innovate achieve lead

BITS Pilani

WILP

# *AIML CLZG516*
# *ML System Optimization*

Session 2

## Parallel Programming Models:

- Pipe-lined, Data-Parallel, Task-Parallel, and Request-Parallel
- Speedup

## Add two numbers:-

(index / memory locn.

a = memory [100].

b = memory [101]

c = a+b

memory[102] = c.

## RAM

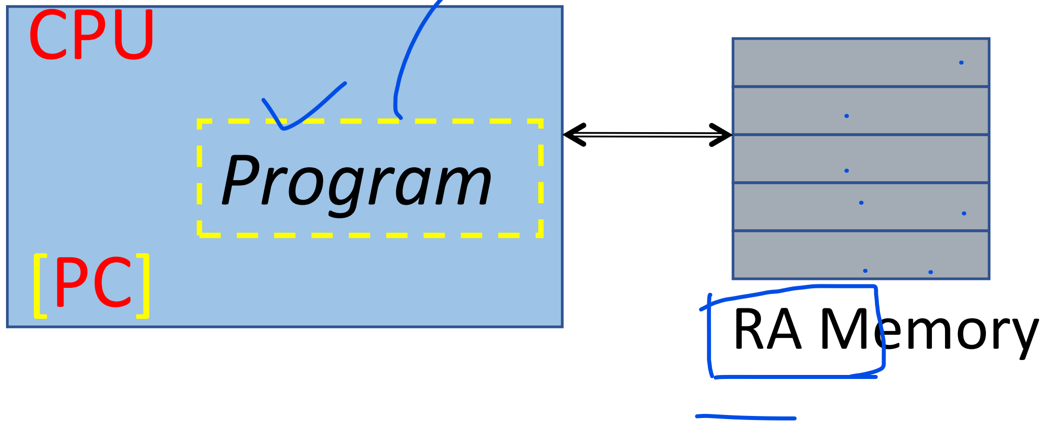
```
LOAD R1, [100].
LOAD R2, [101]
ADD R3, R1, R2
STORE R3, [102]
HALT.
```

# Algorithm Design - Sequential

+, − , AND , OR .
MOD .

- Generic Machine Model
  - Random Access Machine Model

instructions + data .



Typical Instructions

- Arithmetic / logic operations,
- Load / Store, and
- Jump / Branch

**PC**: Program Counter
*(tracks the next instruction to be executed)*

**RAM**: Random Access Memory
(*cost of access is uniform across locations*)

# Executing an Instruction

- Different stages of Instruction Execution:

| | |
|---|---|
| Fetch Instruction | *Move the next instruction (tracked by PC) to a register* |
| Decode Instruction | *Identify Operator and (data) addresses* |
| Load (data) | *Move data into register (if needed)* |
| Execute | *Perform the operation* |
| Store (result) | *Move the resulting data to memory* |

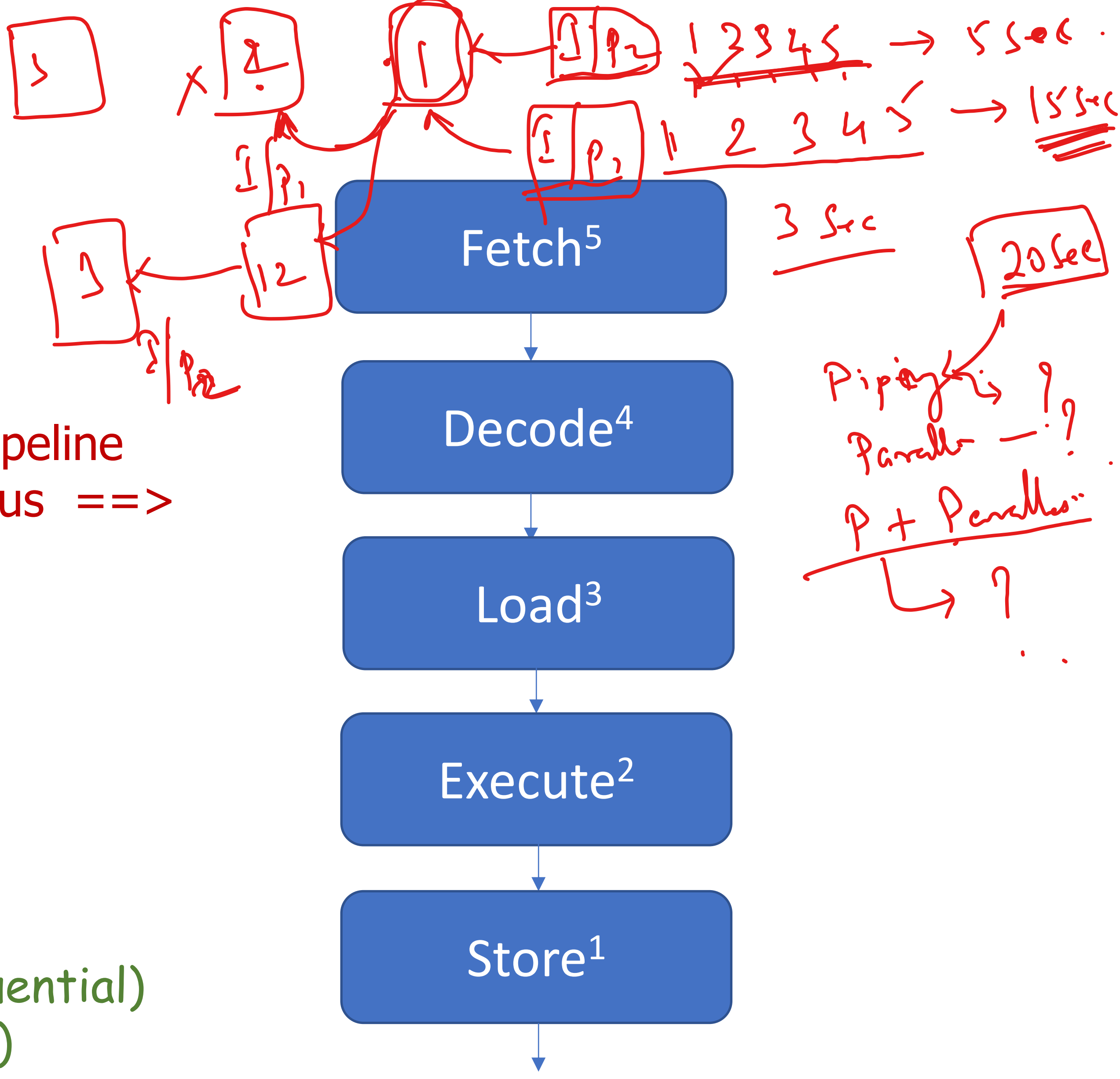
If separate circuitry is designed for each stage-
so that the stage take the same amount of time -
then a sequence of instructions can be executed in a pipeline

# Instruction Pipeline

Given a sequence of instructions of the form:

- I1
- I2
- I3
- I4
- I5
- …

execution in a pipeline would appear thus ==>



If each stage takes 1 clock cycle, then throughput has increased :

- from 1 instruction per 5 cycles (sequential)
- to 1 instruction per 1 cycle (pipelined)

Q: What about Turn-around-Time aka response time?

## Modern Processors

- Modern processors (since Intel Pentium circa 1991)
  - typically include a pipeline that is several stages (>5) deep
- Throughput in processors is measured in CPI (or Cycles Per Instruction):
  - For an ideal pipeline design: CPI is 1
  - In practice, it may be more (Why?)
    - but on an average it is kept close to 1

## Pipeline Throughput

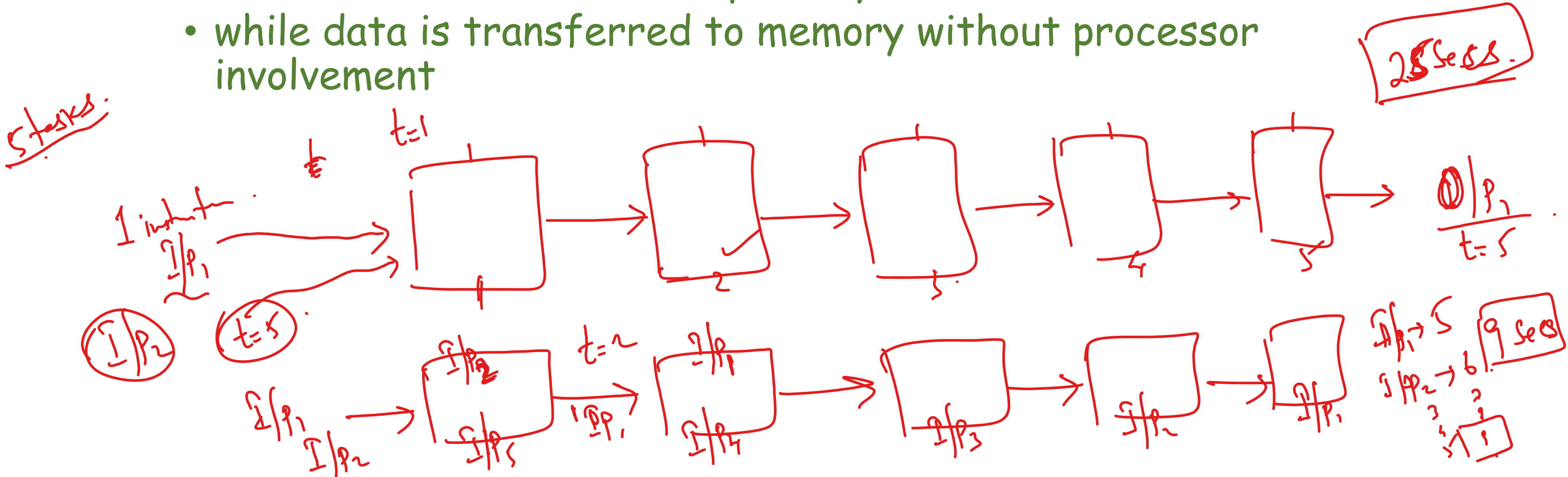
- Speedup (i.e. throughput increase) is k for a k-stage pipeline
- Factors that may slow down the pipeline:
  - Some stage(s) take more time than others
    - Q: What is the impact on CPI if one stage takes 10% extra time?
  - Memory access takes more time (i.e., LOAD and STORE)
- Modern processor pipelines are designed
  - such that all stages take almost the same time (except for LOAD and STORE)

## Pipeline Throughput and Memory Access [2]

- Memory access is slower compared to Processor speed:
  - Typical processor clock cycles
    - e.g. 2 to 3 GHz
    - i.e., in-processor operation may take only 0.33 to 0.5ns
  - Access from Memory (DRAM) will take around
    - 50 to 200ns
  - Access from Cache (SRAM) – if available – may take
    - 5 to 10ns.
- Modern architectures use multiple levels of caching and other techniques to keep the access time low.
- Compiler and processor collaborate to keep the frequency of memory access operations low.

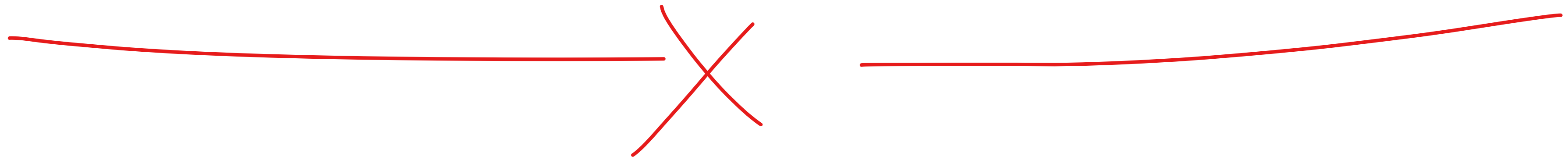
## Pipeline Throughput and Memory Access

- STORE operations may be executed *asynchronously*:
  - i.e. processor does not wait for data to be stored in memory
    - Store buffers (i.e., buffer registers inside the processor) are used to store the data temporarily
    - while data is transferred to memory without processor involvement



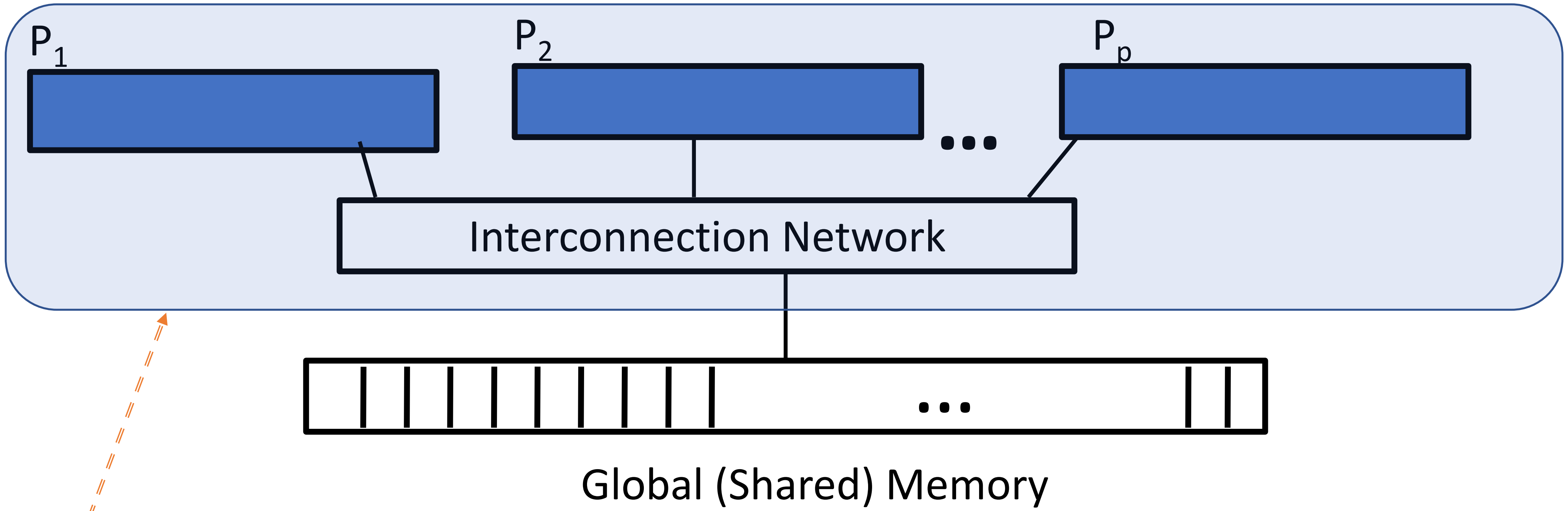
## Software Pipelining

- The idea of a pipeline can be extended to Software Design:
  - Break a long task into multiple stages
    - so that the stages take (roughly) the same amount of time.
  - If there is a stream of data to be processed by the data,
    - then the stream can be fed to the pipeline for improved throughput.
- We will revisit this later!



## Algorithm Design – Parallel: Shared Memory Model

Target environment:



**e.g. a multi-core chip**

Multi-threaded Programming:
*each thread runs on a separate core*

**Typical Instructions**
- Arithmetic / logic operations,
- Load / Store, and
- Jump / Branch

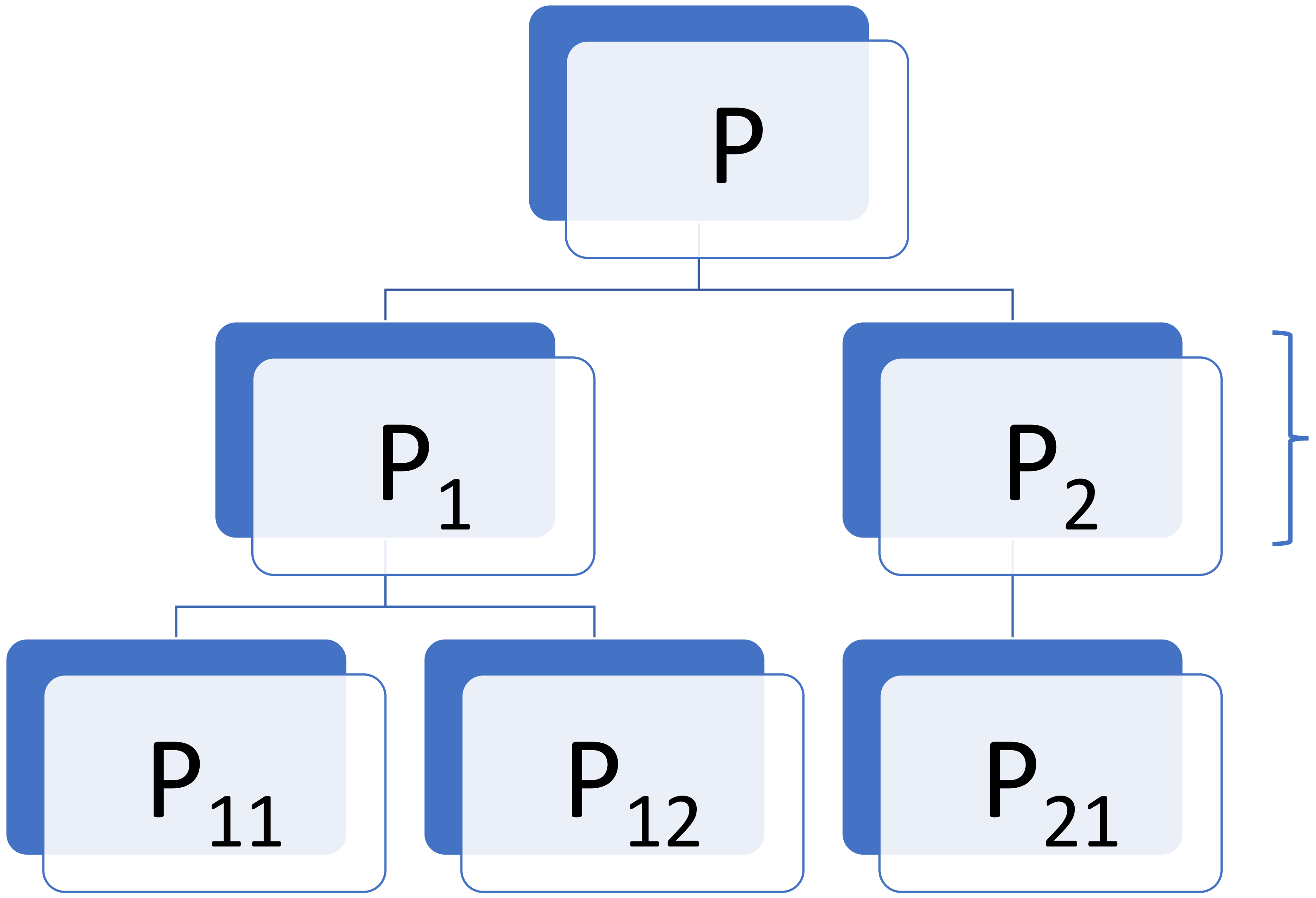
## Algorithm Design

- Top-Down Design (Top Down Decomposition)
  1. Divide the problem into sub-problems.
  2. Find solutions for sub-problems
  3. Combine the sub-solutions.
- How do we find solutions for sub problems?
  - Apply top-down design recursively (i.e. divide each sub-problem further)
    - Q: When do we stop dividing?
    - A: When we reach "atomic" problems.
      - Atomic problems have known solutions
- Does any decomposition work?
  - Divide (the problem) only if you know how to combine (the solutions)

# Top Down Design - Parallel

- Does any decomposition work?
  - Divide (the problem) only if you know how to combine (the solutions)
  - But also:
    - *Mapping sub-problems to processors*
    - *Where is the combination done*?
  - Number of sub-problems?
    - Processor utilization is the key!
      - i.e. More the number of processors more the number of sub-problems!

## Top-Down Design - Parallel



$P_1$ and $P_2$ can run in parallel

$P_{11}$, $P_{12}$ and $P_{21}$ can run in parallel

## Example: Search a key **k** in a list Ls of size **N**

Data: Assume **Ls[0..N-1] is stored in shared memory**

t is N/p

```
for processor $P_j$ from j = 0 to p-1
        do $res_j$ = search(k, Ls[j*t..(j+1)*t-1])

for processor $P_0$ :
do res = TRUE ;
for j = 0 to p-1 do res = res AND $res_j$
```

*This is an example of data parallel programming!*

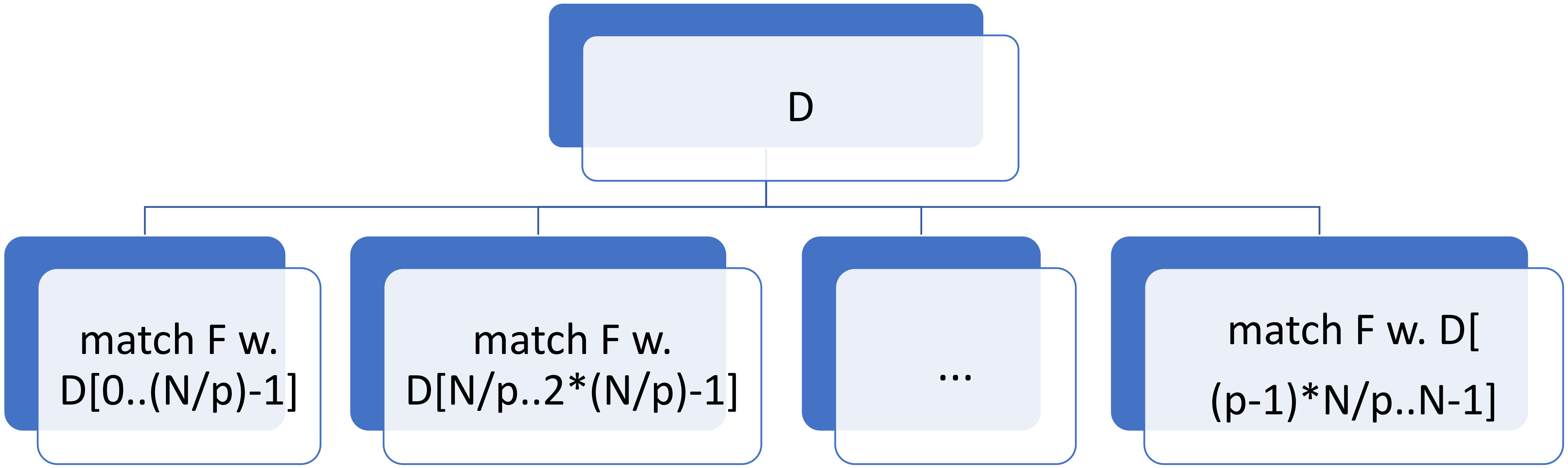
## Data Parallel

- Data Parallel execution (or computation):
  - The same task executes independently (i.e., in parallel) on different data
    - i.e., divide given data into (roughly) equal-sized subsets
      - and the same task is replicated and run on different processors – one for each subset.
- Note that we are assuming a shared memory model
  - i.e., all processors can access the (global) shared memory
    - Dividing data may simply become setting (boundary) markers!
- This may result in memory contention:
  - i.e., performance may not scale (with number of processors)

## Data Parallel Execution - Example

Fingerprint Matching:

• *Match a given print F with a database D of prints available*



# Data Parallel Execution - Exercise

- Vector Product A.B for two vectors – each of length N
- $\Sigma_{j=1 \text{ to } N} A[j]*B[j]$
- N processors:
  - For each processor j=1 to N do: A[j]*B[j]
- How to do the addition? Can it be done in data-parallel fashion?
- p processors: Change the code!

# ✓SPMD

- Data-Parallel execution is also referred to as
  - Single Program Multiple Data (SPMD) programming (because a single program i.e. the same program) is executed on all processors
- This model Data-Parallel or SPMD is preferred where feasible
  - because of ease of programming and efficiency.
- In the parallel programming world, efficiency is measured as ***speedup***:
  - i.e., the ratio of time taken by a parallel algorithm to time taken by a sequential algorithm

## Speedup

- Speedup (in running time) of a given algorithm *A* running on **p** processors is defined as:
  - Speedup(p) = (Time taken by *A* on 1 processor) / (Time taken by *A* on p processors)
- All parts of a program may not run independently or in parallel :
  - Memory contention
  - Data (structure) contention
  - Mutually exclusive access (e.g. update operations or transactions) of shared data
  - Data dependency (result of a task must be input to another)

## ✓ Speedup – Amdahl's Law

- Assume that a fraction **f** of a task is not parallelizable (e.g., due to constraints seen in the last slide)
- **Speedup(p) = 1/(f + (1-f)/p)**
  - i.e., the parallelizable fraction (1-**f**) of the program has been sped-up by a factor of **p**, the number of processors
  - But the other part takes the same (fraction of) time **f**

- By definition, f=0 in data-parallel execution or an SPMD program:
  - and speedup(p) = p
- When speedup(p) is proportional to p, we say that the algorithm is **scalable.**