Classification of C.A. — 2 dimensions (instructions & data) by Michael. Flynn.

1. SISD — Single Instruction & Single data (Sequential)
2. SPMD — Single Program & Multiple data (Data parallel) ←
3. MISD — Multiple instructions & Single data (Pipelined.)
4. MPMD. — " Programs & Multiple data (Task parallel.) ←

✓ Request parallel. (I/O).

Amdahl's Law (theoretical)

1 P ⇒ x mins.
10 P ⇒ x/10 mins

$$\text{max. Speed}_{up} = \frac{1}{f + \frac{(1-f)}{P}}$$

f → Sequential part of the program.
P ← processors.

$f = 0.15$ (data loading, model initialization, model weights / @ checkpoint).

$(1-f) = 0.85$.

processors

$P = 4$

$$= \frac{1}{0.15 + \frac{0.85}{4}} = 2.75$$

$$P = \infty = \frac{1}{f + \frac{(1-f)}{\infty}} = \frac{1}{f+0} = \frac{1}{f}.$$

$\Rightarrow \frac{1}{0.15} = 6.67$

Q.2

1 processor – 240 minutes to complete

8 " – 40 minutes " "

$S = \frac{240}{40} = 6 \times$

$f = ?$

$\frac{240}{40} = \frac{1}{f + \frac{(1-f)}{8}} \Rightarrow 6 = \frac{8}{8f + 1 - f}$

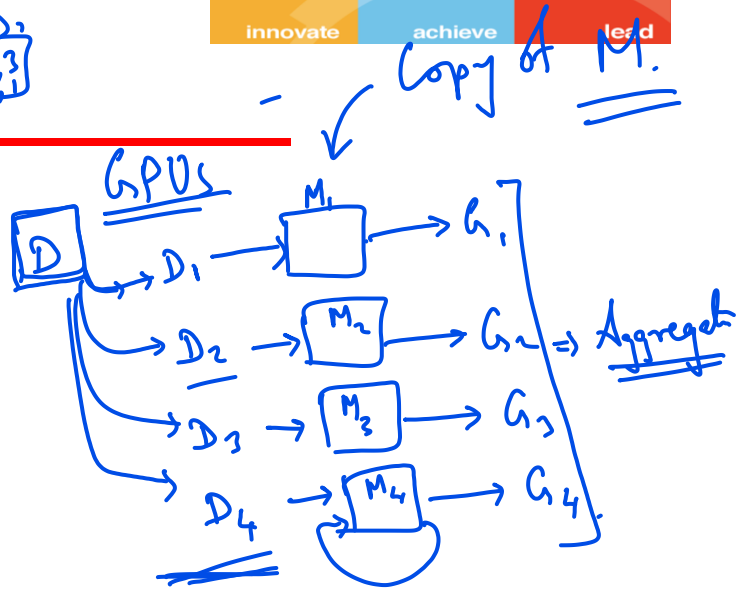
$42f + 6 = 8$

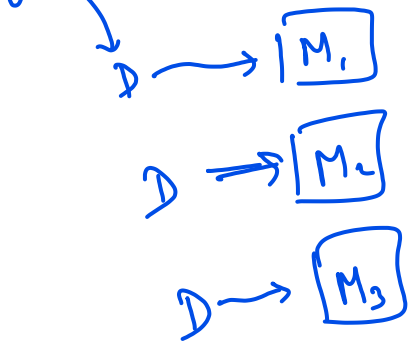
$f = \frac{2}{42} = \frac{1}{21} = 0.0476$

$f = 4.76\%$ Sequential

Sequential parts of ML

1. Data loading
2. Model initialization
3. checkpoint saving.
4. Aggregation – Gradient Synchronization.
5. logging.





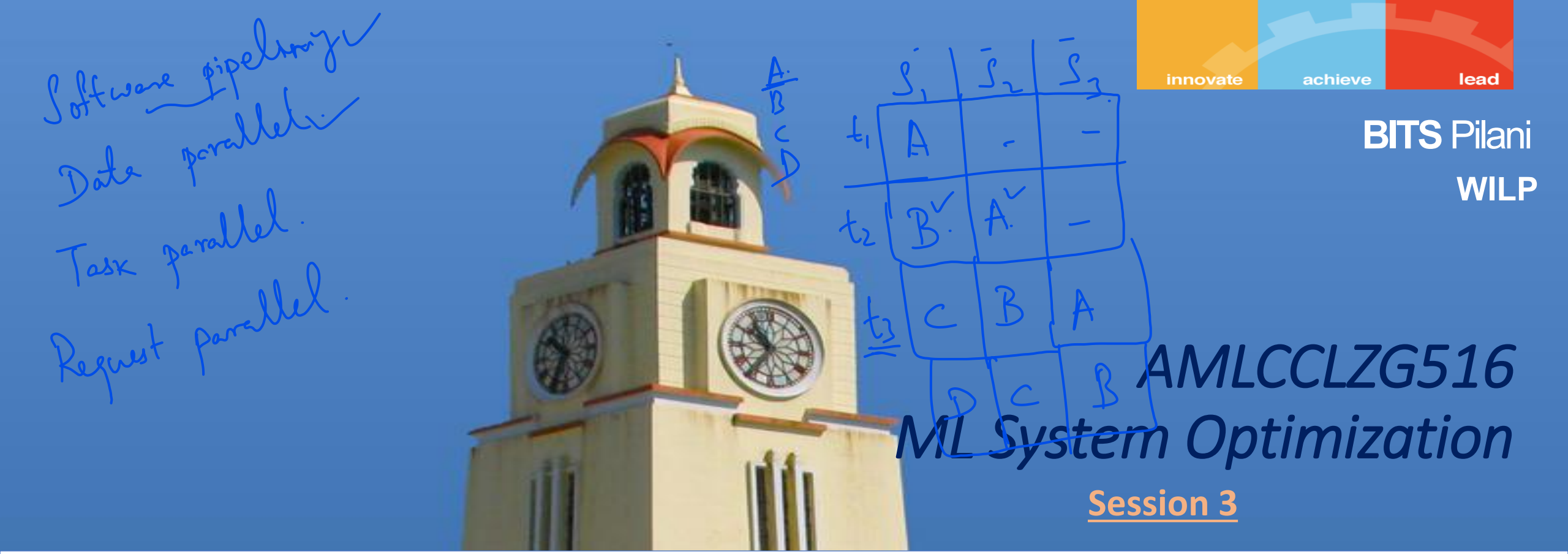
innovate achieve lead

BITS Pilani

WILP

AMLCCLZG516
ML System Optimization

Murali Parameswaran



innovate achieve lead

BITS Pilani

WILP

AMLCCLZG516 ML System Optimization

Session 3

Parallel Programming Models

[continued.]

- Top Down Design
- Map-Reduce Pattern
- Task-Parallel and Request-Parallel

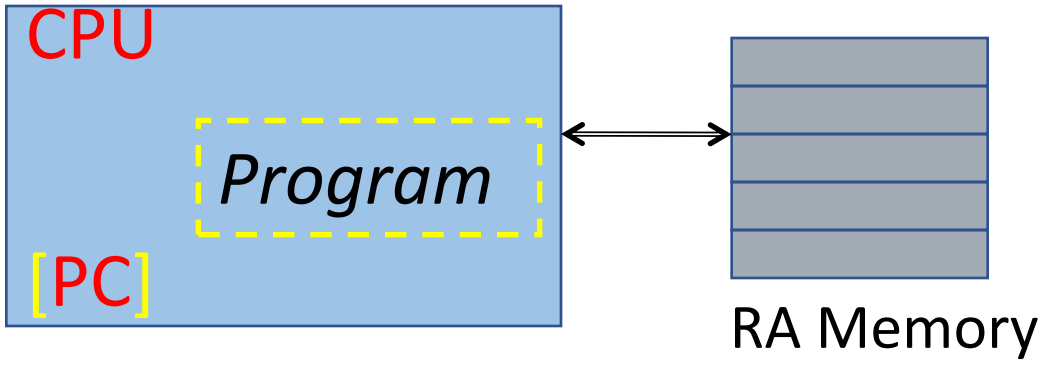
Parallelization of ML Algorithms - Examples

Algorithm Design - Sequential

- Generic Machine Model
  - Random Access Machine Model

Typical Instructions

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



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

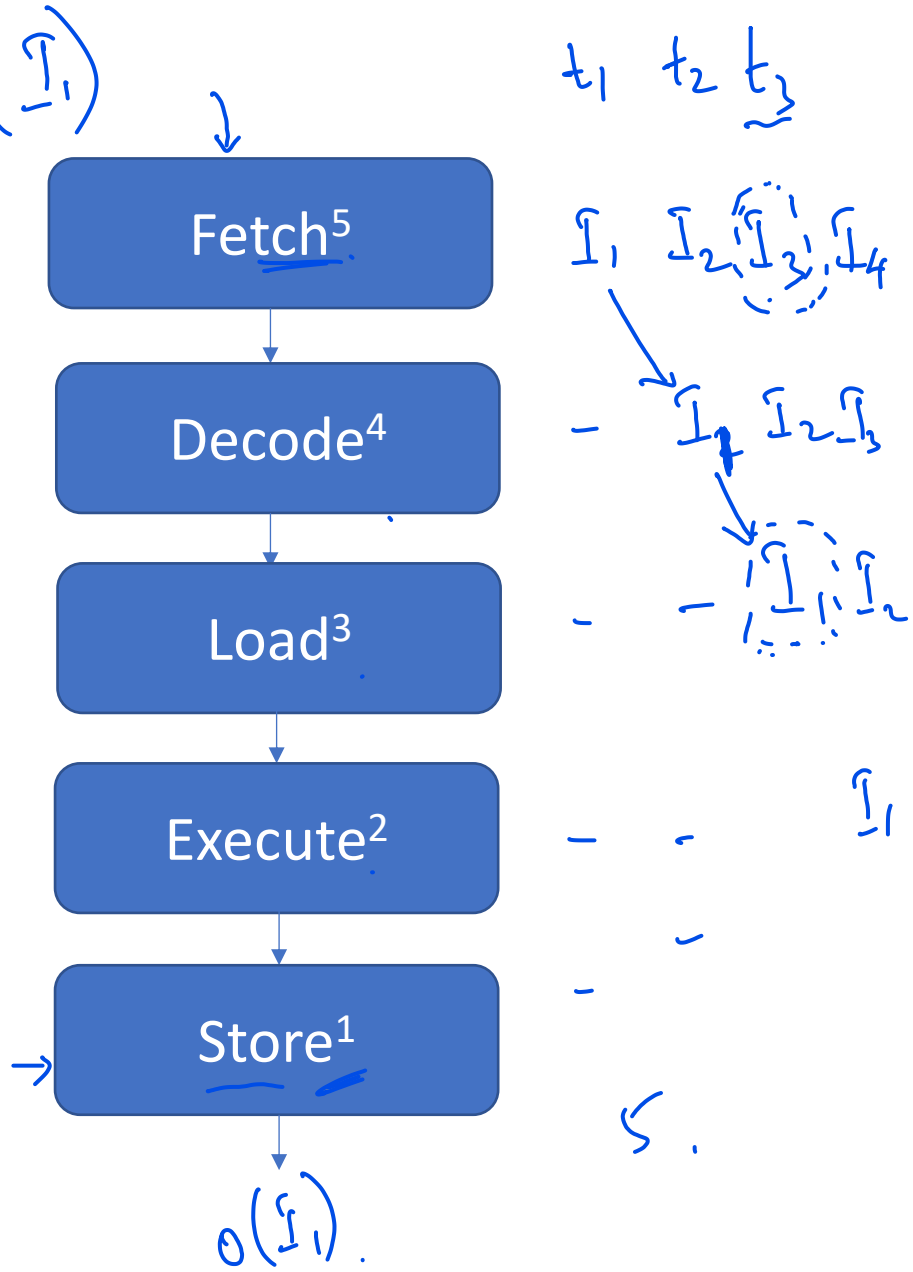
RAM: Random Access Memory
(*cost of access is uniform across locations*)

Instruction Pipeline

Given a sequence of instructions of the form:

- I1
- I2
- I3
- I4
- I5
- ...

execution in a pipeline would appear thus ==>



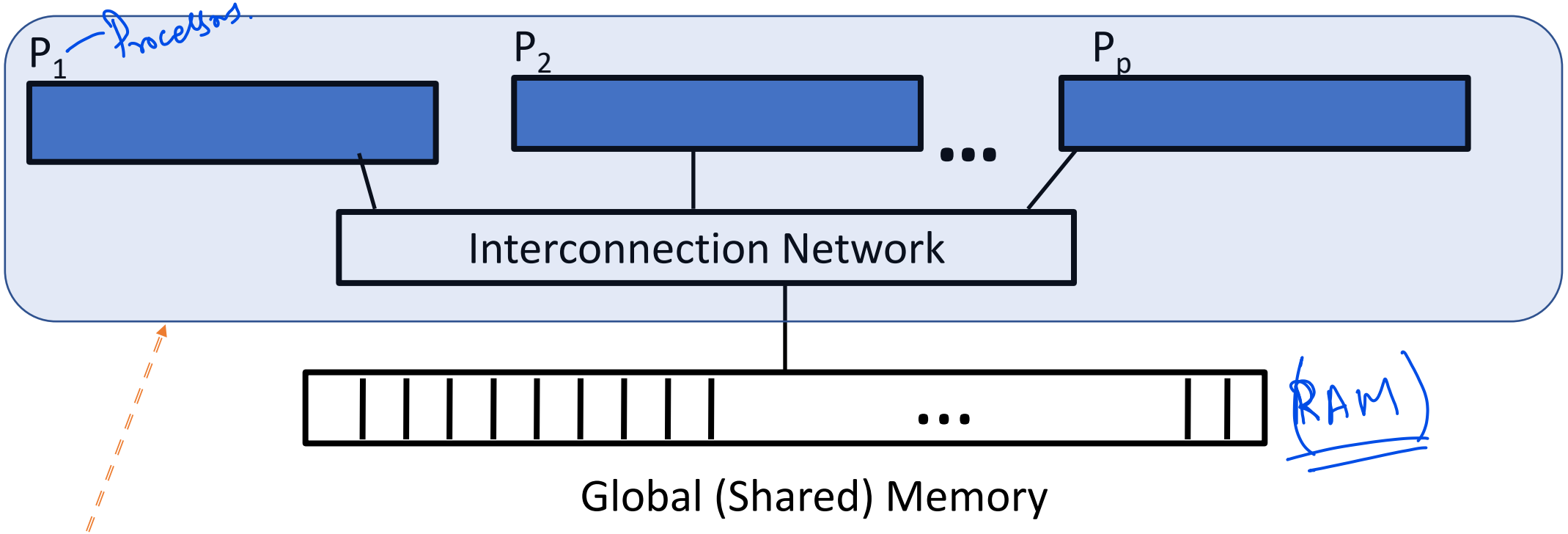
If each stage takes 1 clock cycle, then throughput has increased :

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

Challenge: Data Dependency

Algorithm Design – Parallel: *Shared Memory* Model

Target environment:



e.g. a multi-core chip

Typical Instructions

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

Multi-threaded Programming:

each thread runs on a separate core

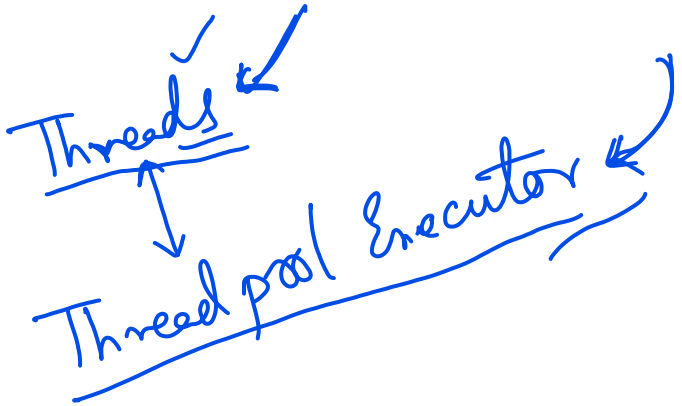
Algorithm Design

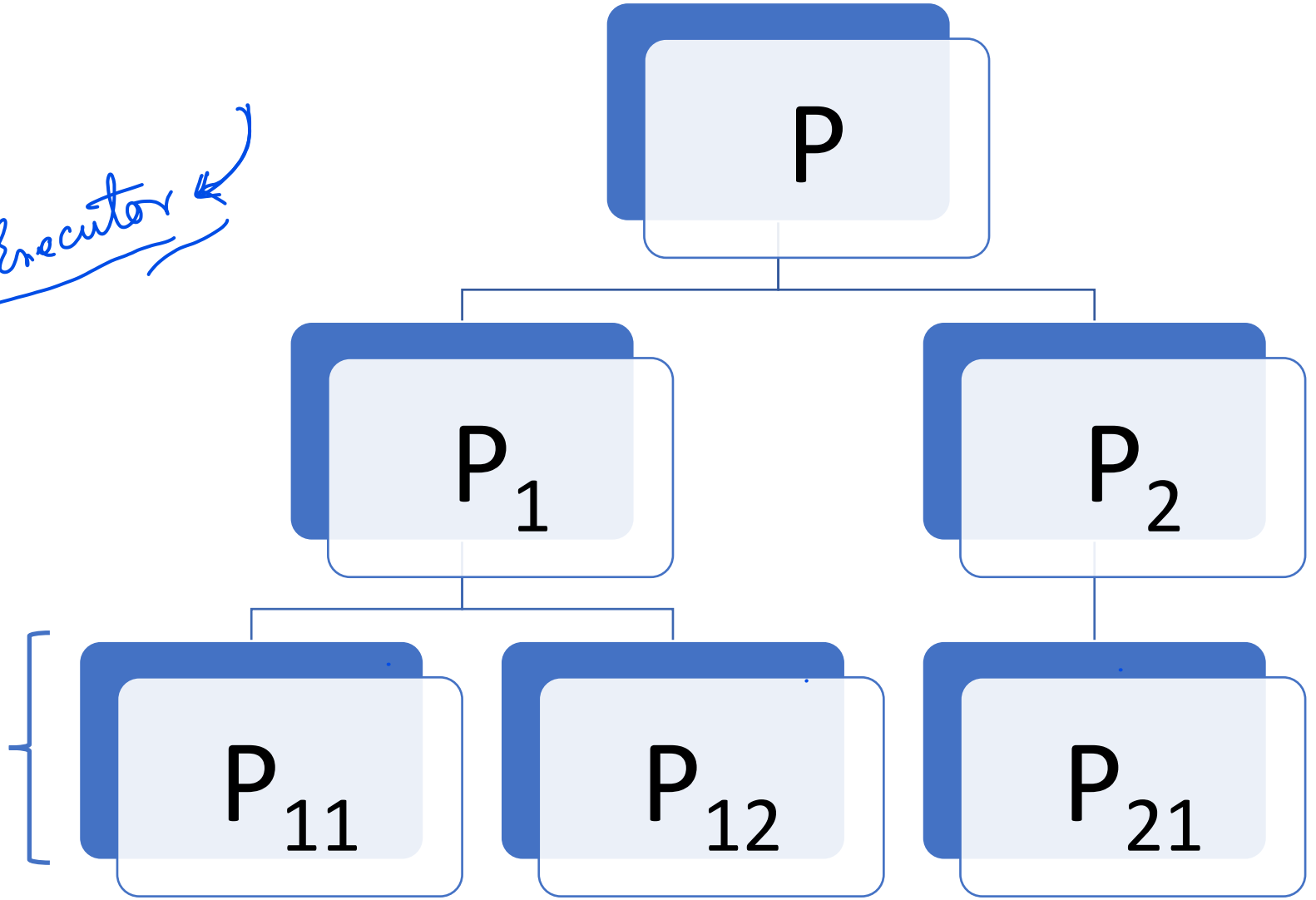
- Top-Down Design (Top Down Decomposition)
  1. Divide the problem into sub-problems.
  2. Find solutions for sub-problems
  3. Combine the sub-solutions.
- How do we find solutions for sub problems?
  - Apply top-down design recursively (i.e. divide each sub-problem further)
    - Q: When do we stop 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!

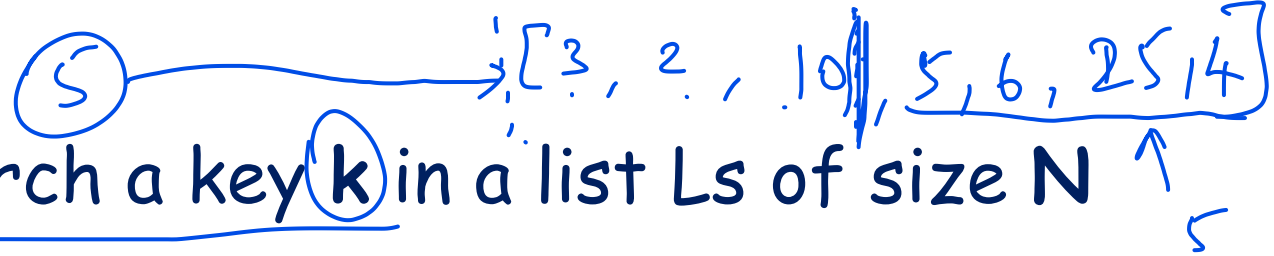
Top-Down Design - Parallel





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

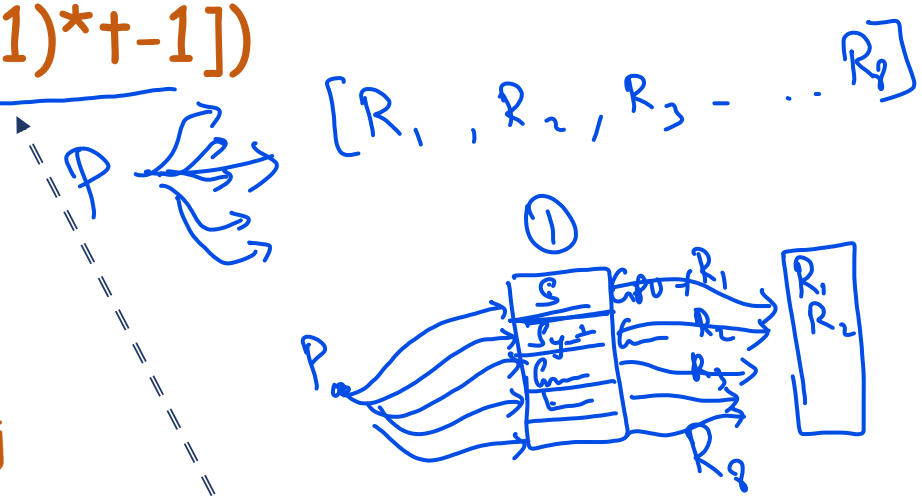
for processor P_j from j = 0 to p-1

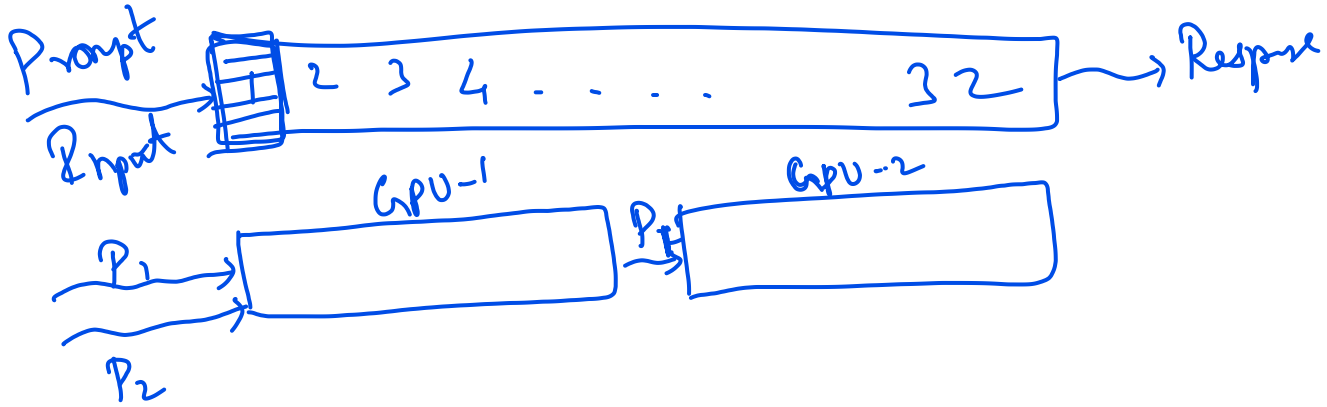
do res_j = search(k, Ls[j*t..(j+1)*t-1])

for processor P_0 :

do res = FALSE ;

for j = 0 to p-1 do res = res OR res_j





This is an example of data parallel programming!

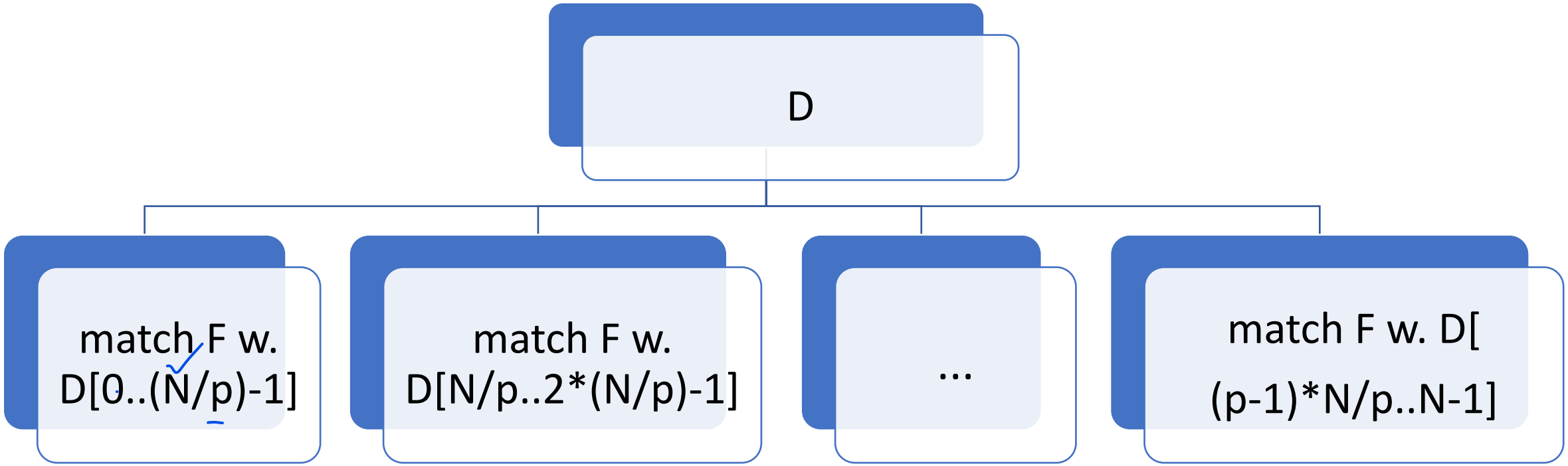
Data Parallel

- 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

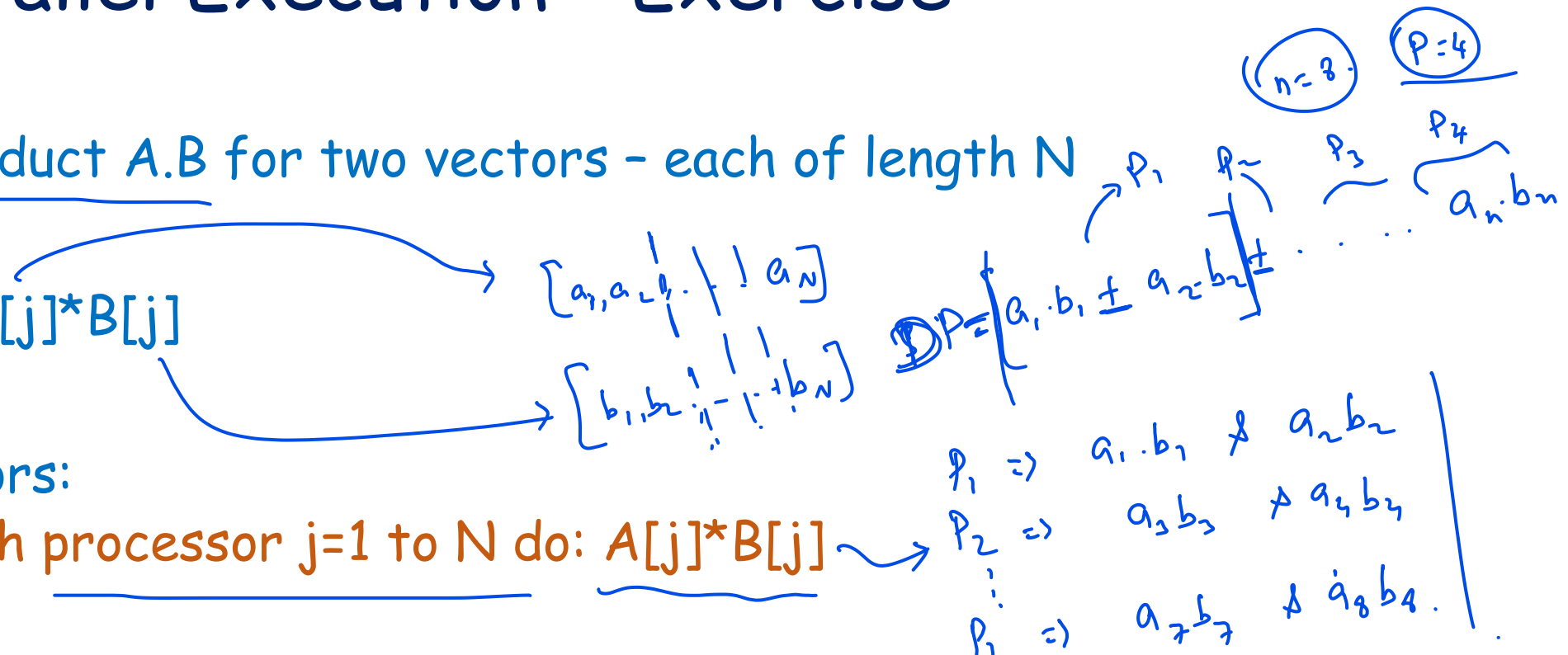
Fingerprint Matching:

- *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.**

innovate achieve lead

BITS Pilani

WILP

AIML CLZG516
ML System Optimization

Session 3: 21 Oct. 2024

Parallel Programming Models

[continued.]

- Map-Reduce Pattern
- Task-Parallel and Request-Parallel

Parallelization of ML Algorithms - Examples

Parallel Design Pattern map

- **map** is a data-parallel programming construct
- **map f Ls** expresses "execute **f** on all elements of **Ls**" using **p** processors
  - Where **Ls** has been distributed among the **p** processors (i.e. their memories)
- **map** is implicitly data parallel

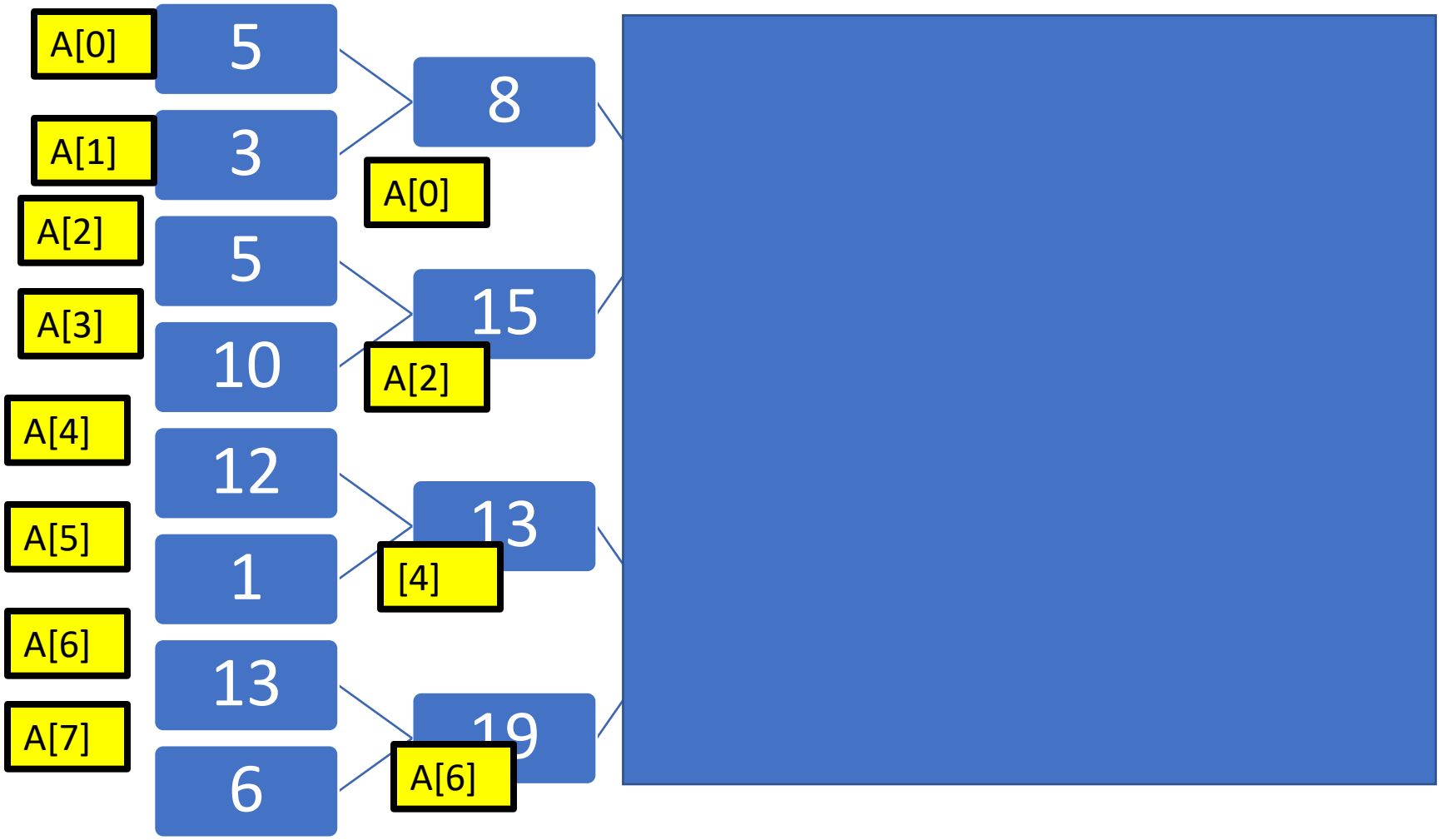
- Exercise:
  - Implement the data-parallel examples (previously discussed)

Parallelization Constraints

- An algorithm may have inherently sequential steps that are not parallelized (or not fully parallelized) naturally.
  - Consider the problem of adding a list of numbers
    - This is an example of
      - (Inverse) Tree Parallelism or
      - The parallel design pattern ***reduce***

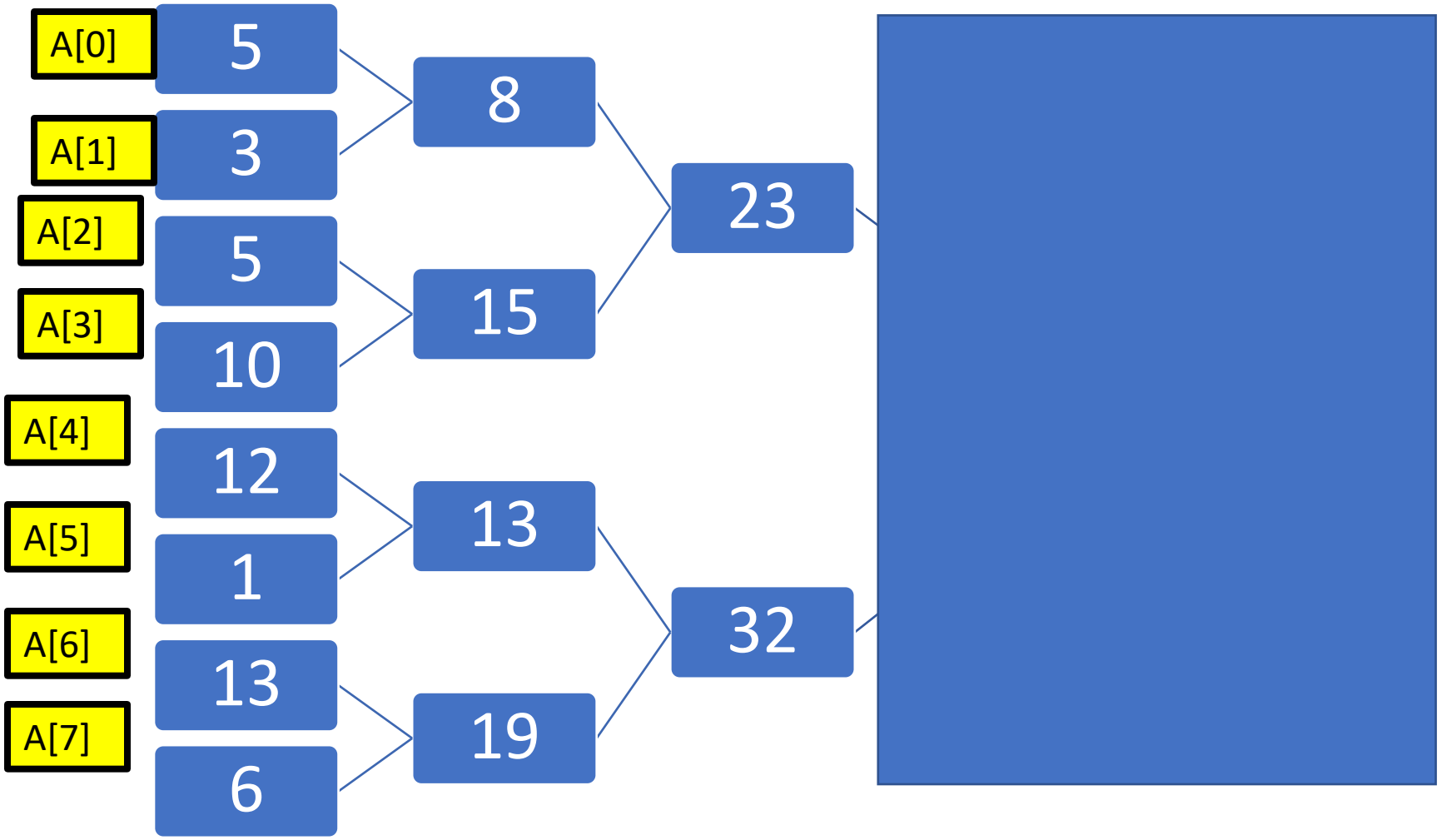
(Inverse) Tree-Parallel Algorithm : Summation

(Inverse) Tree-Parallel Algorithm : Summation



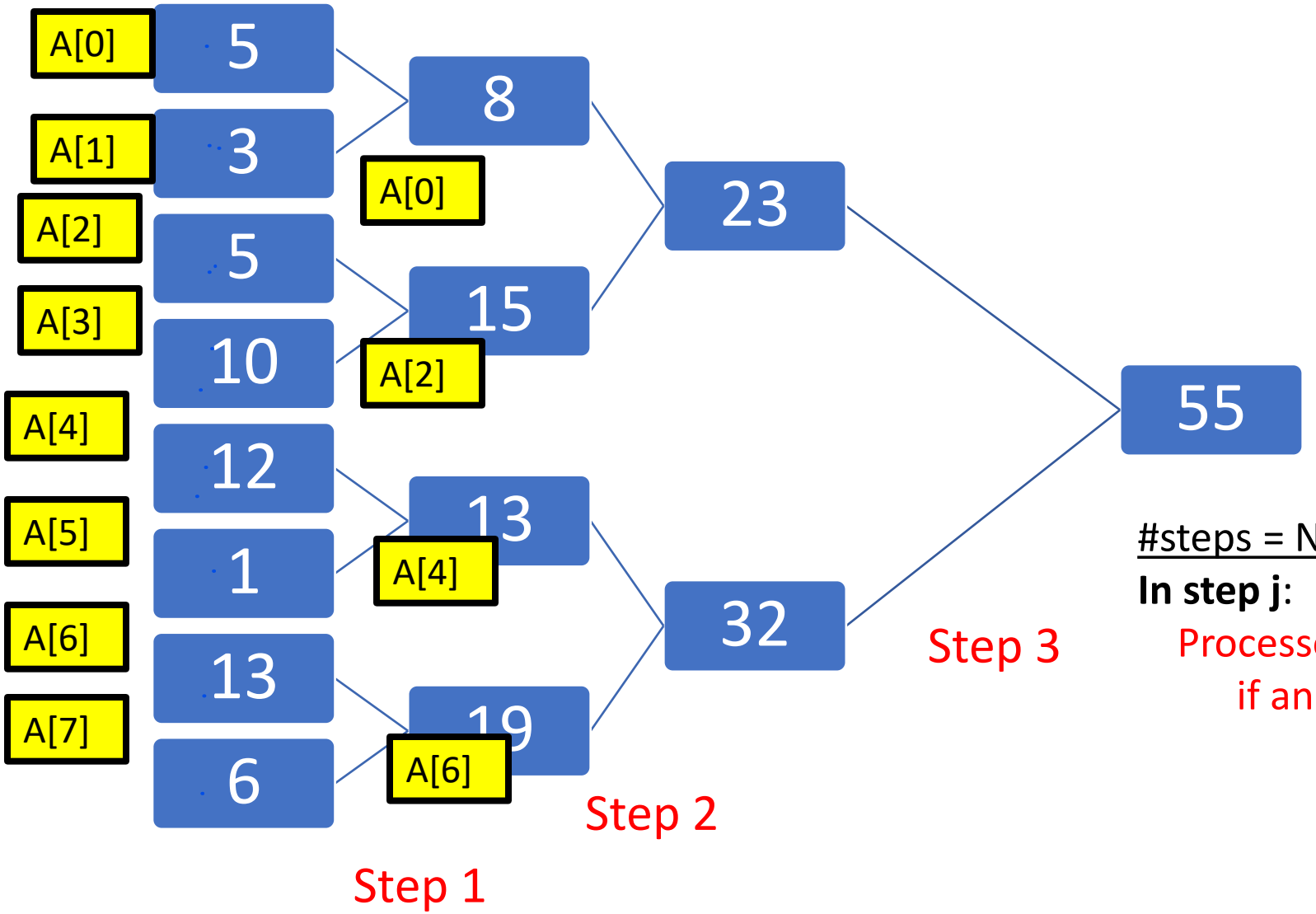
Step 1

(Inverse) Tree-Parallel Algorithm : Summation



Step 2

(Inverse) Tree-Parallel Algorithm : Summation

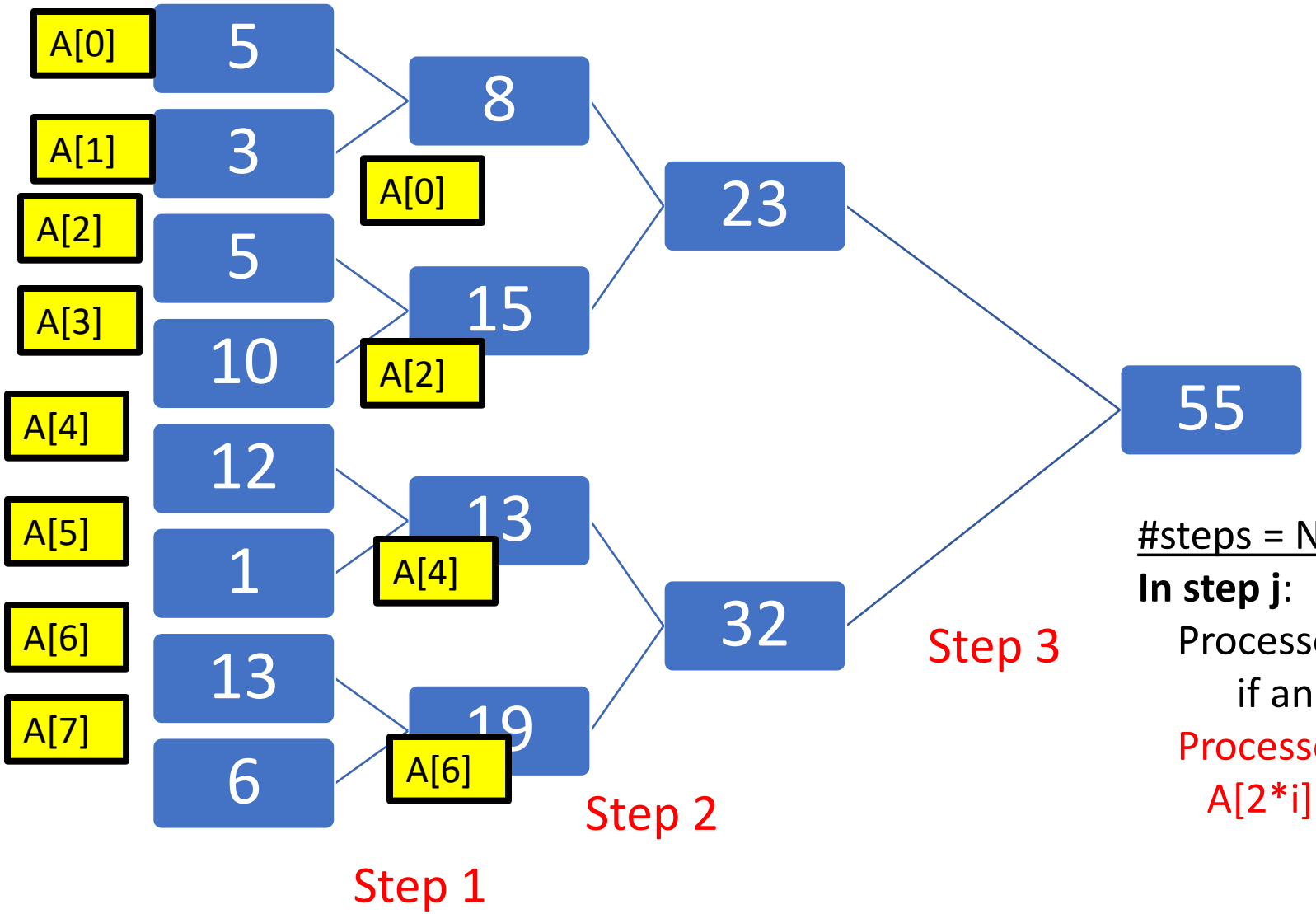


#steps = N for 2^N inputs

In step j:

Processor P_i participates if and only if $i\%2^j == 0$

(Inverse) Tree-Parallel Algorithm : Summation



#steps = N for 2^N inputs

In step j:

Processor P_i participates if and only if $i\%2^j == 0$

Processor P_i does:

$A[2*i] = A[2*i]+A[2*i+2^j]$

(Inverse) Tree-Parallel Algorithm : Summation

- Pre-condition: List A[0..n-1] in global memory
- Post-condition: sum A[0] in global memory
- Global variables: A, n, and j
- begin
  - for all Pi where 0 <= i <= floor(n/2)-1 {
  - for j = 0 to ceil(log(n))-1 {
  - if (i mod 2^j = 0) then
  - A[2*i] = A[2*i] + A[2*i + 2^j]
  - }}
- end

Decide which processors i participate in step j

Distance between self and (processor holding) the other data

(Inverse) Tree-Parallel Algorithm : Summation

- Precondition: List A[0..n-1] in global memory
- Postcondition: sum A[0] in global memory
- Global variables, A, n, and j
- begin
  - spawn (P0, P1, … Pk) where k=floor(n/2)-1
  - for all Pi where 0 <= i <= floor(n/2)-1 {
  - for j = 0 to ceil(log(n))-1 {
  - if (i mod 2^j = 0) and (2*i + 2^j < n) then
  - A[2*i] = A[2*i] + A[2*i + 2^j]
  - }}
- end

Boundary condition when n is not a power of 2

Algorithm : Summation - Performance

- Complexity of the algorithm:
  - Summation requires ceil(log n) steps for n inputs
    - Each step is $O(1)$ time
  - Total time $\theta(\log n)$ given n/2 processors
    - Compare with sequential algorithm
    - Speedup: $T_{seq} / T_{par} = T(n, 1) / T(n,p)$
      - Speedup(n/2) = (n-1) / log(n) = $O(n/\log n)$
        - This is less than ideal speedup!

Parallel Reduction - Template

Template REDUCE

- Precondition: Inputs, G, in global memory
- Postcondition: Result in G[0]
- Global variables: n and j, apart from G
- begin
  - for all Pi where 0 <= i <= floor(n/2)-1 {
  - for j = 0 to ceil(log(n))-1 {
  - if (i mod 2^j = 0) and ($2*i+2^j$ < n) then
  - G[2*i] = G[2*i] OP G[$2*i+2^j$]
  - }}
- end

Example Instances:

- Maximum
- Sum of matrices
- Intersection of sets

Reduce as a construct

- reduce '+ Ls
  - Returns a single value (the sum of all values in Ls)
- Reduce BOP Ls
  - Extends the binary operator BOP over a list of values
- This is valid only if BOP is associative
  - i.e. (x BOP y) BOP z = x BOP (y BOP z)
- Examples
  1. Maximum of a list of values
     - BOP is *max*
  2. Sum of all matrices in a list
     - BOP is *matrix-sum*
  3. *Merge a list of sorted lists*
     - *BOP is (binary) merge*

Speedup(N/2) = N/log(N)

Exercise I: Vector Product

- Problem: Vector Product A.B for two vectors – each of length N
  - $\Sigma_{j=1 \text{ to } N}$ A[j]*B[j]
- Solution:
  - Step 1: N processors:
    - for each processor j=1 to N do: A[j]*B[j]
  - Step 2: N/2 processors:
    - Apply Reduction with + as the operator

This can be achieved using map-reduce:

- Make a list L of (A[j], B[j])
- L1 = map * L
- vp = reduce + L1

Exercise II: Matrix Product

- Problem:
  - Multiply matrices $A_{m \times n}$ and $B_{n \times p}$
- Solution:
  - for each processor $P_{i,j}$ where (i = 1 to m) and (j = 1 to p)
    - C[i][j] = Compute vector product $A[i].B^T[j]$
      - where B^T is the transpose of B:
      - i.e., $B^T[j]$ is the j^{th} column of B

Express this using *map* and *reduce!*

Google's map-reduce framework

- Google's map-reduce has built-in capabilities for:
  - scheduling:
    - i.e., spawn processes depending on available processors
  - load-balancing:
    - i.e., move processes across processors to keep all processors equally utilized
  - fault-tolerance:
    - i.e., restart/resume processes that fail

map-reduce platforms

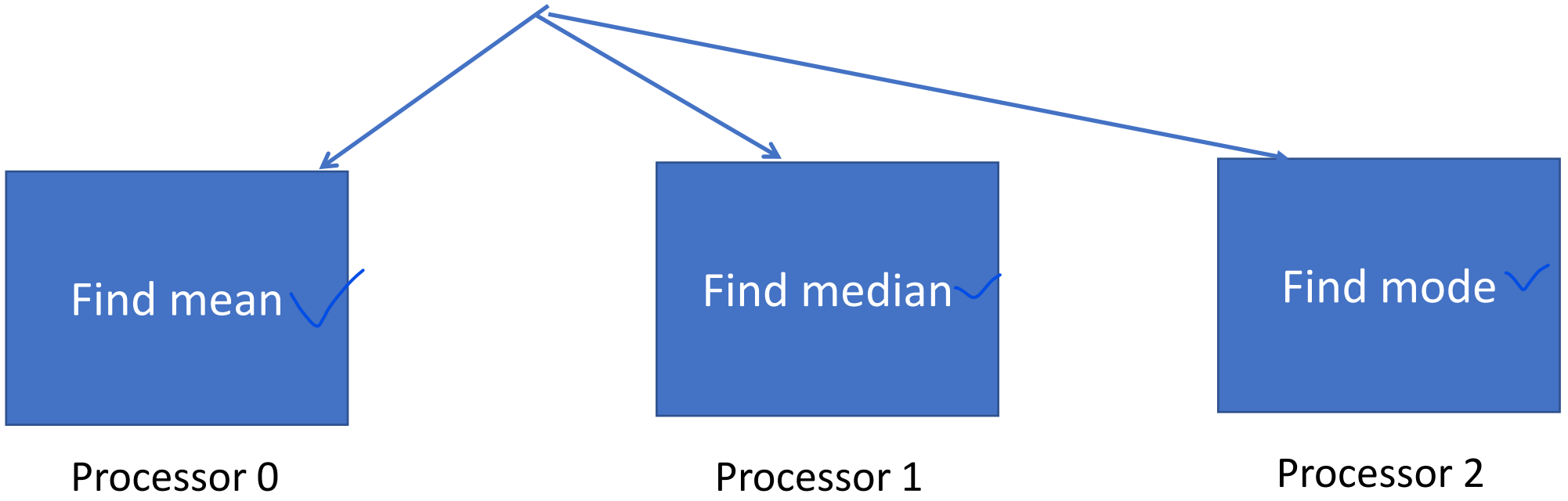
- Map-reduce is supported by different middleware platforms:
  - In particular, **Apache Spark** supports map-reduce on multi-core systems and clusters

- Exercise:
  - Install Apache Spark on your computer and
  - code the *matrix multiplication* example using map-reduce.

Task Parallelism

Task Parallelism – Example

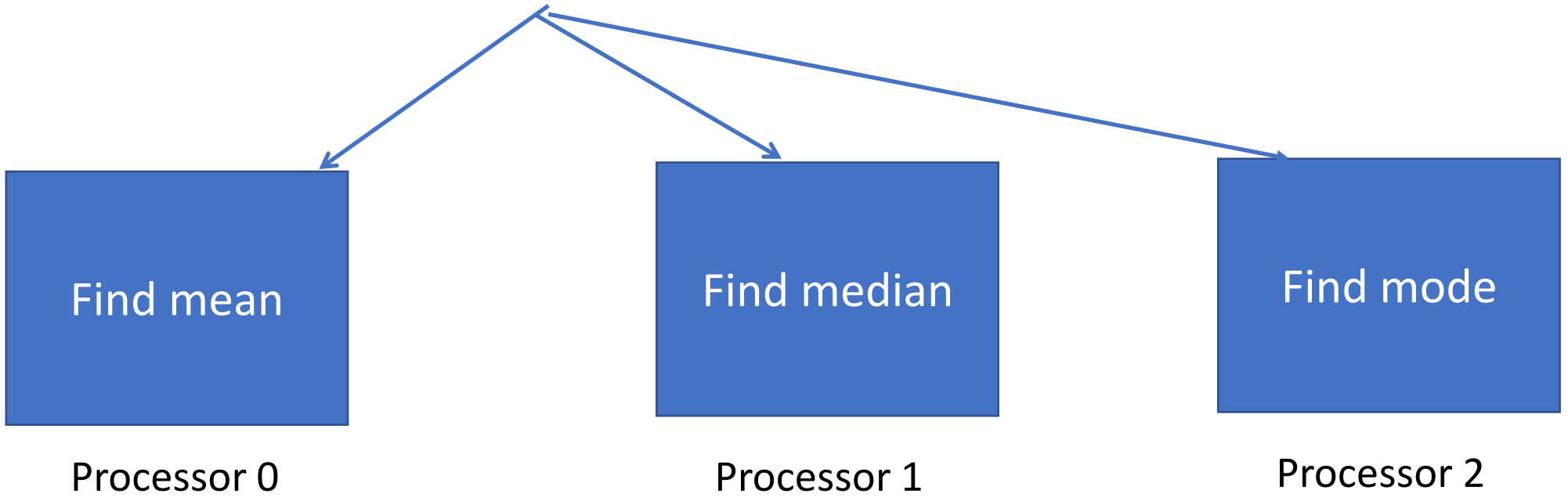
Problem: Given a list Ls of numeric values find the *mean*, *median*, and *mode*.



Task Parallelism – Example

Problem: Given a list Ls of numeric values find the mean, median, and mode.

Speedup = $T_{ser} / T_{par} = (T_{mean} + T_{med} + T_{mode}) / \max(T_{mean}, T_{med}, T_{mode})$

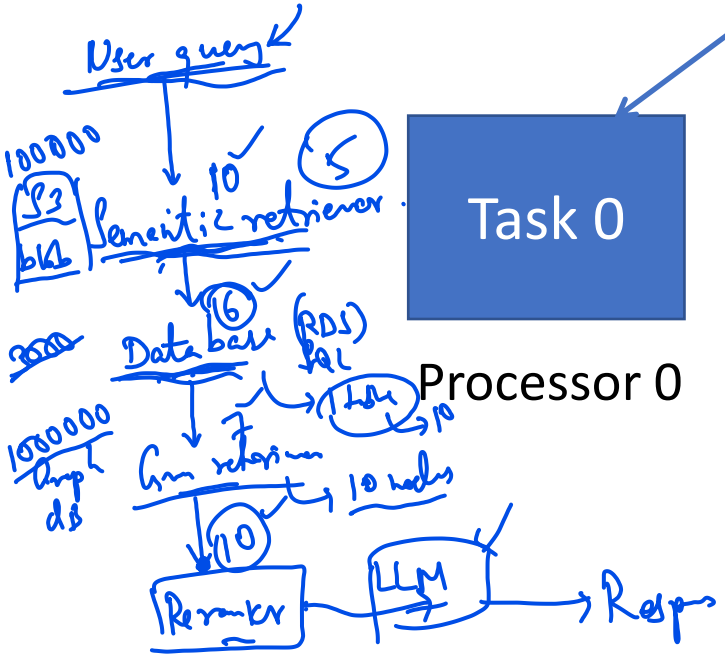


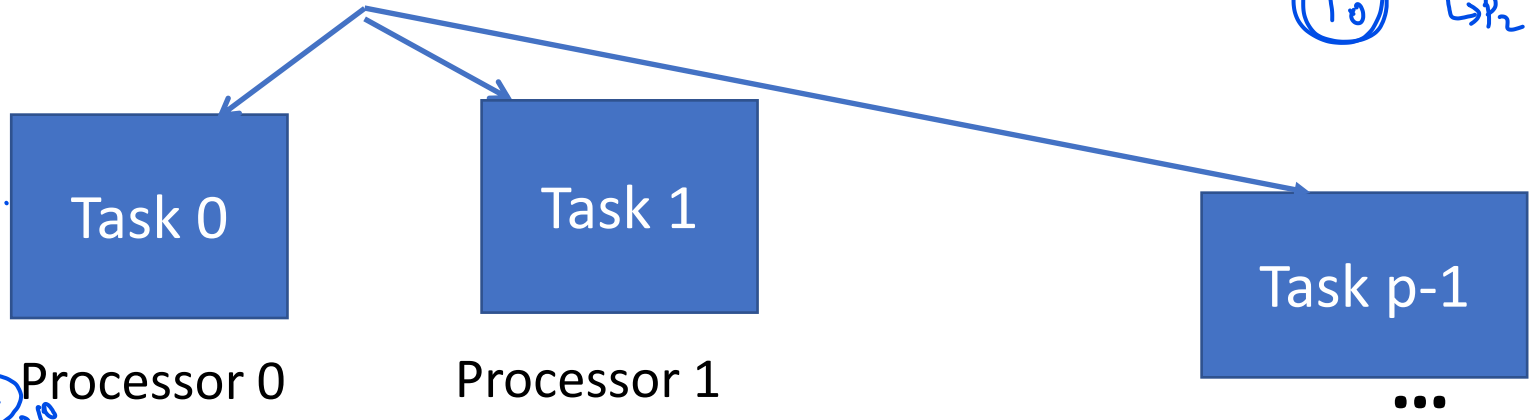
Task Parallelism

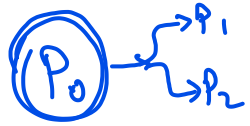
Speedup could be less than p because tasks could be uneven in size.

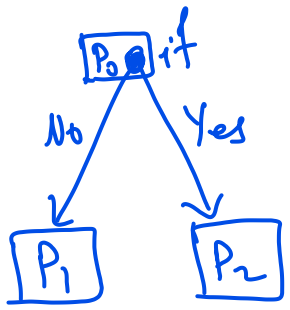
This is not scalable:

- if we put in more processors, we can't get more tasks running in parallel
  - because the number of tasks is fixed and/or small









Task Parallelism

- While task parallelism has speedup limitations, it is suitable for off-the-shelf computers:
  - Modern laptop or desktop computers and workstations are made out of a few (often one) multi-core chip(s):
    - The available parallel processing capacity is limited.

Programming Task Parallelism

- In shared memory computers:
  - Task parallelism can be implemented using multi-threaded programs:
    - One task per thread
      - Where data is stored in shared memory.

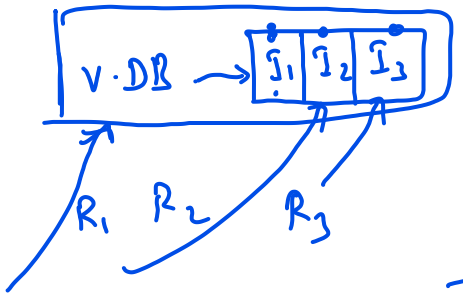
- For instance, in a multi-core system,
  - each thread runs on a separate core.

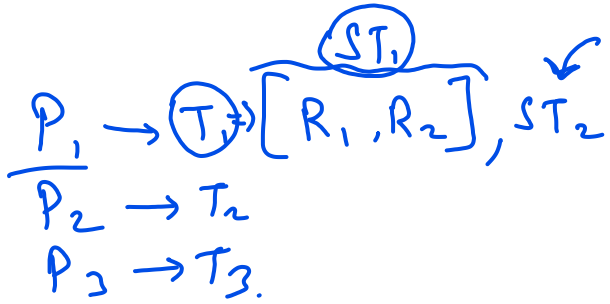
Programming Task Parallelism

- Example and Exercise:
  - Implement the task-parallel computation of mean, median, and mode
    1. using threads in Java
    2. using P-Threads in C/C++
    3. using OpenMP in C/C++

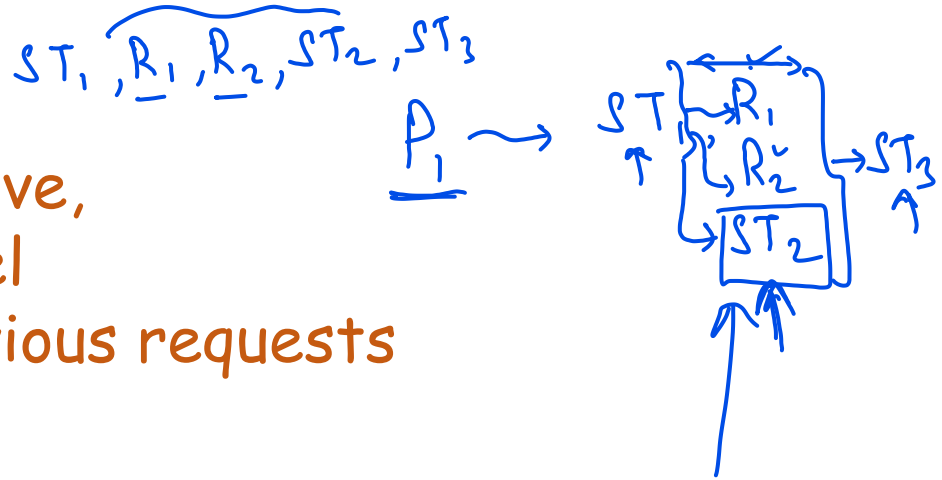
Request Parallelism

Request Parallelism





- Requirement:
  - Scalable execution of repetitive but independent tasks in parallel, with dynamic arrival
- Solution:
  - As independent requests (for services) arrive,
  - Each request is assigned to a task in parallel
    - while other such tasks are servicing previous requests
- (Natural) Systems Fit:
  - Client-Server Model
- Examples:
  - E-mail Server, Web-Server, Cloud



Request Parallelism – Implementation and Performance

- This is typically implemented as a multi-threaded server:
  - A pool of threads are maintained
  - Each new request is assigned to a free thread
  - On completion of (servicing the assigned) request,
    - the thread de-allocates any resources previously allocated and
    - is marked free
- Performance Considerations:
  - Throughput
    - Number of requests serviced per unit time
  - Response Time
    - Turn-around time per request

ML Algorithms – Training Phase vs. Inference Phase

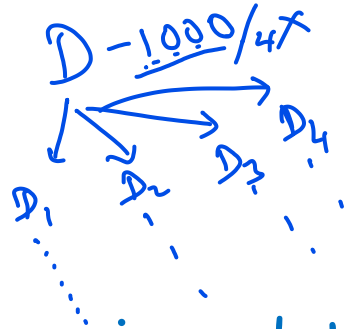
- During the inference phase or the prediction phase:
  - Request parallelism may be used to
    - deploy the model and
    - provide efficient inferences or predictions
- For the individual user submitting a request:
  - Response time is important (even if not critical)
    - Thumb rule in the practical world:
      - Any request on the Internet, the Web, or the Cloud must be serviced within 3 seconds
    - e.g. Recommender System on an e-commerce site
- For the provider:
  - Throughput is business-critical, while the
  - Average response time is important

Parallelization of ML Algorithms - Examples

Ensemble Methods

- Multiple ML algorithms (or learners) are trained on the same dataset:
  - The combination (ensemble learner) is expected to perform better than any of the individual learners.
- e.g. Bagging and Boosting

Bagging or Bootstrap Aggregation



- Generate m bootstrap data sets D_1, D_2, … D_m from the given data set:
  - Bootstrapping is the selection of random points with replacement
- Train each of the new data sets D_j to fit a model M_j and
  - Combine them
    - e.g. by taking the majority output of all classifiers or average of all the regressors.
- Bagging can be easily task-parallelized:
  - Tasks T_1, T_2, … T_m can each run as a different thread (i.e. on a different processor):
    - Where each T_j consisting of bootstrapping from the given set and training to obtain a model M_j

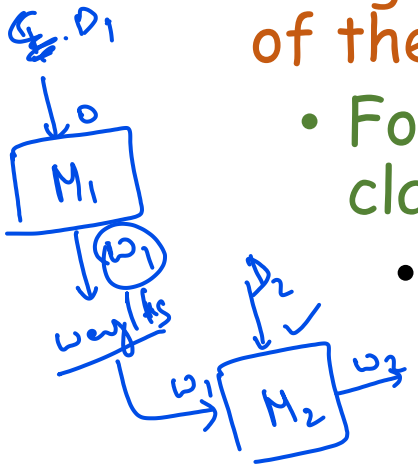
Bagging: Parallelization

- The parallelization discussed (see last slide) is for the training phase.
  - The inference phase requires a combination to be implemented.
    - This is easy (e.g., majority voting or averaging) and
    - parallelization is not critical
      - because m is not large (compared to n, the size of the dataset)

- Note:
  - Typically, bootstrap size B_{size} (or sample size) may be large
  - In bagging:
    - $B_{size} = n$ but
    - The number of bootstraps, m, is small.

AdaBoost (or Adaptive Boosting)

- Boosting:
  - Multiple learners $y_j(x)$ are trained on a weighted form of the training set.
  - Weights for each learner $y_j(x)$ are obtained from the performance of the previous learner $y_{j-1}(x)$
    - For instance, points that are misclassified by previous classifiers
      - receive greater weights in subsequent classifier(s)



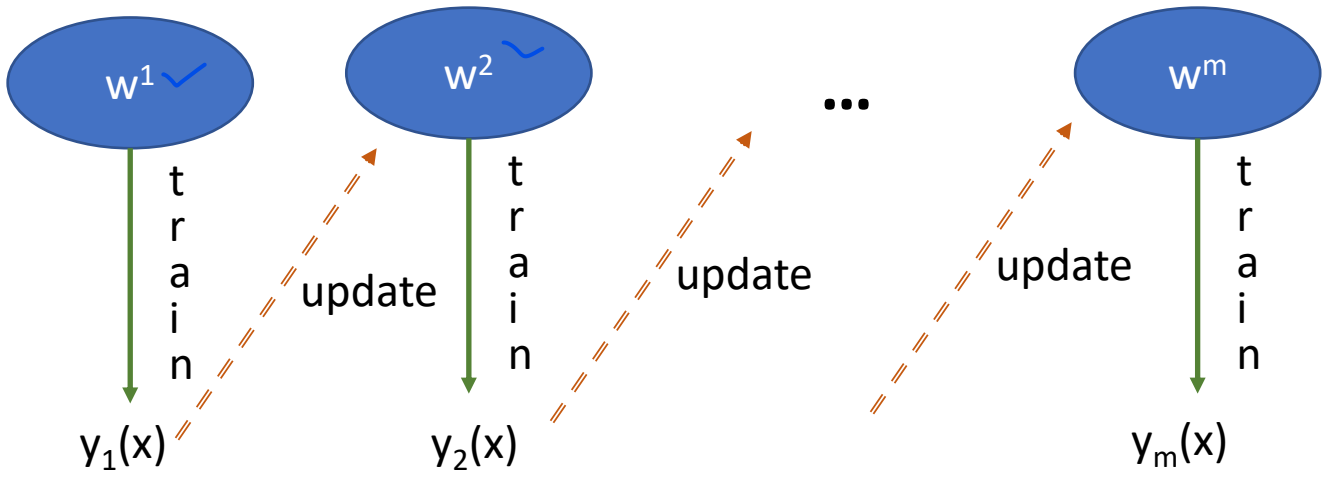
AdaBoost (or Adaptive Boosting) [contd.]

1. Initialize the data weighting coefficients $W^1 = 1/N$ for n= 1..N
2. for j = 1 to m
   1. Train a learner $y_j(x)$
   2. evaluate error e_j
   3. Update weighting coefficients $W^{j+1} = f(W^j, y_j, e^j)$
3. Make predictions using the final model y_m

AdaBoost (or Adaptive Boosting) [contd.]

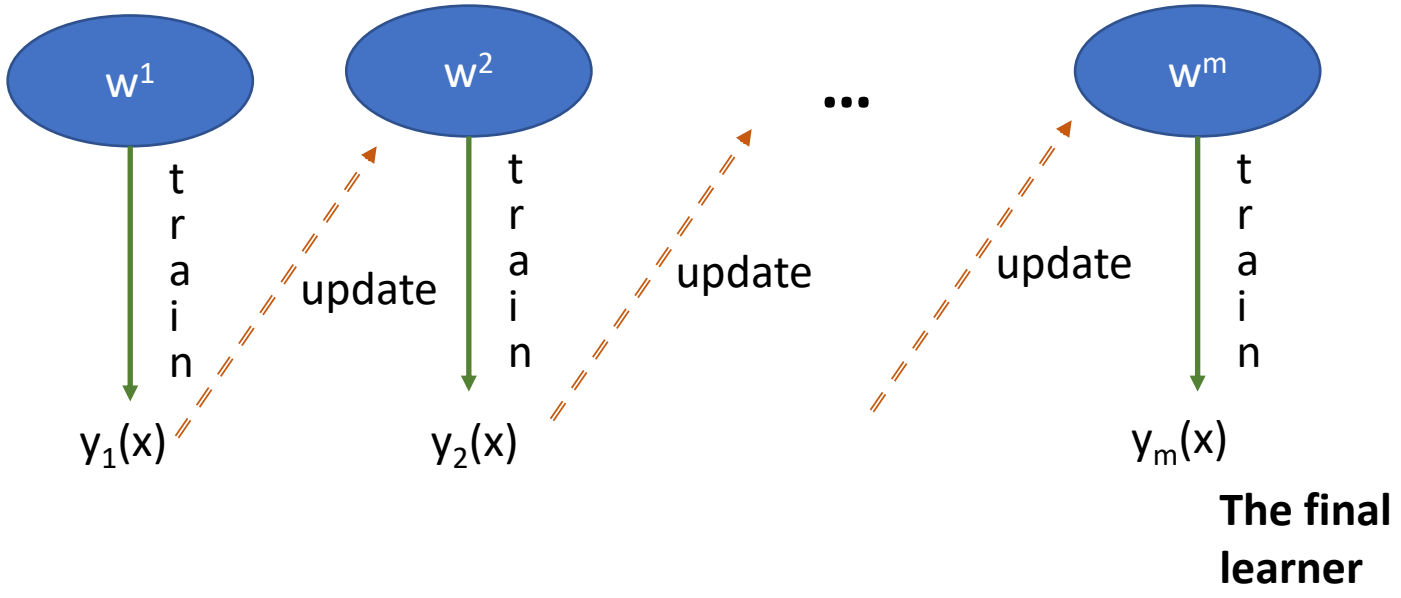
1. Initialize the data weighting coefficients $W^1 = 1/N$ for n= 1..N
2. for j = 1 to m
   1. Train a learner $y_j(x)$
   2. evaluate error e_j
   3. Update weighting coefficients $W^{j+1} = f(W^j, y_j, e^j)$
3. Make predictions using the final model y_m

There is a sequential dependency!
This is not easily parallelizable!



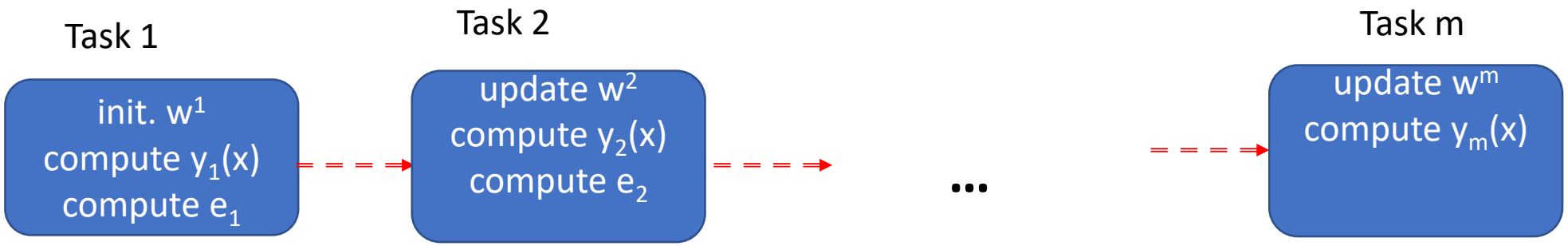
AdaBoost: Pipelining?

1. Initialize the data weighting coefficients $W^1 = 1/N$ for n= 1..N
2. for j = 1 to m
   1. Train a learner $y_j(x)$
   2. evaluate error e_j
   3. Update weighting coefficients $W^{j+1} = f(W^j, y_j, e^j)$
3. Make predictions using the final model y_m



While this algorithm is not amenable for data parallelism or for task parallelism, software pipelining may be attempted!

AdaBoost: Software-Pipelined



This pipeline provides speedup(m) > 1

only if computation of y_j and e_j can proceed in parallel with update w^{j+1}