innovate achieve lead

BITS Pilani

WILP

AMLCCLZG516
ML System Optimization

Murali Parameswaran

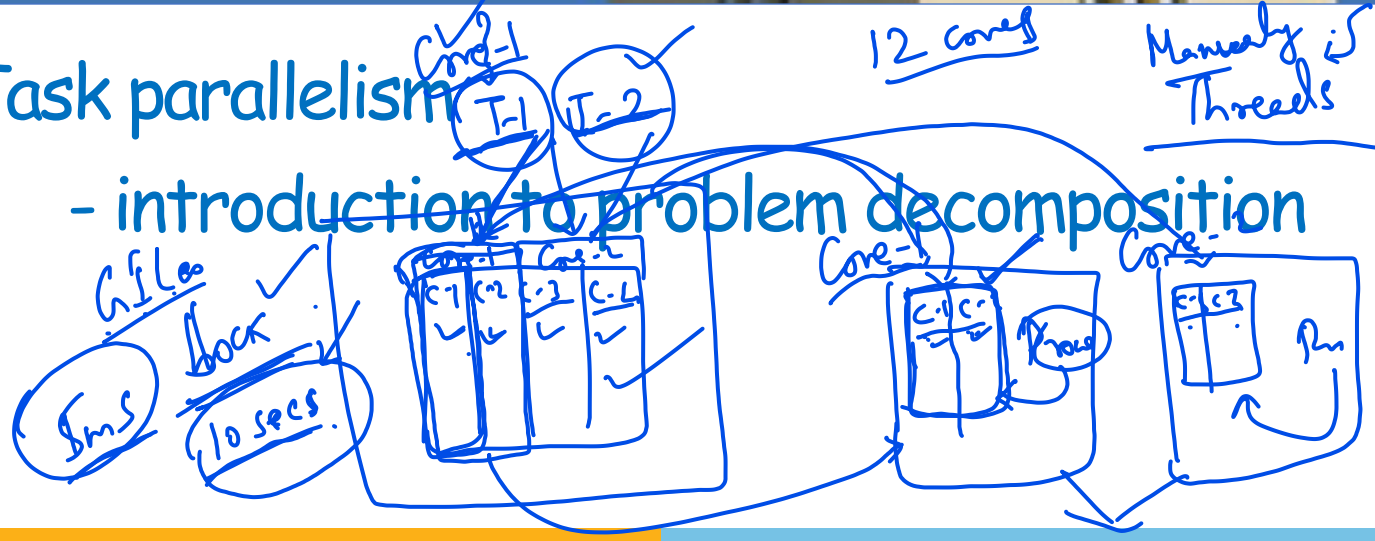
innovate achieve lead

BITS Pilani

WILP

AMLCCLZG516
ML System Optimization

Task parallelism
- introduction to problem decomposition



Threadpool ↔ process pool

- Shared memory
- Ready
- Avoid switch to another memory
- low overhead

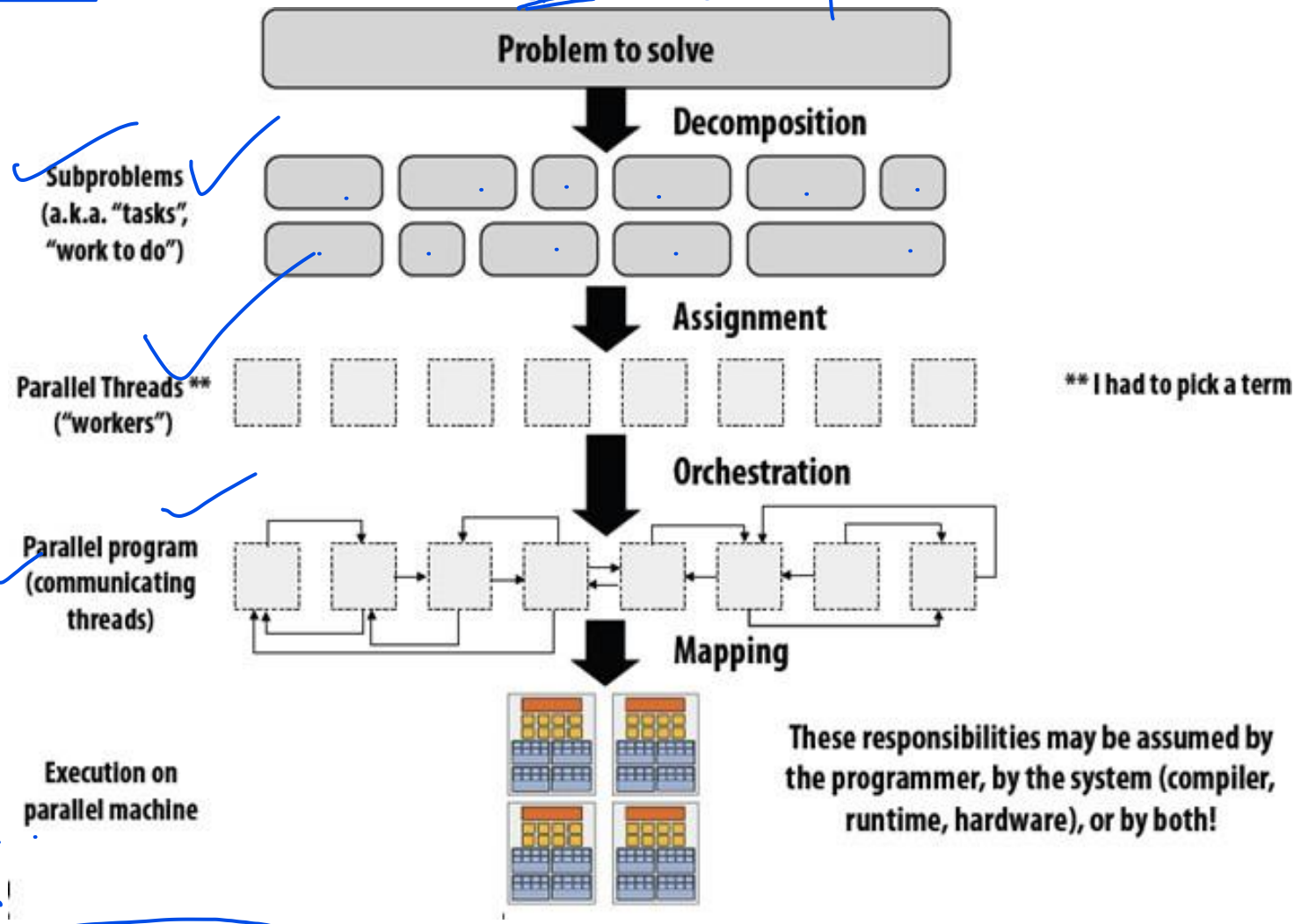
- Separate memory
- Ready & Write
- Overhead of copying

Creating a parallel program

Problem Decomposition

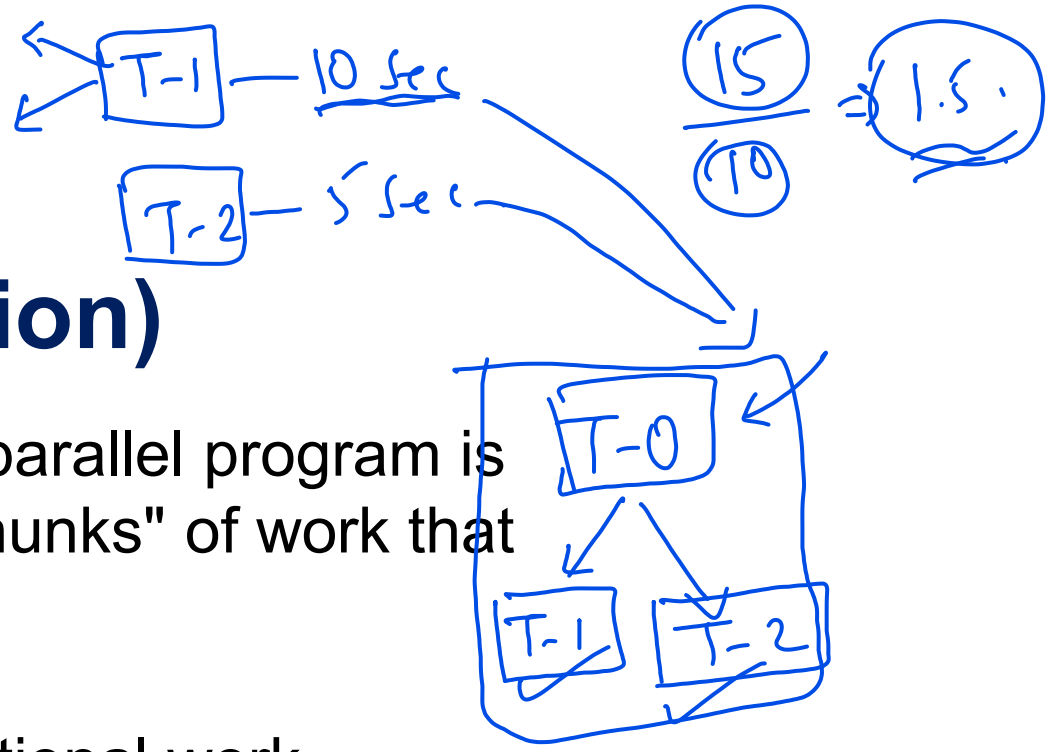
- Break up problem into tasks that can be carried out in parallel
- Main idea: create at least enough tasks to keep all execution units on a machine busy

Key challenge of decomposition: identifying dependencies (or... a lack of dependencies)



Design a Parallel Program:

Partitioning (aka Decomposition)



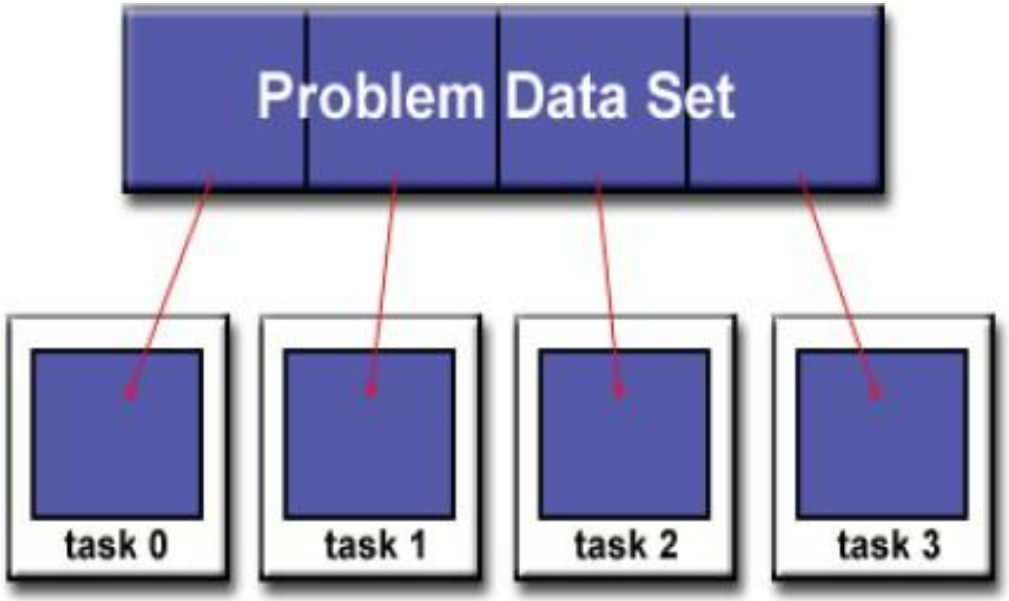
- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks.

- Two basic ways to partition computational work among parallel tasks:
  - ***domain decomposition*** and
  - ***functional decomposition***.

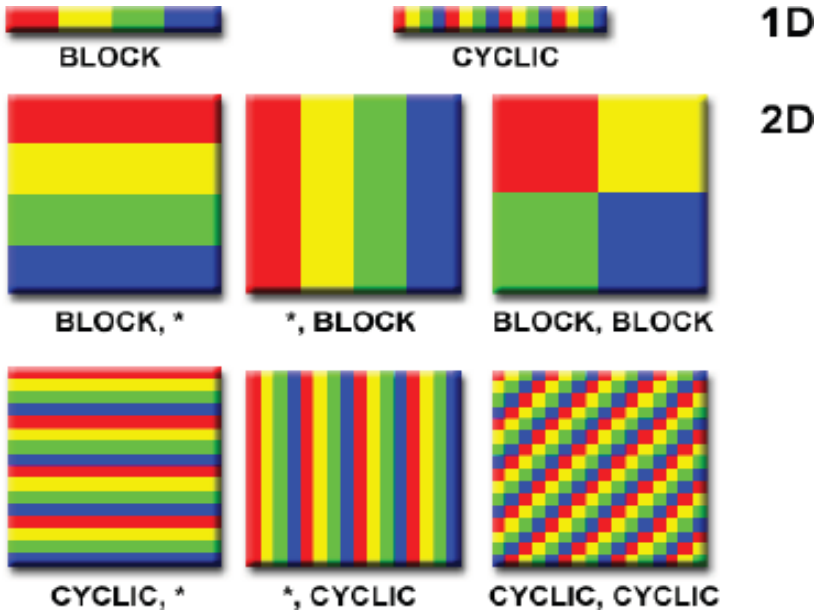
- A combination of these two types of decomposition are used to solve real-world problems.

Partitioning - Domain Decomposition

The data associated with a problem is decomposed. Each parallel task then works on a portion of the data. This is **data parallelism.**



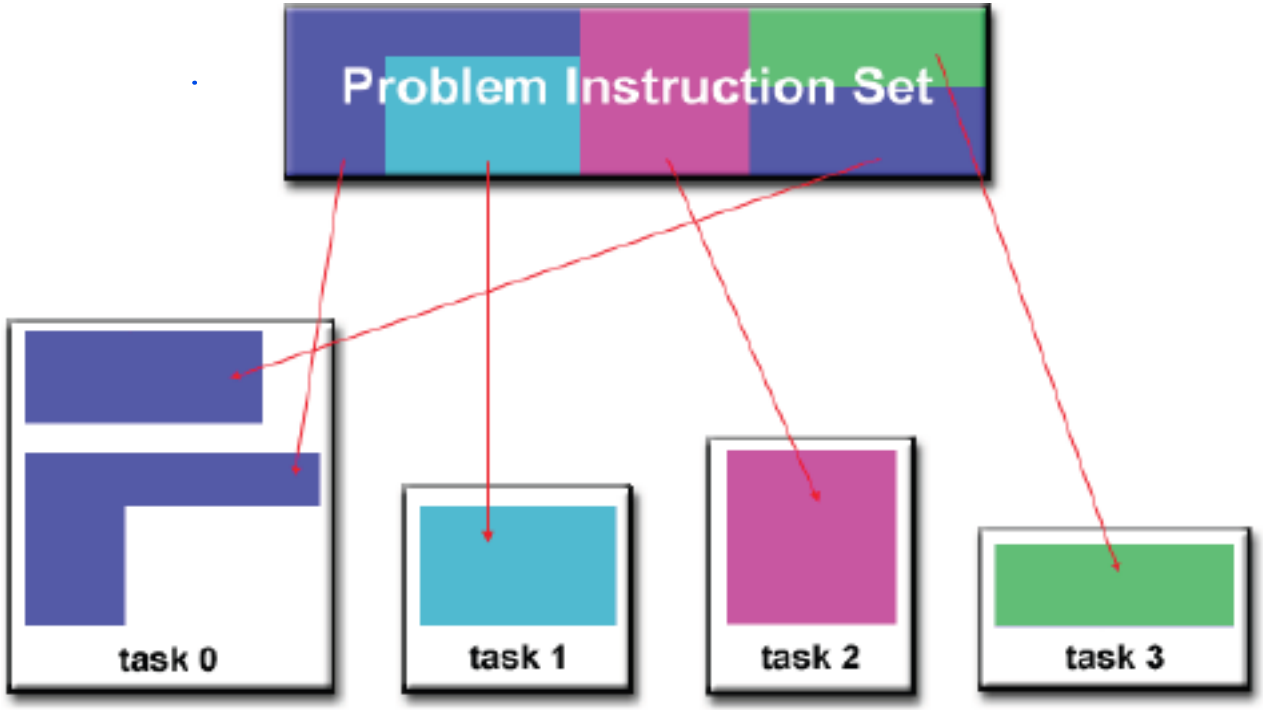
Different ways to partition data



Partitioning -. Functional Decomposition

- the focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.

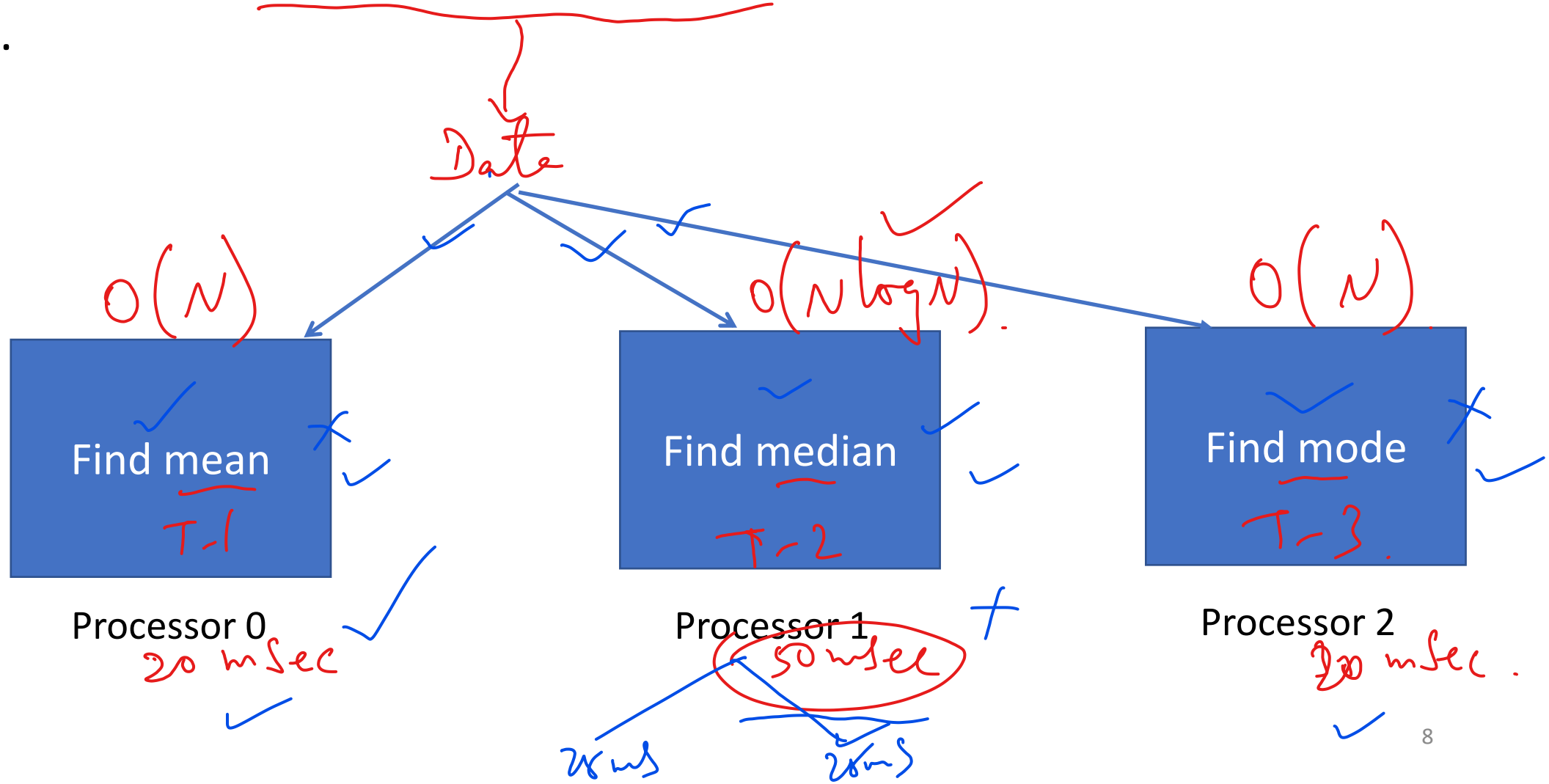
This is **task parallelism.**



Task Parallelism

Task Parallelism – Example

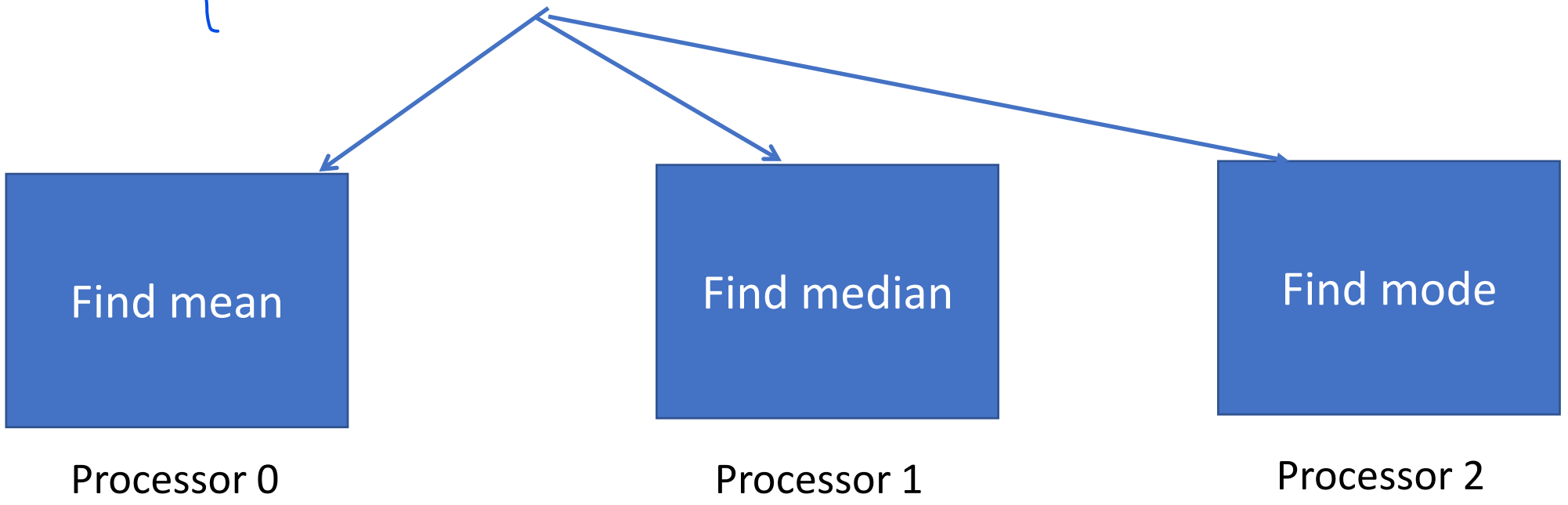
Problem: Given a list Ls of numeric values find the *mean*, *median*, and *mode*.



Task Parallelism – Example

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

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

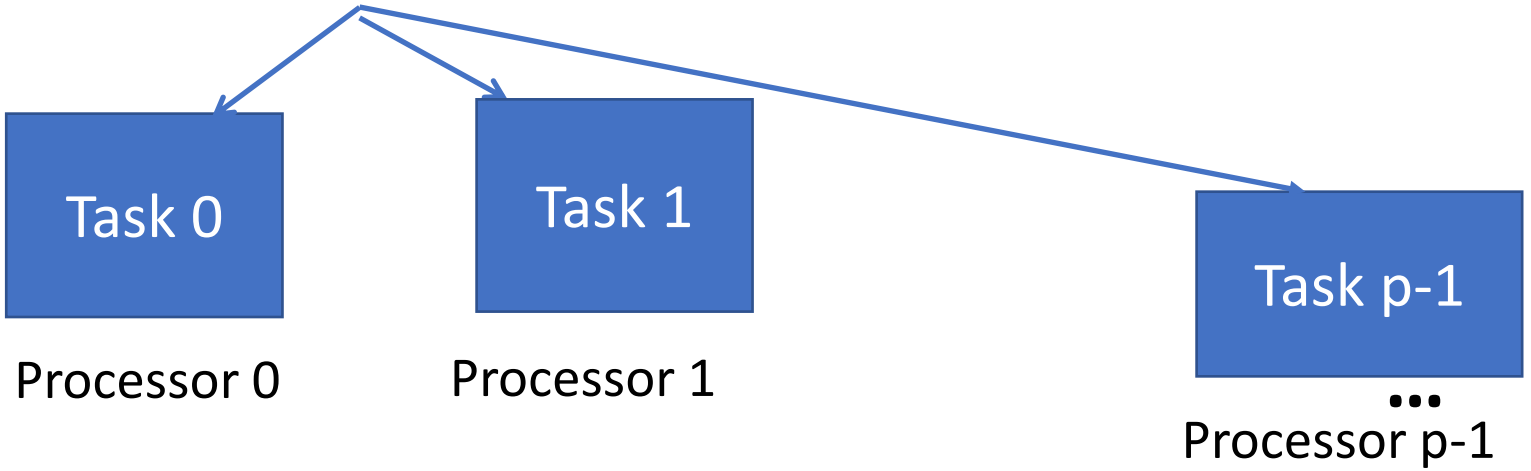


Task Parallelism

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

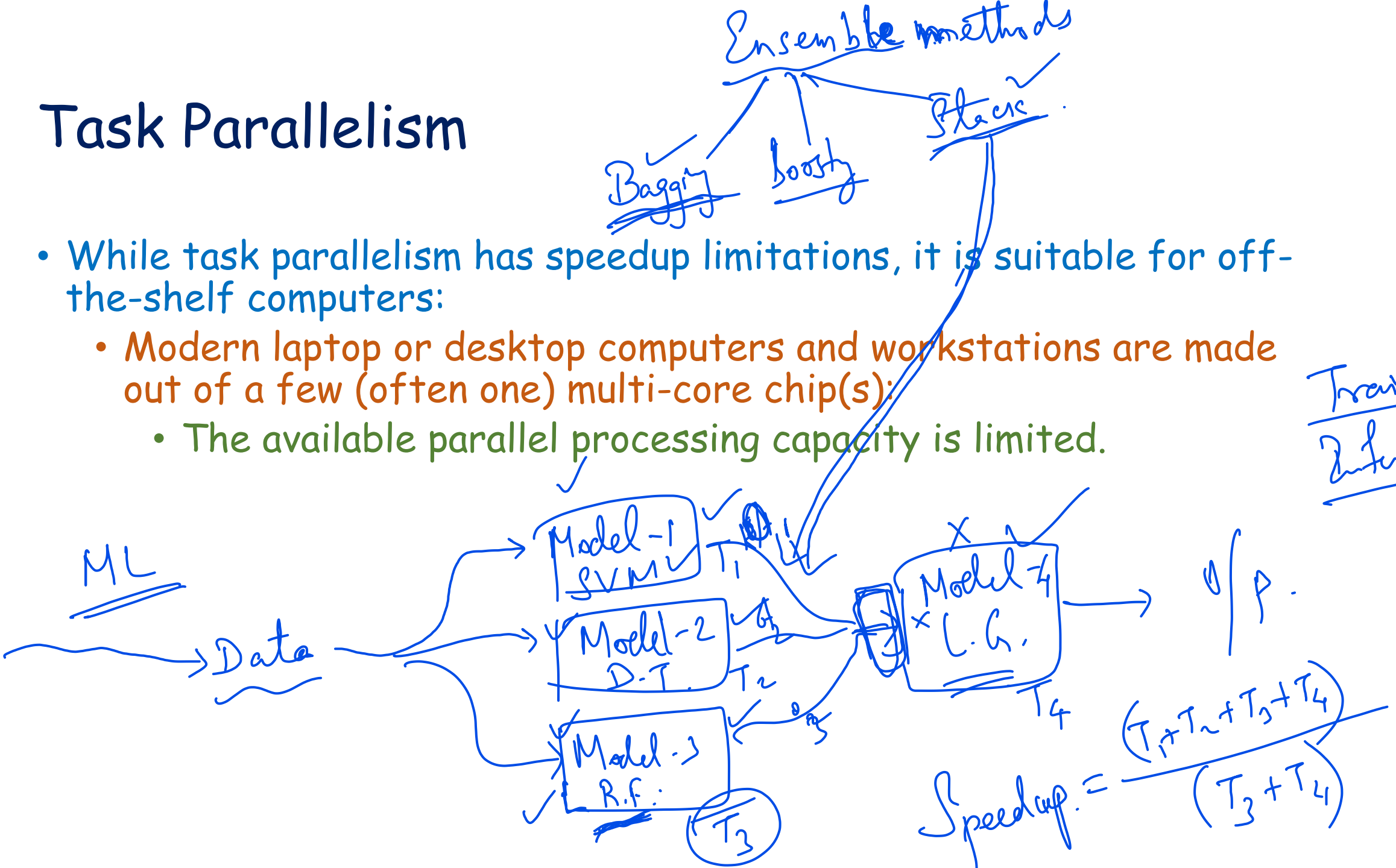
This is not scalable:

- 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

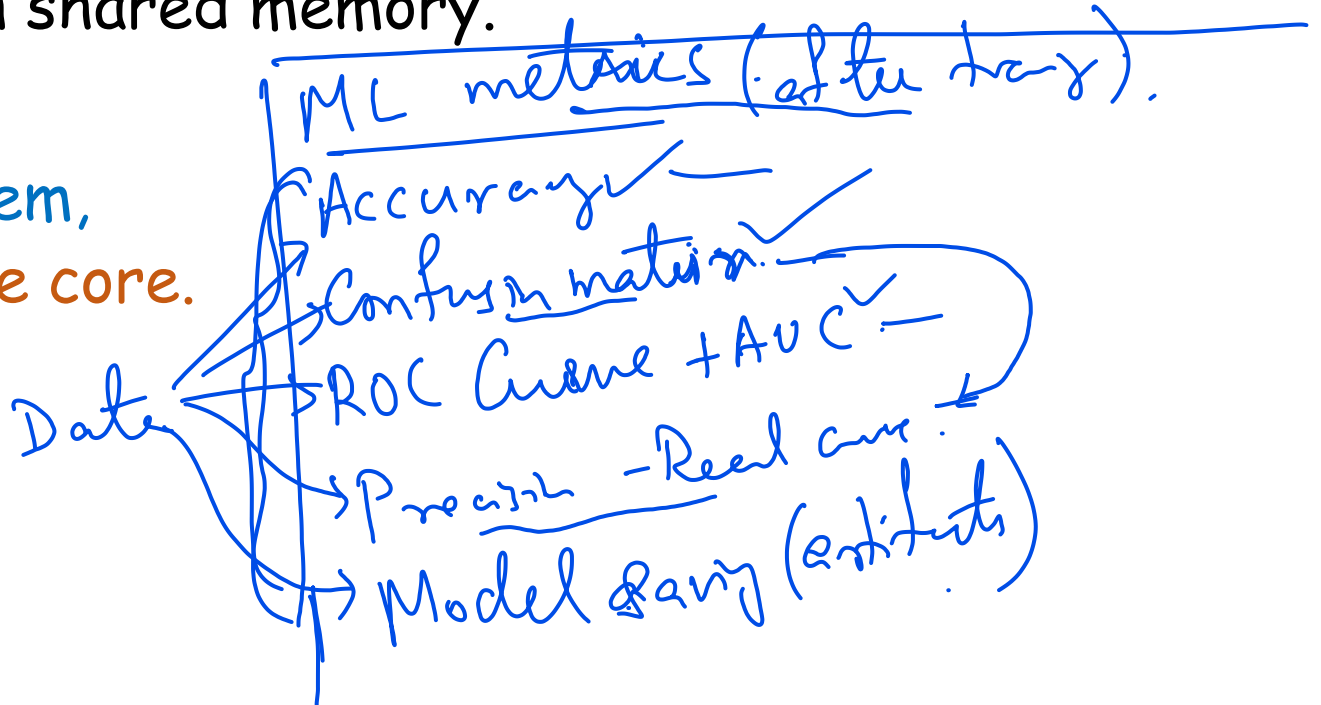
- While task parallelism has speedup limitations, it is suitable for off-the-shelf computers:
  - Modern laptop or desktop computers and workstations are made out of a few (often one) multi-core chip(s):
    - The available parallel processing capacity is limited.



Programming Task Parallelism

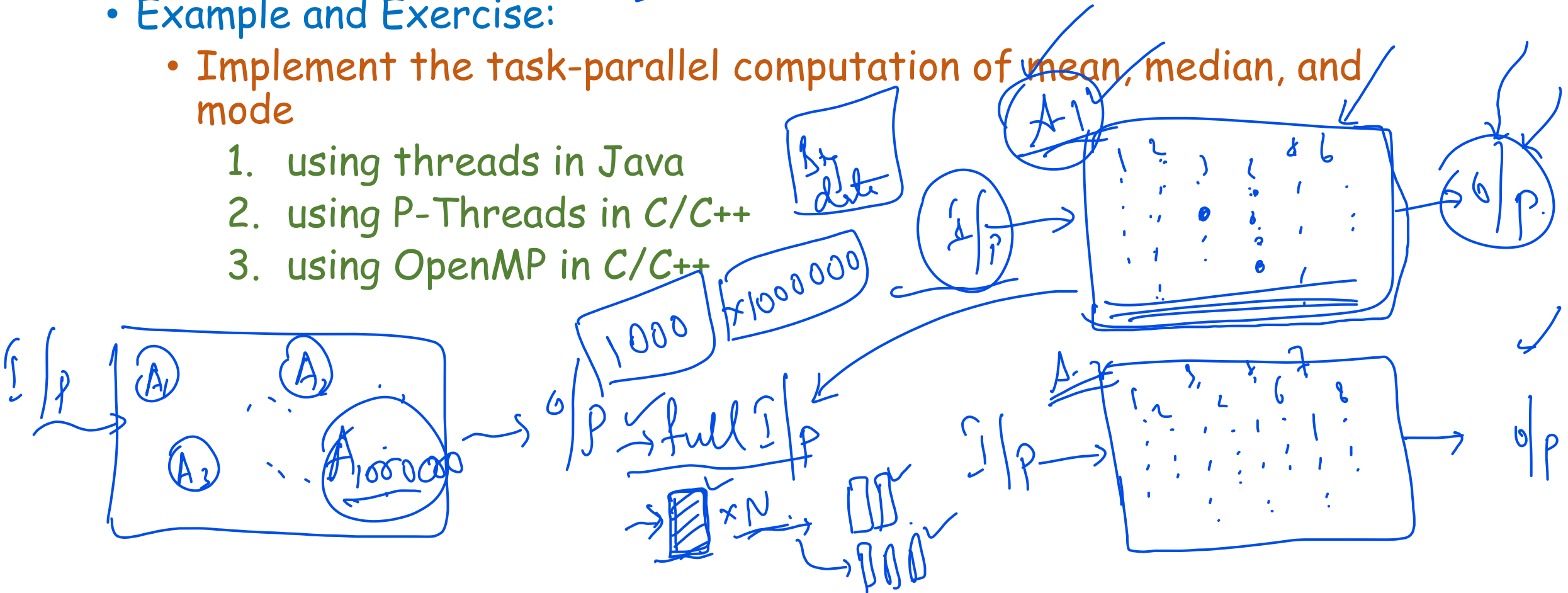
- In shared memory computers:
  - Task parallelism can be implemented using multi-threaded programs:
    - One task per thread
      - Where data is stored in shared memory.

- 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

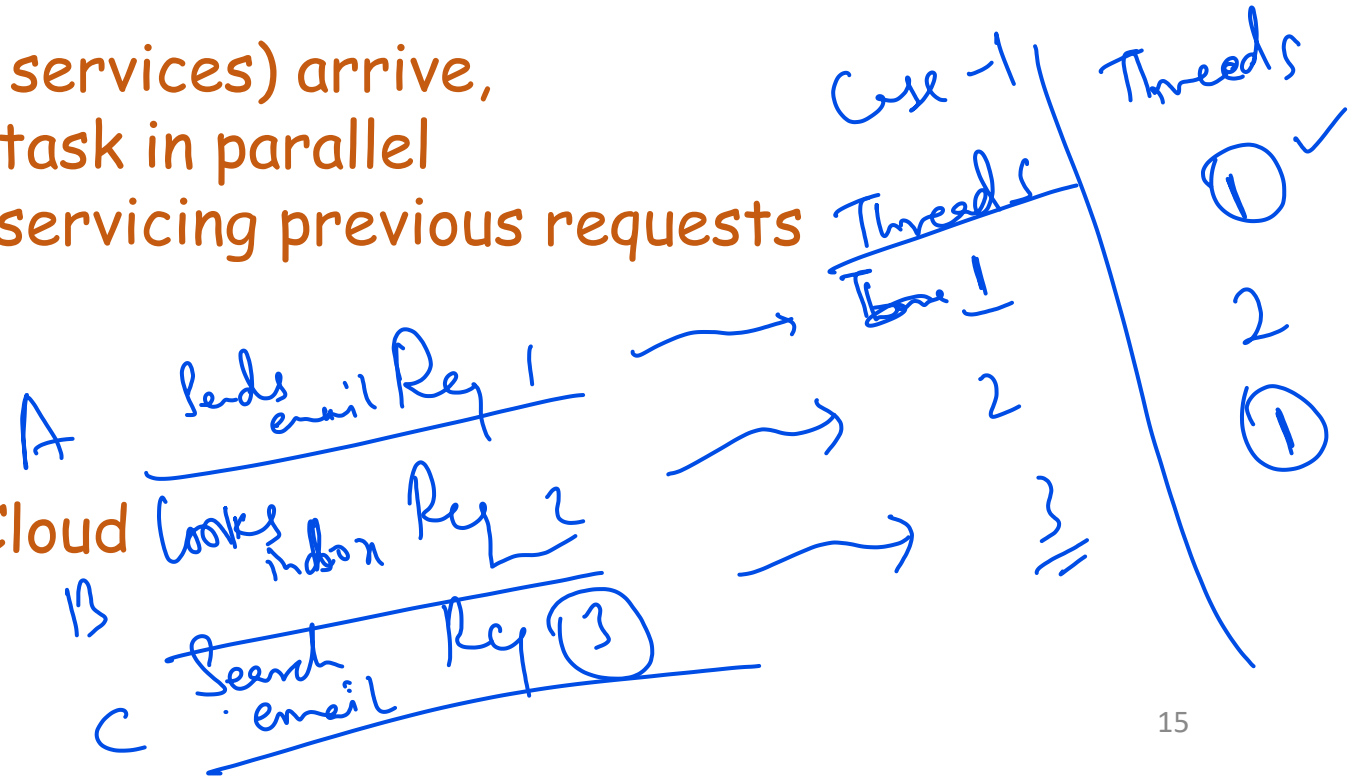
Requests

↳ Dynamic

↳ (100, 1000, 20) user requests

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

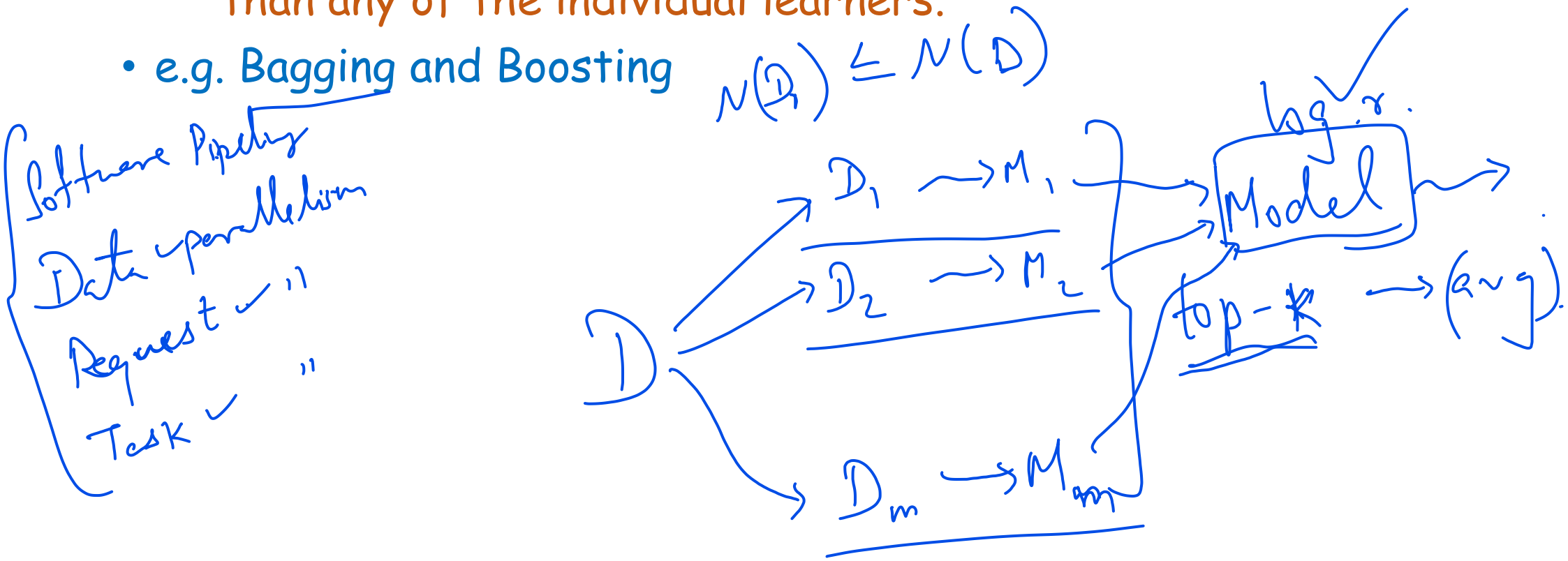
ML Algorithms – Training Phase vs. Inference Phase

- 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(voting) 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)

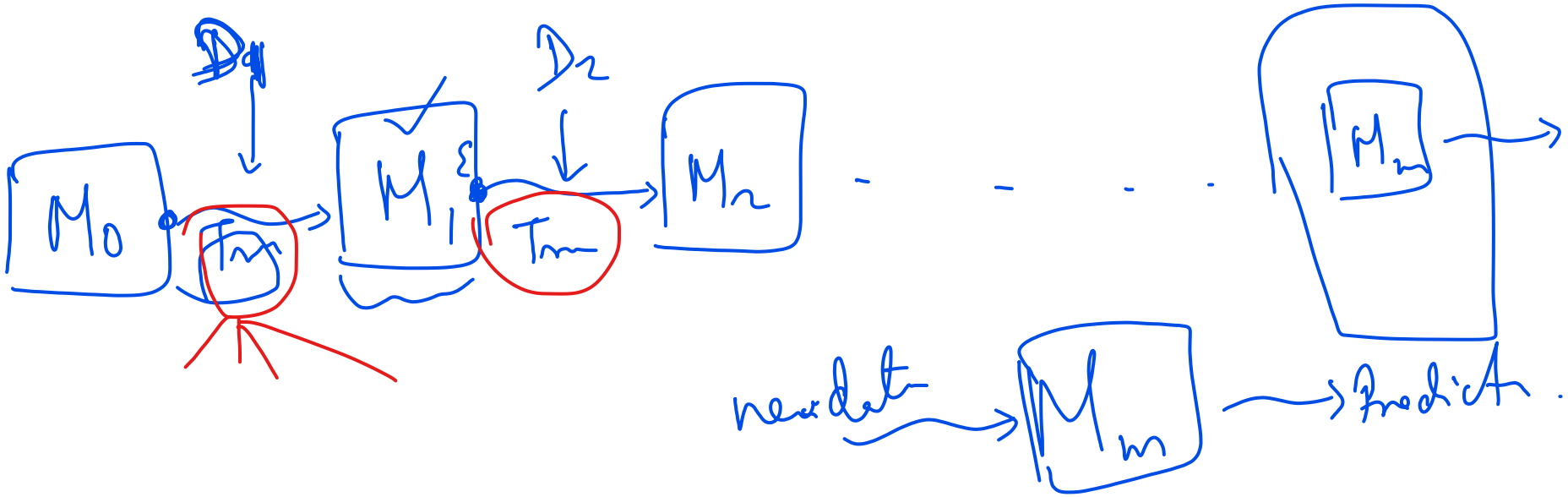
- Note:
  - Typically, bootstrap size B_{size} (or sample size) may be large
  - In bagging:
    - $B_{size} = n$ but
    - The number of bootstraps, m, is small.

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.]

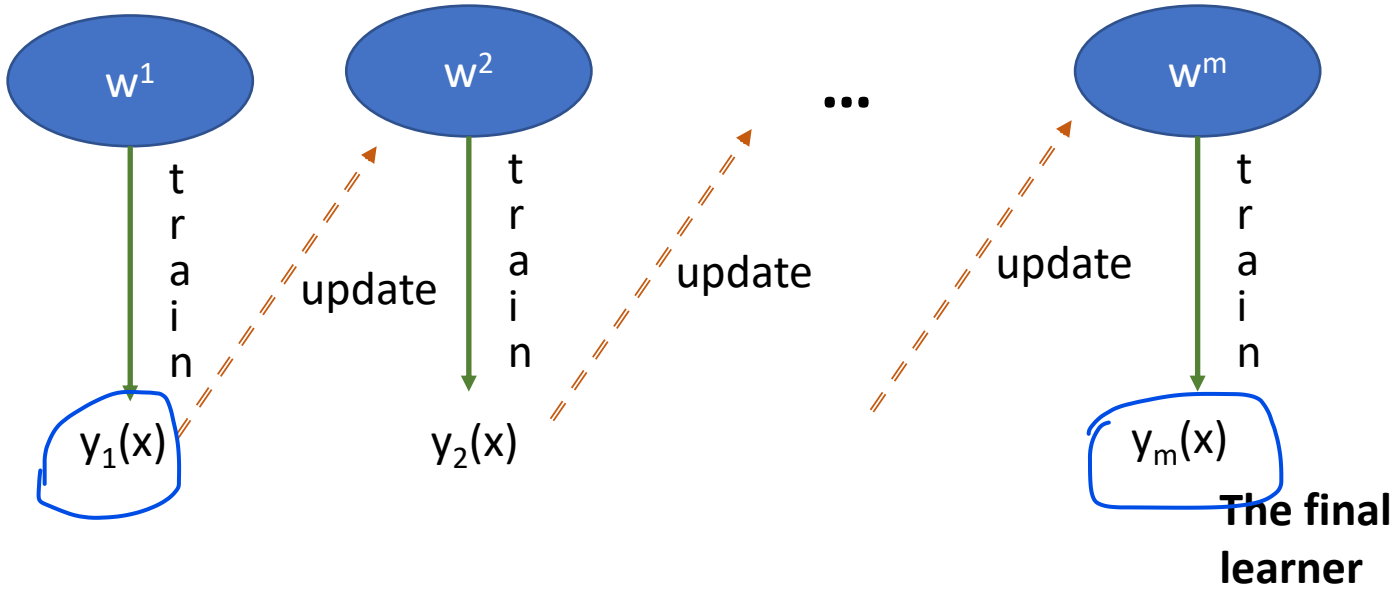
1. Initialize the data weighting coefficients $W^1 = 1/N$ for n= 1..N
2. for j = 1 to m
   1. Train a learner $y_j(x)$
   2. evaluate error e_j
   3. Update weighting coefficients $W^{j+1} = f(W^j, y_j, e^j)$
3. Make predictions using the final model y_m



AdaBoost (or Adaptive Boosting) [contd.]

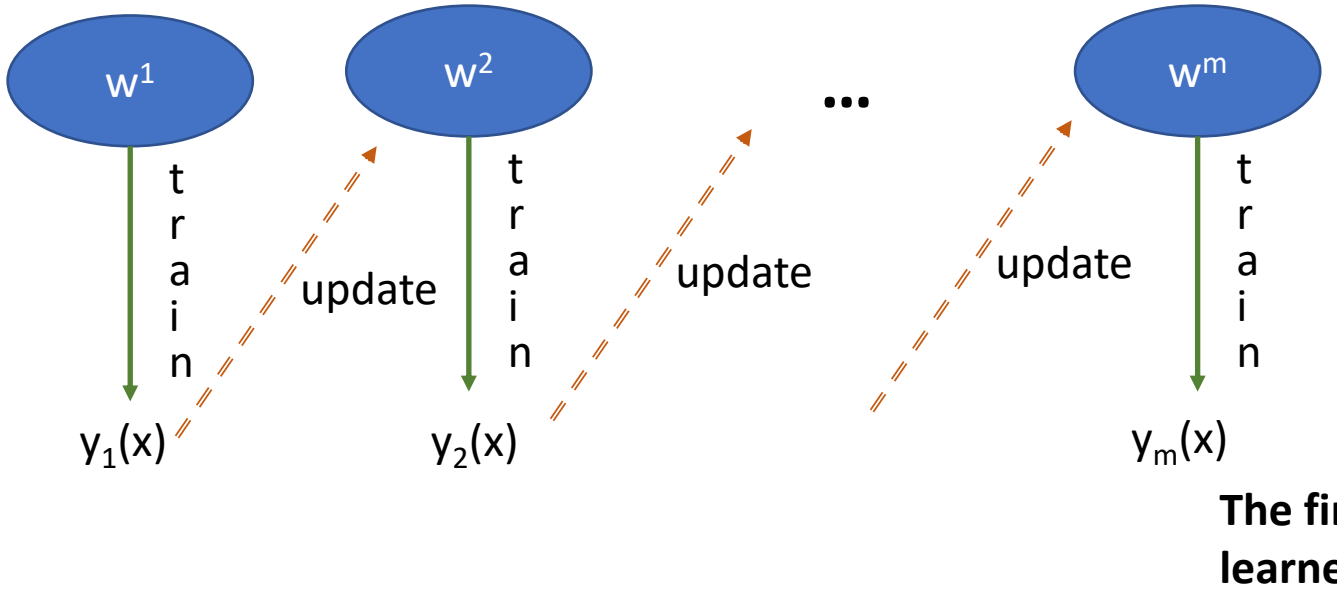
1. Initialize the data weighting coefficients $W^1 = 1/N$ for n= 1..N
2. for j = 1 to m
   1. Train a learner $y_j(x)$
   2. evaluate error e_j
   3. Update weighting coefficients $W^{j+1} = f(W^j, y_j, e^j)$
3. Make predictions using the final model y_m

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



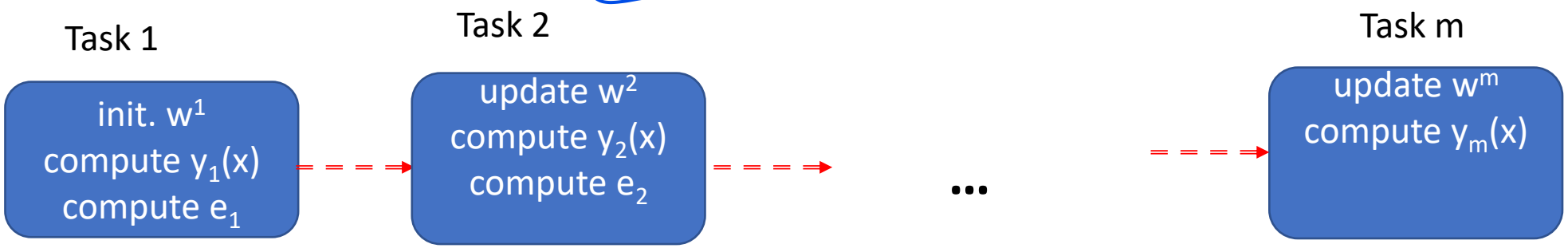
AdaBoost: Pipelining?

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}