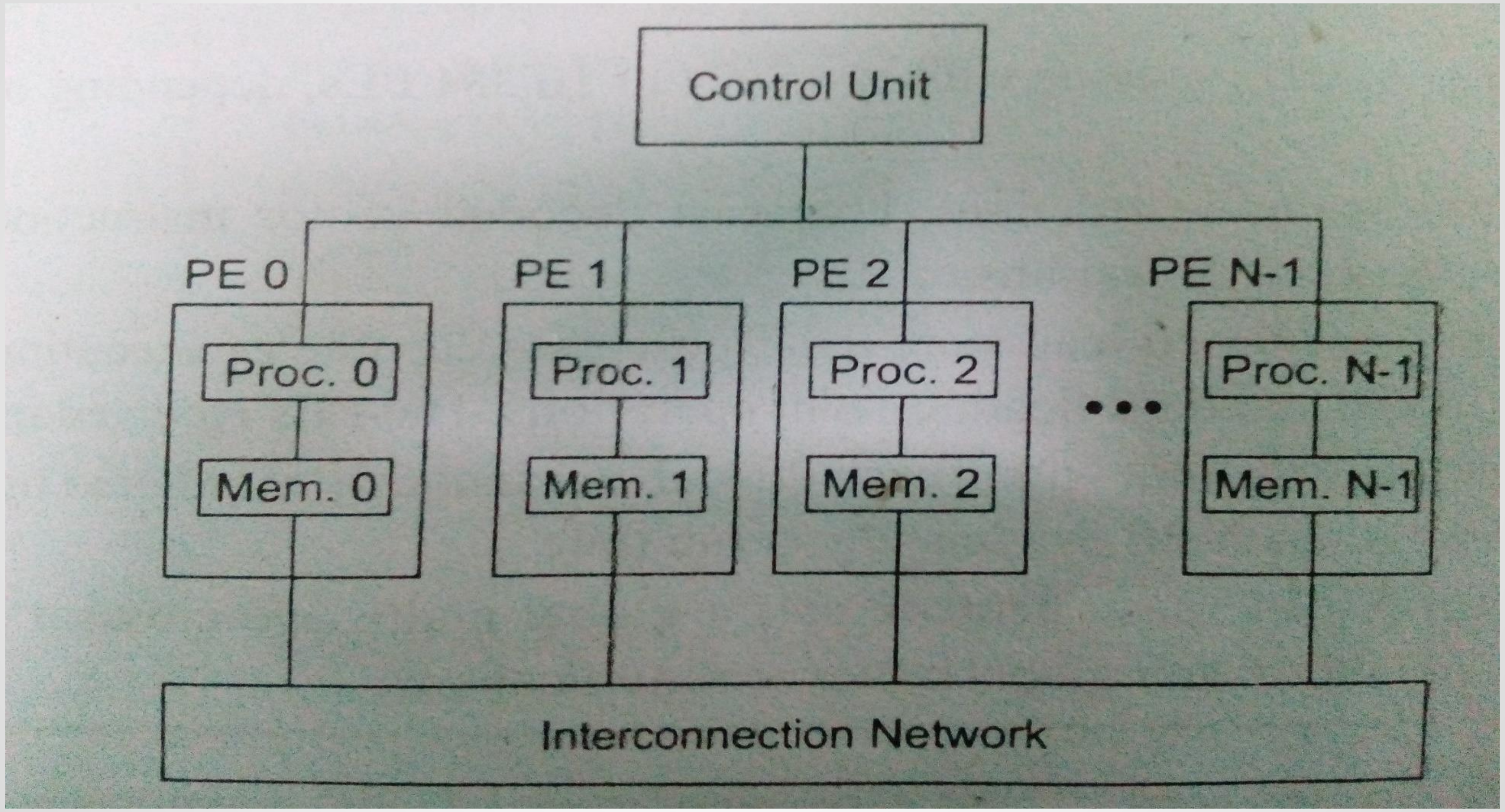


Fig. : Operational Model of SIMD computers

SIMD MACHINE MODEL

- An operational model of an SIMD computer is specified by a 5-tuple: M = (N, C, I, M, R) where
  - N is the number of processing elements (PEs)
  - C is the set of instructions directly executed by the CU, including scalar and program flow control instructions
  - I is the set of instructions broadcast by the CU to all PEs for parallel execution
  - M is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets
  - R is the set of data-routing functions, specifying various patterns to be setup in the interconnection network for inter-PE communication

QUESTIONS?

- THANK YOU

WEEK 4
SLIDES 51-65

Program and Network Properties

- Conditions of parallelism
- Program partitioning and scheduling
- Program flow mechanisms
- System interconnect architectures

Conditions of Parallelism

- The exploitation of parallelism in computing requires understanding the basic theory associated with it. Progress is needed in several areas:
  - **computation models** for parallel computing
  - **interprocessor communication** in parallel architectures
  - **integration of parallel systems** into general environments

Data dependences

The ordering relationship between statements is indicated by the data dependence.

- Flow dependence
- Anti dependence
- Output dependence
- I/O dependence
- Unknown dependence

Data Dependence - 1

- **Flow dependence:** S1 precedes S2, and at least one output of S1 is input to S2.
- **Antidependence:** S1 precedes S2, and the output of S2 overlaps the input to S1.
- **Output dependence:** S1 and S2 write to the same output variable.
- **I/O dependence:** two I/O statements (read/write) reference the same variable, and/or the same file.

Data Dependence - 2

- **Unknown dependence:**
    - The subscript of a variable is itself subscripted.
    - The subscript does not contain the loop index variable.
    - A variable appears more than once with subscripts having different coefficients of the loop variable (that is, different functions of the loop variable).
    - The subscript is nonlinear in the loop index variable.
- Parallel execution of program segments which do not have total data independence can produce non-deterministic results.

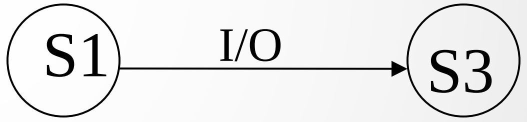
Data dependence example

S1: Load R1, A
S2: Add R2, R1
S3: Move R1, R3
S4: Store B, R1

$$\begin{tikzcd}
& S1 \arrow[dl] \arrow[dr] \arrow[dd, "\circ" marking] & \\
S2 \arrow[dr, "/" marking] & & S4 \\
& S3 \arrow[ur] &
\end{tikzcd}$$

I/O dependence example

S1: Read (4), A(I)
S2: Rewind (4)
S3: Write (4), B(I)
S4: Rewind (4)


$$\begin{tikzcd}\boxed{S1} \arrow[r, "I/O"] & \boxed{S3}\end{tikzcd}$$

Control dependence

- The order of execution of statements cannot be determined before run time
    - Conditional branches
    - Successive operations of a looping procedure

Control dependence examples

```
    Do 20 I = 1, N
    A(I) = C(I)
    IF(A(I) .LT. 0) A(I)=1
20  Continue
```

```
    Do 10 I = 1, N
     IF(A(I-1) .EQ. 0) A(I)=0
10  Continue
```

Resource dependence

- Concerned with the conflicts in using shared resources
  - Integer units
  - Floating-point units
  - Registers
  - Memory areas
  - ALU
  - Workplace storage

Bernstein's conditions

- Set of conditions for two processes to execute in parallel

  $I_1 \cap O_2 = \emptyset$

  $I_2 \cap O_1 = \emptyset$

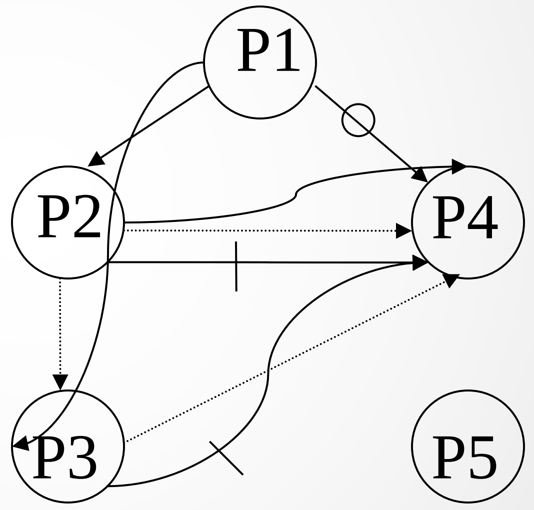
  $O_1 \cap O_2 = \emptyset$

Bernstein's Conditions - 2

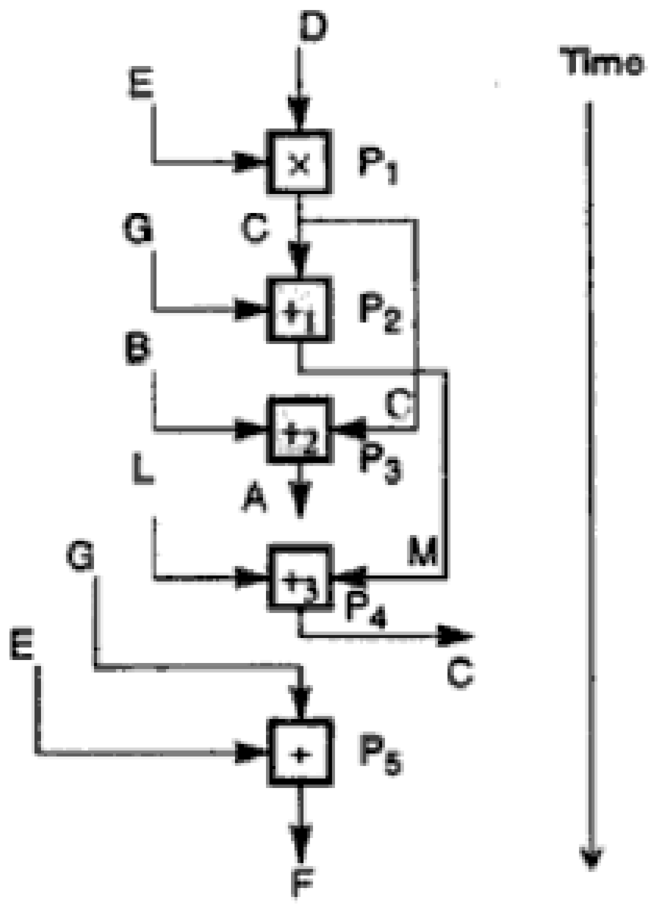
- In terms of data dependencies, Bernstein's conditions imply that two processes can execute in parallel if they are flow-independent, anti-independent, and output-independent.
- The parallelism relation || is commutative ($P_i \,||\, P_j$ implies $P_j \,||\, P_i$), but not transitive ($P_i \,||\, P_j$ and $P_j \,||\, P_k$ does not imply $P_i \,||\, P_k$) . Therefore, || is not an equivalence relation.
- Intersection of the input sets is allowed.

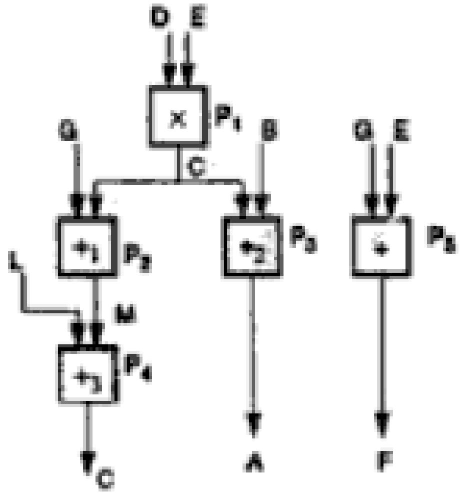
Utilizing Bernstein's conditions

P_1 :	C = D x E
P_2 :	M = G + C
P_3 :	A = B + C
P_4 :	C = L + M
P_5 :	F = G / E



Utilizing Bernstein’s conditions





WEEK 5
SLIDES 66-77

Hardware parallelism

- A function of cost and performance tradeoffs
- Displays the resource utilization patterns of simultaneously executable operations
- Denote the number of instruction issues per machine cycle: *k-issue* processor
- A multiprocessor system with n k-issue processors should be able to handle a maximum number of nk threads of instructions simultaneously

Software parallelism

- Defined by the control and data dependence of programs
- A function of algorithm, programming style, and compiler organization
- The program flow graph displays the patterns of simultaneously executable operations

Mismatch between software and hardware parallelism - 1

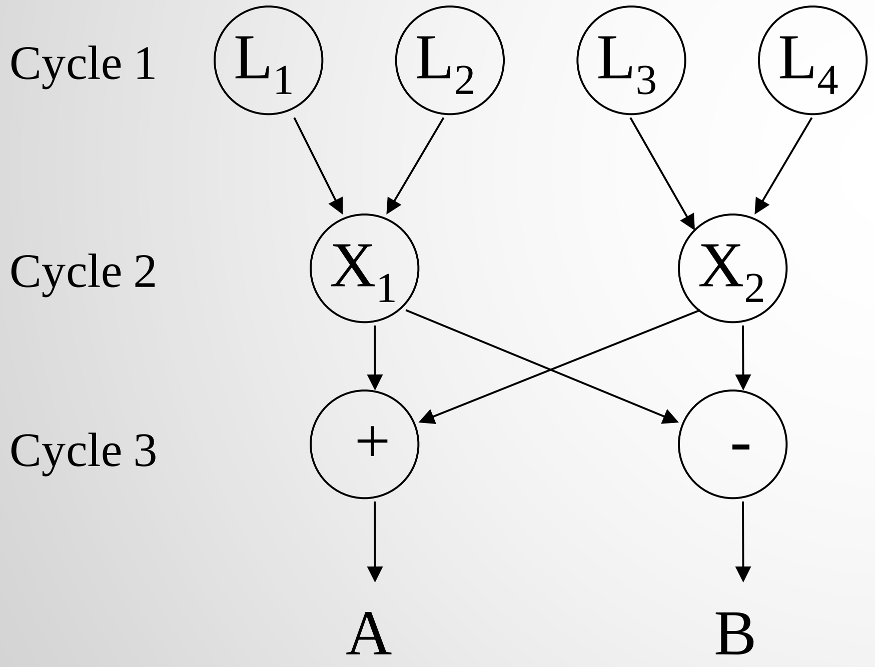


Fig. : Software parallelism

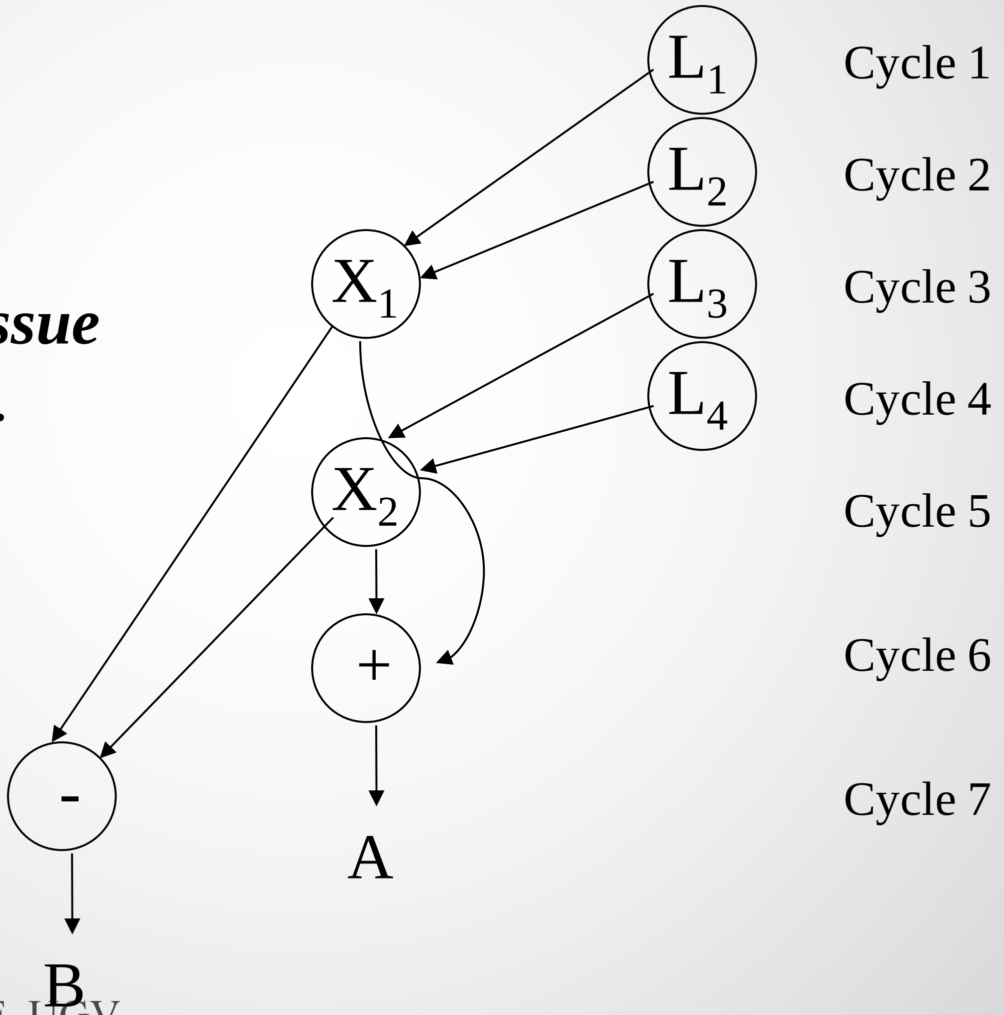
Maximum software parallelism (L=load, X/+/- = arithmetic)

8 instructions(4 loads and 4 arithmetic operations)

Parallelism varies from 4 to 2 in three cycles

Mismatch between software and hardware parallelism - 2

Same problem, but considering the parallelism on a ***two-issue superscalar processor****.*



Mismatch between software and hardware parallelism - 3

Same problem, with two single-issue processors

● *= inserted for synchronization*

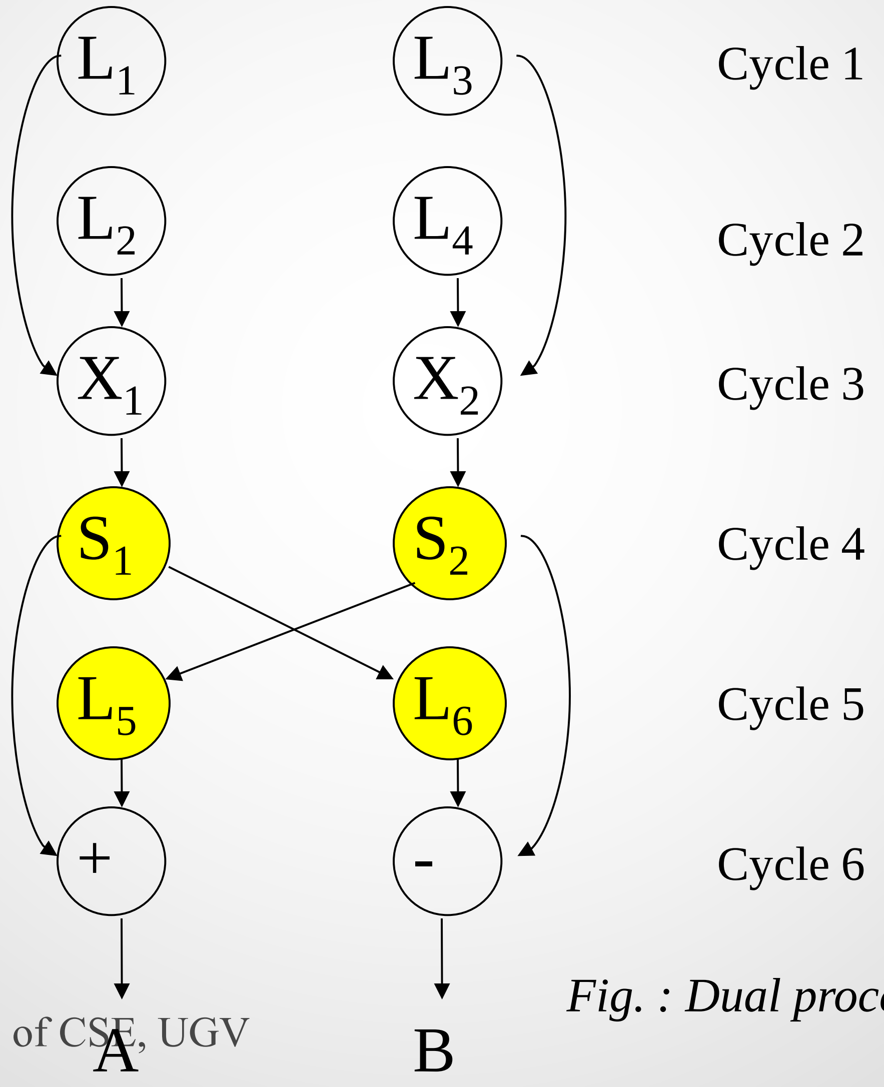


Fig. : Dual processor execution

Software parallelism

- **Control parallelism** – allows two or more operations to be performed concurrently
  - Pipelining, multiple functional units
- **Data parallelism** – almost the same operation is performed over many data elements by many processors concurrently
  - Code is easier to write and debug

Types of Software Parallelism

- **Control Parallelism** – two or more operations can be performed simultaneously. This can be detected by a compiler, or a programmer can explicitly indicate control parallelism by using special language constructs or dividing a program into multiple processes.
- **Data parallelism** – multiple data elements have the same operations applied to them at the same time. This offers the highest potential for concurrency (in SIMD and MIMD modes). Synchronization in SIMD machines handled by hardware.

Solving the Mismatch Problems

- Develop compilation support
- Redesign hardware for more efficient exploitation by compilers
- Use large register files and sustained instruction pipelining.
- Have the compiler fill the branch and load delay slots in code generated for RISC processors.

The Role of Compilers

- Compilers used to exploit hardware features to improve performance.
- Interaction between compiler and architecture design is a necessity in modern computer development.
- It is not necessarily the case that more software parallelism will improve performance in conventional scalar processors.
- The hardware and compiler should be designed at the same time.

Program Partitioning & Scheduling

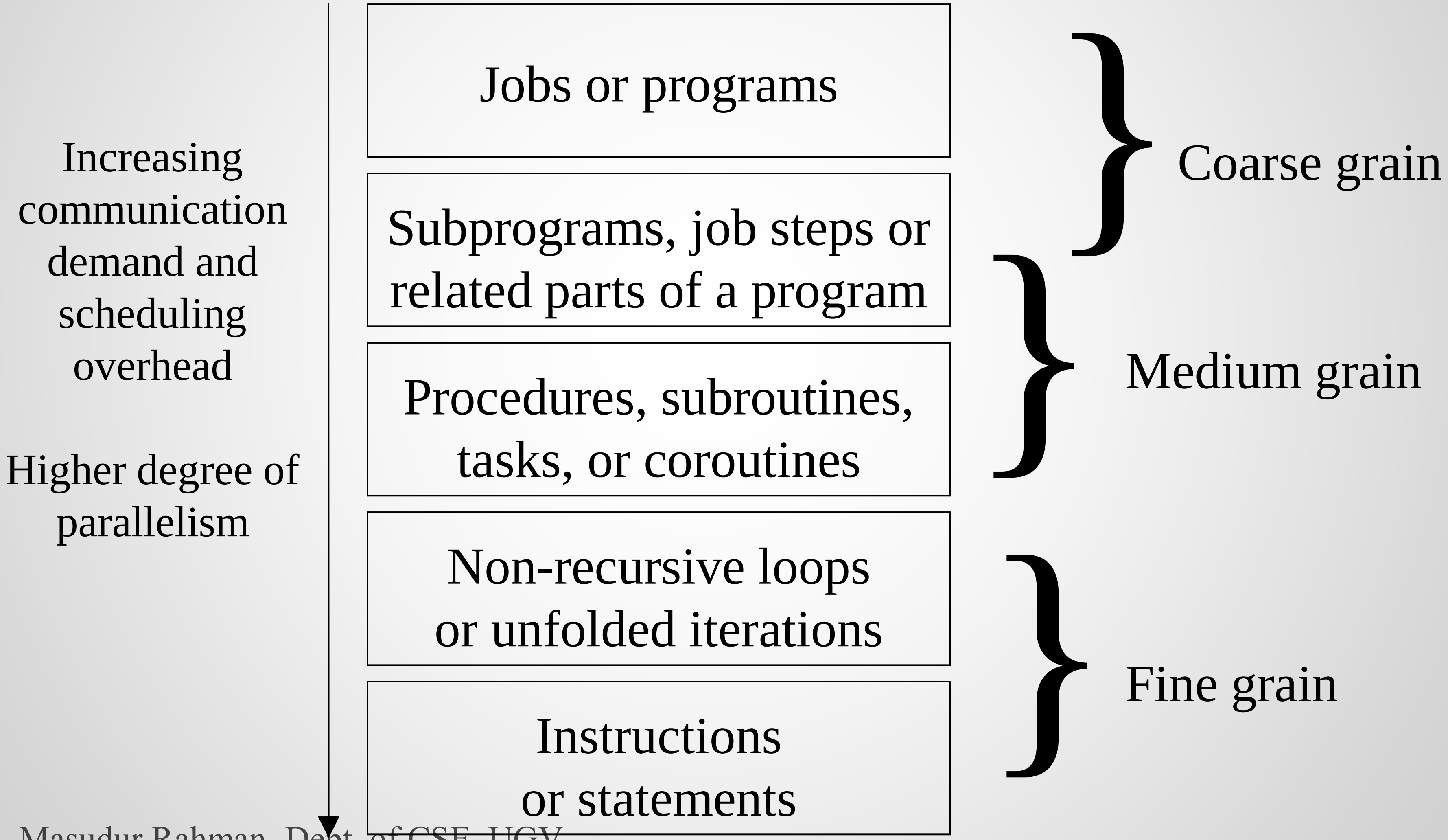
- The size of the parts or pieces of a program that can be considered for parallel execution can vary.
- The sizes are roughly classified using the term “granule size,” or simply “granularity.”
- The simplest measure, for example, is the number of instructions in a program part.
- Grain sizes are usually described as *fine*, *medium* or *coarse*, depending on the level of parallelism involved.

Latency

- *Latency* is the time required for communication between different subsystems in a computer.
- *Memory latency*, for example, is the time required by a processor to access memory.
- *Synchronization latency* is the time required for two processes to synchronize their execution.
- Computational granularity and communication latency are closely related.

WEEK 6
SLIDES 78-98

Levels of Parallelism


Increasing communication demand and scheduling overhead
Higher degree of parallelism
Jobs or programs
Subprograms, job steps or related parts of a program
Procedures, subroutines, tasks, or coroutines
Non-recursive loops or unfolded iterations
Instructions or statements
Coarse grain
Medium grain
Fine grain

Instruction Level Parallelism

- This fine-grained, or smallest granularity level typically involves less than 20 instructions per grain. The number of candidates for parallel execution varies from 2 to thousands, with about five instructions or statements (on the average) being the average level of parallelism.
- Advantages:
  - There are usually many candidates for parallel execution
  - Compilers can usually do a reasonable job of finding this parallelism

Loop-level Parallelism

- Typical loop has less than 500 instructions.
- If a loop operation is independent between iterations, it can be handled by a pipeline, or by a SIMD machine.
- Most optimized program construct to execute on a parallel or vector machine
- Some loops (e.g. recursive) are difficult to handle.
- Loop-level parallelism is still considered fine grain computation.

Procedure-level Parallelism

- Medium-sized grain; usually less than 2000 instructions.
- Detection of parallelism is more difficult than with smaller grains; interprocedural dependence analysis is difficult and history-sensitive.
- Communication requirement less than instruction-level
- SPMD (single procedure multiple data) is a special case
- Multitasking belongs to this level.

Subprogram-level Parallelism

- Job step level; grain typically has thousands of instructions; medium- or coarse-grain level.
- Job steps can overlap across different jobs.
- Multiprograming conducted at this level
- No compilers available to exploit medium- or coarse-grain parallelism at present.

Job or Program-Level Parallelism

- Corresponds to execution of essentially independent jobs or programs on a parallel computer.
- This is practical for a machine with a small number of powerful processors, but impractical for a machine with a large number of simple processors (since each processor would take too long to process a single job).

Communication Latency

- Balancing granularity and latency can yield better performance.
- Various latencies attributed to machine architecture, technology, and communication patterns used.
- Latency imposes a limiting factor on machine scalability. Ex. Memory latency increases as memory capacity increases, limiting the amount of memory that can be used with a given tolerance for communication latency.

Interprocessor Communication Latency

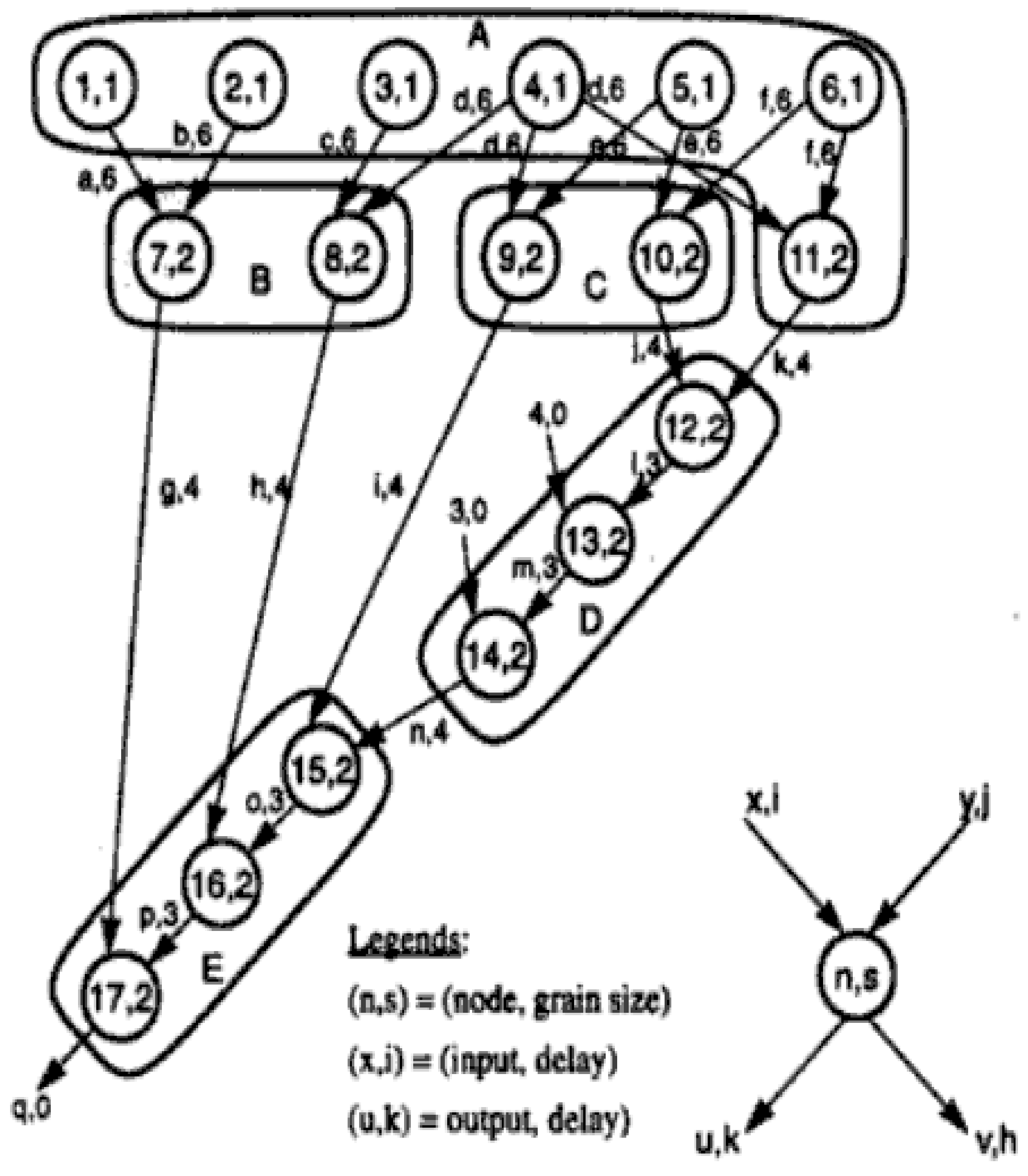
- Needs to be minimized by system designer
- Affected by signal delays and communication patterns
- Ex. n communicating tasks may require $n(n - 1)/2$ communication links, and the complexity grows quadratically, effectively limiting the number of processors in the system.

Grain Packing and Scheduling

- Two questions:
  - How can I partition a program into parallel “pieces” to yield the shortest execution time?
  - What is the optimal size of parallel grains?
- There is an obvious tradeoff between the **time spent scheduling** and **synchronizing parallel grains** and the **speedup** obtained by parallel execution.
- Solution is both **problem-dependent** and **machine-dependent**.
- Goal is to produce a short schedule for fast execution of subdivided program modules.
- One approach to the problem is called “grain packing”

Program Graphs and Packing

- A program graph is similar to a dependence graph
  - Nodes = { (n,s) }, where n = node name, s = size (larger s = larger grain size).
  - Edges = { (v,d) }, where v = variable being "communicated," and d = communication delay.
- Packing two (or more) nodes produces a node with a larger grain size and possibly more edges to other nodes.
- Packing is done to eliminate **unnecessary communication delays** or **reduce overall scheduling overhead**.



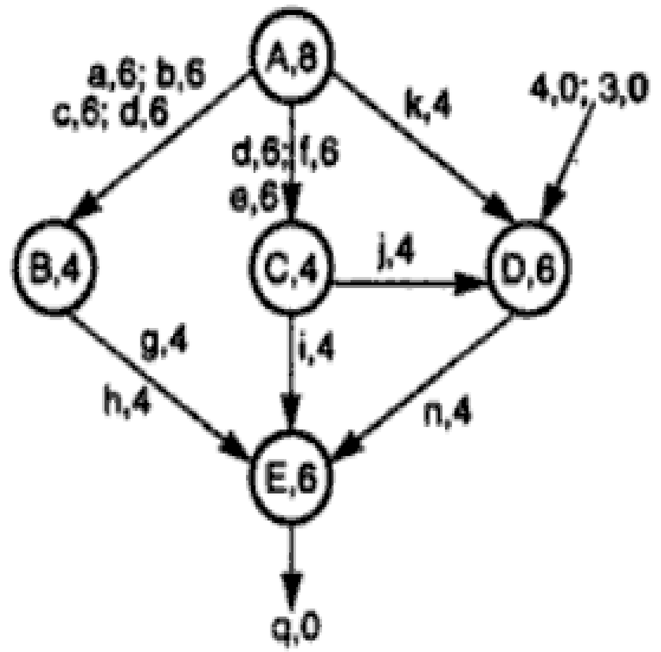
Legends:

(n,s) = (node, grain size)

(x,i) = (input, delay)

(u,k) = output, delay)

(a) Fine-grain program graph before packing

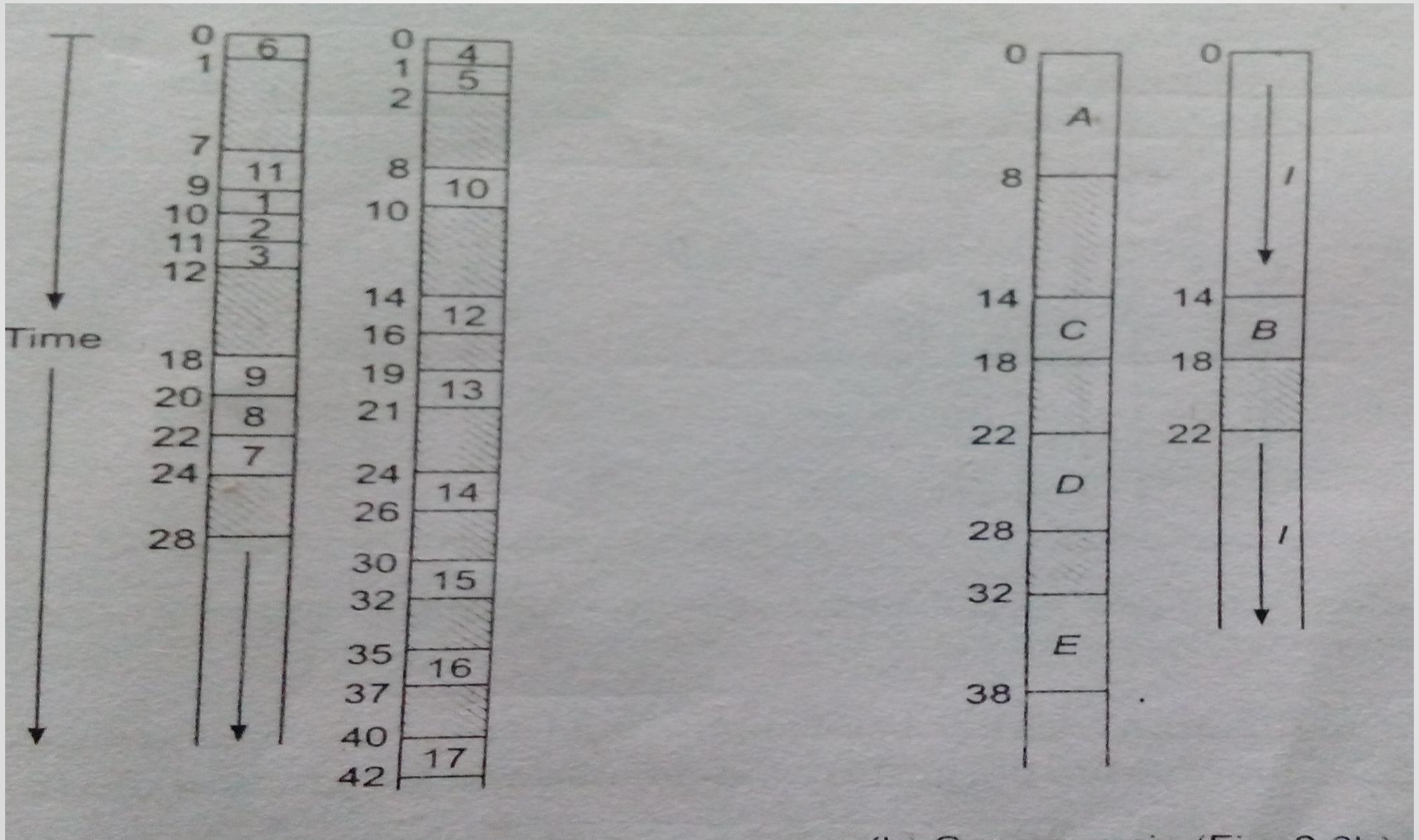


(b) Coarse-grain program graph after packing

Scheduling

- A schedule is a mapping of nodes to processors and start times such that communication delay requirements are observed, and no two nodes are executing on the same processor at the same time.
- Some general scheduling goals
  - Schedule all fine-grain activities in a node to the same processor to minimize communication delays.
  - Select grain sizes for packing to achieve better schedules for a particular parallel machine.

SCHEDULING (CONT)



(a) Fine grain (Fig. 2.6a) (b) Coarse grain (Fig. 2.6b)

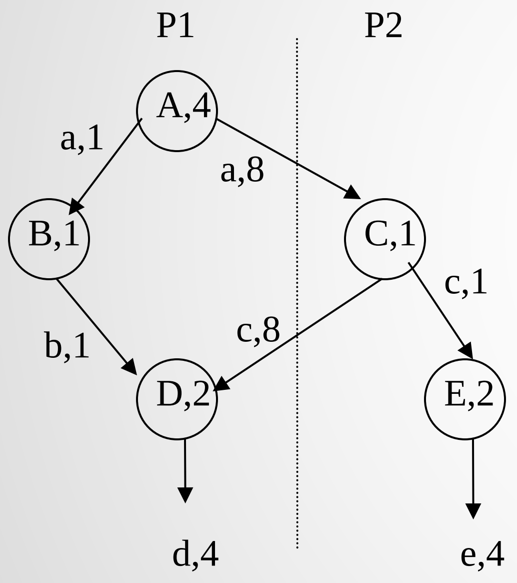
Static multiprocessor scheduling

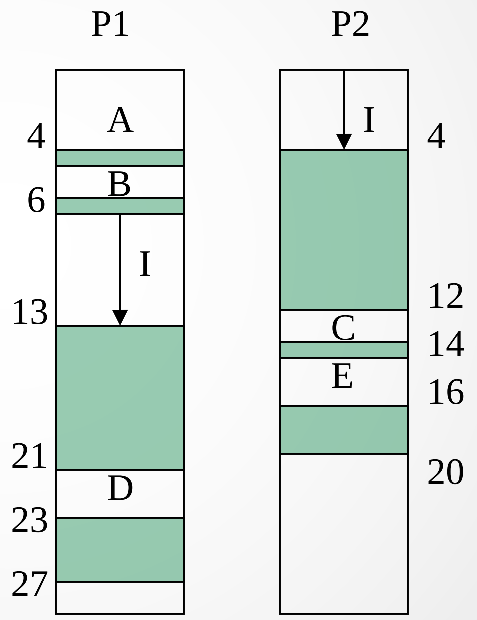
- Grain packing may not be optimal
- Dynamic multiprocessor scheduling is an NP-hard problem
- Node duplication is a static scheme for multiprocessor scheduling

Node duplication

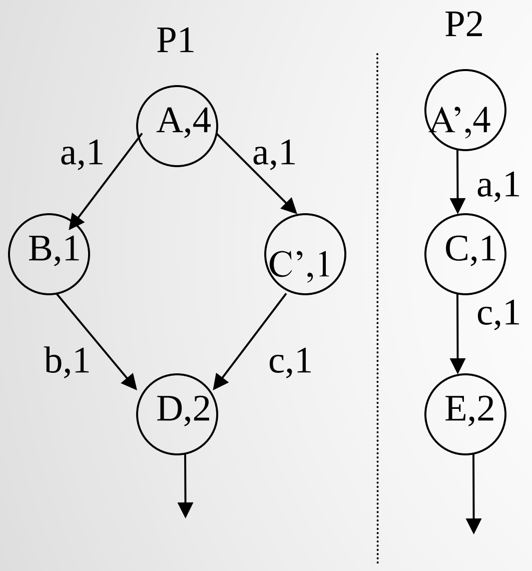
- Duplicate some nodes to eliminate idle time and reduce communication delays
- Grain packing and node duplication are often used jointly to determine the best grain size and corresponding schedule

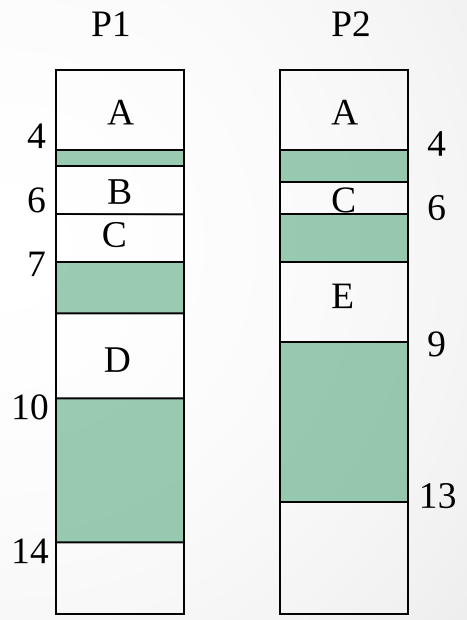
Schedule without node duplication





Schedule with node duplication





Grain determination and scheduling optimization

Step 1: Construct a fine-grain program graph

Step 2: Schedule the fine-grain computation

Step 3: Grain packing to produce coarse grains

Step 4: Generate a parallel schedule based on the packed graph

System Interconnect Architectures

- Direct networks for static connections
- Indirect networks for dynamic connections
- Networks are used for
  - internal connections in a centralized system among
    - processors
    - memory modules
    - I/O disk arrays
  - distributed networking of multicomputer nodes

Goals and Analysis

- The goals of an interconnection network are to provide
  - low-latency
  - high data transfer rate
  - wide communication bandwidth
- Analysis includes
  - latency
  - bisection bandwidth
  - data-routing functions
  - scalability of parallel architecture

WEEK 7
SLIDES 99-107

Network Properties and Routing

- Static networks: point-to-point direct connections that will not change during program execution
- Dynamic networks:
  - switched channels dynamically configured to match user program communication demands
  - include buses, crossbar switches, and multistage networks
- Both network types also used for inter-PE data routing in SIMD computers

Network Parameters

- Network size: The number of nodes (links or channels) in the graph used to represent the network
- Node Degree d: The number of edges incident to a node. In the case of unidirectional channels, the number of channels **into a node is the in degree** and that **out of a node is the out degree**. Then the node degree is the sum of the two.
- Network Diameter D: The maximum shortest path between any two nodes

Network Parameters (cont.)

- Bisection Width:
  - **Channel bisection width b**: The minimum number of edges (channels) along the cut that divides the network in two equal halves
  - Each channel has w bit wires
  - **Wire bisection width**: B=b*w; B is the wiring density of the network. It provides a good indicator of the max communication bandwidth along the bisection of the network

Terminology - 1

- Network usually represented by a graph with a finite number of nodes linked by directed or undirected edges.
- Number of nodes in graph = *network size* .
- Number of edges (links or channels) incident on a node = *node degree d* (also note in and out degrees when edges are directed). Node degree reflects number of I/O ports associated with a node, and should ideally be small and constant.
- *Diameter D* of a network is the maximum shortest path between any two nodes, measured by the number of links traversed; this should be as small as possible (from a communication point of view).

Terminology - 2

- *Channel bisection width b* = minimum number of edges cut to split a network into two parts each having the same number of nodes. Since each channel has *w* bit wires, the *wire bisection width B* = *bw*. Bisection width provides good indication of maximum communication bandwidth along the bisection of a network, and all other cross sections should be bounded by the bisection width.
- *Wire* (or *channel*) *length* = length (e.g. weight) of edges between nodes.
- Network is symmetric if the topology is the same looking from any node; these are easier to implement or to program.
- Other useful characterizing properties: homogeneous nodes? buffered channels? nodes are switches?

Data Routing Functions

Data-routing network used for inter-PE data exchange. This network can be static (i.e: hypercube) or dynamic (i.e: multistage network)

Commonly data-routing functions includes:

- Shifting
- Rotating
- Permutation (one to one)
- Broadcast (one to all)
- Multicast (many to many)
- Personalized broadcast (one to many)
- Shuffle
- Exchange
- Etc.

Hypercube Routing Functions

- If the vertices of a *n*-dimensional cube are labeled with *n*-bit numbers so that only one bit differs between each pair of adjacent vertices, then *n* routing functions are defined by the bits in the node (vertex) address.
- For example, with a 3-dimensional cube, we can easily identify routing functions that exchange data between nodes with addresses that differ in the least significant, most significant, or middle bit.
- Figure: 2.15

Factors Affecting Network Performance

- **Functionality** – how the network supports data routing, interrupt handling, synchronization, request/message combining, and coherence
- **Network latency** – worst-case time for a unit message to be transferred
- **Bandwidth** – maximum data rate
- **Hardware complexity** – implementation costs for wire, logic, switches, connectors, etc.
- **Scalability** – how easily does the scheme adapt to an increasing number of processors, memories, etc.?

WEEK 8

SLIDES 108-127

Static Networks

- Linear Array
- Ring and Chordal Ring
- Barrel Shifter
- Tree and Star
- Fat Tree
- Mesh and Torus

Static Networks – Linear Array

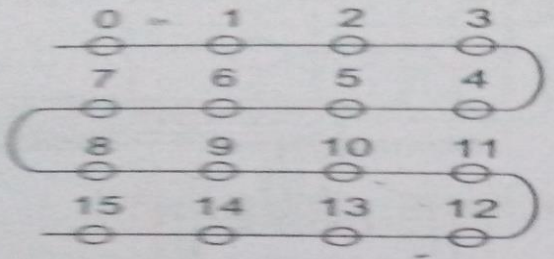
- N nodes connected by *n*-1 links (not a bus); segments between different pairs of nodes can be used in parallel.
- Internal nodes have degree 2; end nodes have degree 1.
- Diameter = n-1
- Bisection = 1
- For small n, this is economical, but for large n, it is obviously inappropriate.

Static Networks – Ring, Chordal Ring

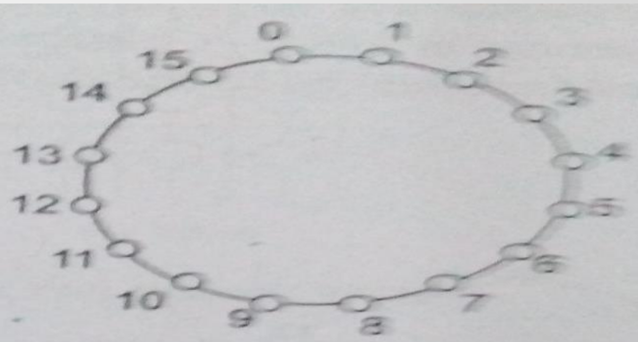
- Like a linear array, but the two end nodes are connected by an *n* th link; the ring can be uni- or bi-directional. Diameter is $\lfloor n/2 \rfloor$ for a bidirectional ring, or n for a unidirectional ring.
- By adding additional links (e.g. "chords" in a circle), the node degree is increased, and we obtain a chordal ring. This reduces the network diameter.
- In the limit, we obtain a fully-connected network, with a node degree of *n* -1 and a diameter of 1.

Static Networks – Barrel Shifter

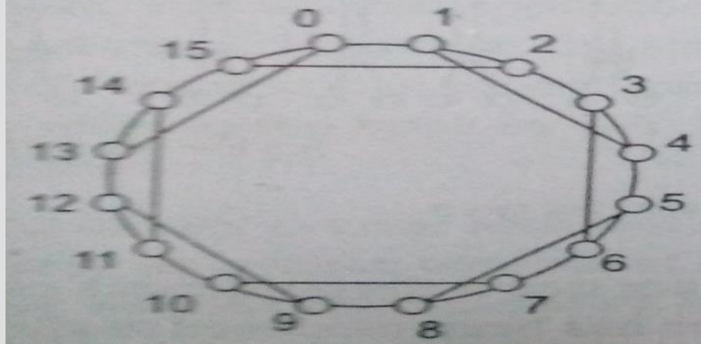
- Like a ring, but with additional links between all pairs of nodes that have a distance equal to a power of 2.
- With a network of size $N = 2^n$, each node has degree $d = 2n - 1$, and the network has diameter $D = n / 2$.
- Barrel shifter connectivity is greater than any chordal ring of lower node degree.
- Barrel shifter much less complex than fully-interconnected network.



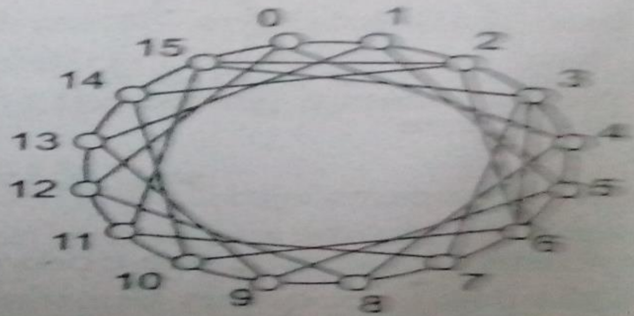
(a) Linear array



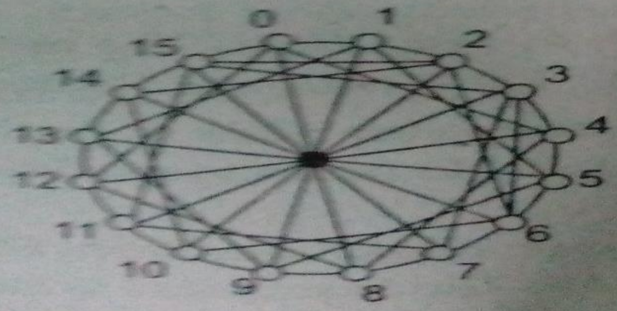
(b) Ring



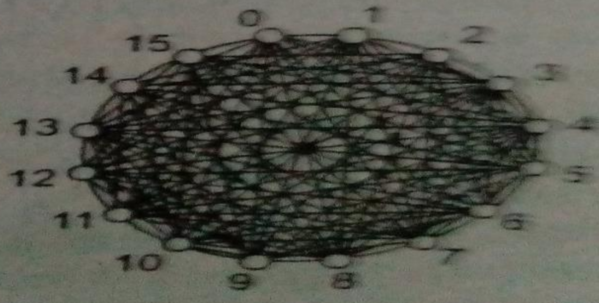
(c) Chordal ring of degree 3



(d) Chordal ring of degree 4 (same as Illiac mesh)



(e) Barrel shifter



(f) Completely connected

Static Networks – Tree and Star

- A k-level completely balanced binary tree will have $N = 2^k - 1$ nodes, with maximum node degree of 3 and network diameter is $2(k - 1)$.
- The balanced binary tree is scalable, since it has a constant maximum node degree.
- A star is a two-level tree with a node degree $d = N - 1$ and a constant diameter of 2.

Static Networks – Fat Tree

- A fat tree is a tree in which the number of edges between nodes increases closer to the root (similar to the way the thickness of limbs increases in a real tree as we get closer to the root).
- The edges represent communication channels (“wires”), and since communication traffic increases as the root is approached, it seems logical to increase the number of channels there.

Static Networks – Mesh and Torus

- *Pure mesh* – $N = n^k$ nodes with links between each adjacent pair of nodes in a row or column (or higher degree). This is not a symmetric network; interior node degree $d = 2k$, diameter $= k(n - 1)$.
- *Illiac mesh* (used in Illiac IV computer) – wraparound is allowed, thus reducing the network diameter to about half that of the equivalent pure mesh.
- A *torus* has ring connections in each dimension, and is symmetric. An $n \times n$ binary torus has node degree of 4 and a diameter of $2 \times \lfloor n / 2 \rfloor$.

Static Networks – Systolic Array

- A systolic array is an arrangement of processing elements and communication links designed specifically to match the computation and communication requirements of a specific algorithm (or class of algorithms).
- This specialized character may yield better performance than more generalized structures, but also makes them more expensive, and more difficult to program.

Network Throughput

- *Network throughput* – number of messages a network can handle in a unit time interval.
- One way to estimate is to calculate the maximum number of messages that can be present in a network at any instant (its *capacity*); throughput usually is some fraction of its capacity.
- A *hot spot* is a pair of nodes that accounts for a disproportionately large portion of the total network traffic (possibly causing congestion).
- *Hot spot throughput* is maximum rate at which messages can be sent between two specific nodes.

Dynamic Connection Networks

- Dynamic connection networks can implement all communication patterns based on program demands.
- In increasing order of cost and performance, these include
  - bus systems
  - multistage interconnection networks
  - crossbar switch networks
- Price can be attributed to the cost of wires, switches, arbiters, and connectors.
- Performance is indicated by network bandwidth, data transfer rate, network latency, and communication patterns supported.

Dynamic Networks – Bus Systems

- A *bus* system (*contention bus*, *time-sharing bus*) has
  - a collection of wires and connectors
  - multiple modules (processors, memories, peripherals, etc.) which connect to the wires
  - data transactions between pairs of modules
- Bus supports only one transaction at a time.
- Bus arbitration logic must deal with conflicting requests.
- Lowest cost and bandwidth of all dynamic schemes.
- Many bus standards are available.

Dynamic Networks – Switch Modules

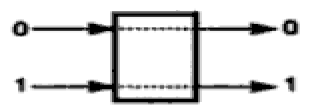
- An $a \times b$ switch module has a inputs and b outputs. A binary switch has $a = b = 2$.
- It is not necessary for $a = b$, but usually $a = b = 2^k$, for some integer k.
- In general, any input can be connected to one or more of the outputs. However, multiple inputs may not be connected to the same output.
- When only one-to-one mappings are allowed, the switch is called a *crossbar switch*.

Multistage Networks

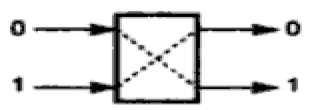
- In general, any multistage network is comprised of a collection of ***a* × *b* switch modules and fixed network modules**. The ***a* × *b*** switch modules are used to provide **variable permutation** or other **reordering of the inputs**, which are then further reordered by the fixed network modules.
- A generic multistage network consists of a **sequence alternating dynamic switches** (with relatively small values for *a* and *b)* with static networks (with larger numbers of inputs and outputs). The static networks are used to implement interstage connections (ISC).

Omega Network

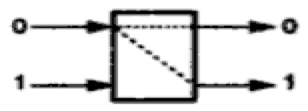
- A 2×2 switch can be configured for
  - Straight-through
  - Crossover
  - Upper broadcast (upper input to both outputs)
  - Lower broadcast (lower input to both outputs)
  - (No output is a somewhat vacuous possibility as well)
- With four stages of eight 2×2 switches, and a static perfect shuffle for each of the four ISCs, a 16 by 16 Omega network can be constructed (but not all permutations are possible).
- In general , an n-input Omega network requires $\log_2 n$ stages of 2×2 switches and $n / 2$ switch modules.



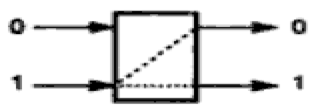
(a) Straight



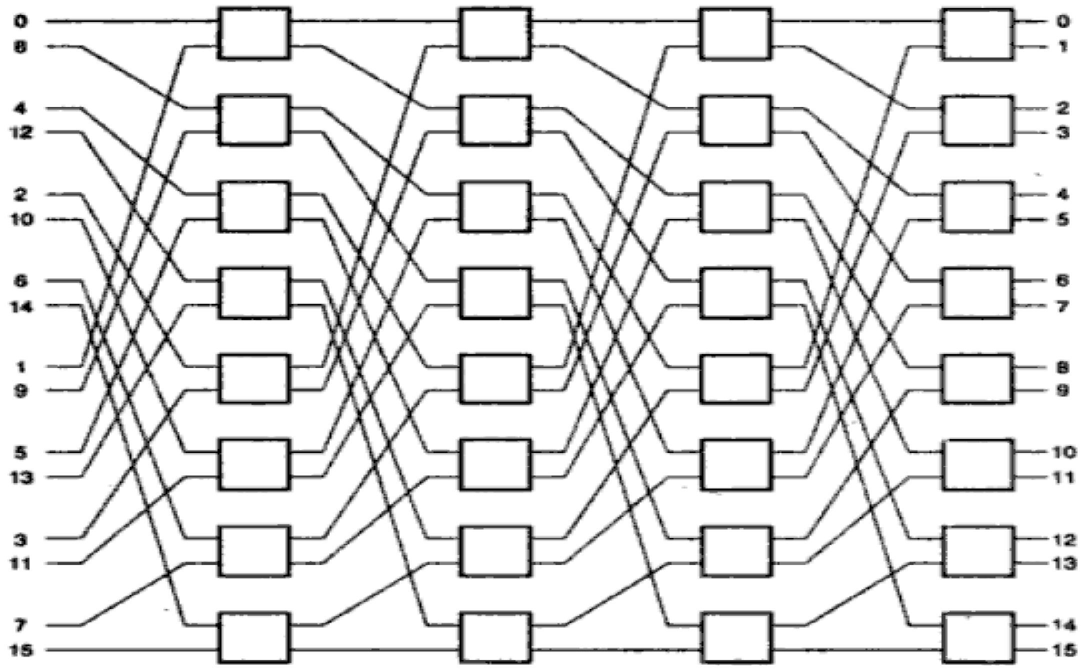
(b) Crossover



(c) Upper broadcast



(d) Lower broadcast

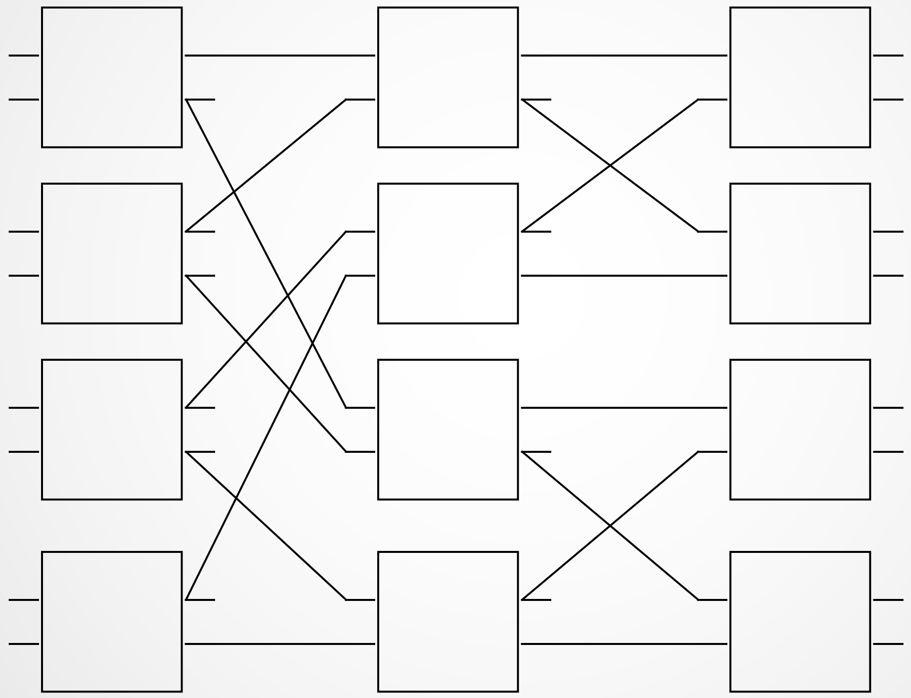


(e) A 16 × 16 Omega network

Baseline Network

- A baseline network can be shown to be topologically equivalent to other networks (including Omega), and has a simple recursive generation procedure.
- Stage k (k = 0, 1, ...) is an $m \times m$ switch block (where $m = N / 2^k$) composed entirely of 2×2 switch blocks, each having two configurations: straight through and crossover.

4×4 Baseline Network



Crossbar Networks

- A m × n crossbar network can be used to provide a constant latency connection between devices; it can be thought of as a single stage switch.
- Different types of devices can be connected, yielding different constraints on which switches can be enabled.
  - With m processors and n memories, one processor may be able to generate requests for multiple memories in sequence; thus several switches might be set in the same row.
  - For m × m interprocessor communication, each PE is connected to both an input and an output of the crossbar; only one switch in each row and column can be turned on simultaneously. Additional control processors are used to manage the crossbar itself.

WEEK 9
SLIDES 128-145

PIPELINING

OVERVIEW

- Pipelining is widely used in modern processors.
- Pipelining improves system performance in terms of throughput.
- Pipelined organization requires sophisticated compilation techniques.

BASIC CONCEPTS

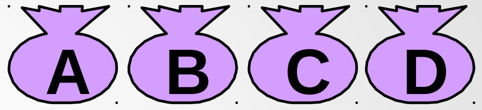
- Pipelining is an implementation technique that overlaps multiple instruction execution.
- An instruction is broken into smaller steps
- Each smaller step (pipeline stage or pipeline segment) takes a fraction of the time needed to complete the entire instruction.

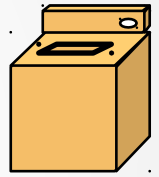
MAKING THE EXECUTION OF PROGRAMS FASTER

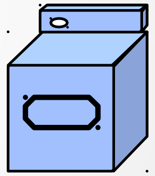
- Use faster circuit technology to build the processor and the main memory.
- Arrange the hardware so that more than one operation can be performed at the same time.
- In the latter way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

TRADITIONAL PIPELINE CONCEPT

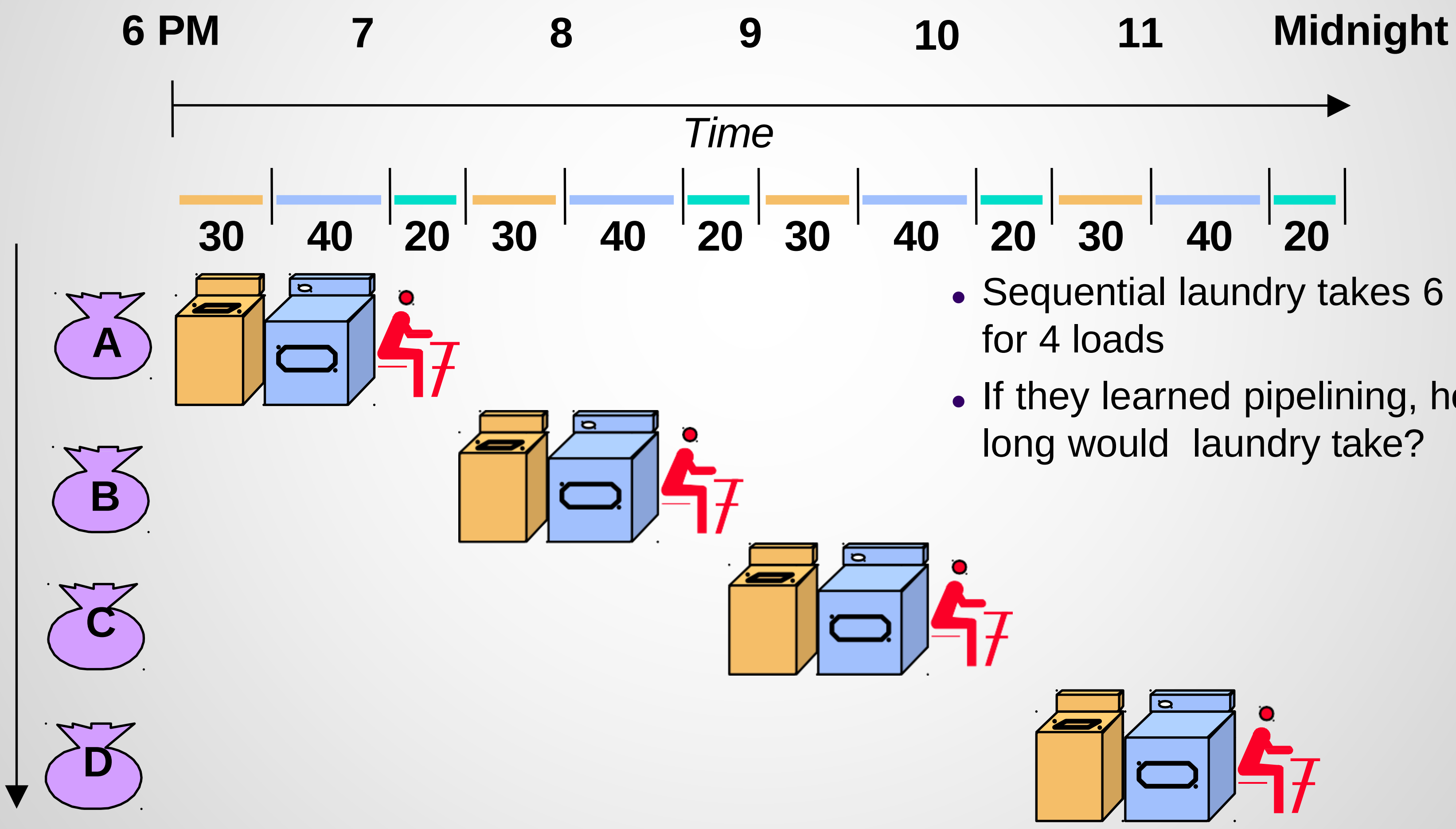
- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folder" takes 20 minutes





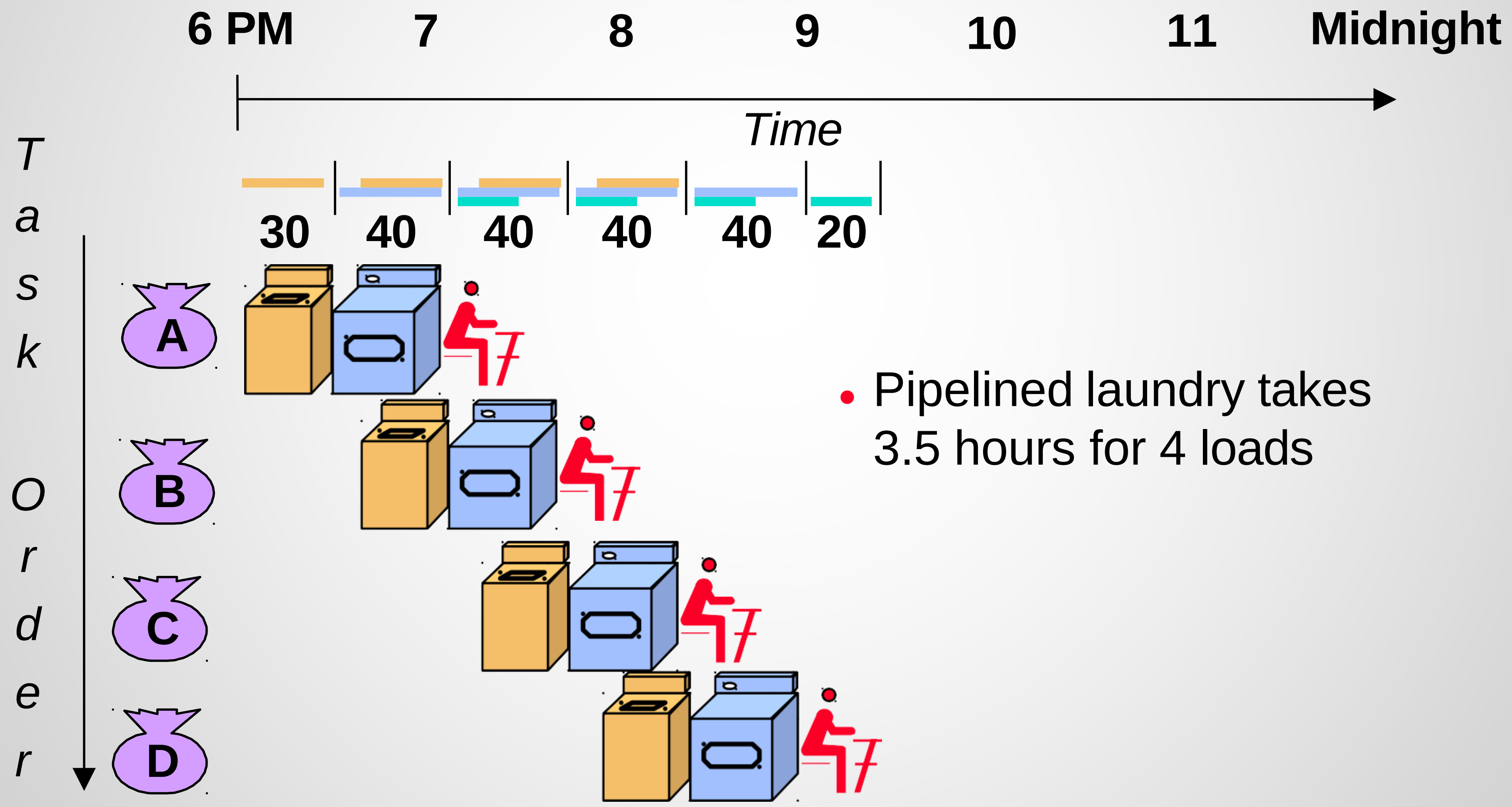


TRADITIONAL PIPELINE CONCEPT



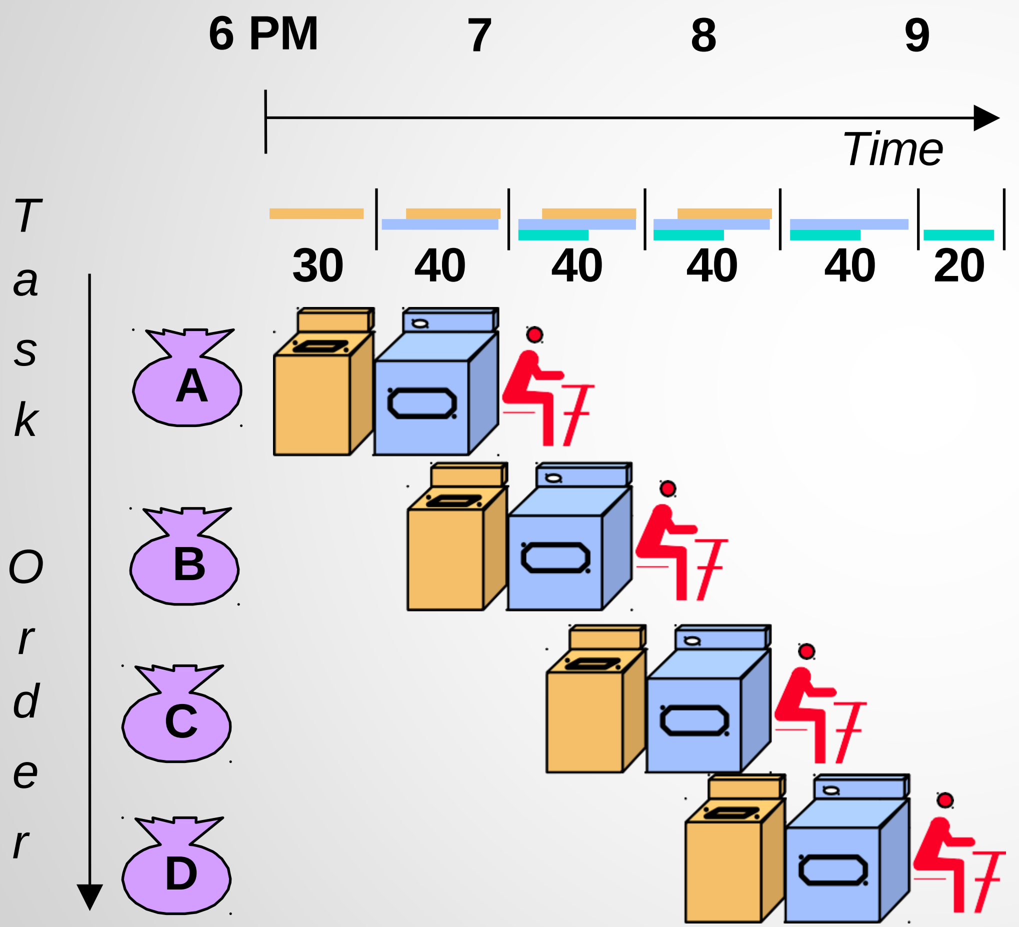
- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

TRADITIONAL PIPELINE CONCEPT



- Pipelined laundry takes 3.5 hours for 4 loads

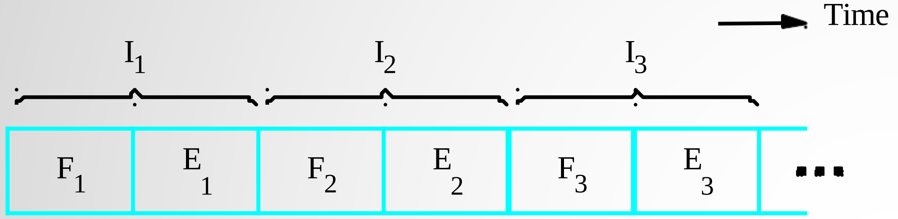
TRADITIONAL PIPELINE CONCEPT



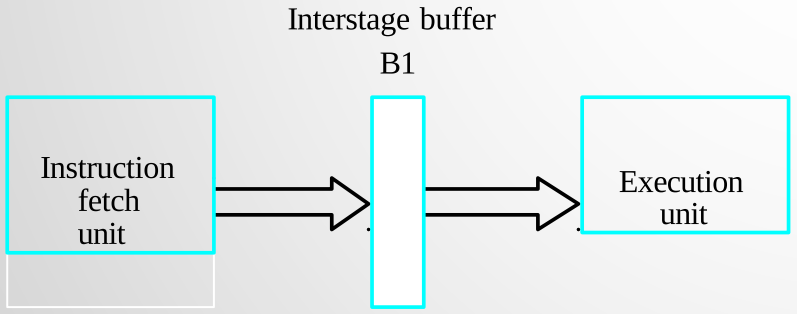
- Pipelining doesn’t help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously using different resources
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to “fill” pipeline and time to “drain” it reduces speedup
- Stall for Dependences

USE THE IDEA OF PIPELINING IN A COMPUTER

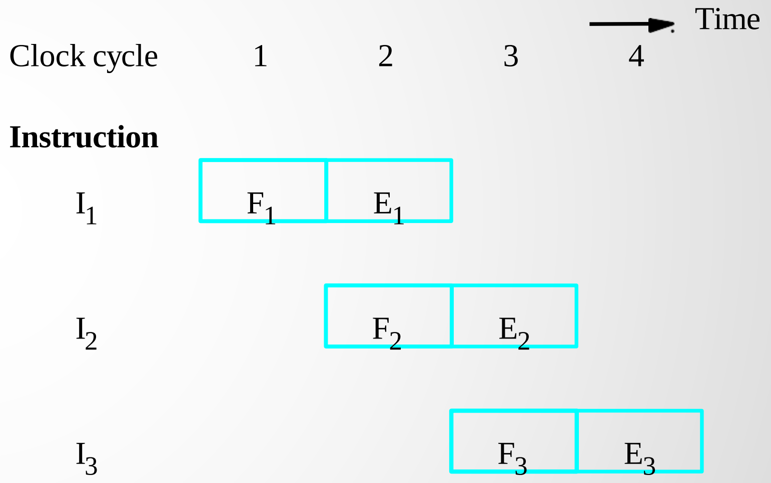
Fetch + Execution



(a) Sequential execution



(b) Hardware organization

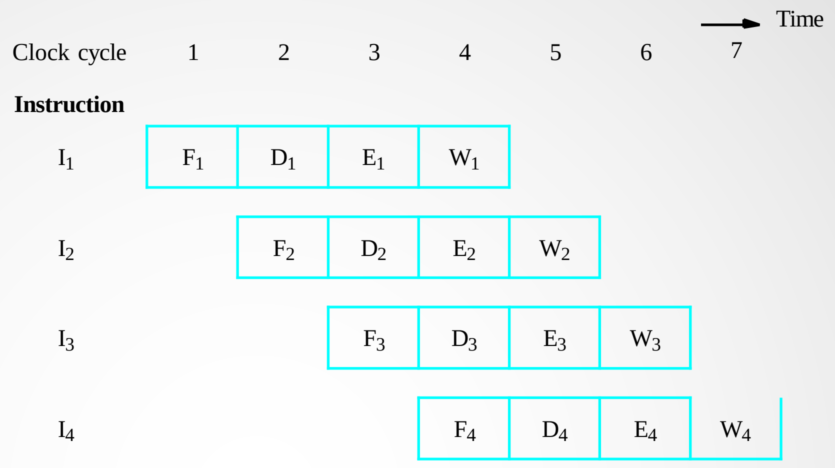


(c) Pipelined execution

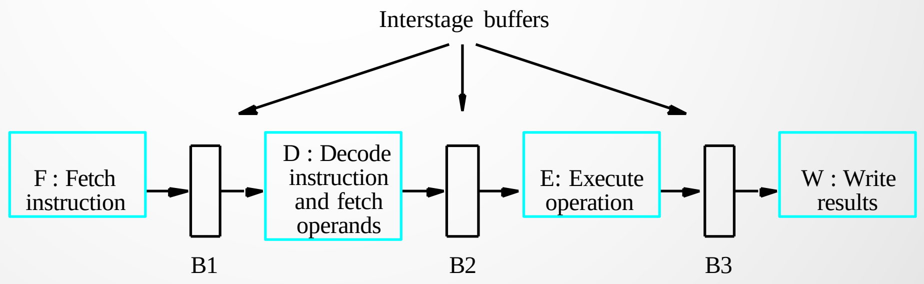
Figure 8.1. Basic idea of instruction pipelining.

USE THE IDEA OF PIPELINING IN A COMPUTER

Fetch + Decode
+ Execution + Write


Time
Clock cycle 1 2 3 4 5 6 7
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4 W_4

(a) Instruction execution divided into four steps



(b) Hardware organization

Figure 8.2. A 4 stage pipeline.

ROLE OF CACHE MEMORY

- Each pipeline stage is expected to complete in one clock cycle.
- The clock period should be long enough to let the slowest pipeline stage to complete.
- Faster stages can only wait for the slowest one to complete.
- Since main memory is very slow compared to the execution, if each instruction needs to be fetched from main memory, pipeline is almost useless.
- Fortunately, we have cache.

PIPELINE PERFORMANCE

- The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages.
- However, this increase would be achieved only if all pipeline stages require the same time to complete, and there is no interruption throughout program execution.
- Unfortunately, this is not true.

PIPELINE PERFORMANCE

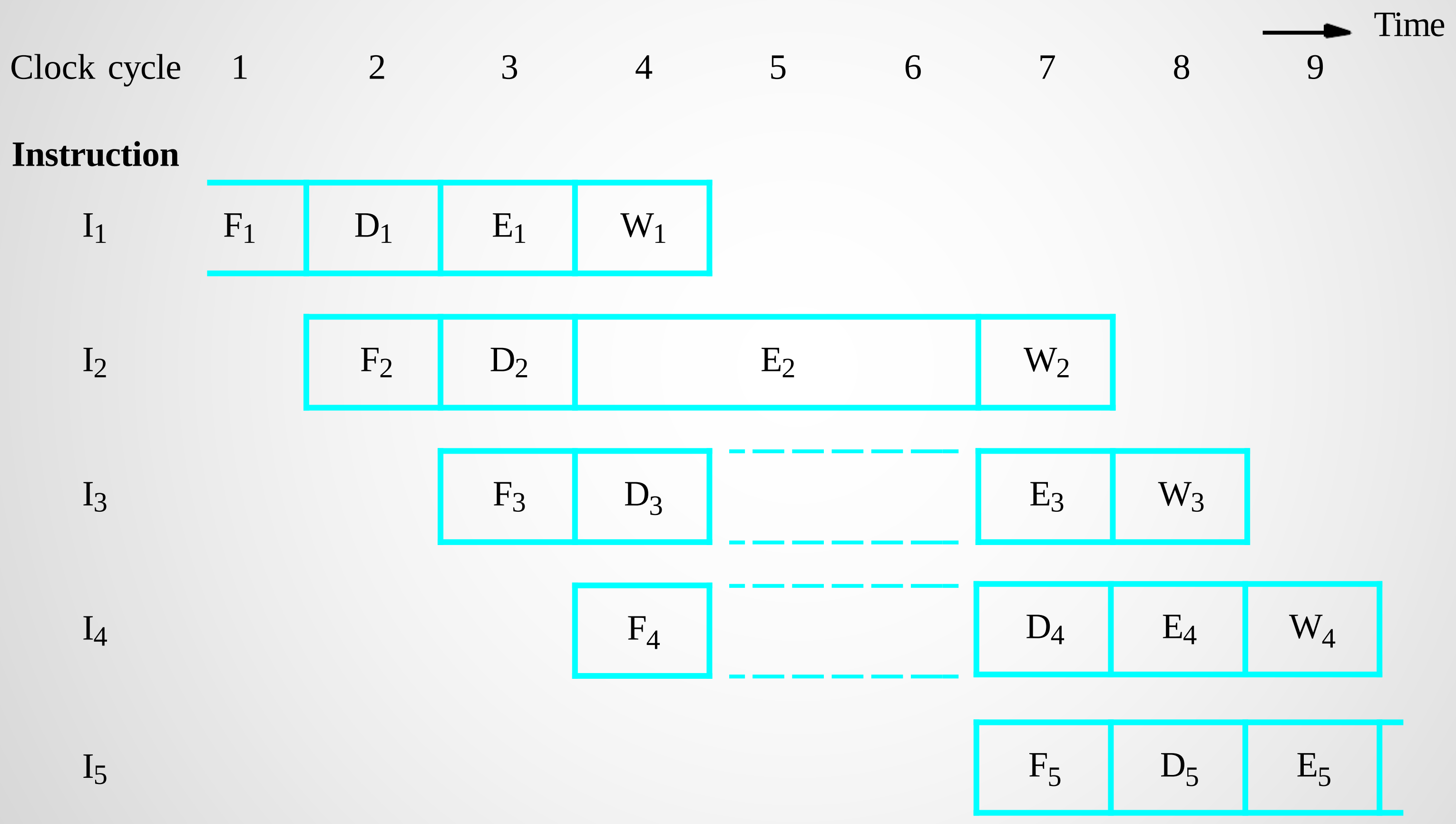


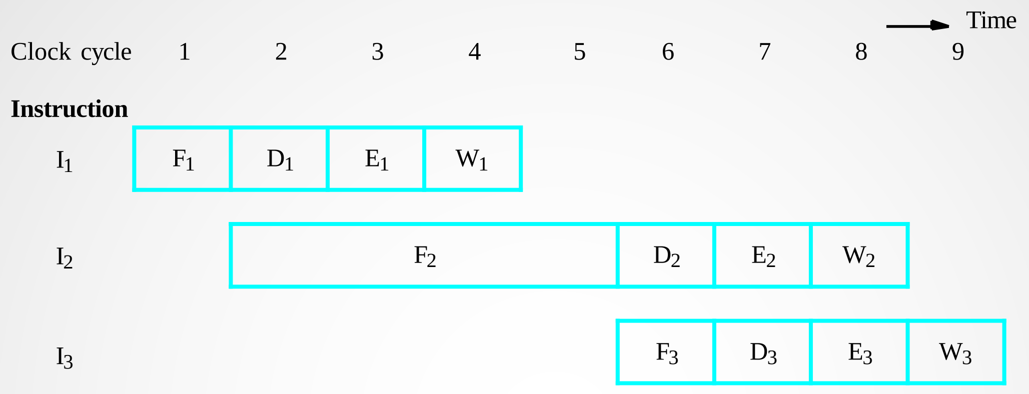
Figure 8.3. Effect of an execution operation taking more than one clock cycle.

PIPELINE PERFORMANCE

- The previous pipeline is said to have been stalled for two clock cycles.
- Any condition that causes a pipeline to stall is called a hazard.
- Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.
- Instruction (control) hazard – a delay in the availability of an instruction causes the pipeline to stall.
- Structural hazard – the situation when two instructions require the use of a given hardware resource at the same time.

PIPELINE PERFORMANCE

Instruction hazard



Time
Clock cycle 1 2 3 4 5 6 7 8 9
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3

(a) Instruction execution steps in successive clock cycles

Time →

Clock cycle	1	2	3	4	5	6	7	8	9
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

(b) Function performed by each processor stage in successive clock cycles

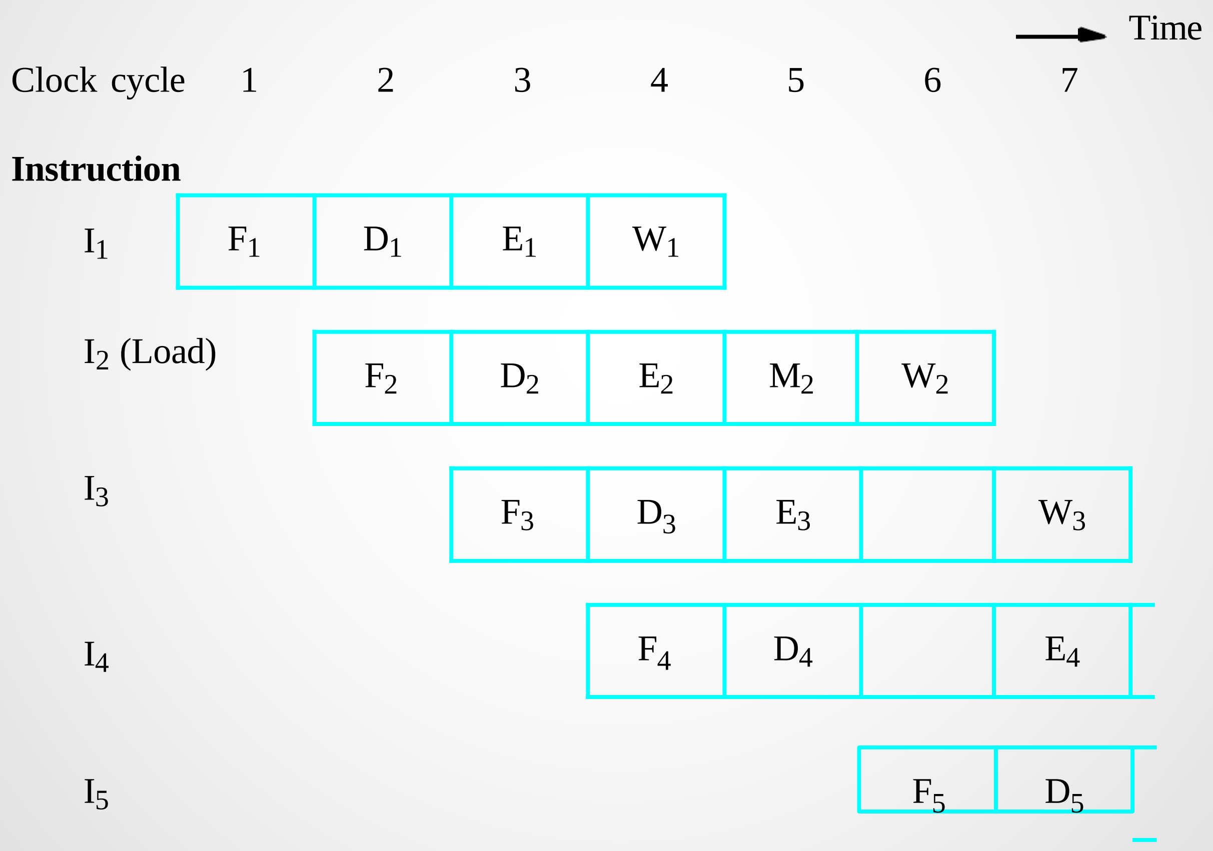
Idle periods – stalls (bubbles)

Figure 8.4. Pipeline stall caused by a cache miss in F2.

PIPELINE PERFORMANCE

Structural hazard

Load X(R1), R2



Time
Clock cycle 1 2 3 4 5 6 7
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 (Load) F_2 D_2 E_2 M_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4
I_5 F_5 D_5

Figure 8.5. Effect of a Load instruction on pipeline timing.

PIPELINE PERFORMANCE

- Again, pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases.
- Throughput is measured by the rate at which instruction execution is completed.
- Pipeline stall causes degradation in pipeline performance.
- We need to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.

WEEK 10
SLIDES 146-153

DATA HAZARDS

DATA HAZARDS

- We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially.
- Hazard occurs

  $A \leftarrow 3 + A$

  $B \leftarrow 4 \times A$
- No hazard

  $A \leftarrow 5 \times C$

  $B \leftarrow 20 + C$
- When two operations depend on each other, they must be executed sequentially in the correct order.
- Another example:

  Mul R2, R3, R4

  Add R5, R4, R6

DATA HAZARDS

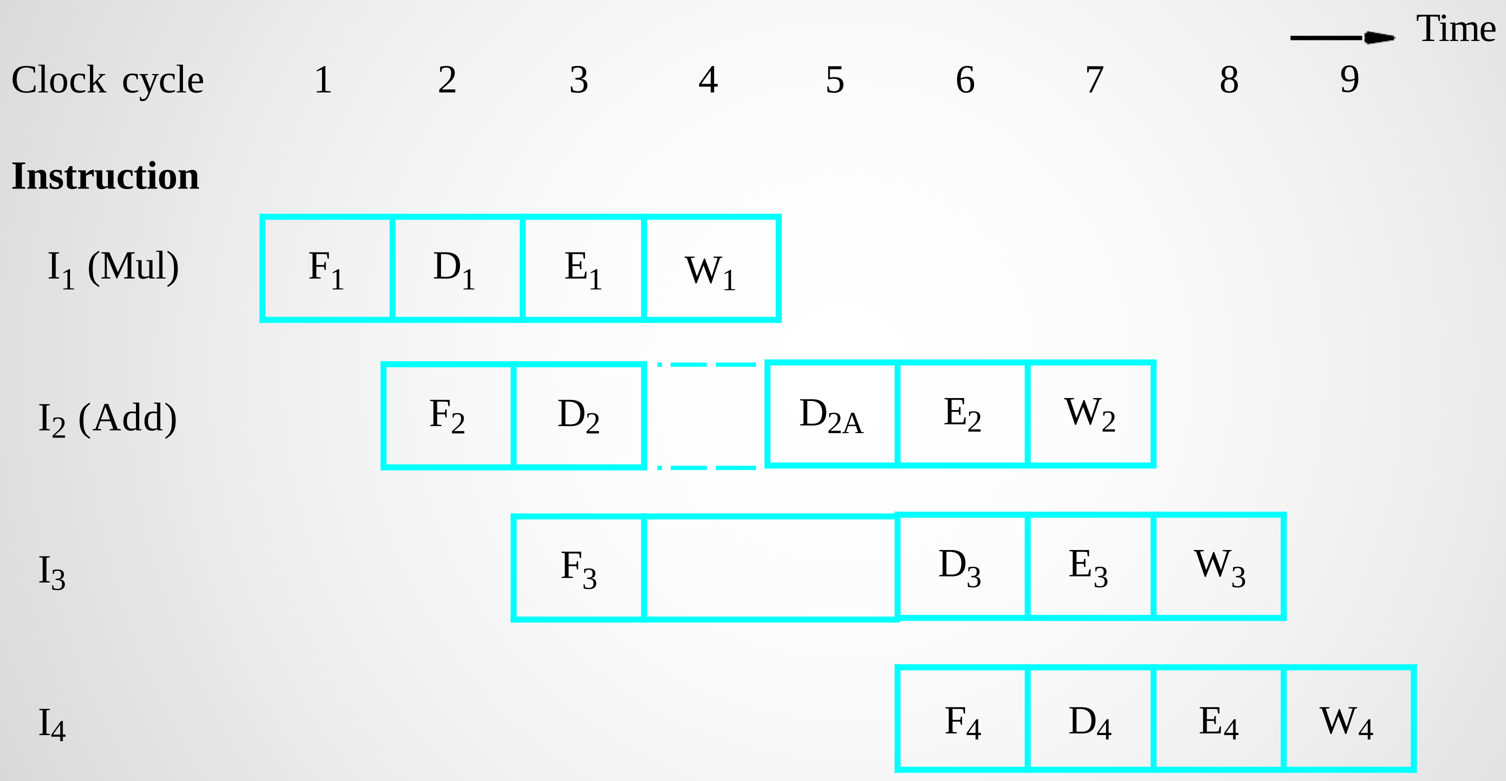
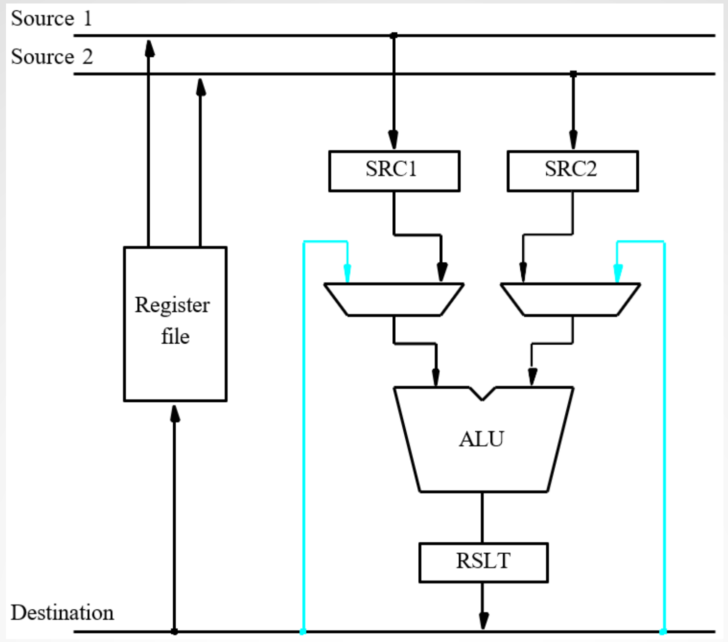


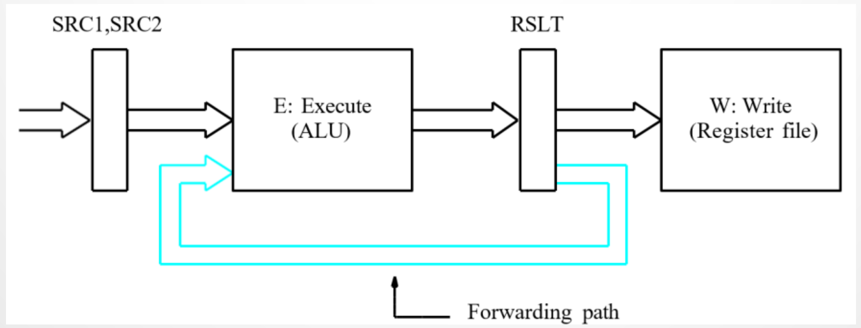
Figure 8.6. Pipeline stalled by data dependency between D_2 and W_1.

OPERAND FORWARDING

- Instead of from the register file, the second instruction can get data directly from the output of ALU after the previous instruction is completed.
- A special arrangement needs to be made to “forward” the output of ALU to the input of ALU.



(a) Datapath



(b) Position of the source and result registers in the processor pipeline

Figure 8.7. Operand forwarding in a pipelined processor.

HANDLING DATA HAZARDS IN SOFTWARE

- Let the compiler detect and handle the hazard:

```
I1: Mul  R2, R3, R4
    NOP
    NOP
I2: Add  R5, R4, R6
```

- The compiler can reorder the instructions to perform some useful work during the NOP slots.

SIDE EFFECTS

- The previous example is explicit and easily detected.
- Sometimes an instruction changes the contents of a register other than the one named as the destination.
- When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect. (Example?)
- Example: conditional code flags:

  Add R1, R3

  AddWithCarry R2, R4
- Instructions designed for execution on pipelined hardware should have few side effects.

WEEK 11
SLIDES 154-163

INSTRUCTION HAZARDS

OVERVIEW

- Whenever the **stream of instructions supplied by the instruction fetch unit** is interrupted, the pipeline stalls.
- Cache miss
- Branch

UNCONDITIONAL BRANCHES

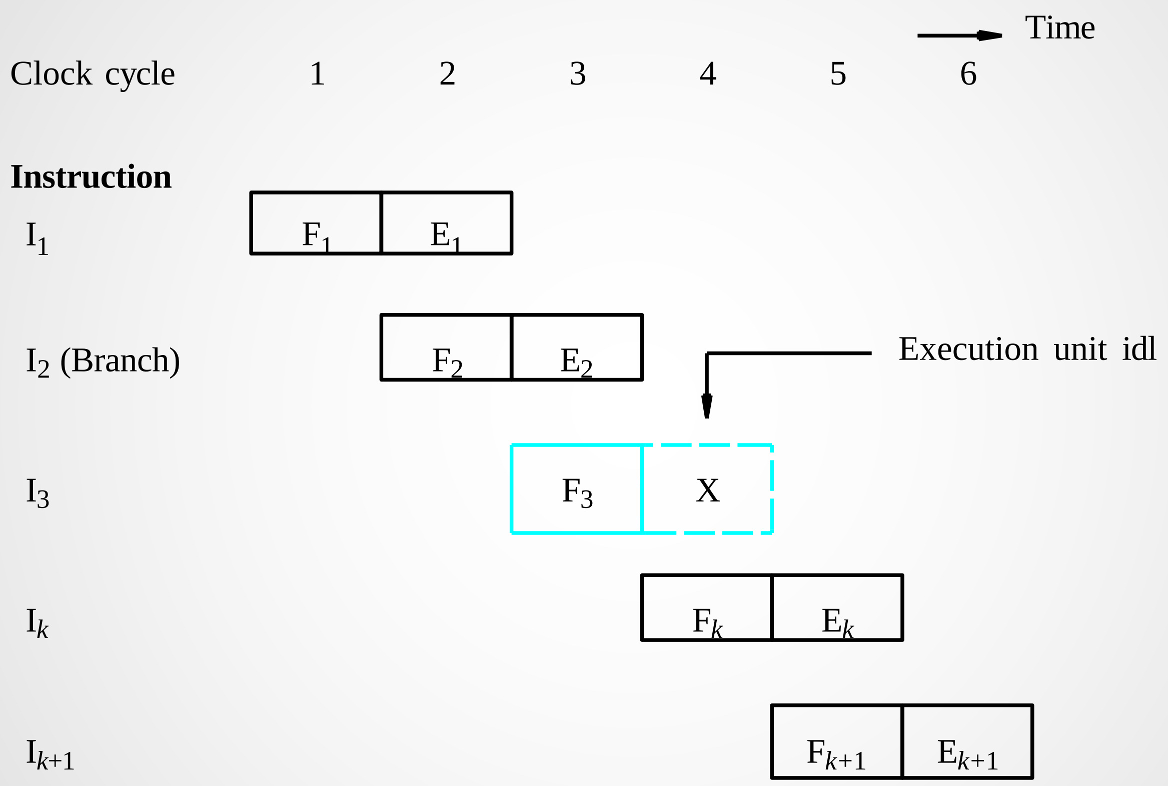
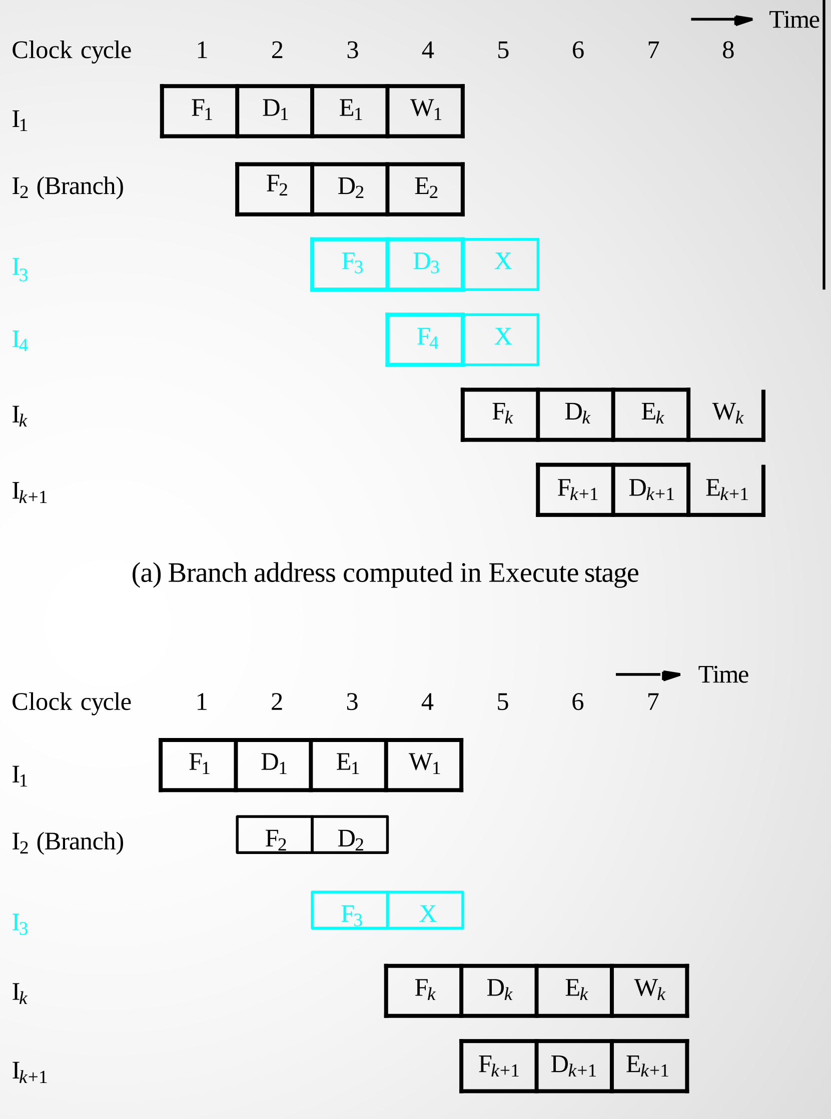


Figure 8.8. An idle cycle caused by a branch instruction.

BRANCH TIMING

- Branch penalty
- Reducing the penalty



Time
Clock cycle 1 2 3 4 5 6 7 8
I_1 F_1 D_1 E_1 W_1
I_2 (Branch) F_2 D_2 E_2
I_3 F_3 D_3 X
I_4 F_4 X
I_k F_k D_k E_k W_k
I_{k+1} F_{k+1} D_{k+1} E_{k+1}
(a) Branch address computed in Execute stage
Time
Clock cycle 1 2 3 4 5 6 7
I_1 F_1 D_1 E_1 W_1
I_2 (Branch) F_2 D_2
I_3 F_3 X
I_k F_k D_k E_k W_k
I_{k+1} F_{k+1} D_{k+1} E_{k+1}
(b) Branch address computed in Decode stage

Figure 8.9. Branch timing.

INSTRUCTION QUEUE AND PREFETCHING

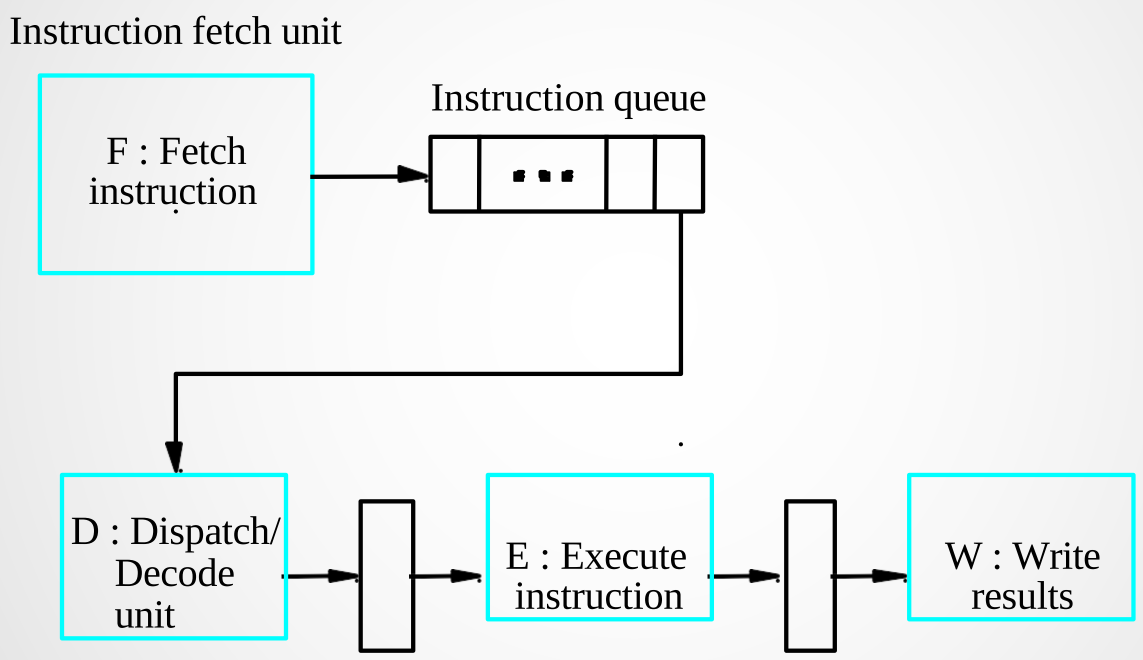


Figure 8.10. Use of an instruction queue in the hardware organization of Figure 8.2*b*.

SUPERSCALAR OPERATION

OVERVIEW

- The maximum throughput of a pipelined processor is one instruction per clock cycle.
- If we equip the processor with multiple processing units to handle several instructions in parallel in each processing stage, several instructions start execution in the same clock cycle – multiple-issue.
- Processors are capable of achieving an instruction execution throughput of more than one instruction per cycle – superscalar processors.
- Multiple-issue requires a wider path to the cache and multiple execution units.

SUPERSCALAR

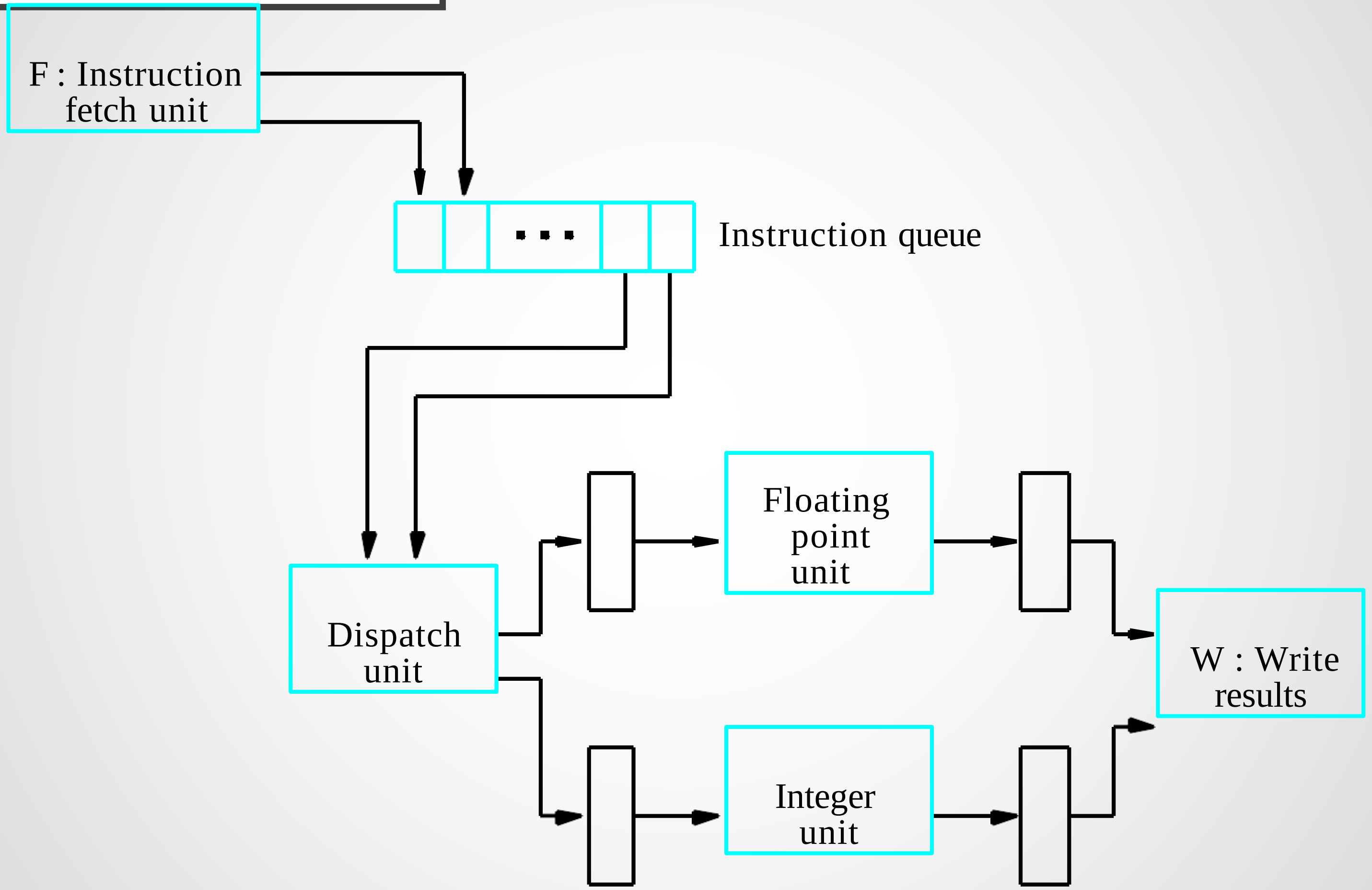


Figure 8.19. A processor with two execution units.

SIX STAGE OF INSTRUCTION PIPELINING

Fetch Instruction(FI)

Read the next expected instruction into a buffer

Decode Instruction(DI)

Determine the opcode and the operand specifiers.

Calculate Operands(CO)

Calculate the effective address of each source operand.

Fetch Operands(FO)

Fetch each operand from memory. Operands in registers need not be fetched.

Execute Instruction(EI)

Perform the indicated operation and store the result

Write Operand(WO)

Store the result in memory.

WEEK 12
SLIDES 164-173

PARALLEL PROGRAMMING MODELS

INTRODUCTION

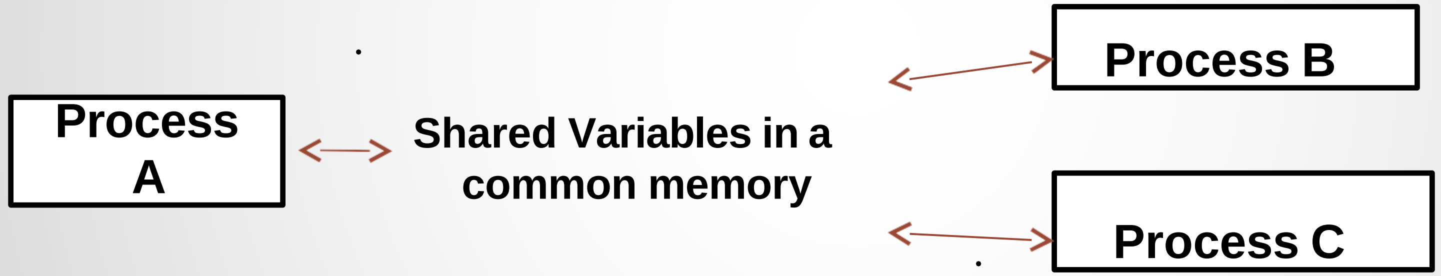
- collection of program abstractions providing a simplified and transparent view of the computer hardware/software system to the programmer
- designed for multiprocessors, multicomputer or vector/SIMD computers
- Five models:
  - Shared-Variable Model
  - Message-Passing Model
  - Data-Parallel Model
  - Object-oriented Model
  - Functional and Logic Models

SHARED-VARIABLE MODEL

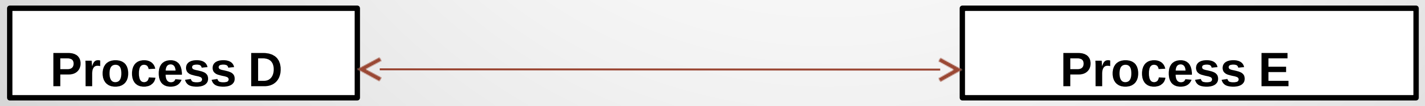
To limit the scope and rights, the process address space may be shared or restricted.

Mechanisms for IPC:

1. IPC using shared variable:



2. IPC using message passing:



FOLLOWING ARE SOME ISSUES OF SHARED-VARIABLE MODEL:

- Shared-Variable communication:
- Critical Section(CS):
  - code segment accessing shared variables.
  - Requirements are –
    - Mutual exclusion
    - No deadlock in waiting
    - Non preemption
    - Eventual entry
- Protected Access: based on CS value
  - Multiprogramming
  - Multiprocessing – two types:
    - MIMD mode
    - MPMD mode
  - Multitasking
  - Multithreading

- Partitioning and Replication:
  - Program partitioning is a technique for decomposing a large program and data set into many small pieces for parallel execution by multiple processors.
  - Program replication is referred to duplication of the same program code for parallel execution on multiple processor over different data sets.
- Scheduling and Synchronization:
  - Scheduling of divided program modules on parallel processor
  - Two types are :
    - Static scheduling
    - Dynamic scheduling
- Cache Coherence and Protection:

  If the value is returned on a read instruction is always the value written by the latest write instruction on the same memory location is called coherent.

MESSAGE-PASSING MODEL

- Synchronous Message Passing –
  - It is must synchronize the sender process and the receiver process in time and space
- Asynchronous Message Passing –
  - It does not require message sending and receiving be synchronized in time and space
  - Non blocking can be achieved
- Distributing the computations:
  - Subprogram level is handled rather than at the instructional or fine grain process level in a tightly coupled multiprocessor

DATA-PARALLEL MODEL

- It is easier to write and to debug because parallelism is explicitly handled by hardware synchronization and flow control.
- It requires the use of pre-distributed data sets
- Synchronization is done at compile time rather than run time.
- the following are some issued handled
  - Data Parallelism-
  - Array Language Extensions
  - Compiler support

OBJECT-ORIENTED MODEL

- Concurrent OOP – 3 application demands
  - There is increased use of interacting processes by
  - individual users
  - Workstation networks have become a cost-effective mechanism
  - Multiprocessor technology in several variants has
  - advanced to the point of providing supercomputing power
- An actor model
  - It is presented as one framework for COOP
  - They are self-contained , interactive, independent components of a computing system.
  - Basic primitives are :create to , send to, become
- Parallelism in COOP:
  - 3 patterns- 1. pipeline concurrency 2.divide and conquer currency 3.cooperative problem solving

FUNCTIONAL AND LOGIC MODELS

- Two types of language oriented programming models are
  - Functional programming model
    - It emphasizes functionality of a program
    - No concepts of storage, assignment and branching
    - All single-assignment and dataflow languages are functional in nature
    - Some e.g. are Lisp, SISAL and strand 88
  - Logic programming model
    - Based on logic ,logic programming that suitable fordealing with large database.
    - Some e.g. are
      - concurrent Prolog -
      - Concurrent Parlog

WEEK 13
SLIDES 174-183

DISTRIBUTING PROCESSING

OUTLINE

- Distributed Processing
- Distributed System
- Architecture
- Form of D.P
- Techniques
- Challenges
- Advantage/Disadvantage

What is Distributed Processing

- Distributed Processing is a technique of distributing the information over a number of devices.
- The devices may be computers or data terminals with some level of intelligence.
- The devices are interconnected with communication facilities.

What is Distributed System

- A distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages.
- Examples
  - The internet
  - An intranet which is a portion of the internet managed by an organization

Architecture

- Software layers
- System architectures
- Interfaces and objects
- Design requirements for distributed architectures

Software layers

- Applications, services
- Middleware
- Operating system
- Computer and network hardware

System architectures

- Client-server model
- Services provided by multiple servers
- Proxy srvers and caches
- Peer processes

Design requirements for distributed architectures

- Performance issues
- Use of caching and replication
- Dependability issues

Design requirements for distributed architectures

- Performance issues
  - Responsiveness
  - Balancing computer loads
  - Quality of services
- Caching and replication
  - The performance issues often appear to be major obstacles to the successful deployment of DS, but much progress has been made in the design of systems that overcome them by the use of data replication and caching.
- Dependability issues
  - Correctness
  - Security
  - Fault tolerance

WEEK 14
SLIDES 184-200

Form of Distributed Processing

- Distributed Applications
- Distributed Devices
- Network Management and Control
- Distributed Data

Distributed Applications

- One application splits up into components that are dispersed among a number of machines
- One application replicated on a number of machines
- A number of different applications distributed among a number of machines
- Can be characterized by vertical or horizontal partitioning

Distributed Devices

- Support a distributed set of devices that can be controlled by processors, e.g. ATMs or laboratory interface equipments
- Distribution of processing technology to various locations of the manufacturing process in factory automation

Techniques of Distributed Processing

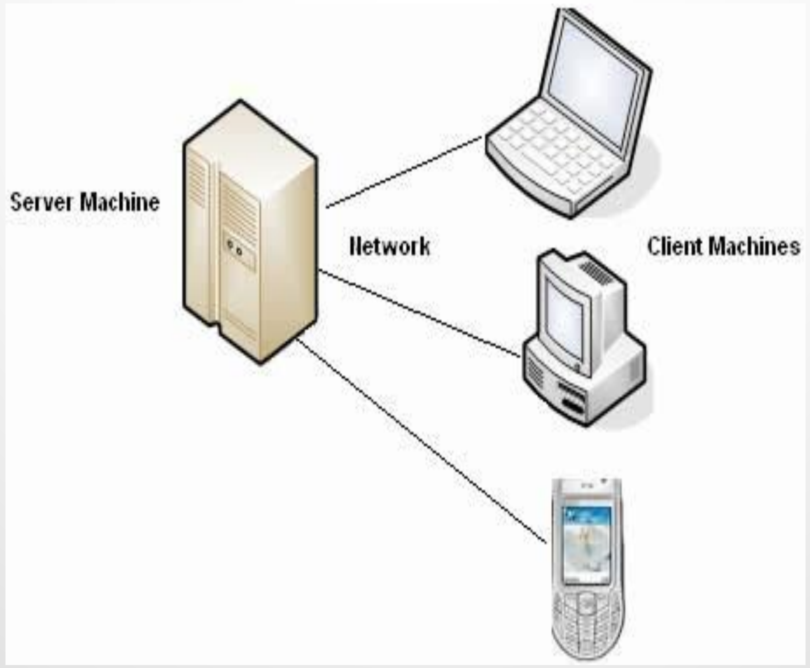
- Centralized
- Decentralized
- Parallel
- Open Distributed Processing
- Clustering

Centralized

- **Centralized Processing** is done at a central location, using terminals that are attached to a central computer
- The central computer performs the computing functions and controls the remote terminals. This type of system relies totally on the central computer.

Centralized

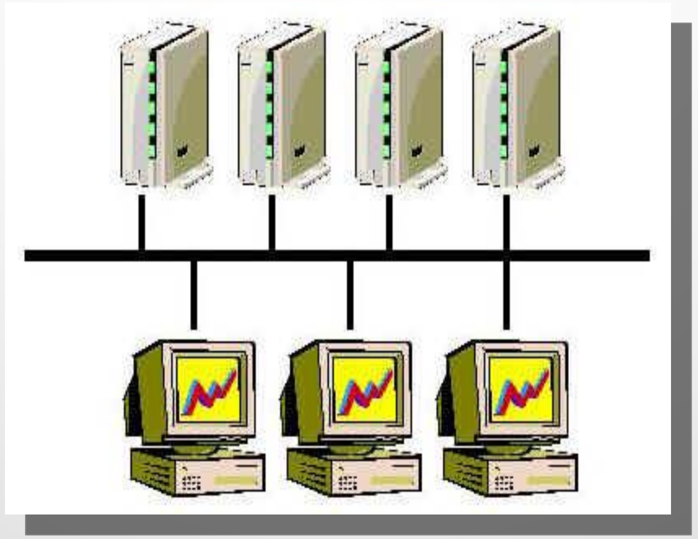
- <u>Example</u>: Client/server is the most common example of centralized processing, where server is controlling all the activities on the network.



Decentralized

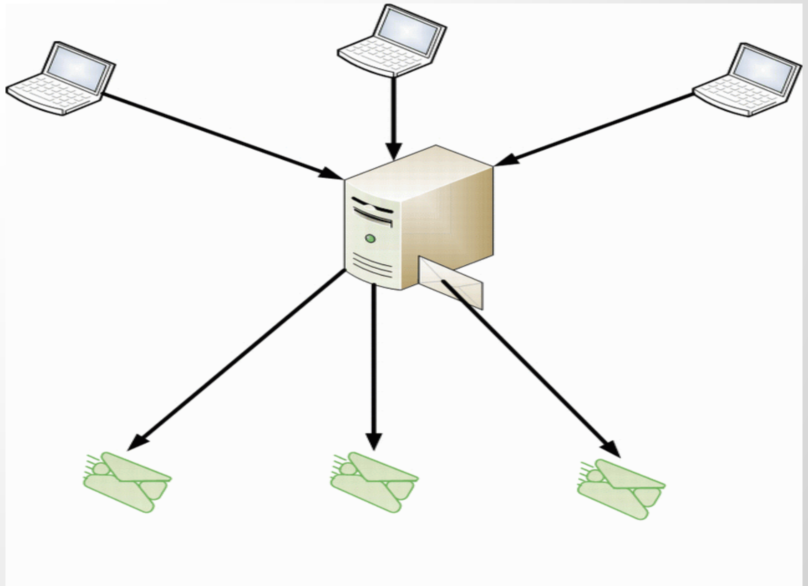
- Computer systems in different locations. Although data may be transmitted between the computers periodically

- **<u>Example:</u>** Yahoo server is the example of the decartelized processing. On each login it connects you to a different server .



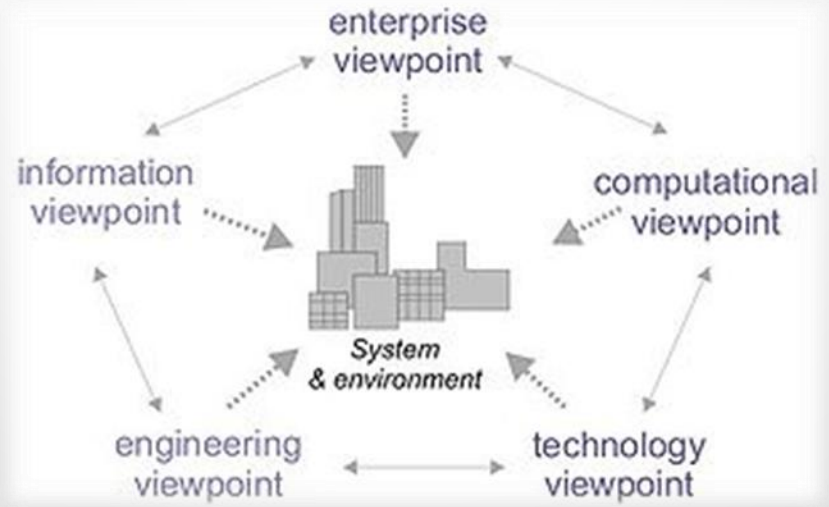
Parallel Processing

- Parallel processing is the simultaneous processing of the same task on two or more microprocessors in order to obtain faster results
- Multiple processor
- Multiple computer
- Shared memory resources


Parallel Processing

Open Distributed Processing

- ODP is a reference model in computer science, which provides a coordinating framework for the standardization of open distributed processing (ODP).
- It supports distribution, interworking, platform and technology independence, and portability, together with an enterprise architecture framework for the specification of ODP systems.



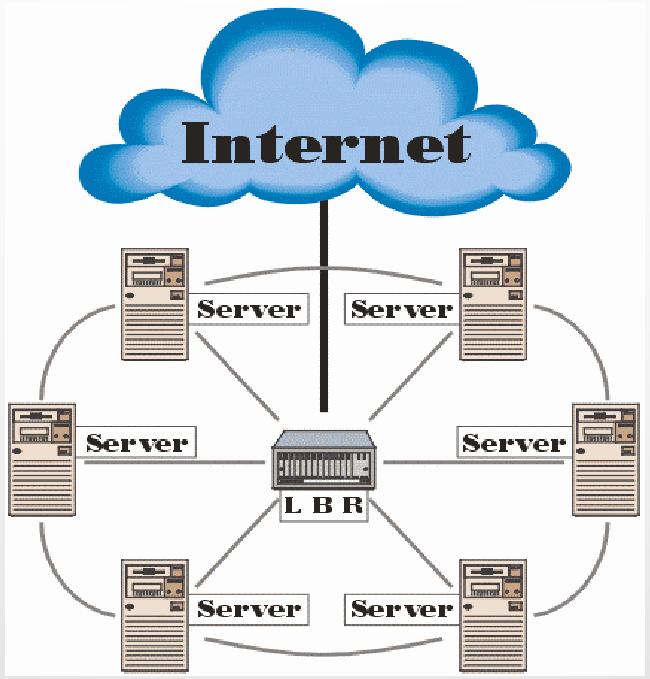
Clustering

- A cluster is a group of individual computer systems that can be made to appear as one computer system.
- Clustering is just one form of parallel computing.
- key points that distinguishes clustering from other is the ability to view the cluster as either a single entity or a collection of stand-alone systems
- **For example:** a cluster of web servers can appear as one large web server, but at the same time, individual systems within the cluster can be accessed as individual systems,

Clustering

- **<u>Example:</u>**

  Internet is an example of clustering, where different server are working but they look like a single server.



Challenges

- Heterogeneity
- Security
- Scalability
- Failure handling
- Concurrency

Advantages

- Quicker response time
  - By locating processing power close to user, response time is typically improved. This means that the system responds rapidly to commands entered by users.
- Lower costs
  - Long-distance communication costs are declining at a slower rate than the cost of computer power
  - Distributed processing can reduce the volume of data that must be transmitted over long-distances and thereby reduce long-distance costs.
- Improved data integrity
  - High degrees of accuracy and correctness may be achieved by giving users control over data entry and storage.

Advantages

- Reduced host processor costs
  - The productive life of a costly mainframe can be extended by off-loading some its processing tasks to other, less expensive machines
- Resource sharing
  - One of the main advantages of developing microcomputer networks is because they make it possible to share expensive resources such as high-speed, color laser printers, fast data storage devices, and high-priced software packages.

Disadvantage

- Complexities
  - A lot of extra programming is required to set up a distributed system
- Network failure
  - Since distributed system will be connected through network and in case of network failure non of the systems will work
- Security
  - The information need to be passed between the network. And it can be tracked and can be used for illegal purpose.
- Costly software
- Not all situations are suitable for distributed computing

WEEK 15
SLIDES 201-217

DISTRIBUTED DATABASE

OUTLINE

- Concept
- Distributed Database Types
  - Homogeneous
  - Heterogeneous
- Distributed Database Design
  - Data Fragmentation
  - Data Allocation
  - Data Replication

CONCEPT

- A distributed database (DDB) is a collection of multiple, *logically interrelated* databases distributed over a *computer network*.

- A distributed database management system (D–DBMS) is the software that manages the DDB and provides an access mechanism that makes this distribution transparent to the users.

- Distributed database system (DDBS) = DDB + D–DBMS

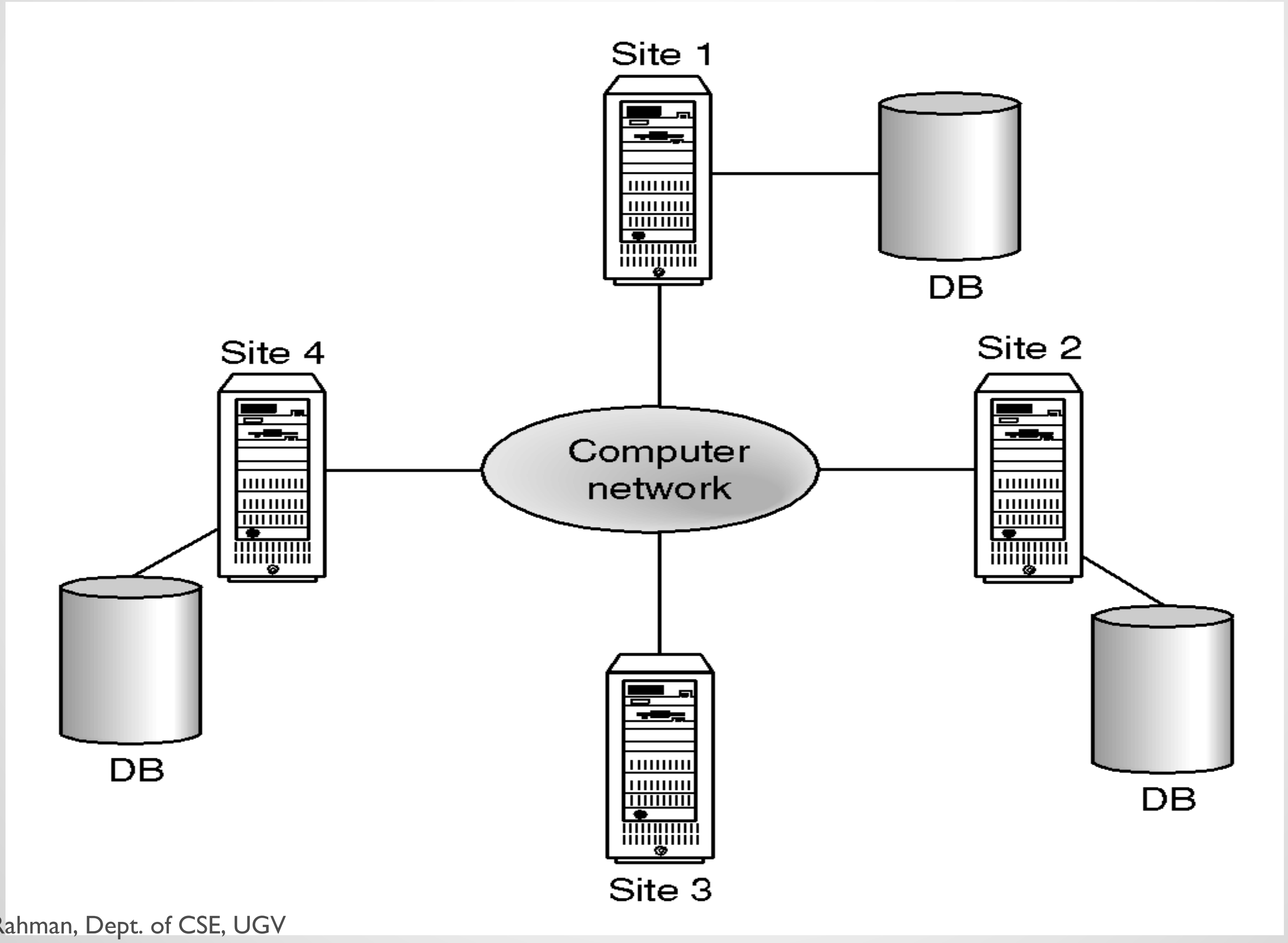
CONCEPT

- Collection of logically-related shared data.
- Data split into fragments.
- Fragments may be replicated.
- Fragments/replicas allocated to sites.
- Sites linked by a communications network.
- Data at each site is under control of a DBMS.
- DBMSs handle local applications autonomously.
- Each DBMS participates in at least one global application.

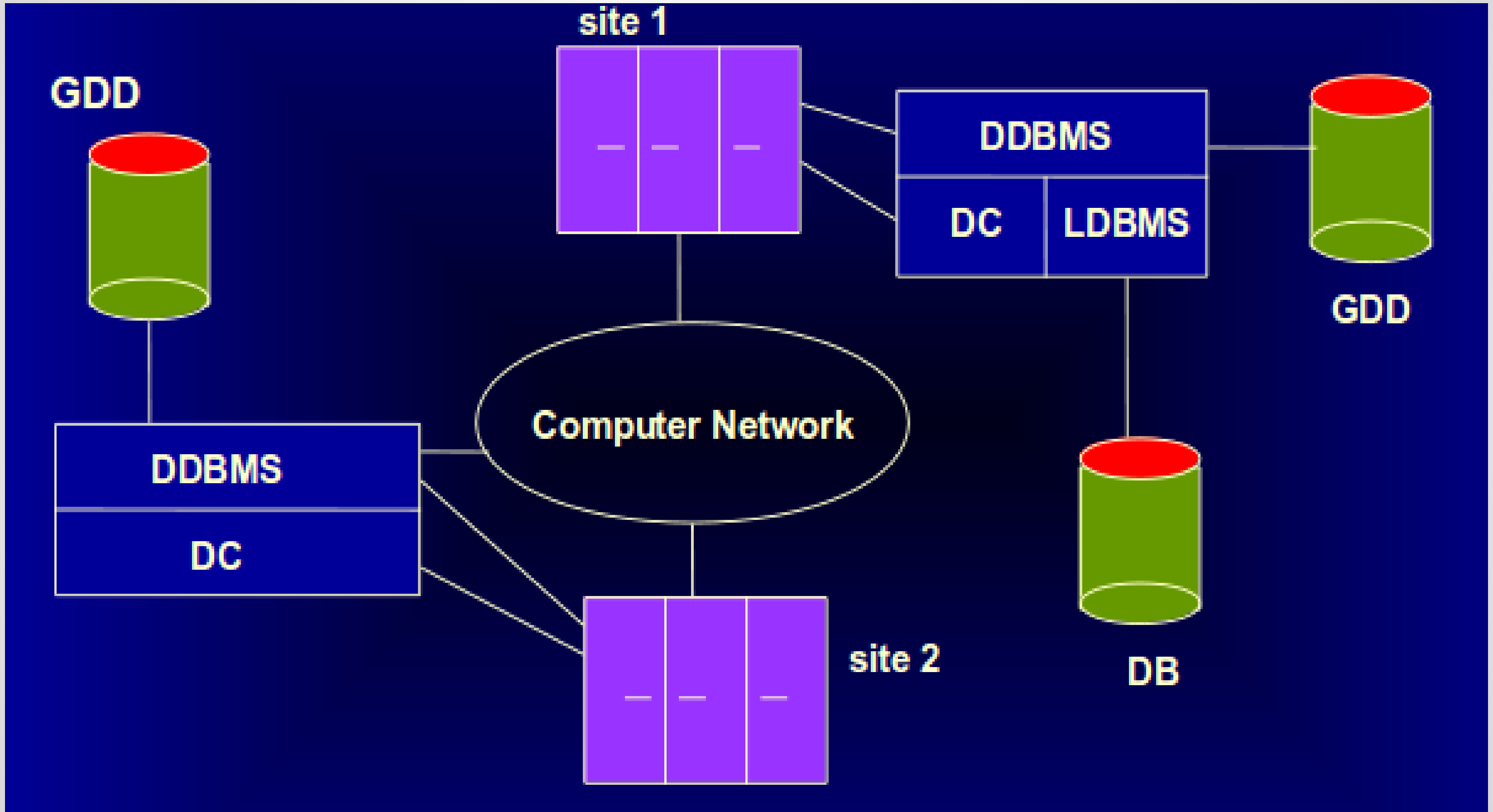
FUNCTIONALITY

- Security
- Keeping track of data
- Replicated data management
- System catalog management
- Distributed transaction management
- Distributed database recovery

DISTRIBUTED DBMS



COMPONENT ARCHITECTURE FOR A D-DBMS



LDBMS : Local DBMS component
DC : Data communication component
GDD : Global Data Dictionary

ADVANTAGES OF D-DBMS

- Organizational Structure
- Share-ability and Local Autonomy
- Improved Availability
- Improved Reliability
- Improved Performance
- Economics
- Modular Growth

DISADVANTAGES OF D-DBMS

- Complexity
- Cost
- Security
- Integrity Control More Difficult
- Lack of Standards
- Lack of Experience
- Database Design More Complex

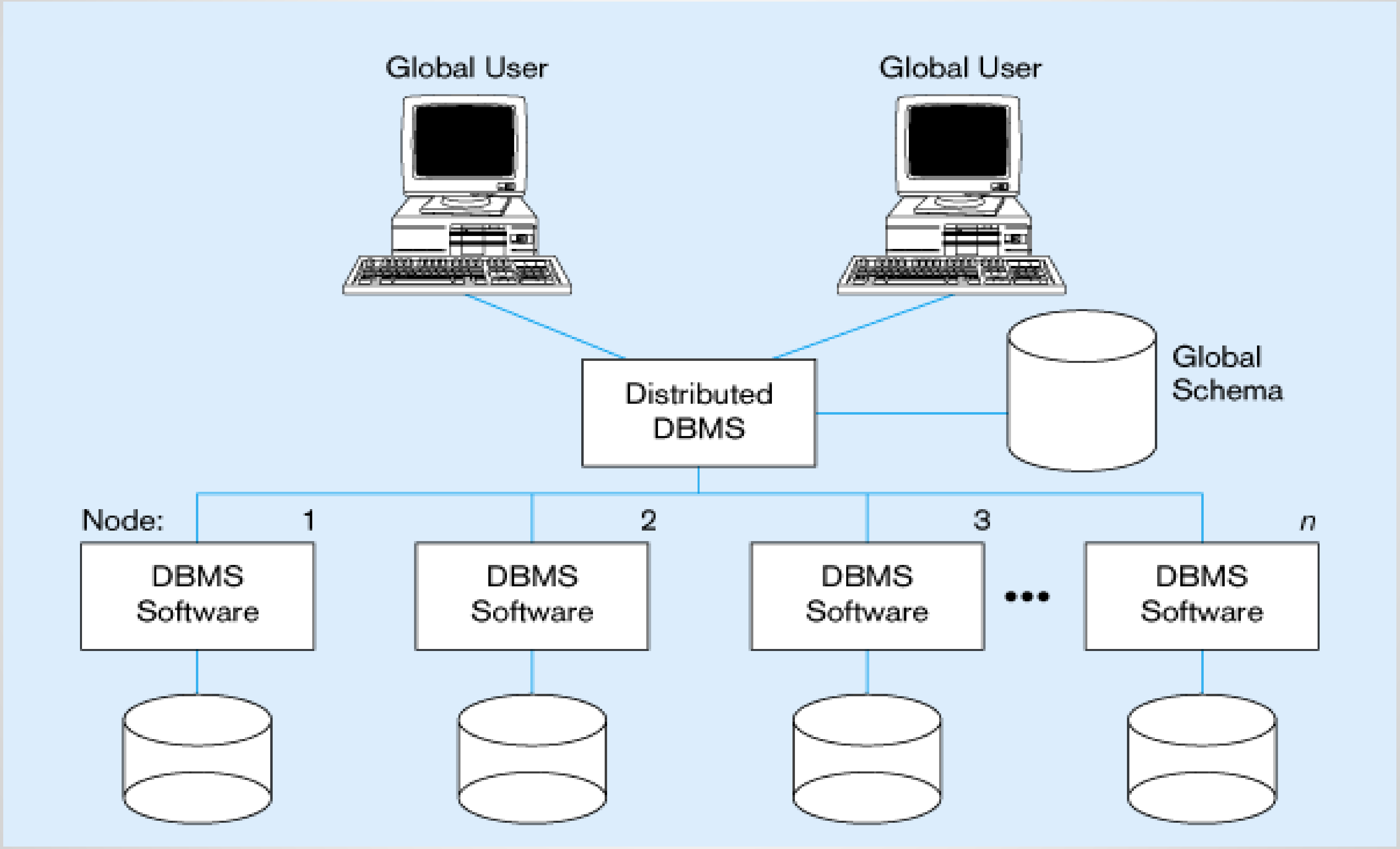
TYPES OF D-DBMS

- Homogeneous D-DBMS
- Heterogeneous D-DBMS

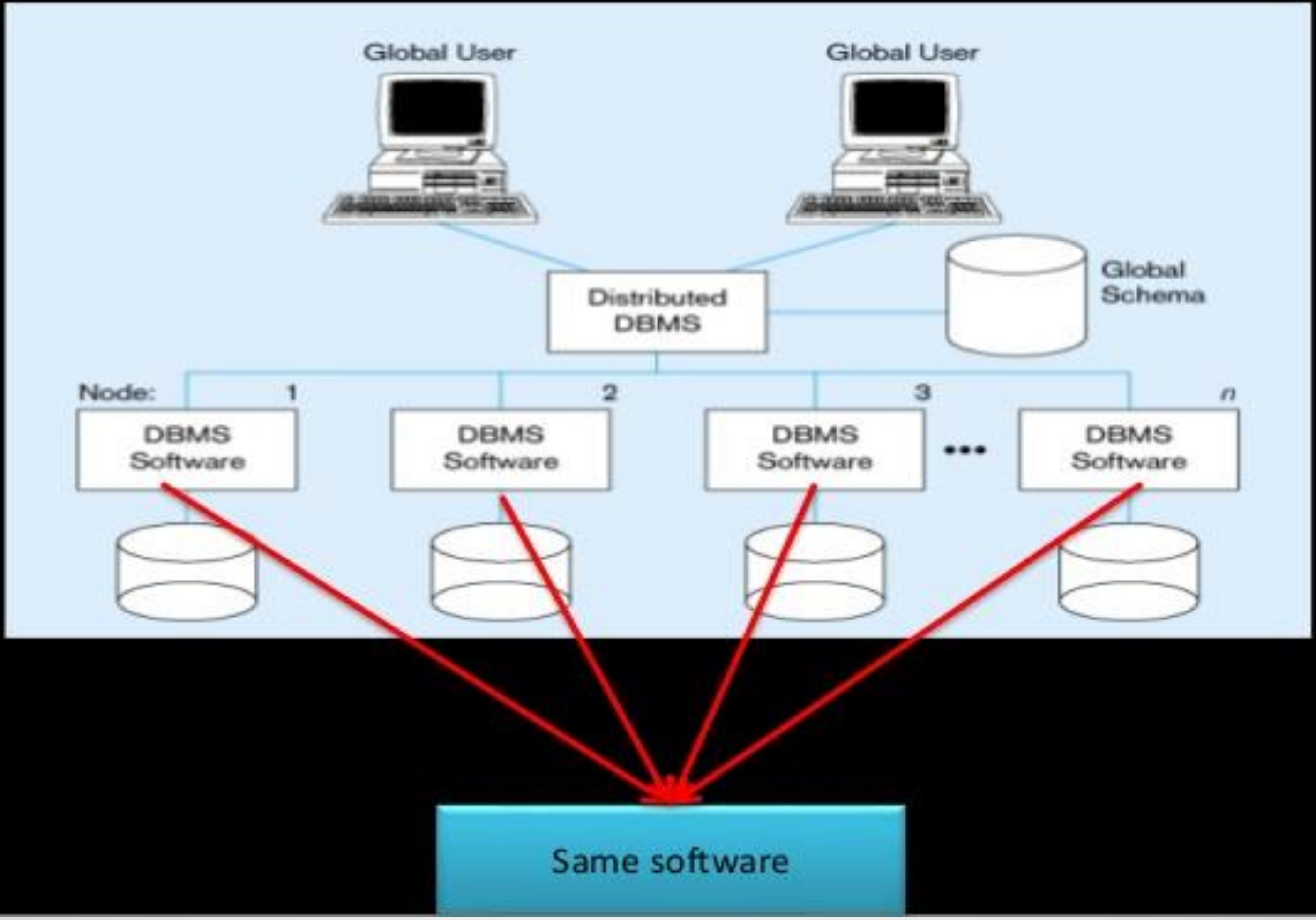
HOMOGENEOUS D-DBMS

- All sites have identical software and are aware of each other and agree to cooperate in processing user requests.
- Much easier to design and manage
- The operating system used, at each location must be same or compatible.
- The database application (or DBMS) used at each location must be same or compatible.
- It appears to user as a single system
- All access is through one, global schema
- The global schema is the *union* of all the local schema

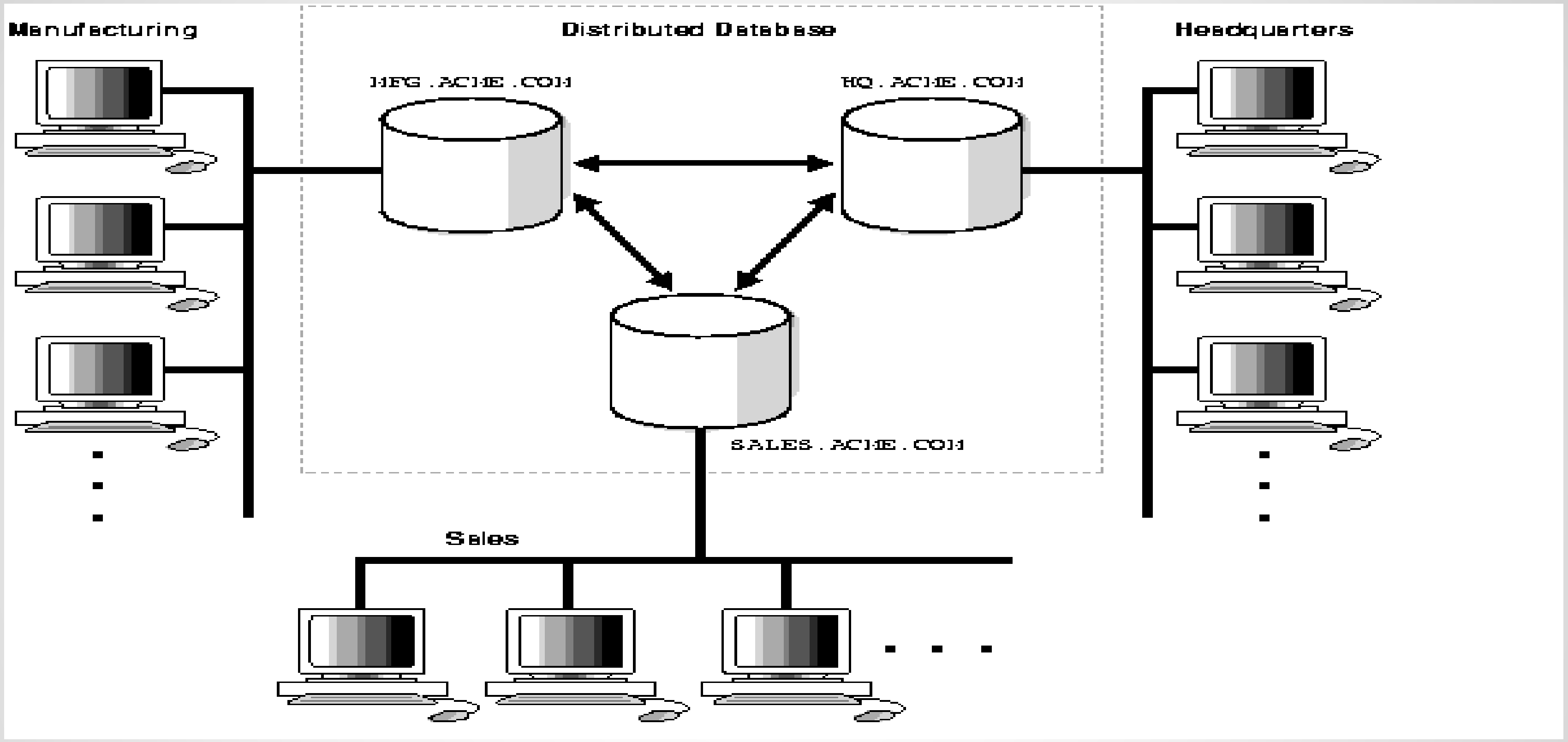
HOMOGENEOUS DATABASE



Homogeneous Database



HOMOGENEOUS DISTRIBUTED DATABASE EXAMPLE

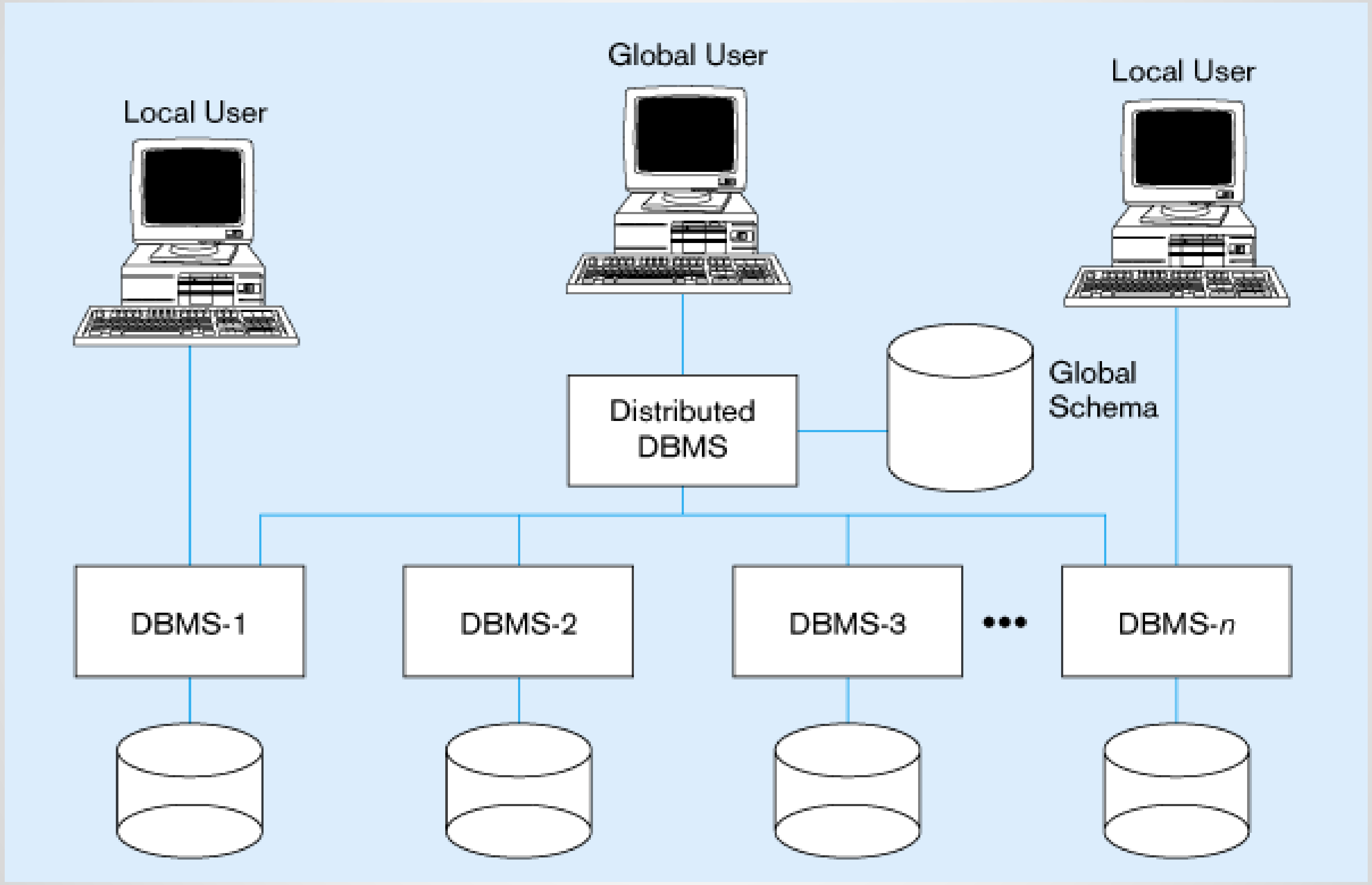


- A distributed system connects three databases: hq, mfg, and sales
- An application can simultaneously access or modify the data in several databases in a single distributed environment.

HETEROGENEOUS D-DBMS

- Different sites may use different schema and software.
- Different nodes may have different hardware & software and data structures at various nodes or locations are also incompatible.
- Different computers and operating systems, database applications or data models may be used at each of the locations.
- Difficult to manage and design.
- Local access is done using the local DBMS and schema
- Remote access is done using the global schema

TYPICAL HETEROGENEOUS ENVIRONMENT



WEEK 16
SLIDES 218-235

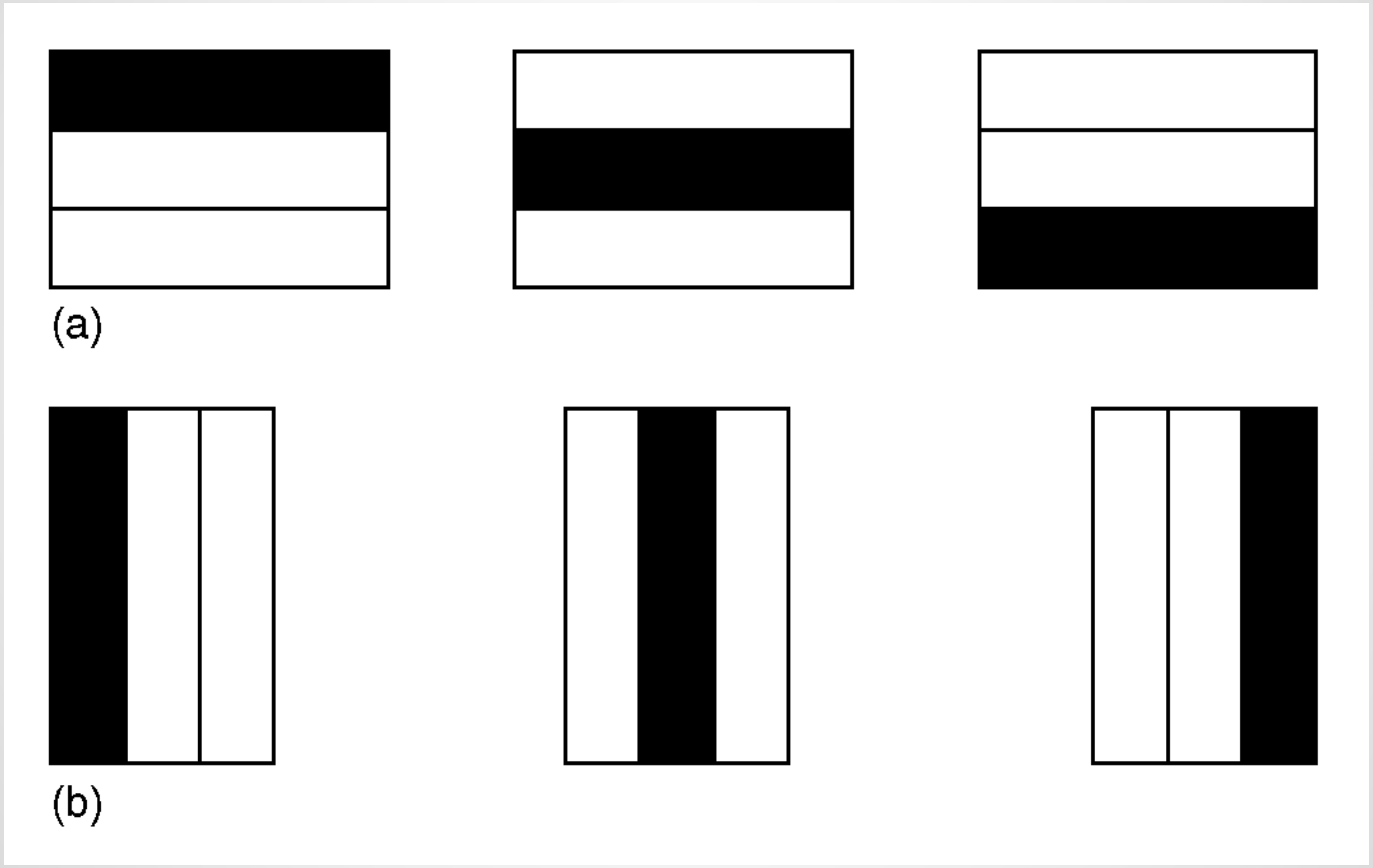
DISTRIBUTED DATABASE DESIGN

- Three key issues:
  - Data Fragmentation
    - Relation may be divided into a number of sub relations, which are then distributed.
    - Breaking up the database into logical units called **fragments** and assigned for storage at various sites.
  - Data Allocation
    - The process of assigning a particular fragment to a particular site in a distributed system.
  - Data Replication
    - Copy of fragment may be maintained at several sites.

DISTRIBUTED DATABASE DESIGN

- Data Fragmentation
  - data can be distributed by storing individual tables at different sites
  - data can also be distributed by decomposing a table and storing portions at different sites – called **Fragmentation**
  - fragmentation can be **horizontal** or **vertical**

HORIZONTAL AND VERTICAL FRAGMENTATION



WHY USE FRAGMENTATION?

- **Usage** – in general applications use views so it's appropriate to work with subsets
- **Efficiency** – data stored close to where it is most frequently used
- **Parallelism** – a transaction can divided into several sub-queries to increase degree of concurrency
- **Security** – data more secure – only stored where it is needed

Disadvantages:

Performance - may be slower

Integrity - more difficult

DISTRIBUTED DATABASE DESIGN

- Horizontal Fragmentation
  - Each fragment, T_i , of table *T* contains a subset of the rows
  - Each tuple of *T* is assigned to one or more fragments
  - Horizontal fragmentation is lossless
  - A selection condition may be composed of several conditions connected by AND or OR
  - Derived horizontal fragmentation: It is the partitioning of a primary relation to other secondary relations which are related with Foreign keys

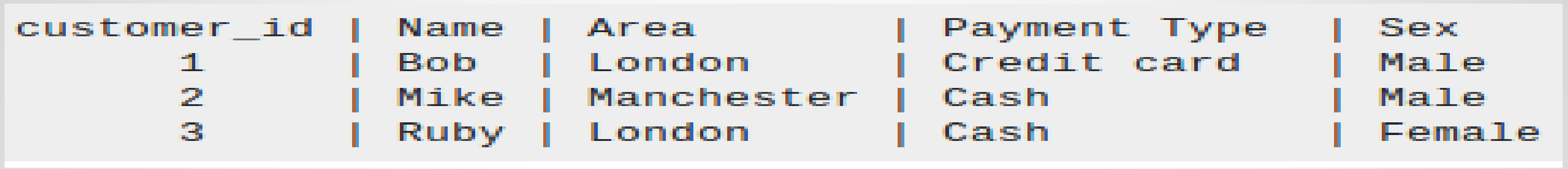
HORIZONTAL FRAGMENTATION EXAMPLE

$$account_1 = \sigma_{branch\text{-}name=\text{“Hillside”}}(account)$$

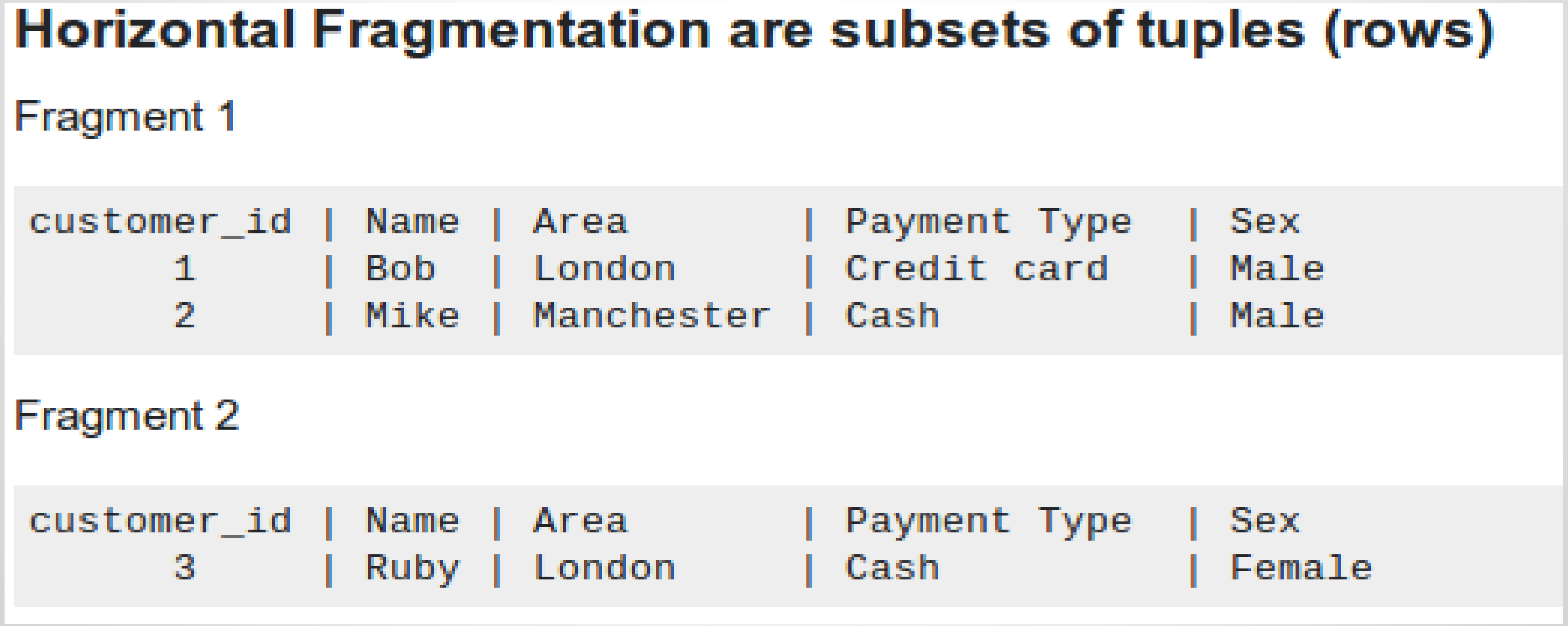
branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

- A bank account schema has a relation
  Account-schema = (*branch-name*, *account-number, balance*).
- It fragments the relation by location and stores each fragment locally: rows with branch-name = `Hillside` are stored in the Hillside in a fragment

HORIZONTAL FRAGMENTATION EXAMPLE



```
customer_id | Name | Area       | Payment Type | Sex
     1      | Bob  | London     | Credit card  | Male
     2      | Mike | Manchester | Cash         | Male
     3      | Ruby | London     | Cash         | Female
```



Horizontal Fragmentation are subsets of tuples (rows)

Fragment 1

```
customer_id | Name | Area       | Payment Type | Sex
     1      | Bob  | London     | Credit card  | Male
     2      | Mike | Manchester | Cash         | Male
```

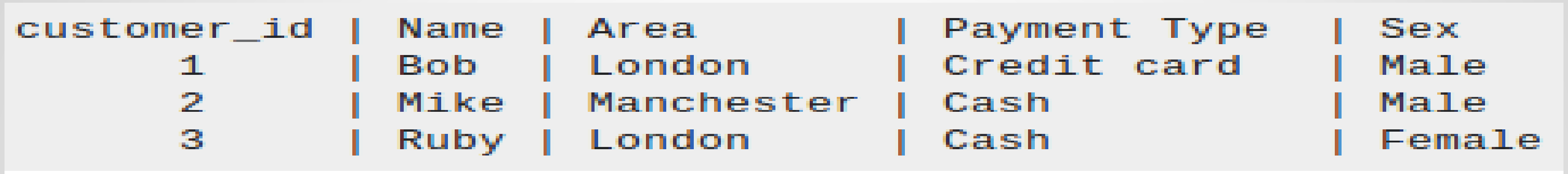
Fragment 2

```
customer_id | Name | Area       | Payment Type | Sex
     3      | Ruby | London     | Cash         | Female
```

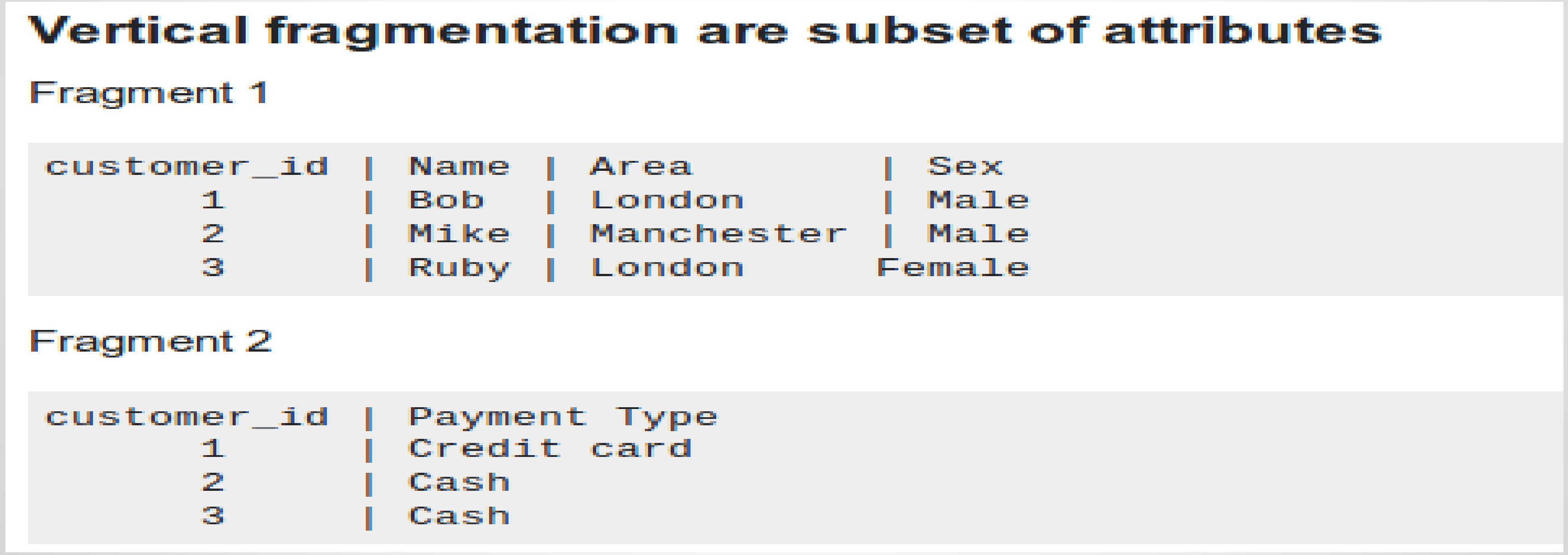
DISTRIBUTED DATABASE DESIGN

- Vertical Fragmentation
  - It is a subset of a relation which is created by a subset of columns. Thus a vertical fragment of a relation will contain values of selected **columns**. There is no selection condition used in vertical fragmentation.
  - Consider the customer relation. A vertical fragment can be created by keeping the values of Name, Area, Sex.
  - Because there is no condition for creating a vertical fragment, each fragment must include the primary key attribute of the parent relation customer. In this way all vertical fragments of a relation are connected.

VERTICAL FRAGMENTATION EXAMPLE



```
customer_id | Name | Area       | Payment Type   | Sex
     1      | Bob  | London     | Credit card    | Male
     2      | Mike | Manchester | Cash           | Male
     3      | Ruby | London     | Cash           | Female
```



Vertical fragmentation are subset of attributes

Fragment 1

```
customer_id | Name | Area       | Sex
     1      | Bob  | London     | Male
     2      | Mike | Manchester | Male
     3      | Ruby | London     Female
```

Fragment 2

```
customer_id | Payment Type
     1      | Credit card
     2      | Cash
     3      | Cash
```

DISTRIBUTED DATABASE DESIGN

- Data Allocation
  - Four alternative strategies regarding placement of data
    - Centralized
    - Partitioned (or Fragmented)
    - Complete Replication
    - Selective Replication

DATA ALLOCATION

- Centralized
    - Consists of single database and DBMS stored at one site with users distributed across the network.
- Partitioned
    - Database partitioned into disjoint fragments, each fragment assigned to one site.
- Complete Replication
    - Consists of maintaining complete copy of database at each site.
- Selective Replication
    - Combination of partitioning, replication, and centralization.

DISTRIBUTED DATABASE DESIGN

- Data Replication
  - System maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance.

ISSUES OF REPLICATION

- **Data timeliness** – high tolerance for out-of-date data may be required
- **DBMS capabilities** – if DBMS cannot support multi-node queries, replication may be necessary
- **Performance implications** – refreshing may cause performance problems for busy nodes
- **Network heterogeneity** – complicates replication
- **Network communication capabilities** – complete refreshes place heavy demand on telecommunications

ADVANTAGES OF REPLICATION

- **Availability**: failure of site containing relation *r* does not result in unavailability of *r* is replicas exist.
- **Parallelism**: queries on *r* may be processed by several nodes in parallel.
- **Reduced data transfer**: relation *r* is available locally at each site containing a replica of *r*.

DISADVANTAGES OF REPLICATION

- **Increased cost of updates**: each replica of relation *r* must be updated.
- **Increased complexity of concurrency control**: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.
- One solution: choose one copy as primary copy and apply concurrency control operations on primary copy.

Thank You!

WEEK 17
SLIDES 236-263

Parallel Computing and Programming Environments

- **MapReduce**
- **Hadoop**
- **Amazon Web Services**

What is MapReduce?

- **Simple data-parallel programming model**
- For large-scale data processing
  - Exploits large set of commodity computers
  - Executes process in distributed manner
  - Offers high availability
- Pioneered by Google
  - Processes 20 petabytes of data per day
- Popularized by open-source Hadoop project
  - Used at Yahoo!, Facebook, Amazon, …

What is MapReduce used for?

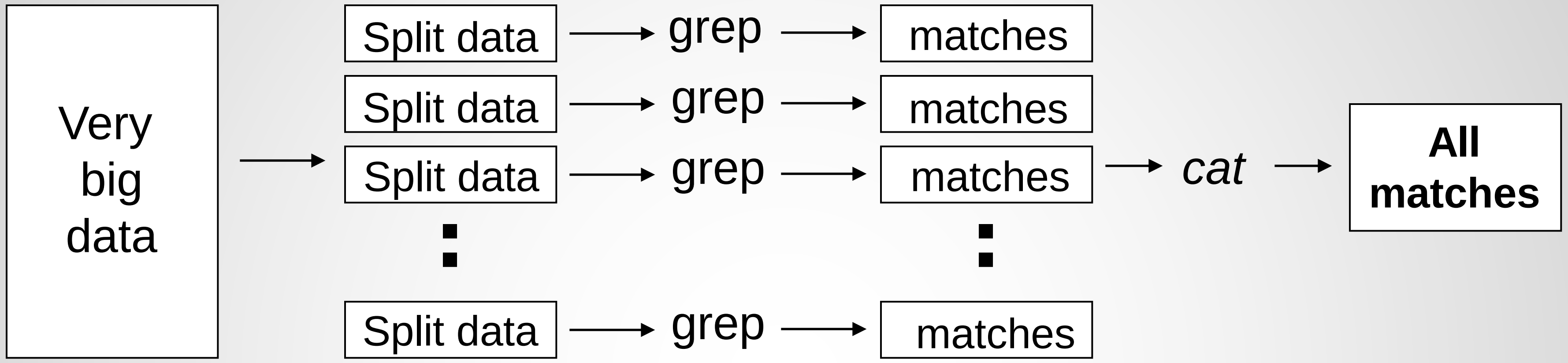
- At Google:
  - Index construction for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!:
  - “Web map” powering Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

What is MapReduce used for?

Many tasks composed of processing lots of data to produce lots of other data

- ➢MapReduce provides
  - ➢ User-defined functions
  - ➢ Automatic parallelization and distribution
  - ➢ Fault-tolerance
  - ➢ I/O scheduling
  - ➢ Status and monitoring
- oIn research:
  - o Astronomical image analysis (Washington)
  - o Bioinformatics (Maryland)
  - o Analyzing Wikipedia conflicts (PARC)
  - o Natural language processing (CMU)
  - o Particle physics (Nebraska)
  - o Ocean climate simulation (Washington)

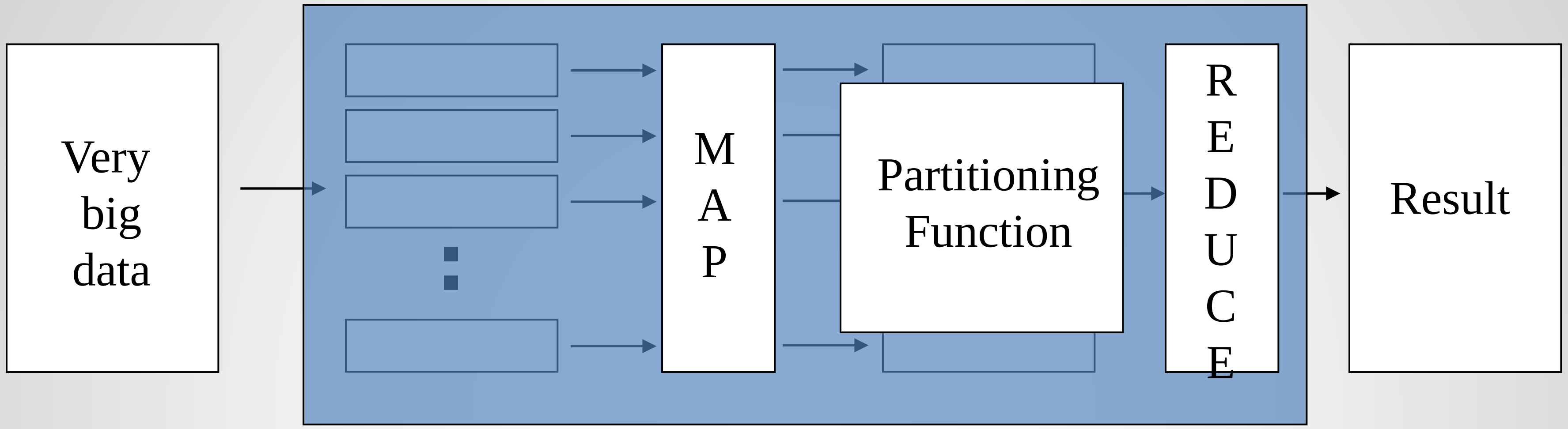
Distributed Grep



grep is a command-line utility for searching plain-text data sets for lines matching a regular expression.

cat is a standard Unix utility that concatenates and lists files

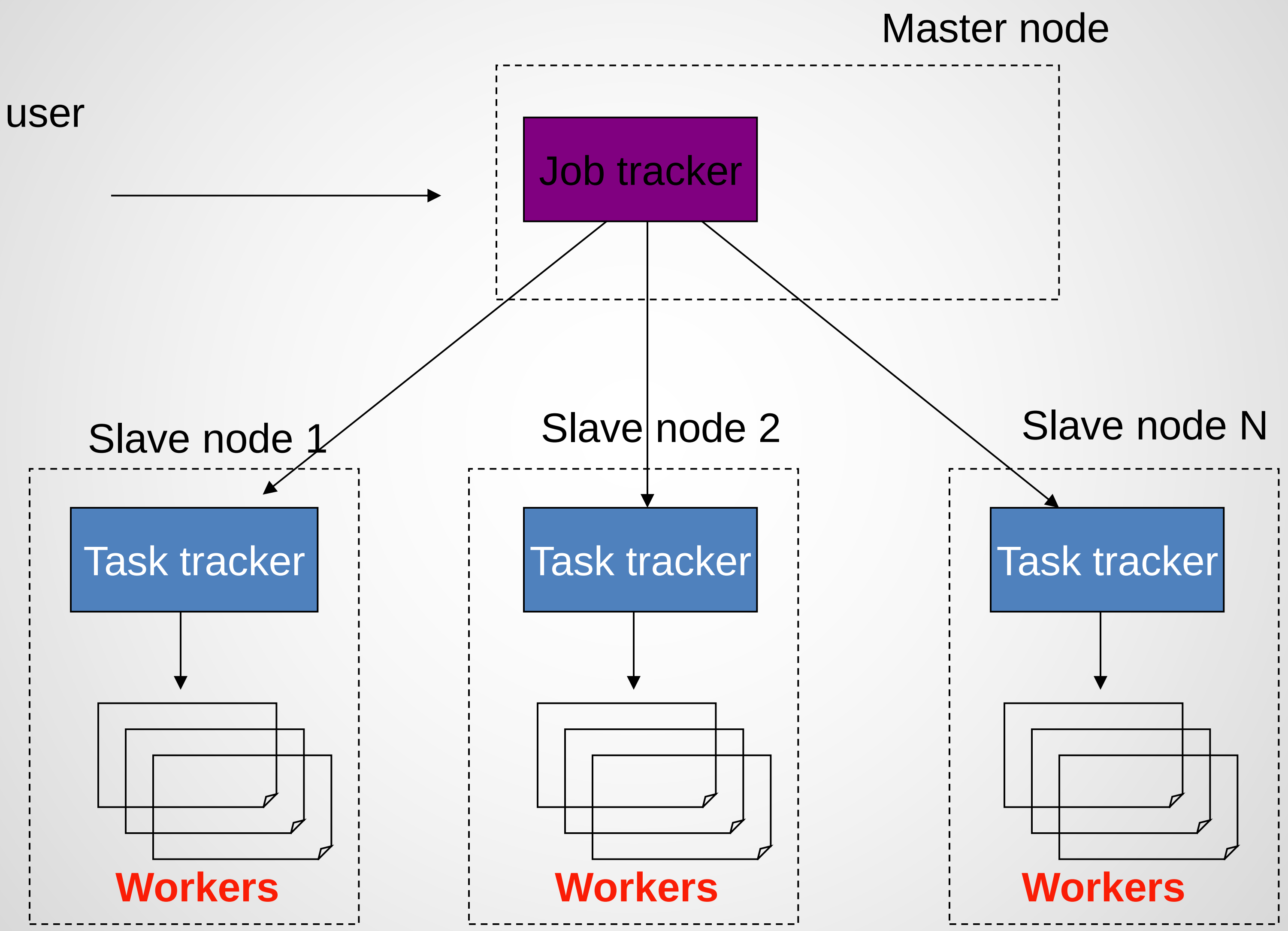
Map+Reduce



- Map:
  - Accepts *input* key/value pair
  - Emits *intermediate* key/value pair

- Reduce :
  - Accepts *intermediate* key/value* pair
  - Emits *output* key/value pair

Architecture Overview



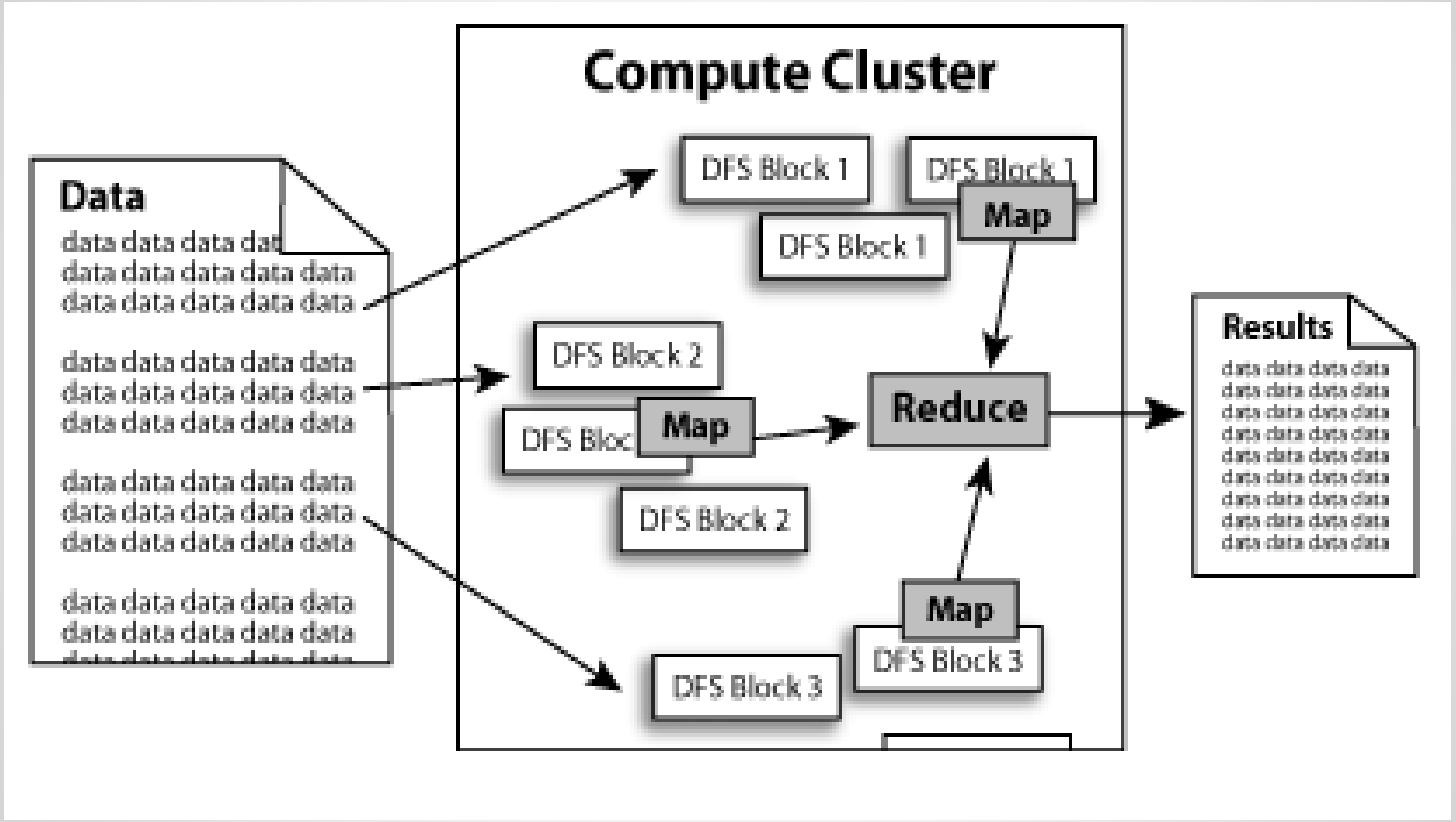
Functions in the Model

- Map
  - ➢ Process a key/value pair to generate intermediate key/value pairs
- Reduce
  - ➢ Merge all intermediate values associated with the same key
- Partition
  - ➢ By default : hash(key) mod R
  - ➢ Well balanced

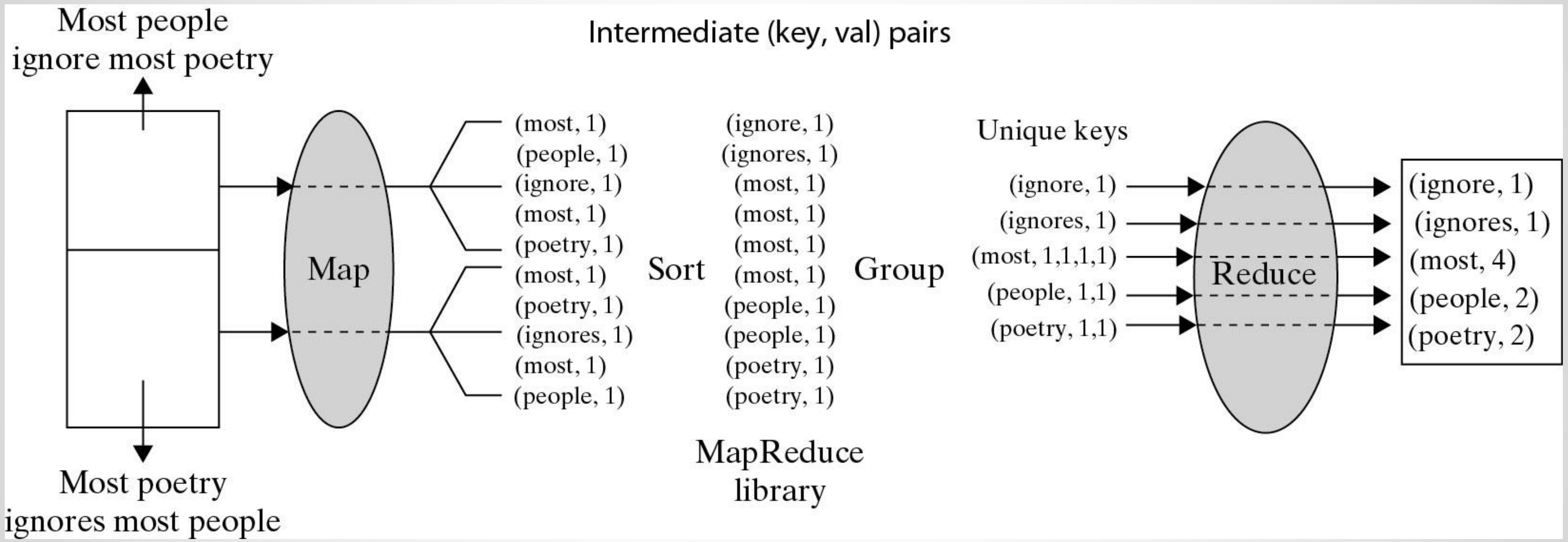
Programming Concept

- Map
  - **Perform** a function on **individual values** in a data set to create a **new list** of values
  - Example: square x = x * x
    map square [1,2,3,4,5]
    returns [1,4,9,16,25]
- Reduce
  - **Combine** values in a data set to create a new **value**
  - Example: sum = (each elem in arr, total +=)
    reduce [1,2,3,4,5]
    returns 15 (the sum of the elements)

Programming Structure



A Word Counting Example on <Key, Count> Distribution



MapReduce : Operation Steps

When the user program calls the MapReduce function, the following sequence of actions occurs :

1) The MapReduce library in the user program first splits the input files into M pieces – 16 megabytes to 64 megabytes (MB) per piece. It then starts up many copies of program on a cluster of machines.
2) One of the copies of program is master. The rest are workers that are assigned work by the master.
3) A worker who is assigned a map task :
   - reads the contents of the corresponding input split
   - parses key/value pairs out of the input data and passes each pair to the user - defined Map function.

   The intermediate key/value pairs produced by the Map function are buffered in memory.

MapReduce : Operation Steps

4) The buffered pairs are written to local disk, partitioned into R regions by the partitioning function.

   The location of these buffered pairs on the local disk are passed back to the master, who forwards these locations to the reduce workers.

5) When a reduce worker is notified by the master about these locations, it reads the buffered data from the local disks of the map workers.

   When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.

6) The reduce worker iterates over the sorted intermediate data and for each unique intermediate key, it passes the key and the corresponding set of intermediate values to the user's Reduce function.

   The output of the Reduce function is appended to a final output file.

7) When all map tasks and reduce tasks have been completed, the master wakes up the user program.

   At this point, MapReduce call in the user program returns back to the user code.

   After successful completion, output of the mapreduce execution is available in the R output files.

Locality Issue

- Master scheduling policy
  - Asks GFS for locations of replicas of input file blocks
  - Map tasks typically split into 64MB (== GFS block size)
  - Map tasks scheduled so GFS input block replica are on same machine or same rack
- Effect
  - Thousands of machines read input at local disk speed
  - Without this, rack switches limit read rate

Fault Tolerance

- **Reactive way**
  - Worker failure
    - Heartbeat, Workers are periodically pinged by master
      - NO response = failed worker
    - If the processor of a worker fails, the tasks of that worker are reassigned to another worker.
  - Master failure
    - Master writes periodic checkpoints
    - Another master can be started from the last checkpointed state
    - If eventually the master dies, the job will be aborted

Fault Tolerance

- Proactive way (**Redundant Execution**)
  - The problem of “stragglers” (slow workers)
    - Other jobs consuming resources on machine
    - Bad disks with soft errors transfer data very slowly
    - Weird things: processor caches disabled (!!)
  - When computation almost done, reschedule in-progress tasks
  - Whenever either the primary or the backup executions finishes, mark it as completed

Fault Tolerance

Input error: bad records

- Map/Reduce functions sometimes fail for particular inputs
- Best solution is to debug & fix, but not always possible
- **On segment fault**
  - Send UDP packet to master from signal handler
  - Include sequence number of record being processed
- **Skip bad records**
  - If master sees two failures for same record, next worker is told to skip the record

Hadoop : software platform originally developed by Yahoo enabling users to write and run applications over vast distributed data.

Attractive Features in Hadoop :

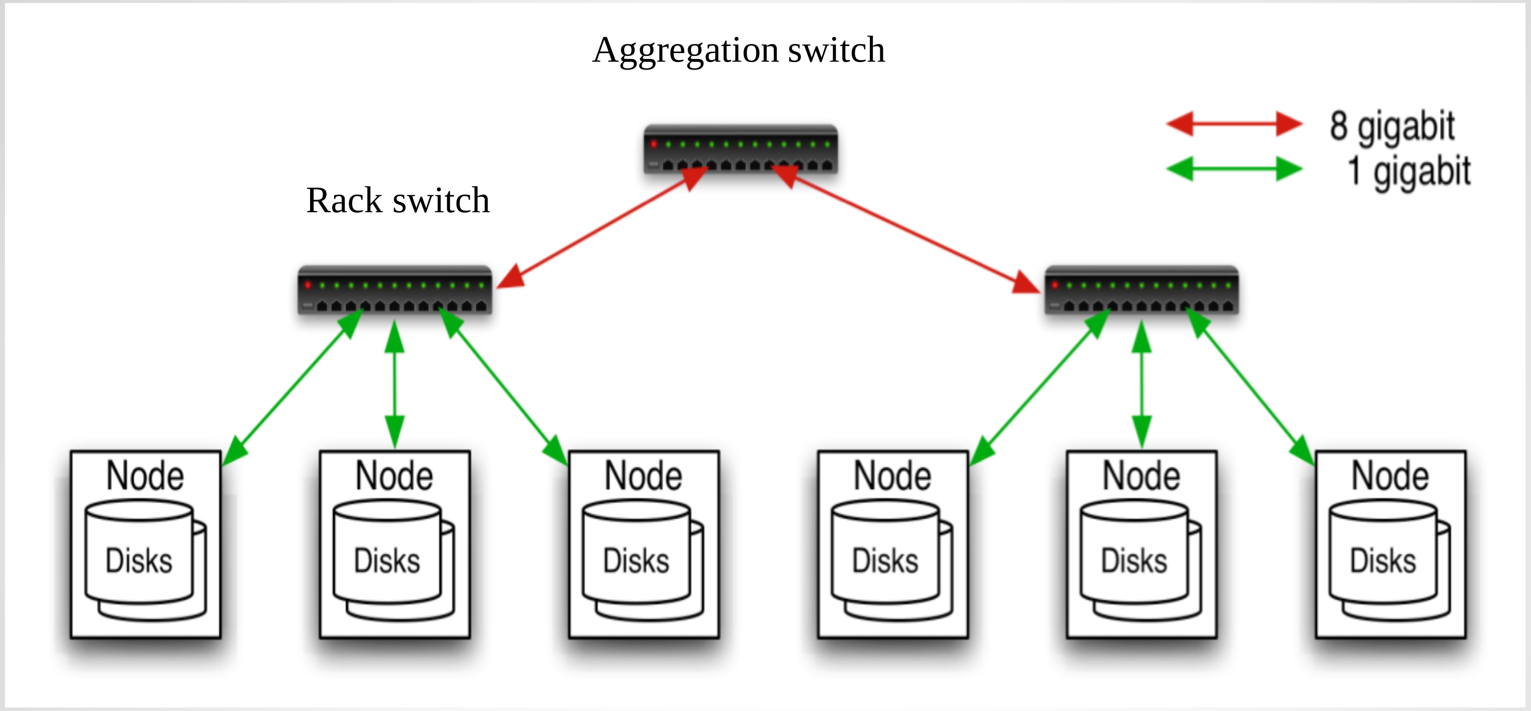
☞ Scalable : can easily scale to store and process petabytes of data in the Web space

☞ Economical : An open-source MapReduce minimizes the overheads in task spawning and massive data communication.

☞ Efficient: Processing data with high-degree of parallelism across a large number of commodity nodes

☞ Reliable : Automatically maintains multiple copies of data to facilitate redeployment of computing tasks on failures

Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- Node specs (Yahoo terasort):
  8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Typical Hadoop Cluster

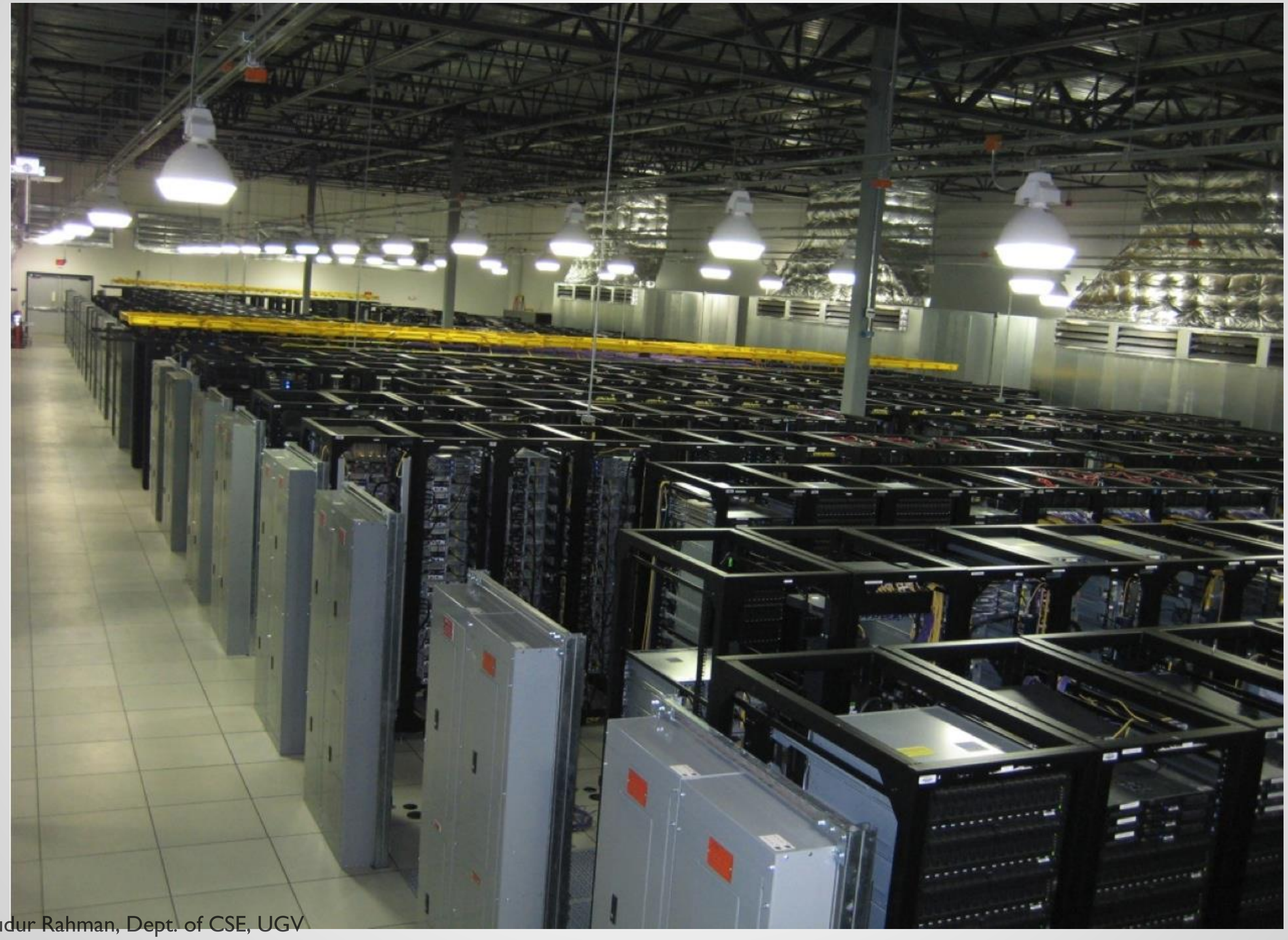


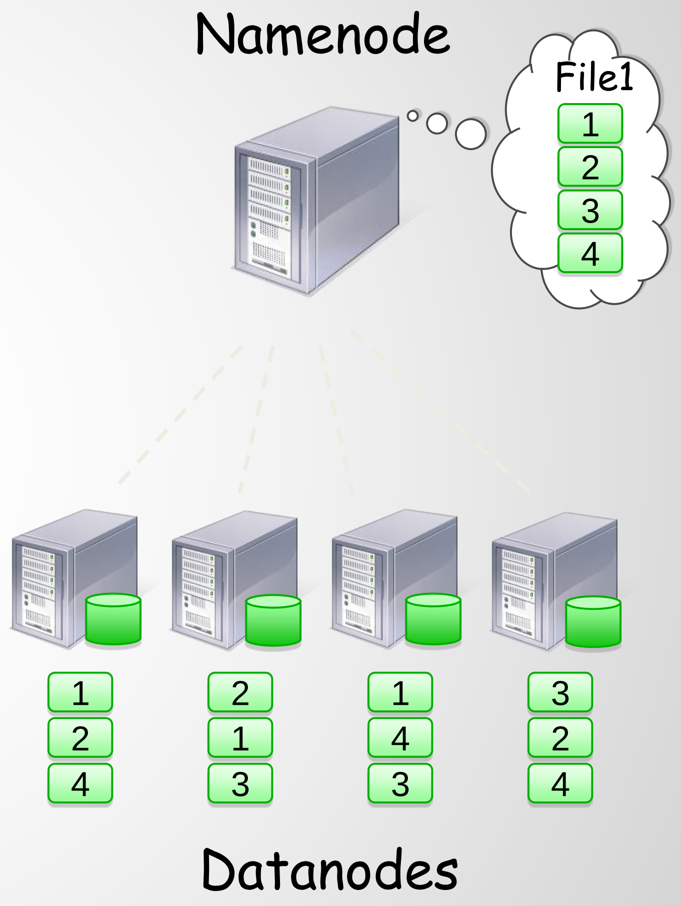
Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf

Hadoop Components

- Distributed file system (HDFS)
  - ➢ Single namespace for entire cluster
  - ➢ Replicates data 3x for fault-tolerance
- MapReduce framework
  - ➢ Executes user jobs specified as “map” and “reduce” functions
  - ➢ Manages work distribution & fault-tolerance

Hadoop Distributed File System

- Files split into 128MB *blocks*
- Blocks replicated across several *datanodes* (usually 3)
- Single *namenode* stores metadata (file names, block locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



HDFS and MapReduce Architecture

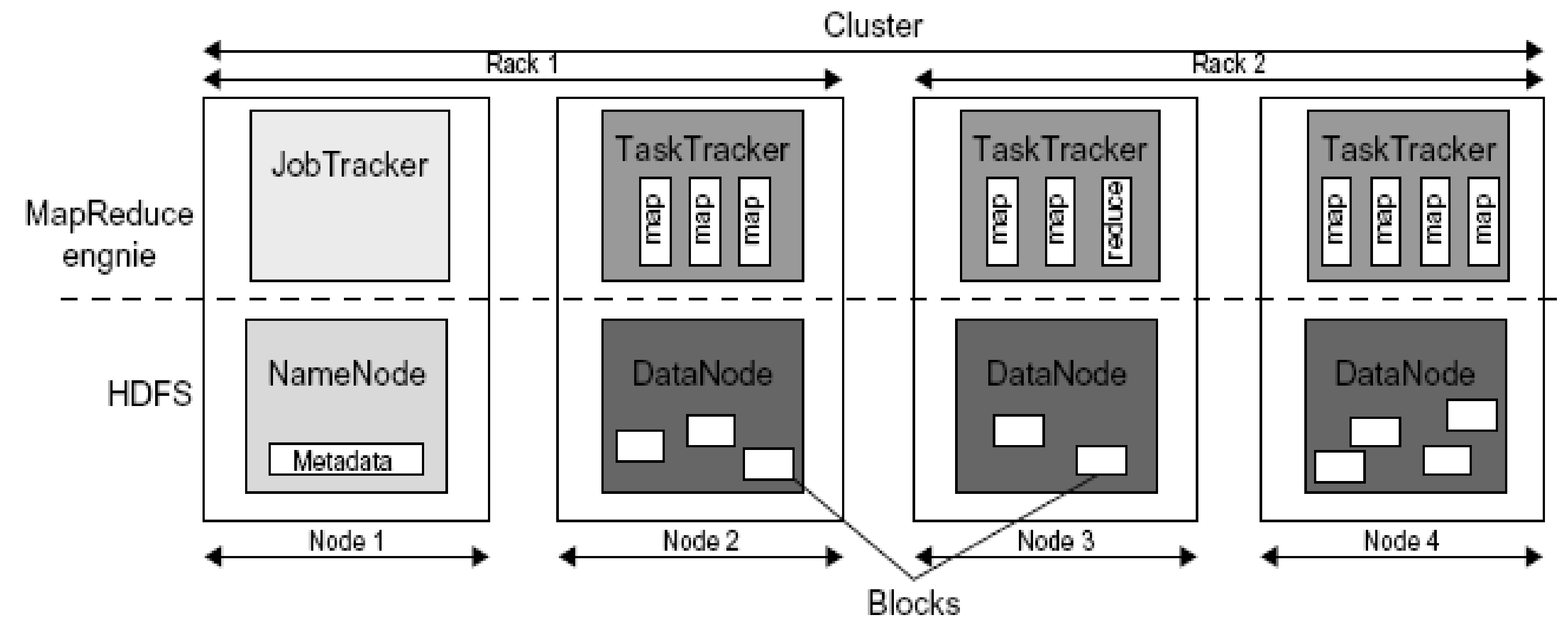
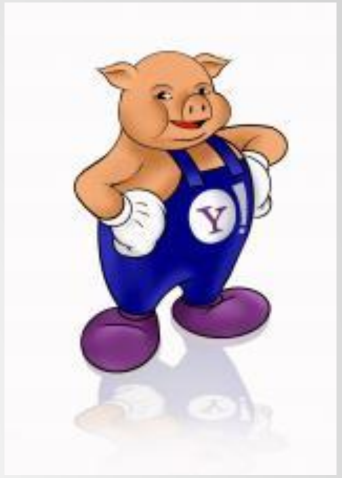


FIGURE 6.11

HDFS and MapReduce architecture in Hadoop.

Higher-level languages over Hadoop: Pig and Hive

Pig :



- Started at Yahoo! Research
- Runs about 30% of Yahoo!'s jobs
- Features:
  - Expresses sequences of MapReduce jobs
  - Data model: nested "bags" of items
  - Provides relational (SQL) operators (JOIN, GROUP BY, etc)
  - Easy to plug in Java functions
  - Pig Pen development environment for Eclipse

Hive

- Developed at Facebook
- Used for majority of Facebook jobs
- “Relational database” built on Hadoop
    - ➢ Maintains list of table schemas
    - ➢ SQL-like query language (HQL)
    - ➢ Can call Hadoop Streaming scripts from HQL
    - ➢ Supports table partitioning, clustering, complex data types, some optimizations

Amazon Elastic MapReduce

- Provides a web-based interface and command-line tools for running Hadoop jobs on Amazon EC2
- Data stored in Amazon S3
- Monitors job and shuts down machines after use
- Small extra charge on top of EC2 pricing
- If you want more control over how you Hadoop runs, you can launch a Hadoop cluster on EC2 manually using the scripts in src/contrib/ec2

Conclusions

- MapReduce programming model hides the complexity of work distribution and fault tolerance

- Principal design philosophies:
  - ➢ ***Make it scalable***, so you can throw hardware at problems
  - ➢ ***Make it cheap***, lowering hardware, programming and admin costs
- MapReduce is not suitable for all problems, but when it works, it may save you quite a bit of time

- Cloud computing makes it straightforward to start using Hadoop (or other parallel software) at scale