innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Murali Parameswaran
muralip@wilp.bits-Pilani.ac.in

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Session 1: 26 Nov 2023

Orientation: Course Introduction

Distributed Computing

Running programs on multiple computers (in a network – possibly geographically distributed)

Distributed Computing

Infrastructure (Hardware & Systems Software)
for
Applications (Algorithms, Software Solutions)

Content & Pedagogy

- Focus on principles, concepts, and design
  - Pragmatics and Implementation to be learnt by doing – enabled by Assignments and Project.

- Lecture Sessions are expected to be interactive:
  - students are expected to raise questions and
  - the instructor will ask questions (which the students are expected to answer)

Evaluation

- Mid-term test and final exam – centrally scheduled by BITS
  - A Total of 50% weight
- 2 Assignments and a Project
  - (10+10+30 =) 50%
- Recommended Programming Environment:
  - Java (preferably) or
  - C/C++/Rust

- Note: No prescribed textbooks for this course.

innovate achieve lead

BITS Pilani

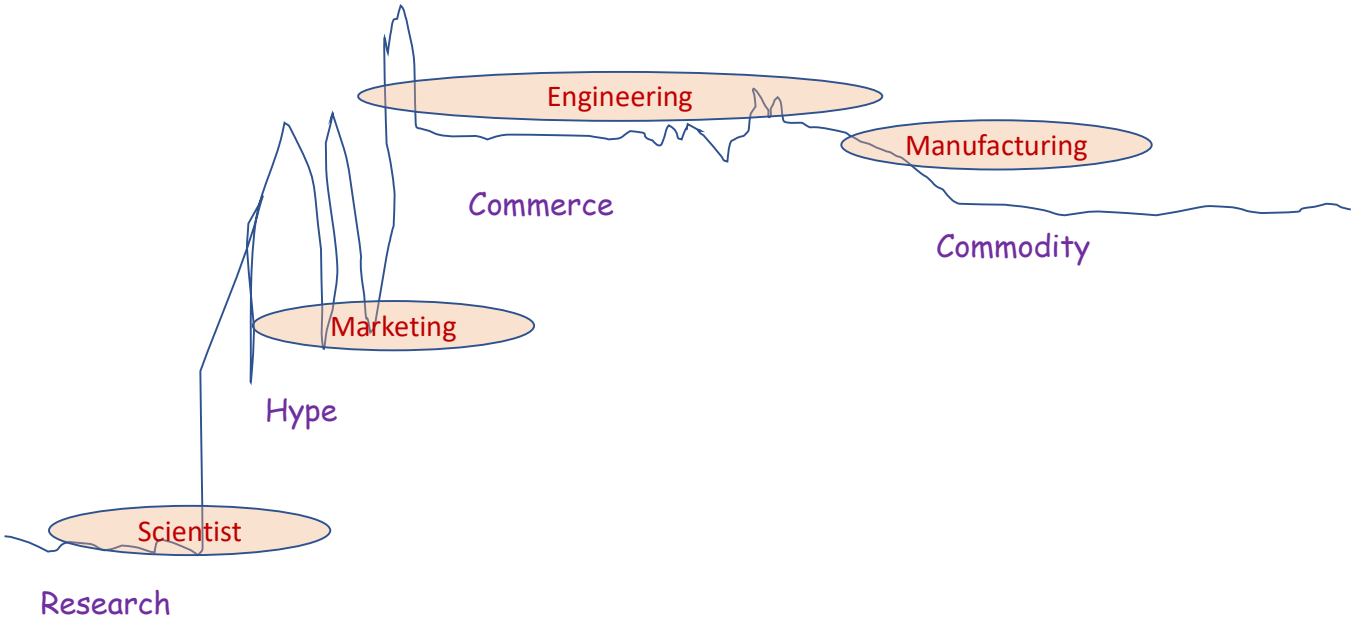
WILP

AIML CLZG516
ML System Optimization

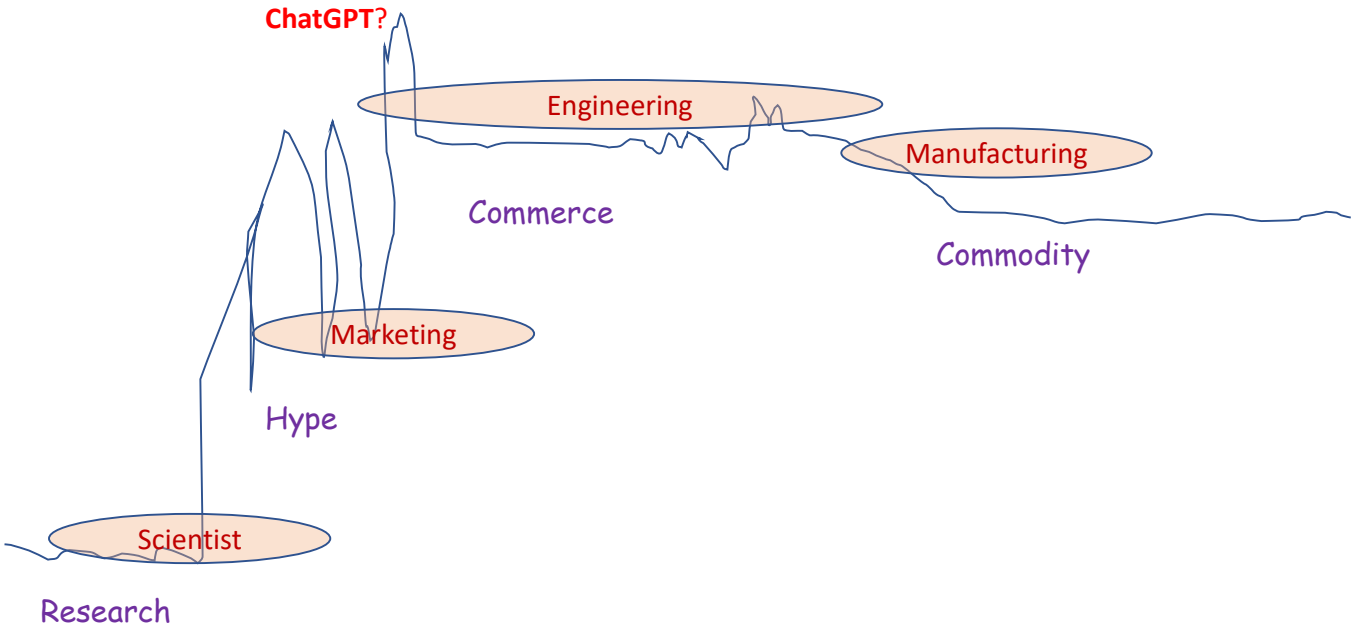
Session 1

Course Introduction

Lifecycle of New Technologies



Lifecycle of AI/ML – Where are we ?

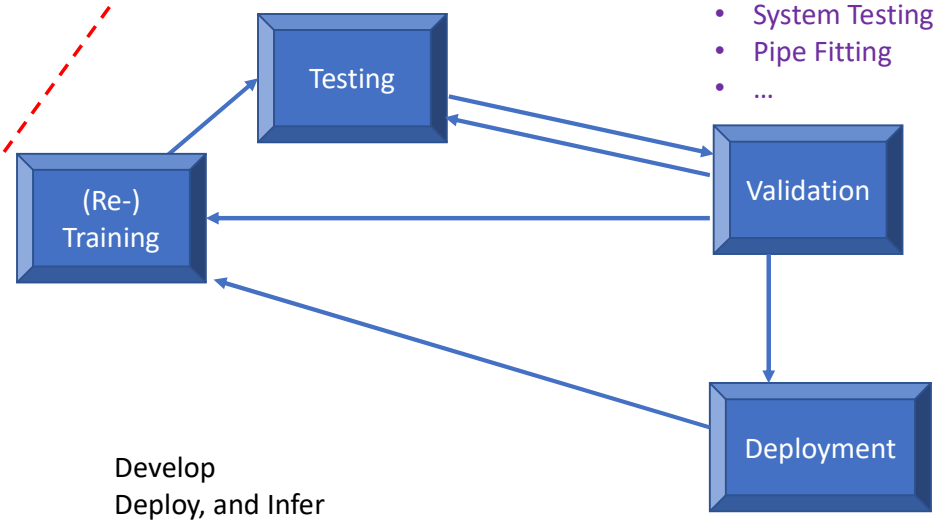


Machine Learning – Enterprise Practice

- AI and Machine Learning is becoming central to organizations:
  - No longer a one-off activity
  - Multiple problems / perspectives addressed through ML
  - Multiple ML solutions deployed
- ML is becoming a continual activity:
  - Data change; Context changes
    - Drift in the solution
  - Problems change; Requirements change;
    - New model(s) required
  - World changes; Expectations change
    - Performance and Standardization critical ==>
      - Packaging vs. Pricing

Operationalizing AI/ML

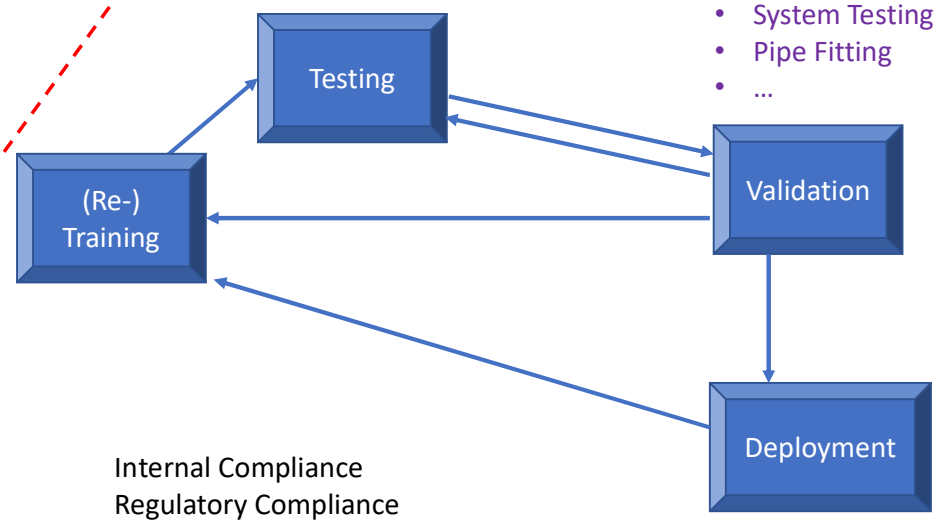
- Class-room View :
  - Development
    - Model Building / Training
  - Deployment
    - Inference
- Enterprise View:
  - Compliance
  - System Testing
  - Pipe Fitting
  - …



Develop
Deploy, and Infer

Operationalizing AI/ML

- Class-room View :
  - Development
    - Model Building / Training
  - Deployment
    - Inference
- Enterprise View:
  - Compliance
  - System Testing
  - Pipe Fitting
  - …

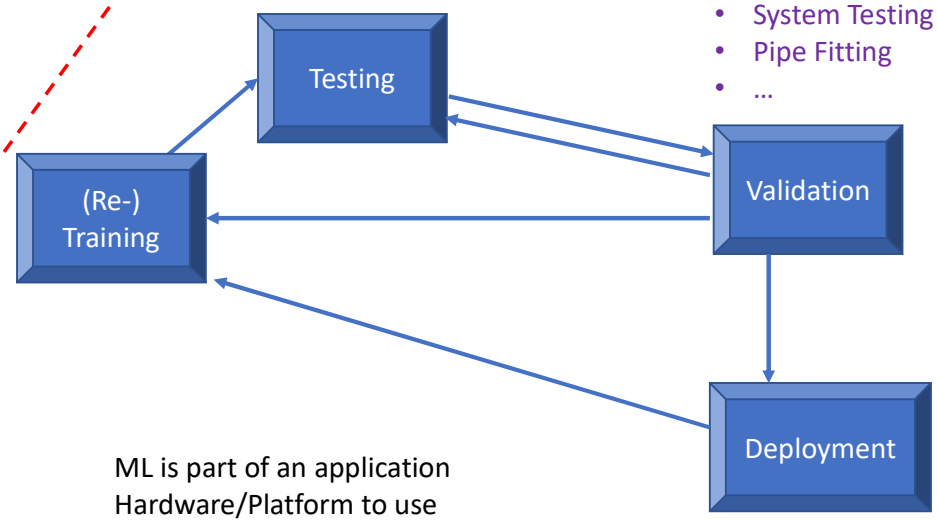


Internal Compliance
Regulatory Compliance

Operationalizing AI/ML

- Class-room View :
  - Development
    - Model Building / Training
  - Deployment
    - Inference

- Enterprise View:
  - Compliance
  - System Testing
  - Pipe Fitting
  - …

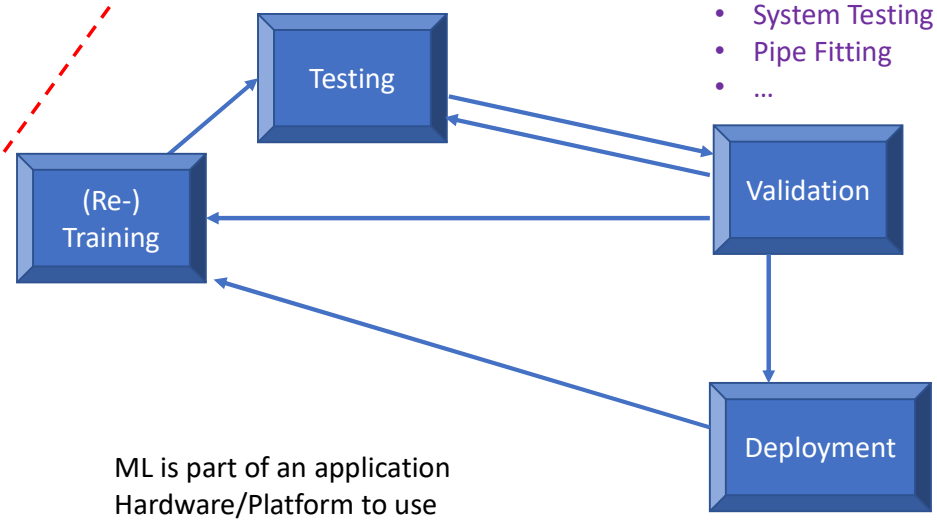


ML is part of an application
Hardware/Platform to use

Operationalizing AI/ML

- Class-room View :
  - Development
    - Model Building / Training
  - Deployment
    - Inference

- Enterprise View:
  - Compliance
  - System Testing
  - Pipe Fitting
  - …

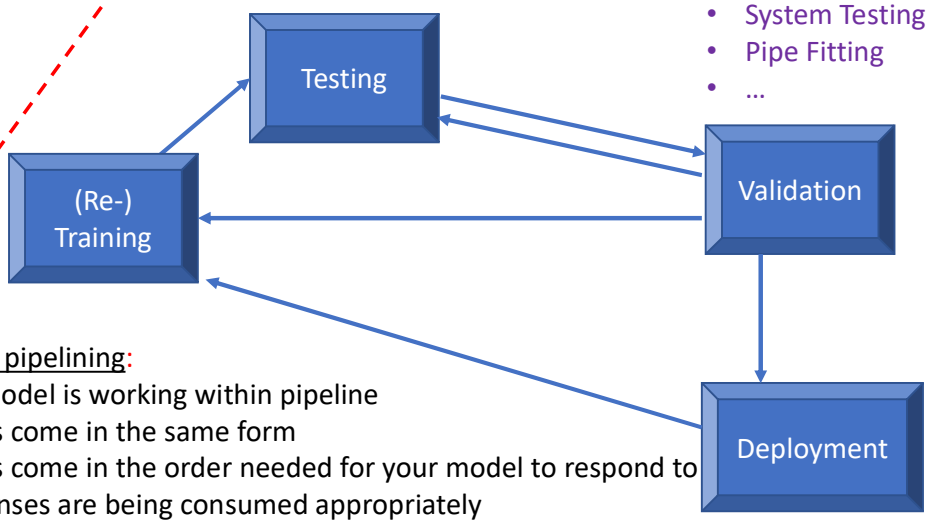


ML is part of an application
Hardware/Platform to use

Will our model fit in the pipeline?

Operationalizing AI/ML

- Class-room View :
  - Development
    - Model Building / Training
  - Deployment
    - Inference
- Enterprise View:
  - Compliance
  - System Testing
  - Pipe Fitting
  - …



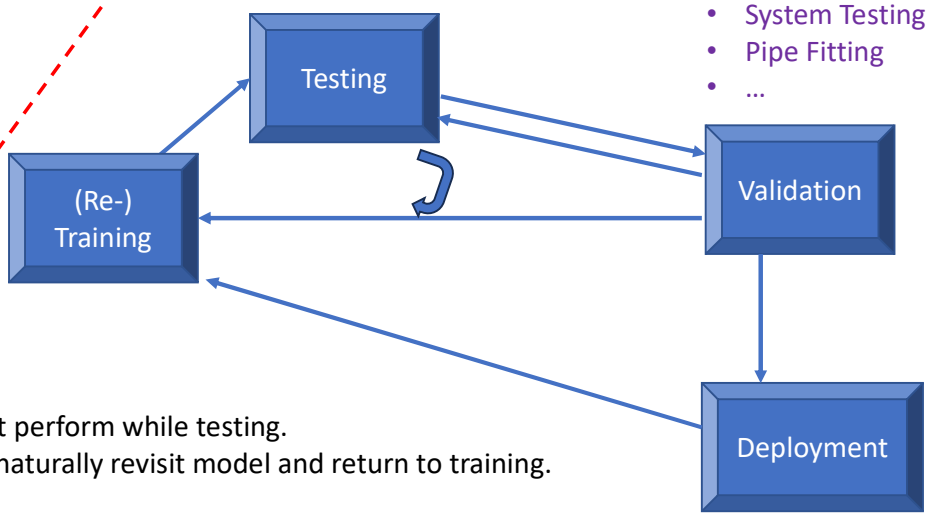
Software pipelining:

Test whether model is working within pipeline
Whether inputs come in the same form
Whether inputs come in the order needed for your model to respond to
Whether responses are being consumed appropriately
Whether responses are to consumed one at a time, or a sequence of responses have to be consumed

May need to return to training after validation/testing.

Operationalizing AI/ML

- Class-room View :
  - Development
    - Model Building / Training
  - Deployment
    - Inference
- Enterprise View:
  - Compliance
  - System Testing
  - Pipe Fitting
  - …

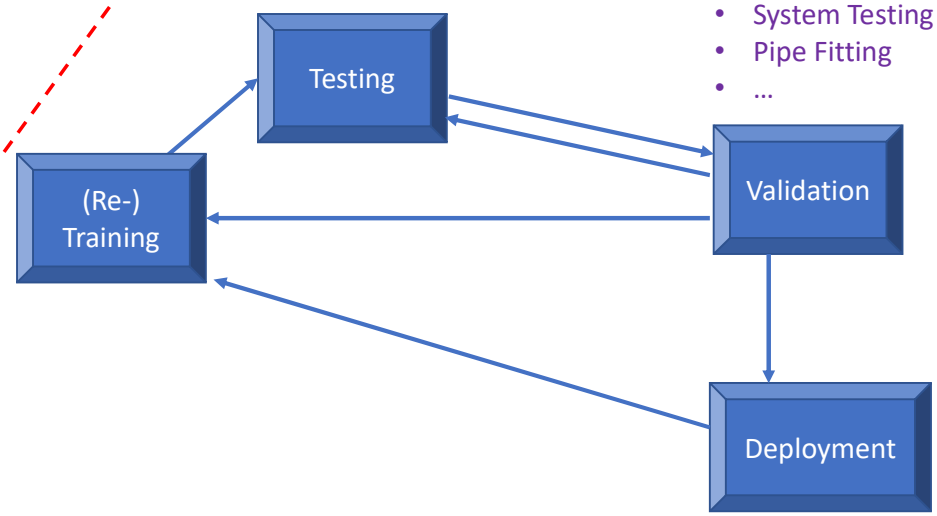


If model doesn't perform while testing.
AI scientist will naturally revisit model and return to training.

Operationalizing AI/ML

Can we stop after deployment?

- Class-room View :
  - Development
    - Model Building / Training
  - Deployment
    - Inference
- Enterprise View:
  - Compliance
  - System Testing
  - Pipe Fitting
  - …



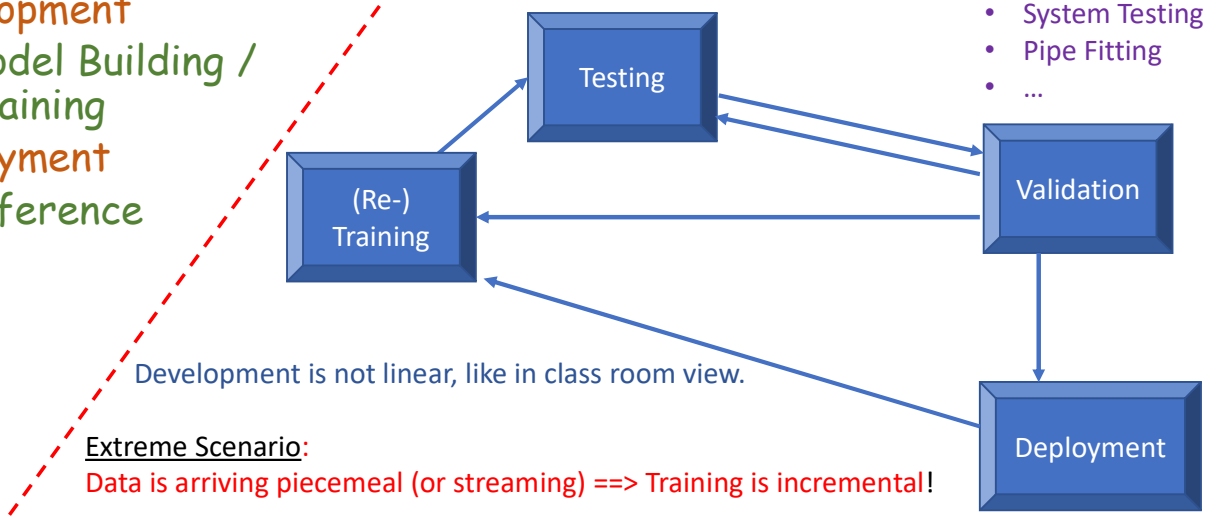
Machine Learning – Enterprise Practice-Recap

- AI and Machine Learning is becoming central to organizations:
  - No longer a one-off activity
  - Multiple problems / perspectives addressed through ML
  - Multiple ML solutions deployed
- ML is becoming a continual activity:
  - Data change; Context changes
    - Drift in the solution
  - Problems change; Requirements change;
    - New model(s) required
  - World changes; Expectations change
    - Performance and Standardization critical ==>
      - Packaging vs. Pricing
        - Depends on competition
        - Cost for model inference

Operationalizing AI/ML

- Class-room View :
  - Development
    - Model Building / Training
  - Deployment
    - Inference

- Enterprise View:



Operationalizing AI / ML: Cost

Focus of this course

- Cost:
  - Time and Resources during Training vs Inference
- During Training:
  - Running Time of an algorithm:
    - E.g. k-means is an $O(N*N)$ algorithm given N data points
    - E.g. SVM has a time complexity between $O(d*N^2)$ and $(d*N^3)$ where
      - d is the number of dimensions (of the data points) and
      - N is the number of data points

Cost during Training

- Example
  - E.g. SVM has a time complexity between $O(d*N^2)$ and $(d*N^3)$ where
    - d is the number of dimensions (of the data points) and
    - N is the number of data points
  - For a large dataset N, say, $N = 10^9$ and d=5 this could be costly:
    - Assuming 2 simple arithmetic operations per data point:
    - this amounts to at least 10^{19} ($=5*2*10^9*10^9$) operations
  - Given a 2.5 GHz processor, i.e. 0.4ns clock cycle
  - and 1 CPI (i.e. cycles per instruction), a measure of processor throughput
    - [simplistic but close to reality!]
  - 10^{19} operations will take close to 5.3 years
- Reducing running time during training is a big focus in this course!

Reducing running time

- Typical methods:
  - Parallelize or distribute computation:
    - Multi-threaded programming on multi-core processors
    - Massively multi-threaded programming on many-core GPGPUs
    - Distributed Programming on Scale-out Clusters of CPUs or GPUs
  - Hand-tuning or compiler-performed code optimization
    - Rewritten for parallelism or generated by compilers
  - Process = Program + Address Space (at run time)
    - Threads share address space:
      - Each thread gets its own call stack
      - Heap and global area are shared by all threads
      - Threads run on a shared memory model (e.g., multi-core, many-core processors)
    - Distributed programming is on Distributed memory ie. Memory of multiple computers (Processor+memory+disk+OS)

Cost during training

- Megatron-Turing NLG:
  - 530 billion parameters
- Microsoft and Nvidia claim to have used hundreds of DGX A100 servers
  - Each server costs ~200,000 $
    - Add the networking cost, the infrastructure cost is ~100M$
  - Each server consumes 6.5kW of power
    - Add a comparable cooling cost!
- We will NOT do much about power consumption in this course!
  - But we will look at reducing model size as an important aspect!

Sizes of NLP models over the years

Figure 1. Trend of sizes of state-of-the-art NLP models over time

Model Size

- LLMs (Large Language Models) like GPT-3 and Bard are notoriously large.
  - But there are systematic approaches to reduce model size
    - Without compromising the accuracy too much.
  - We will look at model compression in this course!

Cost during Deployment

- When a model is deployed:
  - Requests come in and the model responds with inferences
    - E.g. if your model is a classifier:
      - For a new input x,
      - the response is $C(x)$ such that $x \in C(x)$
- Performance Parameters for this phase:
  - Throughput:
    - Number of inferences over a unit of time
  - Response Time: Time take to serve one inference
    - Consider the classifier example with a (data) cluster example!
      - Will there be a difference in response time?

Deployment Range

- The model (that has been trained) or an application using the model could be deployed on a variety of platforms:
  - A server (or a workstation)
    - What if the model is large?
  - The Cloud
    - The cloud can provision large infrastructure to host a large model:
      - Increase the number of servers hosting and accepting requests thereby improving throughput and response time!
        - But there is always delay
          - i.e., network latency in reaching a remote server or a server on the cloud (and getting the response back)
  - A mobile phone:
    - best end-user response time but cannot host large models.

Sequential vs. Batch

- BATCH (Assumption): Requests are collected together and sent
- Responses are collected together and sent

- Ans.
- Part (A) If the model server is parallel multiple threads or processes could respond in parallel thereby improving response time and throughput.

- Part (B) [Always] Messaging/Communication cost may be reduced:
- Communication cost = setup-cost + transmission cost
  - set-up cost is fixed per message
  - Transmission cost is proportional to the length of the message

Content & Pedagogy

- Focus on systems, programming techniques, and analysis
  - Pragmatics and Implementation to be learnt by doing – enabled by Assignments and Project.

- Lecture Sessions are expected to be interactive:
  - students are expected to raise questions and
  - the instructor will ask questions (which the students are expected to answer)

Evaluation

- Mid-term test and final exam – centrally scheduled by BITS
  - A Total of 55% weight = 25% for test + 30% for final exam
- 1 Assignment and 1 Project : Team exercises, Take-home
  - (15+30 =) 45% weight

Assignment and Project

- They are meant for learning
  - Expected:
    - One complex-end-to-end piece of optimization completed
    - One cutting-edge optimization technique learnt
  - Team-work with identifiable and quantifiable individual contributions
    - Evaluation both at team level and individual level

Deployment Range

- The model (that has been trained) or an application using the model could be deployed on a variety of platforms:
  - A server (or a workstation)
    - What if the model is large?
  - The Cloud
    - The cloud can provision large infrastructure to host a large model:
      - Increase the number of servers hosting and accepting requests thereby improving throughput and response time!
        - But there is always delay
          - i.e., network latency in reaching a remote server or a server on the cloud (and getting the response back)
  - A mobile phone:
    - best end-user response time but cannot host large models.

Sequential vs. Batch

- BATCH (Assumption): Requests are collected together and sent
- Responses are collected together and sent

- Ans.
- Part (A) If the model server is parallel multiple threads or processes could respond in parallel thereby improving response time and throughput.

- Part (B) [Always] Messaging/Communication cost may be reduced:
- Communication cost = setup-cost + transmission cost
  - set-up cost is fixed per message
  - Transmission cost is proportional to the length of the message

Content & Pedagogy

- Focus on systems, programming techniques, and analysis
  - Pragmatics and Implementation to be learnt by doing – enabled by Assignments and Project.

- Lecture Sessions are expected to be interactive:
  - students are expected to raise questions and
  - the instructor will ask questions (which the students are expected to answer)

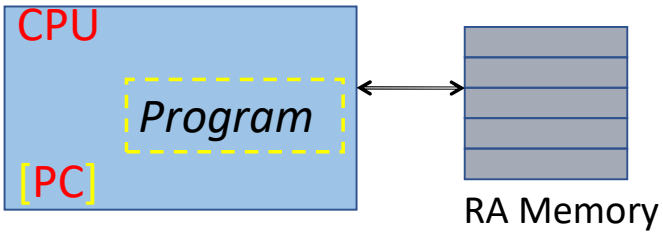
Parallel Programming Models:

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

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*)

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*

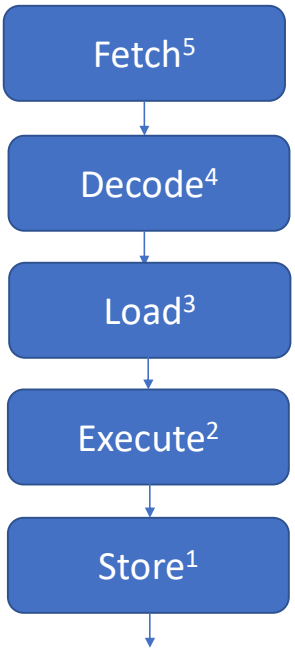
If separate circuitry is designed for each stage-
so that the stage take the same amount of time -
then a sequence of instructions can be executed in a pipeline

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

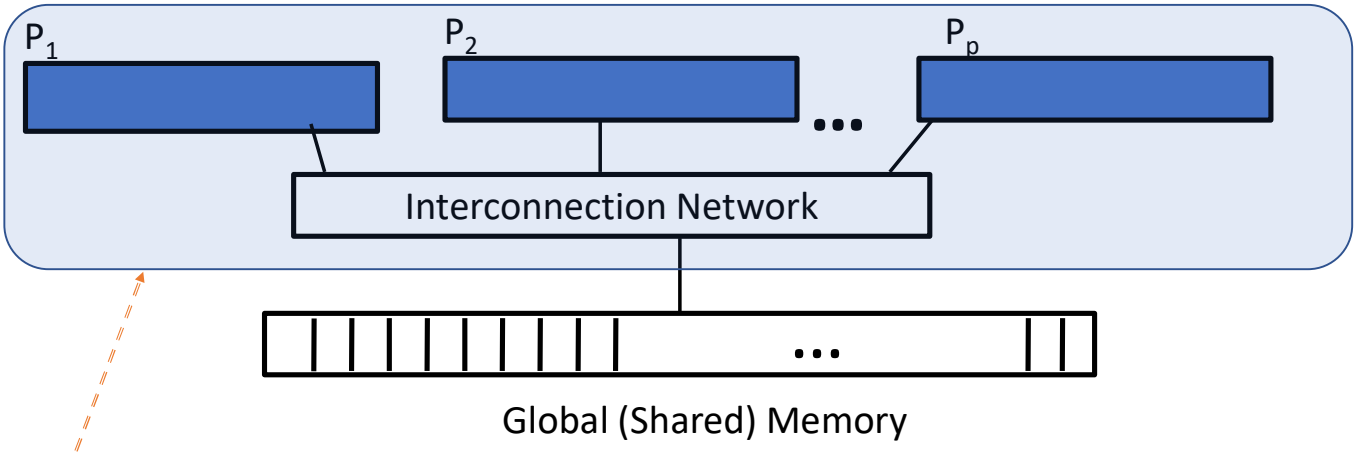
- STORE operations may be executed *asynchronously*:
  - i.e. processor does not wait for data to be stored in memory
    - Store buffers (i.e., buffer registers inside the processor) are used to store the data temporarily
    - while data is transferred to memory without processor involvement

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

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!

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Session 3: 17 Dec. 2023

Parallel Programming Models

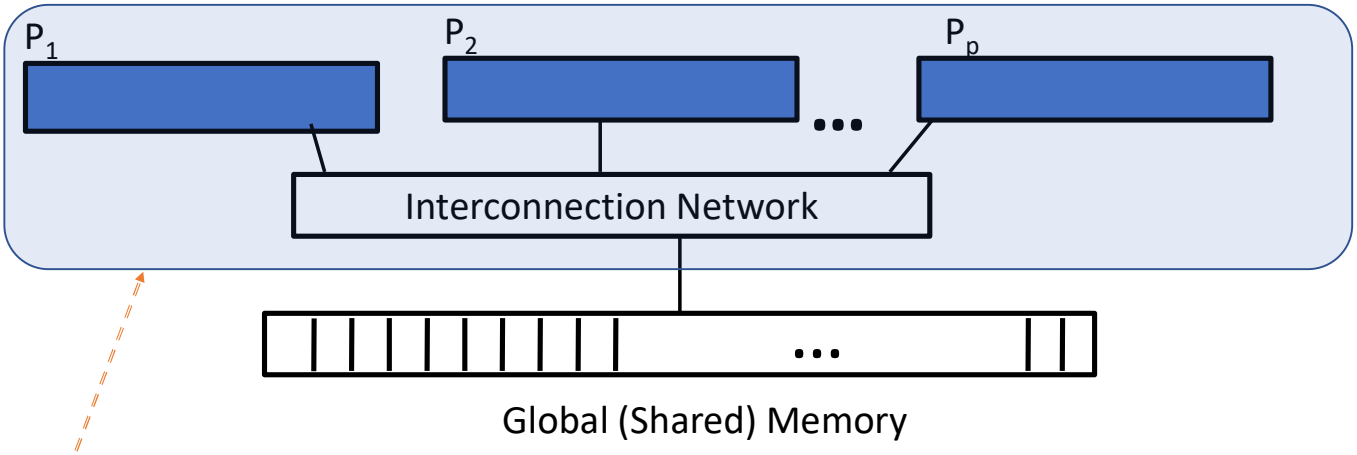
- Top Down Design
- Map-Reduce Pattern
- Task-Parallel and Request-Parallel

Parallelization of ML Algorithms - Examples

[continued.]

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

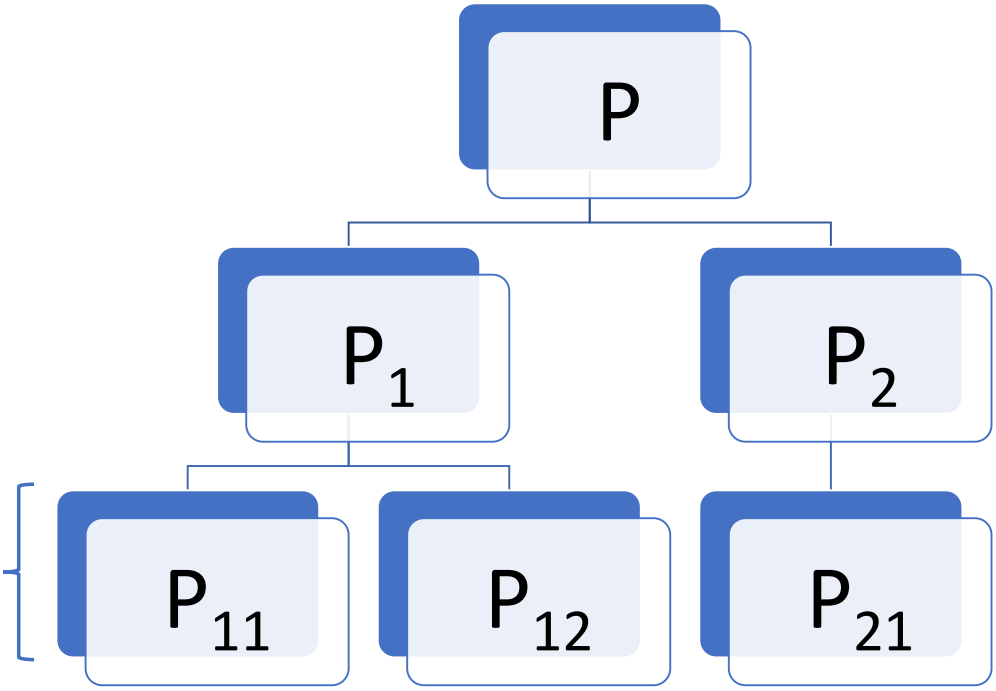
Algorithm Design

- 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!

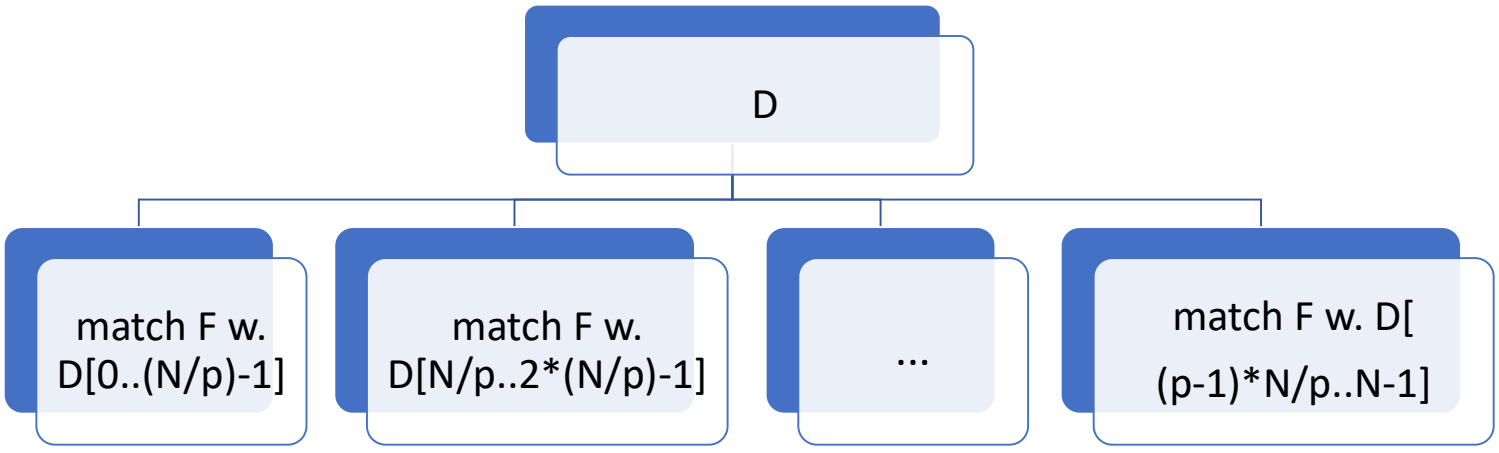
Data Parallel

- 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.**

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Session 3: 4 Jun. 2023

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

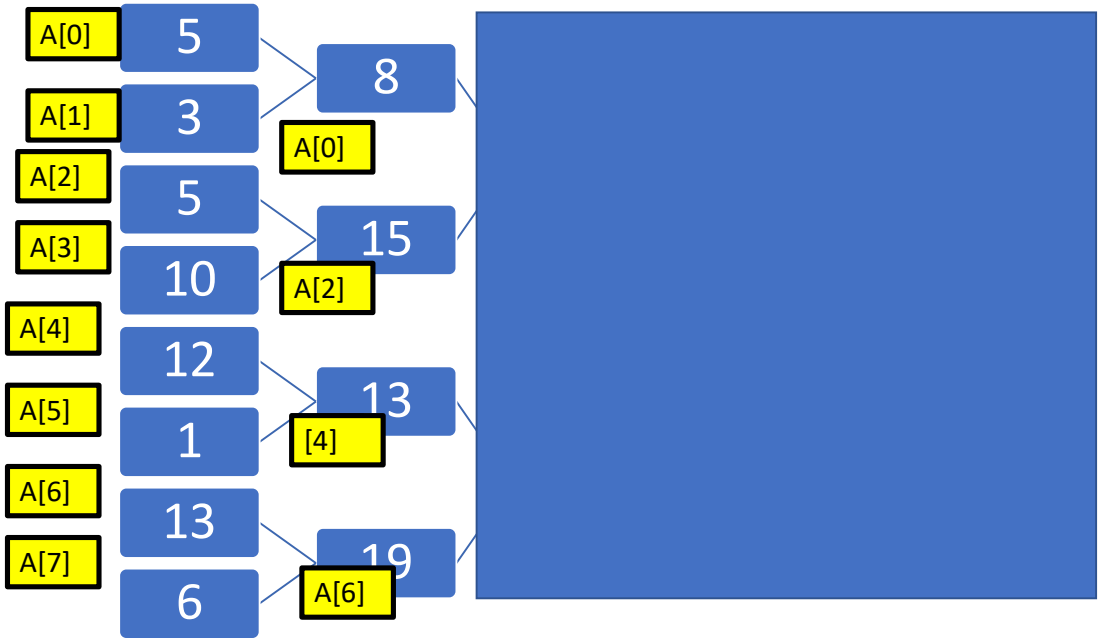
- Exercise:
  - Implement the data-parallel examples (previously discussed)

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***

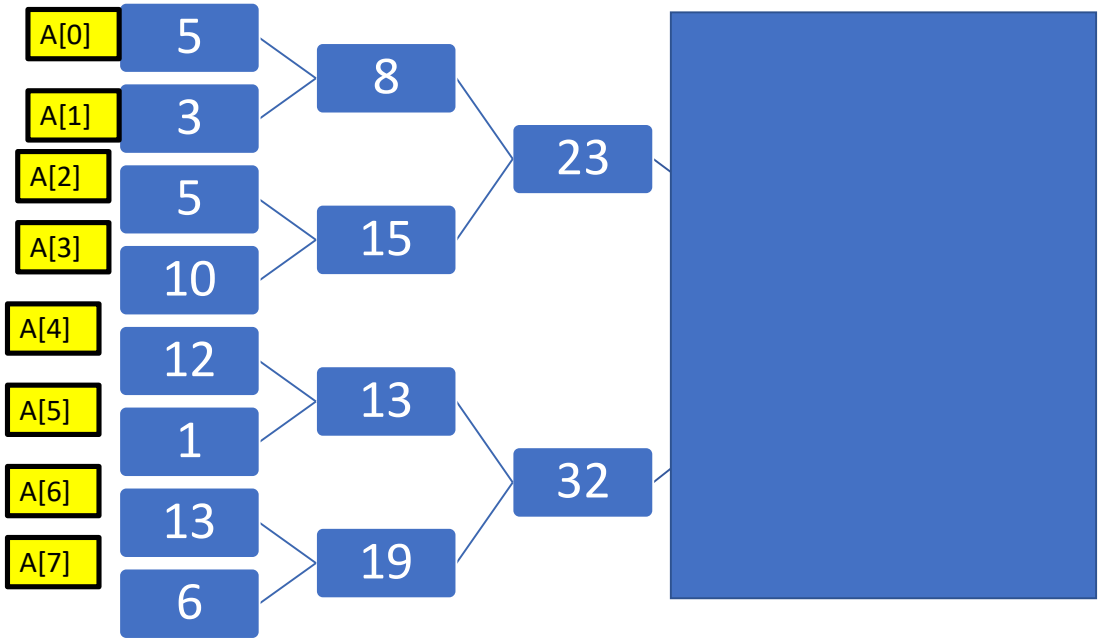
(Inverse) Tree-Parallel Algorithm : Summation

(Inverse) Tree-Parallel Algorithm : Summation



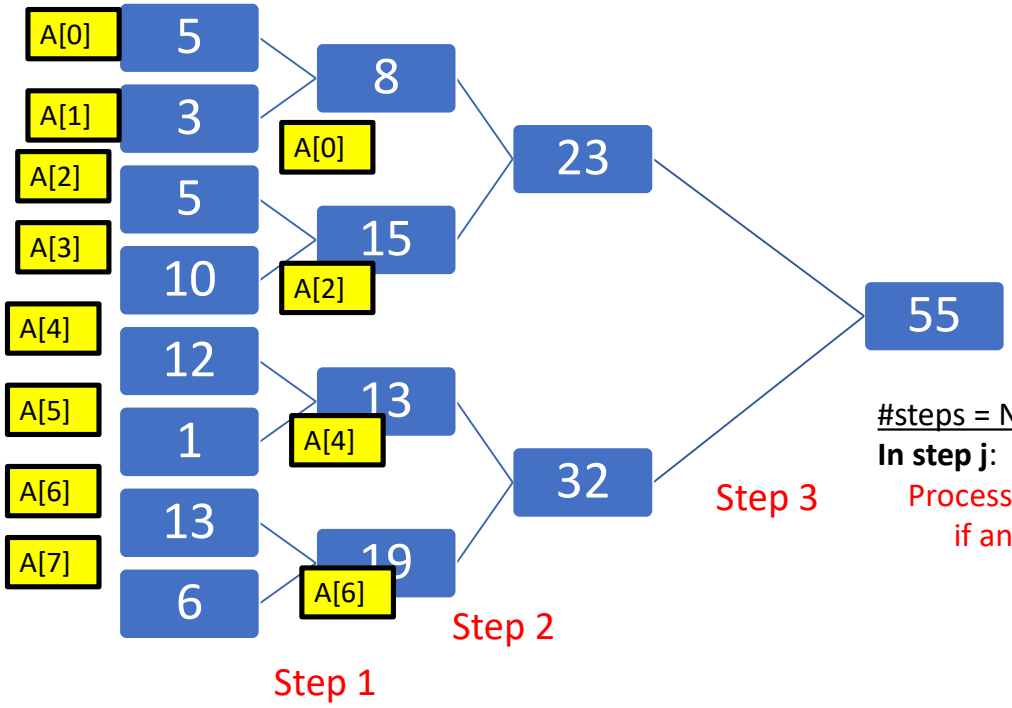
Step 1

(Inverse) Tree-Parallel Algorithm : Summation



Step 2

(Inverse) Tree-Parallel Algorithm : Summation

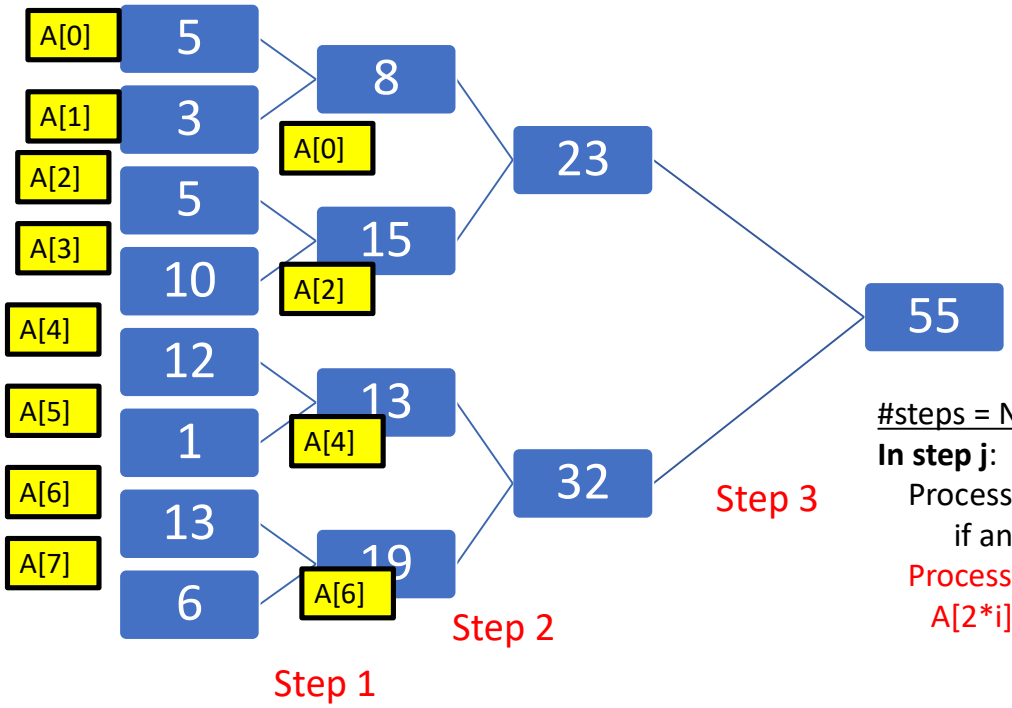


#steps = N for 2^N inputs

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

Distance between self and (processor holding) the other data

Decide which processors i participate in step j

(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 θ(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/logn)
        - 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) 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

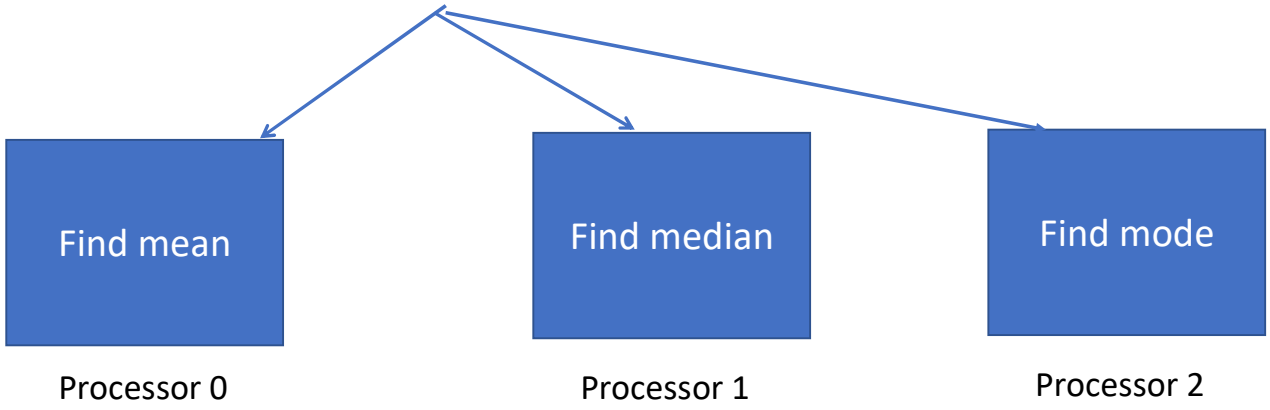
- Map-reduce is supported by different middleware platforms:
  - In particular, **Apache Spark** supports map-reduce on multi-core systems and clusters

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

Task Parallelism

Task Parallelism – Example

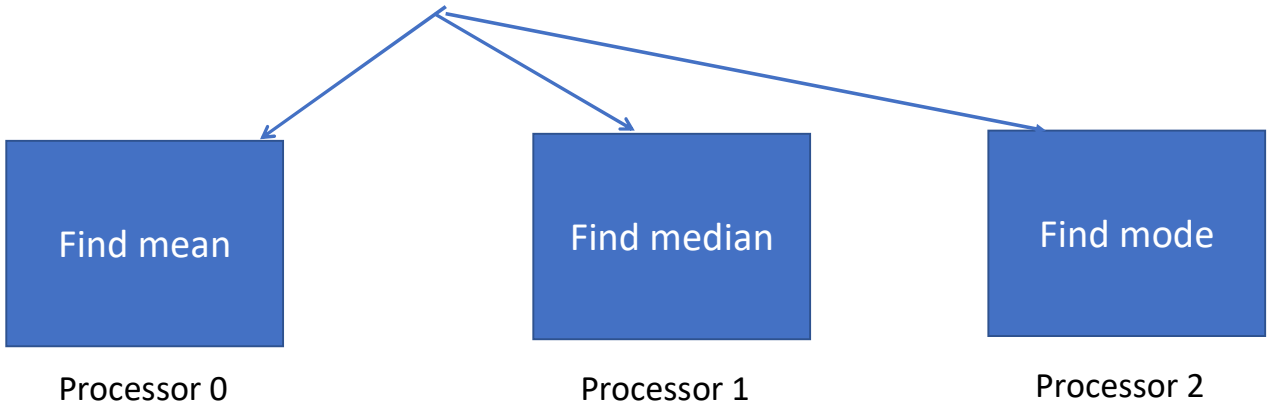
Problem: Given a list Ls of numeric values find the *mean*, *median*, and *mode*.



Task Parallelism – Example

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

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

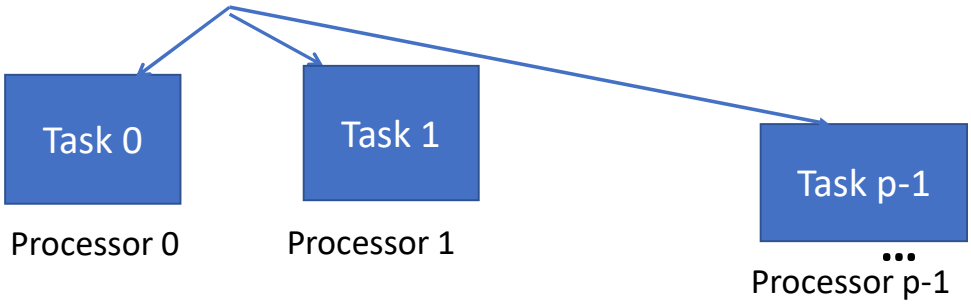


Task Parallelism

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)

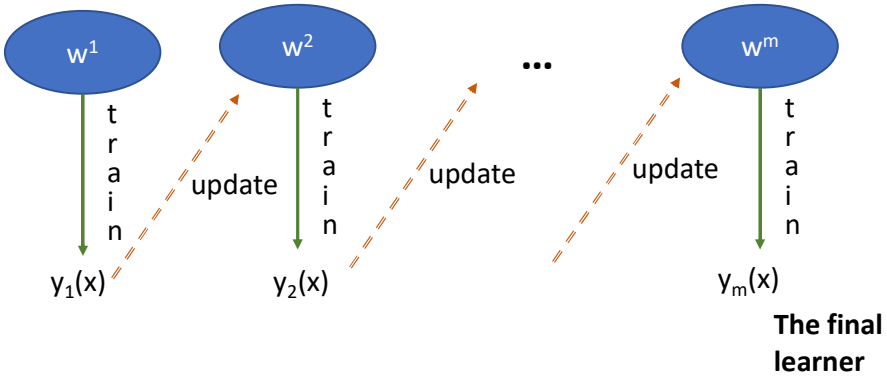
AdaBoost (or Adaptive Boosting) [*contd.*]

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.]

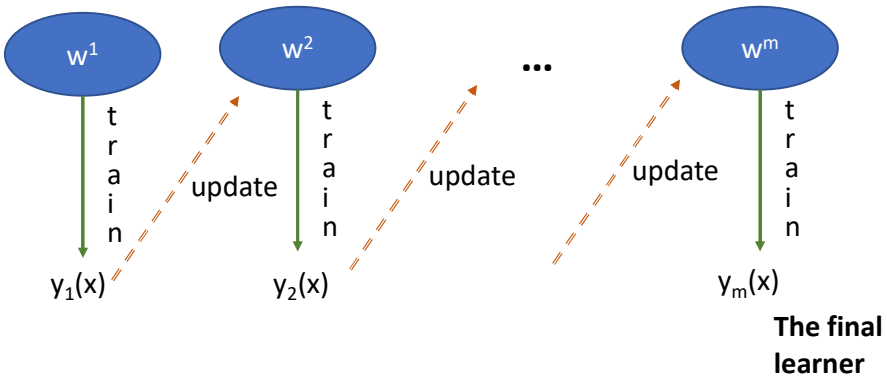
1. Initialize the data weighting coefficients $W^1 = 1/N$ for n= 1..N
2. for j = 1 to m
   1. Train a learner $y_j(x)$
   2. evaluate error e_j
   3. Update weighting coefficients $W^{j+1} = f(W^j, y_j, e^j)$
3. Make predictions using the final model y_m

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



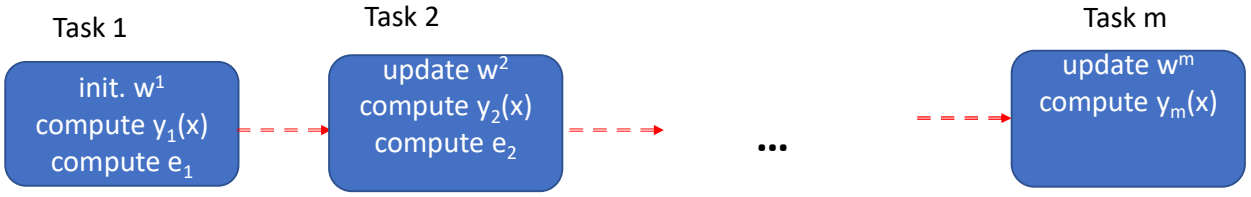
AdaBoost: Pipelining?

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

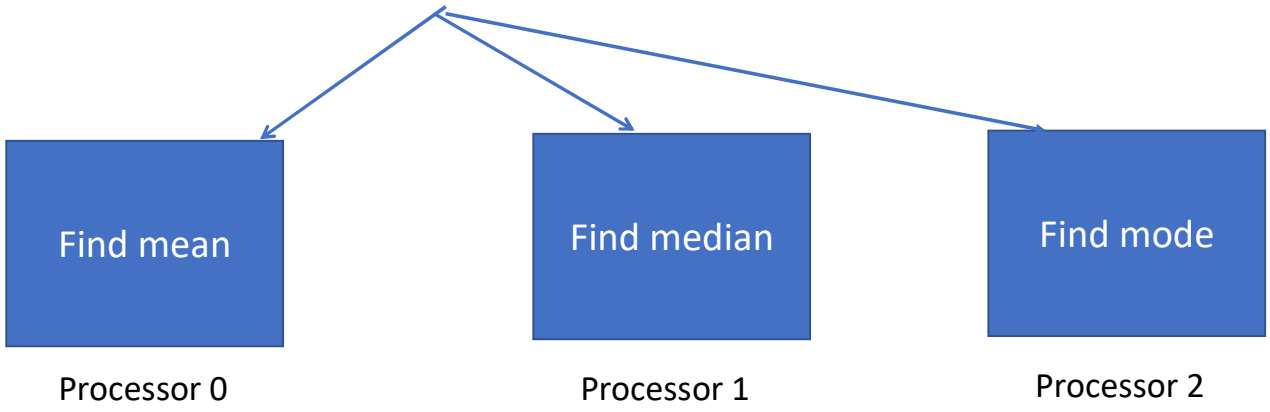


This pipeline provides speedup(m) > 1
only if computation of y_j and e_j can proceed in parallel with update w^{j+1}

Task Parallelism

Task Parallelism – Example

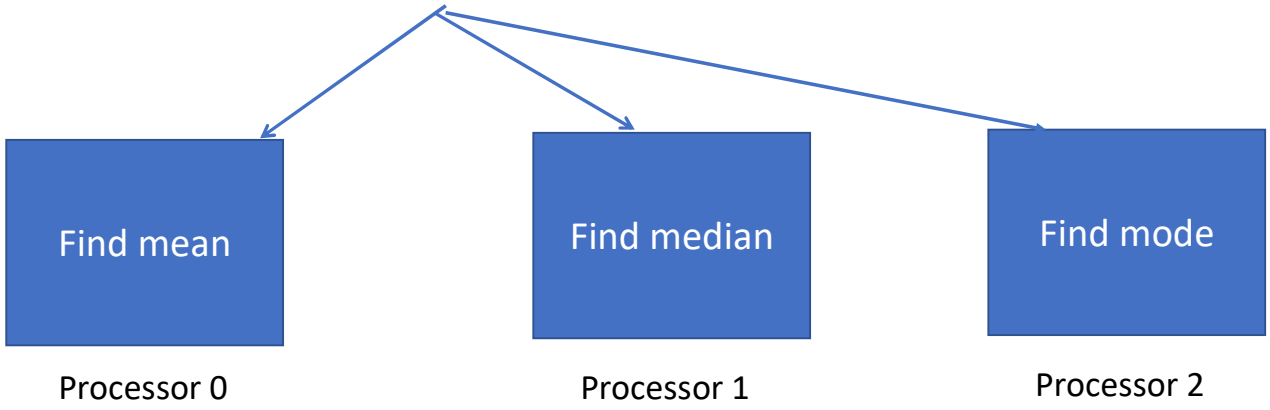
Problem: Given a list Ls of numeric values find the *mean*, *median*, and *mode*.



Task Parallelism – Example

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

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

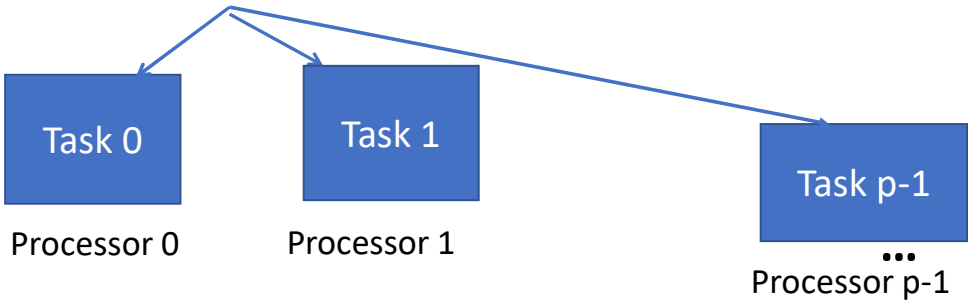


Task Parallelism

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.]

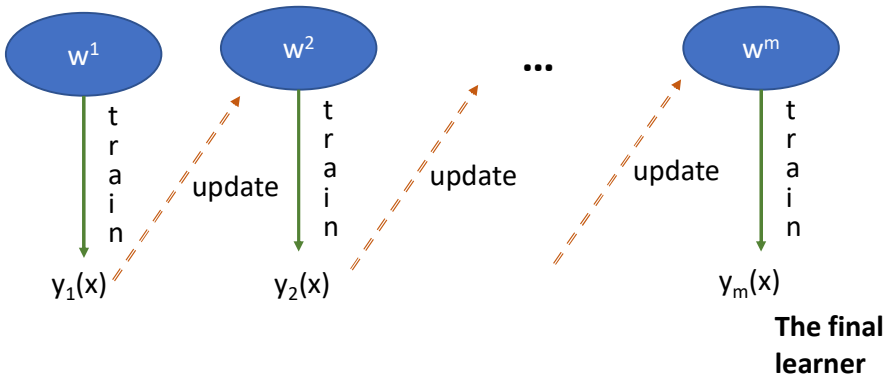
1. Initialize the data weighting coefficients $W^1 = 1/N$ for n= 1..N
2. for j = 1 to m
   1. Train a learner $y_j(x)$
   2. evaluate error e_j
   3. Update weighting coefficients $W^{j+1} = f(W^j, y_j, e^j)$
3. Make predictions using the final model y_m

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

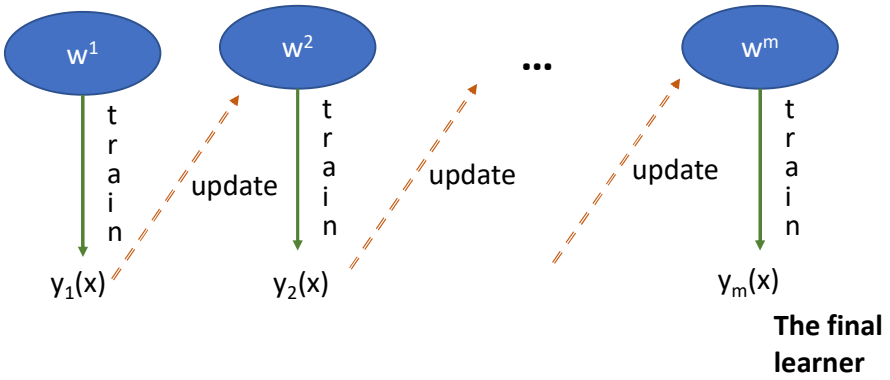
There is a sequential dependency!

This is not easily parallelizable!



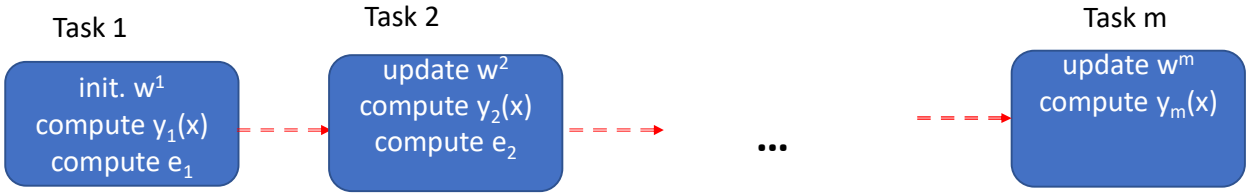
AdaBoost: Pipelining?

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}

innovate achieve lead

BITS Pilani
WILP

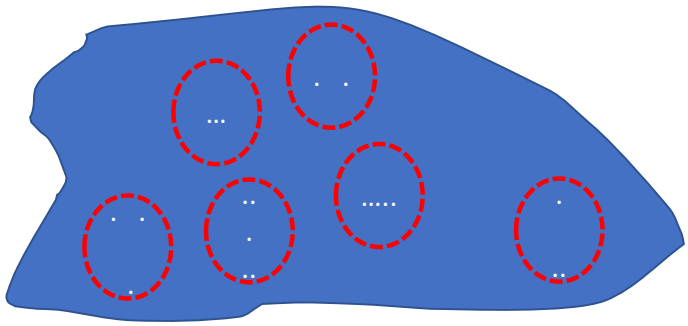
AIML CLZG516
ML System Optimization

Parallelization of ML Algorithms
- Example: k-Means
- Issues with Large Data Size

Scale-out Clusters – Distributed Memory Programming

Example: Data Clustering using k-Means

- Data Clustering is a classic data analytics problem:
  - Given a set of data points group them into disjoint subsets – clusters – such that:
    - Each cluster is cohesive
    - Different clusters are well-separated



Points are in Euclidean space

K – means Clustering

Inputs: Dataset D, A positive integer k

Output: A partition Cs of D with size k

(i.e., k disjoint clusters covering all points in D)

Approach:

1. Chose k data points (as representatives) from D, say c_1, c_2, … c_k
2. Assign each point x in D to the cluster C_j ;
   whose that has the closest center c_j
3. Choose k new representatives based on
   minimizing local average distance within each cluster [Notion of ***cohesion***]
4. Iterate steps 2 and 3 until (the cluster centers converge)

K – means Clustering using map-reduce

- Step 1: "select representative points" for clusters C_j = { c_j } for j=1 to k
- Step 2:
  - map "compute distance" on D x Cs where Cs is the set of clusters
  - map "assign point to the closest cluster" on D
    - This requires: reduce min on point-cluster distances
- Step 3: for each cluster C_j compute its centroid (i.e., mean)
  - map on Cs:
    - c_j = (reduce + C_j)/|C_j|
- Repeat Steps 2 and 3 until all c_j converge

D x Cs = { (x, c_j) … }
map comp_dist D x Cs

K – means Clustering using map-reduce

- Step 1: "select representative points" for clusters
- Step 2:
  - map "compute distance" on D x Cs where Cs is the set of clusters
  - map "assign point to the closest cluster" on D
    - This requires: reduce min on point-cluster distances
- Step 3: for each cluster C_j compute its centroid (i.e., mean)
  - map on Cs:
    - c_j = (reduce + C_j)/$|C_j|$
- Repeat Steps 2 and 3 until all c_j converge

{ (xi, d1j) }
= map comp_dist D
xi is a point in D
dij = distances of xi to clusters

reduce min dij

This reduce is required to return the cluster (with the min distance) and not the min distance:

Refer to reduce-key vs. reduce-val in Spark!

K – means Clustering

- Exercise: Implement k-means clustering using map and reduce.
- [Hints:
  - Step 1: "select k representative points" for clusters (randomly)
  - Step 2
    - map "compute distance" on D x Cs where Cs is the set of clusters
    - map "assign point to the closest cluster" on D
      - This requires: reduce min on point-cluster distances
  - Step 3b: compute the centroid (i.e., mean of) C_j
    - c_j = (reduce + C_j)/$|C_j|$

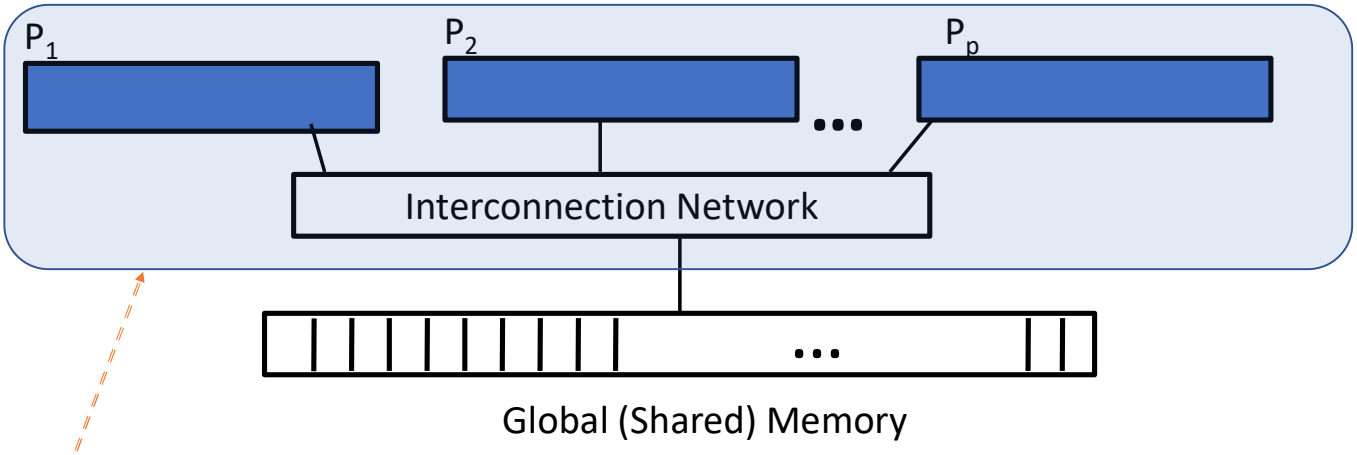
This follows a programming pattern named ***iterative map-reduce*** where map-reduce programming steps applied inside a loop.

Exercise: Speedup of k-means using map-reduce

- For each of the steps:
  - Calculate the speedup (and the number of processors)
- T_{seq} = I*(|D|*(k+k)+(k*|C|)) [Step 2: k distances req. k steps;
- min. computation req k-1 steps;
  - I is number of iterations
- T_{par} (p) = I*(|D|/p*(k+k)+(|C|) - assuming step 3 is done with only k processors; |D|/p points per processor in step 2
- p processors; k<p
- Speedup (p) = T_{seq} / T_{par} = (|D|*2*k + k*|C|)/((|D|/p)*(k+k)+|C|)
- ~p (close to ideal)

Parallel Programming: *Shared Memory* Model

So far we have looked at a target environment that uses a shared memory model:



e.g. a multi-core chip

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

Large volumes of Data

- When the volume of data that we have to process is in 100s of GB if not in TB,
  - Then all the data cannot be kept in one computer
  - And brought into memory for processing
- We a model where data can be stored on multiple computers (i.e., their hard disks)
  - All of which participate in computing.
- This leads us to a distributed computing model (aka message passing model)

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

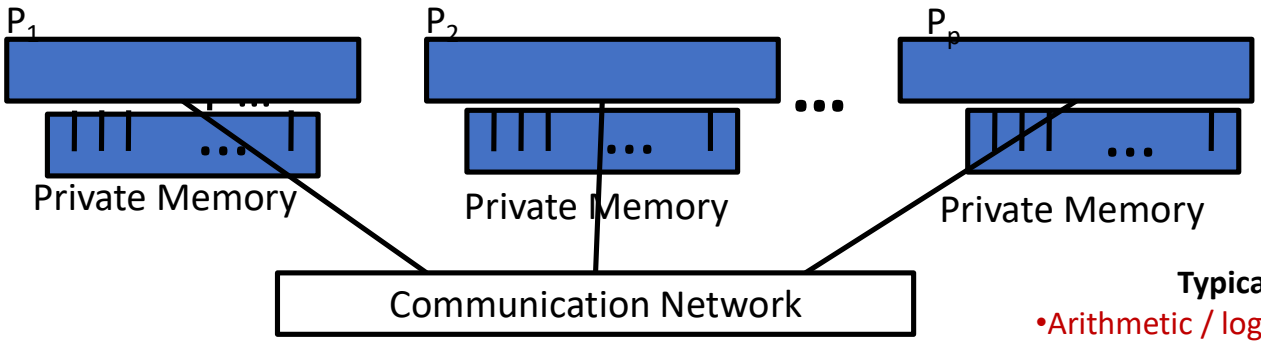
Session 5: 18 Jun. 2023

Distributed Memory Model

Scale-out Clusters – Distributed Memory Programming

Algorithm Design – Parallel: *Distributed Memory* Model

Target environment:



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

Distributed Programming:
- a program is made of multiple processes
- *each process runs on a separate computer*
- *processes exchange messages (i.e., data for collaboration)*

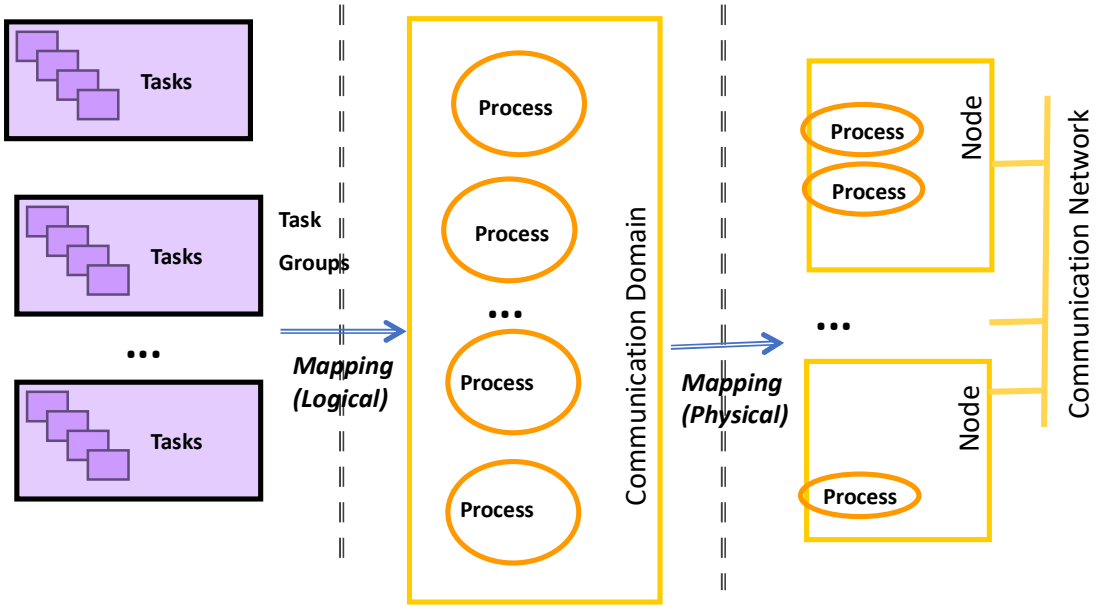
e.g. a cluster

Parallel / Distributed Computing

- A parallel or distributed program is made of multiple tasks that *collaborate* (to achieve a common outcome).
  - Collaboration is achieved by communication:
    - exchange data using shared memory
      - i.e. Task A writes to location L; Task B reads from location L
    - exchange data by passing messages
      - i.e. Task A sends a message to Task B; Task B receives the message from Task A

- Multiple processes each with its own address space:

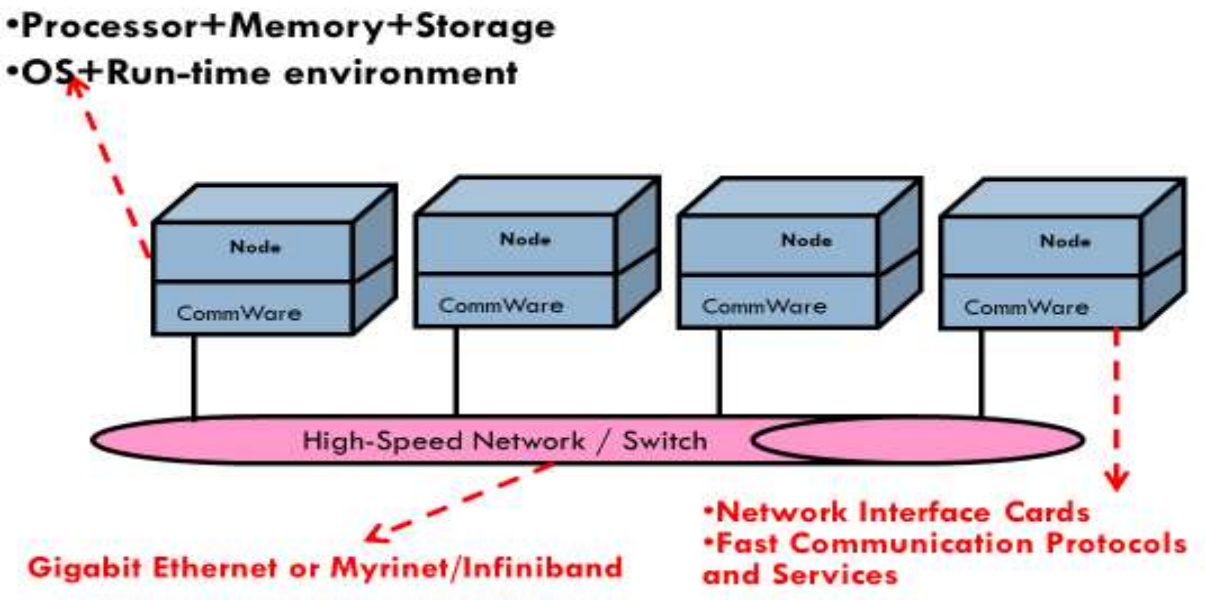
E.g. processes run on nodes connected in a network : (i) each node runs its own OS and (ii) each process is allocated its own (logical) address space that is mapped onto the (physical) resources of that node



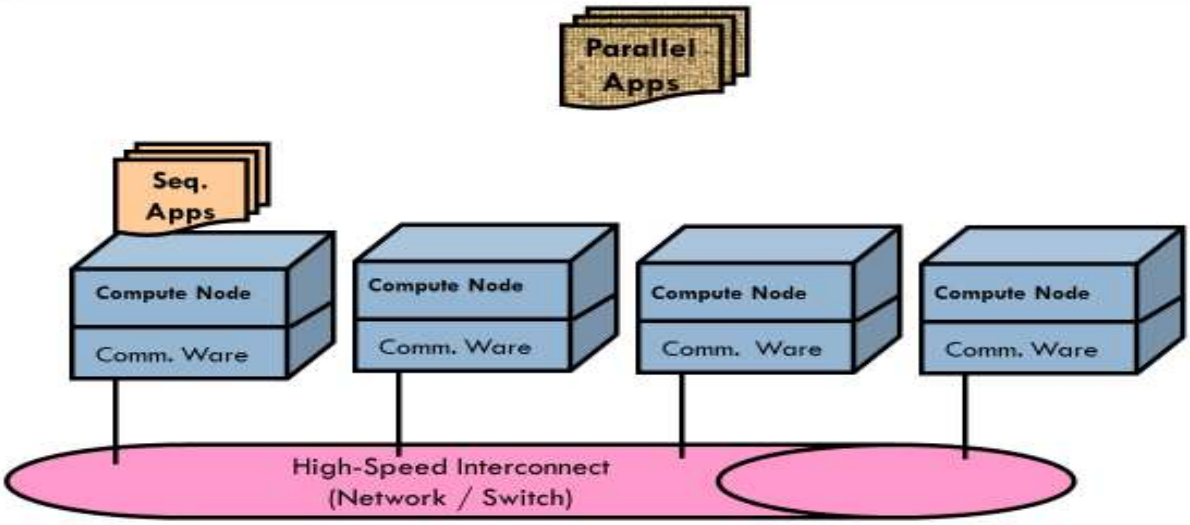
(Compute) Clusters

- Clusters are a modern example of distributed computing often used for high performance or big data computing.
  - Search engines - in the mid-90s - were using supercomputers at the back-end.
  - High obsolescence rate:
    - The supercomputers were getting obsolete (or unable to meet the computing needs) in five years or less.
  - Replacing a super-computer every five years was not cost-effective.
    - Clusters were available since 1980s and
    - Google realized that they are cost effective if older compute-nodes are replaced incrementally.
- Today, they are known as *scale out clusters* or *commodity clusters* and are used
  - with tens of thousands of nodes by Google, Facebook, Netflix, and others
  - for large scale processing of data.

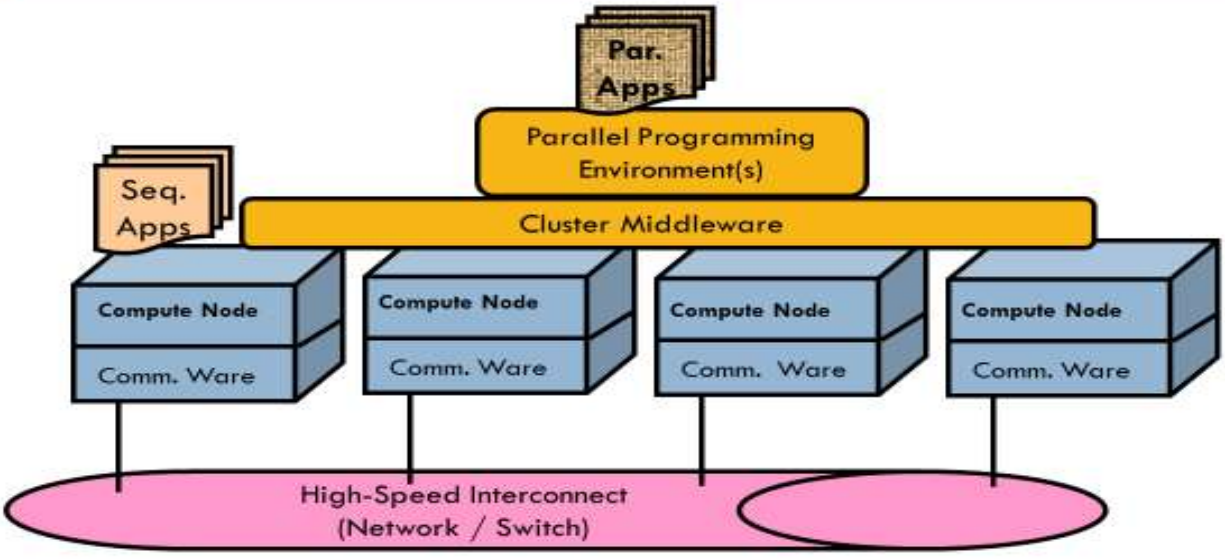
Typical Cluster: Components - Base



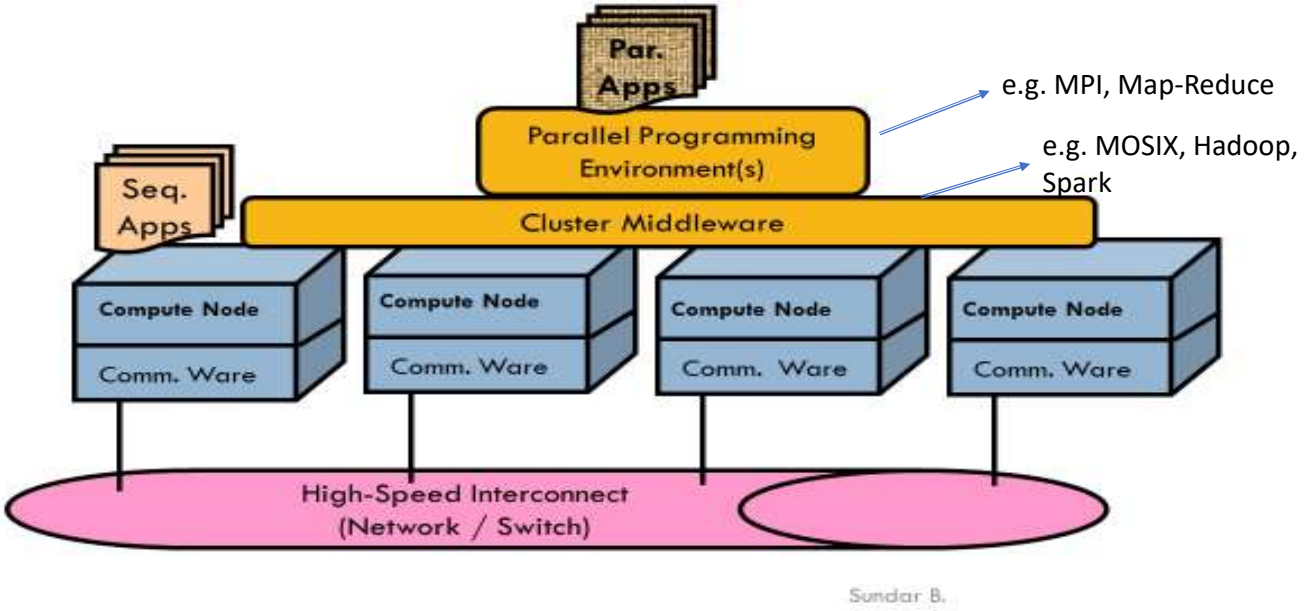
Typical Cluster - Requirements



Typical Cluster Architecture



Typical Cluster Architecture



Scale-out Clusters

- Case in-point:
  - Search engines - in the mid-90s - were using supercomputers at the back-end.
  - High obsolescence rate:
    - The supercomputers were getting obsolete (or unable to meet the computing needs) in five years or less.
  - Replacing a super-computer every five years was not cost-effective.
    - Clusters were available since 1980s and
    - Google realized that they are cost effective if older compute-nodes are replaced incrementally.
- Today, they are known as *scale out clusters* or *commodity clusters* and are used
  - with tens of thousands of nodes by Google, Facebook, Netflix, and others
  - for large scale processing of data.

Hadoop and Spark

- Apache Hadoop and Spark are platforms that run on clusters and support map-reduce programming
- Hadoop supports programming with data stored on files
- Spark supports in-memory distributed programming with RDDs (Resilient Distributed Data)
  - Programmable Data structures
    - Distributed in the memories of multiple nodes in a distributed system (e.g. a cluster)

Exercise

- Implement k-means on Spark
- Calculate – on paper - speedup of k-means using a cluster:
  - Calculate communication cost
- Understand:
  - the difference between this and the previous calculation (for shared memory programming)
- Questions:
  - How do you distribute the data initially?
  - Cost?
  - Pattern?

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Distributed Algorithms – Communication Overhead

Distributed ML Algorithms

- Example: k-Means

- Model Parallelism

K – means Clustering

Inputs: Dataset D, A positive integer k

Output: A partition Cs of D with size k

(i.e., k disjoint clusters covering all points in D)

Approach:

1. Chose k data points (as representatives) from D, say c_1, c_2, … c_k
2. Assign each point x in D to the cluster C_j ;
   that has the closest center c_j
3. Choose k new representatives based on
   minimizing local average distance within each cluster [Notion of ***cohesion***]
4. Iterate steps 2 and 3 until (the cluster centers converge)

K – means Clustering using map-reduce

- Step 1: "select representative points" for clusters $C_j = \{ c_j \}$ for j=1 to k
- Step 2:
  - map "compute distance" on D x Cs where Cs is the set of clusters
  - map "assign point to the closest cluster" on D
    - This requires: reduce min on point-cluster distances
- Step 3: for each cluster C_j compute its centroid (i.e., mean)
  - map on Cs:
    - $c_j = (\text{reduce} + C_j)/|C_j|$
- Repeat Steps 2 and 3 until all c_j converge

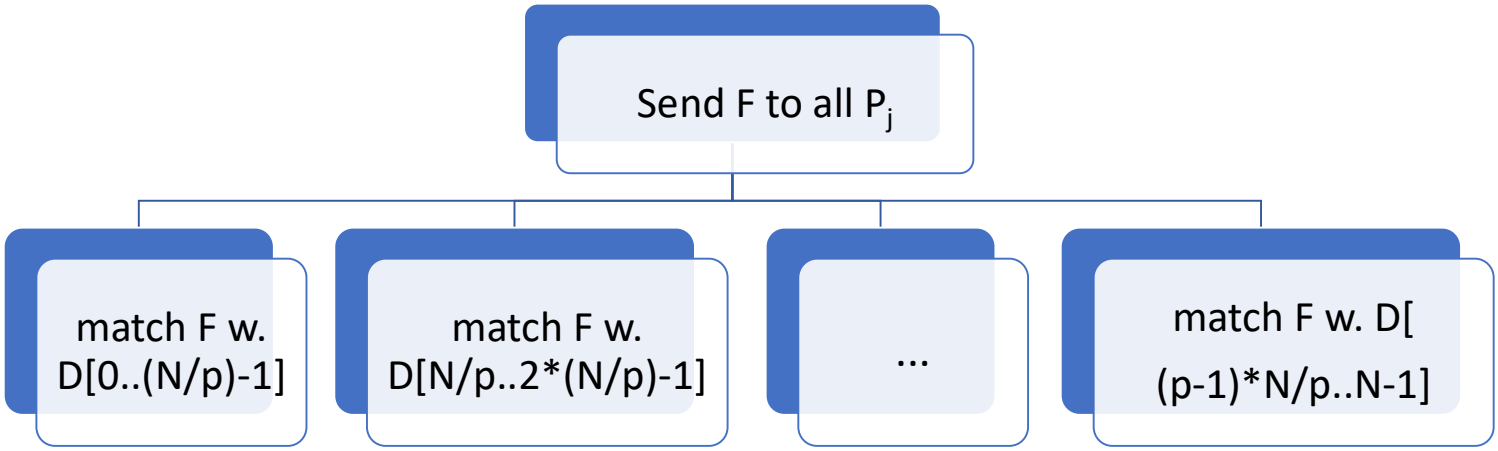
D x Cs = { (x, c_j) … }
map comp_dist D x Cs

Data Parallel Execution - Examples

Fingerprint Matching:

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

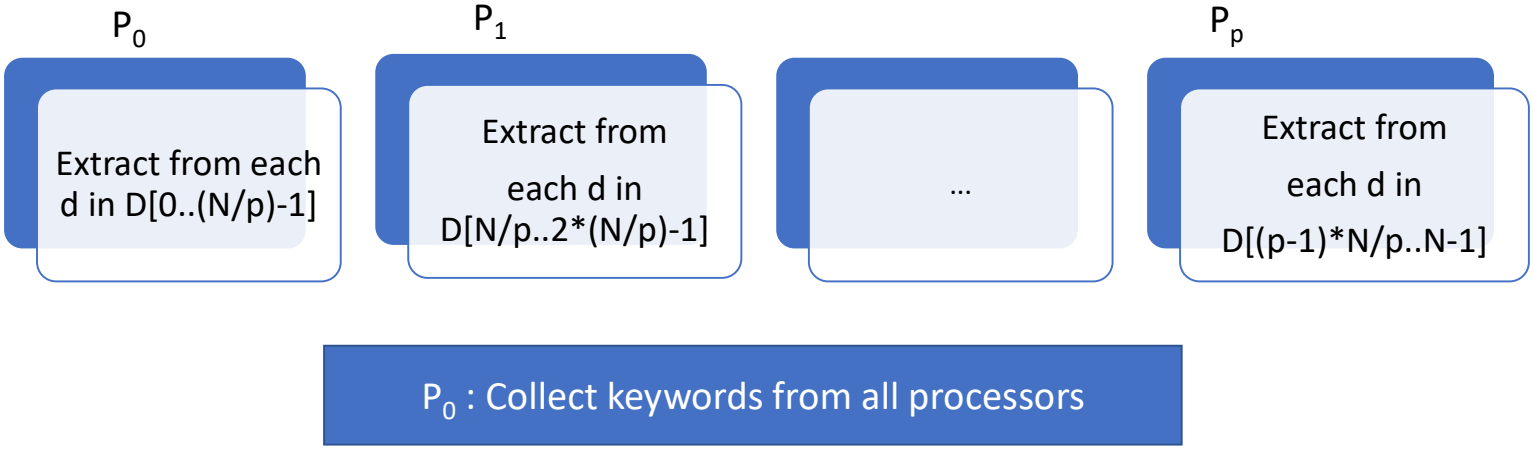
•Assume D is distributed.



Communication cost?

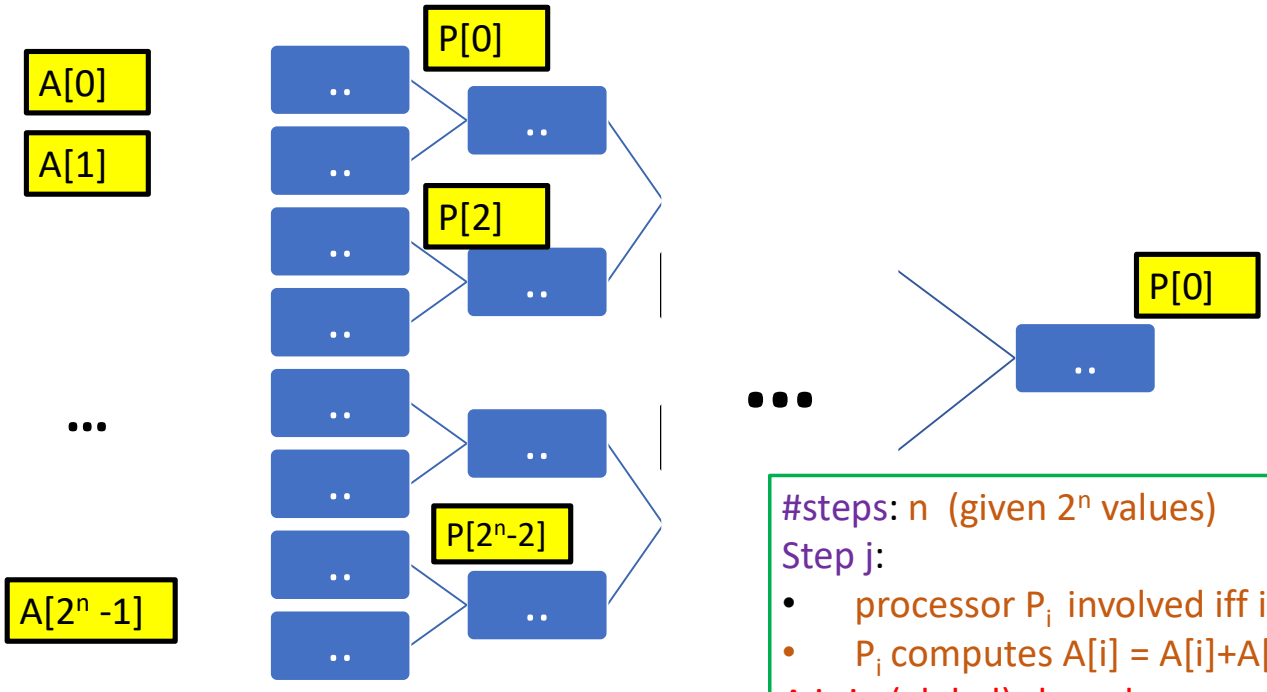
Data Parallel Execution - Examples

Extracting keywords from each document in a distributed collection D:
[Assume|D| = N]



Communication cost?

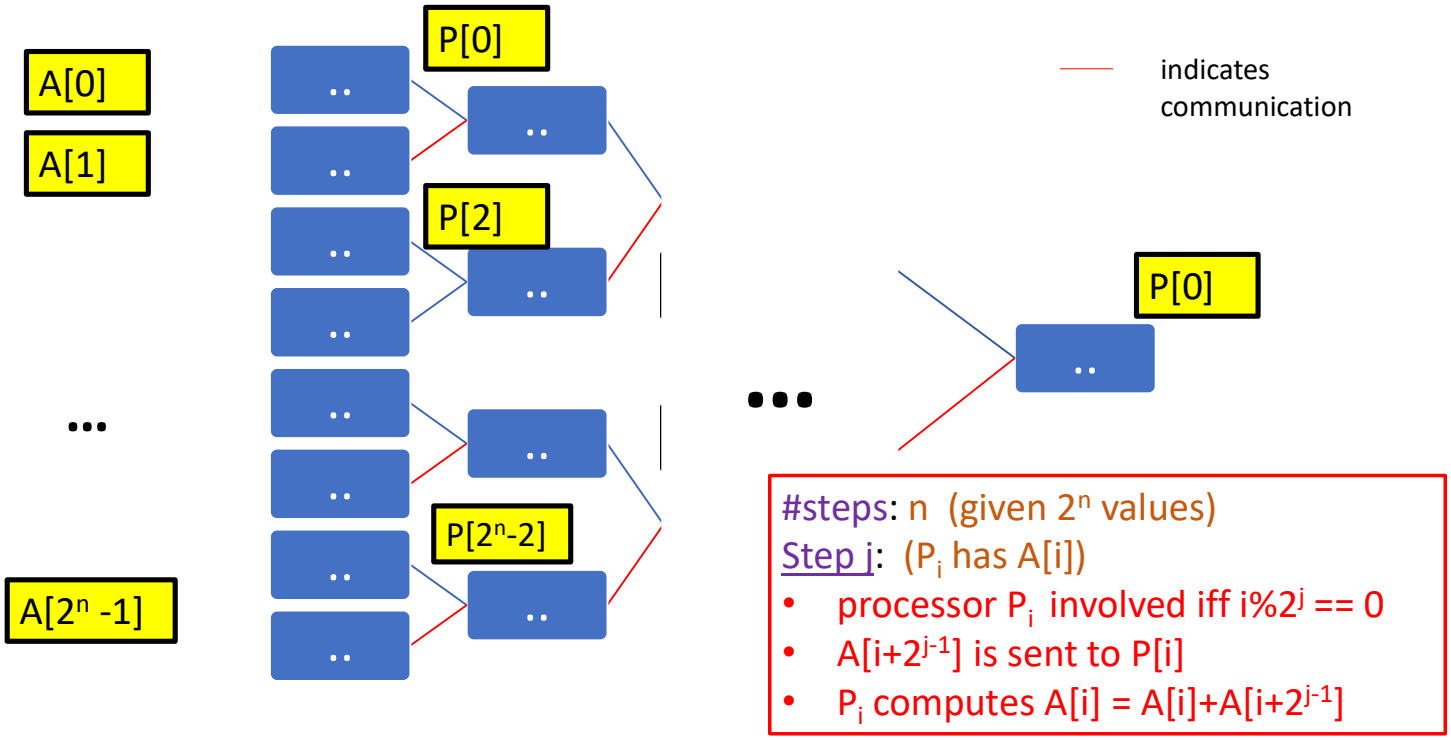
Example: Parallel Summation (Shared memory)



#steps: n (given 2^n values)
Step j:
- processor P_i involved iff $i\%2^j == 0$
- P_i computes $A[i] = A[i]+A[i+2^{j-1}]$

A is in (global) shared memory

Example: Parallel Summation (Distributed memory)



Algorithm Design – Speedup: Caveat

- Summation by reduction executed on a distributed system:
  - before processor P_i performs $A[i] = A[i] + A[i+2^{j-1}]$
    - there is communication involved: $A[i+2^{j-1}]$ sent from $P_{i+2^{j-1}}$ to P_i
- The time complexity of a reduction algorithm will increase
  - i.e. *speedup will decrease*.

- Speedup (of summation by reduction of **N** values with **N/2** processors):

$$S(N,p) = T_{seq}(N) / T_{par}(N,N/2)$$
$$= ((N-1)*T_{add}) / (\log N*(T_{add} + T_{msg}))$$

where T_{add} is the time taken for adding two values and T_{msg} is the time taken for sending (and receiving) a message

Algorithm Design – Speedup: Caveat

- Speedup (of summation by reduction of **N** values with **N/2** processors):

  $S(N,N/2) = T_{seq}(N) / T_{par}(N,N/2)$

  $\approx (N/\log N) * (1 / (1 + T_{msg}/T_{add}))$

- T_{msg} includes
  - set-up cost (for *send* and *recv*), which is typically fixed and
  - transmission cost (which depends on the length of the message)
- Typically, $T_{msg} >> T_{add}$

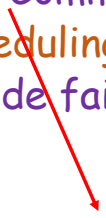
	1990s	Today (2020s)
T_{msg}	*ms*	*us*
T_{add}	*us* (or 10s of *ns*)	< 1*ns*

ms milli-seconds
us micro-seconds
ns nano-seconds

- Implication: Speedup << N/logN

Reduce as in Google's Map-Reduce

- Pragmatics of ***reduce*** (*similar to that of* **map**):
  - (Assumption) Input Data partitioned and stored
  - Management of Messaging / Communication
  - Spawning of processes, Scheduling, and Load Balancing
  - Recovery from process / node failures



Communication is managed transparently:

i.e., programming is easier,

but communication cost is still the same!

K – means Clustering: Algorithm outline

Inputs: Dataset D, A positive integer k

Output: A partition Cs of D with size k

(i.e., k disjoint clusters covering all points in D)

Approach:

1. Chose k data points (as representatives) from D, say c_1, c_2, … c_k
2. Assign each point x in D to the cluster C_j :
   that has the closest center c_j
3. Choose k new representatives based on
   minimizing local average distance within each cluster [Notion of ***cohesion***]
4. Iterate steps 2 and 3 until (the cluster centers converge)

K – means Clustering on a distributed system

- Assume D is distributed in **p** processors (or nodes)
  - Each processor p_i has D_i of size (|D|/p points)
- Step 1: "select representative points" for clusters C_j = { c_j } for j=1 to k
- Step 2:
  - Communicate c_j (for j = 1 to k) to all p nodes (i.e. processors)
  - On each processor p_i
    - for each point x in D_i
    - {
    - for j=1 to k { d_j = distance(x,c_j); }
    - assign x to the cluster with minimum d_j
    - } // each p_i has k clusters

…

K – means Clustering on a distributed system

- Assume D is distributed in **p** processors
  - Each processor p_i has D_i of size (|D|/p points)
- Step 1: "select representative points" for clusters $C_j = \{ c_j \}$ for j=1 to k
- Step 2:
  - Communicate c_j (for j = 1 to k) to all p nodes (i.e. processors)
  - On each processor p_i, "compute distances":
    - for each point x in D_i
    - {
    - for j=1 to k { d_j = distance(x,c_j); }
    - assign x to the cluster with minimum d_j
    - } // each p_i has k clusters
- Step 3: collect and distribute clusters to k different processors
  - for each cluster C_j in processor p_j
    - c_j = (reduce + C_j)/|C_j|
- Repeat Steps 2 and 3 until all c_j converge

K – means Clustering on a distributed system

- Assume D is distributed in **p** processors
  - Each processor p_i has D_i of size (|D|/p points)
- Step 1: "select representative points" for clusters $C_j = \{ c_j \}$ for j=1 to k
- Step 2:
  - Communicate c_j (for j = 1 to k) to all p nodes (i.e. processors)
  - On each processor p_i ← map *(distributed)*
    - for each point x in D_i
    - {
    - for j=1 to k { d_j = distance(x,c_j); } ← map *(shared mem.)*
    - assign x to the cluster with minimum d_j ← reduce *(shared mem.)*
    - } // each p_i has k clusters
- Step 3: collect and distribute the clusters to **k** different processors
  - for each cluster C_j in processor p_j (j=1 to k) ← map *(distributed)*
    - c_j = (reduce + C_j)/|C_j| ← reduce *(shared mem.)*
- Repeat Steps 2 and 3 until all c_j converge

K – means on a distributed system: Communication cost

- Assume D is distributed in **p** processors
  - Each processor p_i has D_i of size (|D|/p points)
- Step 1: "select representative points" for clusters C_j = { c_j } for j=1 to k
- Step 2:
  - Communicate c_j (for j = 1 to k) to all p nodes (i.e. processors) ← k values
  - On each processor p_j, "compute distances":
    - for each point x in D_i
    - {
    - for j=1 to k { d_j = distance(x,c_j); }
    - assign x to the cluster with minimum d_j
    - } // each p_i has k clusters
- Step 3: collect and distribute clusters to k different processors ← |D| values
  - for each cluster C_j in processor p_j
    - c_j = (reduce + C_j)/|C_j|
- Repeat Steps 2 and 3 until all c_j converge

K – means on a distributed system: Implementation

- Use MPI (Message Passing Interface) to program on a distributed system:
  - Defaults to a SPMD model (i.e. same program/code on multiple processors but different data)
    - i.e. map is implicit
  - Constructs available for send, recv, reduce
- Alternative:
  - Use Hadoop map-reduce
  - Data stored on files (of nodes in a cluster)

K – means on a shared memory system: Implementation

- Use OpenMP library or P-Threads library) to program on a shared memory) system:
  - Defaults to a SPMD model (i.e. same program/code on multiple processors but different data)
    - i.e. map is implicit
  - Constructs available for send, recv, reduce
- Alternative:
  - Use multi-threaded programming in Java

K-means on a commodity cluster

- A cluster of nodes (workstations/servers)
  - Each node has a multi-core processor
- Use MPI programming on the cluster (in C or in Java)
  - The code for each node can be multi-threaded
    - Use OpenMP or P-Threads

K – means on Spark

- Distributed memory model (i.e. a cluster) where
  - Every node can be a shared memory processor (i.e. a multi-core processor)
  - Supports map and reduce as constructs
  - Programming in Java or Scala or other languages
  - In-memory processing
    - i.e. data may be stored on files,
    - but map and reduce work on data structures (RDDs) stored in memory
      - Unlike on Hadoop where map and reduce operate on files

Task Parallelism – Example

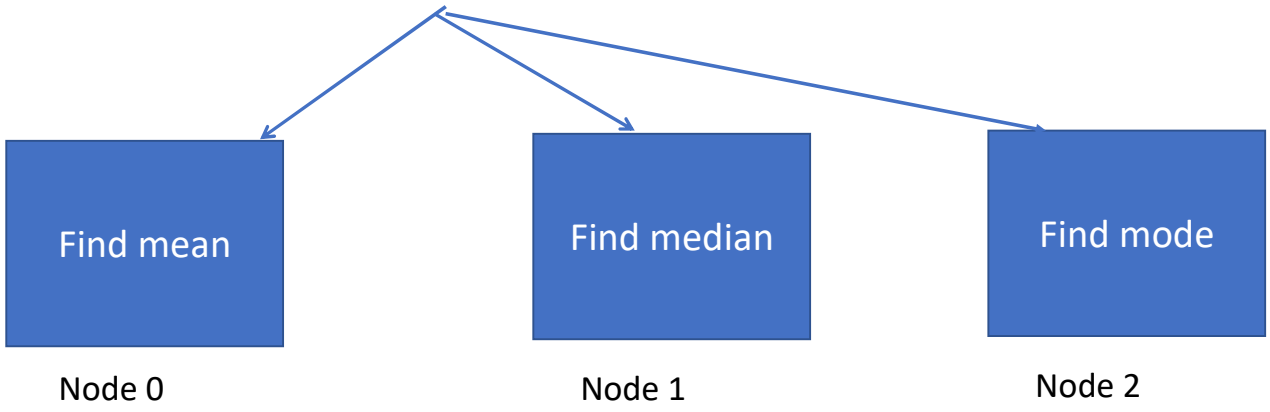
Recall Task Parallelism on Shared memory computers

The same is possible on Distributed memory systems!

Communication?

Assumption:

Input data is available on all nodes.

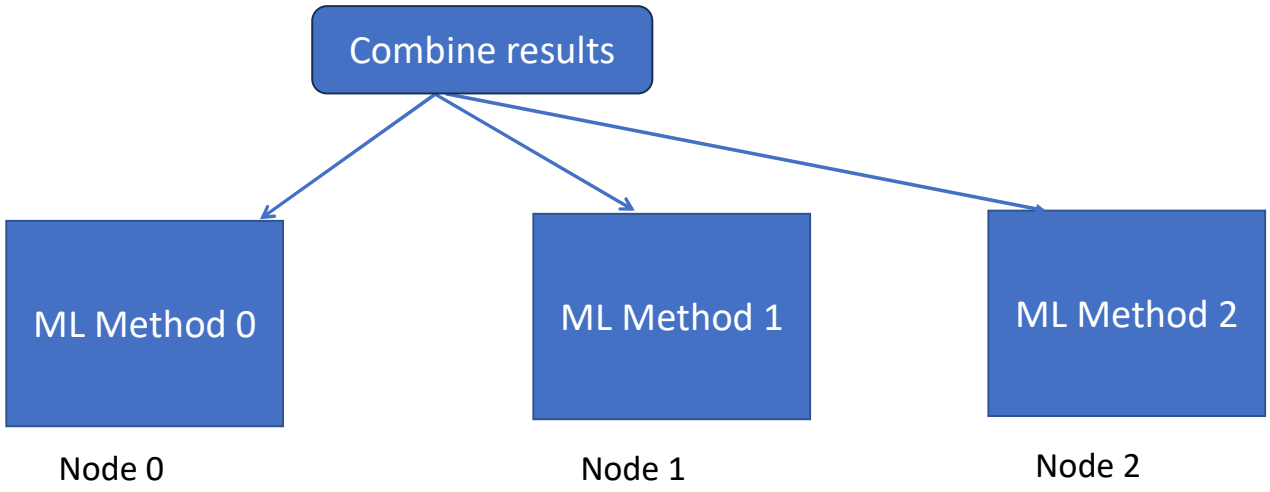


Ensemble Methods

- Task Parallelism – in the context of ML – is also known as Model parallelism
  - Recall the Bagging solution:
    - Multiple different models from the same data (via sampling)
  - Ensemble methods
    - Multiple different models from the same data (via different training methods)

Model Parallelism – Example: Ensembles

Assumption:
Input data is available on all nodes.



Communication Cost?

Introduction to GPGPU

GENERAL PURPOSE COMPUTATION ON GPU

Agenda

GPU as a Co-processor

GPU Programming Languages

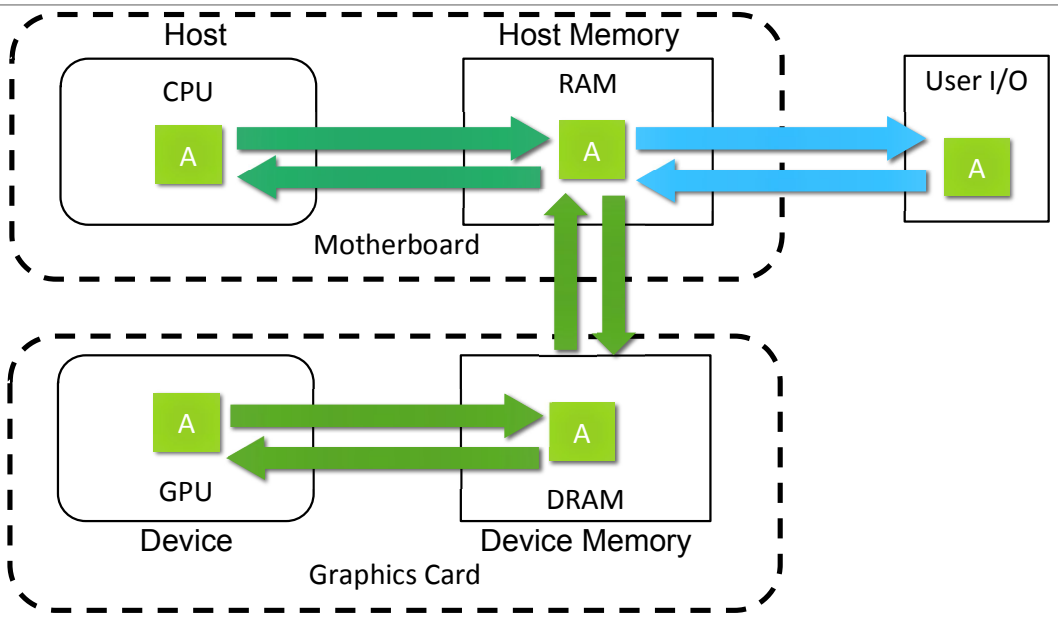
First CUDA Program

GPU Architecture

- Threading Model
- Memory Model
- Scheduling Model

GPU Optimization

GPU as a Co-processor Sequential Programming Model



GPU Architecture (History)

Problem : How to quickly convert mathematical model of computer graphics into actual signals to be sent to Display Unit?

- Break down the graphics processing into a pipeline of dedicated steps & design a hardware for each step
- Graphics Pipeline
  - Vertex Transform & Lighting
  - Triangle Setup & Rasterization
  - Texture & Pixel Shading
  - Depth Test & Blending
  - Framebuffer
- Many stages got added and more and more stages were made programmable
- Ultimately, multiple programmable stages were combined to give an **Unified Scalar Shader Architecture**

Courtesy: David Luebke, SIGGRAPH 2008

GPU Characteristics

- GPUs were designed for graphics processing where vertices & pixels are processed independently
  - Each processing step is usually arithmetic intensive i.e., multiple operations are applied in between memory access
  - So, less control logic and more of arithmetic logic is required

- GPUs are designed for tasks that can tolerate high latency as long as it can process a lot of tasks in one go (i.e., high latency, high throughput)
  - So, data caching is not a priority

- So, more chip area can be given to ALUs instead of Control Logic and Caches. Therefore, a lot of GPU threads (10s of thousands) can (should) execute at a time.

CPU Vs. GPU: Transistor Allocation Ratio

Courtesy: NVIDIA CUDA Programming Guide

GPU Programming Languages

- Kernel Programming Languages
  - Explicit and fine level control over threads and memory operations
  - Example:
    - CUDA for Nvidia GPUs
    - HIP for AMD and Nvidia GPUs
    - OpenCL for AMD and Nvidia GPUs

- Directive-based Programming Languages
  - No control over threads, compiler automatically generates code for parallelization
  - Example:
    - OpenMP for AMD and Nvidia GPUs -□ my job
    - OpenACC for AMD and Nvidia GPUs

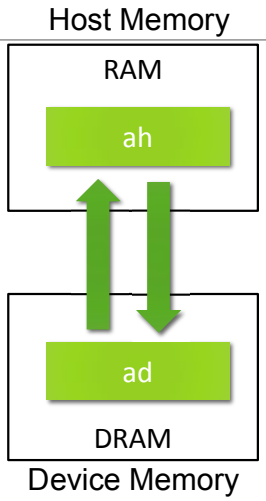
First CUDA Program

Problem-
- Write a program in CUDA to find square of first 500 whole numbers stored in an array.
- Serial implementation-

```
#include <stdio.h>
int main()
{
    int *a, i, N=500;
    a = (int*) malloc (sizeof(int) * N);
    for(i=0; i<N; i++)    a[i] = i;

    // pragma omp parallel target for map(tofrom:a)
    for(i=0; i<N; i++)    a[i] = a[i] * a[i];
    for(i=0; i<N; i++)
        printf("Square of %d = %d\n", i, a[i]);
    return 1;
}
```

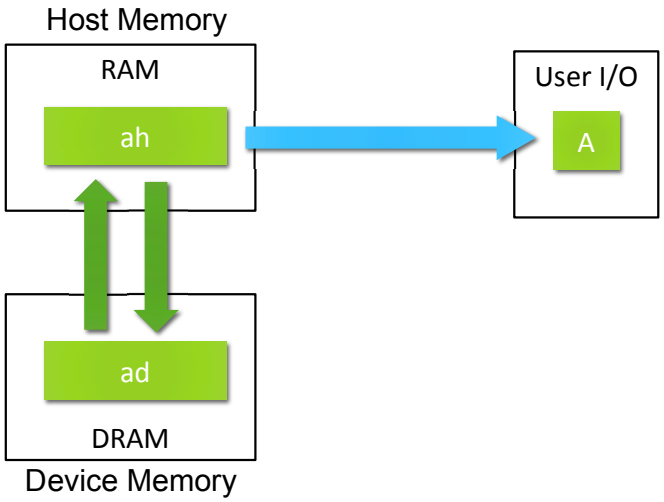
Parallel Implementation

```
#include <stdio.h>
#include <cuda.h>
int main()
{
    int *ad, *ah, i, N=500;

    //allocate memory on host and device
    ah = (int*) malloc (sizeof(int) * N);
    cudaMalloc((void**) &ad, (sizeof(int) * N));

    for(i=0; i<N; i++)       ah[i] = i;

    //copy data from host to device
    cudaMemcpy(ad, ah, sizeof(int) * N, cudaMemcpyHostToDevice);

    //launch CUDA Kernel
    find_square <<< 1, N >>> (ad, N);
```



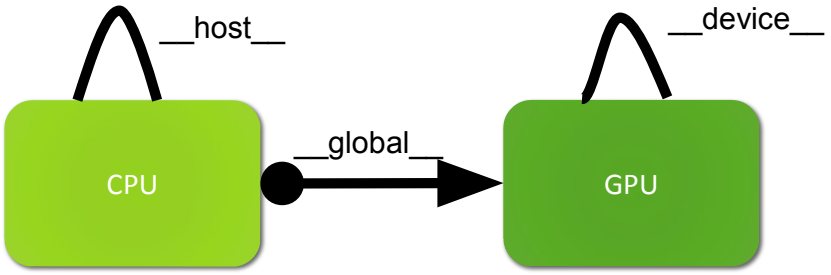
Cont..


```
//copy data from device to host
cudaMemcpy(ah, ad, sizeof(int) * N, cudaMemcpyDeviceToHost);

for(i=0; i<N; i++)
        printf("Square of %d = %d\n", i, ah[i]);
    return 1;
}
```



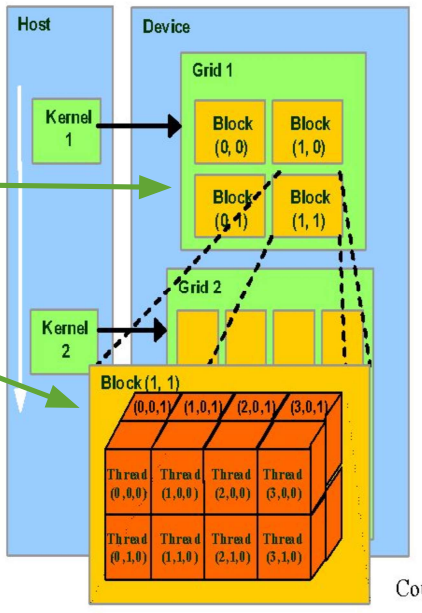
Kernel Code

```
__global__ void find_square(int *ad, int N)
{
    int index = threadIdx.x;
    if(index < N)
        ad[index] = ad[index] * ad[index];
}
```



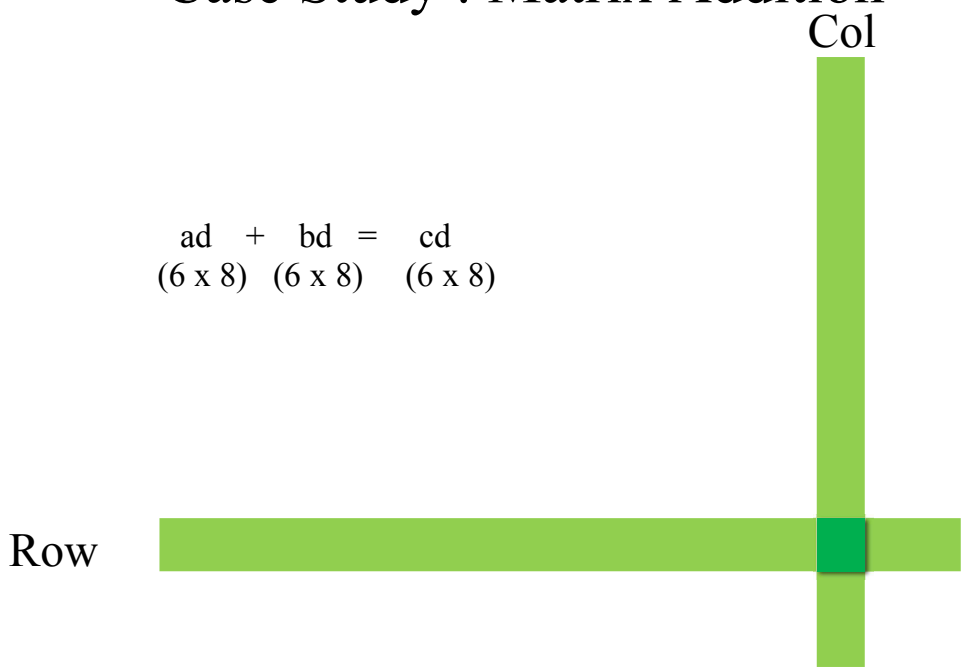
GPU Threading Model

- All threads in a block execute the same kernel program (SPMD)
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D, 2D or 3D
  - Thread ID: 1D, 2D or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
- Threads in the same block share data and synchronize while doing their share of the work
- Threads in different blocks cannot cooperate
  - Each block can execute in any order relative to other blocks



Courtesy: NDVIA

Case Study : Matrix Addition

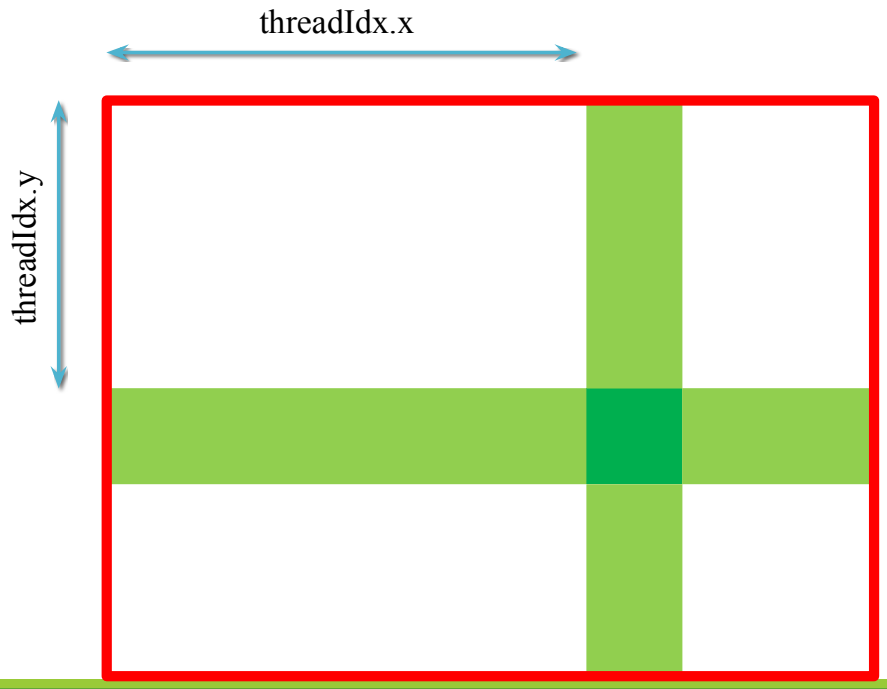


6 x 8 matrix elements = 48 threads

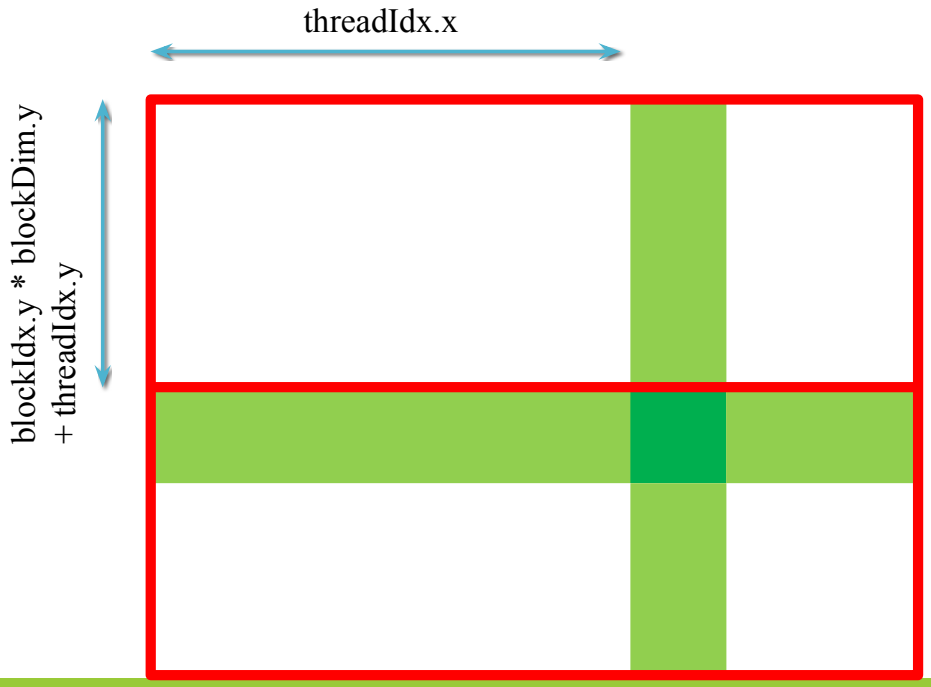

```
__global__ void matrix_add (float *ad, float *bd, float *cd, int  N)
{

      int Row = blockIdx.y * blockDim.y + threadIdx.y; //Row means y direction
      int Col   = blockIdx.x * blockDim.x + threadIdx.x; //Col means x direction

       int index = Row * N + Col; //N is the number of elements in a row

      cd[index] = ad[index] + bd[index];
}
```

```
.
.
dim3 grid_conf( ? , ? );
dim3 block_conf( ?, ?, ? );
matrix_add <<< grid_conf, block_conf>>> (ad, bd, cd, N);
.
.
```

Configuration 1- No of Blocks = 1
No of threads in each Block = (8,6)

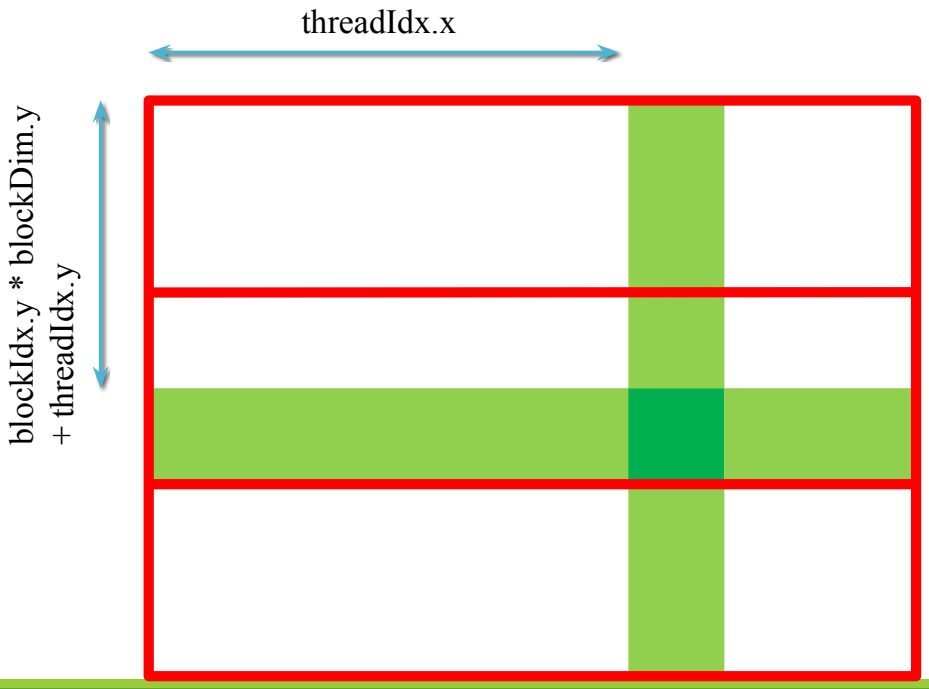


Configuration 2- No of Blocks = 2
No of threads in each Block = (8,3)



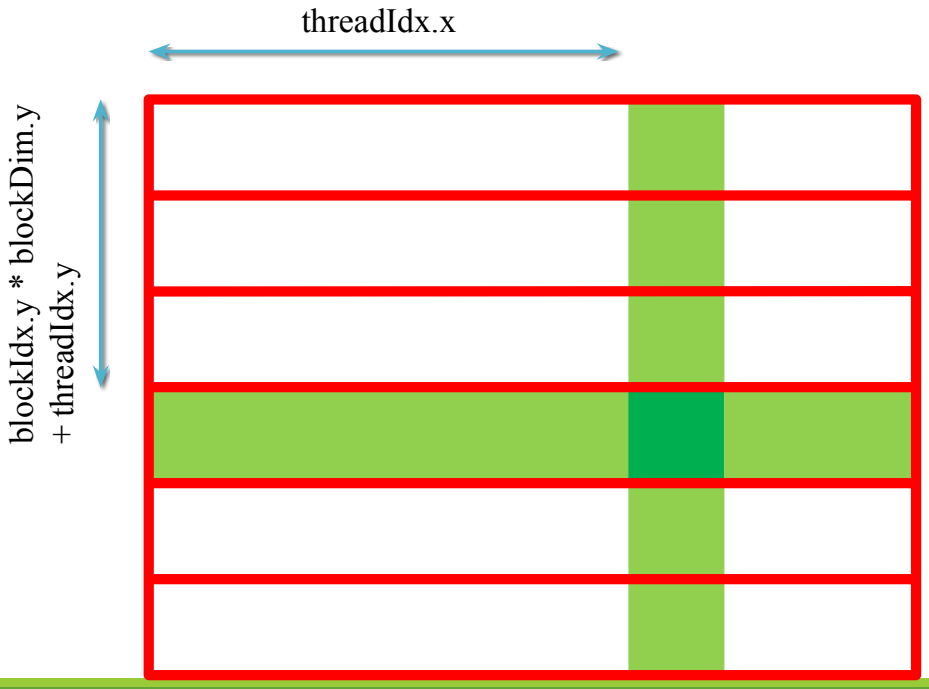
Configuration 3-

No of Blocks = 3
No of threads in each Block = (8,2)



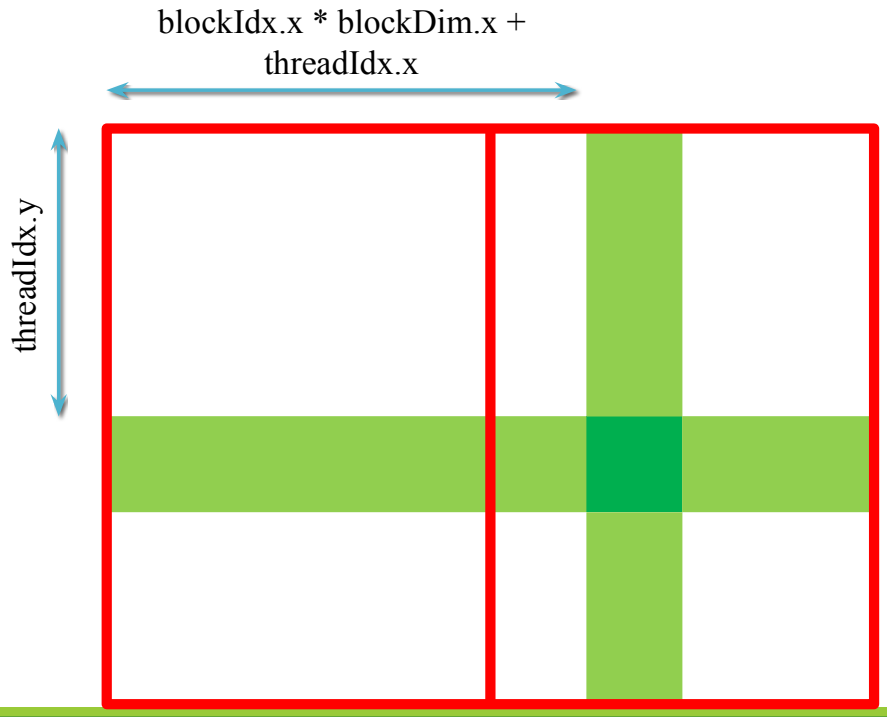
Configuration 4-

No of Blocks = 6
No of threads in each Block = (8,1)



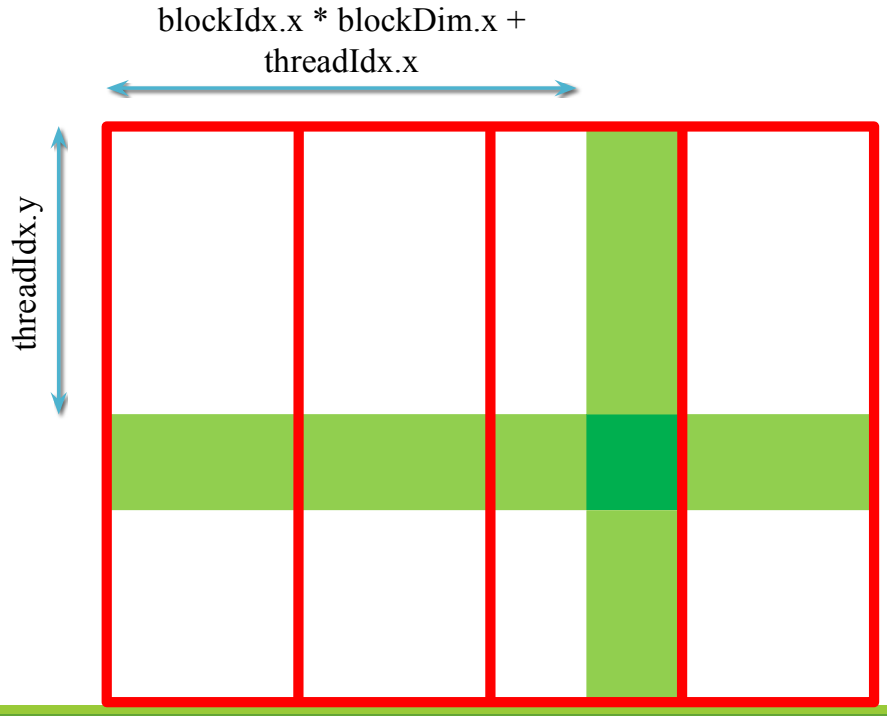
Configuration 5-

No of Blocks = 2
No of threads in each Block = (4,6)



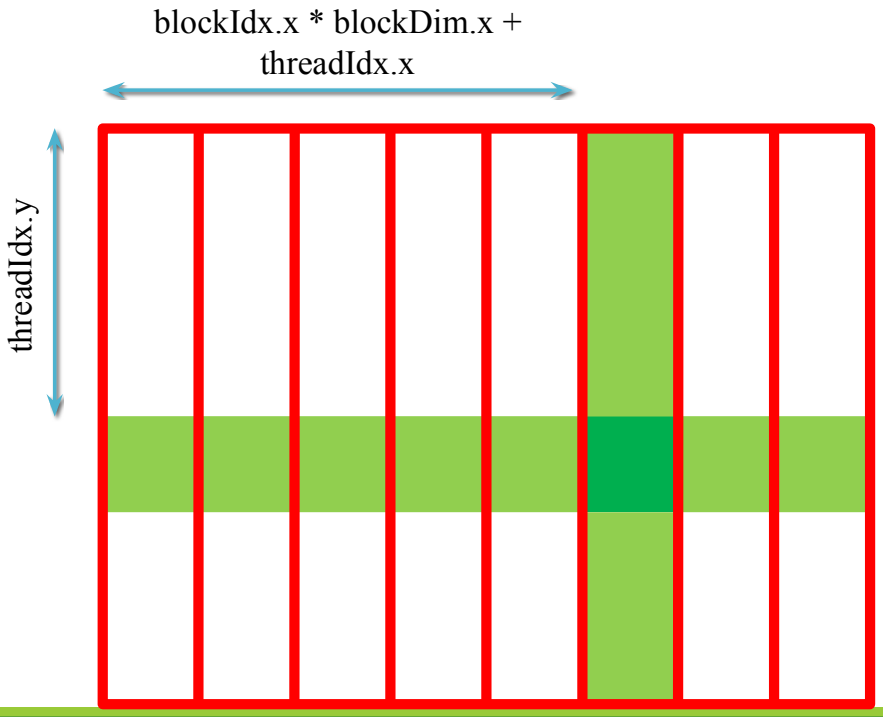
Configuration 6-

No of Blocks = 4
No of threads in each Block = (2,6)



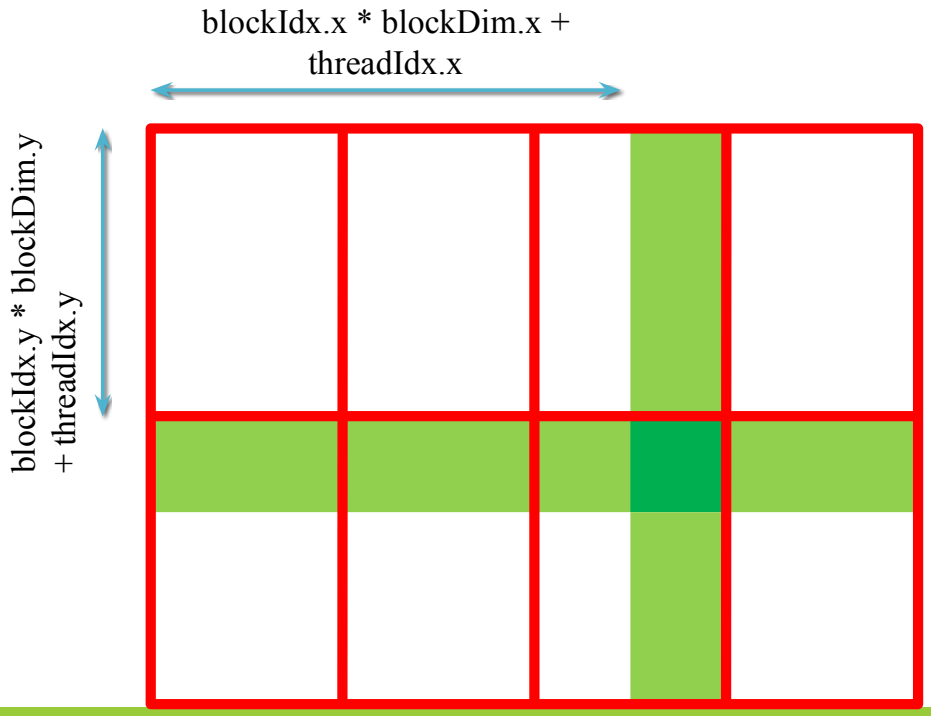
Configuration 7-

No of Blocks = 8
No of threads in each Block = (1,6)



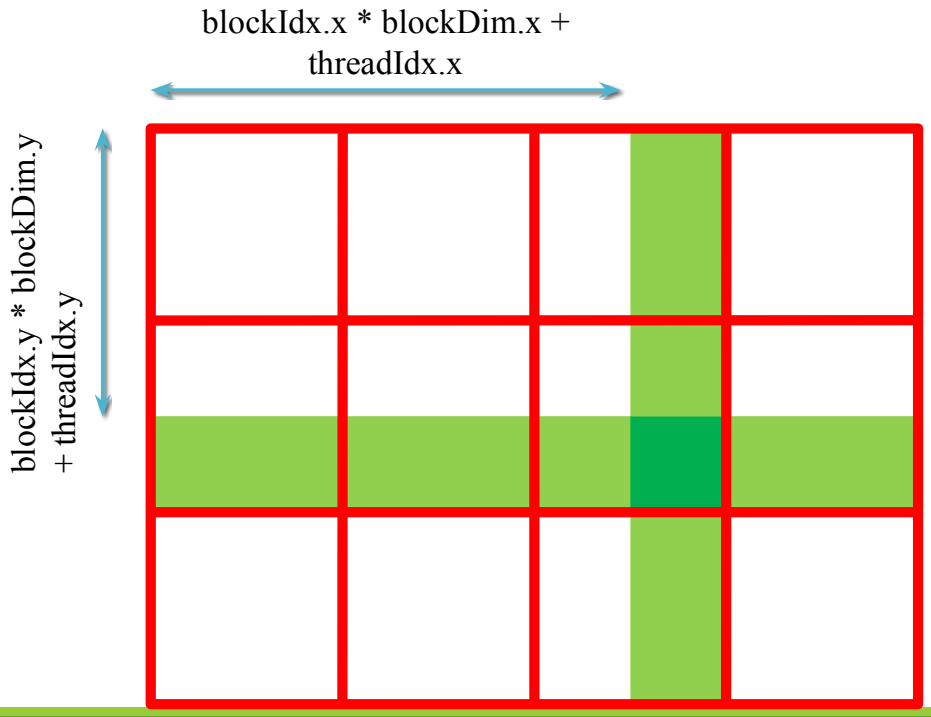
Configuration 8-

No of Blocks = 8
No of threads in each Block = (2,3)



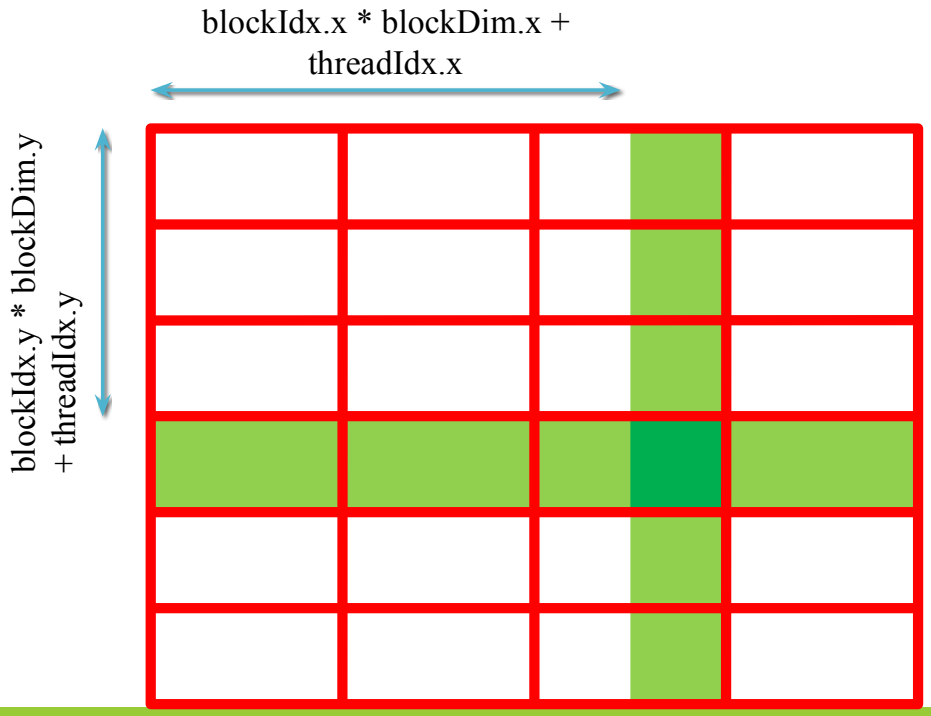
Configuration 9-

No of Blocks = 12
No of threads in each Block = (2,2)

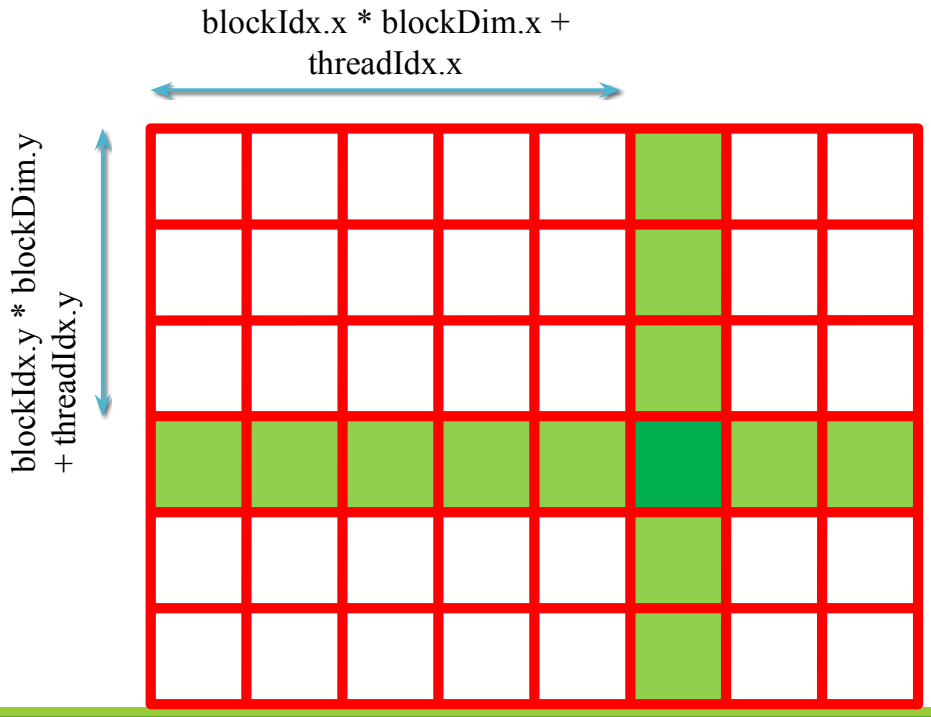


Configuration 10-

No of Blocks = 24
No of threads in each Block = (2,1)



Configuration 11- No of Blocks = 48
No of threads in each Block = (1,1)



Questions?

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Murali Parameswaran

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Session 7: 11 Feb. 2024

Midterm

Q1 9M… On how to parallelize a newer algorithm

Consider the following equation for gradient descent:

- $w = w - \eta * g(L, D, w)$
- where g is the gradient function, L the loss function, and D, the dataset and η denotes the learning rate and the corresponding pseudo-code for the mini-batch variant:

```
1.  for i = 1 to num_iter {
2.        shuffle ( data );
3.        for batch in get_batches (data, batch_size) {
4.              grad = eval_gradient ( loss_function , batch , w );
5.               w = w - learning_rate * grad;
6.         }
```

Identify the line numbers in this code that can be parallelized. Argue how the said lines can be parallelized using a

- Shared memory model
- Distributed memory model.

Q2 8M on cost of the three machine models

What cost is incurred during deployment? With the example of searching for a key value in a large set of values, argue the cost incurred based on a sequential machine model, a shared memory model and a distributed memory model.

What cost is incurred during training of a model? Argue the difference in cost incurred based on a sequential machine model, a shared memory model and a distributed memory model.

Q3 4M on GPGPU Thread model

- Show how the threads work in GPGPU using an application that needs to calculate square of N values. How does this thread model differ from a typical multi-threaded multi-core implementation.

- What is the thread model used in GPGPU. How does this benefit computations used in machine learning operations.

Q4 4M on map reduce

- Demonstrate the use of map and reduce paradigms to compute the following:

$$\mathbf{A} + \mathbf{B} = (a_1 + b_1, a_2 + b_2, \ldots, a_n + b_n)$$

$$\sqrt{\sum_{i=1}^{i=n} (\mathbf{x_i} - \mathbf{y_i})^2}$$

1. L' <- make a list with tuples <ai, bi> /// L' <- list of <xi, yi>
2. Res= map foo L'
3. Result <- reduce + Red
4. Answer <- square root of Result

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Session 7: 11 Feb. 2024

Parallel/Distributed ML Algorithms

- Exercises: kNN, Decision Trees

- Assignment

Nearest Neighbors method for Classification

- Approach:
  - Supervised
  - A (test) point **p** is assigned to a class ***C***, to which belong a majority of the neighbors of **p**
    - This requires determining the nearest neighbors
  - The number of neighbors, K,
    - to be used to identify the class of a given test point is chosen externally
    - and the choice is non-trivial:
      - Too Large K ==> reduced accuracy
      - Too Small K ==> increased noise-sensitivity

K Nearest Neighbors: Algorithm

Inputs: Dataset D, positive integer k // D is (partially) labelled

Approach:

1. for each x in D:
    1. Compute distances to all other points in D
    2. Choose the k neighboring points nearest to x
    3. Identify the class C to which a majority of these neighbors belong
    4. x.class = C

Break ties arbitrarily!

K Nearest Neighbors: Algorithm: Analysis

Inputs: Dataset D, positive integer k // |D|=N

Approach:

1. for each x in D:
    1. Compute distances to all other points in D
    2. Choose the k neighboring points nearest to x
    3. Identify the class C to which a majority of these neighbors belong
    4. x.class = C

- $O(N^2)$ distance computations
- Each computation take O(d) time
    - (assuming d-dimensional Euclidean space).

- O(N) majority computations
- Each computation takes O(k) time

What about step 2?

K Nearest Neighbors: Algorithm: Analysis

Inputs: Dataset D, positive integer k

Approach:

1. for each x in D:
    1. Compute distances to all other points in D
    2. Choose the k neighboring points nearest to x
    3. Identify the class C to which a majority of these neighbors belong
    4. x.class = C

Step 2:

- Naïve approach: Sort the neighbors by distance! // O(NlogN) additional time

A popular alternative: Construct a KD-tree (using the distance metric)

- Combined cost for distance computations and tree construction:
    - O(d*N*logN), where for d-dimensional data

Brief intro of KD trees: https://www.youtube.com/watch?v=Glp7THUpGow

Shared Memory processor

Parallel Algorithm - Data parallel

- How do you parallelize kNN?
    - for each x in D:
    - Step 1: Distance Computation
        - N^2 processors: 1 distance computation each;
        - p processors: N^2/p computation in each processor
        - ==========
        - Step 2: Getting the nearest neighbors

Speedup: p

    - Option 1: for each x in D:
        - Parallel-sort the neighbors by distance (and select the first k)

Speedup: p

    - Option 2: for each processor P_j j = 1 to p
    - for each x in D_j
        - sort the neighbors (all points) by distance // sorting of |D| values
        - ==========

    - Step 3: for each processor P_j j = 1 to p
    - for each x in D_j
        - Finding the majority among k neighbors

Speedup: p

Shared Memory processor

Online Algorithm – Request Parallelism

- Maintain a pool of threads
  - When a point p arrives, it is assigned to a (free) thread
- (thread == processor)

kNN: Distributed Algorithm

- How do you distribute kNN ?
  - Step 1: Distance Computation
  - Step 2: Getting the nearest neighbors
  - Step 3: Finding the majority
  - E.g. Step 1: point 1 (at node) – distance computation with all other points

Communication cost? Entire dataset is broadcast?
Distributed KD-Tree Construction?

N points into p nodes; (N/p) points each

- Data set D has been divided into p nodes (N/p points each; N = |D|)

- Let point x be in node i
- Send point x to all p nodes

  - Parallel: In each node dist(x,y) is computed for all y in D_j j = 1 to p
  - Send back k nearest neighbors of x in node j to node I
  - node i has p*k potential neighbors; extract the k nearest
- O(k) per point

Assignment

- Assignment is released
  - Assignment and Project will form a single end-to-end sequence of take-home team exercise involving:
    - Literature Survey,
    - Problem Formulation,
    - Design,
    - Implementation,
    - Testing and Demo
      - particularly for Performance
- Teams can include at most five students
  - Problems must focus on paralellization/distribution of ML algorithms
  - Target architectures can be multi-core, GPUs, or clusters or a combination thereof.
  - Any Programming language/environment or development platform may be used.

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Murali Parameswaran

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Session 8: 18 Feb 2024

Distributed ML - Models and Platforms

- Implementation Issues
- The Parameter Server Model
- Stochastic Gradient Descent

Decision Trees

- Approach:
  - Construct a tree where each node denotes a binary decision
    - Nodes in the tree correspond to features and the order of features is chosen based on the notion of **information gain** (IG)
    - Information gain is the entropy
      - entropy of the whole set
      - minus the entropy when a particular feature is chosen

Decision Tree Construction

- Algorithm ID3 (input dataset S)
  - If all examples have the same label
    - Return a leaf with that label
  - Else if there are no features left to test
    - Return a leaf with the most common label
  - Else choose the feature F that maximizes IG of dataset S as the next node
    - Add a branch from the node for each possible value f in F
    - For each branch:
      - Calculate S_f by removing F from the set of features
      - ID3(S_f)

Parallel/Distributed Tree Construction?

- When you branch assign each branch (corresponding to one value of a feature)
  - To a different task (task parallelism)
  - At each level : number of parallel tasks = number of possible values of a feature

ML problem and Error

- Given input dataset – a vector of size n,
  - Each training example $\mathbf{x}_i$ of **d** features (or dimensions) is associated
    - with a label $\mathbf{y}_i$ and
    - model parameters (likely corresponding to the features)
  - The problem is to predict **y** corresponding to an unseen example **x**
- Training error
  - Difference between the predicted y the actual label y' for an x

Model complexity

- Relation between model size (number of parameters) and data size (for training):
  - If there is too little data,
    - then a highly detailed model may overfit
  - If the model is too small,
    - then it may fail to capture relevant attributes
- Regularization addresses this issue

ML as regularized error minimization

- Training an ML model is minimizing the function F :
  - $F(w) = \Sigma_i L(x_i, y_i, w) + \Omega(w)$
    - where w denotes the set of parameters and
      - L is the loss function (i.e. prediction error) and
      - Ω is the regularizer that penalizes the model for complexity

Distributed ML

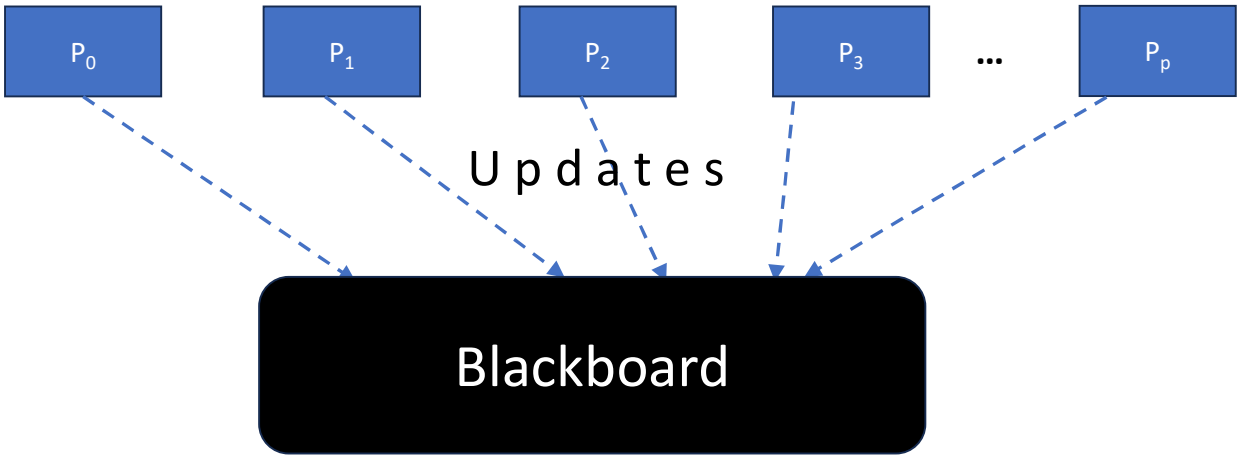
- Complexity:
  - Training Data size: from 1TB to 1PB
  - Model Size: 10^9 to 10^{12} parameters
- Examples:
  - Online Recommender System
    - millions of user profiles
  - Ad click predictor
    - each training example is a feature vector of high dimensions

Distributed ML – System Requirements

- In a distributed system,
  - the training data is partitioned among multiple nodes
    - and the nodes together learn the parameter vector w.
- The algorithm operates iteratively:
  - In each iteration,
    - every node independently uses its own training data to
      - Compute the changes to be made to **w** in order to move closer to an optimal value
    - Each node computes changes to **w** based only on its local data,
      - a central place is needed to aggregate these changes

Distributed Systems – Blackboard architecture

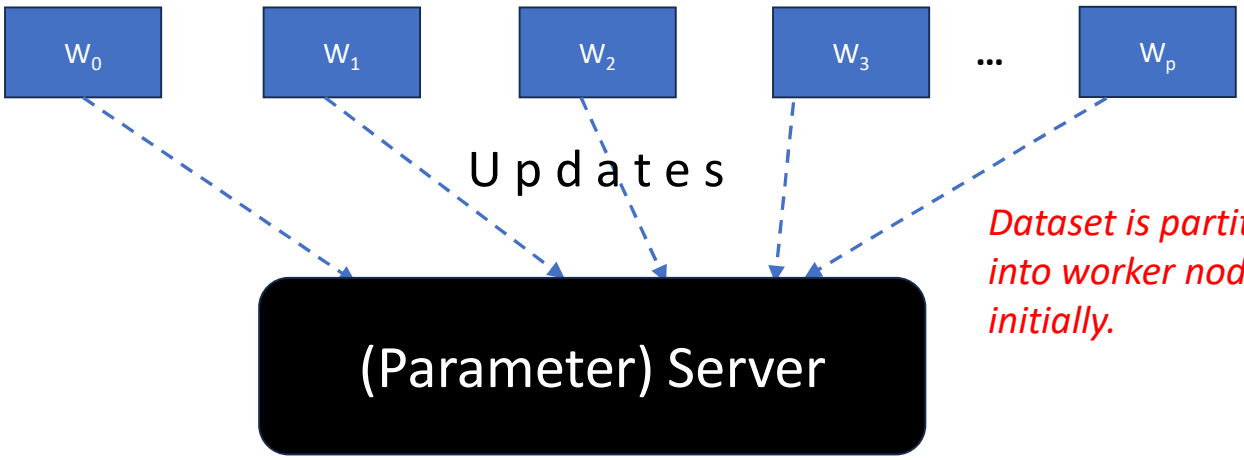
- Blackboard architecture is a pattern for distributed computation
  - where multiple nodes have to combine results
    - *computed locally, in parallel – see processes P_i below*



Regularized Error Minimization : A Distributed Architecture

In each iteration:

- Each worker node W_i pulls (current) parameters w from the server, computes F(w), and posts it back.
- Server updates and minimizes F(w)



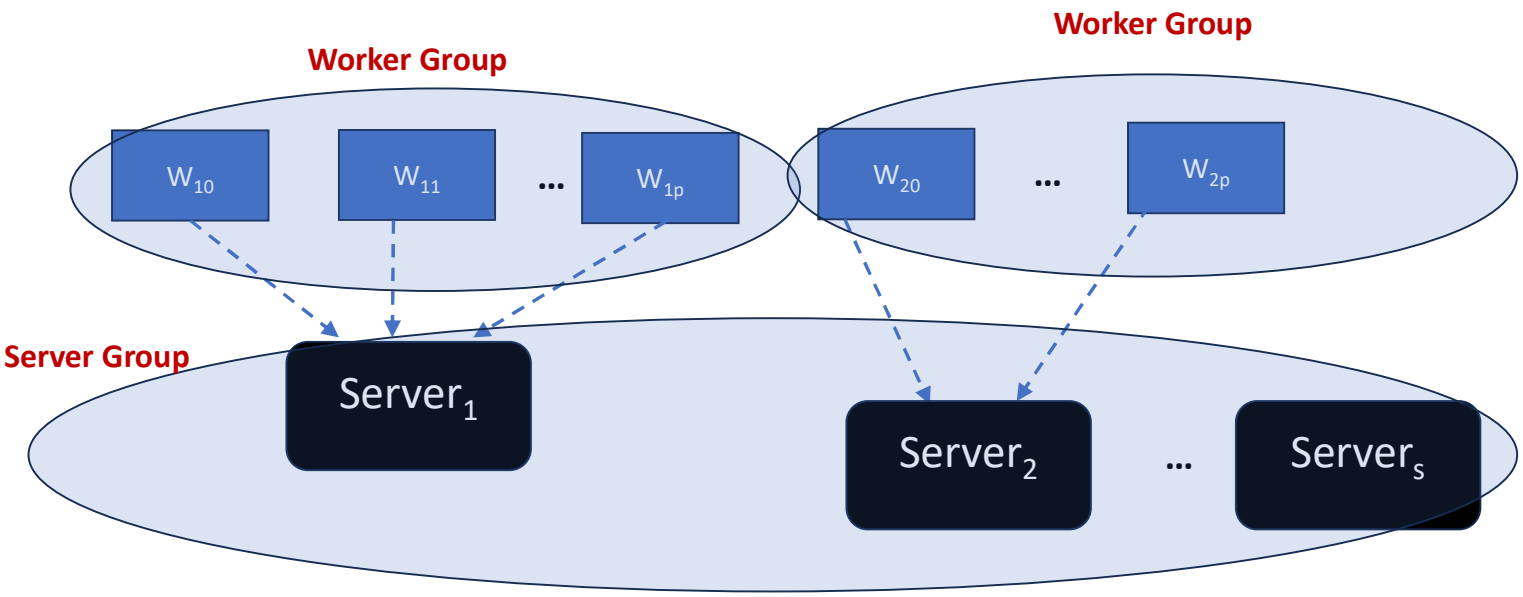
Dataset is partitioned into worker nodes initially.

Distributed Systems and Failures

- Individual Nodes may fail frequently in distributed systems:
  - This is particularly so in commodity clusters
- Rate of node failure increases with the size of the system (*i.e., more nodes and more processes*)
  - **Cloud data centers** are made out of commodity clusters!
- Distributed Systems have to function (i.e. be available) despite node failures
  - This is referred to as fault tolerance and is achieved via
    - redundancy and failure recovery

Scalable and Dependable (*reliable and available*) architecture for ML

Each Worker Group includes a task scheduler.
Server Group must be fault tolerant!



innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Murali Parameswaran

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Session 9:25 Feb. 2024

Gradient Descent Algorithm

Towards Optimizing Neural Nets

- System / Program Optimization Techniques
- Locality-aware Programs

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

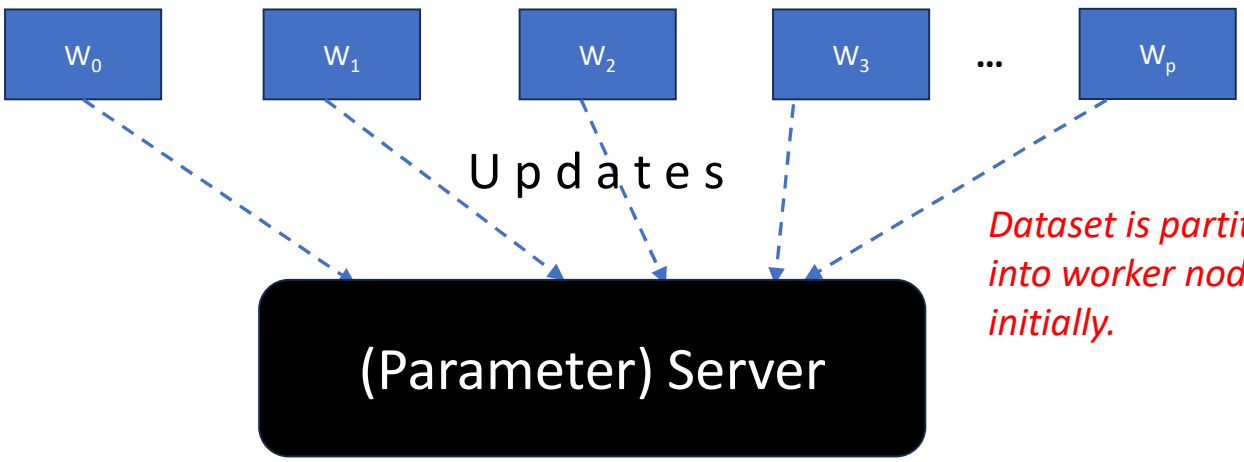
Session 9:25 Feb. 2024

Review:
Gradient Descent Algorithm

Regularized Error Minimization : A Distributed Architecture

In each iteration:

- Each worker node W_i pulls (current) parameters w from the server, computes F(w), and posts it back.
- Server updates and minimizes F(w)



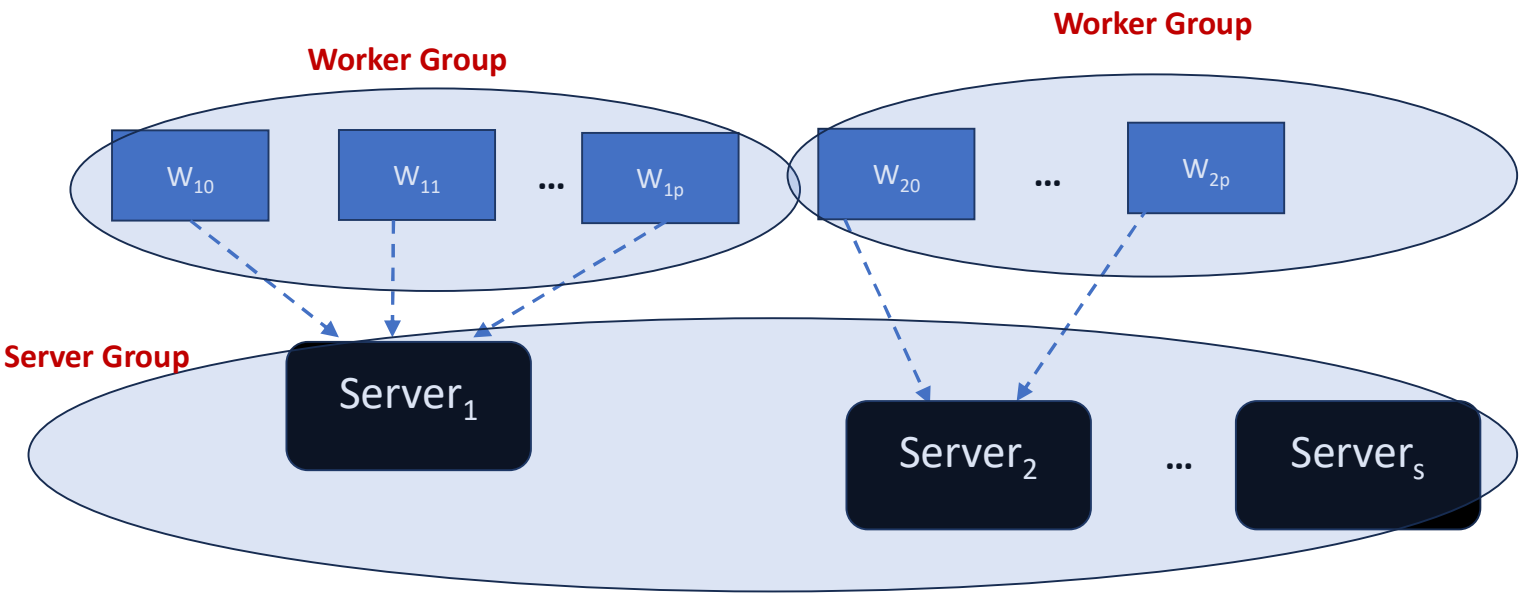
Dataset is partitioned into worker nodes initially.

Distributed Systems and Failures

- Individual Nodes may fail frequently in distributed systems:
  - This is particularly so in commodity clusters
- Rate of node failure increases with the size of the system (*i.e., more nodes and more processes*)
  - **Cloud data centers** are made out of commodity clusters!
- Distributed Systems have to function (i.e. be available) despite node failures
  - This is referred to as fault tolerance and is achieved via
    - redundancy and failure recovery

Scalable and Dependable (*reliable and available*) architecture for ML

Each Worker Group includes a task scheduler.
Server Group must be fault tolerant!



Scalable and Dependable (*reliable and available*) architecture for ML

- This architecture is referred to as the ***parameter server*** model:
  - Different servers may handle different problems
  - Multiple servers may work on the same problem to improve performance
    - This will require additional combination/minimization processes and servers.
  - Multiple servers may work on the same problem for redundancy.

Parameter Server Model

- This model was popular for a few years
  - Google built an internal platform named DistBelief based on this model
    - DistBelief was optimized primarily for large clusters of multi-core nodes
    - GPU clusters were enabled later
  - Later, Google's TensorFlow provided programming flexibilities not available in DistBelief:
    - Adding new layers
    - Adding new ML training workflows
    - Optimizing or tuning ML algorithms

TensorFlow

- GPU acceleration has become a common tool for ML algorithms.
  - Building and testing on GPU workstations before scaling it to a GPU cluster is a common scenario as well.
- TensorFlow provides a unified programming interface and a common runtime on all these hardware platforms
  - while also supporting heterogeneous accelerators.
- e.g. Google's TPUs are special purpose accelerators for ML
  - that enable increased performance-per-watt compared to other state-of-the-art hardware.
- TensorFlow supports a common device abstraction for heterogeneous accelerators.

Gradient Descent

- One approach to error minimization is known as Gradient Descent:
  - Use the slope (i.e., gradient) of the loss function to update the parameters.
- This is particularly useful in Neural Networks in the back-propagation phase

Gradient Descent

- The gradient descent approach to minimize the error requires the following update to the parameters
  - $w = w - \eta * g(L, D, w)$
    - where g is the gradient function, L the loss function, and D, the dataset.
    - η denotes the learning rate – controls the amount by which the parameters are updated.
- The updates are done iteratively.

Gradient Descent

for i = 1 to num_iter :

1. grad = eval_gradient (loss_function , D , w)
2. w = w - learning_rate * grad

- This is Batch GD!
  - i.e., update is done after all the points in the dataset are considered.
- Batch Gradient Descent is slow to converge, particularly if same data (or similar) data is repeated within the dataset.:

Stochastic Gradient Descent (SGD)

1. for i = 1 to num_iter :
    1.1 shuffle (data)
    1.2 for example in data :
        1.2.1 grad = eval_gradient (loss_function , example , w)
        1.2.2. w = w - learning_rate * grad

This may converge faster but is completely sequential i.e., not easy to parallelize!

Mini-batch SGD

for i = 1 to num_iter:
1. shuffle (data)
2. for batch in get_batches (data , batch_size): → Parallelize / distribute
    1. grad = eval_gradient (loss_function , batch , w)
    2. w = w - learning_rate * grad

- Batches in the inner loop can be executed independently (locally)!
  - If necessary, batches can be obtained and stored locally at the start.
- Update has to be done in the parameter server

Elementary System Design

Locality of References

- Consider a "typically random" program and its memory accesses:
  - *Locations* (or *variables* or *objects*) that are accessed are ***likely to be accessed*** repeatedly
    - e.g. instructions and data *in the body of a loop*
    - e.g. instructions and data in a function / procedure

    } Temporal locality
  - Locations nearby - other locations that are accessed - are likely to be accessed
    - e.g. elements of an array (or a matrix)
    - e.g. instructions in a sequence

    } Spatial locality
- In short, the locus of memory accesses (by a program) *is* small:
  - the locus refers to any small window of time during program execution.

Locality and System Design

- Locality of Reference is an observation about "typical programs".
  - Exercise:
    - Design counter-example programs (that violate this observation).
- (Computing) System Designers use this a heuristic in designing and optimizing systems:
  - Memory hierarchy and Data Movement techniques

Locality and Memory hierarchy

- Different levels of memory
  - Large (, cheap, and slow)

  to
  - Small (, costly, and fast):
    - Registers (inside the processor)
    - Cache (between the processor and RAM)
    - RAM addressable but volatile
    - Hard Disk (non-volatile, semi-sequential access)
    - Remote (over the network)

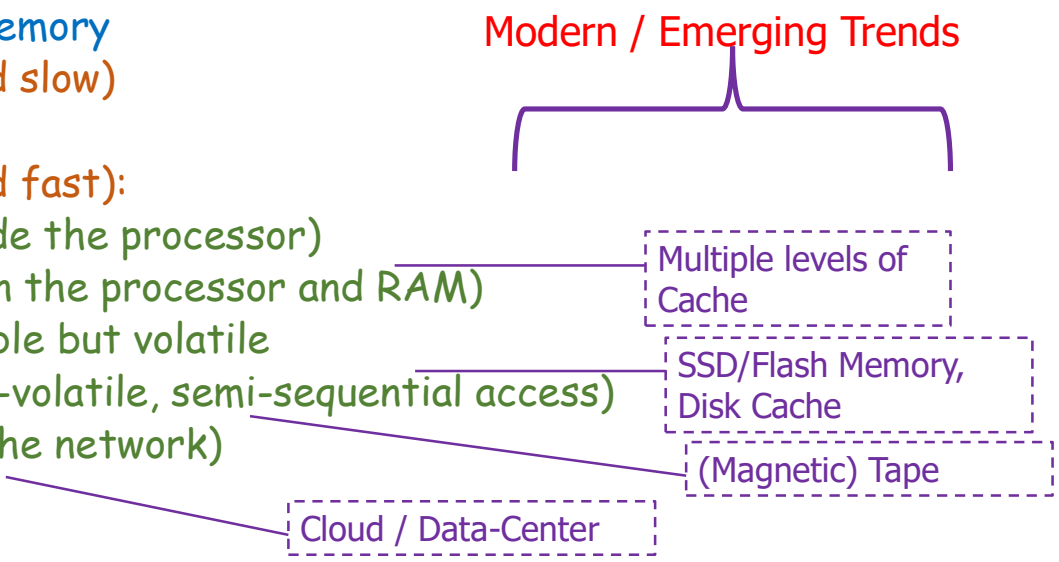
Cost Amortization

- Exercise:
  - Find out and understand the access times and ratios

Locality and Memory hierarchy

- Different levels of memory
  - Large (, cheap, and slow)

  to
  - Small (, costly, and fast):
    - Registers (inside the processor)
    - Cache (between the processor and RAM)
    - RAM addressable but volatile
    - Hard Disk (non-volatile, semi-sequential access)
    - Remote (over the network)



Exercise: Understand and extrapolate the trends: what issues are they addressing and how?

Locality and Data Movement (up the Hierarchy)

- (Computing) System Designers design and optimize Data Movement techniques
  - Caching (addresses temporal locality):
    - "Retain" data – *close to the processor* - that have been accessed recently.
  - Pre-fetching (addresses spatial locality):
    - "Move data" – *closer to the processor* - by anticipating their access
  - Blocked data access and buffering
    - Access data in bulk to amortize (setup) cost in sequential accesses:
      - e.g. seek time, disk rotation time in hard disk
      - e.g. spooling time in tapes
      - e.g. latency in network access

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Murali Parameswaran

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Session 10: 3 Mar. 2024

Towards Optimizing Neural Nets

- System / Program Optimization Techniques
- Locality-aware Programs

innovate achieve lead

BITS Pilani

WILP

Program Design and Optimization

Example: Matrix Multiplication

```
void matMult_IJK (float *a, float *b, float *c, int n)
{       // Classic Sequential algorithm
        // n x n row-major matrices a and b.
    for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    {
        float temp = 0;
        for (int k = 0; k < n; k++)
        temp += a[i*n+k]*b[k*n+j];
        c[i*n+j] = temp;
    }
}
```

Inner product of i^{th} Row of a with k^{th} column of B

a[i*n+0],a[i*n+1],a[i*n+2],..

b[0*n+j],b[1*n+j],a[2*n+j],..

c[i*n+j],b[i*n+j],a[i*n+j],..

Consider the sequence of memory accesses: for **a**, **b**, and **c**

Matrix-multiplication: Cache-aware algorithm

```
void matMult_IKJ (float *a, float *b, float *c, int n)
{   // Sequential cache-aware algorithm
    // n x n row-major matrices a and b
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j < n; j++)
       c[i*n+j] = 0;
      for (int k = 0; k < n; k++)
      for (int j = 0; j < n; j++)
        c[i*n+j] += a[i*n+k]*b[k*n+j];
    }
```

b[i*n+0],b[i*n+1],a[i*n+2],..

Each c[i,j] is cached and accessed n*n times

How does the sequence of memory accesses: (for **a**, **b**, and **c**) compare with those from the previous version?

Running time Comparison

Running Time of the Matrix Multiplication methods in seconds (size n*n)
[on an Intel Xeon Quad-Core using only one core]

Method	n=256	n=512	n=1024	n=2048	n=4096
ijk	0.11	0.93	10.41	~450	~4026
ikj	0.14	1.12	8.98	~73	~581
ijk / ikj	0.80	0.83	1.16	6.19	6.93

Ratio of the running times

Matrix-multiplication: Multi-core version

```
void MultiCore_MM (float *a, float *b, float *c, int n, int t)
{     // Multicore algorithm to multiply two
      // n x n row-major matrices a and b.
    // t is the number of threads
#pragma omp parallel shared (a, b, c, n, t)
{
int tid = omp_get_thread_num(); // thread ID
matMult_IKJ_sc(a, b, c, n, t, tid);
}
}
```

#pragma instructs the compiler to use OpenMP library

SPMD programming i.e. data parallel programming

```
void matMult_IKJ_SC(float *a, float *b, float *c, int n, int t, int tid)
{           // compute an (n/t) x n sub-matrix of c; t is #threads; thread ID (0 <= tid < t )
    int height = n/t;
    int offset = height*n*tid; // offset = 0, height*n, height*2*n, … height*(t-1)*n
    a += offset;
    c += offset;

    for (int i = 0; i < height; i++)
    {
    for (int j = 0; j < n; j++) c[i*n+j] = 0;
    for (int k = 0; k < n; k++)
    for (int j = 0; j < n; j++)
        c[i*n+j] += a[i*n+k]*b[k*n+j];
    }
}
```

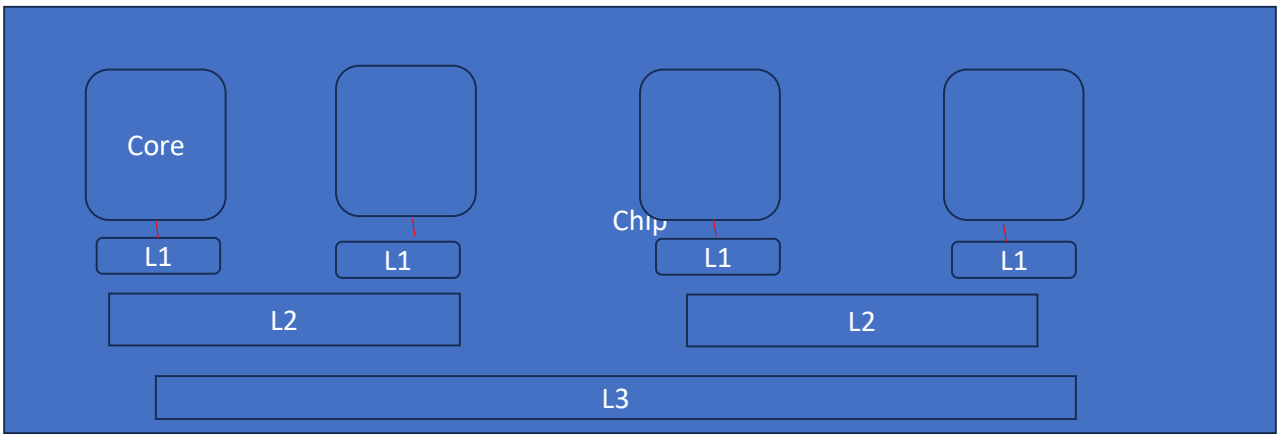
Running Time of the Matrix Multiplication method in seconds (size n*n)
[on an Intel Xeon Quad-Core using t threads]

t | n -->	256	512	1024	2048	4096
1	0.14	1.12	8.98	72.67	~581
2	0.07	0.56	4.50	36.39	~291
4	0.04	0.28	2.26	18.21	~146
8	0.04	0.28	2.28	18.42	~146

T1/T2	2	2	2	2	2
T1/T4	4	3.97	3.98	3.99	3.97
T1/T8	3.89	4.0	3.94	3.95	3.95

Speedups: Tj – Running time with j threads

Typical *Intel Multi-core Architecture*



L1 has I-cache and D-cache

Example: Matrix Multiplication

```
void matMult_IJK (float *a, float *b, float *c, int n)
{   // n x n row-major matrices a and b.
    for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    {
        float temp = 0;
        for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
        c[i*n+j] = temp;
    }

}
```

```
void matMult_IKJ (float *a, float *b, float *c, int n)
{   // Sequential cache-aware algorithm
    // n x n row-major matrices a and b
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j < n; j++) { c[i*n+j] = 0;  }
      for (int k = 0; k < n; k++)
      for (int j = 0; j < n; j++)
        c[i*n+j] += a[i*n+k]*b[k*n+j];
    }

}
```

matMult_IKJ is faster than matMult_IJK (for large **n**): caching effect!

- But matMult_IKJ accesses matrix **c** $O(N^3)$ times compared to **$O(N^2)$** accesses by matMult_IJK
  - Both access **a** and **b** **$O(N^2)$** times

Implication: matMult_IKJ is better suited for large scale multi-threading!

Multi-threaded Matrix Multiplication

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();  // to be defined
    int j = computeColID(); // to be defined
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

1. allocate space for c
2. spawn threads in a grid (i.e., in a matrix)
   1. one thread per element of **c**
   2. Join the threads

Multi-threaded Matrix Multiplication

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();  // to be defined
    int j = computeColID(); // to be defined
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

1. allocate space for c
2. spawn threads in a grid (i.e., in a matrix)
   1. one thread per element of **c**
   2. Join the threads

space in shared memory

Multi-threaded Matrix Multiplication

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();  // to be defined
    int j = computeColID(); // to be defined
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

1. allocate space for c
2. spawn threads in a grid (i.e., in a matrix)
   1. one thread per element of **c**
   2. join the threads

O(n*n) threads:
- each thread computes c[i,j] for one specific (i,j)
- i and j computed from threadID

Multi-threaded Matrix Multiplication

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();  // to be defined
    int j = computeColID(); // to be defined
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

1. allocate space for c
2. spawn threads in a grid (i.e., in a matrix)
   1. one thread per element
   2. join the threads

Typically, computation may have to wait for all threads to finish before proceeding further

- in this example, it is a data-parallel computation
- *a.k.a* Single Program Multiple Data (SPMD) computation
- *a.k.a* Single Instruction Multi-Threading (SIMT)

Multi-threading in CPUs

- Multiple threads run in parallel by sharing resources:
  - In a single processor system:
    - Thread execution is interleaved – at a time one thread gets the processor
    - Threads share virtual memory:
      - Every thread gets its own (call) stack
      - but global data is shared via global memory or heap
    - Threads share physical memory
      - RAM
      - Caches
- Implication:
  - typically not more than a few threads can run in parallel without impacting performance!

Multi-threading in CPUs: Multi-core processors

- Multiple threads run in parallel:
  - Each thread is scheduled to a core (at a time)
  - Cores are shared dynamically:
    - If there are **m** threads and **n** cores:
      - n-m cores are idle when n>m
      - More than one thread may be assigned to a core
        - i.e. threads interleave (like in the default scenario of one processor)
    - Threads share virtual memory:
    - Threads share physical memory:
      - RAM is shared
      - Cache is *private to a core* or *shared among the cores*

Example: Matrix Multiplication

```
void matMult_IJK (float *a, float *b, float *c, int n)
{   // n x n row-major matrices a and b.
    for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    {
        float temp = 0;
        for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
        c[i*n+j] = temp;
    }

}
```

```
void matMult_IKJ (float *a, float *b, float *c, int n)
{   // Sequential cache-aware algorithm
    // n x n row-major matrices a and b
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j < n; j++) { c[i*n+j] = 0;  }
      for (int k = 0; k < n; k++)
      for (int j = 0; j < n; j++)
        c[i*n+j] += a[i*n+k]*b[k*n+j];
    }

}
```

matMult_IKJ is faster than matMult_IJK (for large **n**): caching effect! (*see previous Session*)

- But matMult_IKJ accesses matrix **c** $O(N^3)$ times compared to **$O(N^2)$** accesses by matMult_IJK
- Both access **a** and **b** **$O(N^2)$** times

Implication: matMult_IKJ is better suited for large scale multi-threading!

Multi-threaded Matrix Multiplication

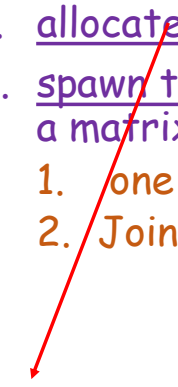
```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();  // to be defined
    int j = computeColID(); // to be defined
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

1. allocate space for c
2. spawn threads in a grid (i.e., in a matrix)
    1. one thread per element of **c**
    2. Join the threads

Multi-threaded Matrix Multiplication

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();  // to be defined
    int j = computeColID(); // to be defined
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

1. allocate space for c
2. spawn threads in a grid (i.e., in a matrix)
    1. one thread per element of **c**
    2. Join the threads



Multi-threaded Matrix Multiplication

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();  // to be defined
    int j = computeColID(); // to be defined
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

1. allocate space for c
2. spawn threads in a grid (i.e., in a matrix)
   1. one thread per element of **c**
   2. join the threads

O(n*n) threads:
- *each thread computes c[i,j] for one specific (i,j)*
- *i and j computed from threadID*

Multi-threaded Matrix Multiplication

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();  // to be defined
    int j = computeColID(); // to be defined
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

1. allocate space for c
2. spawn threads in a grid (i.e., in a matrix)
   1. one thread per element
   2. join the threads

Typically, computation may have to wait for all threads to finish before proceeding further
- in this example, it is a data-parallel computation
- *a.k.a* Single Program Multiple Data (SPMD) computation
- *a.k.a* Single Instruction Multi-Threading (SIMT)

Multi-threading in CPUs

- Multiple threads run in parallel by sharing resources:
  - In a single processor system:
    - Thread execution is interleaved – at a time one thread gets the processor
    - Threads share virtual memory:
      - Every thread gets its own (call) stack
      - but global data is shared via global memory or heap
    - Threads share physical memory
      - RAM
      - Caches
- Implication:
  - typically not more than a few threads can run in parallel without impacting performance!

Multi-threading in CPUs: Multi-core processors

- Multiple threads run in parallel:
  - Each thread is scheduled to a core (at a time)
  - Cores are shared dynamically:
    - If there are **m** threads and **n** cores:
      - n-m cores are idle when n>m
      - More than one thread may be assigned to a core
        - i.e. threads interleave (like in the default scenario of one processor)
    - Threads share virtual memory:
    - Threads share physical memory:
      - RAM is shared
      - Cache is *private to a core* or *shared among the cores*

innovate achieve lead

BITS Pilani
WILP

AIML CLZG516
ML System Optimization

Murali Parameswaran

innovate achieve lead

BITS Pilani
WILP

ML Systems Optimization

• Murali Paramweswaran

- *Matrix Multiplication in GPUs*

innovate achieve lead

BITS Pilani

WILP

ML Systems Optimization

Murali Parameswaran

Review of
Matrix Multiplication

Matrix Multiplication

```
void matMult_IJK (float *a, float *b, float *c, int n)
{   // n x n row-major matrices a and b.
    for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    {
        float temp = 0;
        for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
        c[i*n+j] = temp;
    }

}
```

```
void matMult_IKJ (float *a, float *b, float *c, int n)
{   // Sequential cache-aware algorithm
    // n x n row-major matrices a and b
    for (int i = 0; i < n; i++)
    {
      for (int j = 0; j < n; j++) { c[i*n+j] = 0;  }
      for (int k = 0; k < n; k++)
      for (int j = 0; j < n; j++)
        c[i*n+j] += a[i*n+k]*b[k*n+j];
    }

}
```

Multi-threading in CPUs

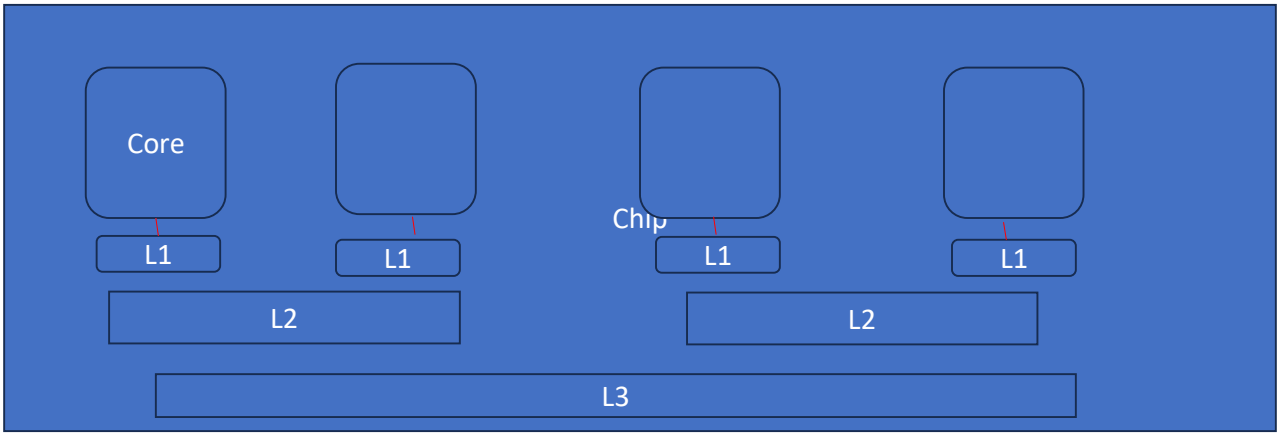
- Multiple threads run in parallel by sharing resources:
  - In a single processor system:
    - Thread execution is interleaved – at a time one thread gets the processor
    - Threads share virtual memory:
      - Every thread gets its own (call) stack
      - but global data is shared via global memory or heap
    - Threads share physical memory
      - RAM
      - Caches
- Implication:
  - typically not more than a few threads can run in parallel without impacting performance!

Multi-threaded Matrix Multiplication

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();  // to be defined
    int j = computeColID(); // to be defined
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

1. allocate space for c
2. spawn threads in a grid (i.e., in a matrix)
   1. one thread per element of **c**
   2. Join the threads

Typical *Intel Multi-core Architecture*



L1 has I-cache and D-cache

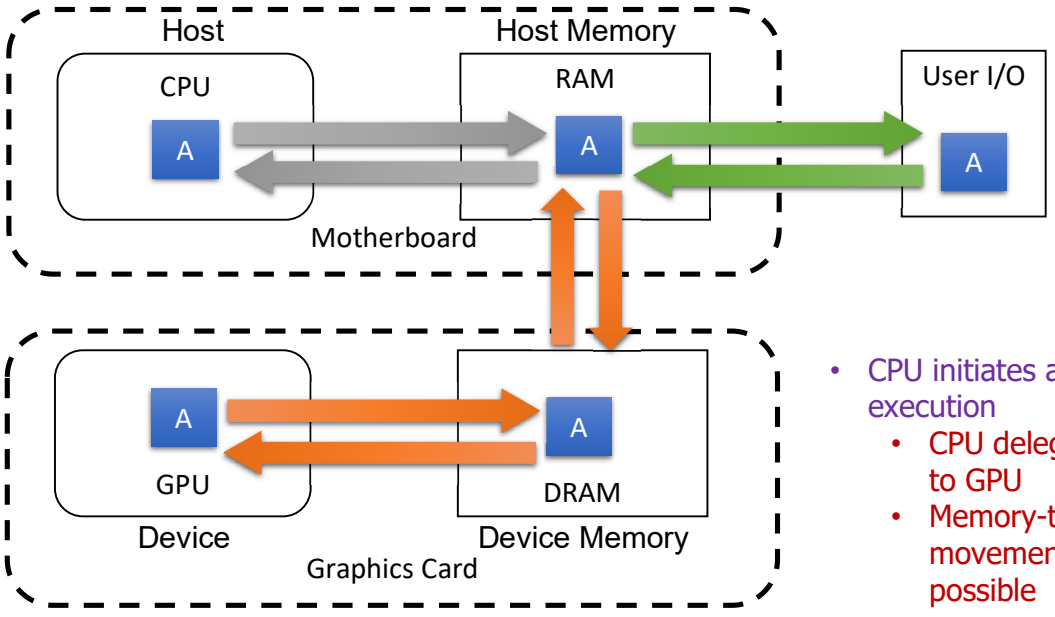
Multi-threading in CPUs: Multi-core processors

- Multiple threads run in parallel:
  - Each thread is scheduled to a core (at a time)
  - Cores are shared dynamically:
    - If there are **m** threads and **n** cores:
      - n-m cores are idle when n>m
      - More than one thread may be assigned to a core
        - i.e. threads interleave (like in the default scenario of one processor)
    - Threads share virtual memory:
    - Threads share physical memory:
      - RAM is shared
      - Cache is *private to a core* or *shared among the cores*

Review of GPGPUs

General Purpose Graphics Processing Units

GPU as a Co-processor



- CPU initiates and controls execution
  - CPU delegates work to GPU
  - Memory-to –memory movement of data possible

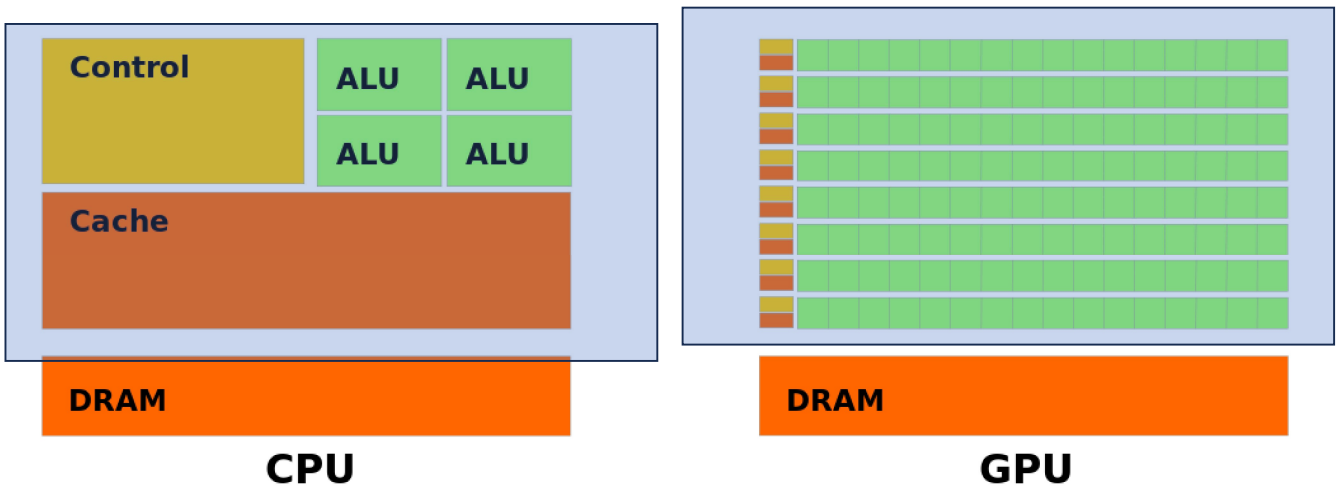
GPU Characteristics

- GPUs were designed for graphics processing where vertices and pixels are processed independently
  - Each processing step is usually arithmetic intensive
    - i.e., multiple operations are applied in between memory access
  - So, less control logic and more of arithmetic logic is required

- GPUs are designed for tasks that can tolerate high latency
  - as long as it can process a lot of tasks in one go
    - (i.e., high latency, high throughput)
- So, data caching is not a priority!

- More of the chip area can be given to ALUs
  - instead of Control Logic and Caches.
- Therefore, a lot of GPU threads (10s of thousands) can (should) execute at a time.

CPU Vs. GPU: Transistor Allocation Ratio



cf: **NVIDIA CUDA Programming Guide**

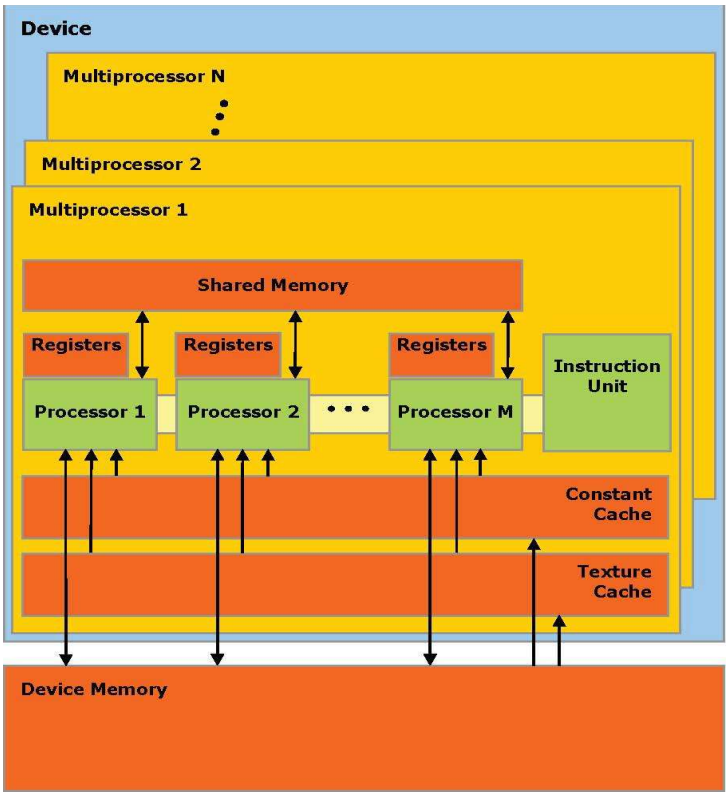
GPU Programming

- Kernel Programming Languages
  - Explicit and fine level control over threads and memory operations
  - Example:
    - CUDA (Compute Unified Driver Architecture) for Nvidia GPUs
      HIP for AMD and Nvidia GPUs
    - OpenCL for AMD and Nvidia GPUs
- Directive-based Programming Languages
  - No control over threads, compiler automatically generates code for parallelization
  - Example:
    - OpenMP for AMD and Nvidia GPUs
    - OpenACC for AMD and Nvidia GPUs

https://github.com/ROCm/HIP
https://www.openacc.org/

GPU Hardware

- Each GPU device has N streaming multiprocessors (SM)
  - each with M scalar processors (SP)
- e.g. NVIDIA Tesla C1060 (N=30, M=8)
  - 4GB off-chip device memory (global)
  - Each SM has 16KB of shared memory
  - Each SM has 16K registers (32-bit each)
- e.g. NVIDIA Tesla C2050 (N=14, M=32)
  - Each SM has 32K registers
  - Each SM has 64KB on-chip memory
    - divisible as 48KB + 16KB for shared memory and L1 cache (or the other way round)



GPU Multi-threading

- Single Instruction Multi-threading model
- Threads are light-weight:
  - Low context-switching overhead
- Thousands of threads can execute in parallel
  - Threads are grouped into blocks
    - 1D, 2D, or 3D (thread IDs)
      - Recall the matrix example.
  - Blocks are arranged in a grid
    - 1D, 2D, or 3D

Thread Execution

- When a kernel is launched:
  - blocks of the grid are enumerated and distributed to streaming multiprocessors (SM) with available capacity
- Each block is bound to one of the SM
  - Each block is divided into a group of 32 threads, known as a *Warp.*
  - Warp is the scheduling unit of .
- SM hardware implements zero-overhead Warp scheduling
  - Warps, whose next instruction has its operands ready for consumption,
    - are eligible for execution
  - Eligible Warps are selected for execution
    - on a prioritized scheduling policy
  - All threads in a Warp
    - execute the same instruction when selected

Thread Execution Support

- Instructions are pipelined
  - to leverage instruction-level parallelism
- Execution context for each warp
  - is maintained on-chip during the entire lifetime of the warp.
- Register file is partitioned among the warps.

(Back to) Matrix Multiplication

in GPGPUs

Multi-threaded Matrix Multiplication

GPU Kernel

```
void MM_kernel (float *a, float *b, float *c, int n)
{      // thread code to compute one element of c
       // element (i,j) of the product to be computed
    int i = computeRowID();
   int j = computeColID();
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

CPU code

1. allocate space for c
2. invoke threads in a grid (i.e., in a matrix)
   1. one thread per element of **c**

O(n*n) threads:
- each thread computes c[i,j] for one specific (i,j)
- i and j computed from blockID, threadID

Alternatives?

Mapping and Grouping

- Suppose we tile an n * n matrix using tiles of dimensions p * q:
  - Assume p|n and q|n for convenience
- Tiles are indexed by
  - (u,v), 0 <= u < n/q, 0 <= v < n/p
- Example: Tiling a 16 * 16 matrix with 2*4-tiles

u →
v ↓

(0,0)	(1,0)	(2,0)	(3,0)
(0,1)	(1,1)	(2,1)	(3,1)
(0,2)	(1,2)	(2,2)	(3,2)
(0,3)	(1,3)	(2,3)	(3,3)
(0,4)	(1,4)	(2,4)	(3,4)
(0,5)	(1,5)	(2,5)	(3,5)
(0,6)	(1,6)	(2,6)	(3,6)
(0,7)	(1,7)	(2,7)	(3,7)

- *A thread may determine the co-ordinates of the block that it is part of using the variables* blockIdx.x and blockIdx.y
  i.e., (u,v) = (blockIdx.x, blockIdx.y)
- *The dimensions of a thread block may be obtained by using the variables* blockDim.x and blockDim.y
  i.e, p = blockDim.y and q = blockDim.x
- *The index of a thread within a block is* (threadIdx.x, threadIdx.y)
  where 0 <= threadIdx.x < q and 0 <= threadIdx.y < p

Tile/Block Dimensions

- How do you decide p and q?
  - Hardware limits the number of threads in a block
    - E.g. 512 in (some) Nvidia GPUs
  - Thus p*q <= 512

- Threads are scheduled in units of warps and a warp is of size 32.
  - Thus 32 | p*q
- This limits the options to
  - 8*8 or 16*16 if tiles have to be squares
  - 1*64, 1*128, 2*128, 4*128, 4*32, 8*32, 8*64, 16*32 … otherwise

Multi-threaded Matrix Multiplication: Tiled version

GPU Kernel

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be
computed
 int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
 float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

CPU code

```
allocate space for c
// ..
// define grid and block dimensions
dim3 grid (n/q, n/p);
dim3 block (q, p);
// invoke kernel
MM_kernel <<<grid, block>>> (d_A, d_B, d_C, n);
```

Multi-threaded Matrix Multiplication

GPU Kernel

```
void MM_kernel (float *a, float *b, float *c, int n)
{       // thread code to compute one element of c
        // element (i,j) of the product to be computed
    int i = computeRowID();
   int j = computeColID();
    float temp = 0;
    for (int k = 0; k < n; k++)
            temp += a[i*n+k]*b[k*n+j];
    c[i*n+j] = temp;
}
```

CPU code

1. allocate space for c
2. invoke threads in a grid (i.e., in a matrix)

   each thread computes a **1*4** sub-matrix of **c**

```
MatMult_kernel_1x4(float *a, float *b, float *c, int n)
{// thread to compute a 1 x 4 sub-matrix of c
// determine index of 1 x 4 c sub-matrix to compute
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
int nDiv4 = n/4;
int aNext = i*nDiv4;
int bNext = j;
float4 temp4;
temp4.x = temp4.y = temp4.z = temp4.w = 0;
```

```
for (int k = 0; k < nDiv4; k++)
{
    float4 aIn = a4[aNext++];  float4 bIn = b4[bNext];

    temp4.x += aIn.x*bIn.x; temp4.y += aIn.x*bIn.y;
    temp4.z += aIn.x*bIn.z; temp4.w += aIn.x*bIn.w;
    bNext += nDiv4;  bIn = b4[bNext];

    temp4.x += aIn.y*bIn.x; temp4.y += aIn.y*bIn.y;
    temp4.z += aIn.y*bIn.z; temp4.w += aIn.y*bIn.w;
    bNext += nDiv4;   bIn = b4[bNext];

    temp4.x += aIn.z*bIn.x; temp4.y += aIn.z*bIn.y;
    temp4.z += aIn.z*bIn.z; temp4.w += aIn.z*bIn.w;
    bNext += nDiv4;  bIn = b4[bNext];

    temp4.x += aIn.w*bIn.x; temp4.y += aIn.w*bIn.y;
    temp4.z += aIn.w*bIn.z; temp4.w += aIn.w*bIn.w;
    bNext += nDiv4;
  }
   c4[i*nDiv4+j] = temp4;
}
```

Memory Access Cost

- The costliest access is from device memory:
  - It is off-chip
    - and hence slower than on-chip memories
  - and data movement takes additional time
    - Due to bandwidth limitations of the bus and the pins
- Usually, a single access fetches multiple words
  - Typically 128 bytes (or four 32-byte words).
- So, unless the spatial locality of such accesses is very high,
  - bandwidth utilization will be low (i.e. fetch and throw).
- Alternative?
  - Use shared memory as much as possible.