



DSECL ZG517 - Systems for Data Analytics

Session #1 – Course Introduction



BITS Pilani
Pilani Campus

Murali P

muralip@wilp.bits-pilani.ac.in

[Saturday – 04:30 PM]

Agenda

- About the course and operation
- Motivation
- Systems Attributes for Data Analytics - Single System
 - Kinds of storage
 - Levels of storage
 - Processing

About

Learn about fundamentals of data engineering; Basics of systems and techniques for data processing - comprising of relevant database, cloud computing and distributed computing concepts.

Objectives of the course

- Introduce students to a systems perspective of data analytics: to leverage systems effectively, understand, measure, and improve performance while performing data analytics tasks
- Enable students to develop a working knowledge of how to use parallel and distributed systems for data analytics
- Enable students to apply best practices in storing and retrieving data for analytics
- Enable students to leverage commodity infrastructure (such as scale-out clusters, distributed data- stores, and the cloud) for data analytics.

Administrivia

What to expect in this course?

- Analysing scenarios based on Hardware concepts
- Designing solutions for a particular scenario
- Analysing the optimality of the hardware solution proposed
- Analysing the cost effectiveness of the solution

What We'll Cover in this Course

- **Introduction to Data Engineering**
 - Systems Attributes for Data Analytics - Single
 - System Systems Attributes for Data Analytics - Parallel and Distributed Systems
- **Systems Architecture for Data Analytics**
 - Introduction to Systems Architecture
 - Performance Attributes of Systems
- **Data Storage and Organization for Analytics**
- **Distributed Data Processing for Analytics**
 - (Re-)Designing Algorithms for Distributed Systems
 - Distributed Data Analytics

Books

Text books and Reference book(s)

T1	Kai Hwang, Geoffrey Fox, and Dongarra. Distributed Computing and Cloud Computing . Morgan Kauffman
T2	

Expect to see more references in CANVAS

Books[2]

Reference book(s)

R1	Nikolas Roman Herbst, Samuel Kounev, Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. 10th International Conference on Autonomic Computing (ICAC '13). USENIX Association.
R2	Mohammed Alhamad, Tharam Dillon, Elizabeth Chang. Conceptual SLA Framework for Cloud Computing. 4th IEEE International Conference on Digital Ecosystems and Technologies. April 2010, Dubai, UAE.
R3	Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.
R4	Apache CouchDB. Technical Overview. http://docs.couchdb.org/en/stable/intro/overview.html
R5	Apache CouchDB. Eventual Consistency. http://docs.couchdb.org/en/stable/intro/consistency.html
R6	Seth Gilbert and Nancy A. Lynch. Perspectives on the CAP Theorem. IEEE Computer. vol. 45. Issue 2. Feb. 2012
R7	Werner vogels. Eventually Consistent. january 2009. vol. 52. no. 1 Communications of the acm.
R8	Eric Brewer. CAP Twelve Years Later: How the “Rules” Have Changed. IEEE Computer. vol. 45. Issue 2. Feb. 2012
R9	M. Burrows, The Chubby Lock Service for Loosely-Coupled Distributed Systems, in: OSDI’06: Seventh Symposium on Operating System Design and Implementation, USENIX, Seattle, WA, 2006, pp. 335–350.
R10	MATEI ZAHARIA et. al. Apache Spark: A Unified Engine for Big Data Processing .COMMUNICATIONS OF THE ACM NOVEMBER 2016 VOL. 59 NO. 11.
R11	YASER MANSOURI, ADEL NADJARAN TOOSI, and RAJKUMAR BUYYA. Data Storage Management in Cloud Environments: Taxonomy, Survey, and Future Directions . ACM Computing Surveys, Vol. 50, No. 6, Article 91. December 2017

Evaluation Plan

Name	Type	Weight
Quiz-I	Online	5%
Assignment-I	Take Home	25% both
Assignment-II	Take Home	A1+A2
Mid-Semester Test	Closed Book	30%
Comprehensive Exam	Open Book	40%

Lab Plan

Lab No.	Lab Objective
1	Programming exercises on map-reduce
2	Setting up a simple 3-tier application on the Cloud
3	Synchronization exercise on CouchDB
4	Pen-and-paper exercise on Locality, Memory Contention, and Communication Requirement
5	Pen-and-paper exercise on calculations of speedup, MTTF, and MTTR.

- **Labs not graded**
- **Lab recordings will be available**
- **Webinars will be conducted for lab sessions**

What's already done

- 2 year, 4 semester programme

Year	First Semester		U	Second Semester		U
I	DSE* ZC415	Data Mining	3	DSE* ZC413	Introduction to Statistical Methods	3
	DSE* ZC416	Mathematical Foundations for Data Science	4	DSE* ZG523	Introduction to Data Science	3
	DSE* ZG519	Data Structures and Algorithms Design	5		Elective -I	
	DSE* ZG516	Computer Organization & Software Systems	5		Elective-II	
	Total		17	Total		15 (min)
II	Elective-III			DSE*ZG628T	Dissertation	16
	Elective-IV					16
	Elective-V					
	Elective-VI		16 (min)			16

Administrivia

What to NOT expect in this course?

- Preliminary knowledge on Hardware concepts and fundamentals
- Preliminary knowledge on Data Engineering and Data Science

We expect you to have done well
Computer Organization and Software Systems
You should know processor, memory, storage,
performance metrics and measurement.



BITS Pilani
Pilani Campus

Systems for Data Analytics

Motivation



Systems Attributes for Data Analytics - Single System

Courtesy: Prof Sundar B slides

Storage for Data

- **Structured Data**
 - E.g. Relational Data
- **Semi-structured Data**
 - E.g. HTML pages, XML data, JSON, CSV files, Email, NoSQL DB, etc.
- **Un-structured Data**
 - E.g. X-ray images, audio/video/photo files, word processing docs, books, journals, health records, metadata, etc.

Anecdotal Evidence

- *Most of the data today is semi-structured / unstructured.*

Kinds of Data and Forms of Storage

- Historically,
 - **Relational Databases** were used for storing *structured data*
 - **File Systems** were used for *unstructured data*

Question

What are the typical characteristics of (relational) databases vs. file systems?

Storage for Data- Attributes

- Granularity
- Way of accessing
 - Random Access
 - Sequential
- Structure of DB and facilities as compared to file systems
 - Querying

There is a link between
form of data and form of storage

Kinds of Data and Forms of Storage

[2]



- Later,
 - XML databases were introduced to store semi-structured data
 - Object storages were introduced to store unstructured data (with ***object type*** information)

Exercise

Find examples of these two types of storage and their typical characteristics!

Kinds of Data and Forms of Storage



[3]

- Today,

Kind of Data	Form of Storage	Example (Products)
Structured (Relational)	Relational Databases	Oracle, MSSQL, and MySQL; SimpleDB (Amazon)
Semi- structured / Unstructured	File Systems or Object Storages or NOSQL databases (including XML databases)	MongoDB; S3, Elastic FS (Amazon)

Discussion/Assignment:

How we access data in relational vs semi-structured vs unstructured data?

Compare with real examples

Data Location: Memory vs. Storage

- Computational Data is stored in
 - Primary Memory (a.k.a. Memory)
- vs.
- Persistent Data is stored in
 - i.e. Secondary Memory (a.k.a Storage)

Data Location: Memory vs. Storage

- Computational Data is stored in
 - Primary Memory (a.k.a. Memory)

Use and Throw

vs.

- Persistent Data is stored in
 - i.e. Secondary Memory (a.k.a Storage)

Multiple runs

Questions / Exercises

1. What does “persistent” refer to? Is it same as non-volatile?
2. Identify examples of these two kinds of data
3. Identify technologies suitable for the two kinds of data

Data Location: Memory vs. Storage vs. Network

- Data accessed from a *local store*
 - i.e. storage attached to a computer
vs.
- Data accessed from a *remote store* / remote processor
 - i.e. storage hosted on the network (or *storage attached to a computer hosted on the network*)

Question

- What is the difference in the form of access?

There is a link between
form of data and form of storage

Cost of Access: *Memory vs. Storage* *vs. Network*



- Exercise:
 - What are the typical access times?
 1. RAM
 2. Hard Disk
 3. Ethernet LAN
 - Access Time Parameters: ***Latency*** and ***Bandwidth***
 - When and how do these parameters matter?
 - Identify mechanisms used to alleviate the access time delays in each case.

Memory Bandwidth Requirement



[1]

- How does all this impact processing capability?
 - Design intellect, money => processor is the key
- Let's take a typical processor
 - Consider a 2.5 GHz processor with 4 cores:
 - each with a CPI of 1.25 and
 - a RISC ISA where
 - 1 out of 4 instructions require a memory access and
 - word size is 4 bytes.
 - pipelining
 - Calculate the memory bandwidth required!

Memory Bandwidth Requirement

[2]



- Processor Clock: 2.5 GHz
- One cycle: 0.4 ns
- In one core:
- CPI = 1.25 (conservative estimate)
- Typical instruction executes hence in $1.25 * 0.4 = 0.5\text{ns}$
- RISC ISA
 - 1 out of 4 instructions requires memory data access
- 4 cores
 - 4 instruction access + 1 data access = 5 accesses.

Memory Bandwidth Requirement

[3]



- Total bandwidth to constantly feed processor:
 - 40GBps

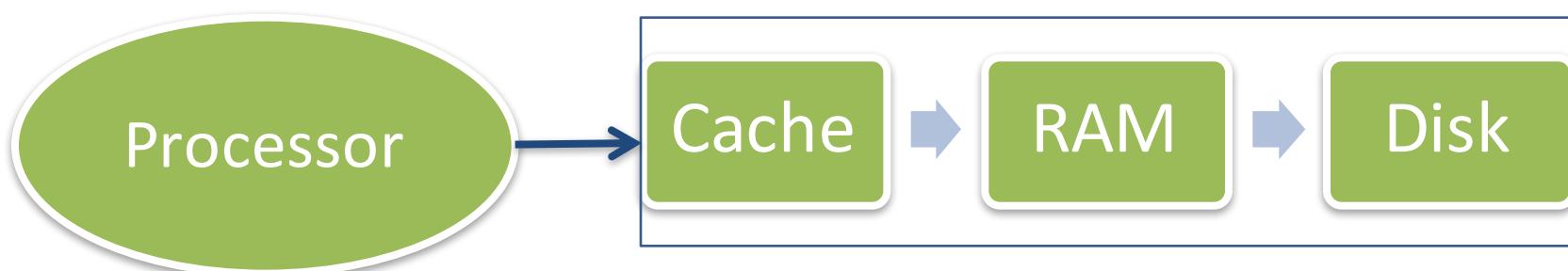
How to give a typical 4-core processor
about 40GB of data every second?

Memory Hierarchy – Motivation

- A **Memory Hierarchy** amortizes cost in computer architecture:
 - fast (*and therefore costly*) but small-sized memory to
 - large-sized but slow (*and therefore cheap*) memory

Memory Hierarchy

- Original:



- Modern:



Discussion/Assignment:

How do we
Reconcile Memory Bandwidth Requirement
with the Memory Hierarchy?



Thank you !



BITS Pilani
Pilani Campus



DSECL ZG517 - Systems for Data Analytics

Session #1 – Systems Attributes for Data Analytics – Single System

Murali P
muralip@wilp.bits-pilani.ac.in
[Saturday – 04:30 PM]

Agenda

- Review of Systems Attributes for Data Analytics
 - Single System
 - Memory Heirarchy
 - Locality of Reference



Review: Systems Attributes for Data Analytics - Single System

Courtesy: Prof Sundar B slides

Storage for Data

- **Structured Data**
 - E.g. Relational Data
- **Semi-structured Data**
 - E.g. HTML pages, XML data, JSON, CSV files, Email, NoSQL DB, etc.
- **Un-structured Data**
 - E.g. X-ray images, audio/video/photo files, word processing docs, books, journals, health records, metadata, etc.

Anecdotal Evidence

- *Most of the data today is semi-structured / unstructured.*

Kinds of Data and Forms of Storage



[3]

- Today,

Kind of Data	Form of Storage	Example (Products)
Structured (Relational)	Relational Databases	Oracle, MSSQL, and MySQL; SimpleDB (Amazon)
Semi- structured / Unstructured	File Systems or Object Storages or NOSQL databases (including XML databases)	MongoDB; S3, Elastic FS (Amazon)

Discussion/Assignment:

How do we access data in relational vs semi-structured vs unstructured data?

Compare with real examples

Data Location: Memory vs. Storage

- Computational Data is stored in
 - Primary Memory (a.k.a. Memory)

Use and Throw

vs.

- Persistent Data is stored in
 - i.e. Secondary Memory (a.k.a Storage)

Multiple runs

Questions / Exercises

1. What does “persistent” refer to? Is it same as non-volatile?
2. Identify examples of these two kinds of data
3. Identify technologies suitable for the two kinds of data

Data Location: Memory vs. Storage vs. Network

- Data accessed from a *local store*
 - i.e. storage attached to a computer
- vs.
- Data accessed from a *remote store* / remote processor
 - i.e. storage hosted on the network (or *storage attached to a computer hosted on the network*)

Question

- What is the difference in the form of access?

There is a link between
form of data and form of storage

Cost of Access: *Memory vs. Storage* *vs. Network*



- Exercise:
 - What are the typical access times?
 1. RAM
 2. Hard Disk
 3. Ethernet LAN
 - Access Time Parameters: ***Latency*** and ***Bandwidth***
 - When and how do these parameters matter?
 - Identify mechanisms used to alleviate the access time delays in each case.

Memory Bandwidth Requirement

[3]



- Total bandwidth to constantly feed processor:
 - 40GBps

How to give a typical 4-core processor
about 40GB of data every second?

Memory Hierarchy – Motivation

- A **Memory Hierarchy** amortizes cost in computer architecture:
 - fast (*and therefore costly*) but small-sized memory to
 - large-sized but slow (*and therefore cheap*) memory

Memory Hierarchy

- Original:



- Modern:



Discussion/Assignment:

How do we
Reconcile Memory Bandwidth Requirement
with the Memory Hierarchy?



Locality of Reference

Courtesy: Prof Sundar B slides

Locality of Reference(s)

- The Principle of Locality of Reference(s)
 - The locus of data access – and hence that of *memory references* – *is small at any point during program execution.*
 - *more like an Observation*
 - **Temporal Locality**
 - **Spatial Locality**
-

Locality of References - Temporal Locality

- **Temporal Locality**
 - Data that is accessed (at a point in program execution) is *likely to be accessed again in the near future:*
 - i.e. data is likely to be repeatedly accessed in a short span of time during execution

Locality of References - Temporal Locality

- **Temporal Locality**
 - Data that is accessed (at a point in program execution) is *likely to be accessed again in the near future:*
 - i.e. data is likely to be repeatedly accessed in a short span of time during execution
- Examples (*of manifestation of Temporal Locality*)
 1. Instructions in the body of a loop
 2. Parameters / Local variables of a function / procedure
 3. Data (or a variable) that is computed iteratively
 - e.g. a cumulative sum or product

Locality of References - Spatial Locality

- **Spatial Locality**
 - Data accessed (at a point in program execution) is likely located adjacent to data that is to be accessed in near future:
 - i.e. data accessed in a short span during execution is likely to be within a small region (in memory)

Locality of References - Spatial Locality

- **Spatial Locality**
 - Data accessed (at a point in program execution) is likely located adjacent to data that is to be accessed in near future:
 - i.e. data accessed in a short span during execution is likely to be within a small region (in memory)
- Examples (*of manifestation of Spatial Locality*)
 - Linear sequences of instructions
 - Elements of Arrays (accessed sequentially)

Locality of References - Spatial Locality

- **Spatial Locality**
 - Data accessed (at a point in program execution) is likely located adjacent to data that is to be accessed in near future:
 - i.e. data accessed in a short span during execution is likely to be within a small region (in memory)
- Examples (*of manifestation of Spatial Locality*)
 - Linear sequences of instructions
 - Elements of Arrays (accessed sequentially)

Question: Accessing nodes in a linked list sequentially may violate this principle. Why?

Memory Hierarchy and Locality

- A memory hierarchy is effective only due to **Locality** exhibited by programs (and the data they access)!
 - Longer the range of execution time of the program, larger is the locus of data accesses:
 - this aligns with the memory hierarchy:
 - *increasing size with increasing access time of memory levels*

Locality Example 1: Matrices

- Consider matrices and an operation such as the *addition of two matrices*:
 - elements $M[i,j]$ may be accessed either *row by row* or *column by column*

Locality Example 1: Matrices

- Consider matrices and an operation such as the *addition of two matrices*:
 - elements $M[i,j]$ may be accessed either *row by row* or *column by column*
- i.e. one can write the addition algorithm as:

```
for (i=0; i<N; i++)
  for(j=0; j<N; j++)
    M3 [i,j] = M2 [i,j] + M1 [i,j]
```

1	2	3
4	5	6
7	8	9

Locality Example 1: Matrices

- Consider matrices and an operation such as the *addition of two matrices*:
 - elements $M[i,j]$ may be accessed either *row by row* or *column by column*
- i.e. one can write the addition algorithm as:

```
for (i=0; i<N; i++)
  for(j=0; j<N; j++)
    M3 [i,j] = M2 [i,j] + M1 [i,j]
```

- or as

```
for (j=0; j<N; j++)
  for(i=0; i<N; i++)
    M3 [i,j] = M2 [i,j] + M1 [i,j]
```

1	2	3
4	5	6
7	8	9

1	4	7
2	5	8
3	6	9

Locality Example 1: Matrices

- Consider matrices and an operation such as the *addition of two matrices*:
 - elements $M[i,j]$ may be accessed either *row by row* or *column by column*
- i.e. one can write the addition algorithm as:

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    M3 [i,j] = M2 [i,j] + M1 [i,j]
```

- or as

```
for (j=0; j<N; j++)
  for (i=0; i<N; i++)
    M3 [i,j] = M2 [i,j] + M1 [i,j]
```

Time Complexity of both these algorithms is the same **but**:

- locality varies (Why?) and
- so does performance!

Locality Example 1: Matrices *[contd.]*

- Matrices are 2-dimensional but memory is 1-dimensional!
 - i.e. matrices are stored in row-major order or in column-major order.
- Impact?

Locality Example 1: Matrices

[contd.]

- Matrices are 2-dimensional but memory is 1-dimensional!
 - i.e. matrices are stored in row-major order or in column-major order.
- Impact?

#rows	#cols	# elements	rowTime	colTime
512	512	262144	1000	1000
1024	1024	1048576(1M)	3999	5999
2048	2048	4M	15997	32995
4096	4096	16M	62990	141978
8192	8192	64M	253961	670898
16384	16384	256M	1014846	3013541

Locality Example 2: Partitioning in QuickSort

- Partitioning is a step in QuickSort:
 - Given a *pivot p*, partition a list L such that:
 - $p = L[i]$ for some i where $0 \leq i < n$ and
 - for all j where $0 < j < n$ and $n = \text{length}(L)$
 - $j < i$ implies $L[j] \leq p$ and
 - $j > i$ implies $L[j] > p$

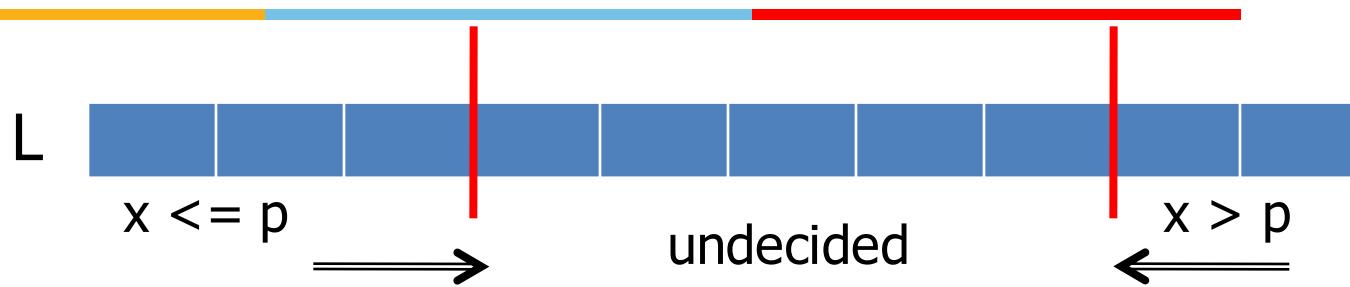
Locality Example 2: Partitioning in QuickSort

- Partitioning is a step in QuickSort:
 - Given a *pivot p*, partition a list L such that:
 - $p = L[i]$ for some i where $0 \leq i < n$ and
 - for all j where $0 < j < n$ and $n = \text{length}(L)$
 - $j < i$ implies $L[j] \leq p$ and
 - $j > i$ implies $L[j] > p$

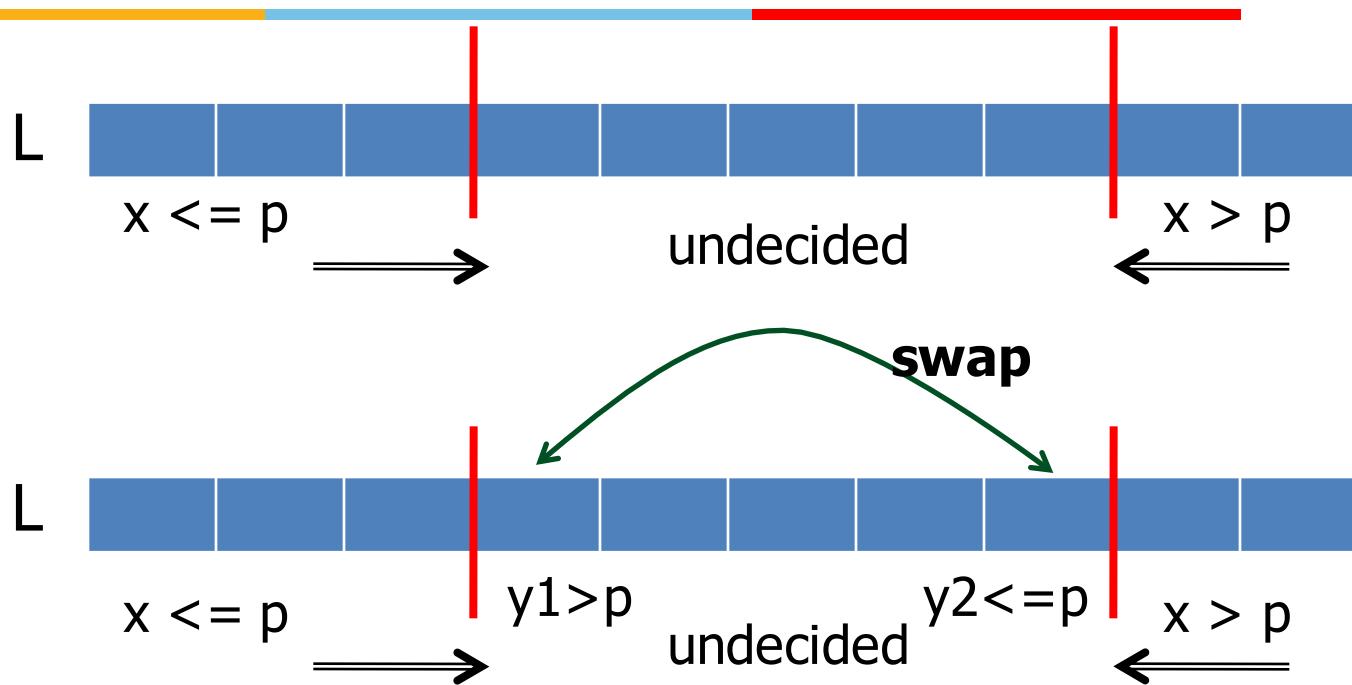
- Hoare's Partitioning Algorithm:

```
/* assume L[lo] = p */  
i=lo+1; j=hi;  
while (i <= j) {  
    while (L[i]<=p) i++; while (L[j]>p) j--;  
    if (i<j) swap(L[i], L[j]);  
}
```

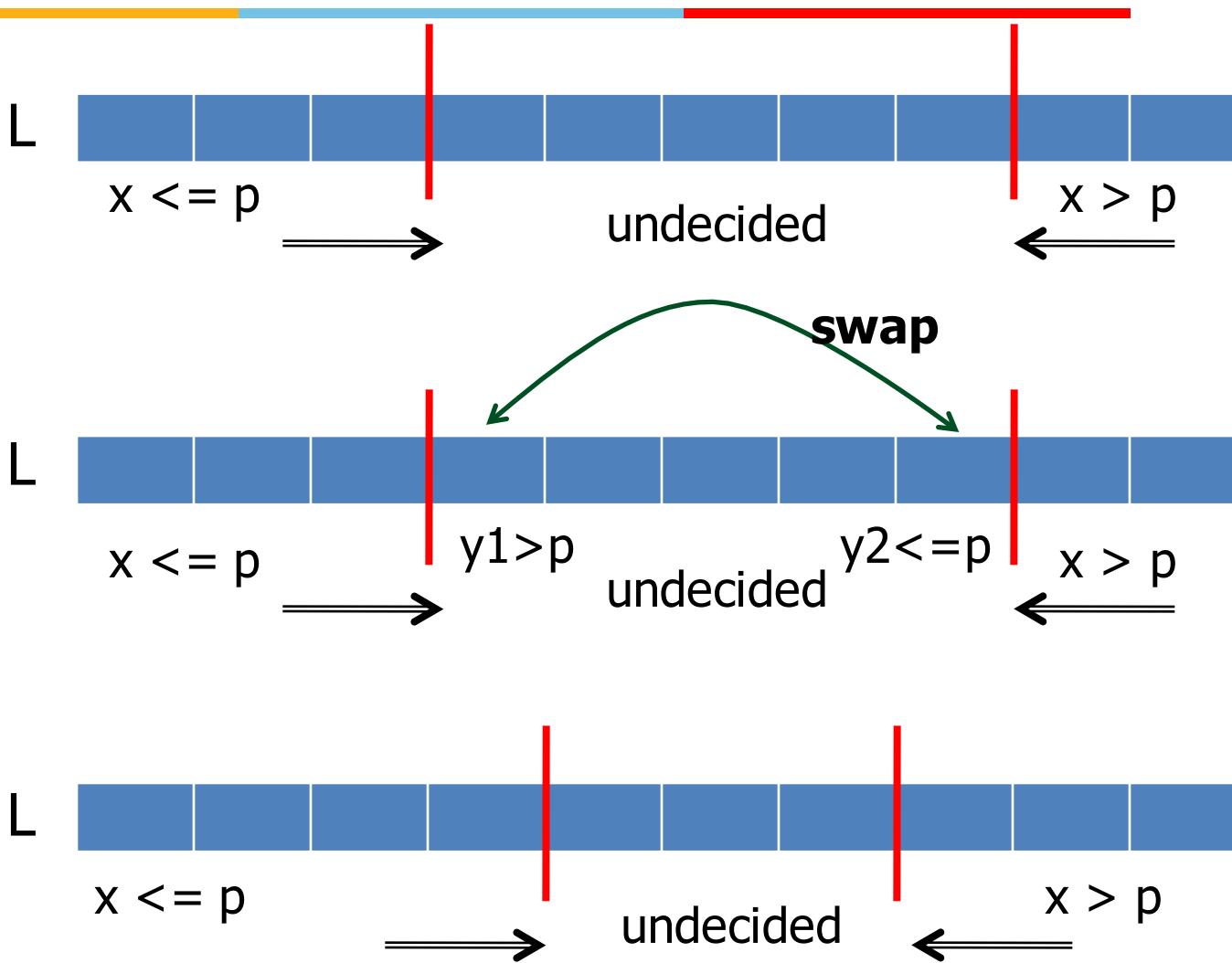
(Hoare's) Partitioning : (pivot p in L)



(Hoare's) Partitioning : (pivot p in L)



(Hoare's) Partitioning : (pivot p in L)



Partitioning and Locality

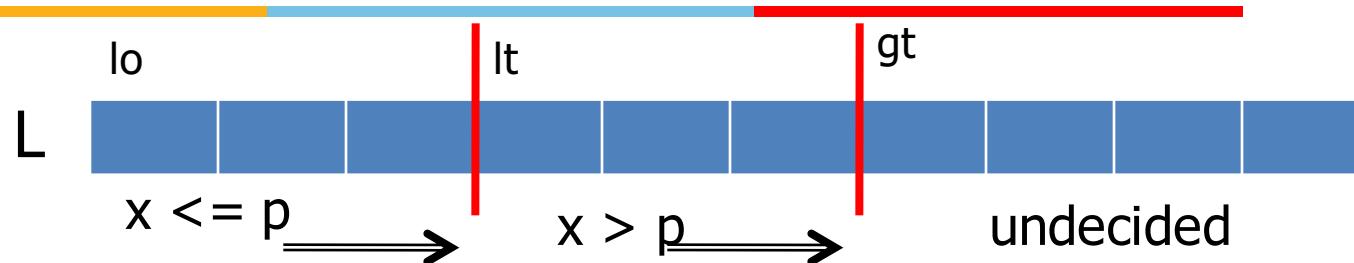
- Recall the inner loops of the Partition algorithm:
 - Array is accessed left-to-right (**L-R**) in one loop, and right-to-left (**R-L**) in the next.
- And these two loops are repeated in the outer loop
 - i.e. the ***locus*** alternates within each iteration of the outer loop, and from the end of one iteration to the start of the next

I1: L-R	X	X								
I1: R-L									Y	Y
I2: L-R			X	X						
I2: R-L								Y		
I3: L-R					X					
I3: R-L							Y			

Back to the Partitioning Algorithm:

- Can you access the array elements from one end instead of both ends?

Locality-Aware Partitioning : (pivot p in L)



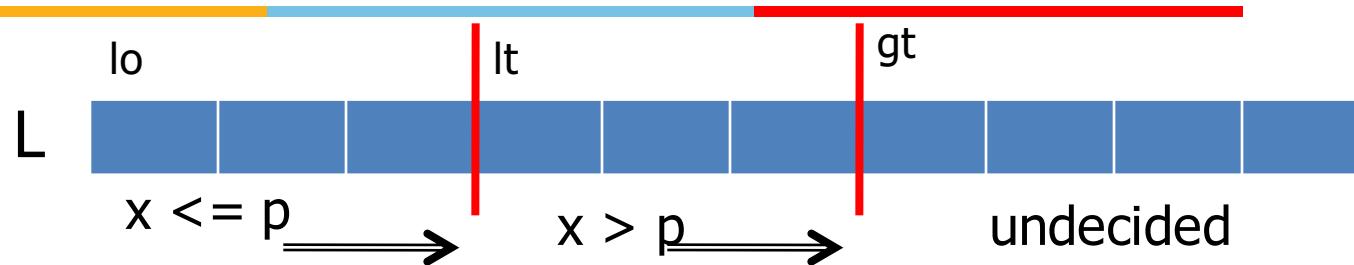
Maintain these sub-lists (and *the invariants*)
i.e.

$\text{forall } j: lo \leq j \leq lt \rightarrow L[j] \leq \text{piv}$

and

$\text{forall } j: lt < j \leq gt \rightarrow L[j] > \text{piv}$

Locality-Aware Partitioning : (pivot p in L)



Lomuto's partitioning

Maintain these sub-lists (and *the invariants*)
i.e.

$\text{forall } j: \text{lo} \leq j \leq \text{lt} \rightarrow L[j] \leq \text{piv}$

and

$\text{forall } j: \text{lt} < j \leq \text{gt} \rightarrow L[j] > \text{piv}$

Exercise:

- Code the locality-aware partitioning algorithm.
- Determine the impact of locality by measuring the time taken – for different data sizes – the classic algorithm and this one.



Thank you !



BITS Pilani
Pilani Campus



DSECL ZG517 - Systems for Data Analytics

Session #3 – Systems Attributes for Data Analytics –
Parallel/Distributed System

Murali P
muralip@wilp.bits-pilani.ac.in
[Saturday – 04:30 PM]

Agenda

- Systems Attributes for Data Analytics – Parallel and Distributed Systems
 - Motivation
 - Parallel/Distributed Processing and Data



Review

Locality Example 2: Partitioning in QuickSort

- Partitioning is a step in QuickSort:
 - Given a *pivot p*, partition a list L such that:
 - $p = L[i]$ for some i where $0 \leq i < n$ and
 - for all j where $0 < j < n$ and $n = \text{length}(L)$
 - $j < i$ implies $L[j] \leq p$ and
 - $j > i$ implies $L[j] > p$

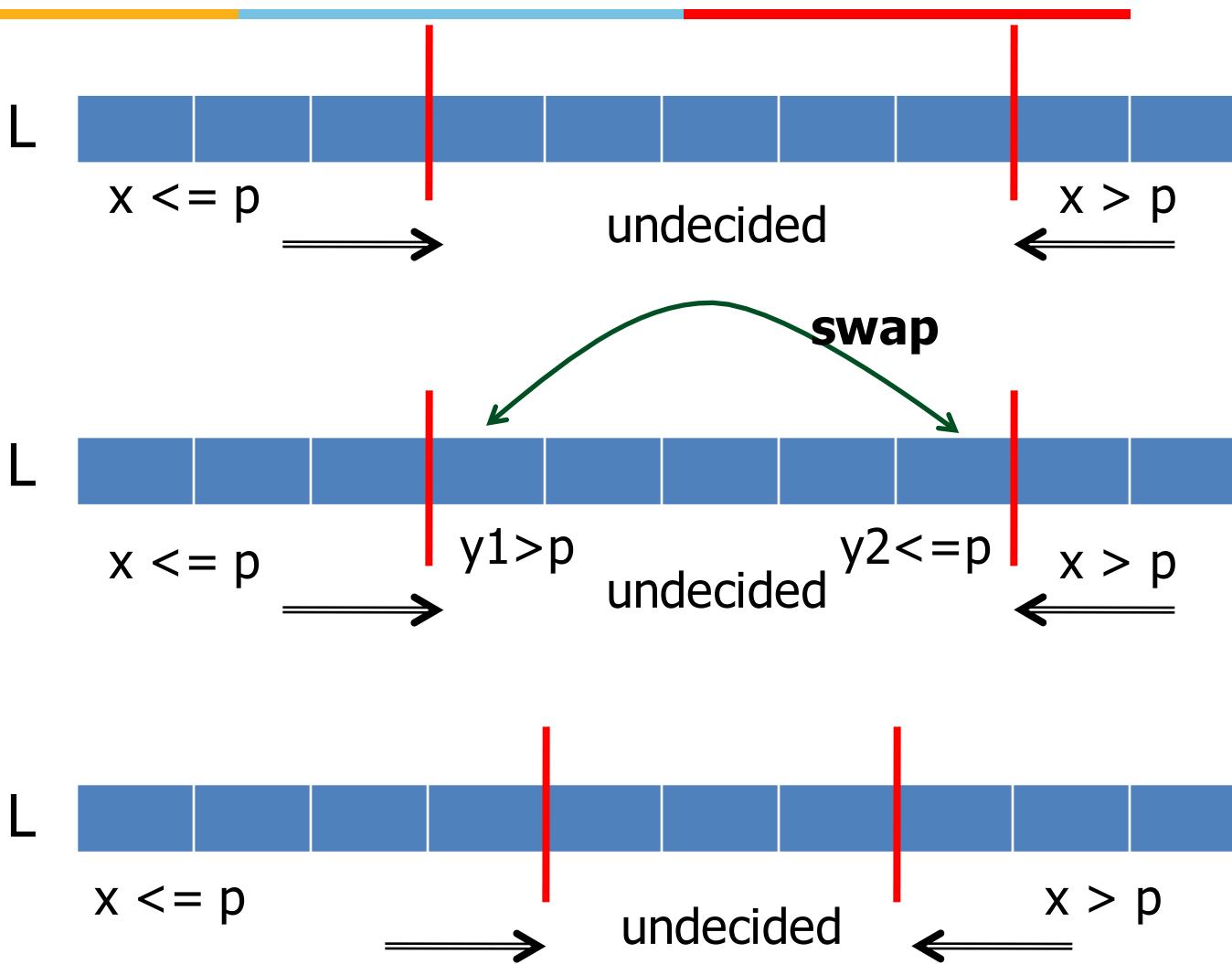
Locality Example 2: Partitioning in QuickSort

- Partitioning is a step in QuickSort:
 - Given a *pivot p*, partition a list L such that:
 - $p = L[i]$ for some i where $0 \leq i < n$ and
 - for all j where $0 < j < n$ and $n = \text{length}(L)$
 - $j < i$ implies $L[j] \leq p$ and
 - $j > i$ implies $L[j] > p$

- Hoare's Partitioning Algorithm:

```
/* assume L[lo] = p */  
i=lo+1; j=hi;  
while (i <= j) {  
    while (L[i]<=p) i++; while (L[j]>p) j--;  
    if (i<j) swap(L[i], L[j]);  
}
```

(Hoare's) Partitioning : (pivot p in L)



Partitioning and Locality

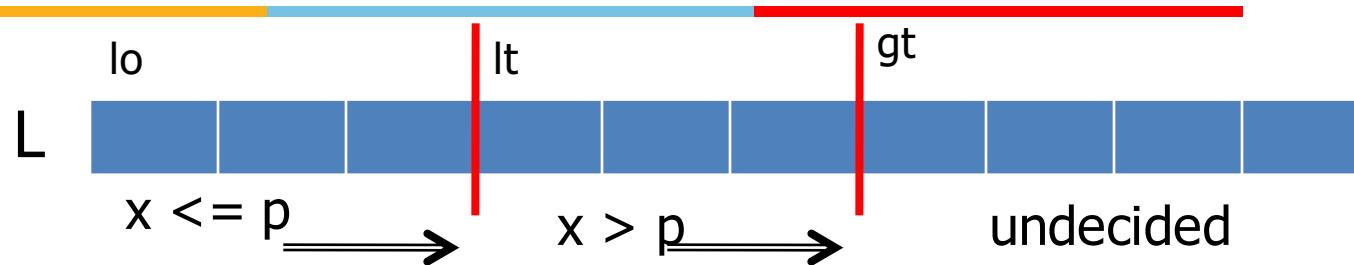
- Recall the inner loops of the Partition algorithm:
 - Array is accessed left-to-right (**L-R**) in one loop, and right-to-left (**R-L**) in the next.
- And these two loops are repeated in the outer loop
 - i.e. the ***locus*** alternates within each iteration of the outer loop, and from the end of one iteration to the start of the next

I1: L-R	X	X								
I1: R-L									Y	Y
I2: L-R			X	X						
I2: R-L								Y		
I3: L-R					X					
I3: R-L							Y			

Back to the Partitioning Algorithm:

- Can you access the array elements from one end instead of both ends?

Locality-Aware Partitioning : (pivot p in L)



Lomuto's partitioning

Maintain these sub-lists (and *the invariants*)
i.e.

$\text{forall } j: \text{lo} \leq j \leq \text{lt} \rightarrow L[j] \leq \text{piv}$

and

$\text{forall } j: \text{lt} < j \leq \text{gt} \rightarrow L[j] > \text{piv}$

Exercise:

- Code the locality-aware partitioning algorithm.
- Determine the impact of locality by measuring the time taken – for different data sizes – the classic algorithm and this one.



Review: Systems Attributes for Data Analytics – Parallel/Distributed

Courtesy: Prof Sundar B slides

Motivation: High Performance

Question

- *Can performance be improved by doing multiple computations in parallel i.e. at the same time?*

Contexts for High Performance

High Performance requirement may arise due to

- Problem complexity
 - Size of data
- or

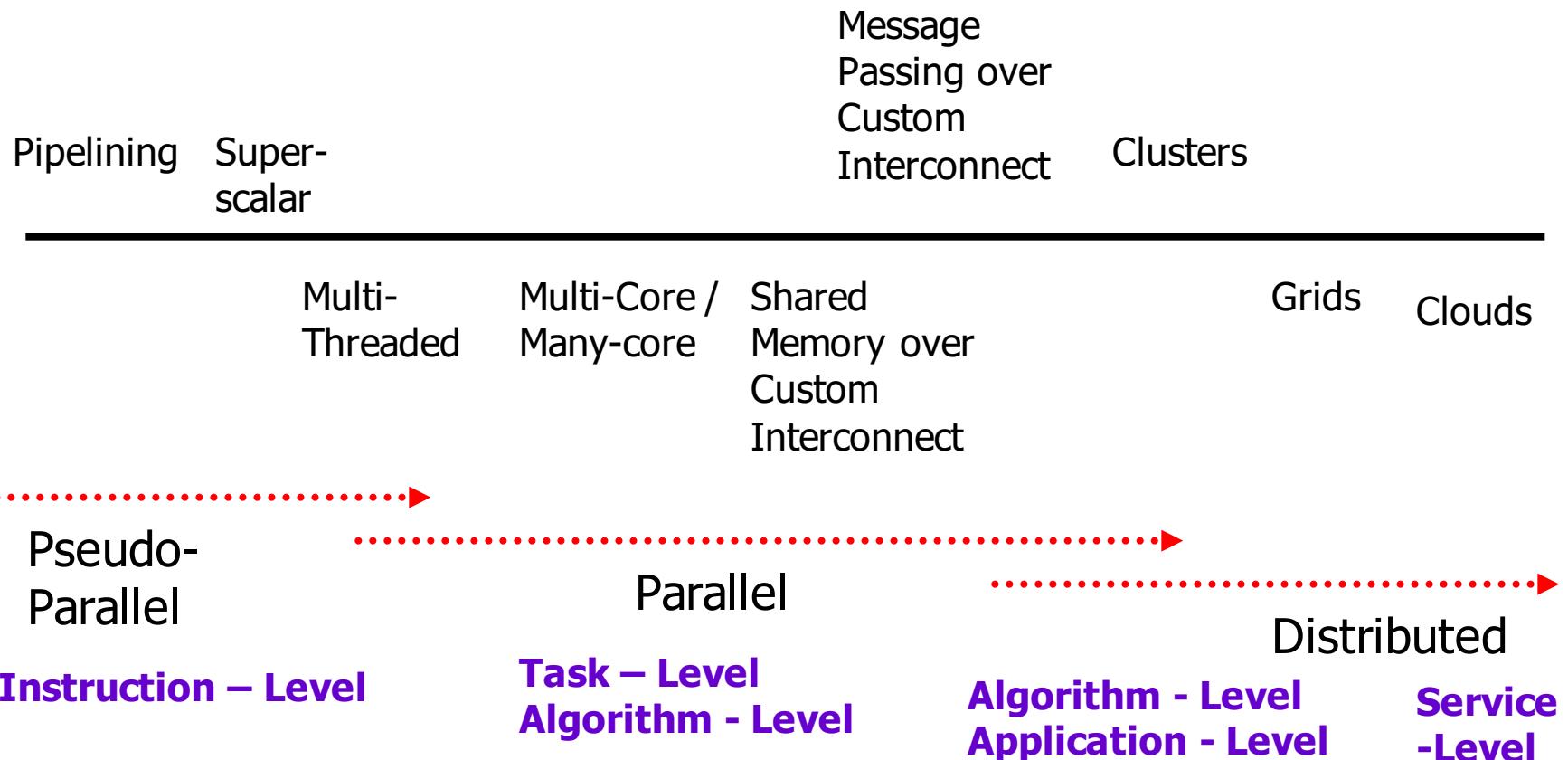
or both.

Exercise

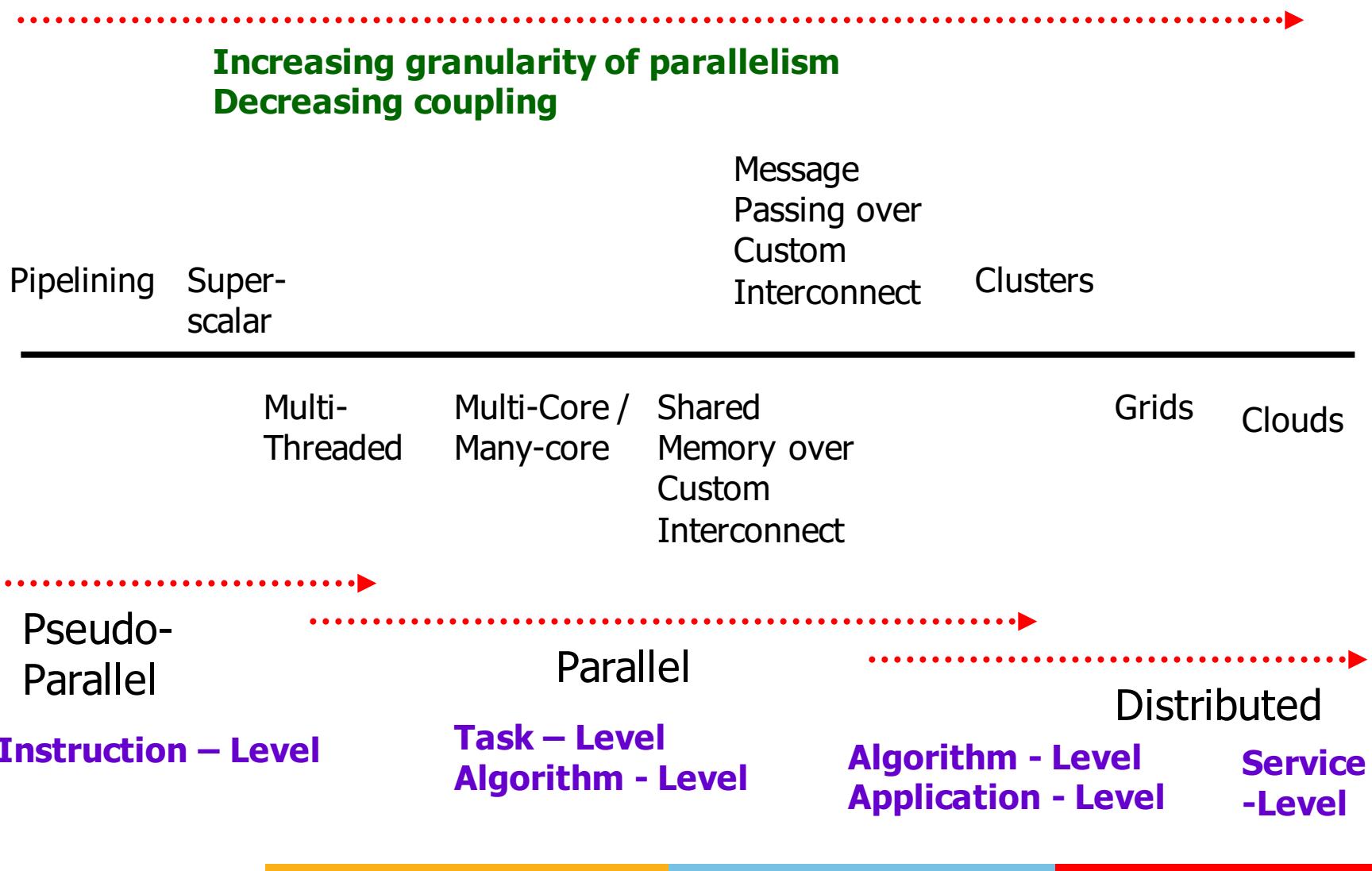
In each of the following cases, identify / argue the motivation for high performance requirement:

1. Airline scheduling
2. Summary Statistics of Historic Sales Data of a Retailer
3. Web Crawler

Spectrum of Parallelism



Spectrum of Parallelism





BITS Pilani
Pilani Campus



Parallel Processing: Memory Access Models

Shared Memory Model:

- Multiple tasks access data via a shared logical address space (i.e. a single virtual memory)

Distributed Memory Model:

- Multiple tasks – *executing a single program* – access data from separate (and isolated) address spaces.(i.e. separate virtual memories)

Questions:

1. Does the shared memory model refer to a single program? Why or why not?
2. If a single program is being executed in a distributed memory model, but memories are isolated, how can processors access all the data?

Distributed Memory and Message Passing

In a **Distributed Memory model**,

- Data has to be moved across Virtual Memories:
 - i.e. a data item in VMem_1 produced by task T_1 has to be “communicated” to task T_2 so that
 - T_2 can make a copy of the same in VMem_2 and use it.

Whereas in a **Shared Memory model**,

- task T_1 write the data item into a memory location and T_2 can read the same
 - as that *memory location is part of the logical address space that is shared between the tasks*

Computing Model for Message Passing



Implications:

- Each data item must be located in one of the address spaces
 - i.e. data must be partitioned explicitly and placed (i.e. distributed)
 - All interactions between processes require explicit communication i.e. ***passing of messages***
 - In the simplest form:
 - a sender (who has the data) and
 - a receiver (who has to access the data)
- must co-operate for exchange of data**

Message Passing Model – Separate Address Spaces

Use of separate address spaces complicates programming but this complication is usually restricted to one or two phases:

- Partitioning the input data
 - This improves locality
 - i.e. each process is enabled to access data from within its address space,
 - » which in turn is likely to be mapped to the memory hierarchy of the processor in which the process is running
- Merging / Collecting the output data
 - This is required if each task is producing outputs that have to be combined.

Message Passing Model - Interactions



Use of message passing for interaction complicates programming:

- Process that owns or produces the data must participate in message exchanges
 - even if these have nothing to do with its own flow of computation.
- Communication patterns that are dynamic and/or unstructured result in complex programs:
 - Messaging code – *which may be housekeeping code for data producer* – is scattered and tangled with other code.

Shared Memory Model: Implications for Architecture

- The most straightforward way to realize a shared memory model onto an architecture is a ***shared memory system***:
- i.e.
 - Physical memory (or memories) are accessible by all processors
 - A single program is implemented as a collection of threads (with one or more threads scheduled in a processor) and
 - The single (logical) address space is mapped onto the physical memory (or memories).

Shared Memory Model: Multi-Threaded Programming

- The most straightforward way to run a task on a shared memory model is a ***multi-threaded program*** i.e.
 - A program is a *collection of threads* (with threads scheduled by the OS or the runtime environment of a language).
- Logical Model
 - Shared address space
- Protection Model
 - Fully shared space
- Running Model (for application Threads)
 - Stack is local
 - (Why?)
 - Rest are shared
 - - typically these include *code area*, *global area*, and *heap*.

Computing Model for Message Passing



- Implications:
 - Each data item must be located in one of the address spaces
 - i.e. data must be partitioned explicitly and placed (i.e. distributed)
 - All interactions between processes require explicit communication i.e. ***passing of messages***
 - In the simplest form:
 - a sender (who has the data) and
 - a receiver (who has to access the data)
- must co-operate for exchange of data

Message Passing Model – Separate Address Spaces

- Use of separate address spaces complicates programming
- but this complication is usually restricted to one or two phases:
 - Partitioning the input data
 - This improves locality
 - i.e. each process is enabled to access data from within its address space,
 - » which in turn is likely to be mapped to the memory hierarchy of the processor in which the process is running
 - Merging / Collecting the output data
 - This is required if each task is producing outputs that have to be combined.

Message Passing Model - Interactions



- Use of message passing for interaction complicates programming:
 - Process that owns or produces the data must participate in message exchanges
 - even if these have nothing to do with its own flow of computation.
 - Communication patterns that are dynamic and/or unstructured result in complex programs:
 - Messaging code – *which may be housekeeping code for data producer* – is scattered and tangled with other code.

Structure of Message Passing Programs



- Synchronous execution of tasks is difficult to achieve given an environment of separate processes (and address spaces)
 - Tasks may not be the same.
 - Although processes may run on a homogeneous environment (e.g. a *cluster where all nodes are of the same configuration*),
 - execution speeds may vary.
- On the other hand asynchronous execution is hard to reason about:
 - Proving properties about the progress of programs is dependent on timing – or at least, precedence, – information.



Thank you !



DSECL ZG517 - Systems for Data Analytics

Session #4 – Systems Attributes for Data Analytics –
Parallel/Distributed–2

Murali P

muralip@wilp.bits-pilani.ac.in

[Saturday – 04:15 PM]

BITS Pilani
Pilani Campus

This presentation uses public contents shared by authors of text books and other relevant web resources. Further, works of Professors from BITS are also used freely in preparing this presentation.

Agenda

Review

Example programming environments

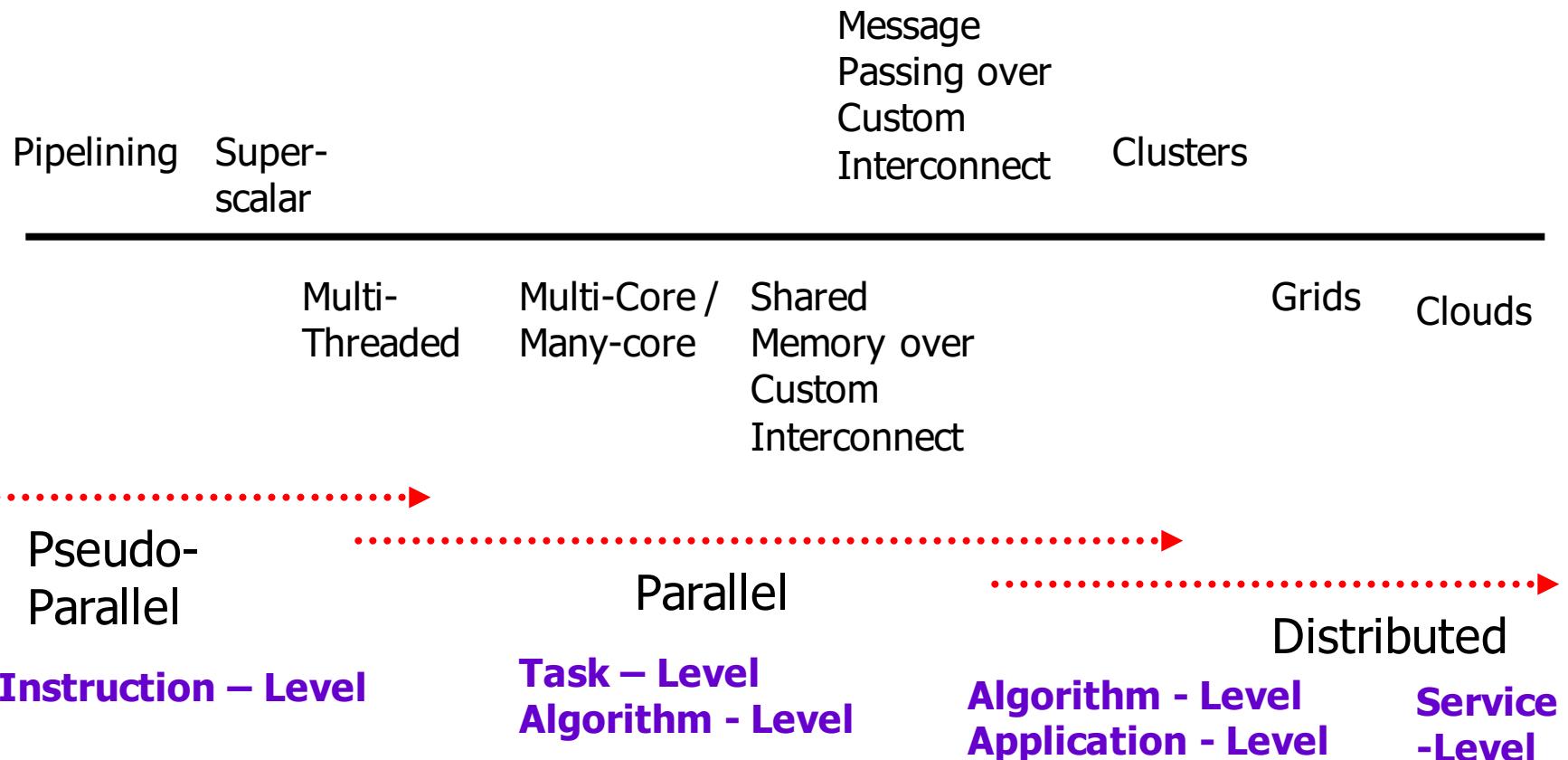
- Message passing
- Shared memory



Review: Systems Attributes for Data Analytics – Parallel/Distributed

Courtesy: Prof Sundar B slides

Spectrum of Parallelism



Parallel Processing: Memory Access Models

Shared Memory Model:

- Multiple tasks access data via *a shared logical address space* (i.e. a single virtual memory)

Distributed Memory Model:

- Multiple tasks – *executing a single program* – access data from separate (and isolated) address spaces.(i.e. separate virtual memories)

Distributed Memory and Message Passing

In a **Distributed Memory model**,

- Data has to be moved across Virtual Memories:
 - i.e. a data item in VMem_1 produced by task T_1 has to be “communicated” to task T_2 so that
 - T_2 can make a copy of the same in VMem_2 and use it.

Whereas in a **Shared Memory model**,

- task T_1 write the data item into a memory location and T_2 can read the same
 - as that *memory location is part of the logical address space that is shared between the tasks*

Computing Model for Message Passing



Implications:

- Each data item must be located in one of the address spaces
 - i.e. data must be partitioned explicitly and placed (i.e. distributed)
 - All interactions between processes require explicit communication i.e. ***passing of messages***
 - In the simplest form:
 - a sender (who has the data) and
 - a receiver (who has to access the data)
- must co-operate for exchange of data**

Message Passing Model – Separate Address Spaces

Use of separate address spaces complicates programming but this complication is usually restricted to one or two phases:

- Partitioning the input data
 - This improves locality
 - i.e. each process is enabled to access data from within its address space,
 - » which in turn is likely to be mapped to the memory hierarchy of the processor in which the process is running
- Merging / Collecting the output data
 - This is required if each task is producing outputs that have to be combined.

Message Passing Model - Interactions



Use of message passing for interaction complicates programming:

- Process that owns or produces the data must participate in message exchanges
 - even if these have nothing to do with its own flow of computation.
- Communication patterns that are dynamic and/or unstructured result in complex programs:
 - Messaging code – *which may be housekeeping code for data producer* – is scattered and tangled with other code.

Shared Memory Model: Implications for Architecture

- The most straightforward way to realize a shared memory model onto an architecture is a ***shared memory system***:
- i.e.
 - Physical memory (or memories) are accessible by all processors
 - A single program is implemented as a collection of threads (with one or more threads scheduled in a processor) and
 - The single (logical) address space is mapped onto the physical memory (or memories).

Shared Memory Model: Multi-Threaded Programming

- The most straightforward way to run a task on a shared memory model is a ***multi-threaded program*** i.e.
 - A program is a *collection of threads* (with threads scheduled by the OS or the runtime environment of a language).
- Logical Model
 - Shared address space
- Protection Model
 - Fully shared space
- Running Model (for application Threads)
 - Stack is local
 - (Why?)
 - Rest are shared
 - - typically these include *code area*, *global area*, and *heap*.



Message passing MPI

The Message-Passing Model

A **process** is (traditionally) a program counter and address space.

Processes may have multiple **threads** (program counters and associated stacks) sharing a single address space.

MPI is for communication among processes, which have separate address spaces.

Interprocess communication consists of

Synchronization

Movement of data from one process's address space to another's.

What is MPI?

A *message-passing library specification*

extended message-passing model

not a language or compiler specification

not a specific implementation or product

For parallel computers, clusters, and heterogeneous networks

Designed to provide access to advanced parallel hardware for

end users

library writers

tool developers

Why Use MPI?

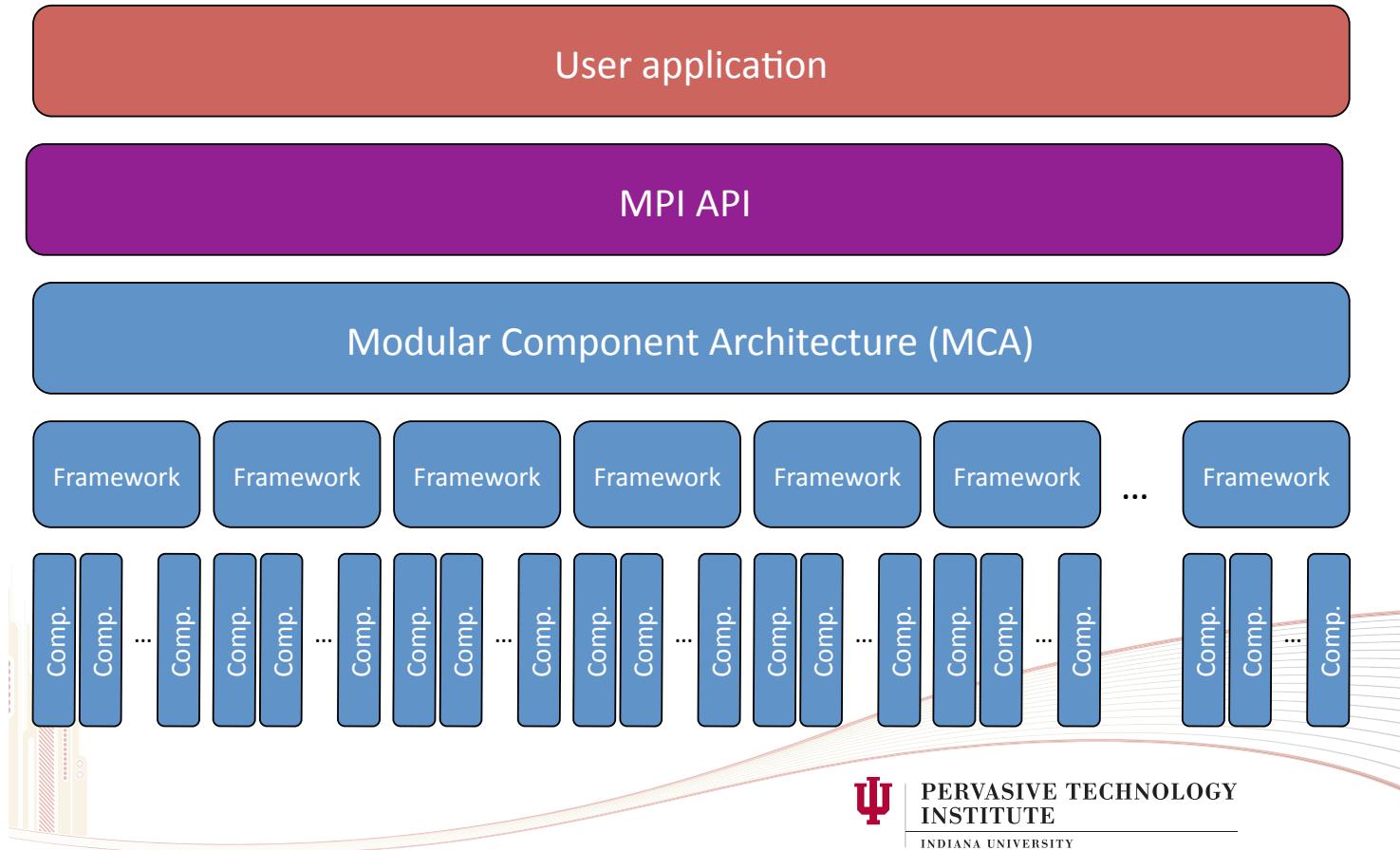
MPI provides a powerful, efficient, and *portable* way to express parallel programs

MPI was explicitly designed to enable libraries...

... which may eliminate the need for many users to learn (much of) MPI



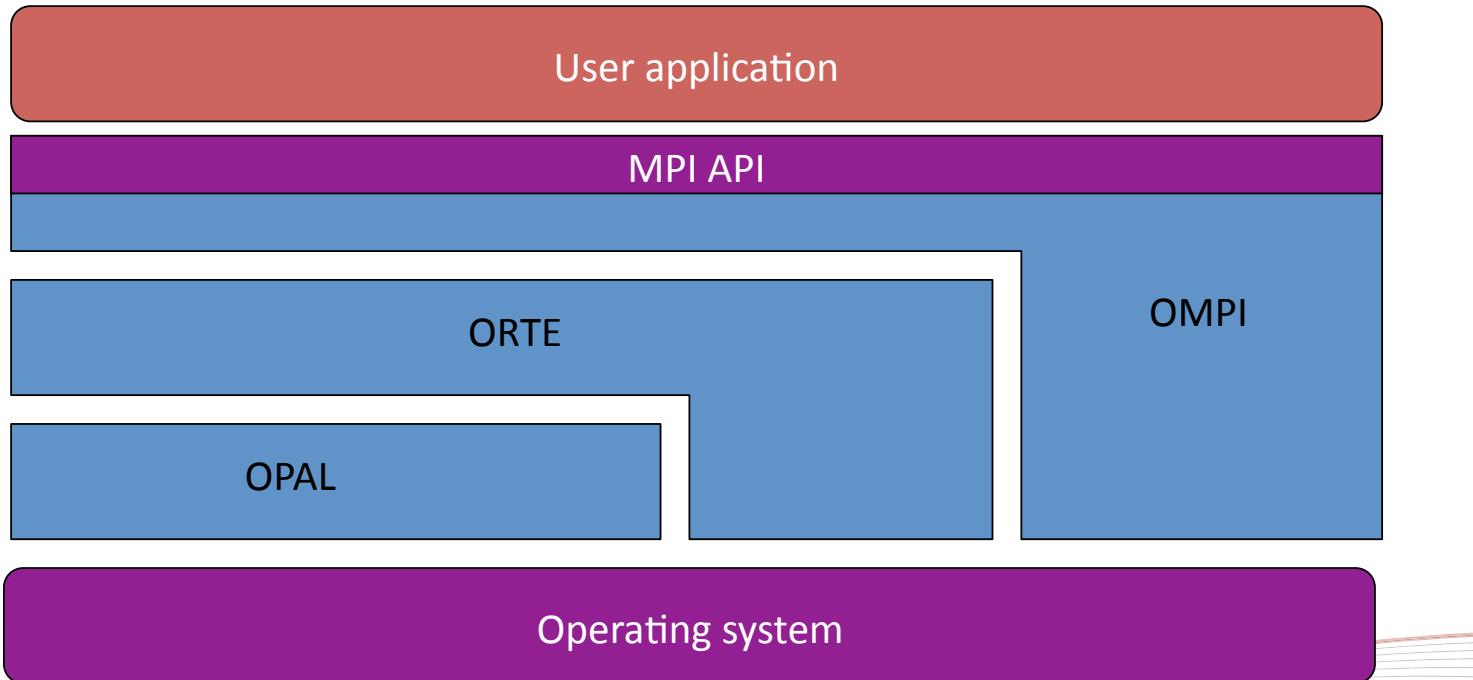
Plugin high level view



PERVASIVE TECHNOLOGY
INSTITUTE
INDIANA UNIVERSITY

Src: <https://www.open-mpi.org/papers/sc-2009/jjhursey-iu-booth.pdf>

Three main code sections



OMPI: Open MPI Layer

ORTE: Open MPI Runtime Environment

 PERVERSIVE TECHNOLOGY
INSTITUTE
INDIANA UNIVERSITY

OPAL: Open Portability Access Layer

A Minimal MPI Program (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    printf( "Hello, world!\n" );
    MPI_Finalize();
    return 0;
}
```

Finding Out About the Environment

Two important questions that arise early in a parallel program are:

How many processes are participating in this computation?

Which one am I?

MPI provides functions to answer these questions:

`MPI_Comm_size` reports the number of processes.

`MPI_Comm_rank` reports the *rank*, a number between 0 and *size*-1, identifying the calling process

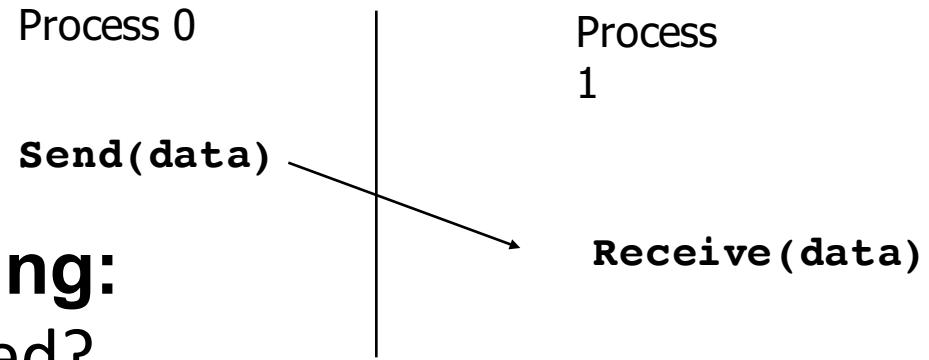
Better Hello (C)

```
#include "mpi.h"
#include <stdio.h>

int main( int argc, char *argv[] )
{
    int rank, size;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );
    printf( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

MPI Basic Send/Receive

We need to fill in the details in

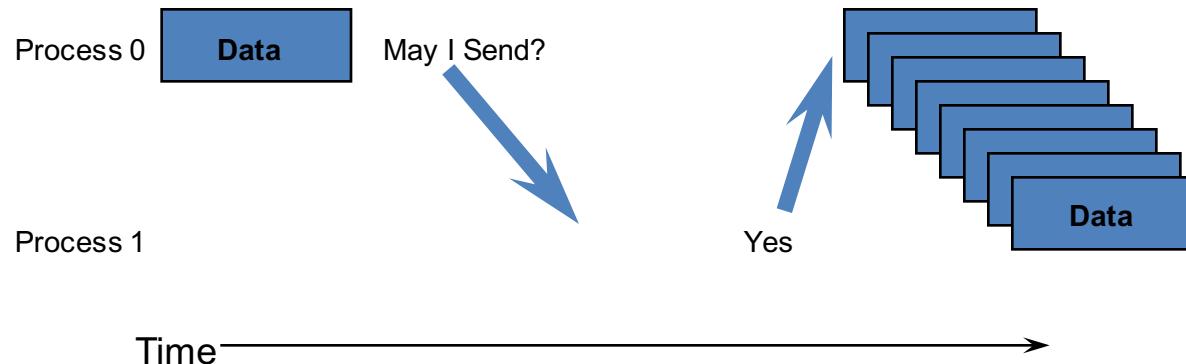


Things that need specifying:

- How will “data” be described?
- How will processes be identified?
- How will the receiver recognize/screen messages?
- What will it mean for these operations to complete?

What is message passing?

Data transfer plus synchronization



- Requires cooperation of sender and receiver
- Cooperation not always apparent in code

Some Basic Concepts

Processes can be collected into *groups*.

Each message is sent in a *context*, and must be received in the same context.

A group and context together form a ***communicator***.

A process is identified by its *rank* in the group associated with a communicator.

There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**.

MPI Basic (Blocking) Send

MPI_SEND (start, count, datatype, dest, tag, comm)

The message buffer is described by (start, count, datatype).

The target process is specified by dest, which is the rank of the target process in the communicator specified by comm.

When this function returns, the data has been delivered to the system and the buffer can be reused.

The message may not have been received by the target process.



MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

Waits until a matching (on source and tag) message is received from the system, and the buffer can be used.

source is rank in communicator specified by **comm**, or **MPI_ANY_SOURCE**.

status contains further information

Receiving fewer than count occurrences of datatype is OK, but receiving more is an error.

MPI is Simple

Many parallel programs can be written using just these six functions, only two of which are non-trivial:

MPI_INIT

MPI_FINALIZE

MPI_COMM_SIZE

MPI_COMM_RANK

MPI_SEND

MPI_RECV

26

Point-to-point (send/recv) isn't the only way...



openMPI

Tutorial

<https://computing.llnl.gov/tutorials/mpi/>



openMP

Source: Prof Chandra sekhar slides

OpenMP

- Motivation
- OpenMP basics
- OpenMP directives, clauses, and library routines

OpenMP: Motivation

- We have a sequential code,
 - we know which loop can be executed in parallel;
 - the program conversion is quite mechanical: we should just say that the loop is to be executed in parallel and let the compiler do the rest.
-
- OpenMP does exactly that!!!
-

What is OpenMP?

What does OpenMP stands for?

- Open specifications for Multi Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

OpenMP is an Application Program Interface (API) that may be used to explicitly direct *multi-threaded, shared memory parallelism*.

- API components: Compiler Directives, Runtime Library Routines, Environment Variables

OpenMP is a directive-based method to invoke parallel computations on share-memory multiprocessors

What is OpenMP?

- OpenMP API is specified for C/C++ and Fortran.
- OpenMP is not intrusive to the original serial code: instructions appear in comment statements for fortran and pragmas for C/C++.
- OpenMP website: <http://www.openmp.org>
 - Please note: Materials in this part of the lecture are taken from various OpenMP tutorials in the website and other places.

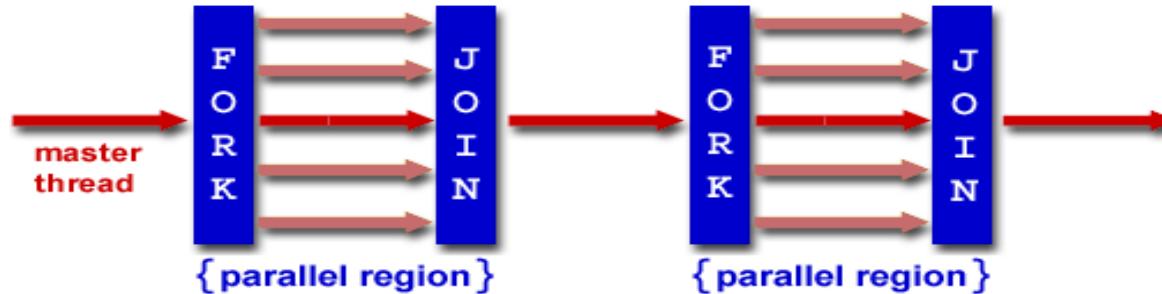
Why OpenMP?

- OpenMP is portable: supported by HP, IBM, Intel, SGI, SUN, and others
 - It is the de facto standard for writing **shared memory** programs.
- OpenMP can be implemented incrementally, one function or even one loop at a time.
 - A nice way to get a parallel program from a sequential program.

How to compile and run OpenMP programs?

- **Gcc 4.2 and above supports OpenMP 3.0**
 - `gcc –fopenmp a.c`
- **To run: ‘./a.out’**
 - To change the number of threads:
 - `setenv OMP_NUM_THREADS 4` (tcsh) or `export OMP_NUM_THREADS=4`(bash)

OpenMP execution model



- OpenMP uses the fork-join model of parallel execution.
 - All OpenMP programs begin with a single **master thread**.
 - The master thread executes sequentially until a **parallel region** is encountered, when it creates a **team of parallel threads (FORK)**.
 - When the team threads complete the parallel region, they synchronize and terminate, leaving only the master thread that executes sequentially (JOIN).

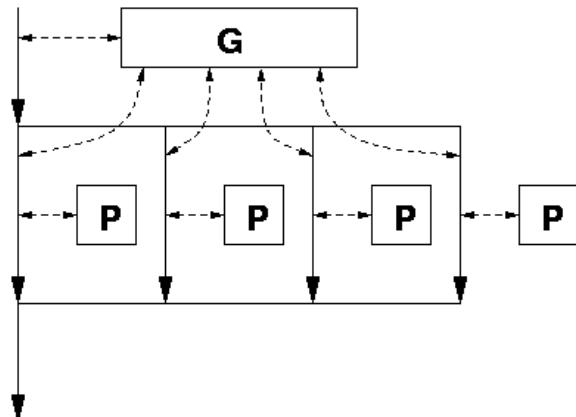
OpenMP general code structure



```
#include <omp.h>
main () {
    int var1, var2, var3;
    Serial code
    . . .
/* Beginning of parallel section. Fork a team of threads.
Specify variable scoping*/
#pragma omp parallel private(var1, var2) shared(var3)
{
    /* Parallel section executed by all threads */
    . . .
    /* All threads join master thread and disband*/
}
Resume serial code
. . .
}
```

Data model

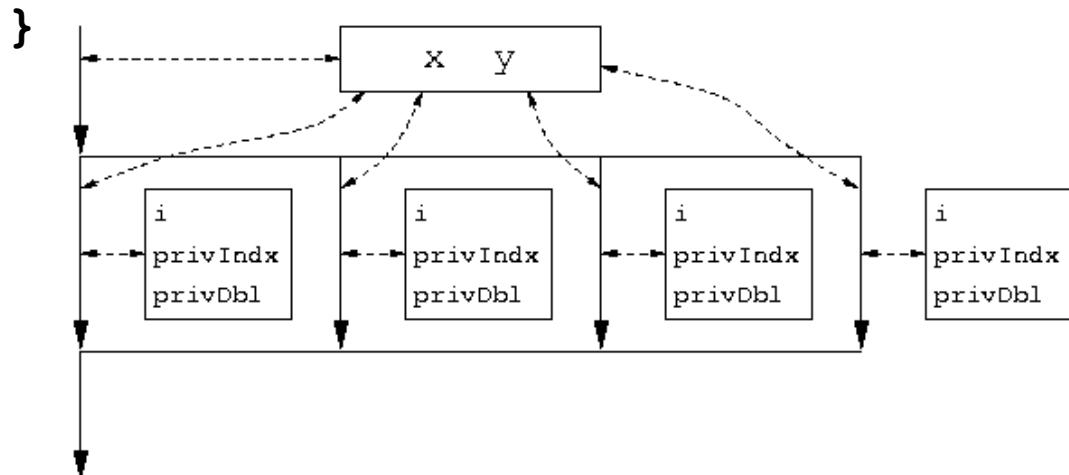
- Private and shared variables



P = private data space
 G = global data space

- Variables in the global data space are accessed by all parallel threads (**shared variables**).
- Variables in a thread's private space can only be accessed by the thread (**private variables**)
 - several variations, depending on the initial values and whether the results are copied outside the region.

```
#pragma omp parallel for private( privIdx, privDbl )
for ( i = 0; i < arraySize; i++ ) {
    for ( privIdx = 0; privIdx < 16; privIdx++ ) {
privDbl = ( (double) privIdx ) / 16;
        y[i] = sin( exp( cos( - exp( sin(x[i]) ) ) ) )
    ) + cos( privDbl );
}
}
```



Parallel for loop index is
Private by default.

execution context for "arrayUpdate_II"

- **The reduction clause:**

```
Sum = 0.0;  
#pragma parallel shared (n, x) private (l) reduction(+ : sum)  
{  
    For(l=0; l<n; l++)  
        sum = sum + x(l);  
}
```

- Updating sum must avoid racing condition
- With the reduction clause, OpenMP generates code such that the race condition is avoided.

The reduction clause

```
Sum = 0.0;  
#pragma parallel shared (n, x) private  
    (I) reduction(+ : sum)  
{  
    for(I=0; I<n; I++)  
        sum = sum + x(I);  
}
```

- Updating sum must avoid racing condition
- With the **reduction** clause, OpenMP generates code such that the race condition is avoided.

The `omp for` directive: example

```
#pragma omp parallel default(none) \
    shared(n,a,b,c,d) private(i)
{
    #pragma omp for nowait
    for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;

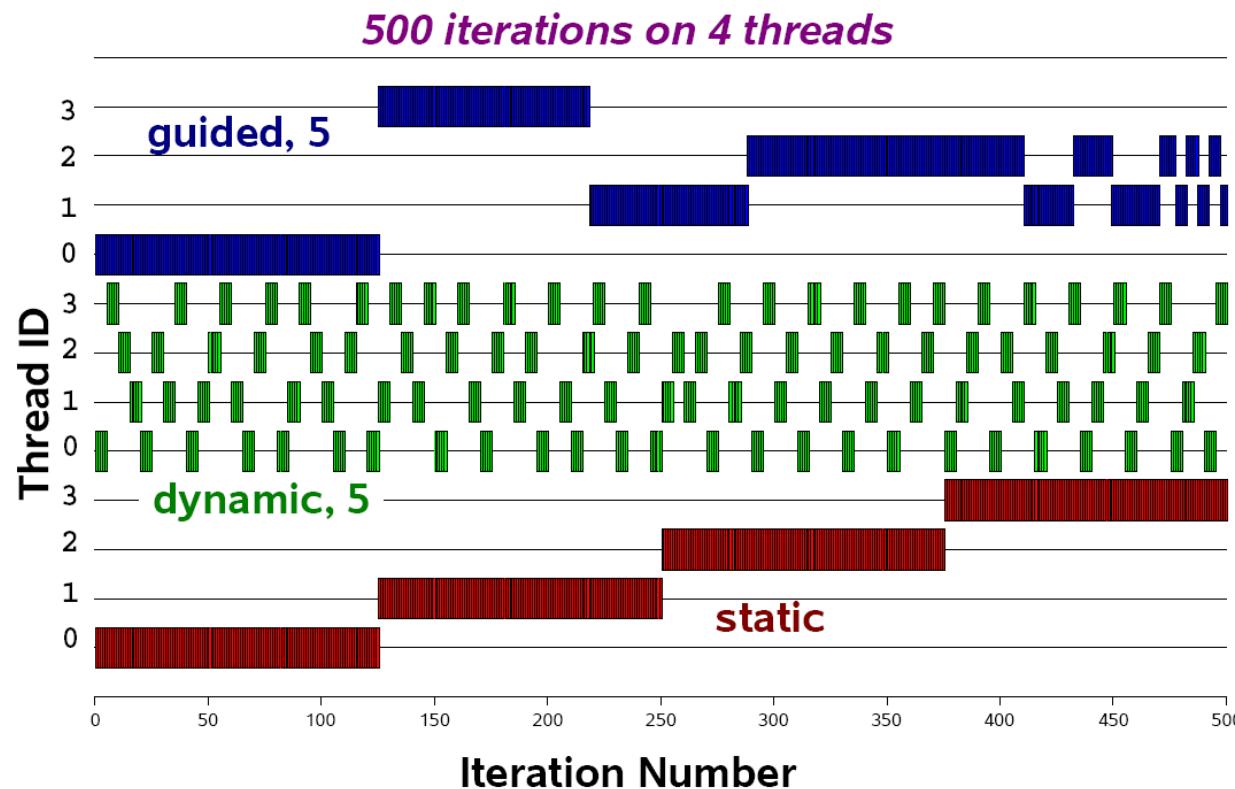
    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
}

/*-- End of parallel region --*/
(implied barrier)
```

schedule clause

- Schedule clause (decide how the iterations are executed in parallel):

schedule (static | dynamic | guided [, chunk])



omp sections clause

```
#pragma omp parallel default(none) \
    shared(n,a,b,c,d) private(i)
{
    #pragma omp sections nowait
    {
        #pragma omp section
        {
            for (i=0; i<n-1; i++)
                b[i] = (a[i] + a[i+1])/2;
        }

        #pragma omp section
        {
            for (i=0; i<n; i++)
                d[i] = 1.0/c[i];
        }
    } /*-- End of sections --*/
} /*-- End of parallel region --*/
```

Synchronization: barrier

```
For(I=0; I<N; I++)  
    a[I] = b[I] + c[I];
```

```
For(I=0; I<N; I++)  
    d[I] = a[I] + b[I]
```

Both loops are in parallel region
With no synchronization in between.
What is the problem?

Fix:

```
For(I=0; I<N; I++)  
    a[I] = b[I] + c[I];
```

```
#pragma omp barrier
```

```
For(I=0; I<N; I++)  
    d[I] = a[I] + b[I]
```

critical section

```
For(I=0; I<N; I++) {  
    .....  
    sum += A[I];  
    .....  
}
```

Cannot be parallelized if sum is shared.

Fix:

```
For(I=0; I<N; I++) {  
    .....  
    #pragma omp critical  
    {  
        sum += A[I];  
    }  
    .....  
}
```

OpenMP environment variables

- **OMP_NUM_THREADS**

```
setenv OMP_NUM_THREADS 8
```

- **OMP_SCHEDULE**

```
setenv OMP_SCHEDULE "guided, 4"
```

```
setenv OMP_SCHEDULE "dynamic"
```

And many more.

Sequential Matrix Multiply

```
for (I=0; I<n; I++)
    for (j=0; j<n; j++)
        c[I][j] = 0;
    for (k=0; k<n; k++)
        c[I][j] = c[I][j] + a[I][k] * b[k][j];
```

OpenMP Matrix Multiply

```
#pragma omp parallel for private(j, k)
for (I=0; I<n; I++)
    for (j=0; j<n; j++)
        c[I][j] = 0;
    for (k=0; k<n; k++)
        c[I][j] = c[I][j] + a[I][k] * b[k][j];
```



OpenMP

Refer:

<https://computing.llnl.gov/tutorials/openMP/>



Systems Architecture for Data Analytics – Introduction to Systems Architecture

Courtesy: Prof Sundar B slides

Flynn's Taxonomy of classifying computers

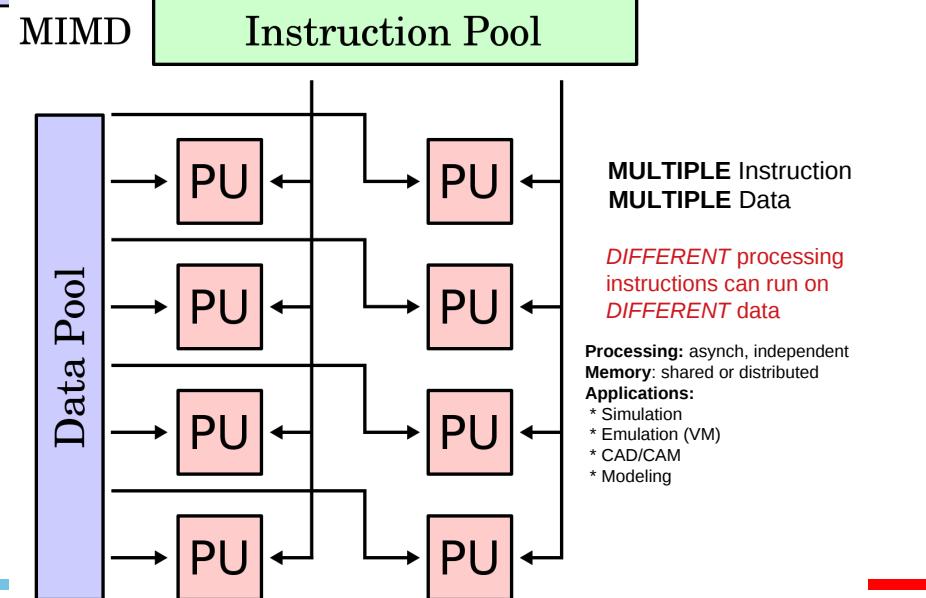
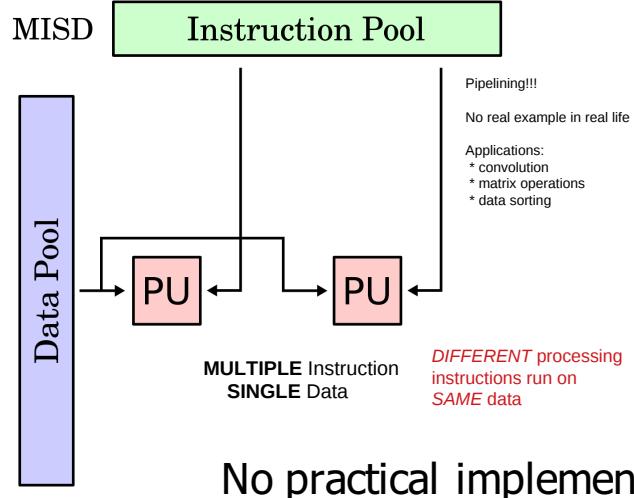
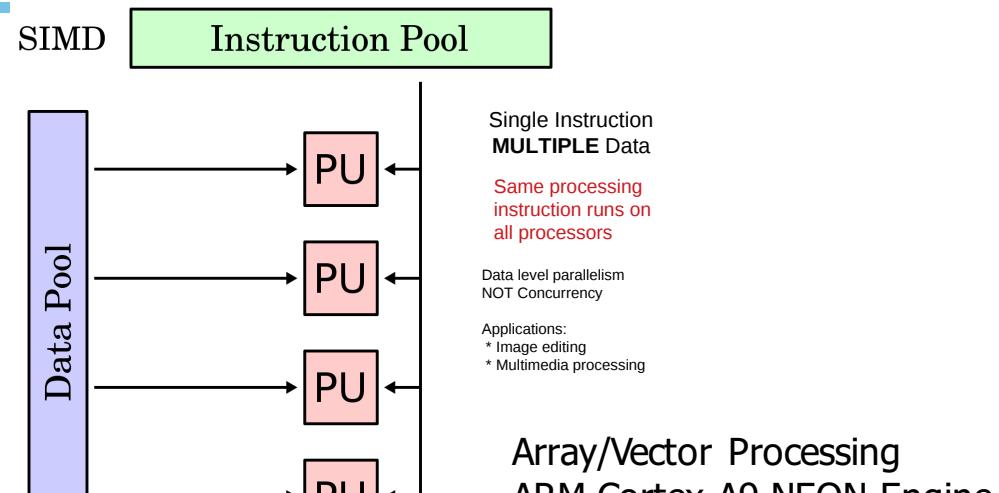
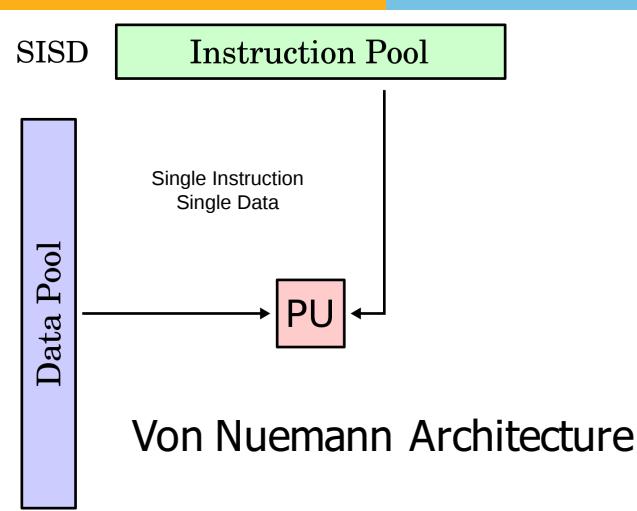
- Michael J Flynn classified computers on the basis of multiplicity of instruction stream and data streams in a computer system in 1966
- how sequence of instructions or data will be executed upon a single processor
- Instruction stream is the sequence of instructions as executed by the machine
- Data Stream is a sequence of data including input, or partial or temporary result, called by the instruction Stream.

Flynn's Taxonomy[2]

Flynn's taxonomy

	Single Instruction	Multiple Instruction
Single Data	<u>SISD</u>	<u>MISD</u>
Multiple Data	<u>SIMD</u>	<u>MIMD</u>

Flynn's Taxonomy[3]



Parallel Programming

SPMD

- Single program Multiple Data
- Multiple instances working on different data
- Typical in parallel programming

MPMD

- Multiple program Multiple Data
- Two or more programs
 - Master/controller
- Typical distributed scenario

Parallel Processing Models

- Data Parallelism
- Task Parallelism
- Request Level Parallelism

Parallel Processing Models

- ## Data Parallelism

- scale the throughput of processing based on the ability to decompose the data set into concurrent processing streams, all performing the same set of operations
- Data that can be operated in parallel
- Data in multiple disks that can be operated in parallel
- SPMD

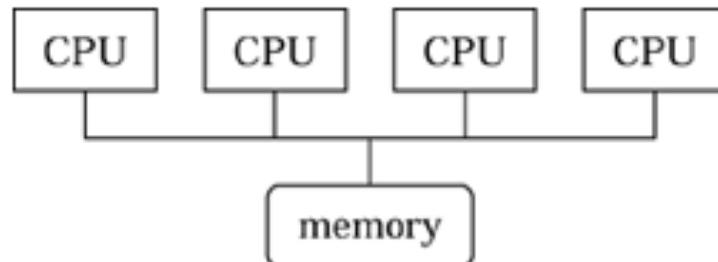
Parallel Processing Models

- Task Parallelism
 - Distributing tasks across different processors
 - MPMD

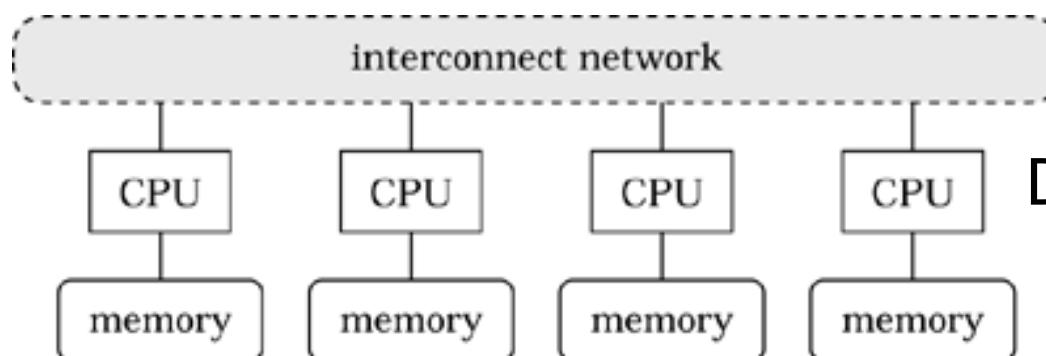
Parallel Processing Models

- Request Level Parallelism
 - Job requests are sent to parallel nodes for execution
 - Cloud/Services

Shared vs Distributed



Shared Memory



Distributed Memory
Message Passing



Thank you !



BITS Pilani
Pilani Campus



DSECL ZG517 - Systems for Data Analytics

Session #5 – Introduction to Systems Architecture – Parallel Architectures and Models

Murali P
muralip@wilp.bits-pilani.ac.in
[Saturday – 04:15 PM]

Agenda

- Introduction and Motivation
- Parallel Architectures and Programming Models
- Parallel Processing Models



Webinar Key points: Review Locality of reference

In C

```
#include <time.h>  
.....  
clock_t t;  
t = clock();  
function_to_time();  
t = clock() - t;  
double time_taken =  
((double)t)/CLOCKS_PER_SEC; // in seconds  
.....
```

Locality of Reference

Try matrix addition code.

Put add code in a function `add(int m1[N][N], int m2[N][N],
int N)`

//using pointers then, `add(int** m1, int** m2, int N)`

Write one add with row-major, other with column-major

Compare the two add methods

Quick sort

Right quick sort.

Partition method (Hoare's)

Partition method (Lomuto's)



BITS Pilani
Pilani Campus



OpenMP

openMP – hello world

```
#include <stdio.h>
#include <omp.h>                                //serial
int main()                                         printf("\n Bye! \n");
{                                                 return 0;
    //serial                                         }
    printf("\n Welcome! \n");
#pragma omp parallel
{ // parallel thread
    printf("Hello world(thread %d)\n",
           omp_get_thread_num());
}
```

```
shell$ export OMP_NUM_THREADS=4
```

Compile:

```
$ gcc -fopenmp -lomp -o hello hell_omp.c
```

```
$./hello
```



BITS Pilani
Pilani Campus



Open MPI

Simple send receive

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
// Initialize the MPI environment
MPI_Init(NULL, NULL);
// Find out rank, size
int world_rank;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
// We are assuming at least 2 processes for this task
if(world_size< 2)
{
fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
MPI_Abort(MPI_COMM_WORLD, 1);
}
```

Send/receive – contd

```

int number;
if(world_rank == 0) {
// If we are rank 0, set the number to -1 and
// send it to process 1
number = -1;
MPI_Send(
/* data = */ &number,
/* count = */ 1,
/* datatype = */ MPI_INT,
/* destination = */ 1,
/* tag = */ 0,
/* communicator = */ MPI_COMM_WORLD);
}

else if(world_rank == 1) {
MPI_Recv(
/* data = */ &number,
/* count = */ 1,
/* datatype = */ MPI_INT,
/* source = */ 0,
/* tag = */ 0,
/* communicator = */ MPI_COMM_WORLD,
/* status = */ MPI_STATUS_IGNORE);
printf("Process 1 received number %d from
process 0\n", number);
}

// close everything
MPI_Finalize();
}

```

compile

```
mpicc -o hello hello.c
```

Ensure file hello is in every machine

```
mpirun -np 4 ./hello
```



BITS Pilani
Pilani Campus

Systems Architecture for Data Analytics

2nd logical component of the course



Systems Architecture for Data Analytics – Introduction to Systems Architecture

Courtesy: Prof Sundar B slides

Motivation

Why Parallel?

Parallel Systems for Data Analytics

Why?

Flynn's Taxonomy of classifying computers

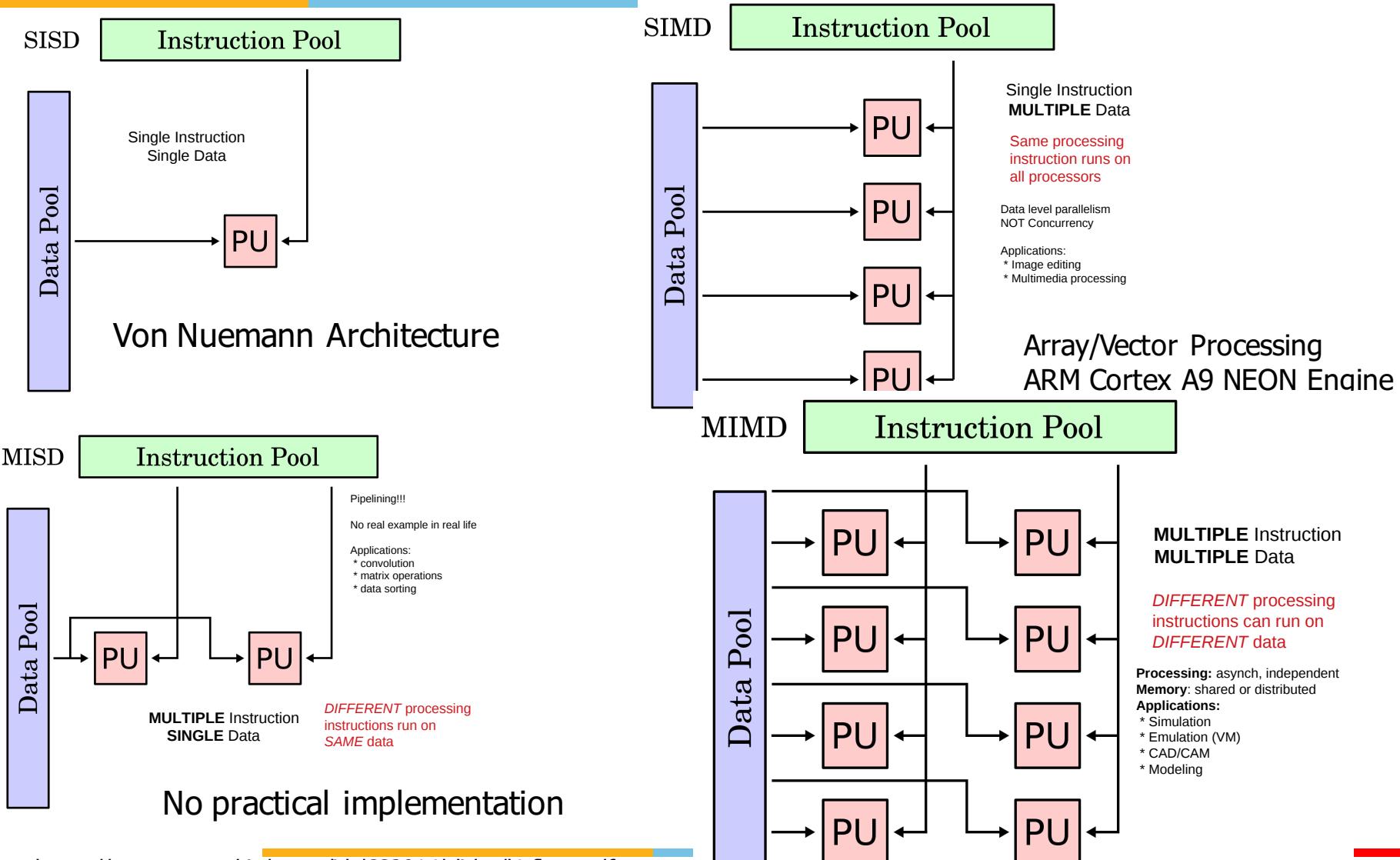
- Michael J Flynn classified computers on the basis of multiplicity of instruction stream and data streams in a computer system in 1966
- how sequence of instructions or data will be executed upon a single processor
- Instruction stream is the sequence of instructions as executed by the machine
- Data Stream is a sequence of data including input, or partial or temporary result, called by the instruction Stream.

Flynn's Taxonomy[2]

Flynn's taxonomy

	Single Instruction	Multiple Instruction
Single Data	<u>SISD</u>	<u>MISD</u>
Multiple Data	<u>SIMD</u>	<u>MIMD</u>

Flynn's Taxonomy[3]



Parallel Programming

SPMD

- Single program Multiple Data
- Multiple instances working on different data
- Typical in parallel programming

MPMD

- Multiple program Multiple Data
- Two or more programs
 - Master/controller
- Typical distributed scenario

Parallel Processing Models

- Bit-level parallelism
- Instruction level parallelism
- Data Parallelism
- Task Parallelism
- Request Level Parallelism

Parallel Processing Models

- **Data Parallelism**
 - scale the throughput of processing based on the ability to decompose the data set into concurrent processing streams, all performing the same set of operations
 - Data that can be operated in parallel
 - Data in multiple disks that can be operated in parallel
 - SPMD

Parallel Processing Models

- Task Parallelism
 - Distributing tasks across different processors
 - MPMD

Parallel Processing Models

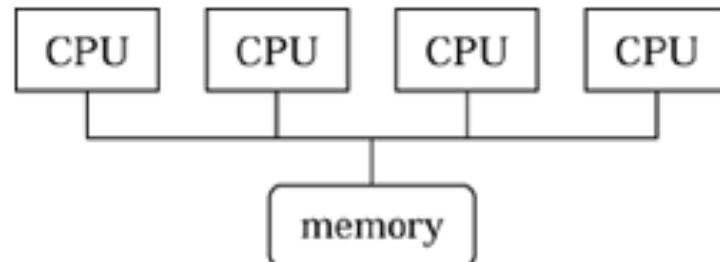
- Request Level Parallelism
 - Job requests are sent to parallel nodes for execution
 - Cloud/Services

Distributed Computing

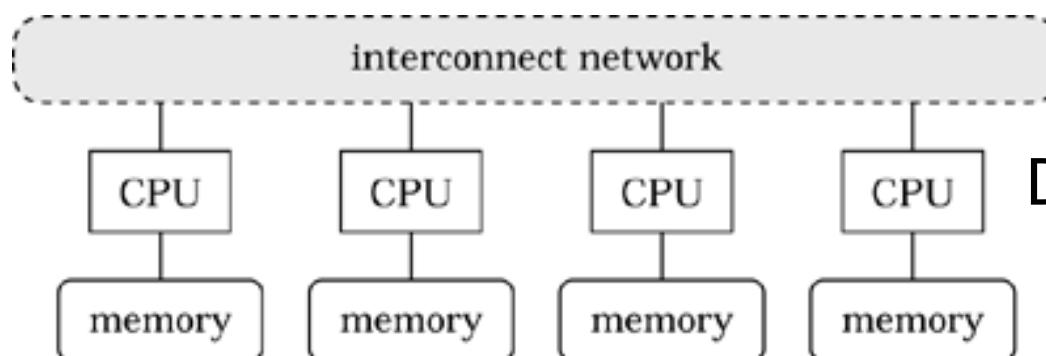
Client/Server Model

Peer-to-peer model

Shared vs Distributed



Shared Memory



Distributed Memory
Message Passing



Thank you !



BITS Pilani
Pilani Campus



DSECL ZG517 - Systems for Data Analytics

Session #5 – Introduction to Systems Architecture –
Client-server, P2P systems; Cluster Computing

Murali P
muralip@wilp.bits-pilani.ac.in
[Saturday – 04:15 PM]

Agenda

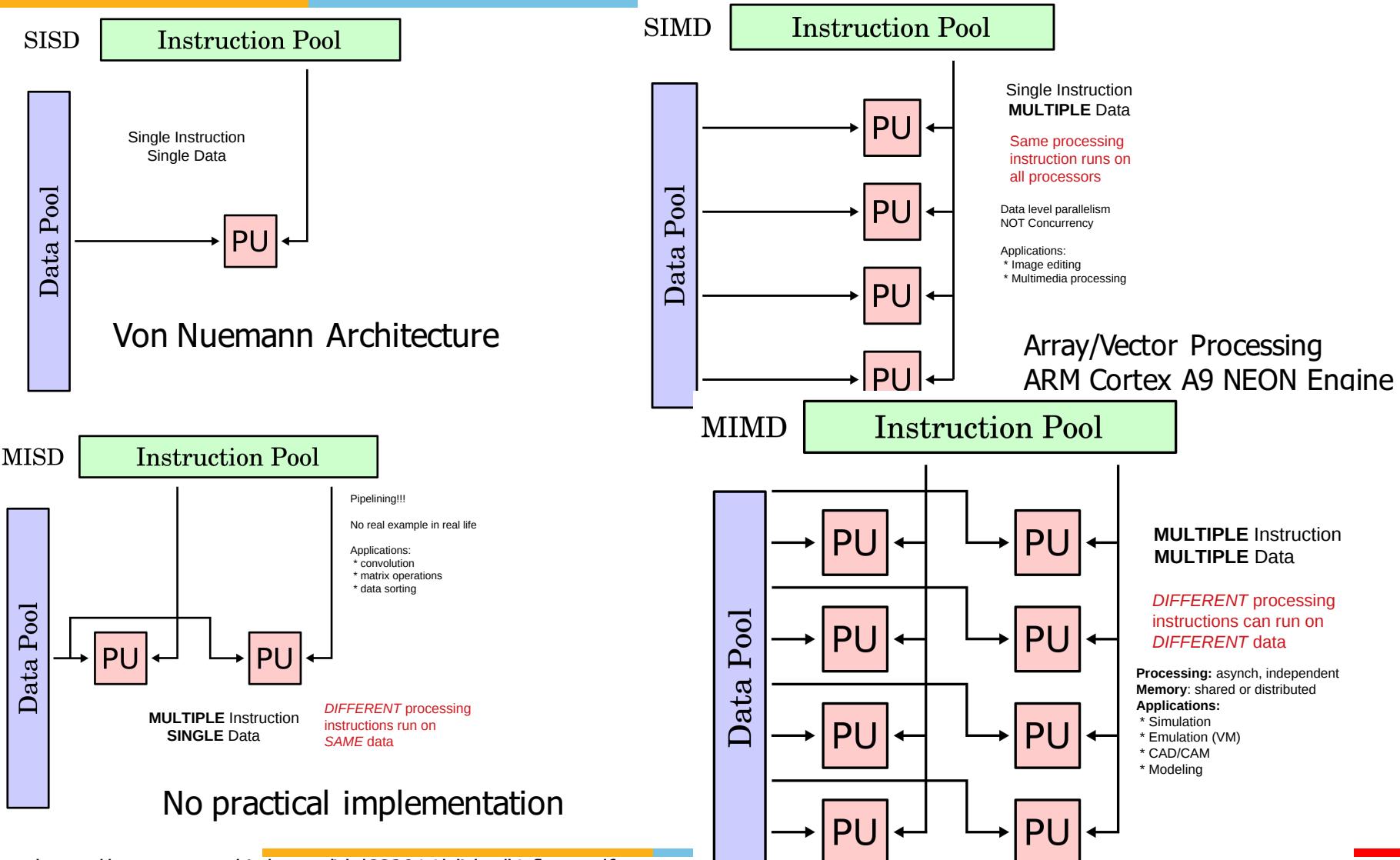
- Client Server systems
- P2P computing
- Distributed Systems
- Cluster Computing



Review: Systems Architecture for Data Analytics – Introduction to Systems Architecture

Courtesy: Prof Sundar B slides

Flynn's Taxonomy[3]



Parallel Programming

SPMD

- Single program Multiple Data
- Multiple instances working on different data
- Typical in parallel programming

MPMD

- Multiple program Multiple Data
- Two or more programs
 - Master/controller
- Typical distributed scenario

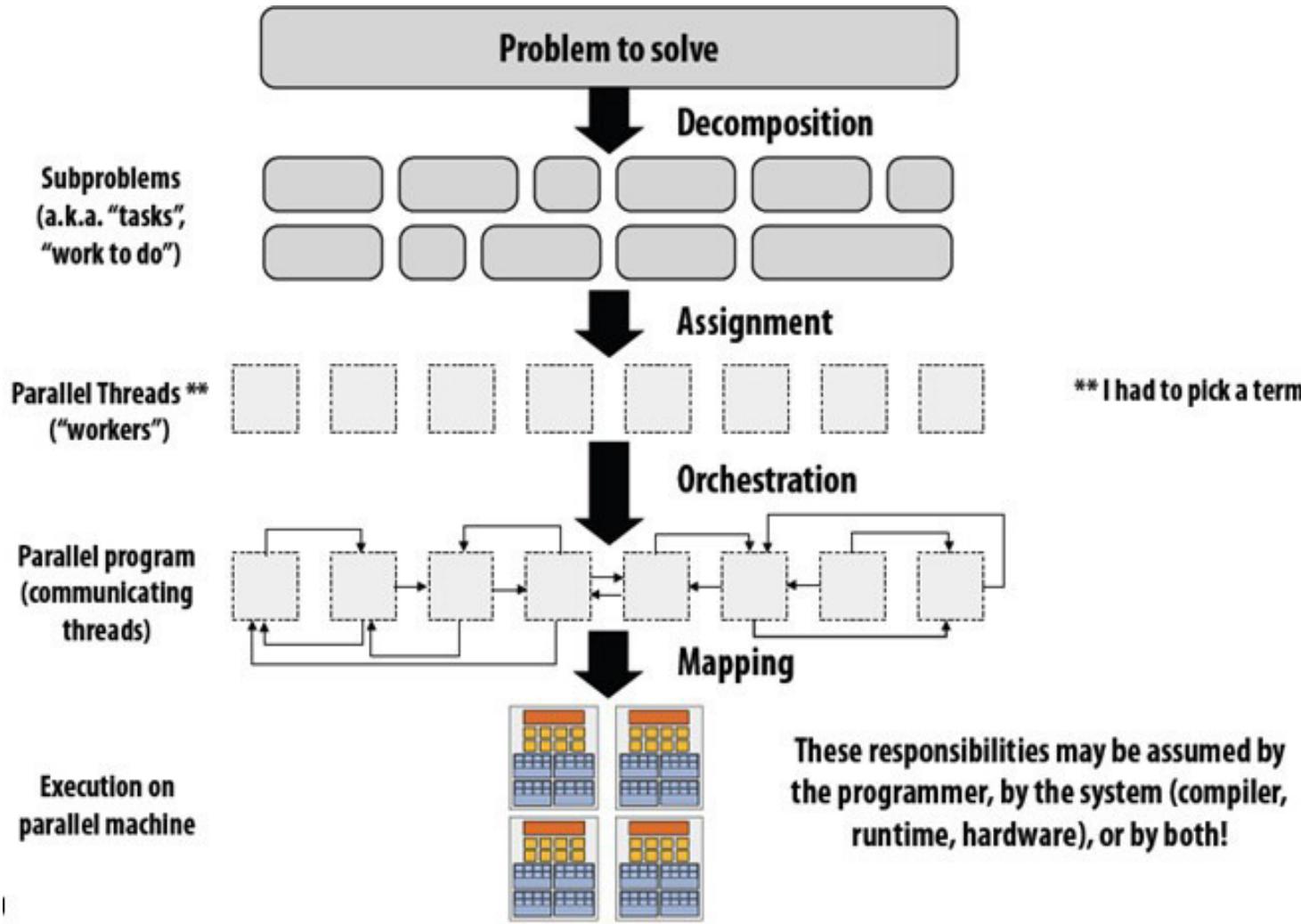
Parallel Processing Models

- Bit-level parallelism
- Instruction level parallelism
- Data Parallelism
- Task Parallelism
- Request Level Parallelism



Client/Server P2P Distributed Systems

Creating Parallel programs



Only for published slide

Creating a Parallel Program – Problem Decomposition

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

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

Only for published slide

Creating a Parallel Program – Problem Decomposition

- Who is responsible for performing problem decomposition?
 - In most cases: the programmer
- Automatic decomposition of sequential programs continues to be a challenging research problem (very difficult in general case)
- Compiler must analyze program, identify dependencies
 - What if dependencies are data dependent (not known at compile time)?
 - Researchers have had modest success with simple loop nests
 - The “magic parallelizing compiler” for complex, general-purpose code has not yet been achieved

Only for published slide

Creating a Parallel Program – Assignment

- Assigning tasks to threads
 - Think of “tasks” as things to do
 - Think of threads as “workers”
- Goals: balance workload, reduce communication costs
- Although programmer is often responsible for decomposition, many languages/runtimes take responsibility for assignment.
- Assignment be performed statically, or dynamically during execution

Only for published slide

Creating a Parallel Program – Assignment

- Assigning tasks to threads
 - Think of “tasks” as things to do
 - Think of threads as “workers”
- Goals: balance workload, reduce communication costs
- Although programmer is often responsible for decomposition, many languages/runtimes take responsibility for assignment.
- Assignment be performed statically, or dynamically during execution

Only for published slide

Creating a Parallel Program – Orchestration

- **Involves:**
 - Communicating between workers
 - Adding synchronization to preserve dependencies if necessary
 - Organizing data structures in memory
 - Scheduling tasks
- **Goals: reduce costs of communication/sync, preserve locality of data reference, reduce overhead, etc.**
- **Machine details impact many of these decisions**
 - If synchronization is expensive, might use it more sparsely

Only for published slide

Creating a Parallel Program – Mapping

- Mapping “threads” (“workers”) to hardware execution units
- Example 1: mapping by the operating system
 - e.g., map thread to hardware execution context on a CPU core
- Example 2: mapping by the compiler
 - Exploiting data parallelism
- Example 3: mapping by the hardware
 - Map CUDA thread blocks to GPU cores
- Some interesting mapping decisions:
 - Place related threads (cooperating threads) on the same processor (maximize locality, data sharing, minimize costs of comm/sync)
 - Place unrelated threads on the same processor (one might be bandwidth limited and another might be compute intensive) to use machine more efficiently

Only for published slide

Creating a Parallel Program – Mapping

- Mapping “threads” (“workers”) to hardware execution units
- Example 1: mapping by the operating system
 - e.g., map thread to hardware execution context on a CPU core
- Example 2: mapping by the compiler
 - Exploiting data parallelism
- Example 3: mapping by the hardware
 - Map CUDA thread blocks to GPU cores
- Some interesting mapping decisions:
 - Place related threads (cooperating threads) on the same processor (maximize locality, data sharing, minimize costs of comm/sync)
 - Place unrelated threads on the same processor (one might be bandwidth limited and another might be compute intensive) to use machine more efficiently

Factors while designing: Parallelization

- **Fully Automatic**

- The compiler analyzes the source code and identifies opportunities for parallelism.
- The analysis includes identifying inhibitors to parallelism and possibly weigh cost on whether or not the parallelism would actually improve performance.
 - Loops are the most frequent target for automatic parallelization.

- **Programmer Directed**

- Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
- May be able to be used in conjunction with some degree of automatic parallelization also.
- The most common compiler generated parallelization is done using on-node shared memory and threads (such as OpenMP).

Factors while designing:

Understand the Problem and the Program

- If you are starting with a serial program, this necessitates understanding the existing code also.
- Can the program be parallelized?
 - Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.
- Identify the program's **hotspots**:
 - Where is real work done?
- Identify **bottlenecks** in the program
 - What is slowing things down?
- Identify **inhibitors** to parallelism
 - Data dependence
 - Alternative algorithms?

Only for published slide

- Identify the program's **hotspots**:
 - Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
 - Profilers and performance analysis tools can help here
 - Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.
- Identify **bottlenecks** in the program:
 - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
 - May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas
- Identify **inhibitors** to parallelism.
 - One common class of inhibitor is *data dependence*
 - Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.
 - Take advantage of optimized third party parallel software and highly optimized math libraries available from leading vendors (IBM's ESSL, Intel's MKL, AMD's AMCL, etc.).

Factors while designing: Partitioning

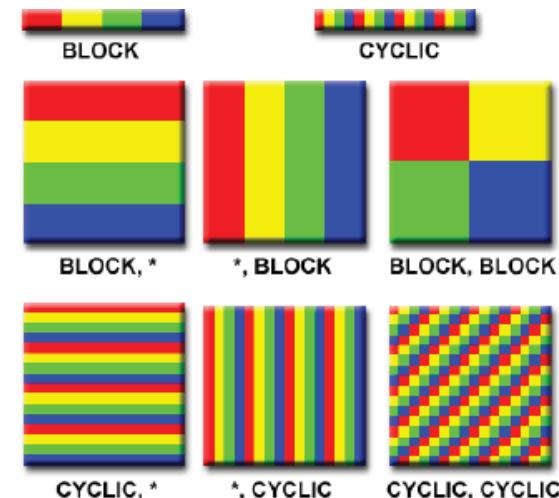
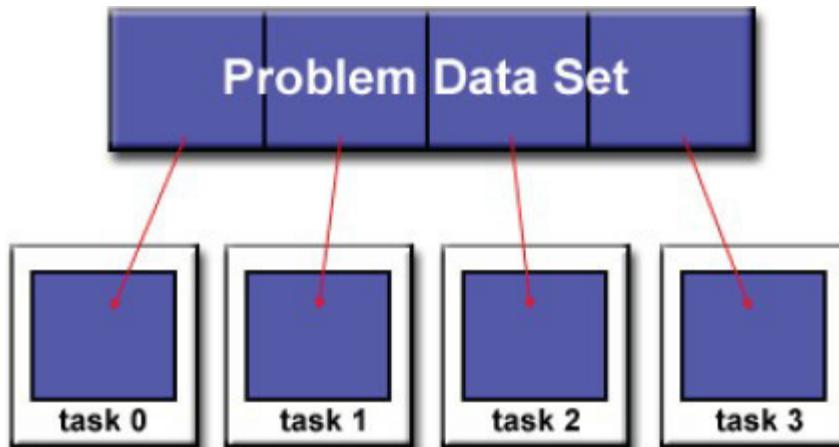
- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks.
- There are two basic ways to partition computational work among parallel tasks:
- ***domain decomposition*** and
- ***functional decomposition*.**

Factors while designing: Partitioning[2]

Domain Decomposition

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

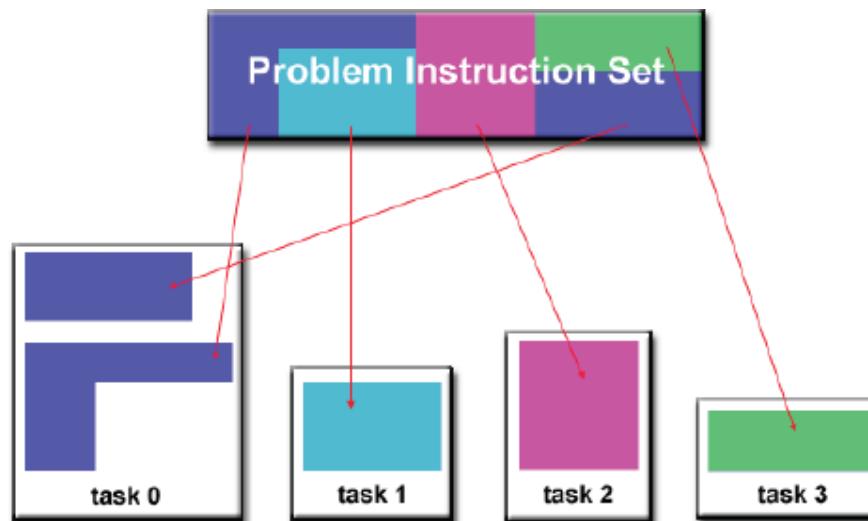
Different ways to partition data



Factors while designing: Partitioning[3]

Functional Decomposition

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



Factors while designing: Communications between tasks/threads

- Embarrassingly parallel
 - No communication required
 - Example: imagine an image processing operation where every pixel in a black and white image needs to have its color reversed
- **Communication overhead**
 - Inter-task communication virtually always implies overhead.
 - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
 - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
 - Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.

Factors while designing: Communications between tasks/threads[2]

- **Latency vs. Bandwidth**
 - ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
 - ***bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.
 - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

Factors while designing:

Communications between tasks/threads[3]

- **Visibility of communications**
 - With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
 - With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures.
- **Synchronous vs. asynchronous communications**
 - Synchronous communications require some type of "handshaking" between tasks that are sharing data.
 - Synchronous communications are often referred to as ***blocking*** communications since other work must wait until the communications have completed.
 - Asynchronous communications allow tasks to transfer data independently from one another.
 - Asynchronous communications are often referred to as ***non-blocking*** communications since other work can be done while the communications are taking place. Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

Factors while designing:

Communications between tasks/threads[3]

- **Scope of communications**
 - Identifying which tasks must communicate with each other is critical during the design stage of a parallel code. The two types described below can be implemented synchronously or asynchronously.
 - ***Point-to-point***
 - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
 - ***Collective***
 - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective.



Distributed Systems

Reference:

Chapters 1 & 2 of DISTRIBUTED SYSTEMS

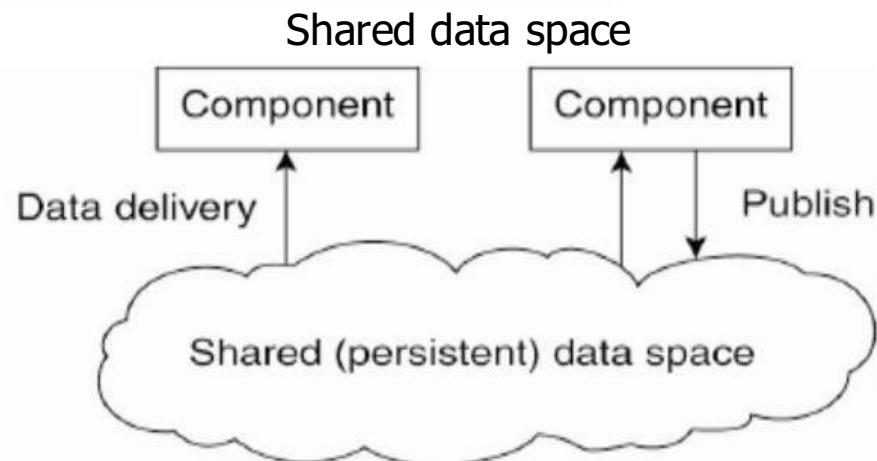
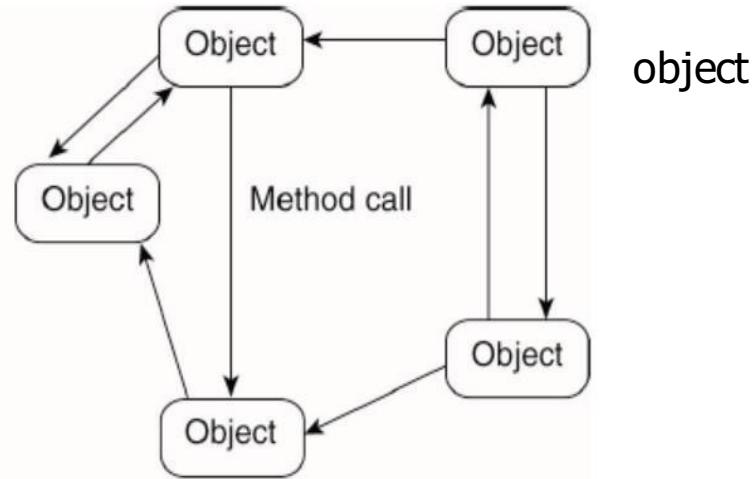
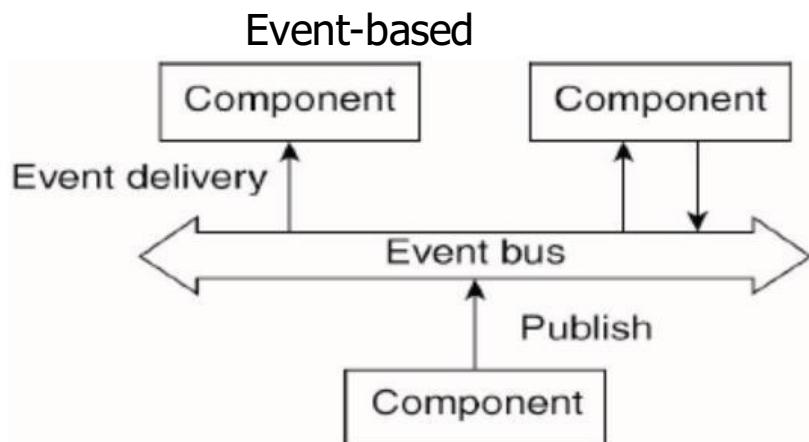
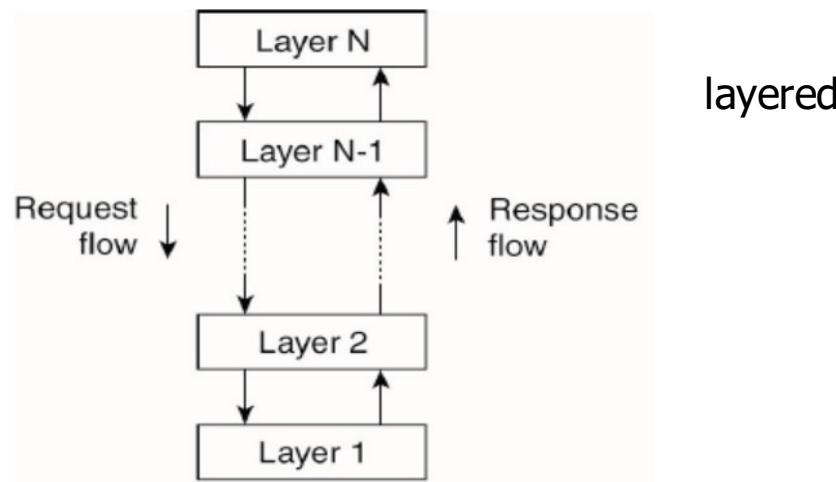
Concepts and Design, George Coulouris, Jean Dollimore, Tim Kindberg,
Gordon Blair (ISBN 13: 978-0-13-214301-1)

Distributed Systems

A Definition

*A distributed system is a collection of entities, each of which is **autonomous, programmable, asynchronous** and **failure-prone**, and which communicate through an **unreliable** communication medium using message passing.*

System Architectures

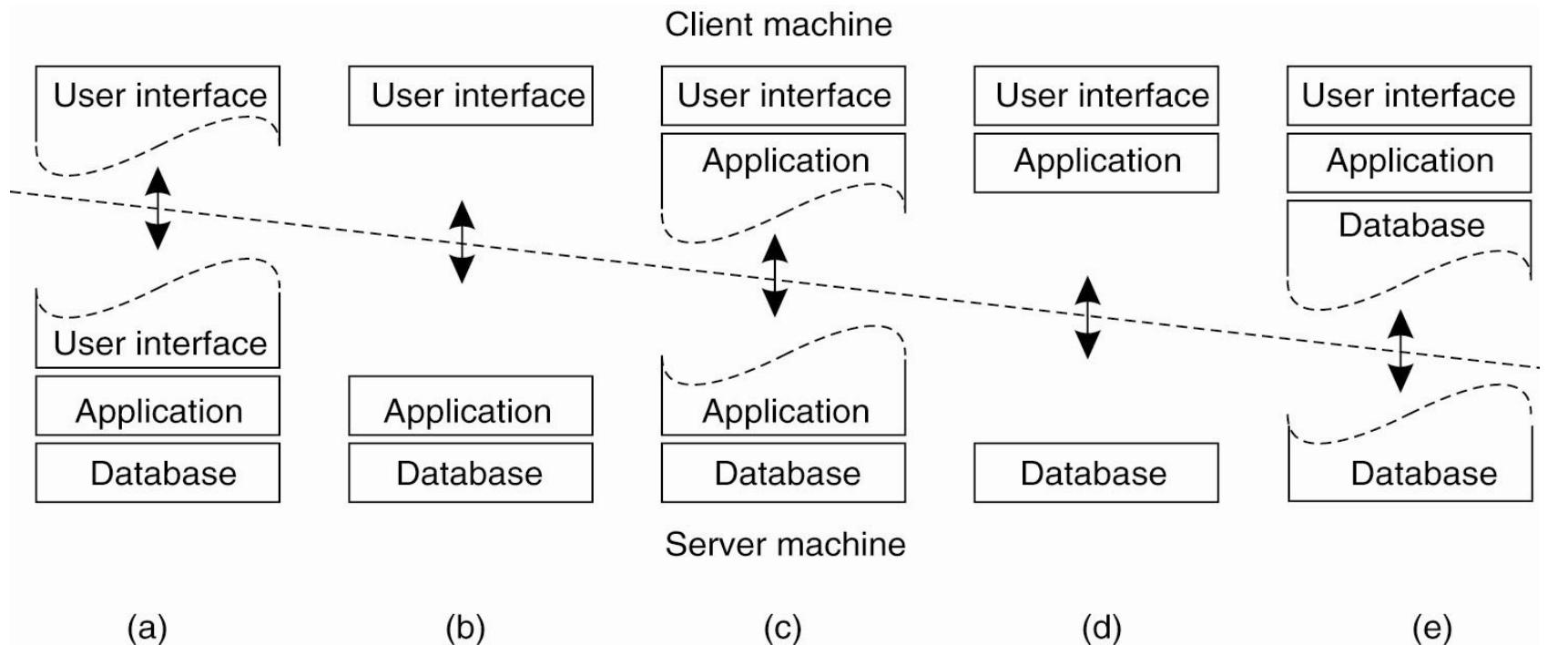


Distributed Computing

Client/Server Model

Peer-to-peer model

Client/Server Architecture



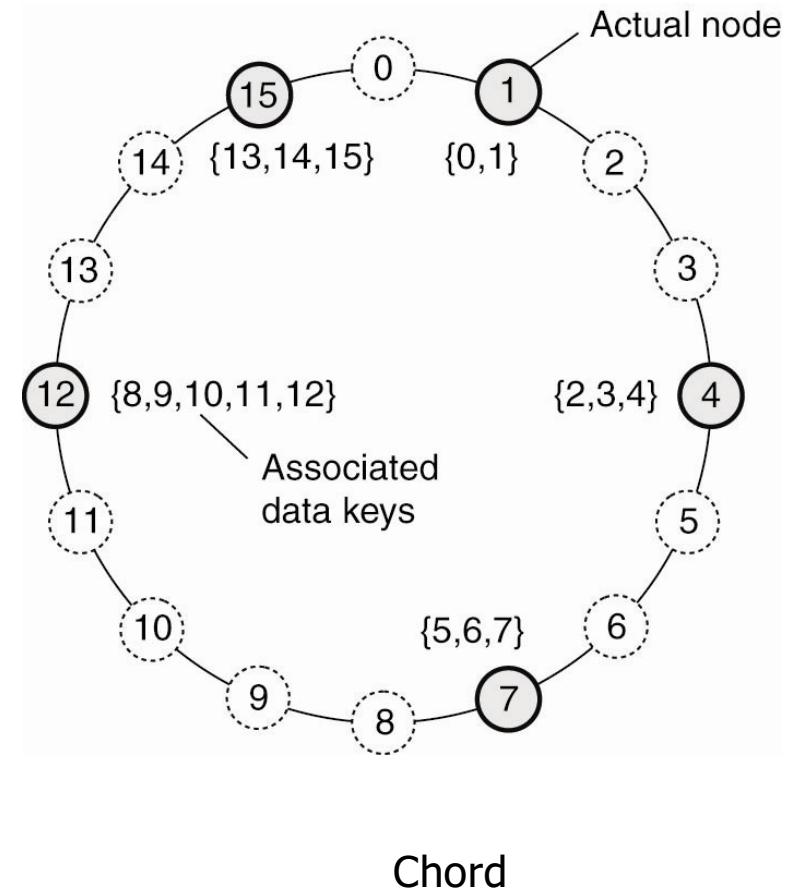
Thin client



Fat client

Structured Peer-to-Peer

- Structured: the overlay network is constructed in a deterministic procedure
 - Most popular: distributed hash table (DHT)
- Key questions
 - How to map data item to nodes
 - How to find the network address of the node responsible for the needed data item
- Two examples
 - Chord and Content Addressable Network (CAN)



Unstructured P2P Architectures

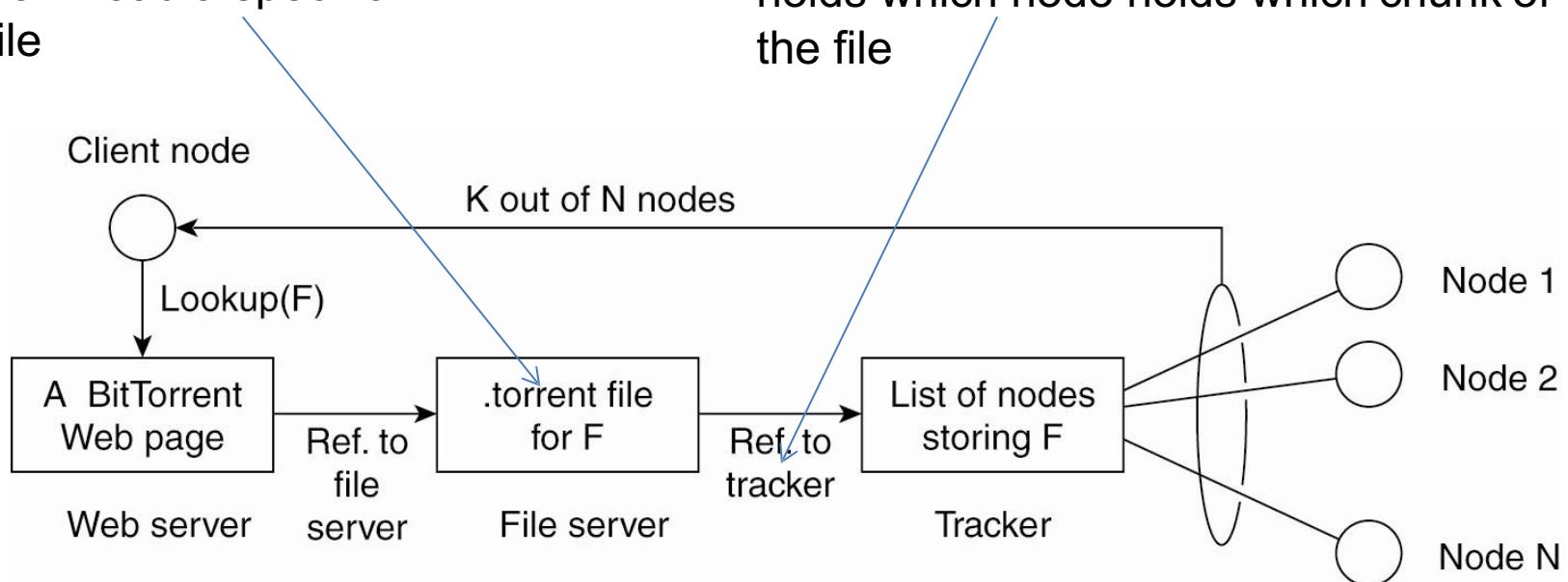
- Largely relying on randomized algorithm to construct the overlay network
 - Each node has a list of neighbors, which is more or less constructed in a random way
- One challenge is how to efficiently locate a needed data item
 - **Flood the network?**
- Many systems try to construct an overlay network that resembles a **random graph**
 - Each node maintains **a partial view**, i.e., a set of live nodes randomly chosen from the current set of nodes

Hybrid Architecture

Collaborative Distributed System - BitTorrent

Information needed to download a specific file

Many trackers, one per file, tracker holds which node holds which chunk of the file



The principal working of BitTorrent (Pouwelse et al. 2004)

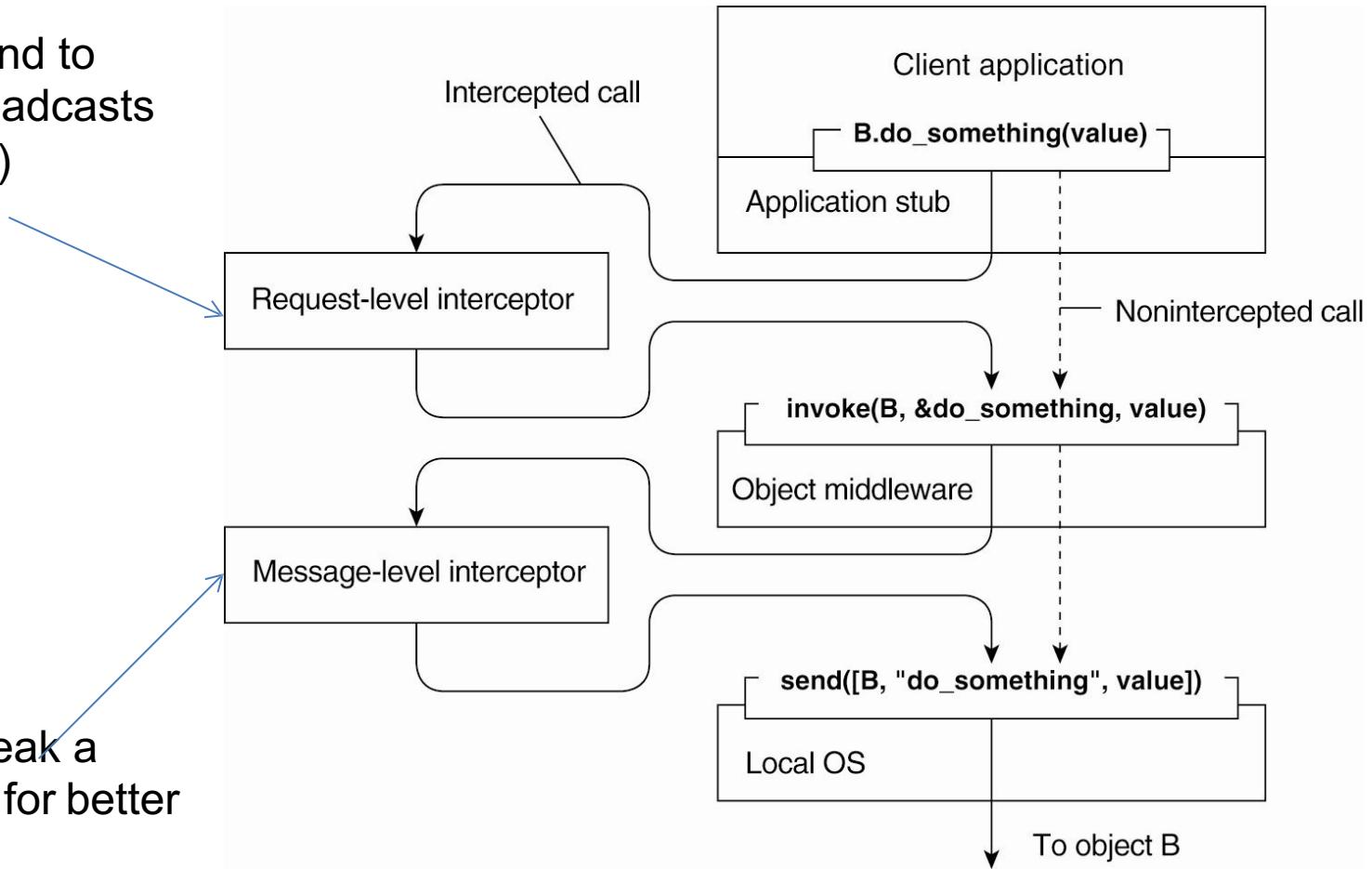
Middleware

- A middleware layer between application and the distributed platforms for **distribution transparency**
 - Role is to make application development easier, by providing common programming abstractions, by masking the heterogeneity and the distribution of the underlying hardware and operating systems, and by hiding low-level programming details.
- Many middleware follows a specific architecture style
 - Object-based style, event-based style
 - Benefits: simpler to design application
 - Limitations: the solution may not be optimal
- Should be **adaptable** to application requirements
 - **Separating policies from mechanism**

Middleware: Interceptor

May want to send to many other Broadcasts (i.e. Replicated)

May want to break a large message for better performance





BITS Pilani
Pilani Campus

Cluster Computing

Src: Prof Sunder B slides

Computer Cluster - Definition

- **Definition [Buyya]:**
 - **A *cluster* is a type of parallel or distributed processing system**
 - **consisting of a collection of inter-connected stand-alone computers**
 - **working together as a single, integrated computing resource.**

Cluster - Objectives

- A computer cluster is typically built for one of the following two reasons:
 - ▣ **High Performance**
 - These are referred to as *compute-clusters*
 - What is the primary motivation to build a compute-cluster (*as opposed to a parallel computer*) for performance?
 - ▣ **High Availability**
 - Availability

Cluster Motivation

- Cost and Incremental Cost Model:
 - Custom Parallel Computer
 - Initial Cost and Obsolescence
 - Scale-out Cluster
 - Incremental performance by additional nodes (i.e. Stand-alone systems) – a loosely coupled model
 - Commodity Cluster
 - A special-case of scale-out cluster, wherein nodes are COTS (i.e. Commercial Off-The-Shelf) computers
 - i.e. Currently whatever is the cheapest

Clusters and Grids

- Clusters and Grids are both
 - ▣ distributed systems built by Connecting stand-alone computers over network
 - ▣ and provide the abstraction of a single computer

	Cluster	Grid
Nodes	Homogeneous	Heterogeneous
Network	Local Area	Wide Area
Node membership	Dedicated / Non-dedicated	Non-dedicated
Resource Abstraction	Process Level	Process Level and Resource Level

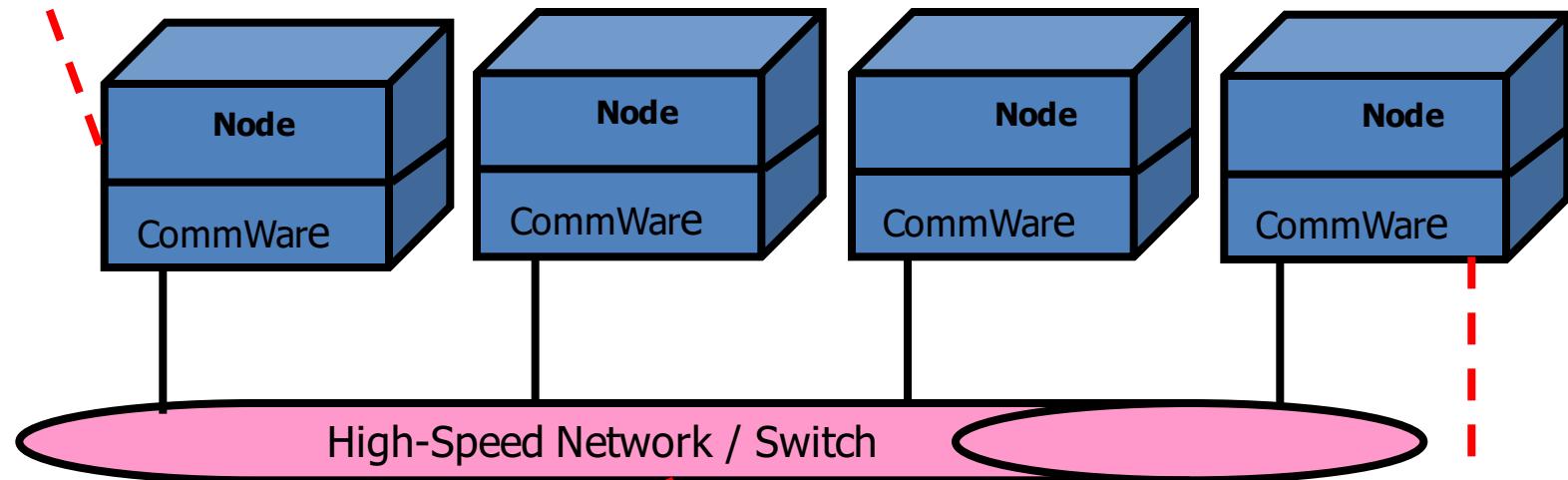
Clusters, Data Centers, and Clouds



- Clouds provide abstract services (including infrastructure) :
 - Services are implemented on large scale computing infrastructure built into central facilities (i.e. Data Centers)
- Typical data center architectures use clusters as the building blocks for:
 - High performance , and more critically, for High Availability

Typical Cluster: Components - Base

- Processor + Memory + Storage
- OS + Run-time environment

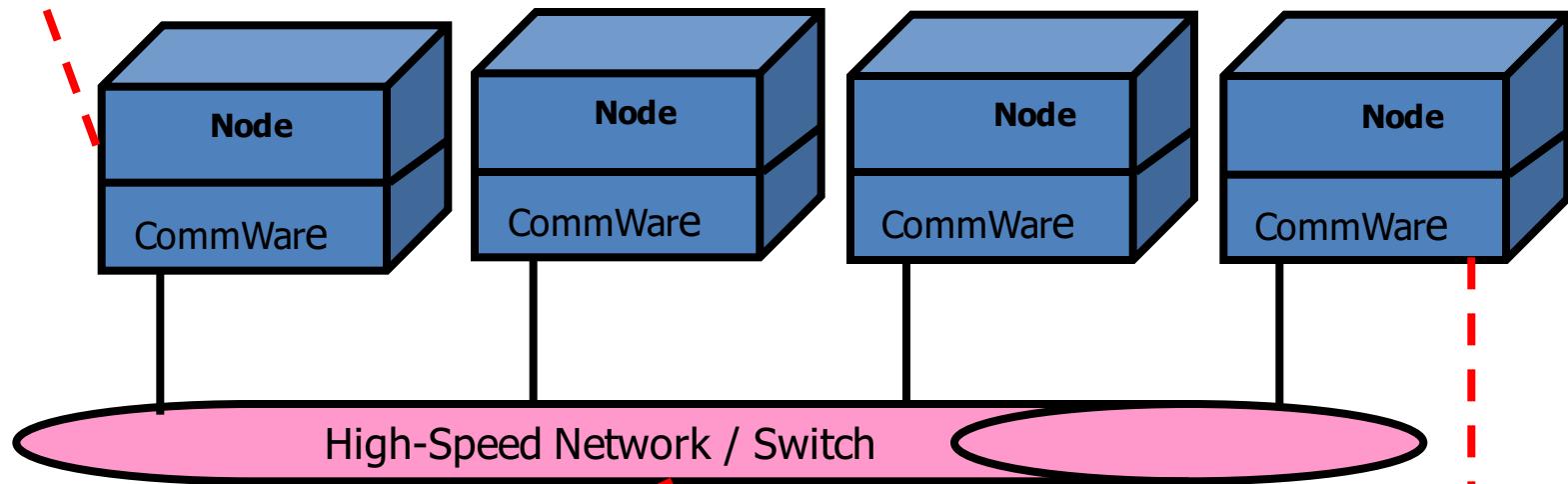


Gigabit Ethernet or
Myrinet/Infiniband

- Network Interface Cards
- Fast Communication Protocols and Services

Typical Cluster: Components - Base

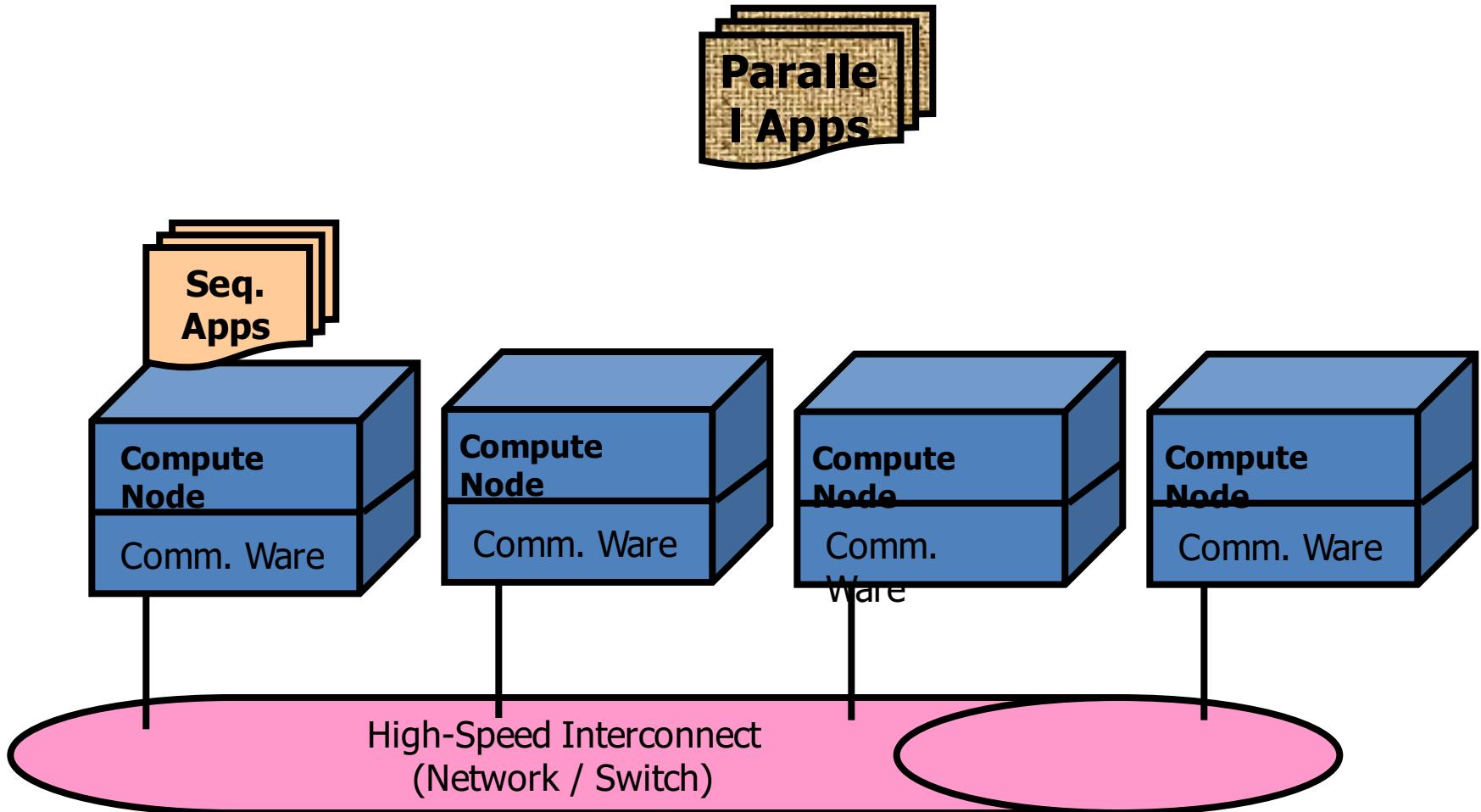
- Processor+Memory+Storage
- OS+Run-time environment



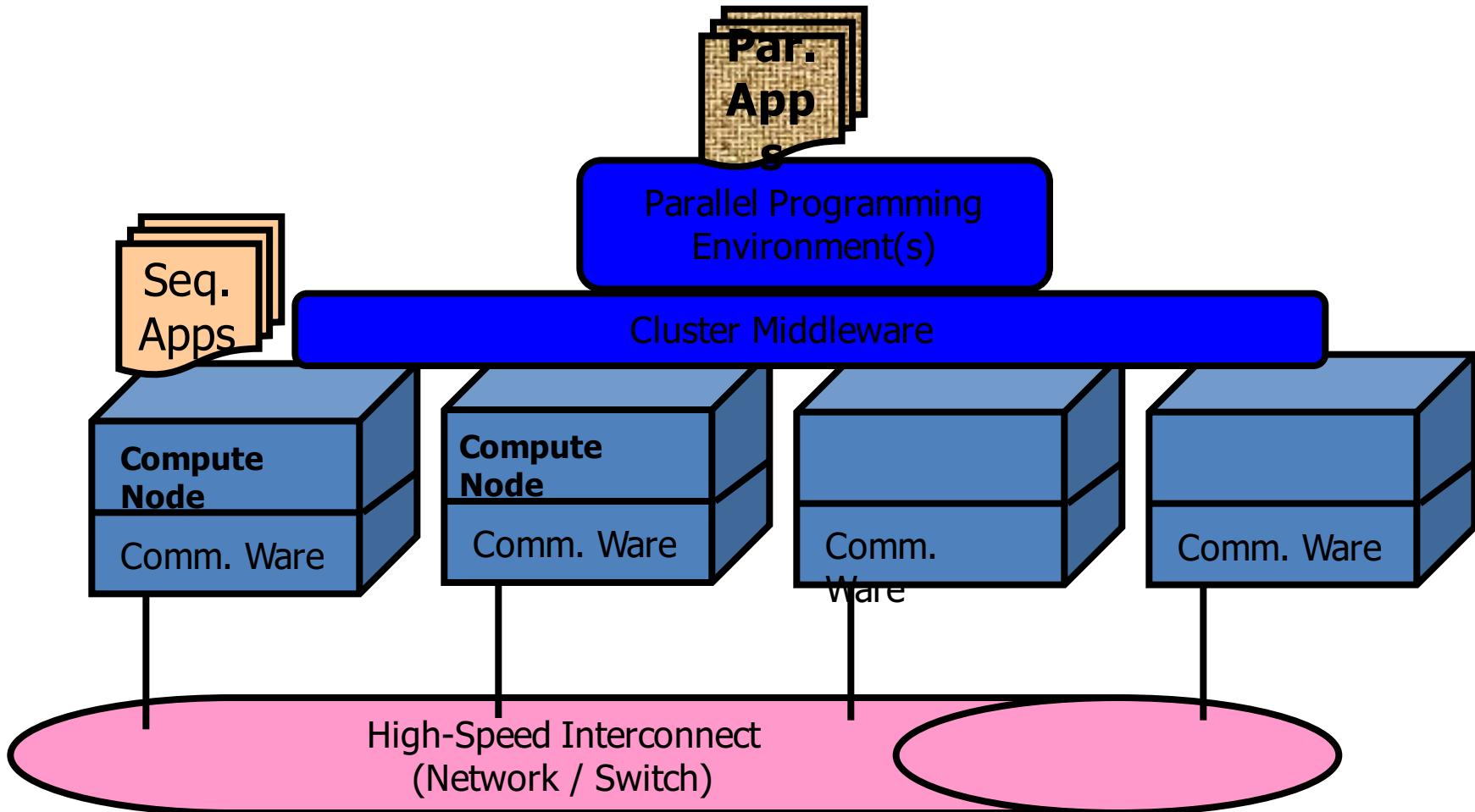
Gigabit Ethernet or
Myrinet/Infiniband

• Network Interface Cards
• Fast Communication Protocols and Services

Typical Cluster: Requirements



Typical Cluster: Architecture



Cluster Middleware - Functions



- There are two main functions:
 - Single System Image (SSI) infrastructure
 - Glues together OSs on all nodes to offer unified access to system resources
 - (High) System Availability (HA) Infrastructure
 - Cluster services for availability
 - Redundancy
 - Fault-tolerance
 - Recovery from failures

Features of Single System Image (SSI)

- Single Point of Control
 - The user submits and manages tasks at a single (logical) point – which of course may be any of the nodes
 - Tracking & Control:
 - Heartbeat messages, status/progress updates of tasks/processes, control messages (e.g. terminate a process)
 - Must be internally distributed for redundancy and load sharing

Features of Single System Image (SSI) [2]

- Single Entry Point
 - Single authentication for all nodes
 - Node failures and load imbalance (when multiple users attempt to log in) must be handled
 - e.g. Run a local DNS mapping hostnames to IP addresses dynamically

Features of Single System Image (SSI) [3]

- Single File System
 - Single File Hierarchy
 - e.g. Network File System on Unix/Linux
 - Visibility of Files
 - Processes on node A should be able to access files (say via, fopen, fread, or fwrite on Unix) on another node

Features of Single System Image (SSI) [4]

- Single I/O space
 - ▣ Any device connected to any node must be accessible from (any process) in any node
 - e.g. Are disks shared?
 - Typically, yes, but not always.

Features of Single System Image (SSI) [5]

- Single Process Space
 - All processes across the cluster have a unique identifier
 - A process on any node can communicate with any other process on any (other) node
 - say, using pipes or signals in Unix

Failure Recovery – Backward Recovery Scheme

- Backward recovery
 - Checkpointing:
 - processes periodically save consistent state information (a.k.a. *checkpoint*) on a stable storage
 - Rollback
 - Post failure,
 - the failed component is isolated,
 - previous checkpoint is restored, and
 - normal operation is resumed
 - Pros and Cons:
 - Easy to implement, application independent
 - Rollback results in wasted execution

Failure Recovery – Forward Recovery Scheme

- Forward recovery is useful in systems
 - where execution time is critical
 - e.g. Real-Time System
 - but may
 - be application specific and
 - require additional hardware
- Forward Recovery
 - No rollback
 - Failure diagnosis is used to reconstruct a valid system state:
 - Assumptions?

Recovery: Checkpointing

- Checkpointing:
 - Periodic saving of state of execution of a program to stable storage
 - so that a system may recover after a failure
 - Saved state is referred to as a *checkpoint*
 - Storage:
 - Default:
 - Hard Disk
 - Alternatives:
 - SSD
 - Node memory??

Checkpointing - Implementation

- Checkpointing can be realized at different levels
 - OS Kernel
 - OS transparently checkpoints and (on failure) restarts processes
 - Library
 - User-space checkpointing program
 - Program has to link to this library
 - explicit calls for checkpointing and restarting
 - link library with source code
 - implicit checkpointing
 - i.e. Compiler inserts checkpointing library calls
 - Application
-

Checkpointing – Overheads

- Checkpointing Overheads:
 - Time Overhead
 - Number of checkpoints * Time taken to store a checkpoint
 - Storage Overhead
 - Number of checkpoints * Space required for a checkpoint

Checkpointing – Reducing Overheads

- Choosing a checkpoint interval
 - Longer interval results in less overhead but a shorter interval results in faster recovery
 - Optimal Checkpoint Interval
 - $\text{sqrt}(\text{MTTF} * T_c) / h$
 - h is average % of normal computation performed in a check-point interval before the system fails
 - T_c is the time consumed to save a checkpoint

Checkpointing – Reducing Overheads [2]

- Incremental Checkpoint
 - Save only the difference since the last checkpoint
 - sequence of checkpoint files must be kept on stable storage
 - Implications:
 - Increased Storage and Increased Recovery Time vs. Reduced Checkpointing Time
- User-Directed Checkpointing
 - User specifies when and what to save.

Checkpointing – Reducing Overheads [3]

- Forked Checkpointing
 - Typical checkpointing schemes are *blocking* schemes
 - Instead,
 - one can copy a program state in memory and
 - another thread can save the state asynchronously
 - e.g. use *fork* in Unix to create a child process that duplicates the address space and checkpoint it.
 - Why does this save time?
 - Optimization:
 - Use Copy-on-Write

Checkpointing – Parallel Programs

- Parallel programs
 - Size:
 - State is much larger (processes and network)
 - Timing and Consistency
 - A *global snapshot* is a set of checkpoints (of different processes).
 - Note that processes may not be synchronous.
 - A snapshot is consistent if
 - a message that is received in a checkpoint of a process has already been sent by another process (as recorded in its checkpoint)

Checkpointing Parallel Programs - Strategies

- Co-ordinated Checkpointing:
 - ▣ a.k.a. Consistent Checkpointing:
 - Program is blocked and all processes checkpoint at the same time.
 - Large overhead and Implementation Difficulties
- Independent Checkpointing:
 - ▣ Each process checkpoints independent of other processes:
 - Small overhead and existing techniques for sequential programs can be used
 - Consistency guarantees cannot be provided

SSI Feature: Process Migration

- Process migration :
 - ▣ **Transparent remote execution**
 - e.g. Sockets are migrated automatically
 - e.g. system calls are intercepted and handled
 - ▣ **Binary compatibility**
 - Migration does not require rewriting or re-linking applications

SSI support:

OS Kernel or Gluing Layer - Example

- e.g. Mosix,
 - A cluster OS for nodes running Unix/Linux kernels
 - Consists of:
 - Algorithms for *Adaptive Resource Sharing*
 - Pre-emptive Process Migration
 - Implemented:
 - at kernel level, as a loadable module
 - without changing the kernel interface



Thank you !



DSECL ZG517 - Systems for Data Analytics

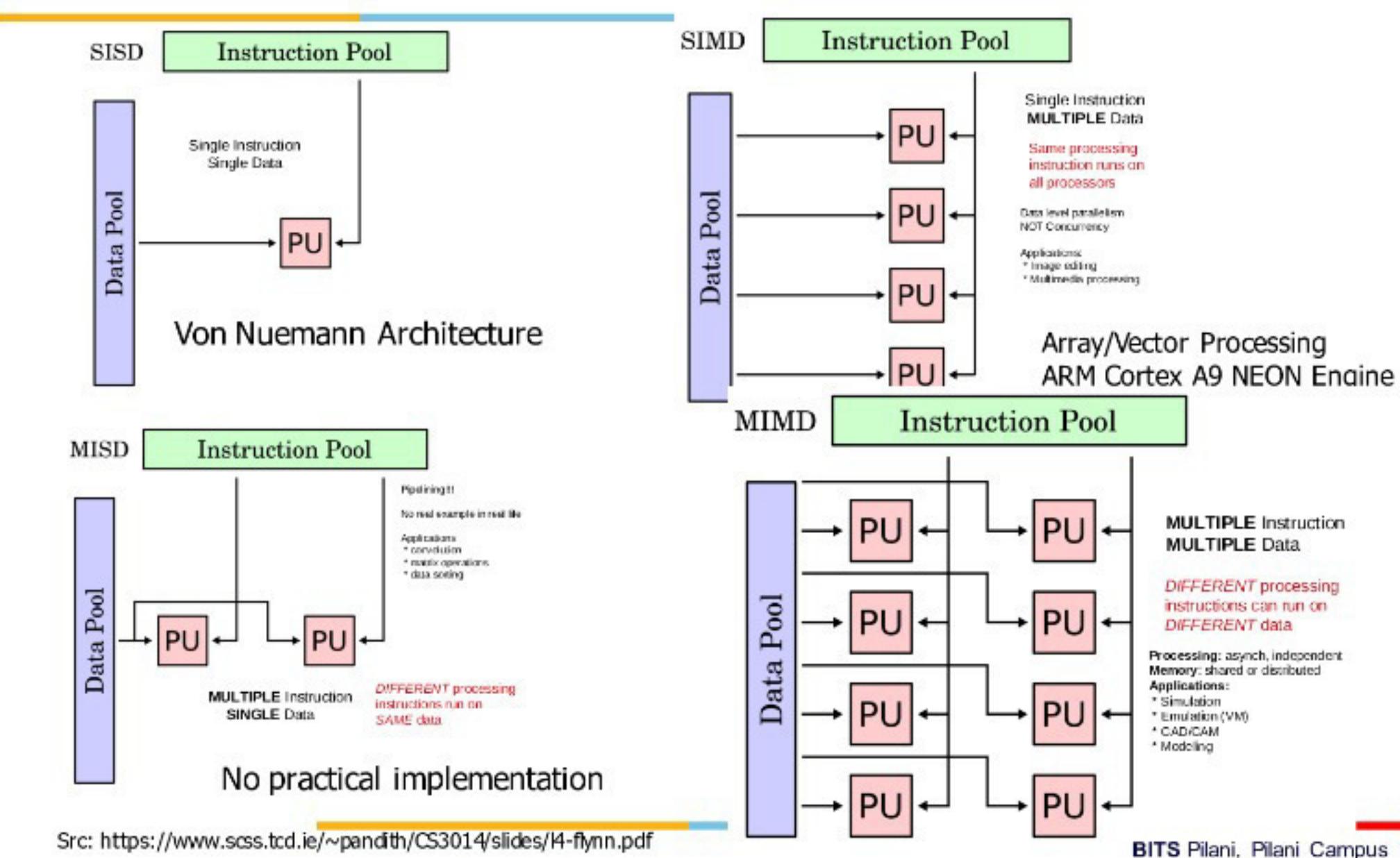
Session #5 – Introduction to Systems Architecture – Parallel Architectures and Models

Murali P
muralip@wila.bits-pilani.ac.in
[Saturday – 04:15 PM]

This presentation uses public contents shared by authors of text books and other relevant web resources. Further, works of Professors from BITS are also used freely in preparing this presentation.

BITS Pilani
Pilani Campus

Flynn's Taxonomy[3]





Parallel Processing Models

- Bit-level parallelism
- Instruction level parallelism
- Data Parallelism
- Task Parallelism
- Request Level Parallelism



Parallel Processing Models

- **Data Parallelism**
 - scale the throughput of processing based on the ability to decompose the data set into concurrent processing streams, all performing the same set of operations
 - Data that can be operated in parallel
 - Data in multiple disks that can be operated in parallel
 - SPMD



Parallel Processing Models

- **Task Parallelism**
 - Distributing tasks across different processors
 - MPMD



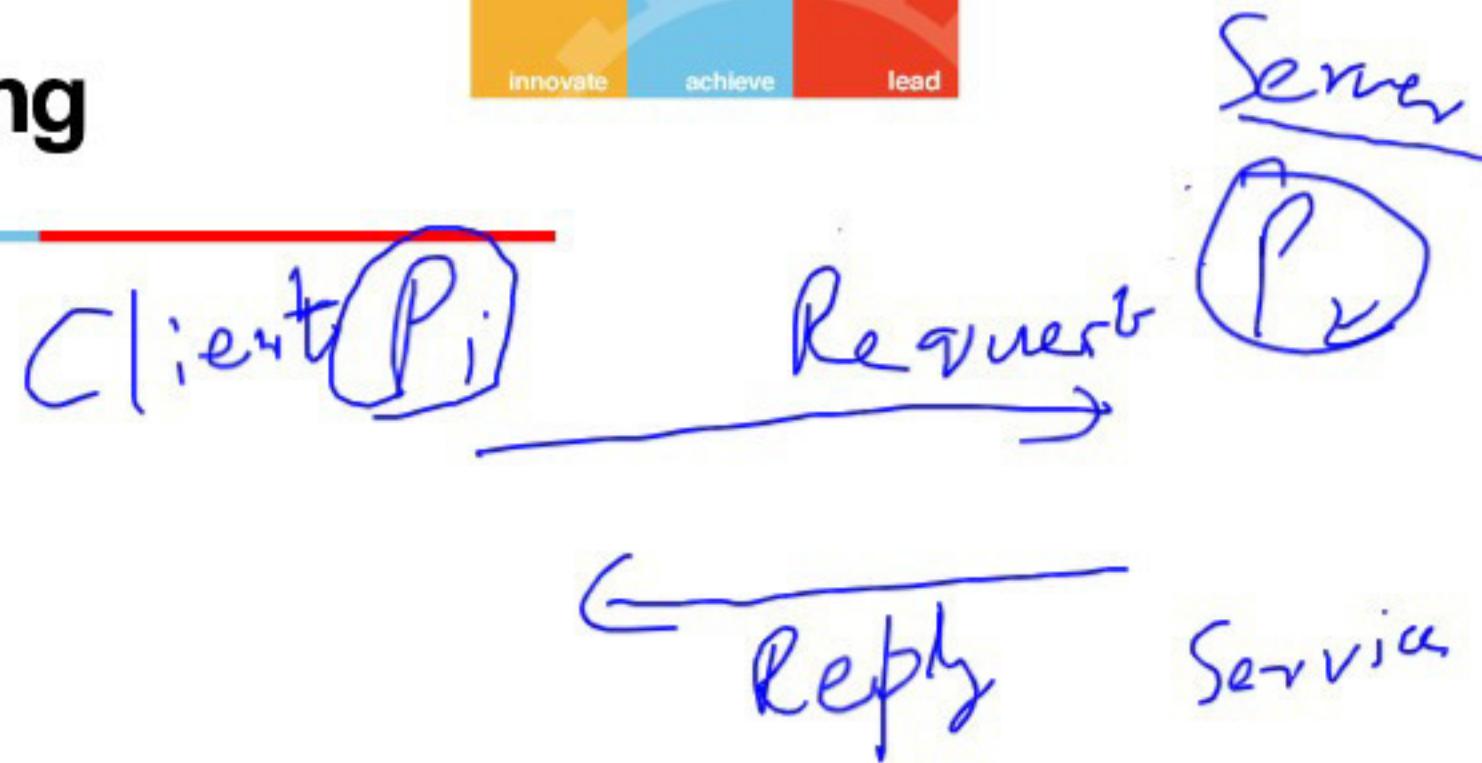
Parallel Processing Models

- **Request Level Parallelism**
 - Job requests are sent to parallel nodes for execution
 - Cloud/Services

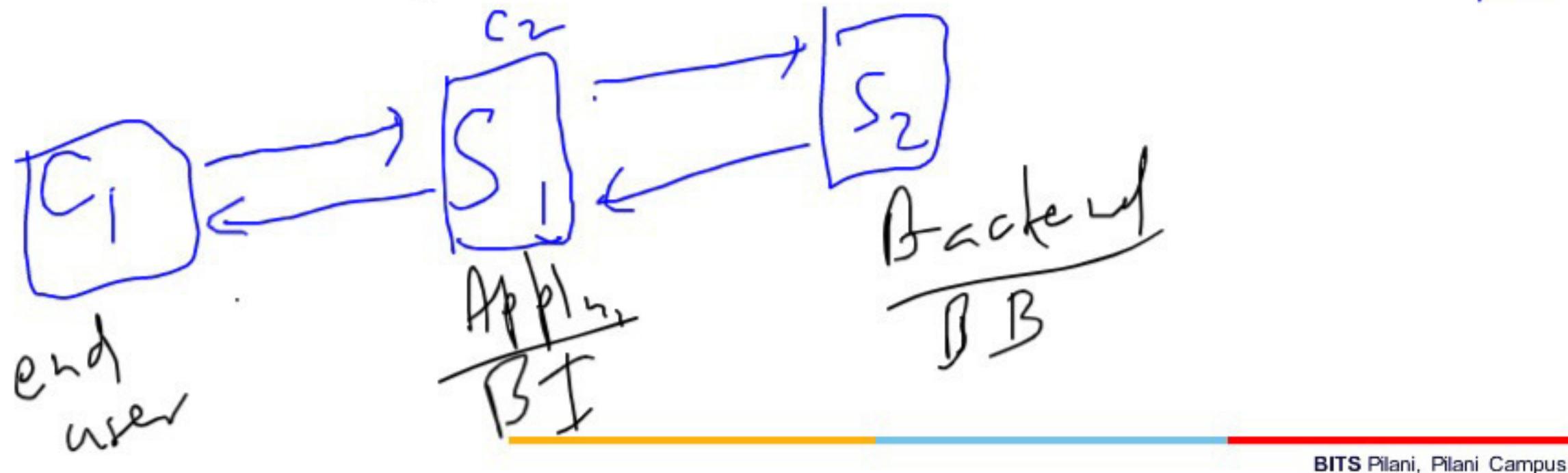
Distributed Computing

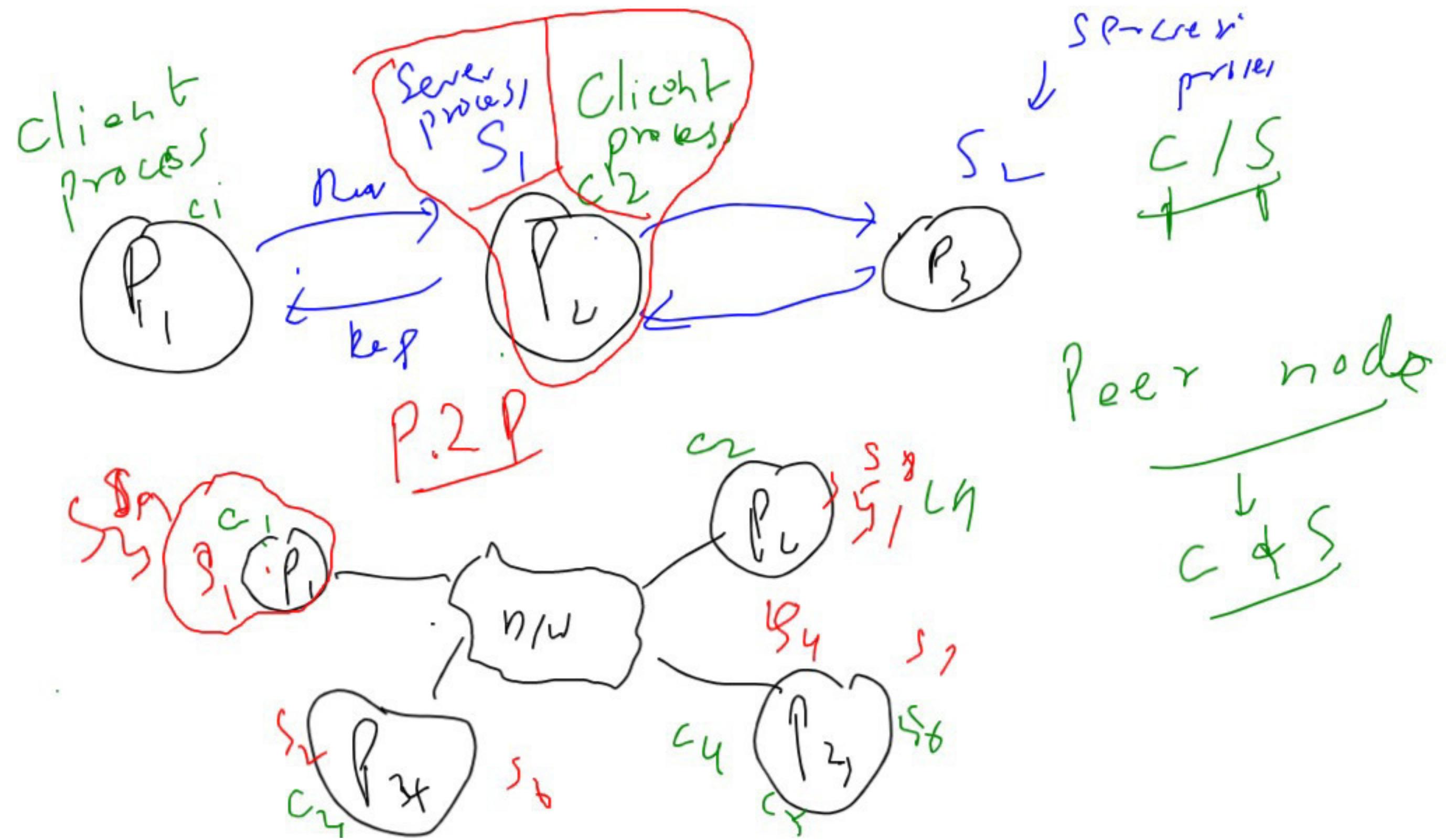


Client/Server Model

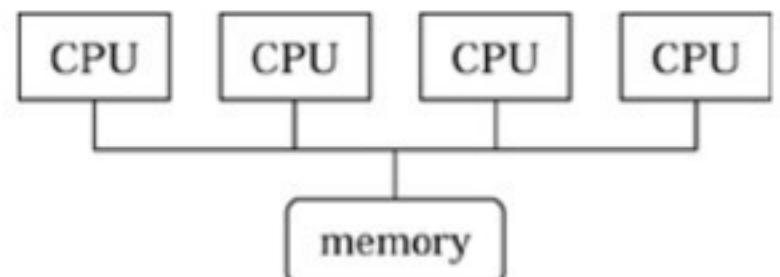


Peer-to-peer model

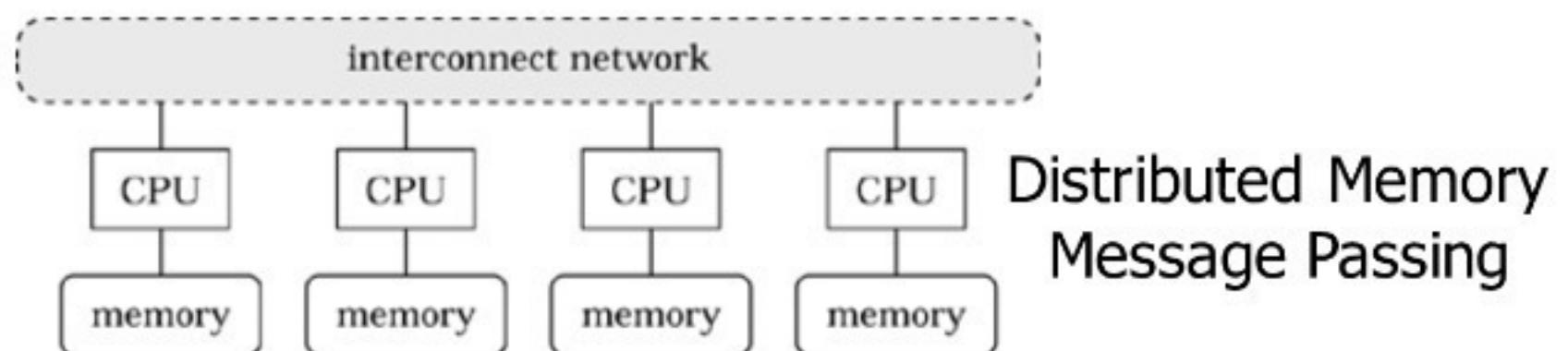




Shared vs Distributed



Shared Memory



Distributed Memory
Message Passing

↳ Size

↳ Scale \rightarrow

↳ cost benefit.



DSECL ZG517 - Systems for Data Analytics

Session #5 – Introduction to Systems Architecture – Client-server, P2P systems; Cluster Computing

Murali P
muralip@wila.bits-pilani.ac.in
[Saturday – 04:15 PM]

This presentation uses public contents shared by authors of text books and other relevant web resources. Further, works of Professors from BITS are also used freely in preparing this presentation.

BITS Pilani
Pilani Campus



Agenda

- Client Server systems
- P2P computing
- Distributed Systems
- Cluster Computing

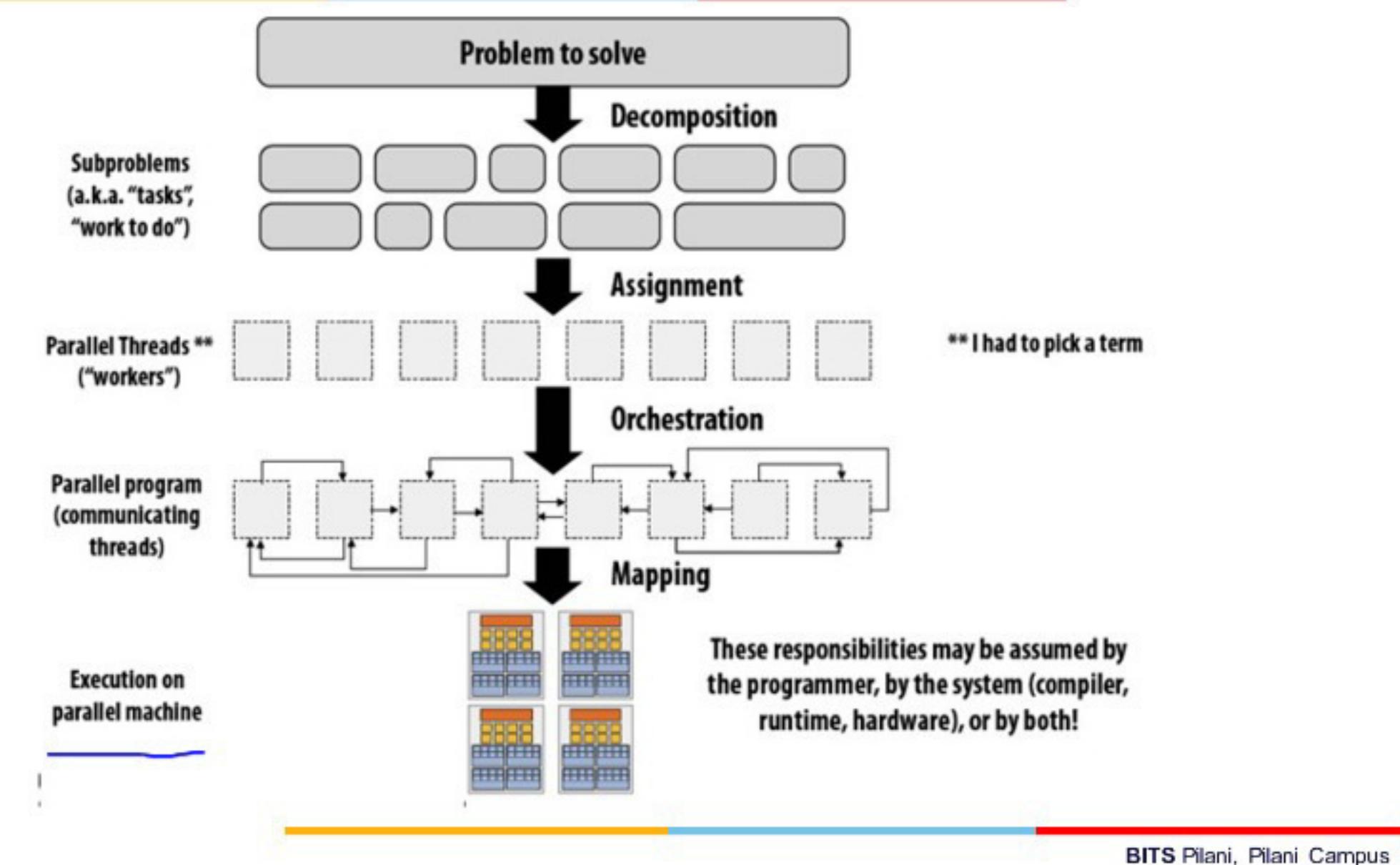


BITS Pilani
Pilani Campus



Client/Server P2P Distributed Systems

Creating Parallel programs



Only for published slide

Creating a Parallel Program – Problem Decomposition

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

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

Only for published slide

Creating a Parallel Program – Problem Decomposition

- Who is responsible for performing problem decomposition?
 - In most cases: the programmer
- Automatic decomposition of sequential programs continues to be a challenging research problem (very difficult in general case)
- Compiler must analyze program, identify dependencies
 - What if dependencies are data dependent (not known at compile time)?
 - Researchers have had modest success with simple loop nests
 - The “magic parallelizing compiler” for complex, general-purpose code has not yet been achieved

Only for published slide

Creating a Parallel Program – Assignment

- Assigning tasks to threads
 - Think of “tasks” as things to do
 - Think of threads as “workers”
- Goals: balance workload, reduce communication costs
- Although programmer is often responsible for decomposition, many languages/runtimes take responsibility for assignment.
- Assignment be performed statically, or dynamically during execution

Only for published slide

Creating a Parallel Program – Assignment

- Assigning tasks to threads
 - Think of “tasks” as things to do
 - Think of threads as “workers”
- Goals: balance workload, reduce communication costs
- Although programmer is often responsible for decomposition, many languages/runtimes take responsibility for assignment.
- Assignment be performed statically, or dynamically during execution

Only for published slide

Creating a Parallel Program – Orchestration

- Involves:
 - Communicating between workers
 - Adding synchronization to preserve dependencies if necessary
 - Organizing data structures in memory
 - Scheduling tasks
- Goals: reduce costs of communication/sync, preserve locality of data reference, reduce overhead, etc.
- Machine details impact many of these decisions
 - If synchronization is expensive, might use it more sparsely

Only for published slide

Creating a Parallel Program – Mapping

- Mapping “threads” (“workers”) to hardware execution units
- Example 1: mapping by the operating system
 - e.g., map thread to hardware execution context on a CPU core
- Example 2: mapping by the compiler
 - Exploiting data parallelism
- Example 3: mapping by the hardware
 - Map CUDA thread blocks to GPU cores
- Some interesting mapping decisions:
 - Place related threads (cooperating threads) on the same processor (maximize locality, data sharing, minimize costs of comm/sync)
 - Place unrelated threads on the same processor (one might be bandwidth limited and another might be compute intensive) to use machine more efficiently

Only for published slide

Creating a Parallel Program – Mapping

- Mapping “threads” (“workers”) to hardware execution units
- Example 1: mapping by the operating system
 - e.g., map thread to hardware execution context on a CPU core
- Example 2: mapping by the compiler
 - Exploiting data parallelism
- Example 3: mapping by the hardware
 - Map CUDA thread blocks to GPU cores
- Some interesting mapping decisions:
 - Place related threads (cooperating threads) on the same processor (maximize locality, data sharing, minimize costs of comm/sync)
 - Place unrelated threads on the same processor (one might be bandwidth limited and another might be compute intensive) to use machine more efficiently

Factors while designing: Parallelization



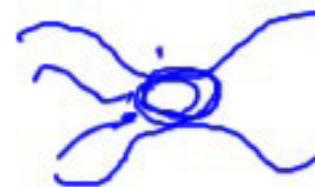
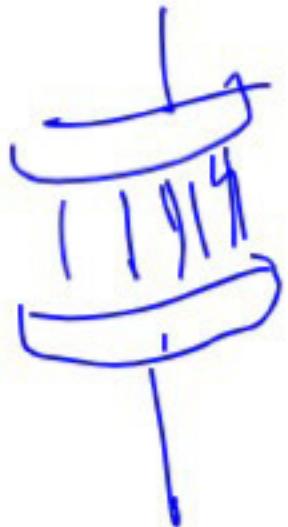
- **Fully Automatic**

- The compiler analyzes the source code and identifies opportunities for parallelism.
- The analysis includes identifying inhibitors to parallelism and possibly weigh cost on whether or not the parallelism would actually improve performance.
 - Loops are the most frequent target for automatic parallelization.

- **Programmer Directed**

- Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
- May be able to be used in conjunction with some degree of automatic parallelization also.
- The most common compiler generated parallelization is done using on-node shared memory and threads (such as OpenMP).

Factors while designing: Understand the Problem and the Program



- If you are starting with a serial program, this necessitates understanding the existing code also.
- Can the program be parallelized?
 - Before spending time in an attempt to develop a parallel solution for a problem, determine whether or not the problem is one that can actually be parallelized.
- Identify the program's **hotspots**:
 - Where is real work done?
- Identify **bottlenecks** in the program
 - What is slowing things down?
- Identify **inhibitors** to parallelism
 - Data dependence
 - Alternative algorithms?



Only for published slide

- Identify the program's **hotspots**:
 - Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
 - Profilers and performance analysis tools can help here
 - Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.
- Identify **bottlenecks** in the program:
 - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
 - May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas
- Identify **inhibitors** to parallelism.
 - One common class of inhibitor is *data dependence*
 - Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.
 - Take advantage of optimized third party parallel software and highly optimized math libraries available from leading vendors (IBM's ESSL, Intel's MKL, AMD's AMCL, etc.).

Factors while designing: Partitioning



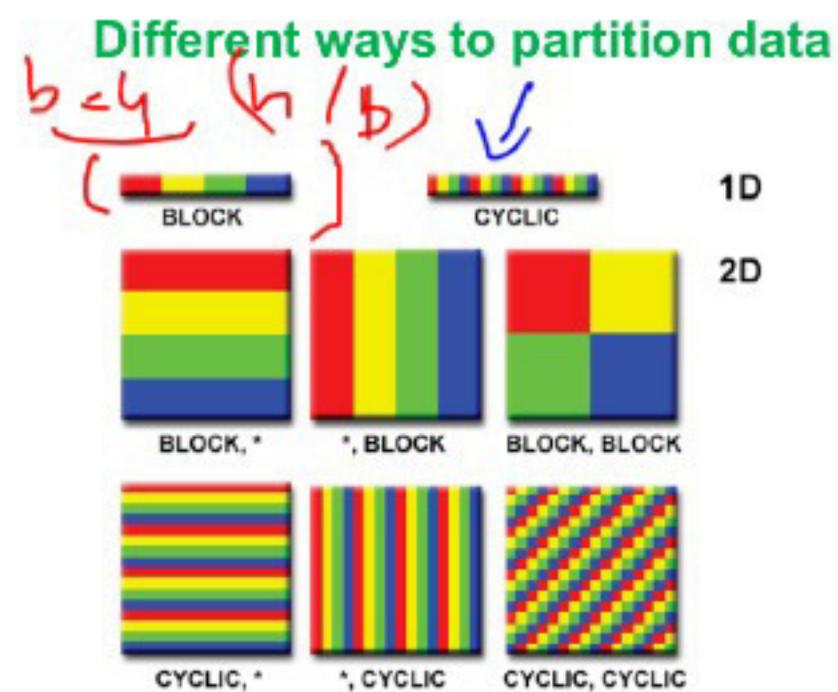
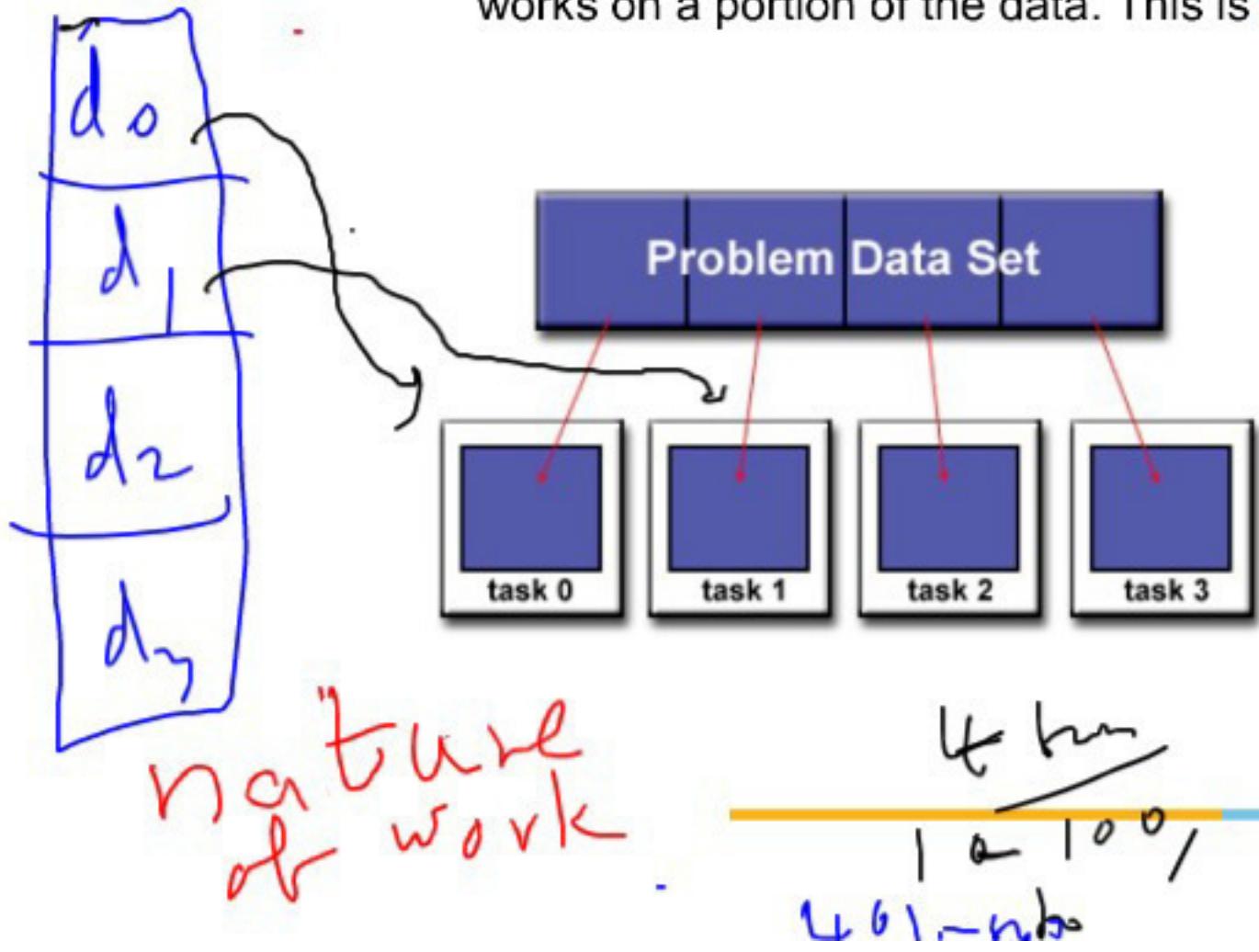
- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks.
- There are two basic ways to partition computational work among parallel tasks:
- ***domain decomposition*** and
- ***functional decomposition***.

Factors while designing: Partitioning[2]

4 days

Domain Decomposition

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



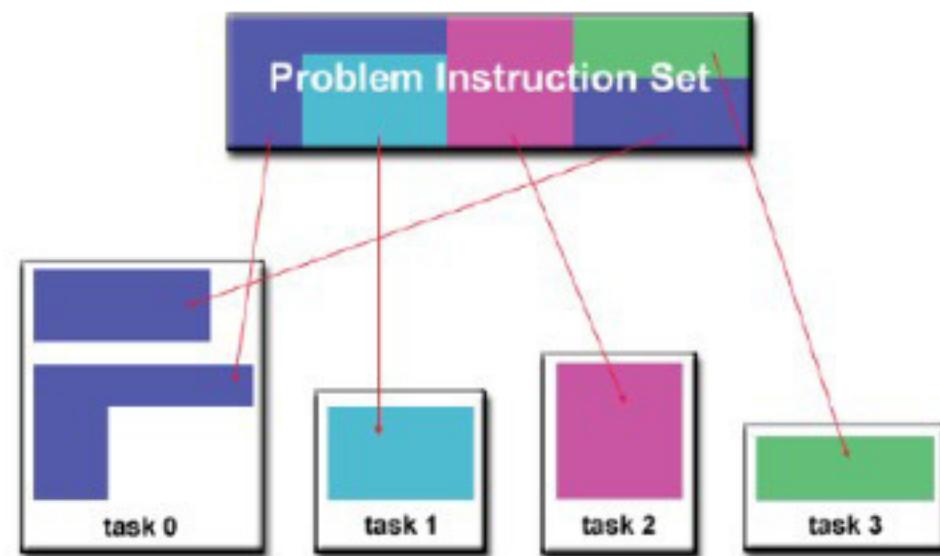
$$\begin{array}{r}
 \cancel{10,000} \\
 4 \\
 = 2500 \\
 t_0 \rightarrow 1 - 2500 \\
 t_1 \rightarrow 2500 - 5000 \\
 t_2 \rightarrow 5000 - 7500 \\
 t_3 \rightarrow 7500 - 10000 \\
 -4000
 \end{array}$$

Factors while designing: Partitioning[3]



Functional Decomposition

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





Factors while designing: Communications between tasks/threads

- Embarrassingly parallel
 - No communication required
 - Example: imagine an image processing operation where every pixel in a black and white image needs to have its color reversed
- **Communication overhead**
 - Inter-task communication virtually always implies overhead.
 - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
 - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
 - Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.



Factors while designing: Communications between tasks/threads[2]

- **Latency vs. Bandwidth**

- **latency** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
- **bandwidth** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec or gigabytes/sec.
- Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.



Factors while designing: Communications between tasks/threads[3]

- **Visibility of communications**
 - With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
 - With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures.
- **Synchronous vs. asynchronous communications**
 - Synchronous communications require some type of "handshaking" between tasks that are sharing data.
 - Synchronous communications are often referred to as **blocking** communications since other work must wait until the communications have completed.
 - Asynchronous communications allow tasks to transfer data independently from one another.
 - Asynchronous communications are often referred to as **non-blocking** communications since other work can be done while the communications are taking place. Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

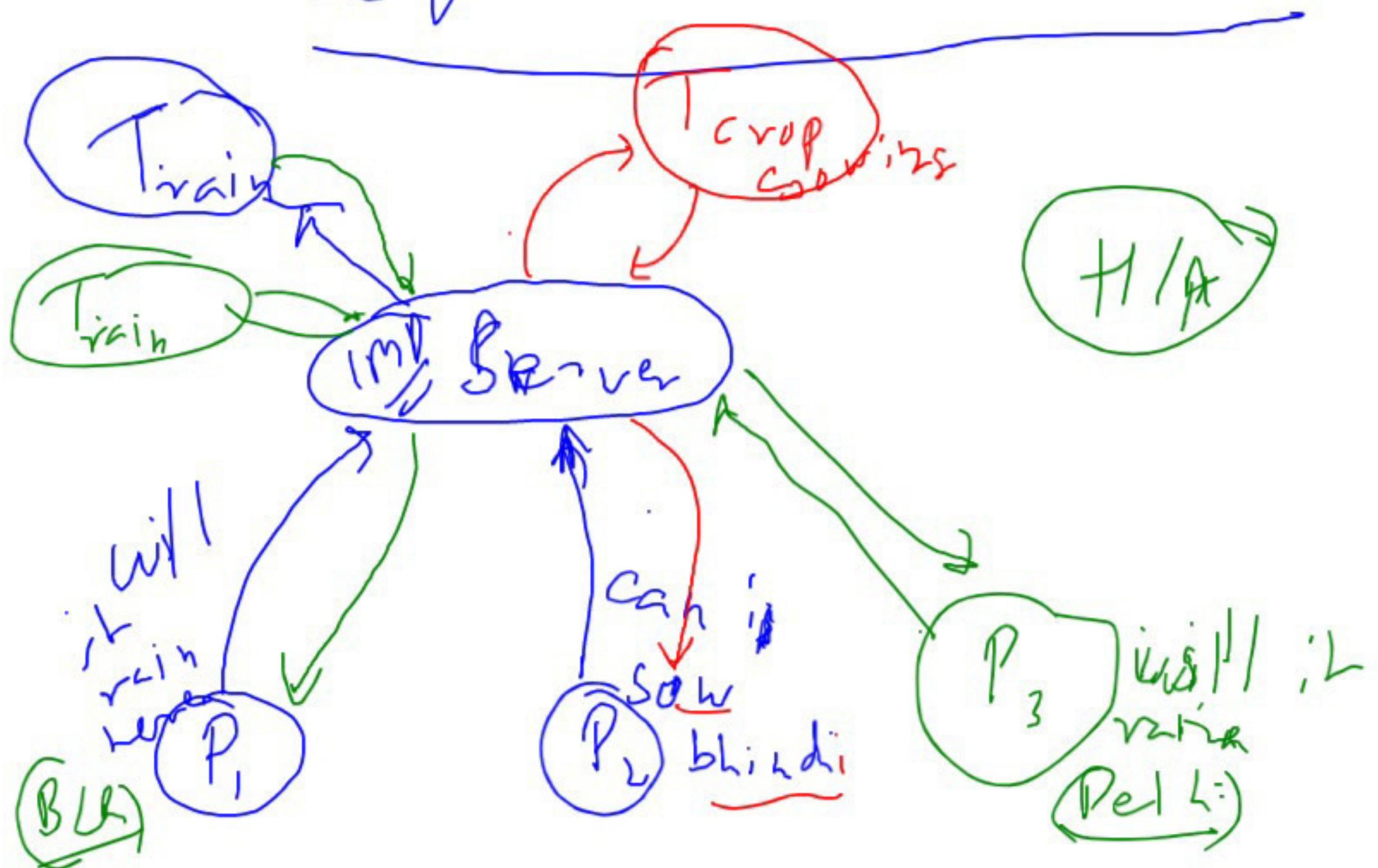


Factors while designing: Communications between tasks/threads[3]

• Scope of communications

- Identifying which tasks must communicate with each other is critical during the design stage of a parallel code. The two types described below can be implemented synchronously or asynchronously.
- ***Point-to-point***
- involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
- ***Collective***
- involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective.

Request Level Parallelism





BITS Pilani
Pilani Campus



Distributed Systems

Reference:

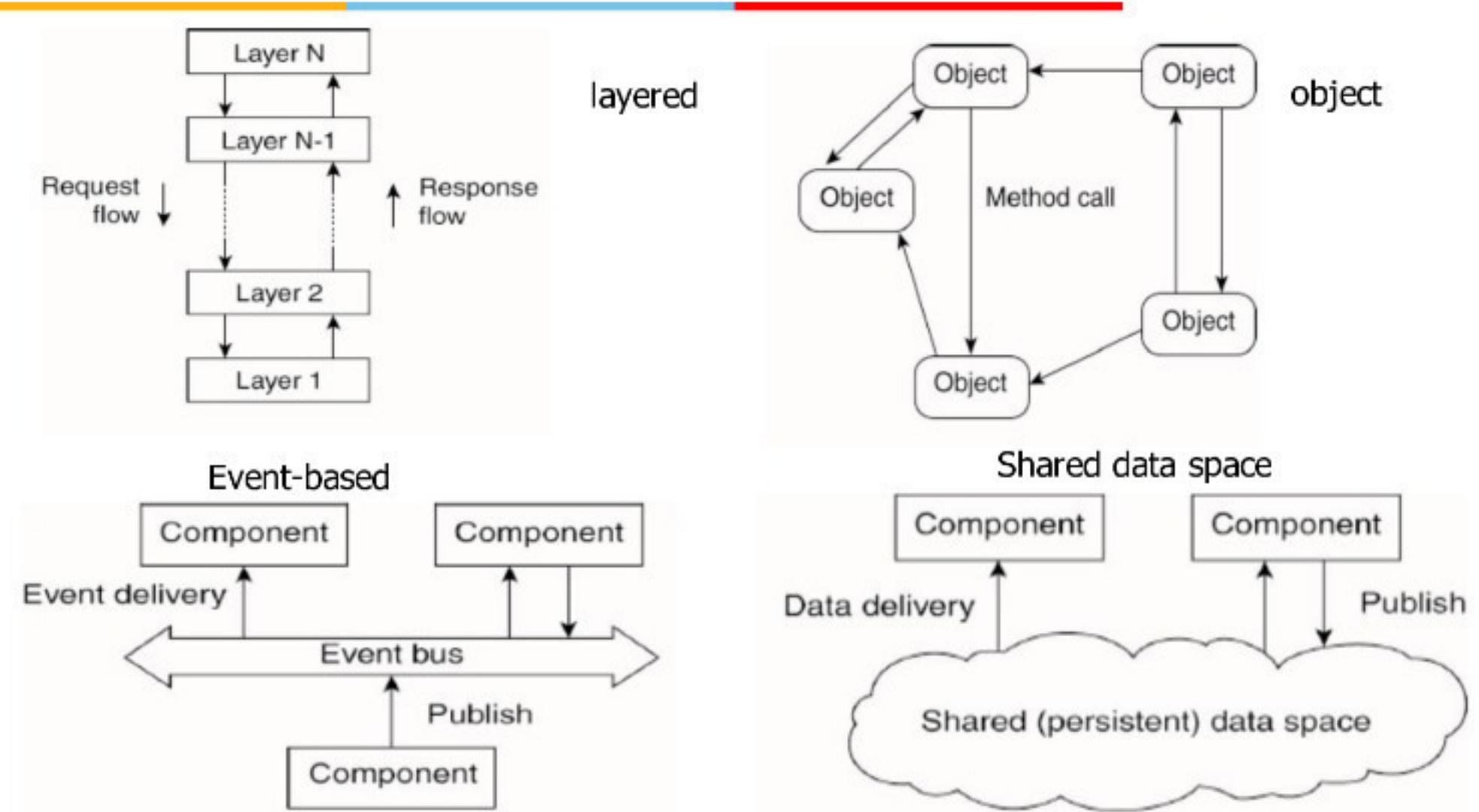
Chapters 1 & 2 of DISTRIBUTED SYSTEMS

Concepts and Design, George Coulouris, Jean Dollimore, Tim Kindberg,
Gordon Blair (ISBN 13: 978-0-13-214301-1)

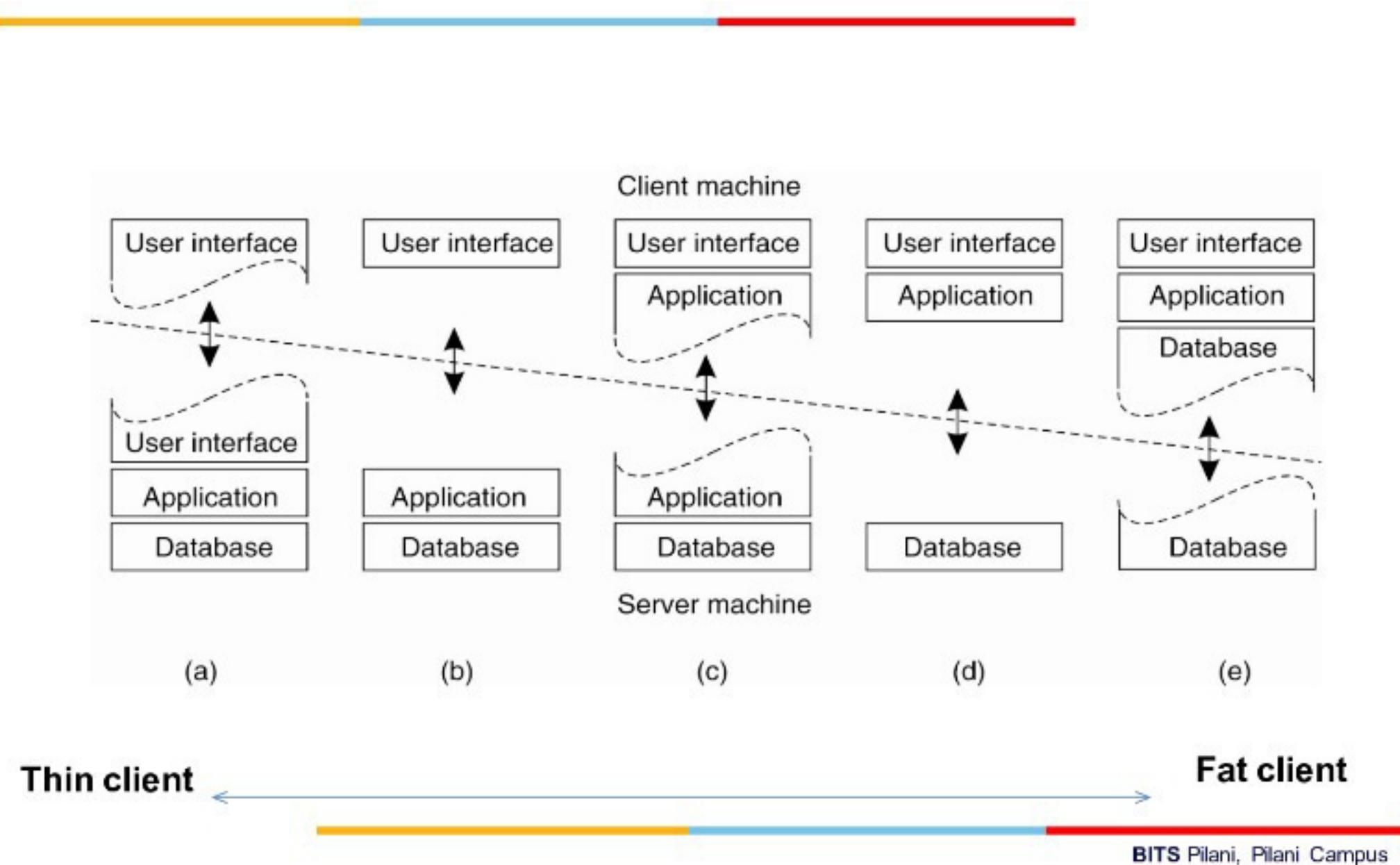
A Definition

*A distributed system is a collection of entities, each of which is **autonomous, programmable, asynchronous** and **failure-prone**, and which communicate through an **unreliable** communication medium using message passing.*

System Architectures

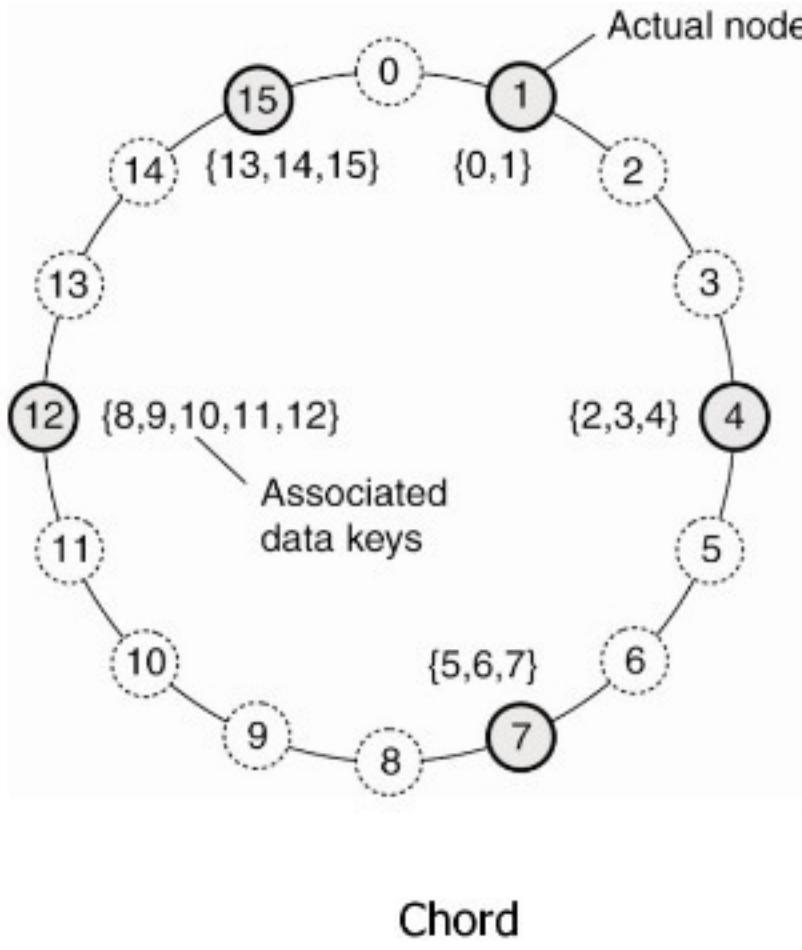


Client/Server Architecture



Structured Peer-to-Peer

- Structured: the overlay network is constructed in a deterministic procedure
 - Most popular: distributed hash table (DHT)
- Key questions
 - How to map data item to nodes
 - How to find the network address of the node responsible for the needed data item
- Two examples
 - Chord and Content Addressable Network (CAN)



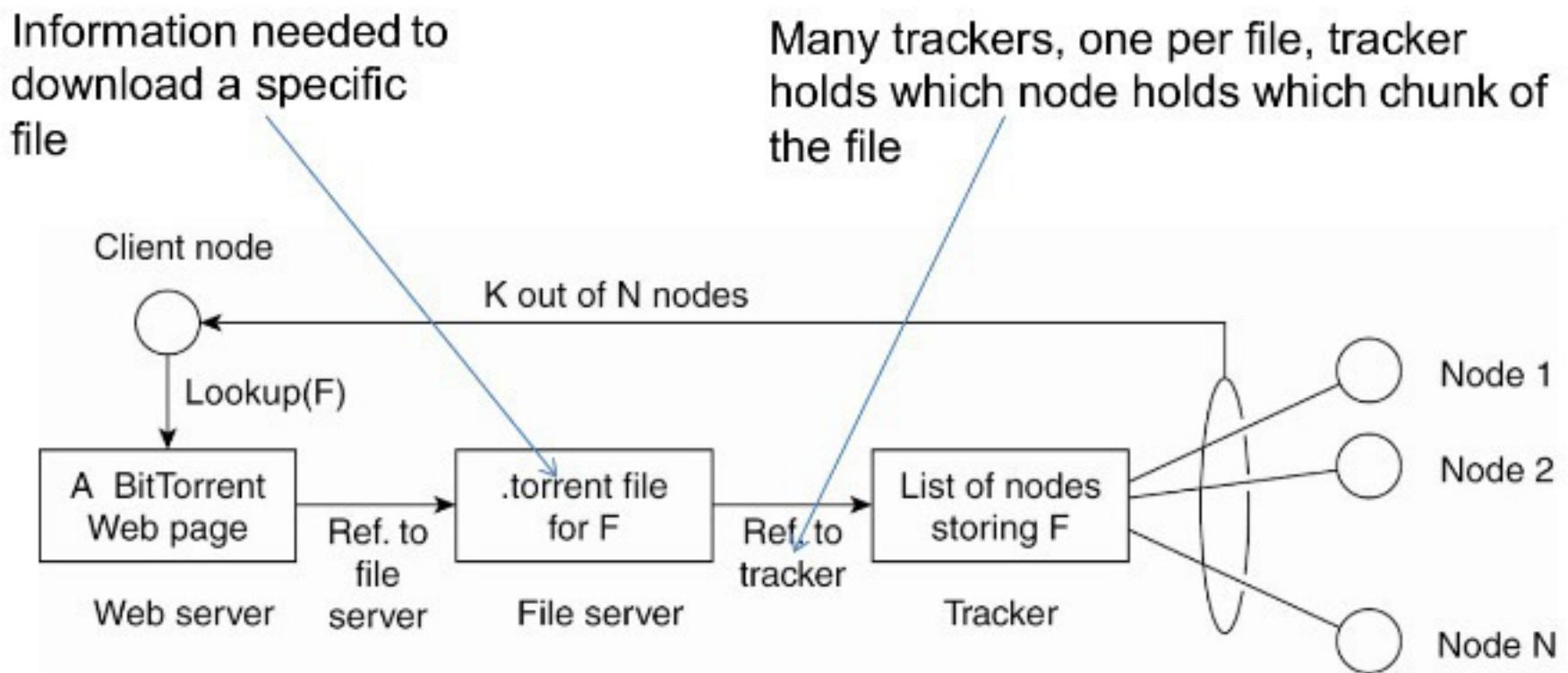


Unstructured P2P Architectures

- Largely relying on randomized algorithm to construct the overlay network
 - Each node has a list of neighbors, which is more or less constructed in a random way
- One challenge is how to efficiently locate a needed data item
 - [Flood the network?](#)
- Many systems try to construct an overlay network that resembles a **random graph**
 - Each node maintains [a partial view](#), i.e., a set of live nodes randomly chosen from the current set of nodes

Hybrid Architecture

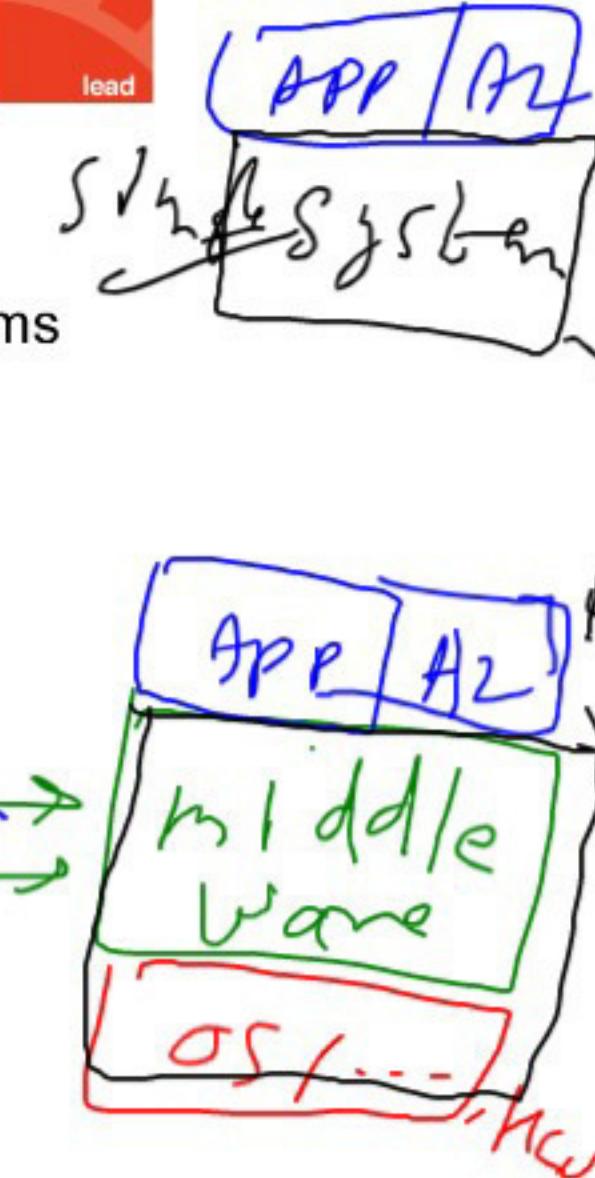
Collaborative Distributed System - BitTorrent



The principal working of BitTorrent (Pouwelse et al. 2004)

Middleware

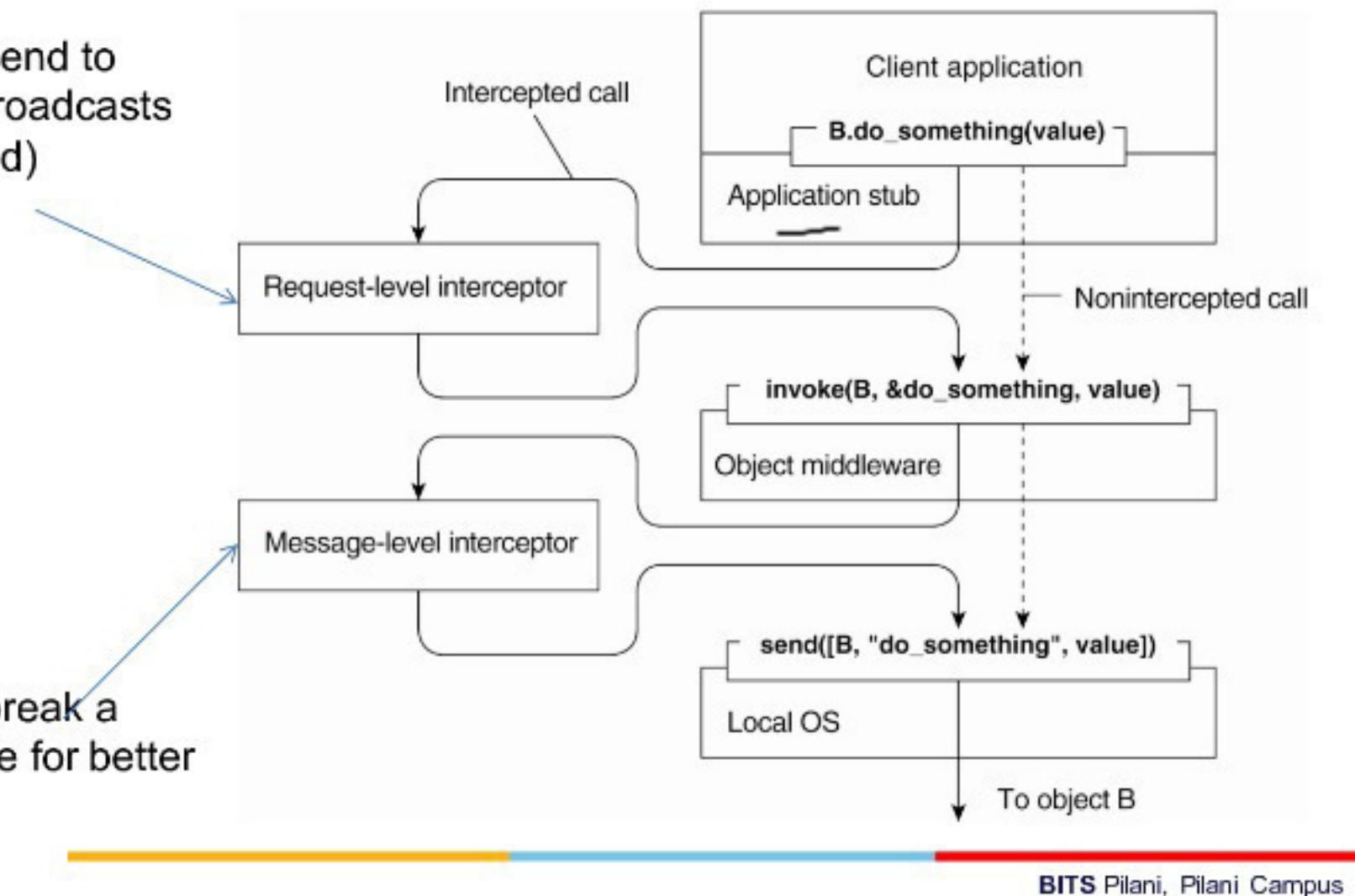
- A middleware layer between application and the distributed platforms for **distribution transparency**
 - Role is to make application development easier, by providing common programming abstractions, by masking the heterogeneity and the distribution of the underlying hardware and operating systems, and by hiding low-level programming details.
- Many middleware follows a specific architecture style
 - Object-based style, event-based style
 - Benefits: simpler to design application
 - Limitations: the solution may not be optimal
- Should be **adaptable** to application requirements
 - **Separating policies from mechanism**

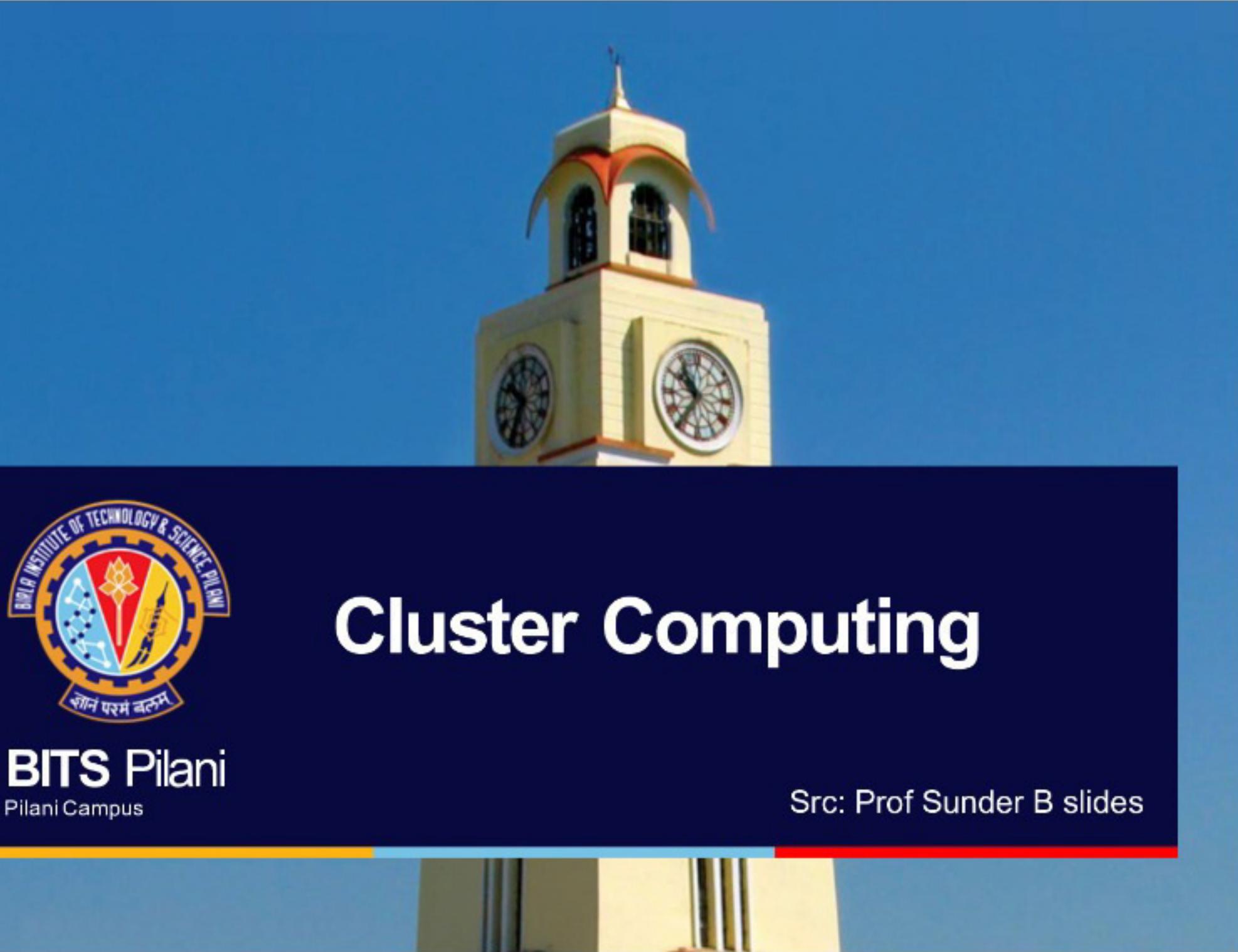


Middleware: Interceptor

May want to send to many other Broadcasts (i.e. Replicated)

May want to break a large message for better performance





BITS Pilani
Pilani Campus

Src: Prof Sunder B slides



Computer Cluster - Definition

- **Definition [Buyya]:**
 - A *cluster* is a type of parallel or distributed processing system
 - consisting of a collection of inter-connected stand-alone computers
 - working together as a single, integrated computing resource.

Cluster - Objectives

- A computer cluster is typically built for one of the following two reasons:
 - ▣ High Performance
 - These are referred to as *compute-clusters*
 - What is the primary motivation to build a compute-cluster (*as opposed to a parallel computer*) for performance?
 - ▣ High Availability
 - Availability

Cluster Motivation

- Cost and Incremental Cost Model:
 - Custom Parallel Computer
 - Initial Cost and Obsolescence
 - Scale-out Cluster
 - Incremental performance by additional nodes (i.e. Stand-alone systems) – a loosely coupled model
 - Commodity Cluster
 - A special-case of scale-out cluster, wherein nodes are COTS (i.e. Commercial Off-The-Shelf) computers
 - i.e. Currently whatever is the cheapest

Clusters and Grids

- Clusters and Grids are both
 - distributed systems built by Connecting stand-alone computers over network
 - and provide the abstraction of a single computer

	Cluster	Grid
Nodes	Homogeneous	Heterogeneous
Network	Local Area	Wide Area
Node membership	Dedicated / Non-dedicated	Non-dedicated
Resource Abstraction	Process Level	Process Level and Resource Level

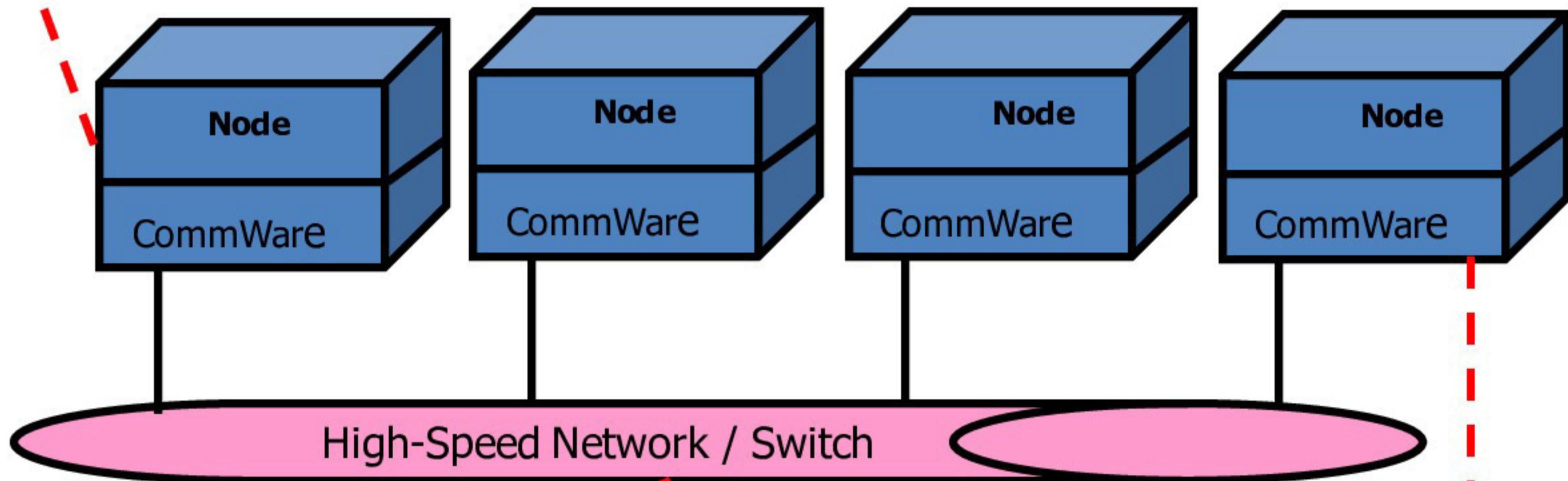
Clusters, Data Centers, and Clouds



- Clouds provide abstract services (including infrastructure) :
 - Services are implemented on large scale computing infrastructure built into central facilities (i.e. Data Centers)
- Typical data center architectures use clusters as the building blocks for:
 - High performance , and more critically, for High Availability

Typical Cluster: Components - Base

- Processor + Memory + Storage
- OS + Run-time environment

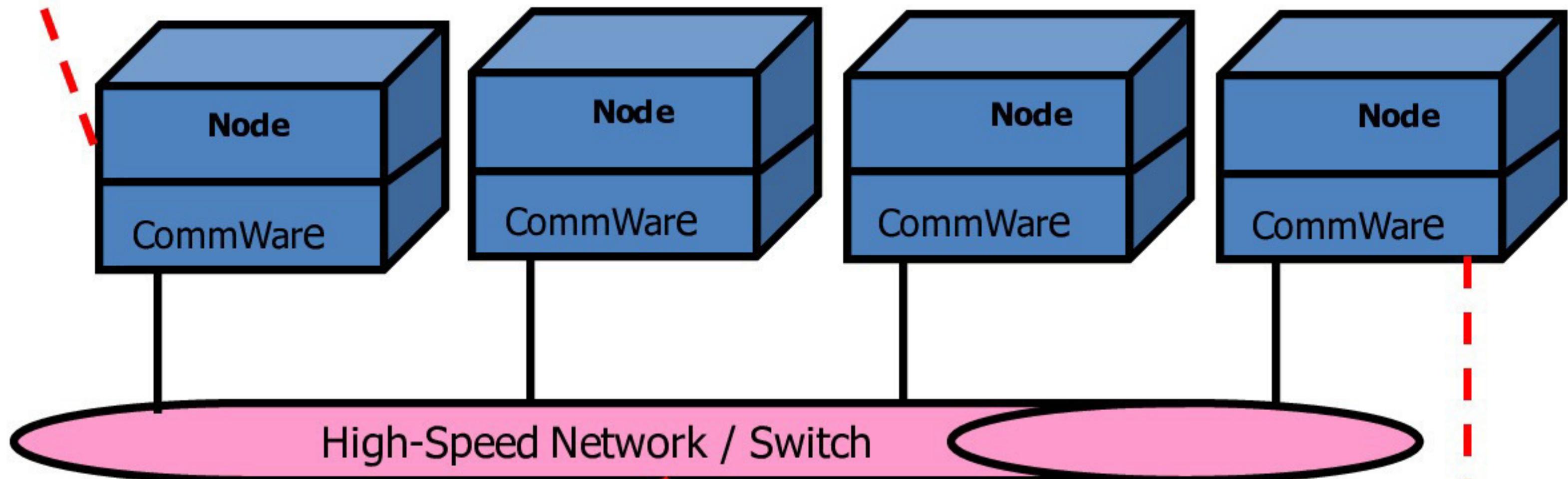


Gigabit Ethernet or
Myrinet/Infiniband

- Network Interface Cards
- Fast Communication Protocols and Services

Typical Cluster: Components - Base

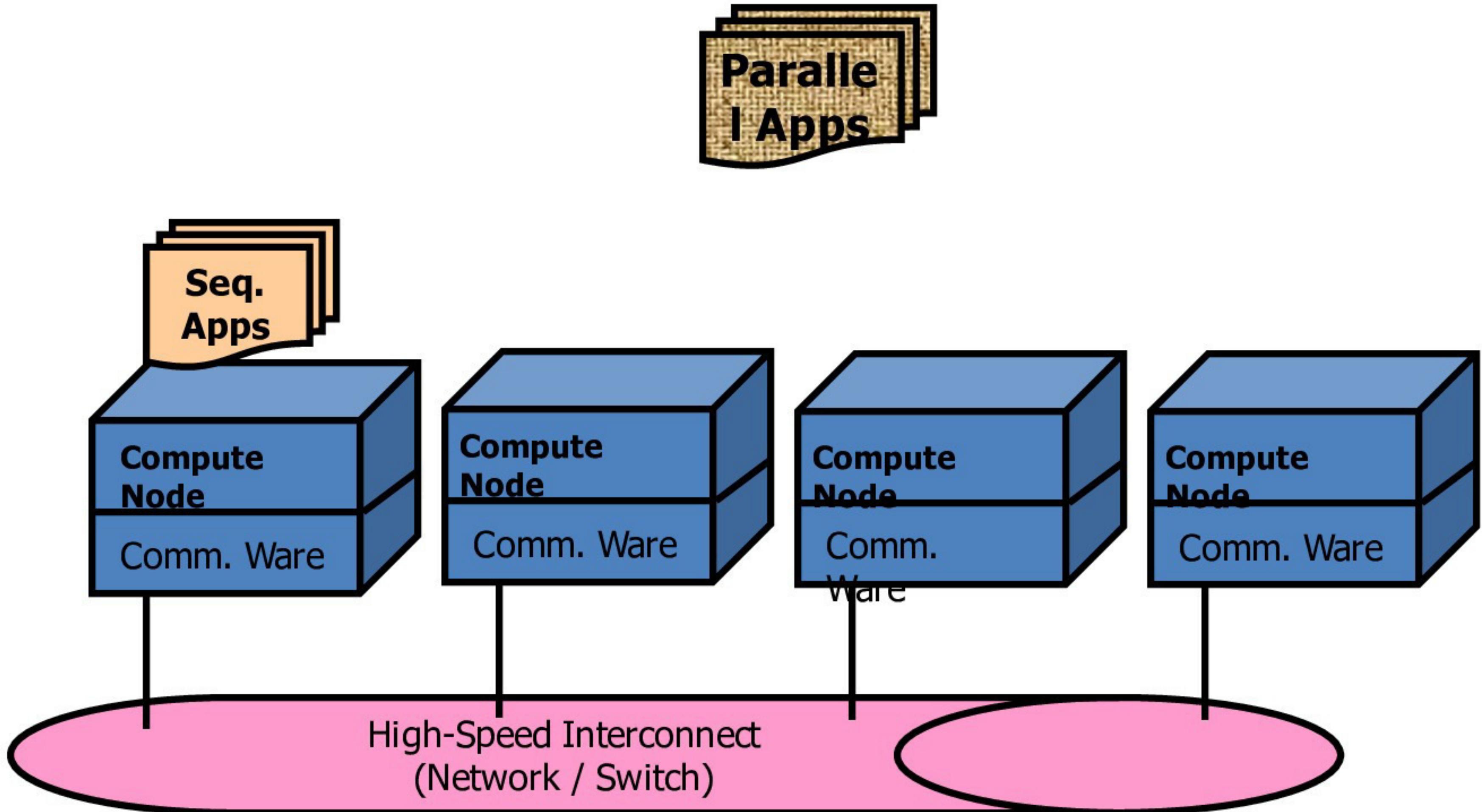
- Processor + Memory + Storage
- OS + Run-time environment



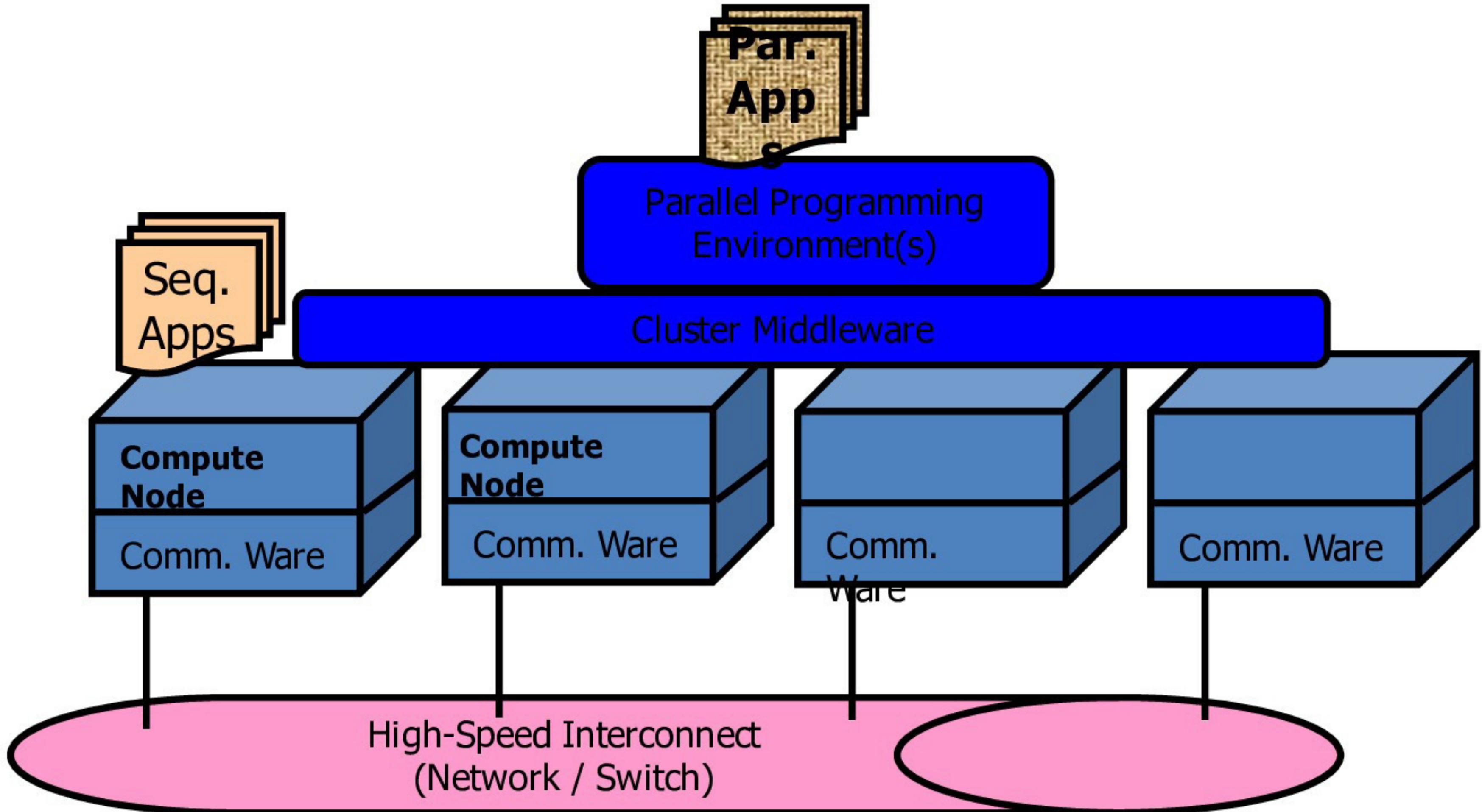
Gigabit Ethernet or
Myrinet/Infiniband

- Network Interface Cards
- Fast Communication Protocols and Services

Typical Cluster: Requirements



Typical Cluster: Architecture



Cluster Middleware - Functions

- There are two main functions:
 - Single System Image (SSI) infrastructure
 - Glues together OSs on all nodes to offer unified access to system resources
 - (High) System Availability (HA) Infrastructure
 - Cluster services for availability
 - Redundancy
 - Fault-tolerance
 - Recovery from failures

Features of Single System Image (SSI)

- **Single Point of Control**
 - The user submits and manages tasks at a single (logical) point – which of course may be any of the nodes
 - **Tracking & Control:**
 - Heartbeat messages, status/progress updates of tasks/processes, control messages (e.g. terminate a process)
 - Must be internally distributed for redundancy and load sharing

Features of Single System Image (SSI) [2]

- **Single Entry Point**
 - **Single authentication for all nodes**
 - **Node failures and load imbalance (when multiple users attempt to log in) must be handled**
 - e.g. Run a local DNS mapping hostnames to IP addresses dynamically

Features of Single System Image (SSI) [3]

- **Single File System**
 - **Single File Hierarchy**
 - e.g. Network File System on Unix/Linux
 - **Visibility of Files**
 - Processes on node A should be able to access files (say via, fopen, fread, or fwrite on Unix) on another node

Features of Single System Image (SSI) [4]

- Single I/O space
 - Any device connected to any node must be accessible from (any process) in any node
 - e.g. Are disks shared?
 - Typically, yes, but not always.

Features of Single System Image (SSI) [5]

- Single Process Space
 - All processes across the cluster have a unique identifier
 - A process on any node can communicate with any other process on any (other) node
 - say, using pipes or signals in Unix

Failure Recovery – Backward Recovery Scheme

- Backward recovery
 - Checkpointing:
 - processes periodically save consistent state information (a.k.a. *checkpoint*) on a stable storage
 - Rollback
 - Post failure,
 - the failed component is isolated,
 - previous checkpoint is restored, and
 - normal operation is resumed
- Pros and Cons:
 - Easy to implement, application independent
 - Rollback results in wasted execution

Failure Recovery – Forward Recovery Scheme

- Forward recovery is useful in systems
 - where execution time is critical
 - e.g. Real-Time System
 - but may
 - be application specific and
 - require additional hardware
- Forward Recovery
 - No rollback
 - Failure diagnosis is used to reconstruct a valid system state:
 - Assumptions?

Recovery: Checkpointing

- **Checkpointing:**
 - Periodic saving of state of execution of a program to stable storage
 - so that a system may recover after a failure
 - Saved state is referred to as a *checkpoint*
 - **Storage:**
 - Default:
 - Hard Disk
 - Alternatives:
 - SSD
 - Node memory??

Checkpointing - Implementation

- Checkpointing can be realized at different levels
 - OS Kernel
 - OS transparently checkpoints and (on failure) restarts processes
 - Library
 - User-space checkpointing program
 - Program has to link to this library
 - explicit calls for checkpointing and restarting
 - link library with source code
 - implicit checkpointing
 - i.e. Compiler inserts checkpointing library calls
 - Application

Checkpointing – Overheads

- Checkpointing Overheads:
 - Time Overhead
 - Number of checkpoints * Time taken to store a checkpoint
 - Storage Overhead
 - Number of checkpoints * Space required for a checkpoint

Checkpointing – Reducing Overheads

- Choosing a checkpoint interval
 - Longer interval results in less overhead but a shorter interval results in faster recovery
 - Optimal Checkpoint Interval
 - $\text{sqrt}(\text{MTTF} * T_c) / h$
 - h is average % of normal computation performed in a check-point interval before the system fails
 - T_c is the time consumed to save a checkpoint

Checkpointing – Reducing Overheads [2]

- Incremental Checkpoint
 - Save only the difference since the last checkpoint
 - sequence of checkpoint files must be kept on stable storage
 - Implications:
 - Increased Storage and Increased Recovery Time vs. Reduced Checkpointing Time
- User-Directed Checkpointing
 - User specifies when and what to save.

Checkpointing – Reducing Overheads [3]

- Forked Checkpointing
 - Typical checkpointing schemes are *blocking* schemes
 - Instead,
 - one can copy a program state in memory and
 - another thread can save the state asynchronously
 - e.g. use *fork* in Unix to create a child process that duplicates the address space and checkpoint it.
 - Why does this save time?
 - Optimization:
 - Use Copy-on-Write

Checkpointing – Parallel Programs

- Parallel programs
 - ▣ Size:
 - State is much larger (processes and network)
 - ▣ Timing and Consistency
 - A *global snapshot* is a set of checkpoints (of different processes).
 - Note that processes may not be synchronous.
 - A snapshot is consistent if
 - a message that is received in a checkpoint of a process has already been sent by another process (as recorded in its checkpoint)

Checkpointing Parallel Programs - Strategies

- Co-ordinated Checkpointing:
 - a.k.a. Consistent Checkpointing:
 - Program is blocked and all processes checkpoint at the same time.
 - Large overhead and Implementation Difficulties
- Independent Checkpointing:
 - Each process checkpoints independent of other processes:
 - Small overhead and existing techniques for sequential programs can be used
 - Consistency guarantees cannot be provided

SSI Feature: Process Migration

- Process migration :
 - **Transparent remote execution**
 - e.g. Sockets are migrated automatically
 - e.g. system calls are intercepted and handled
 - **Binary compatibility**
 - Migration does not require rewriting or re-linking applications

SSI support:

OS Kernel or Gluing Layer - Example

- e.g. Mosix,
 - A cluster OS for nodes running Unix/Linux kernels
 - Consists of:
 - Algorithms for Adaptive Resource Sharing
 - Pre-emptive Process Migration
 - Implemented:
 - at kernel level, as a loadable module
 - without changing the kernel interface



Thank you !



Systems for Data Analytics

Session 6: Cluster Computing

Murali P

muralip@wilp.bits-pilani.ac.in

[Sunday: 4:15 PM to 6:15 PM]

BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Agenda

Clusters

Example: Mosix

Original slides by Prof Sundar B.

Computer Cluster - Definition

- **Definition [Buyya]:**
 - A *cluster* is a type of parallel or distributed processing system
 - consisting of a collection of inter-connected stand-alone computers
 - working together as a single, integrated computing resource.

Cluster - Objectives

- A computer cluster is typically built for one of the following two reasons:
 - High Performance
 - These are referred to as *compute-clusters*
 - What is the primary motivation to build a compute-cluster (*as opposed to a parallel computer*) for performance?
 - High Availability
 - Availability is achieved via redundancy.

Cluster Motivation

- Cost and Incremental Cost Model:
 - Custom Parallel Computer
 - Initial Cost and Obsolescence
 - Scale-out Cluster
 - Incremental performance by additional nodes (i.e. Stand-alone systems) – a loosely coupled model
 - Commodity Cluster
 - A special-case of scale-out cluster, wherein nodes are COTS (i.e. Commercial Off-The-Shelf) computers
 - i.e. Currently whatever is the cheapest

Clusters and Grids

- Clusters and Grids are both
 - distributed systems built by Connecting stand-alone computers over network
 - and provide the abstraction of a single computer

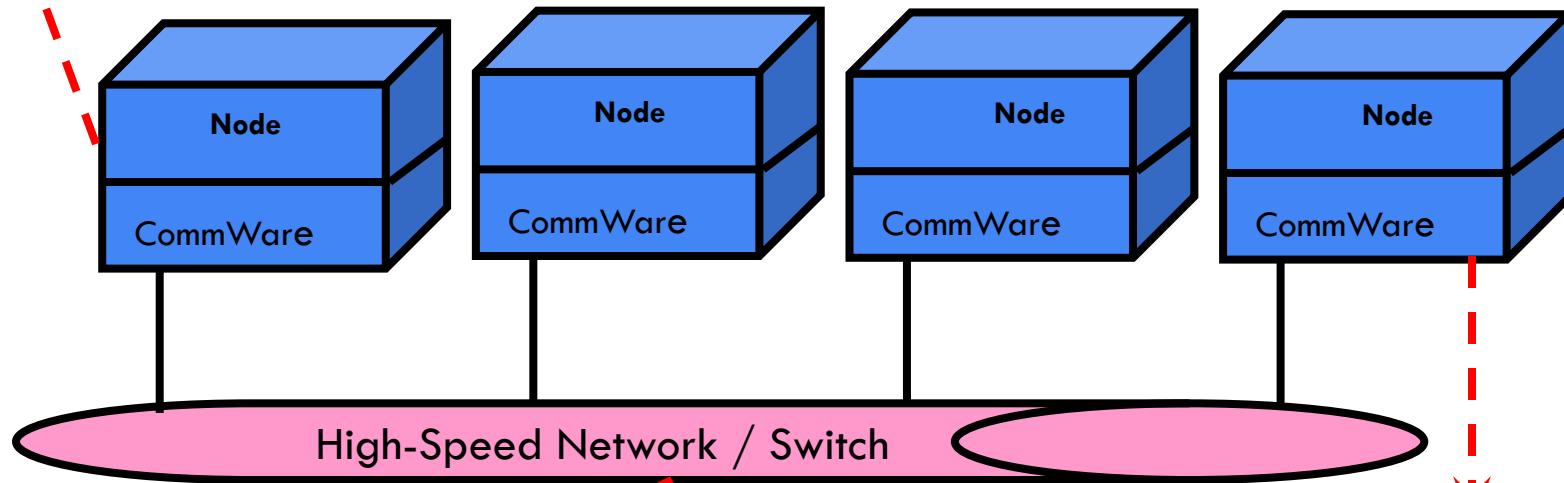
	Cluster	Grid
Nodes	Homogeneous	Heterogeneous
Network	Local Area	Wide Area
Node membership	Dedicated / Non-dedicated	Non-dedicated
Resource Abstraction	Process Level	Process Level and Resource Level

Clusters, Data Centers, and Clouds

- Clouds provide abstract services (including infrastructure) :
 - Services are implemented on large scale computing infrastructure built into central facilities (i.e. Data Centers)
- Typical data center architectures use clusters as the building blocks for:
 - High performance , and more critically, for High Availability

Typical Cluster: Components - Base

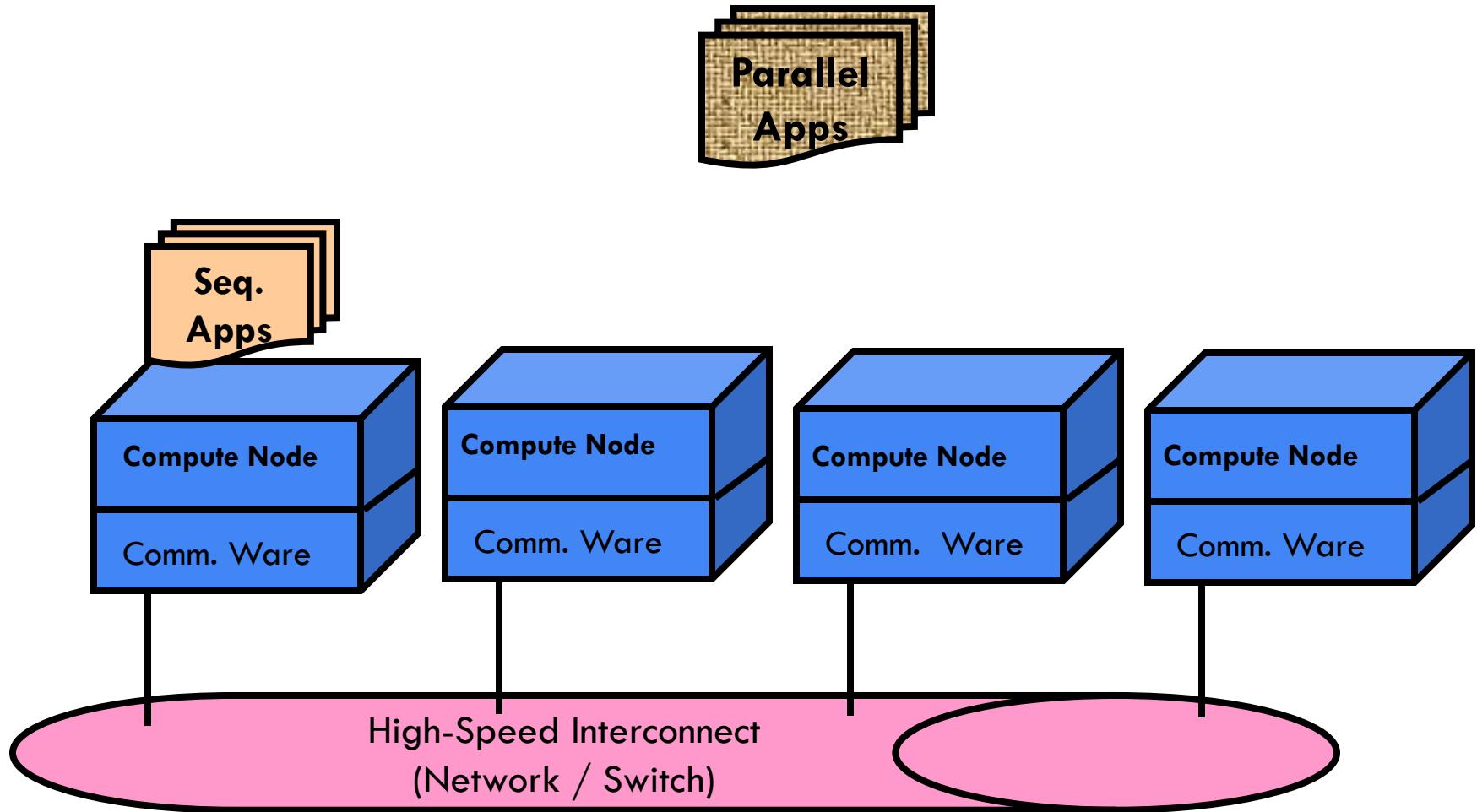
- Processor+Memory+Storage
- OS+Run-time environment



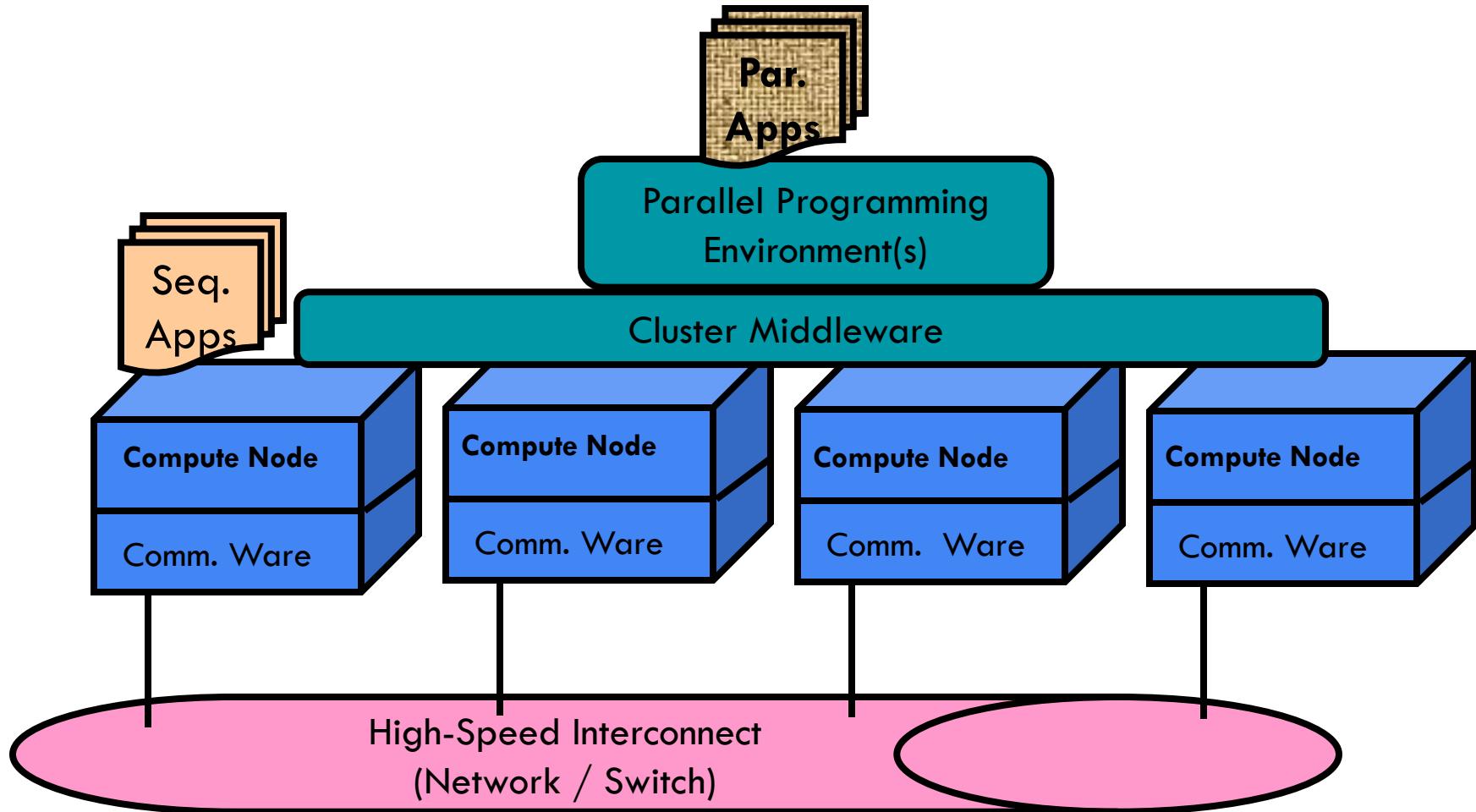
- Network Interface Cards
- Fast Communication Protocols and Services

Gigabit Ethernet or Myrinet/Infiniband

Typical Cluster - Requirements



Typical Cluster Architecture



Cluster Middleware - Functions

- There are two main functions:
 - Single System Image (SSI) infrastructure
 - Glues together OSs on all nodes to offer unified access to system resources
 - (High) System Availability (HA) Infrastructure
 - Cluster services for availability
 - Redundancy
 - Fault-tolerance
 - Recovery from failures

Features of Single System Image (SSI)

- Single Point of Control
 - The user submits and manages tasks at a single (logical) point – which of course may be any of the nodes
 - Tracking & Control:
 - Heartbeat messages, status/progress updates of tasks/processes, control messages (e.g. terminate a process)
 - Must be internally distributed for redundancy and load sharing

Features of Single System Image (SSI)

[2]

-
- Single Entry Point
 - Single authentication for all nodes
 - Node failures and load imbalance (when multiple users attempt to log in) must be handled
 - e.g. Run a local DNS mapping hostnames to IP addresses dynamically

Features of Single System Image (SSI)

[3]

- Single File System
 - Single File Hierarchy
 - e.g. Network File System on Unix/Linux
 - Visibility of Files
 - Processes on node A should be able to access files (say via, fopen, fread, or fwrite on Unix) on another node

Features of Single System Image (SSI)

[4]

- Single I/O space
 - Any device connected to any node must be accessible from (any process) in any node
 - e.g. Are disks shared?
 - Typically, yes, but not always.

Features of Single System Image (SSI) [5]

- Single Process Space
 - All processes across the cluster have a unique identifier
 - A process on any node can communicate with any other process on any (other) node
 - say, using pipes or signals in Unix

Failure Recovery – Backward Recovery Scheme

- Backward recovery
 - Checkpointing:
 - processes periodically save consistent state information (a.k.a. *checkpoint*) on a stable storage
 - Rollback
 - Post failure,
 - the failed component is isolated,
 - previous checkpoint is restored, and
 - normal operation is resumed
- Pros and Cons:
 - Easy to implement, application independent
 - Rollback results in wasted execution

Failure Recovery – Forward Recovery Scheme

- Forward recovery is useful in systems
 - where execution time is critical
 - e.g. Real-Time System
 - but may
 - be application specific and
 - require additional hardware
- Forward Recovery
 - No rollback
 - Failure diagnosis is used to reconstruct a valid system state:
 - Assumptions?

Recovery: Checkpointing

- **Checkpointing:**
 - Periodic saving of state of execution of a program to stable storage
 - so that a system may recover after a failure
 - Saved state is referred to as a *checkpoint*
 - Storage:
 - Default:
 - Hard Disk
 - Alternatives:
 - SSD
 - Node memory??

Checkpointing - Implementation

- Checkpointing can be realized at different levels
 - OS Kernel
 - OS transparently checkpoints and (on failure) restarts processes
 - Library
 - User-space checkpointing program
 - Program has to link to this library
 - explicit calls for checkpointing and restarting
 - link library with source code
 - implicit checkpointing
 - i.e. Compiler inserts checkpointing library calls
 - Application

Checkpointing - Overheads

- Checkpointing Overheads:
 - Time Overhead
 - Number of checkpoints * Time taken to store a checkpoint
 - Storage Overhead
 - Number of checkpoints * Space required for a checkpoint

Checkpointing – Reducing Overheads

- Choosing a checkpoint interval
 - Longer interval results in less overhead but a shorter interval results in faster recovery
 - Optimal Checkpoint Interval
 - $\sqrt{MTTF * T_c} / h$
 - h is average % of normal computation performed in a check-point interval before the system fails
 - T_c is the time consumed to save a checkpoint

Checkpointing - Reducing Overheads [2]

- **Incremental Checkpoint**
 - Save only the difference since the last checkpoint
 - sequence of checkpoint files must be kept on stable storage
 - Implications:
 - Increased Storage and Increased Recovery Time vs. Reduced Checkpointing Time
- **User-Directed Checkpointing**
 - User specifies when and what to save.

Checkpointing - Reducing Overheads

[3]

- Forked Checkpointing
 - Typical checkpointing schemes are *blocking* schemes
 - Instead,
 - one can copy a program state in memory and
 - another thread can save the state asynchronously
 - e.g. use *fork* in Unix to create a child process that duplicates the address space and checkpoint it.
 - Why does this save time?
 - Optimization:
 - Use Copy-on-Write

Checkpointing – Parallel Programs

- Parallel programs
 - Size:
 - State is much larger (processes and network)
 - Timing and Consistency
 - A *global snapshot* is a set of checkpoints (of different processes).
 - Note that processes may not be synchronous.
 - A snapshot is consistent if
 - a message that is received in a checkpoint of a process has already been sent by another process (as recorded in its checkpoint)

Checkpointing Parallel Programs - Strategies

- Co-ordinated Checkpointing:
 - a.k.a. Consistent Checkpointing:
 - Program is blocked and all processes checkpoint at the same time.
 - Large overhead and Implementation Difficulties
- Independent Checkpointing:
 - Each process checkpoints independent of other processes:
 - Small overhead and existing techniques for sequential programs can be used
 - Consistency guarantees cannot be provided

SSI Feature: Process Migration

- Process migration :
 - Transparent remote execution
 - e.g. Sockets are migrated automatically
 - e.g. system calls are intercepted and handled
 - Binary compatibility
 - Migration does not require rewriting or re-linking applications

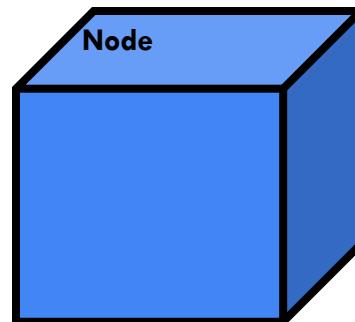
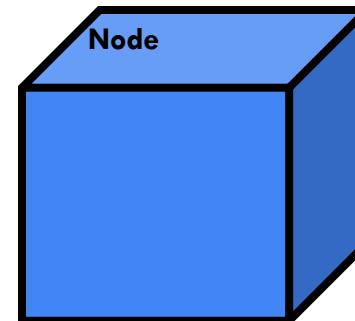
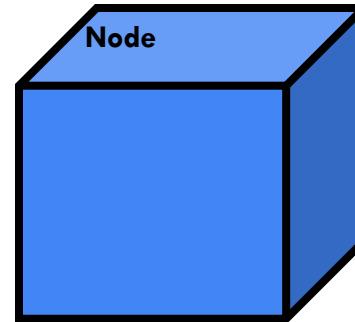
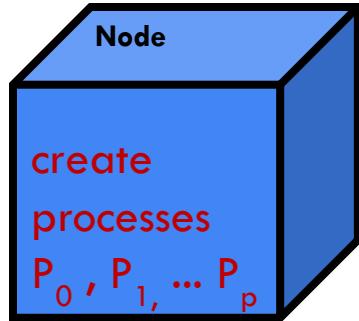
SSI support: OS Kernel or Gluing Layer - Example

- e.g. Mosix,
 - A cluster OS for nodes running Unix/Linux kernels
 - Consists of:
 - Algorithms for Adaptive Resource Sharing
 - Pre-emptive Process Migration
 - Implemented:
 - at kernel level, as a loadable module
 - without changing the kernel interface

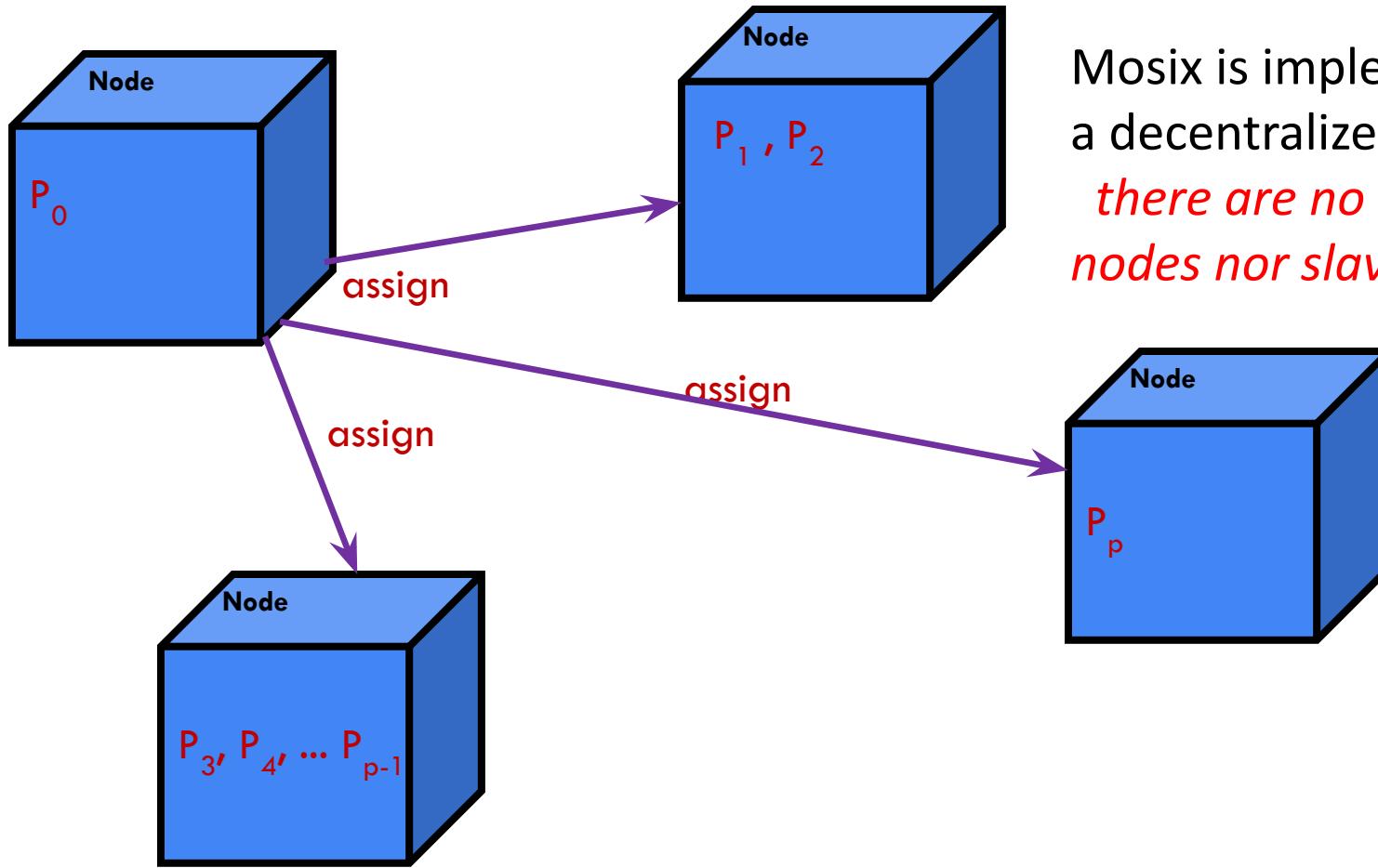


Example: MOSIX

SSI support: Mosix: Work Distribution

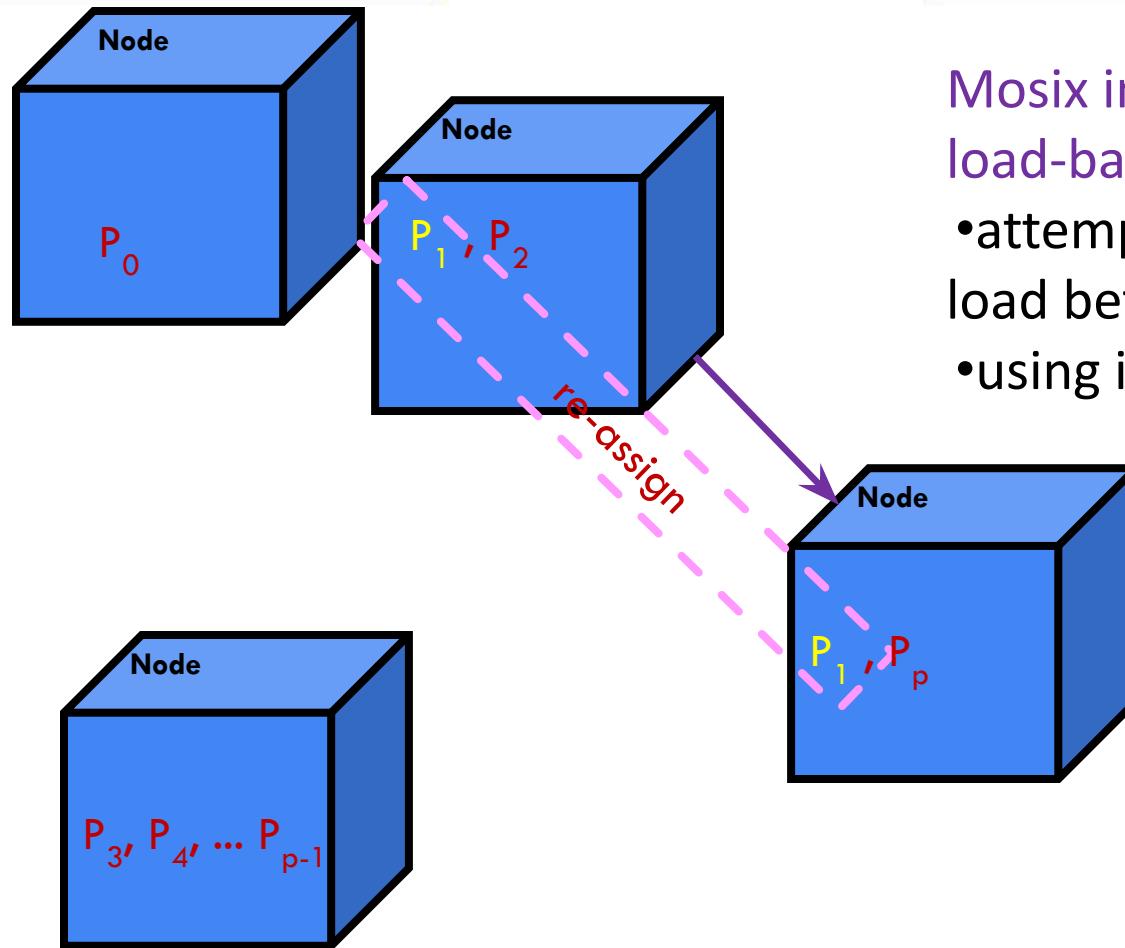


SSI support: Mosix: Work Distribution



Mosix is implemented in a decentralized fashion:
there are no master nodes nor slave nodes.

SSI support: Mosix: Load Balancing



Mosix implements dynamic load-balancing:

- attempts to reduce differences in load between nodes
- using its process migration facility

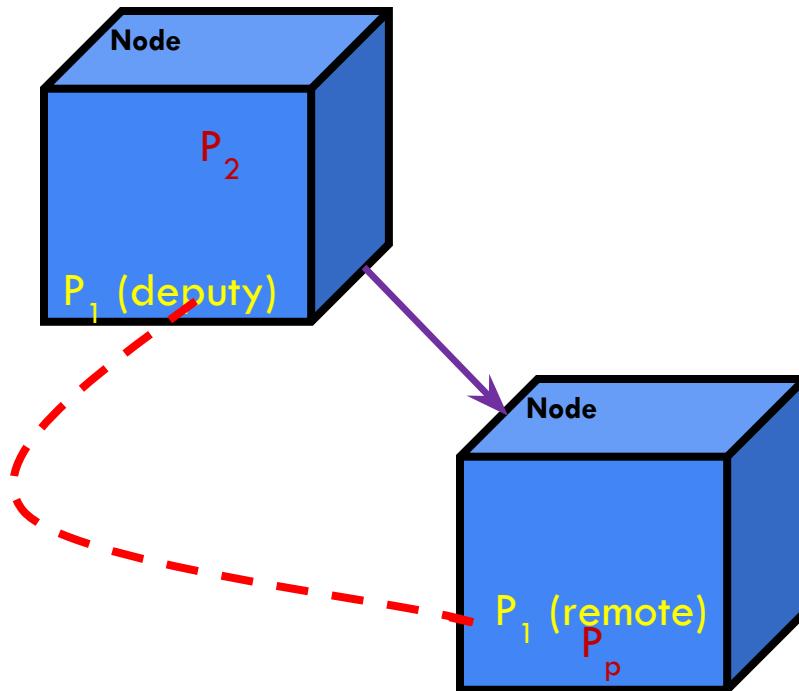
Load balancing adapts to:

- changes in the loads of nodes or
- the runtime characteristics of the processes

SSI support: Mosix: Process Migration

- **Mosix :Process Migration:**
 - Each process has a Unique Home Node (UHN)
 - A migrating process is divided into two contexts:
 - user context (referred to as the remote):
 - code, stack, data, memory-maps, and registers
 - system context , dependent on UHN (referred to as the deputy)
 - resource descriptors, kernel-stack
 - The deputy is never migrated
 - only the remote is migrated

Mosix: Process Migration: Implementation



Interactions between the system context and the user context:

- are intercepted and forwarded over the network
- e.g. system calls (in remote):
 - if a call is site independent, then it is executed locally by the *remote*
 - if a call is site dependent it is forwarded to the *deputy*

Mosix: Process Migration: Overhead

Migration Overhead:

- Interception and Forwarding
 - System calls
 - signal delivery
 - process wakeup events
 - file access operations
 - etc.

will have to be intercepted and forwarded resulting in performance overhead.

- Copy Overhead:
 - Memory-copies between user memory and kernel memory will now become network-copies

Mosix: Process Migration: Overhead

Techniques for mitigating overheads:

- Migratable Sockets
- Caching: Buffering and Prefetching to avoid copying overhead
- Copy-on-demand: copy only dirty pages

Andah's

S.A



BITS Pilani
Pilani Campus

DSECL ZG515 - SDA

Session #7 – CLuster & Performance

Attributes of Systems

S.P.Vimal

vimalsp@wilp.bits-pilani.ac.in

[Sunday – 09:00 AM]

This presentation uses public contents shared by authors of these text books like Tom Mitchell, Christopher Bishop and many others.
Further, works of Professors from BITS are also used freely in preparing this presentation.

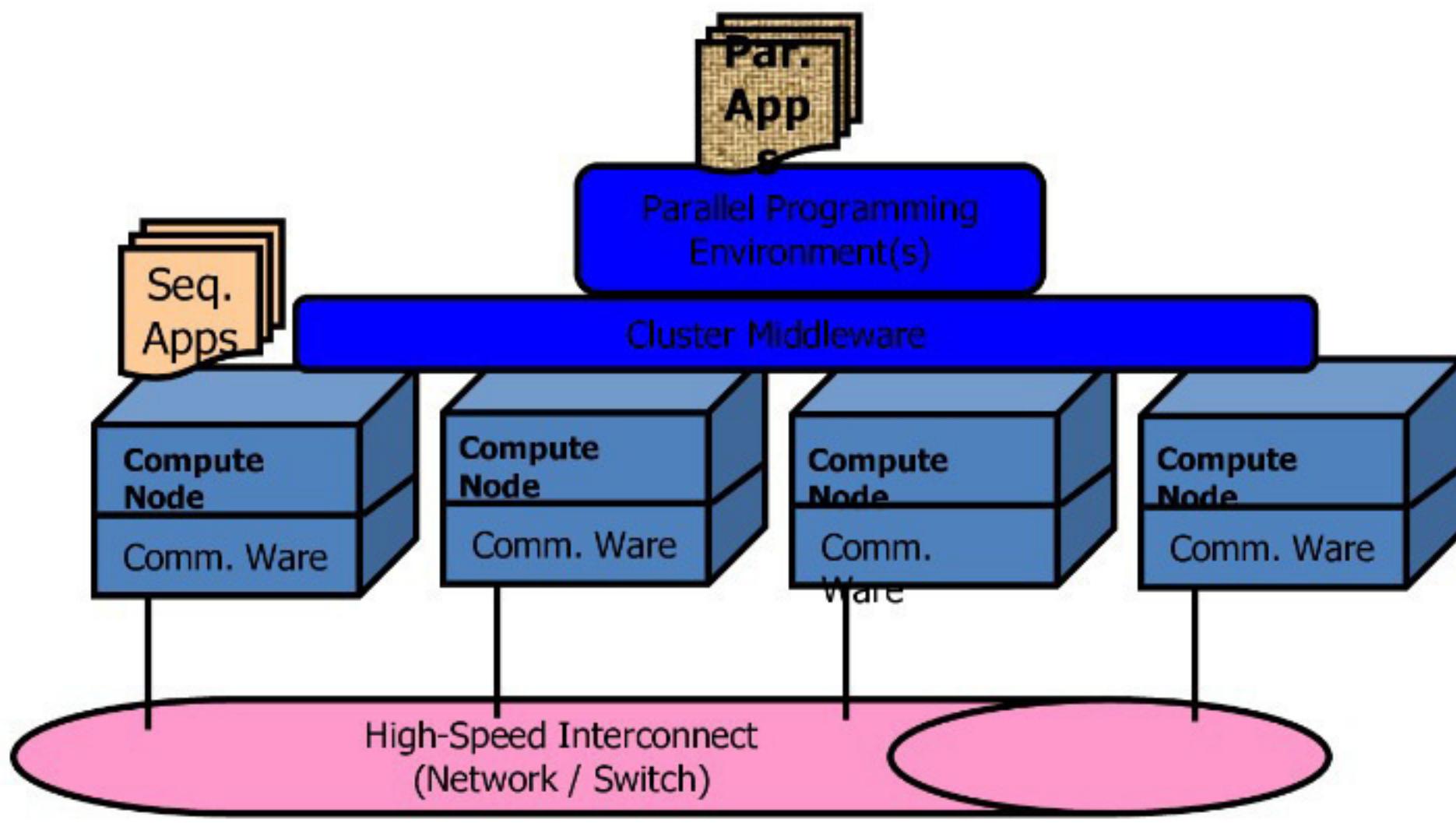


Agenda

- Cluster wrap up
- ↘ Performance Attributes
 - Speedup
 - Reliability

Failure - Fault recovery

Typical Cluster Architecture





Cluster Middleware - Functions

- There are two main functions:
 - Single System Image (SSI) infrastructure
 - Glues together OSs on all nodes to offer unified access to system resources
 - (High) System Availability (HA) Infrastructure
 - Cluster services for availability
 - Redundancy
 - Fault-tolerance
 - Recovery from failures

Features of Single System Image (SSI)



- Single Point of Control
 - The user submits and manages tasks at a single (logical) point – which of course may be any of the nodes
 - Tracking & Control:
 - Heartbeat messages, status/progress updates of tasks/processes, control messages (e.g. terminate a process)
 - Must be internally distributed for redundancy and load sharing

Features of Single System Image (SSI) [2]



- Single Entry Point
 - Single authentication for all nodes
 - Node failures and load imbalance (when multiple users attempt to log in) must be handled
 - e.g. Run a local DNS mapping hostnames to IP addresses dynamically

Features of Single System Image (SSI) [3]



- Single File System
 - Single File Hierarchy
 - e.g. Network File System on Unix/Linux
 - Visibility of Files
 - Processes on node A should be able to access files (say via, fopen, fread, or fwrite on Unix) on another node

Features of Single System Image (SSI) [4]



- Single I/O space
 - Any device connected to any node must be accessible from (any process) in any node
 - e.g. Are disks shared?
 - Typically, yes, but not always.

Features of Single System Image (SSI) [5]



- Single Process Space
 - All processes across the cluster have a unique identifier
 - A process on any node can communicate with any other process on any (other) node
 - say, using pipes or signals in Unix

Failure Recovery – Backward Recovery Scheme



- Backward recovery
 - Checkpointing:
 - processes periodically save consistent state information (a.k.a. **checkpoint**) on a stable storage
 - Rollback
 - Post failure,
 - the failed component is isolated,
 - previous checkpoint is restored, and
 - normal operation is resumed
- Pros and Cons:
 - Easy to implement, application independent
 - Rollback results in wasted execution

Failure Recovery – Forward Recovery Scheme



- Forward recovery is useful in systems
 - where execution time is critical
 - e.g. Real-Time System
 - but may
 - be application specific and
 - require additional hardware
- Forward Recovery
 - No rollback
 - Failure diagnosis is used to reconstruct a valid system state:
 - Assumptions?



Recovery: Checkpointing

- Checkpointing:
 - Periodic saving of state of execution of a program to stable storage
 - so that a system may recover after a failure
 - Saved state is referred to as a **checkpoint**
 - Storage:
 - Default:
 - Hard Disk
 - Alternatives:
 - SSD
 - Node memory??



Checkpointing - Implementation

- Checkpointing can be realized at different levels
 - OS Kernel
 - OS transparently checkpoints and (on failure) restarts processes
 - Library
 - User-space checkpointing program
 - Program has to link to this library
 - » explicit calls for checkpointing and restarting
 - link library with source code
 - » implicit checkpointing
 - » i.e. Compiler inserts checkpointing library calls
 - Application



Checkpointing - Overheads

- Checkpointing Overheads:
 - Time Overhead
 - Number of checkpoints * Time taken to store a checkpoint
 - Storage Overhead
 - Number of checkpoints * Space required for a checkpoint

Checkpointing – Reducing Overheads



- Choosing a checkpoint interval
 - Longer interval results in less overhead but a shorter interval results in faster recovery
 - Optimal Checkpoint Interval
 - $\sqrt{\text{MTTF} * T_c} / h$
 - h is average % of normal computation performed in a check-point interval before the system fails
 - T_c is the time consumed to save a checkpoint

Checkpointing – Reducing Overheads [2]



- Incremental Checkpoint
 - Save only the difference since the last checkpoint
 - sequence of checkpoint files must be kept on stable storage
 - Implications:
 - Increased Storage and Increased Recovery Time vs. Reduced Checkpointing Time
- User-Directed Checkpointing
 - User specifies when and what to save.

Checkpointing – Reducing Overheads [3]



- Forked Checkpointing
 - Typical checkpointing schemes are **blocking** schemes
 - Instead,
 - one can copy a program state in memory and
 - another thread can save the state asynchronously
 - e.g. use *fork* in Unix to create a child process that duplicates the address space and checkpoint it.
 - Why does this save time?
 - Optimization:
 - Use Copy-on-Write

Checkpointing – Parallel Programs

• Parallel programs

– Size:

- State is much larger (processes and network)

– Timing and Consistency

- A **global snapshot** is a set of checkpoints (of different processes).
 - Note that processes may not be synchronous.
- A snapshot is consistent if
 - a message that is received in a checkpoint of a process has already been sent by another process (as recorded in its checkpoint)

Checkpointing Parallel Programs - Strategies



- Co-ordinated Checkpointing:
 - a.k.a. Consistent Checkpointing:
 - Program is blocked and all processes checkpoint at the same time.
 - Large overhead and Implementation Difficulties
- Independent Checkpointing:
 - Each process checkpoints independent of other processes:
 - Small overhead and existing techniques for sequential programs can be used
 - Consistency guarantees cannot be provided



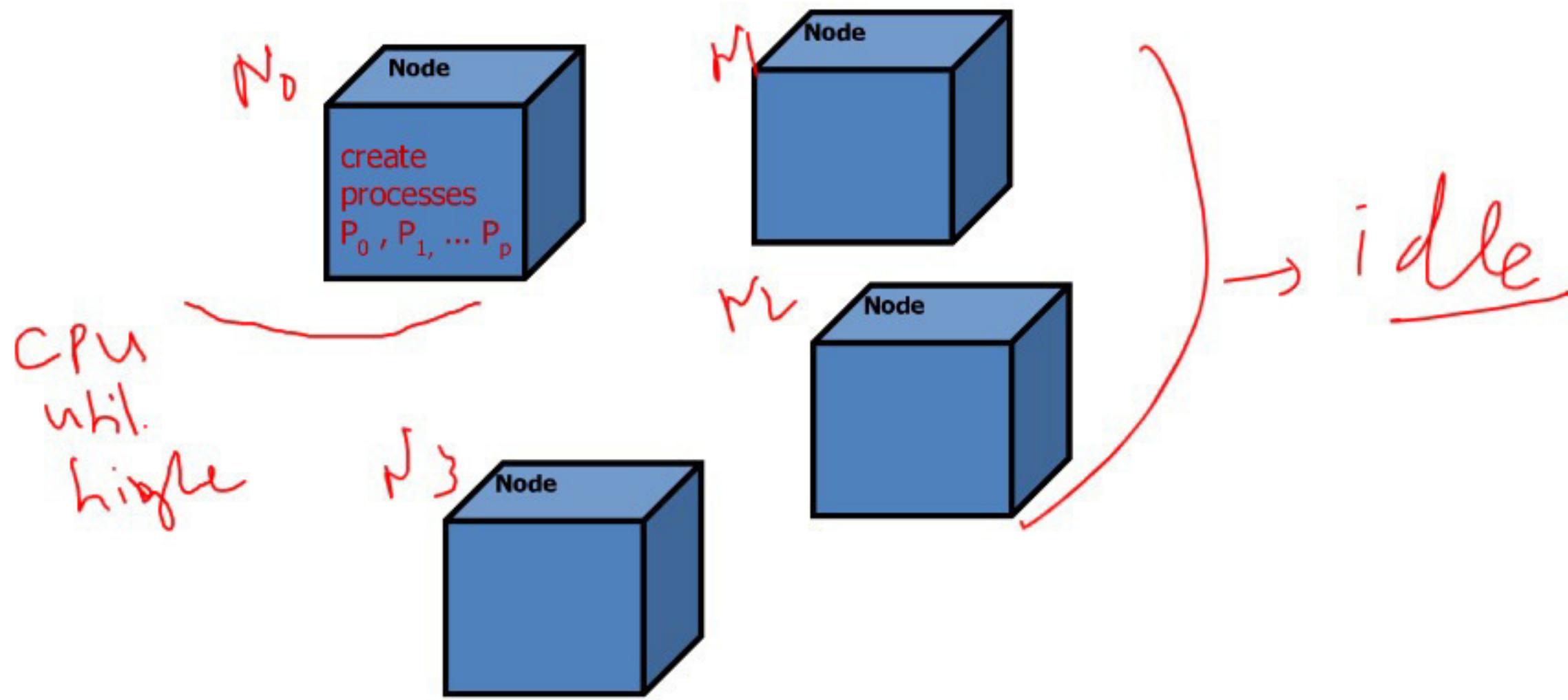
SSI Feature: Process Migration

- Process migration :
 - Transparent remote execution
 - e.g. Sockets are migrated automatically
 - e.g. system calls are intercepted and handled
 - Binary compatibility
 - Migration does not require rewriting or re-linking applications

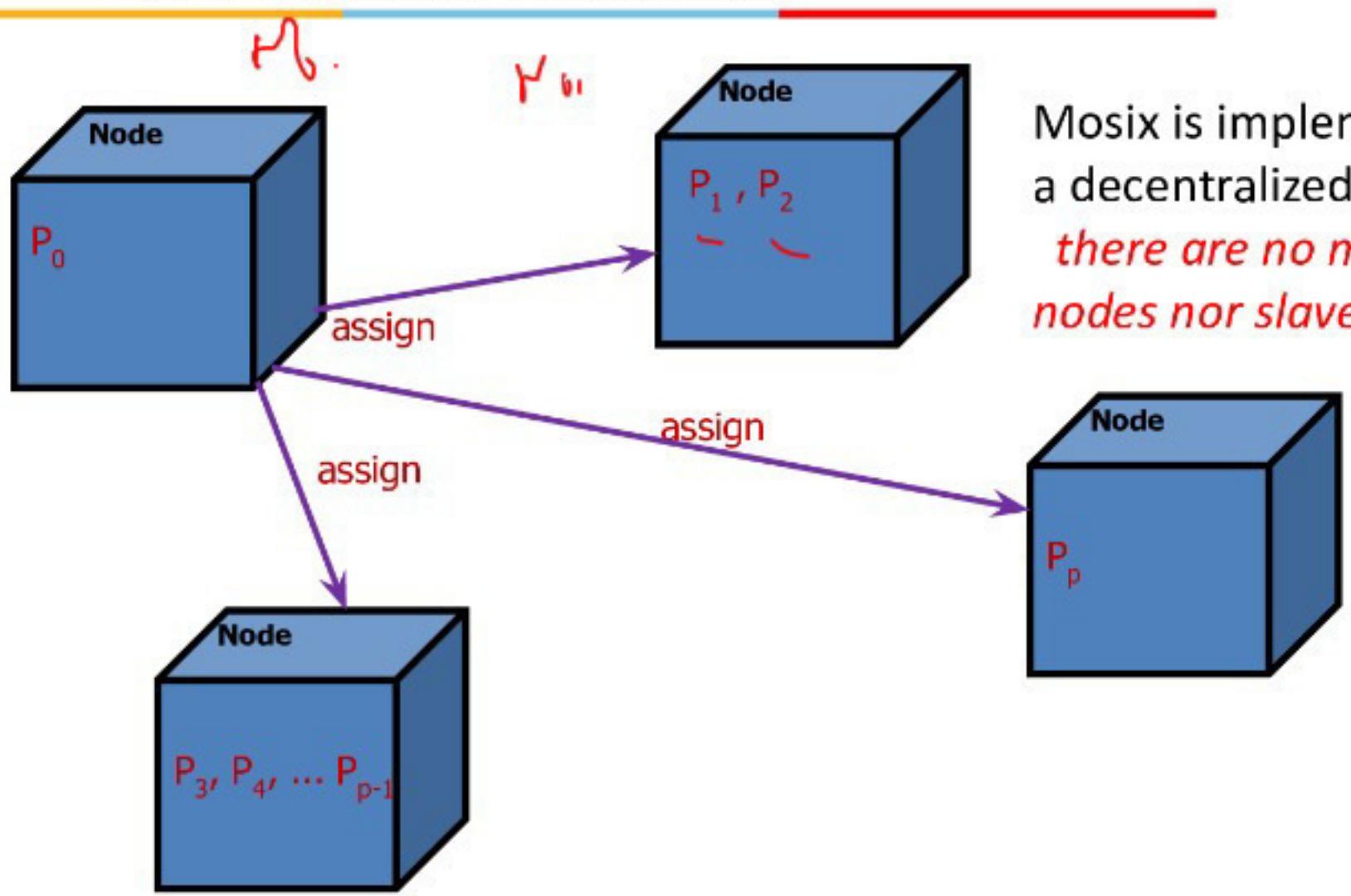
Example

- e.g. Mosix,
 - A cluster OS for nodes running Unix/Linux kernels
 - Consists of:
 - Algorithms for Adaptive Resource Sharing
 - Pre-emptive Process Migration
 - Implemented:
 - at kernel level, as a loadable module
 - without changing the kernel interface

SSI support: Mosix: Work Distribution



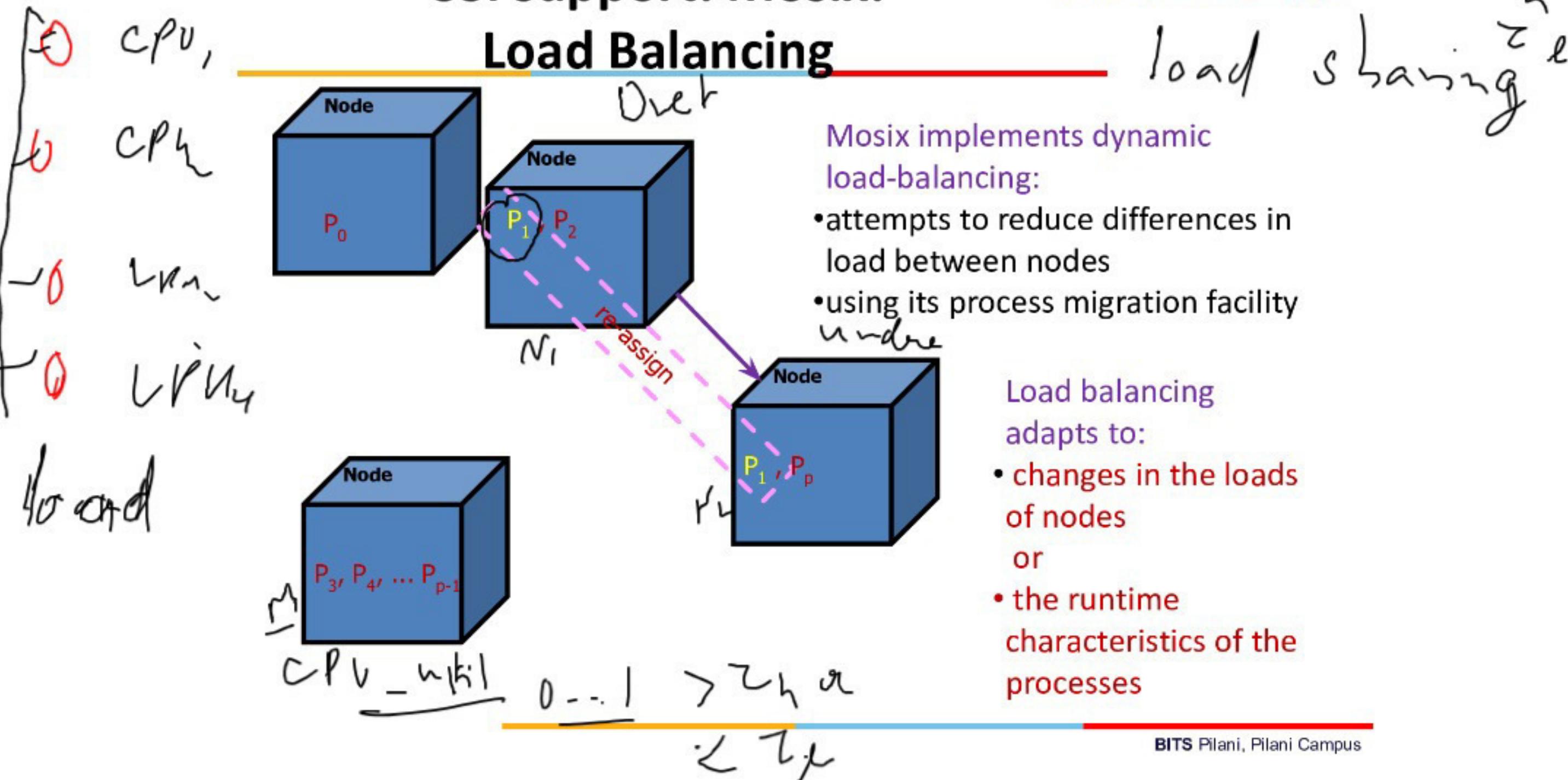
SSI support: Mosix: Work Distribution



Mosix is implemented in a decentralized fashion:
there are no master nodes nor slave nodes.

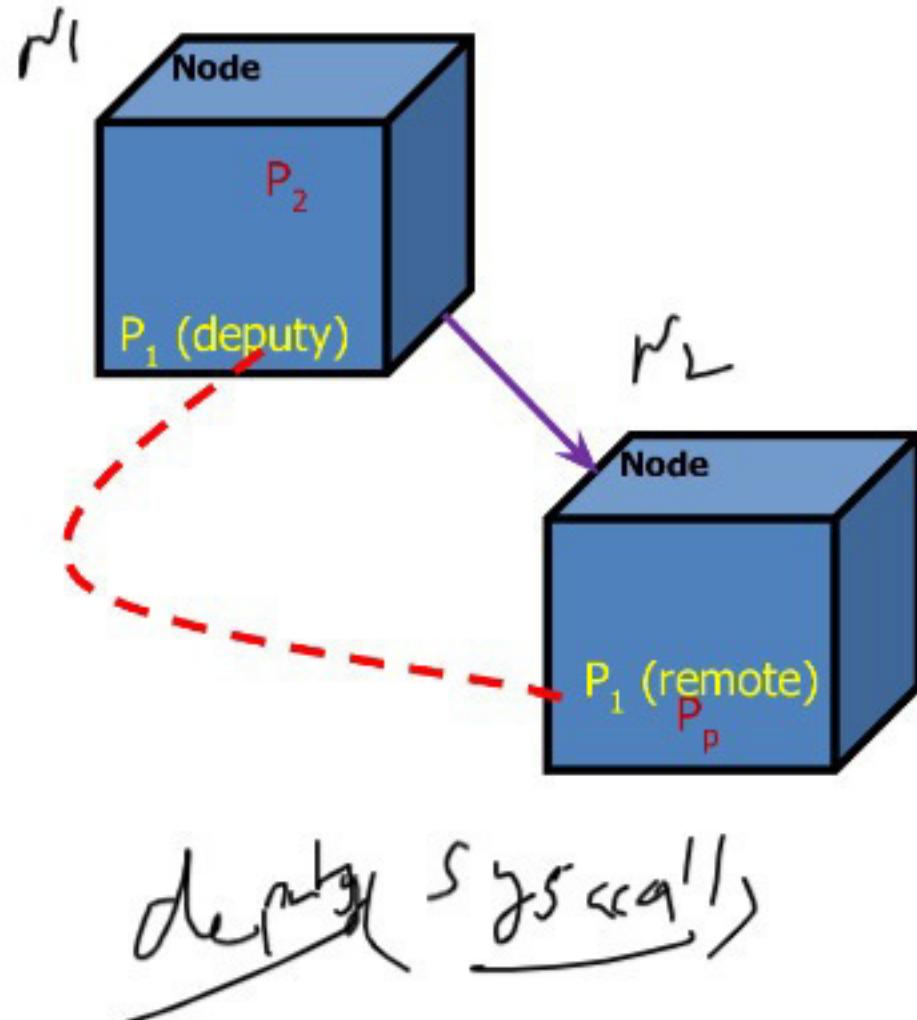
SSI support: Mosix:

Load Balancing



- **Mosix :*Process Migration*:**
 - Each process has a Unique Home Node (UHN)
 - A migrating process is divided into two contexts:
 - user context (referred to as the *remote*):
 - code, stack, data, memory-maps, and registers
 - system context , dependent on UHN (referred to as the *deputy*)
 - resource descriptors, kernel-stack
 - The *deputy* is never migrated
 - only the *remote* is migrated

Mosix: Process Migration: Implementation



Interactions between the system context and the user context:

- are intercepted and forwarded over the network
- e.g. system calls (in remote):
 - if a call is site independent, then it is executed locally by the *remote*
 - if a call is site dependent it is forwarded to the *deputy*



Mosix: Process Migration: Overhead

Migration Overhead:

- Interception and Forwarding

- System calls
- signal delivery
- process wakeup events
- file access operations
- etc.

will have to be intercepted and forwarded resulting in performance overhead.

- Copy Overhead:

- Memory-copies between user memory and kernel memory will now become network-copies

Mosix: Process Migration: Overhead



Techniques for mitigating overheads:

- Migratable Sockets
- Caching: Buffering and Prefetching to avoid copying overhead
- Copy-on-demand: copy only dirty pages



C APP

Performance Attributes



BITS Pilani, Pilani Campus

Performance Model

- (System level) Performance relates to Processor Utilization:

– If processor(s) is (or are) 100% utilized then you cannot hope to do better:

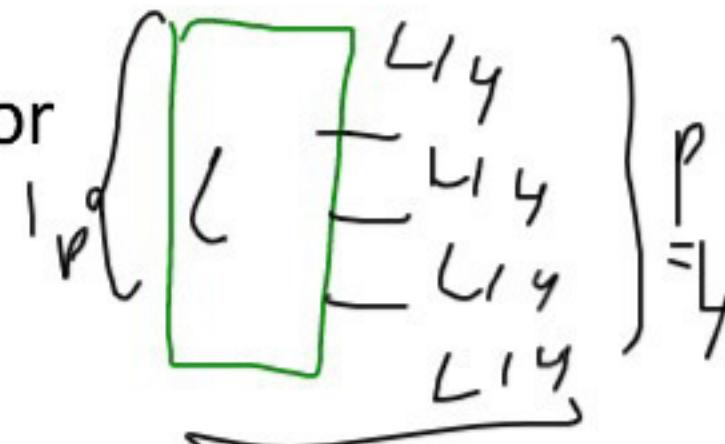
- Assumption: Processor is the fastest and most critical component in the system

- This model can be extended to parallel systems

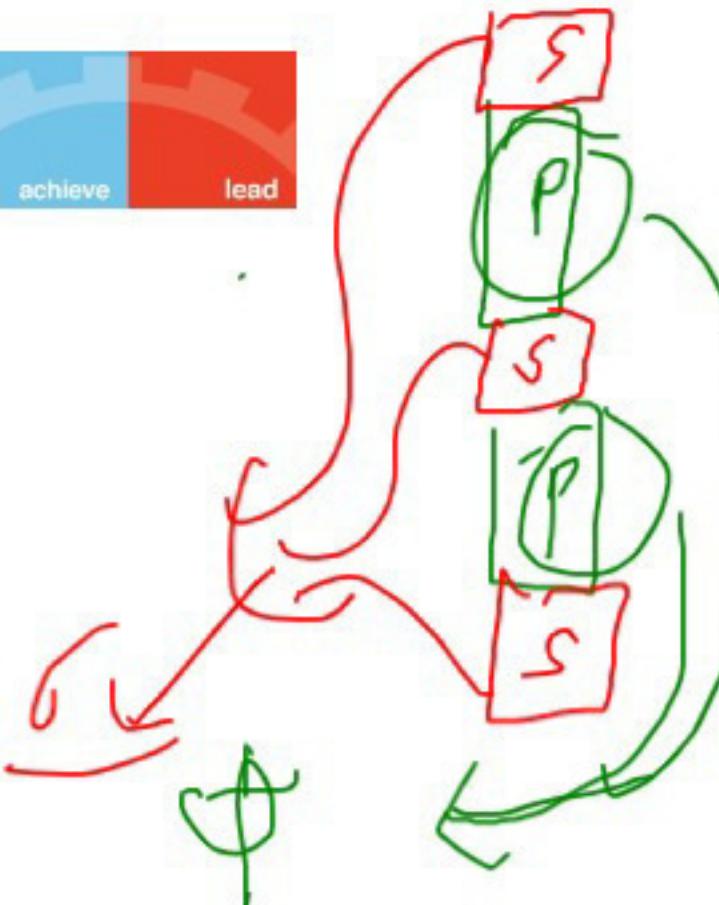
– Speedup = $T_{\text{serial}} / T_{\text{parallel}}$

– Ideal speedup with p processors:

- p , if the program can be fully parallelized (e.g. data parallel execution)



Speedup
~ 4



Basic Notation

- n - problem size
- p - number of processors
- $\sigma(n)$ inherently sequential portion of computation
- $\varphi(n)$ portion of parallelizable computation
- $\kappa(n,p)$ parallelization overhead
- Speedup $\Psi(n,p) = \frac{\text{Seq exec time}}{\text{parallel exec time}}$
- Efficiency $\varepsilon(n,p) = \frac{\text{Seq exec time}}{(\text{Processors used} \times \text{parallel exec time})}$

Fundamental Concepts

- Sequential execution time on a uniprocessor,

$$T(n,1) = \sigma(n) + \varphi(n)$$

- Parallel execution time

$$T(n,p) \geq \sigma(n) + \left(\frac{\varphi(n)}{p}\right) + \kappa(n,p)$$

$$\sigma + \varphi = t$$

$$\frac{\sigma}{P} + \frac{\varphi}{P} = \frac{t}{P}$$

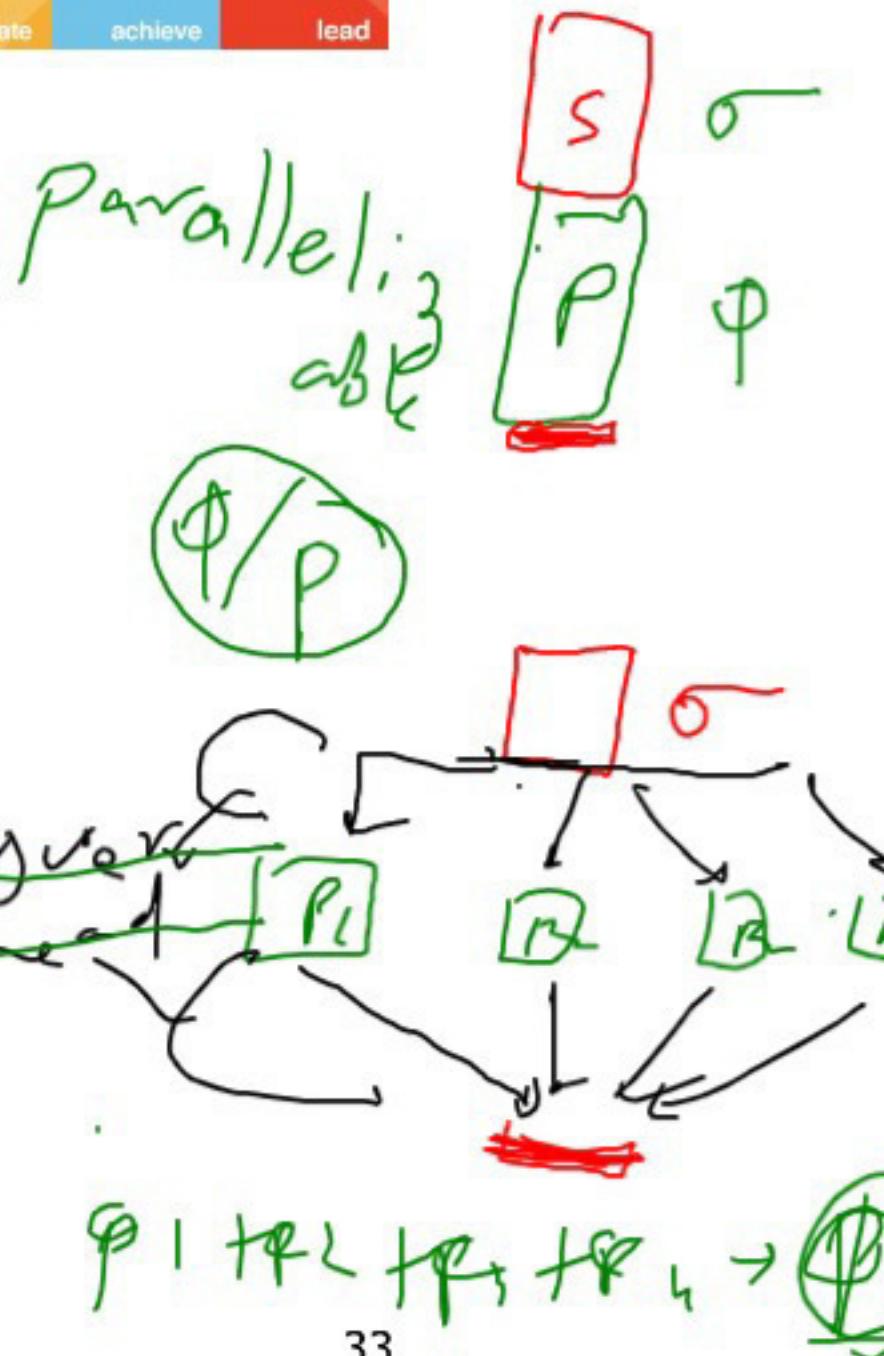
overhead

Speedup $\Psi(n,p) \leq \frac{\sigma(n)+\varphi(n)}{\sigma(n)+\frac{\varphi(n)}{p}+\kappa(n,p)}$

Efficiency $\varepsilon(n,p) \leq \frac{\sigma(n)+\varphi(n)}{p\left(\sigma(n)+\frac{\varphi(n)}{p}+\kappa(n,p)\right)}$

$$\sigma \gg \kappa$$

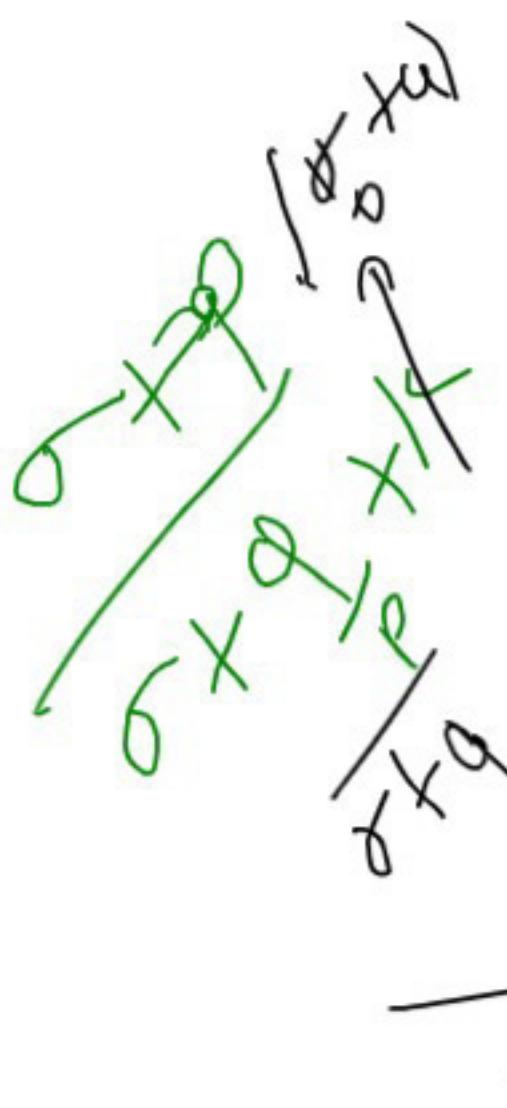
$$\varphi_p \gg \kappa$$



Speedup – Amdahl's Law

σ
 Q

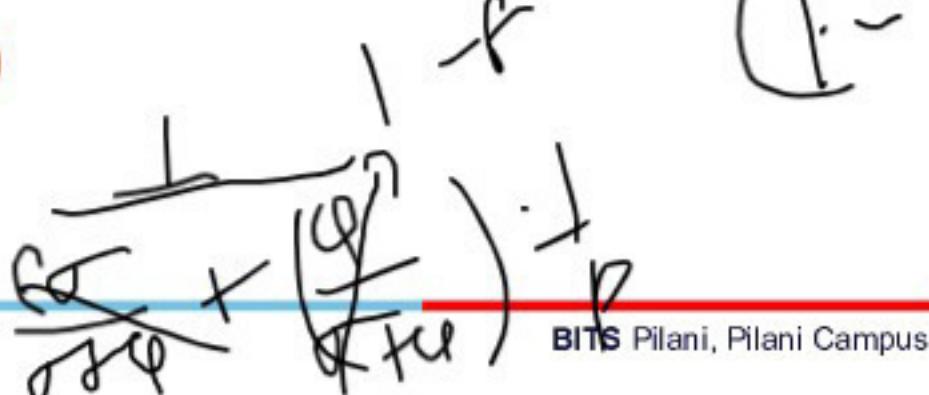
$t_f + t_q$
 $t_f = \alpha \cdot t_q$



- Amdahl derived a basic formulation of speedup in the context of parallel computing:
 - Say, a fraction f of the total work-load cannot be parallelized:
 - i.e. it must be run sequentially
 - And the rest i.e. $1-f$ of the work-load is parallelized on p processors:
 - Then the effective speedup is:

$$\begin{aligned} T_{\text{seq}} / T_{\text{par}} &= T_{\text{seq}} / (f * T_{\text{seq}} + (1-f) * T_{\text{seq}} / p) \\ &= 1 / (f + (1-f) / p) \end{aligned}$$

$$(1-f) = \frac{q}{\sigma + q}$$



Speedup – Amdahl's Law: Implications

- Amdahl's Law :
 - Effective speedup $S(p) = 1/(f + (1-f)/p)$, where p is # processors, f is the fraction that must run sequentially.
- Implications of Amdahl's Law:
 - Example:
 - if **10%** of the program is sequential and **100** processors are used
 - then the effective speedup is $1/(0.1 + 0.9/100) = 1/0.109 = \underline{9.17}$ (approx.)
 - At the limit
 - $\lim_{p \rightarrow \infty} S(p) = 1/(f + 0) = 1/f$
- Assumptions?

A. W.S.

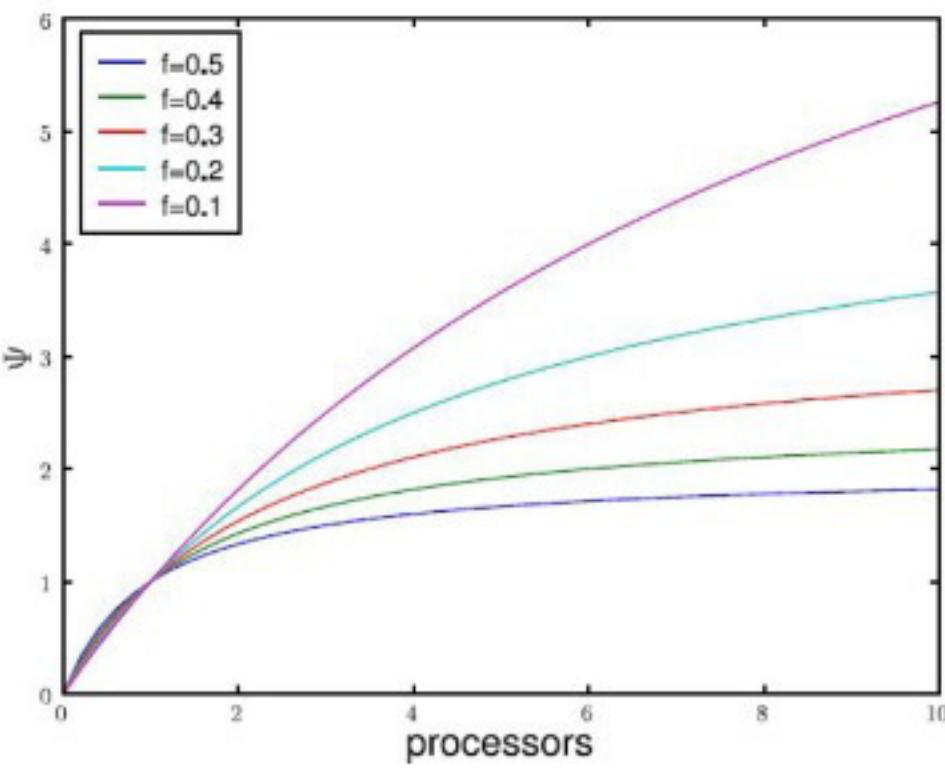
$$\begin{aligned} & 1 \\ & \overline{0.1 + 0.9} \\ & \quad \overbrace{}^{100} \\ & = \overline{0.1} \approx 10 \end{aligned}$$

Amdahl's Law [Example]

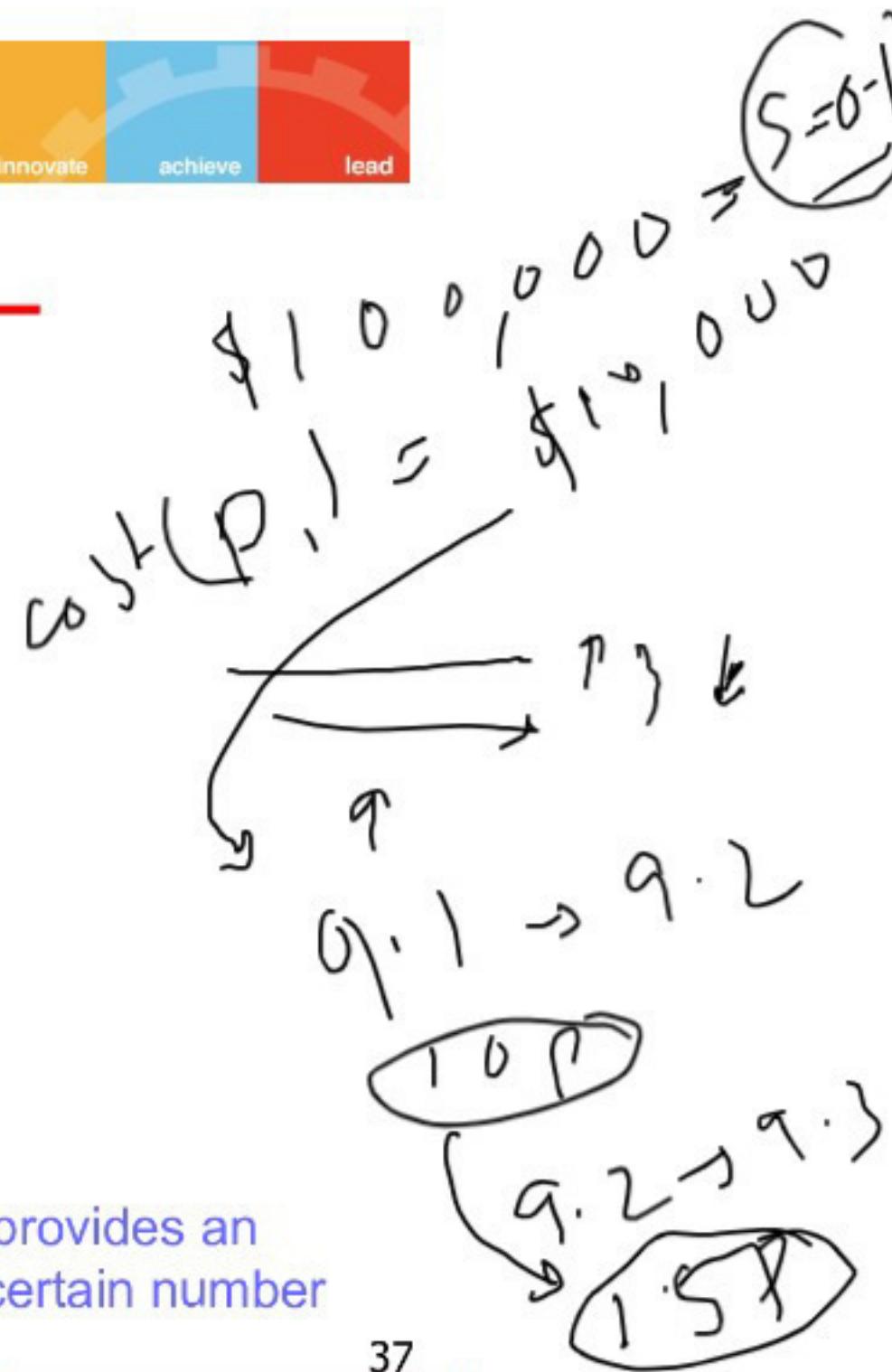
Example 1

- Suppose 70% of a program can be sped up if parallelized and run on multiple CPUs instead of one CPU.
 - If $N = 4$ processors
$$S = 1 / [0.3 + (1 - 0.3) / 4] = 2.105$$
 - Doubling the number of processors to $N = 8$ processors
$$S = 1 / [0.3 + (1 - 0.3) / 8] = 2.581$$
 - Implies that **throwing in more hardware is not necessarily the optimal approach.**
- Example 2
- $\alpha = 0.25$ and so $(1 - 0.25) = 0.75$ then the maximum speedup, $S = 4$ even if one uses hundreds of processors.

Amdahl's Law – Speedup vs f



Amdahl's law assumes that the problem size is fixed. It provides an upper bound on the speedup achievable by applying a certain number of processors.



Amdahl's Law - Outcomes

-
- Parallelization overhead $\kappa(n,p)$ is ignored by Amdahl's law
 - Optimistic estimate of speedup
 - The problem size n is constant for various p values
 - Amdahl's law shows how execution time decreases as number of processors increases.

Super-Linear Speedup – (Possible) Reasons:



- Memory Hierarchy Effect
 - Consider a program with data size N and
 - compare its execution in two environments:
 - i. Single processor with cache of size $D < N$
 - ii. K processors each with cache of size D
 - In the parallel environment:
 - the amount of data that can be accessed from cache has increased K -fold

Super-Linear Speedup – (Possible) Reasons:



- Latency Hiding
 - Even in a uni-processor environment:
 - running two (or more) processes / threads may yield better utilization:
 - When one process / thread is blocked / waiting – say for I/O or for other event(s) – the other thread(s) may utilize the process

Super-Linear Speedup – (Possible) Reasons



- Speculative / Exploratory Execution:
 - Consider a computation of the form (A OR B)
 - What is the expected execution time?
 - What if A and B are executed in parallel?
 - Compare this with randomized choice!
 - Examples:
 - Evaluation of multiple (alternative) clauses in a Prolog program.
 - Evaluation in a backtracking / branch-and-bound algorithm
 - Game Tree Evaluation

Gustafson's Law (Barsis' Law)

- Speedup formulated by Amdahl's Law makes the assumption
 - that one fixes the workload but increases the number of processors:
 - $S(p) = T_s(N) / ((1-f)*T_s(N) + f*T_s(N)/p)$
 - $= 1/((1-f)+f/p)$
 - i.e. p is independent of N
- In practice, f, does not increase with N, for many problems:
 - therefore p should be increased only along with increase in N



Gustafson's Law (Barsis' Law)

- Scaled Speedup $S(p) = ((1-f) + f*p) / ((1-f) + f)$
 - $= p + (1-p)*f$
- What is the difference between this and Amdahl's Law?

-
- Simplest model: Byzantine failures
 - A failed process may do “any thing”
 - other processes are unaffected (by assumption)
 - communication between other processes unaffected
 - More restrictive models:
 - Omission Failures – a faulty process may fail to send messages
 - In timed systems late messages are also treated as failed messages
 - Halting Failures - a failed process does nothing
 - Fail-stop model: other processes know when a process has failed

-
- Restart models:
 - Failure models may include assumptions for “re-starting” after failure:
 - e.g. Halting failure models assume failed computers have a “stable storage” and may be restarted with “stable storage” as it was before failure.
 - Implementers choose a failure model based on the reliability required:
 - Byzantine failure model provides the highest reliability
 - But solutions that tolerate Byzantine failures are costlier

-
- Failure is a common concern in Distributed Computing and
 - hence **Fault-Tolerance** is a required aspect of designing Distributed Systems.
 - **Fault-Tolerance** is a measure of
 - *how well a (distributed) system functions in the presence of faults (of components) in the system*

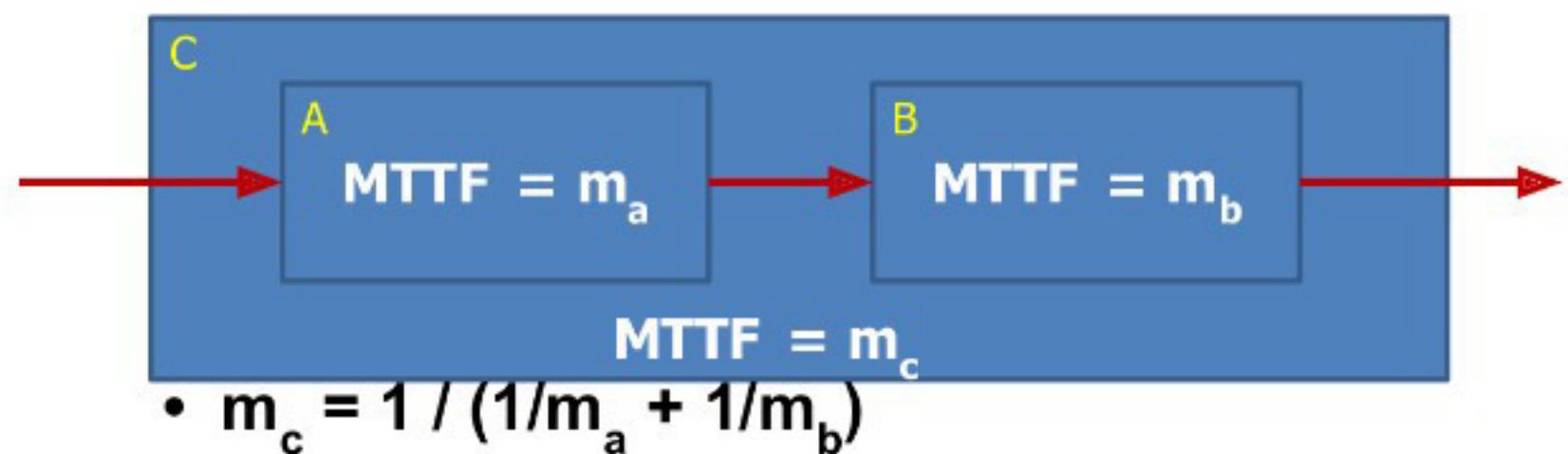


Fault Tolerance

- Tolerance for component faults is measured by two parameters:
 - Reliability:
 - an (inverse) indicator of how often a system fails (i.e. how soon a system is likely to fail).
 - Availability:
 - an indicator of (the fraction of) time a system is available for use
 - note that a system is not available for the duration it remains failed

Reliability

- Reliability of a system is measured by the metric
 - **Mean Time-To-Failure (MTTF)**
- MTTF of a system can be translated to a function of that of each of its components:
 - for example, in a **serial assembly** (where failure of any one component results in a failed system):



Reliability - MTTF

- Generalizing this model, in a serial assembly of Components,
 - $\text{MTTF} = 1 / (\sum_{C \in \text{Components}} (1 / \text{MTTF}_C))$
- Consider instead a ***parallel assembly*** of Components:
 - *a system fails (completely) only when all its components fail.*
- Questions:
 - What is the **MTTF** in such a case?
 - Consider a distributed system made of N separate computing systems:
 - Is it a parallel assembly?
 - If so, argue why it is.
 - If not, argue how to ensure that it is.

Reliability - MTTF

- Special Cases:
 - Consider an assembly with a ***single point of failure***:
 - *a system fails (completely) if this component (SPoF) fails.*
 - Questions:
 1. Can you specify a bound on the **MTTF** in this case?
 2. How do you increase the **MTTF** in this case?
 3. Generalize your answers of 1 and 2 for the case when there are multiple **SPoFs**

Availability

- A system is said to be ***highly available*** if it has
 - a long MTTF and
 - a short MTTR (Mean Time-To-Recover)
- i.e. one can define availability of a system as follows
 - **System Availability = MTTF / (MTTF + MTTR)**

$$A = \frac{m\text{t}\text{f}}{m\text{t}\text{f} + m\text{t}\text{r}}$$

$$SA = R_1 \times R_L$$

✓ Serial Availability, $R_S = \prod_{i=1}^n R_i$



Parallel Availability, $P_f = \prod_{i=1}^n (1 - R_i)$

↳ $R_P = 1 - P_f$



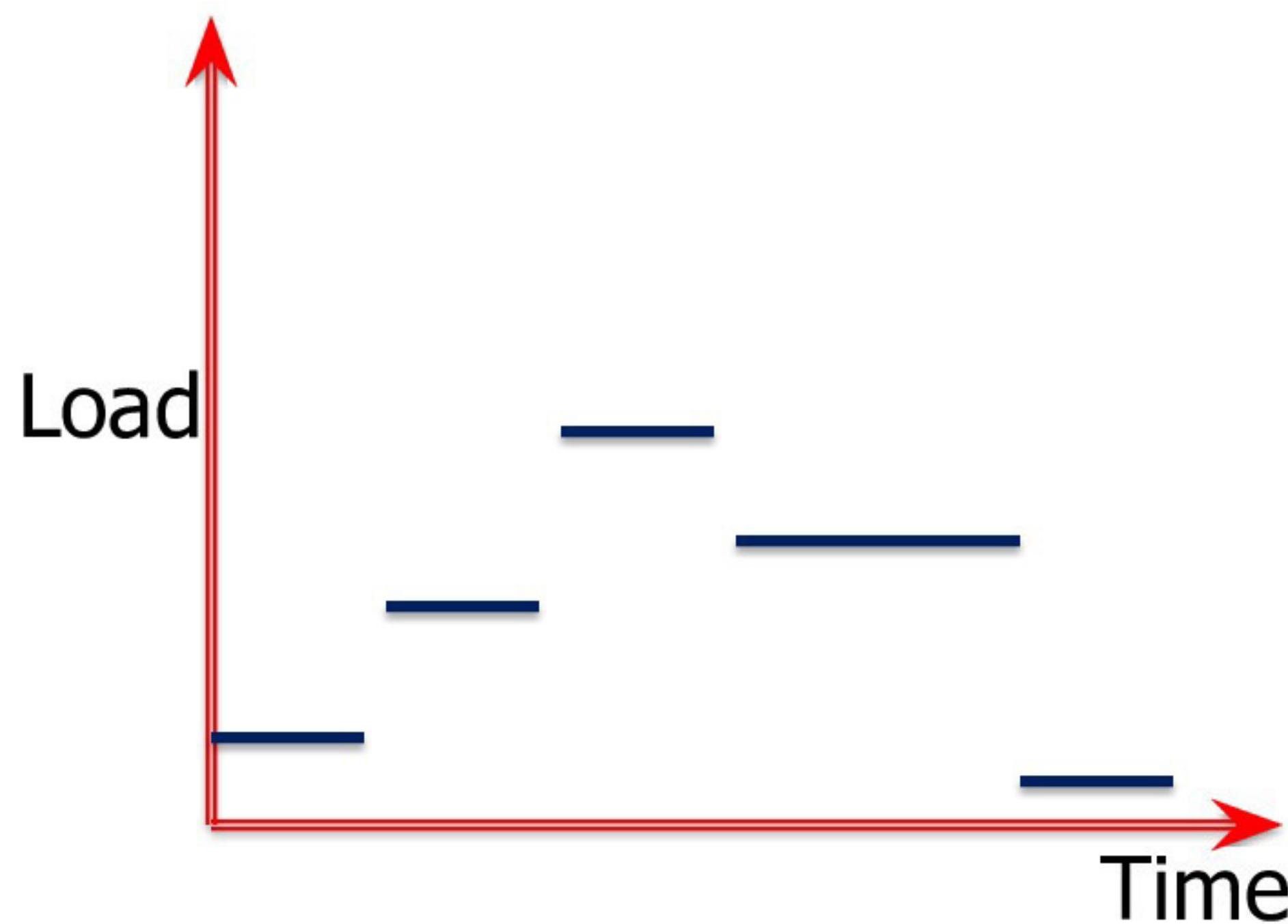
Scalability and Elasticity

- Scalability:
 - Ability to sustain increasing workload with adequate performance.
- Scalability is a pre-requisite for Elasticity
- Elasticity:
 - Extent to which a system is able to adapt to workload changes
 - by automatic (de-)provisioning of resources to closely match the demand

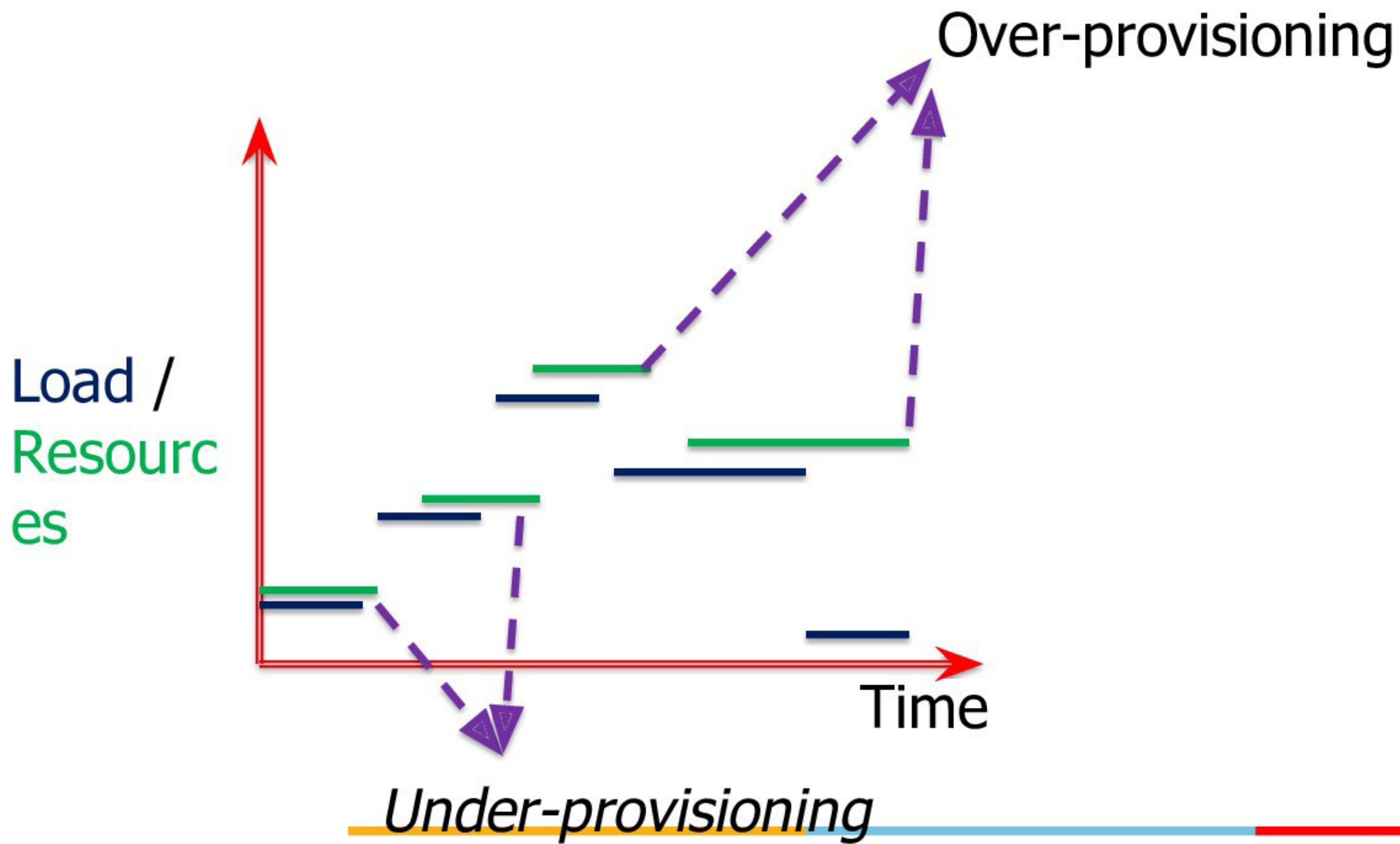
Elasticity

- Elasticity is measured by two parameters
 - Speed of scaling
 - Precision of Scaling
- under two different contexts
 - Scaling-up:
 - From Under-provisioned state to Optimal / Over-provisioned state
 - Scaling-down:
 - From Over-provisioned to Optimal / Under-provisioned state

Elasticity - Varying Demand



Elasticity - Demand vs. Allocation



Speed and Precision of Scaling (Up)

- Speed of scaling is characterized by the
 - Average time taken to switch from an under-provisioned state to an optimal / over-provisioned state:
 - $Av(\delta_{u,t})$
- Precision of scaling is characterized by the
 - Average amount of resources under-provisioned
 - $Av(\delta_{u,r})$

Reference:

Sec. 1.5.1, 1.5.2 of 2.3.3 **Chapter 2**, Kai Hwang, Geoffrey C. Fox, and Jack J. Dongarra, “Distributed and Cloud Computing: From Parallel processing to the Internet of Things”, Morgan Kaufmann, 2012 Elsevier Inc.

Chapter 7, Parallel Programming in C with MPI and Open MP,
Michael Quinn, McGraw Hill, 2004.



Thank you !



BITS Pilani
Pilani Campus

DSECL ZG517 - Systems for Data Analytics

Webinar #1

Murali P

muralip@wilp.bits-pilani.ac.in

[– Thursday – 07:30 PM]

This presentation uses public contents shared by authors of these text books like Tom Mitchell, Christopher Bishop and many others.
Further, works of Professors from BITS are also used freely in preparing this presentation.

Agenda

- Review ↵
- ↘ Locality of Reference
- ↗ Time measurement (in C)
- ↗ OpenMPI
- ↗ OpenMP



BITS Pilani
Pilani Campus



Review: System Attributes for Data Analytics

This presentation uses public contents shared by authors of these text books like Tom Mitchell, Christopher Bishop and many others.
Further, works of Professors from BITS are also used freely in preparing this presentation.

Review

#	Topics
1	<u>Introduction to Data Engineering</u>
1.1	Systems Attributes for Data Analytics - Single System Storage for Data: Structured Data (Relational Databases) , Semi-structured data (Object Stores), Unstructured Data (file systems) Processing: In-memory vs. (from) secondary storage vs. (over the) network Storage Models and Cost: Memory Hierarchy, Access costs, I/O Costs (i.e. number of disk blocks accessed); Locality of Reference: Principle, examples Impact of Latency: Algorithms and data structures that leverage locality, data organization on disk for better locality
1.2	Systems Attributes for Data Analytics - Parallel and Distributed Systems Motivation for Parallel Processing (Size of data and complexity of processing) Storing data in parallel and distributed systems: Shared Memory vs. Message Passing
	Strategies for data access: Partition, Replication, and Messaging Memory Hierarchy in Parallel Systems: Shared memory access and memory contention; shared data access and mutual exclusion Memory Hierarchy in Distributed Systems: In-node vs. over the network latencies, Locality, Communication Cost

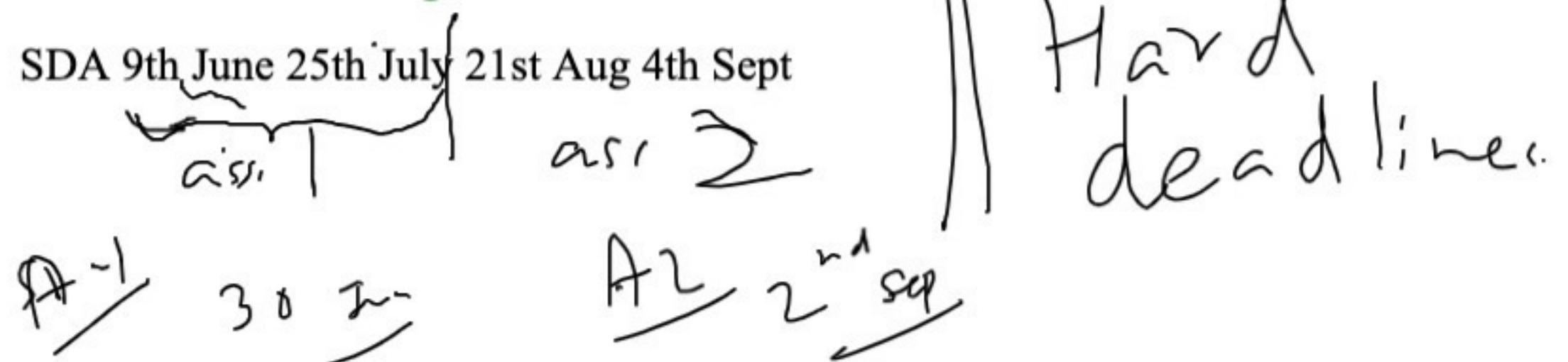
Quiz 1 (S1) Best 1 of 2.

Sunday, June 12, 2022 7:00:00 PM Monday, June 13, 2022 7:00:00 PM Quiz-1 SDA

Quiz 2 (S2)

Sunday, August 21, 2022 7:00:00 PM Monday, August 22, 2022 7:00:00 PM Quiz-2 SDA

Assignment



A.1 $\alpha_1 \rightarrow$ discussion

$\alpha_2 \rightarrow$ LOR

$\alpha_3 \rightarrow$ [open MPI
open MP] This takes
issue it installation time.

PyMPI
PyMPI



BITS Pilani
Pilani Campus



Review: System Attributes for Data Analytics

This presentation uses public contents shared by authors of these text books like Tom Mitchell, Christopher Bishop and many others.
Further, works of Professors from BITS are also used freely in preparing this presentation.



Locality of Reference

- 1> Matrix Multiplication
- 2> Quicksort

Locality o

N
10, 10 0 10 0 0 10 0 6 0

→ 1> Matrix Mi
→ 2> Quicksor

Try matrix ac
Put add cod
m2[N][N],
//using point
Write one
column-m
for some K, m h

execution time

In [8]:

```
# Look at the aggregated times
df.groupby(by = ["Size", "Order"]).mean()
```

Out[8]:

Size	Order	Time	
		Column Major	Row Major
16	Column Major	0.000623	
	Row Major	0.000718	
32	Column Major	0.002730	
	Row Major	0.002396	
64	Column Major	0.003790	
	Row Major	0.004139	
128	Column Major	0.008749	
	Row Major	0.009035	
256	Column Major	0.035518	
	Row Major	0.035212	
512	Column Major	0.139944	
	Row Major	0.138150	
768	Column Major	0.314282	
	Row Major	0.312699	
1024	Column Major	0.564374	
	Row Major	0.558147	
1536	Column Major	1.284221	
	Row Major	1.263556	
2048	Column Major	2.286282	
	Row Major	2.210600	
3072	Column Major	5.180689	
	Row Major	5.003087	
4096	Column Major	9.185274	
	Row Major	8.913544	

ine?

N], int

n2, int N)
th
ethods

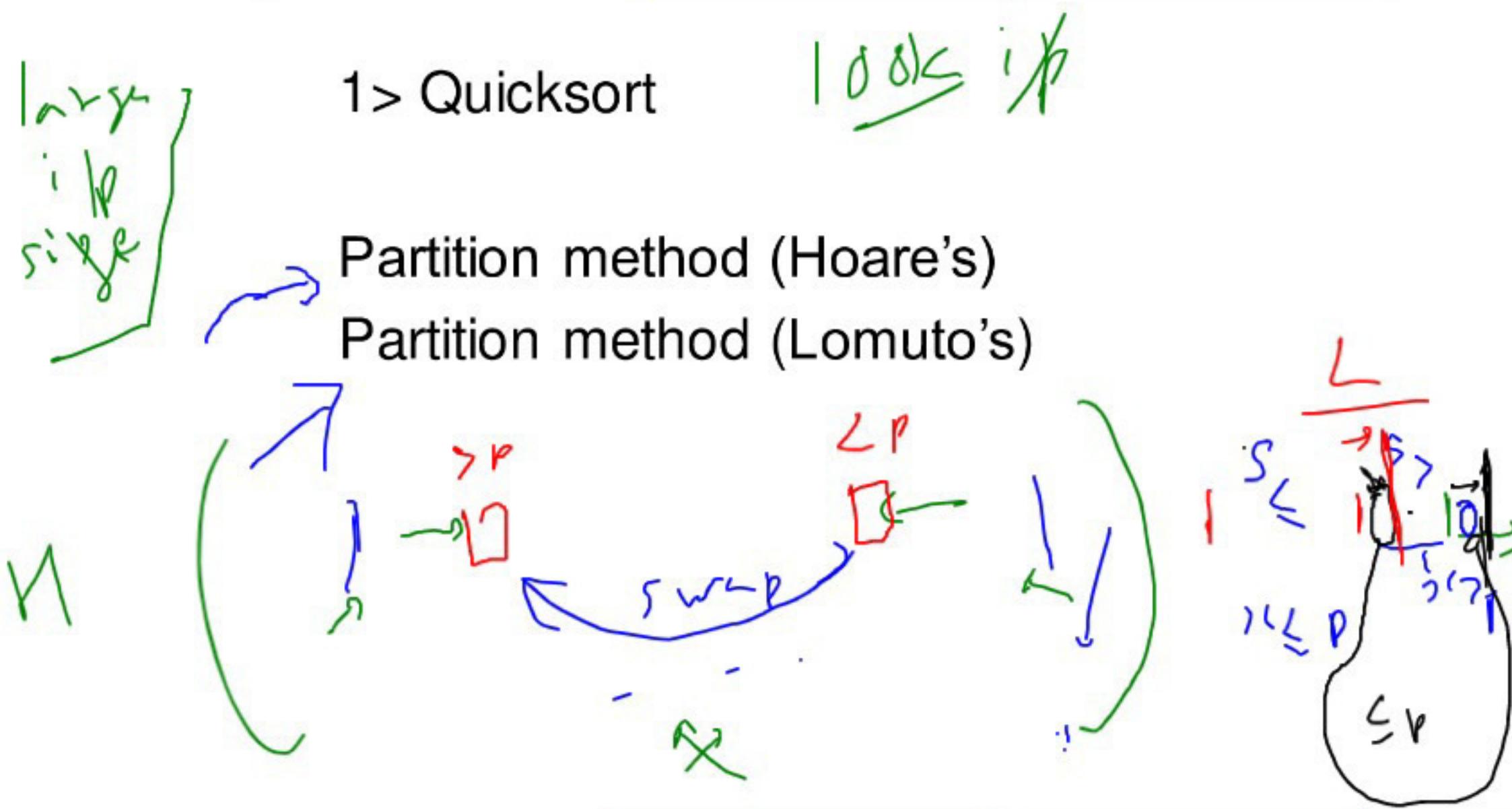
Measuring Time

In C

```
#include <time.h>
.....
start { clock_t t;
          t = clock();
          function_to_time();
stop { t = clock() - t;
       double time_taken =
           ((double)t)/CLOCKS_PER_SEC; // in seconds
.....
```

N

Locality of Reference





The background image shows the iconic yellow clock tower of BITS Pilani, set against a clear blue sky.



BITS Pilani
Pilani Campus

This presentation uses public contents shared by authors of these text books like Tom Mitchell, Christopher Bishop and many others.
Further, works of Professors from BITS are also used freely in preparing this presentation.

Open MPI



Open MPI

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) { // Initialize the MPI
    environmentMPI_Init(NULL, NULL);
    // Find out rank, size
    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    // We are assuming at least 2 processes for this task
    if (world_size < 2)
    {
        fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
        MPI_Abort(MPI_COMM_WORLD, 1);
    }
}
```

Src:https://github.com/mpitutorial/mpitutorial/blob/gh-pages/tutorials/mpi-send-and-receive/code/send_recv.c

↓

```
MPI_Send(&ping_pong_count, 1, MPI_INT, partner_rank, 0, MPI_COMM_WORLD);  
partner_rank, 0, MPI_COMM_WORLD);
```

MPI_Send(data, size, bytes) MPI-BYTE, —
void * data
What orig-d?
data = orig-d

https://www.rookiehpc.com/mpi/docs/mpi_datatype.php

```
int intsToSend[2] = { 12345, 67890 };  
printf("[MPI process %d] I send ints: %d and %d.\n", my_rank, intsToSend[0], intsToSend[1]);  
MPI_Ssend(intsToSend, 2, MPI_INT, RECEIVER, 0, MPI_COMM_WORLD);
```

array ↑ $\left\{ \begin{matrix} 1 & 2 \end{matrix} \right\}$ $\text{int } [2]$



BITS Pilani
Pilani Campus

OpenMP

This presentation uses public contents shared by authors of these text books like Tom Mitchell, Christopher Bishop and many others.
Further, works of Professors from BITS are also used freely in preparing this presentation.

```
#include <stdio.h> #include <omp.h> int main()
{
//serial
printf("\n Welcome! \n"); //pragma omp parallel
{ // parallel thread
printf("Hello world(thread %d)\n",
omp_get_thread_num()); }
//serial
printf("\n Bye! \n"); return 0;
}
```

Quick sort how to partition in parallel ?

Is it possible to parallelize and if yes how?

As a whole I don't think no, only smaller chunks within the algo may be parallelizable?

Just some intuition with a small 7 element array if you can give...

Like [9, 3, 1, 2, 5, 4, 7]

Not all serial code will
parallelizable

$9, 3, 1, 2, 5, 4, 7$ / $P = 5$ || is $AG > \underline{P}$?

\downarrow
 $4, 3, 1, 2 \boxed{5} 9, 7$

quicksort

Mergesort

$T_1 \cup T_2$ $T_3 \cup T_4$





Thank you !



BITS Pilani
Pilani Campus

DSECL ZG517 - SDA

Webinar #2

Murali. P

muralip@wilp.bits-pilani.ac.in

[Tuesday – 07:30 PM]

This presentation uses public contents shared by authors of these text books like Tom Mitchell, Christopher Bishop and many others.
Further, works of Professors from BITS are also used freely in preparing this presentation.



Agenda

- Review
 - A few problems for reliability
-



What we covered so far

Systems Attributes for Data Analytics - Single System

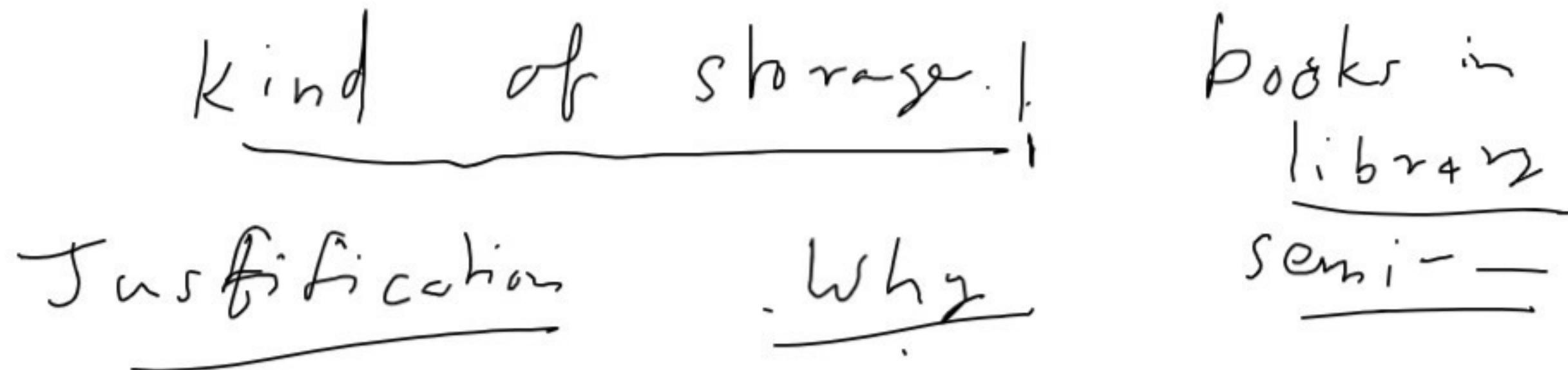
Storage for Data: Structured Data (Relational Databases), Semi-structured data (Object Stores), Unstructured Data (file systems)

Processing: In-memory vs. (from) secondary storage vs. (over the) network

Storage Models and Cost: Memory Hierarchy, Access costs, I/O Costs (i.e. number of disk blocks accessed);

Locality of Reference: Principle, examples

Impact of Latency: Algorithms and data structures that leverage locality, data organization on disk for better locality



For the following applications, which is the most appropriate method to store data: structured, semi-structured or unstructured? How will this data be queried? Justify your answer.

- i. A survey is done to collect food habits of people residing in a city for a food tech company.
- ii. player data in espncricinfo

For a particular stock trading program, we need to compute 50 day, 100 day and 200 day moving averages of share price of a group of stocks. Assuming that a page of memory can only hold data for 10 days and each of the 20 devices can store 10 pages, would you recommend partitioning of data or replication of data or something else? Justify your choice.

What we covered so far

Systems Attributes for Data Analytics - Parallel and Distributed Systems

Motivation for Parallel Processing (Size of data and complexity of processing)

Storing data in parallel and distributed systems: Shared Memory vs. Message Passing

Strategies for data access: Partition, Replication, and Messaging

Memory Hierarchy in Parallel Systems: Shared memory access and memory contention; shared data access and mutual exclusion

Memory Hierarchy in Distributed Systems: In-node vs. over the network latencies, Locality, Communication Cost

What we covered so far

Introduction to Systems Architecture

Parallel Architectures and Programming Models: Flynn's Taxonomy (SIMD, MISD, MIMD) and Parallel Programming (SPMD, MPSD, MPMD)

Parallel Processing Models:, {Data, Task, and Request}-Parallelism; Mapping: Data Parallel - SPMD, Task Parallel - MPMD, Request Parallel - Services/Cloud,
Client-Server vs. Peer-to-Peer models of distributed Computing.

Parallel vs. Distributed Systems: Shared Memory vs. Distributed Memory (i.e. message passing)
Motivation for distributed systems (large size, easy scalability, cost-benefit)

Cluster Computing: Components and Architecture.

Theory part of the following
be covered later in regular
sessions.



Availability

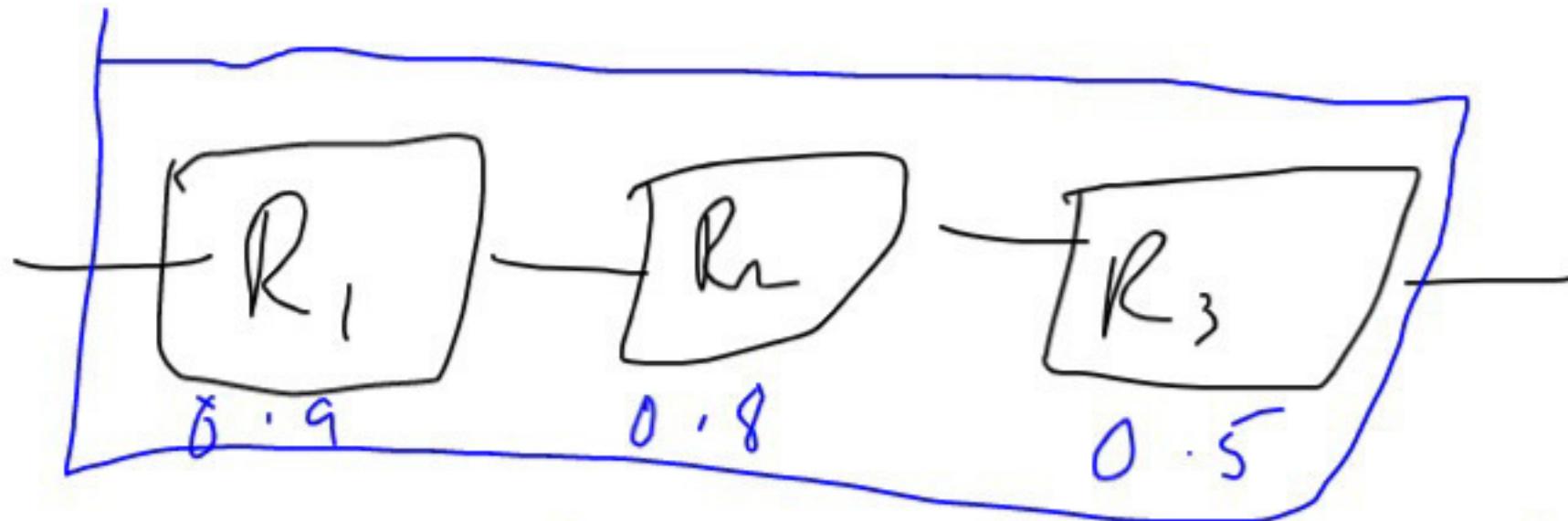
$$\text{System Availability} = \frac{MTTF}{MTTF + MTTR}$$

$$\text{Serial Availability}, R_s = \prod_{i=1}^n R_i$$

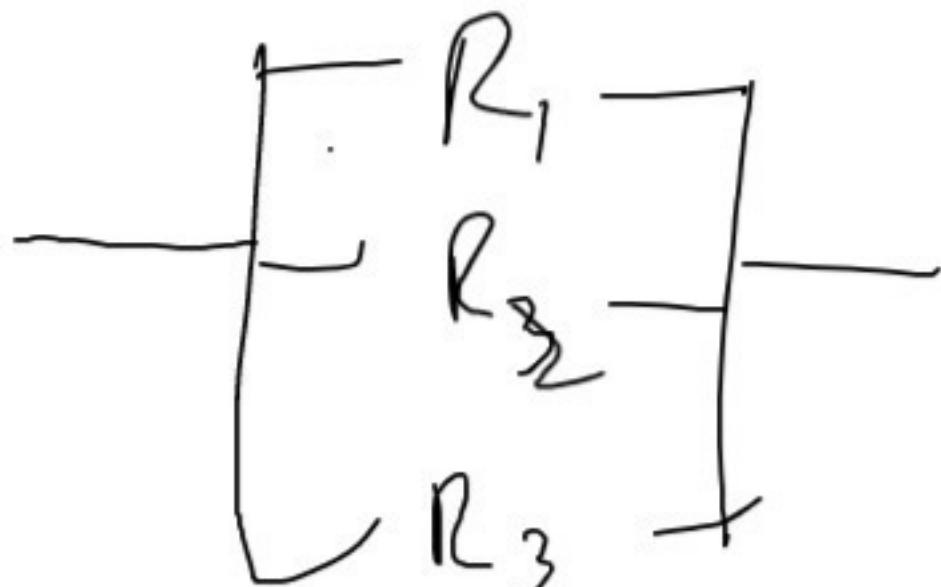
$$\text{Parallel Availability}, P_f = \prod_{i=1}^n (1 - R_i)$$

$$A \leftarrow 1 - P_f$$

To add a diagram here in whiteboard



$$R_{\parallel} = R_1 \times R_2 \times R_3 = 0.9 \times 0.8 \times 0.5 = 0.36$$



$$P_F = \prod(1 - R_i)$$

$$R_A = 1 - P_F$$

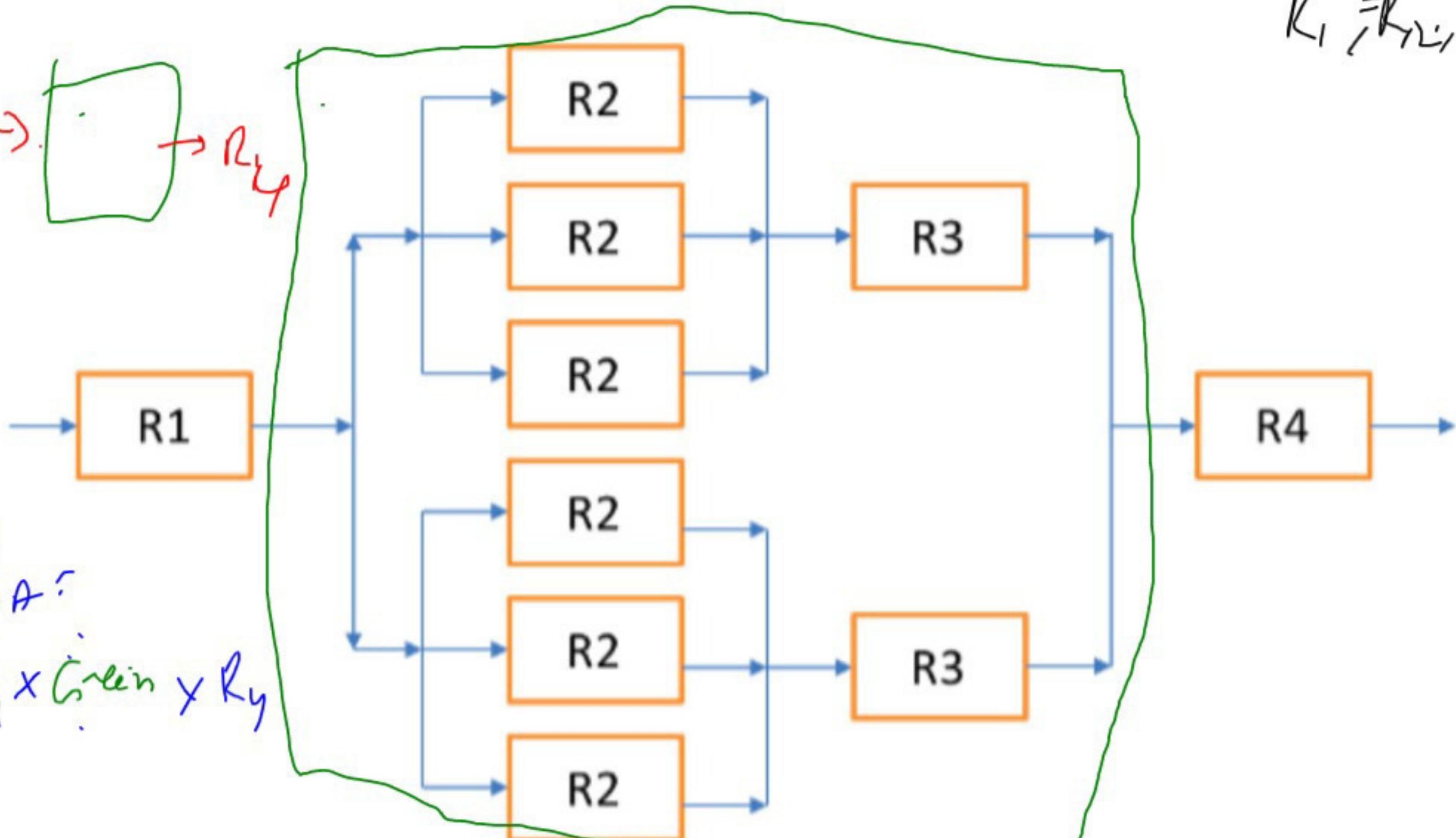
$$= 1 - \left(\frac{(1 - R_1) \times (1 - R_2) \times (1 - R_3)}{1} \right) = 1 - \left(\frac{(1 - 0.9) \times (1 - 0.8) \times (1 - 0.5)}{1} \right) = 1 - (0.1 \times 0.2 \times 0.5) = 1 - 0.01 = 0.99$$

$$\begin{aligned} & 0.9 \\ & 0.8 \\ & 0.5 \\ & 0.1 \times 0.2 \times 0.5 \end{aligned}$$

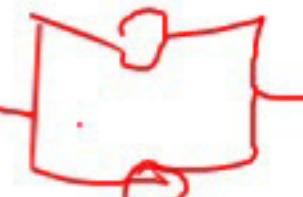
$$= 1 - \left(\begin{array}{l} (1 - 0.9) \times \\ (1 - 0.8) \times \\ (1 - 0.5) \end{array} \right)$$

$$R_1 = R_2, R_3, \ell$$

$$R_1 \rightarrow \boxed{\cdot} \rightarrow R_y$$



$$G =$$

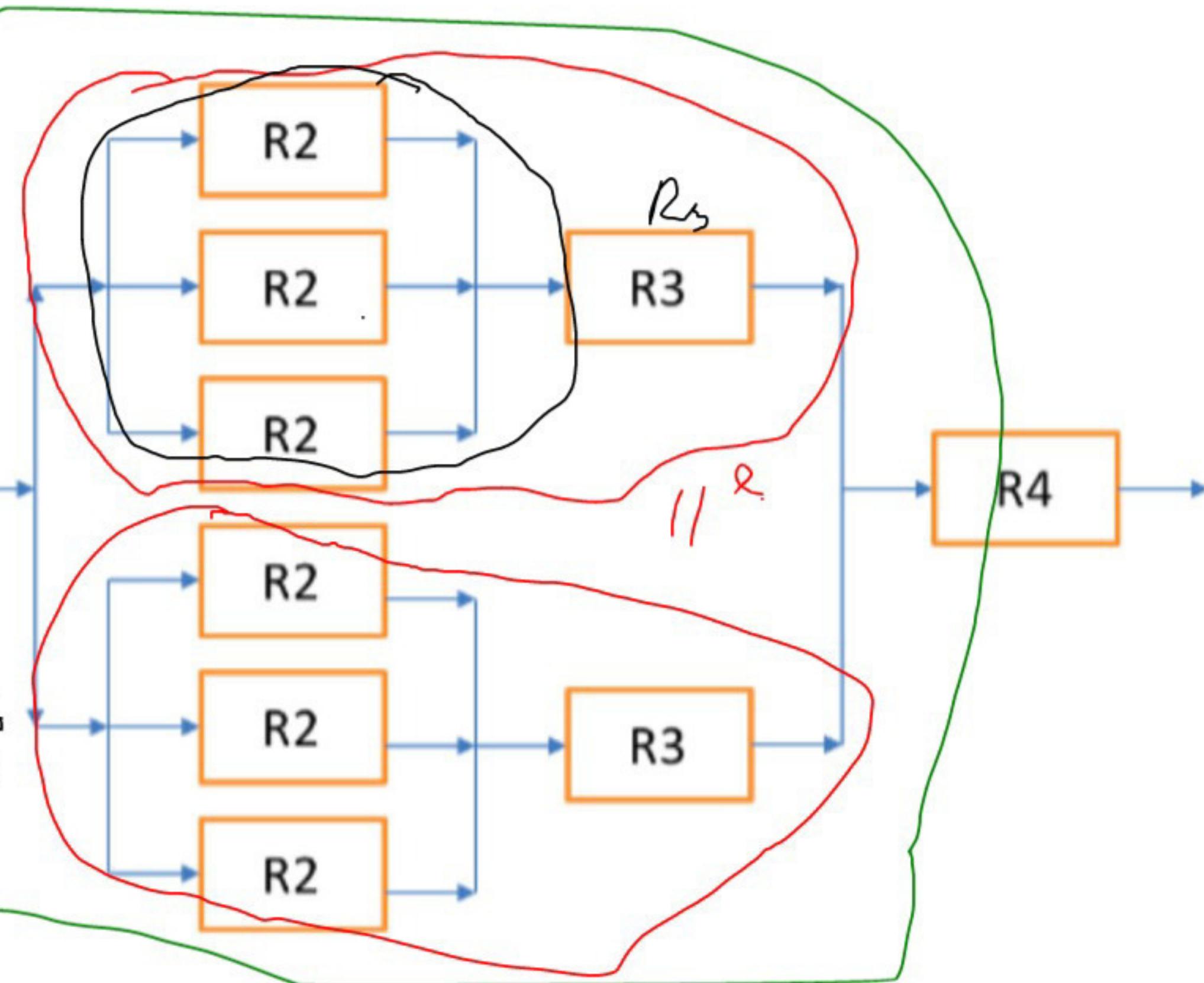


$$= 1 - (1 - R_{ed})^2$$



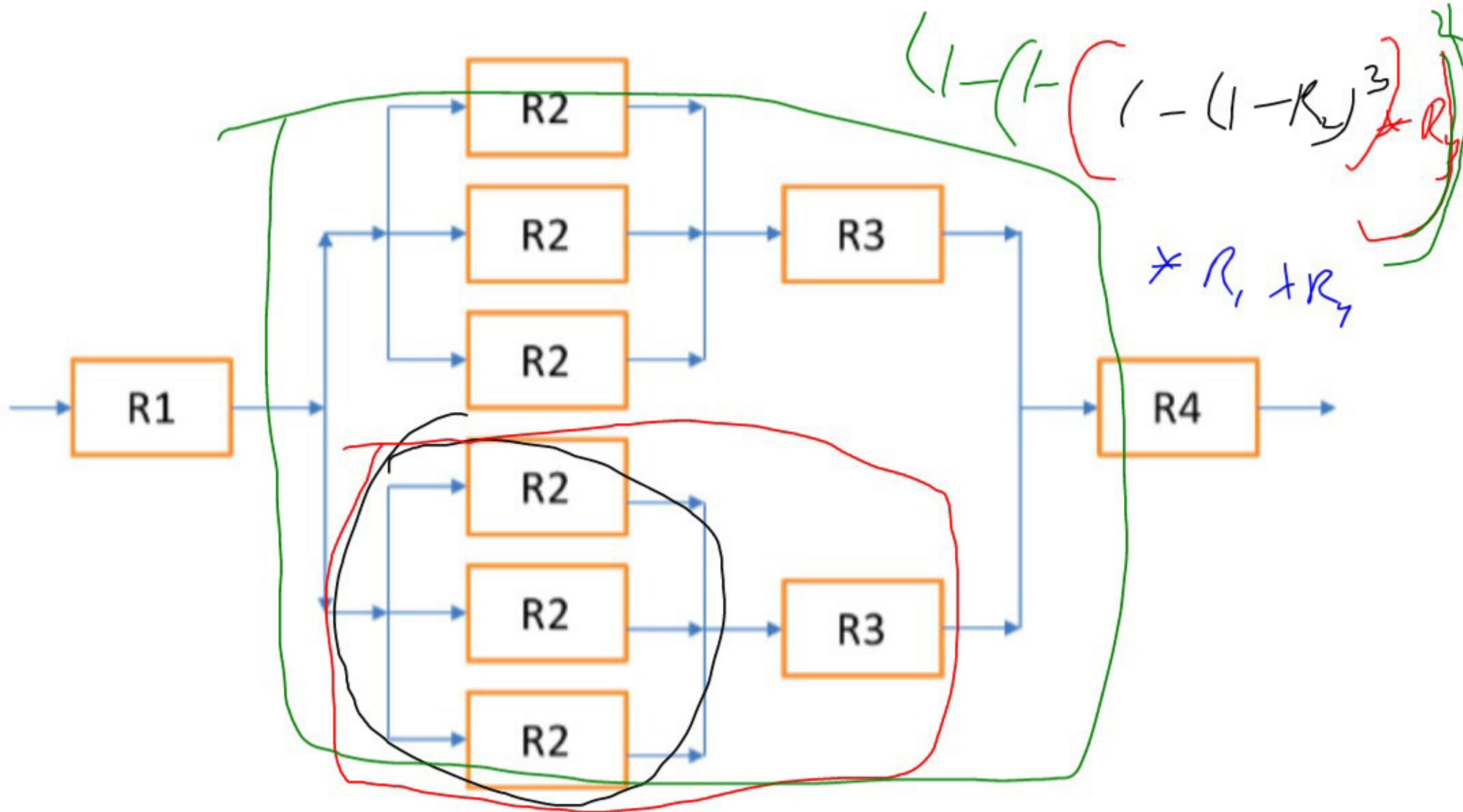
$$R_{ed} = \text{Black} \times R_3$$

$$\text{Black} = 1 - (1 - R_v)^3$$



$$\begin{aligned}
 R_A &= R_1 * \text{Corr}x R_4 \\
 &= R_1 * (1 - (-R_{\text{ad}})^2) * R_4 \\
 &= R_1 * (1 - (1 - [B k_L (L \times R_3)])^2) * R_4
 \end{aligned}$$

$$= R_1 * \left[1 - \left[1 - \left[1 - (1 - R_2)^3 \right] * R_3 \right]^2 \right] * R_4$$



A parallel program executes as shown in figure. MTTF values for R₁, R₂, R₃, R₄ are 4, 6, 8 and 2 respectively; and MTTR values are 5, 10, 10 and 15 respectively.

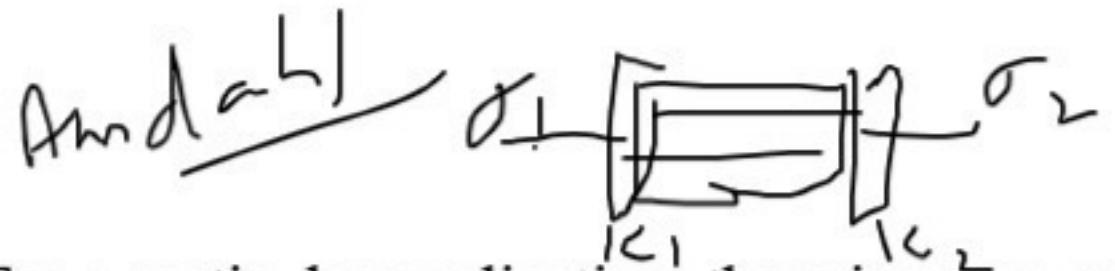
$$A_1 = \frac{m_{PTF}}{m_{PTF} + m_{TTR}}$$

$$R_1 = \frac{4}{4+5} = \frac{4}{9}$$



-
- Amdahl's Law :
 - Effective speedup $S(p) = \frac{1}{(f + (1-f)/p)}$, where p is # processors, f is the fraction that must run sequentially.

$$S(P) = \frac{1}{(f + (1 - f)p)}$$



$$f = ? - \frac{600}{120 + 60 + 30}$$

For a particular application, there is a sequential code that runs for 120 seconds. Afterwards, the execution is carried forward 10 parallel devices for 30 seconds. After that all the 10 devices return result to a server node that collates the results for another 60 seconds. If the whole work was done by a single sequential program, it would have taken 10 minutes.

i. What is the speedup achieved?

ii. If communication overhead is 10 seconds, what would be the actual speedup?

~~No. of Pages~~ 1
~~No. of Questions~~ 4

①

Amdahl's

②

S =

$$T_S = 10 \text{ min}$$

$$= 2.85$$

$$S = \frac{\sigma + k + \alpha_{op}}{\sigma_1 + \sigma_2 + 20 + 15} = \frac{120 + 60 + 30}{120 + 60 + 20 + 15} = 2.85$$

iii. If we could increase the parallel part to 20 nodes that only need to execute for 15 seconds each, what would be the speedup?

$$P = 6 \times 30$$

→ 180

For a particular application, there is a sequential code that runs for 60 seconds. Afterwards, the execution is carried forward 6 parallel devices for 30 seconds. After that all the 10 devices return result to a server node that collates the results for another 60 seconds. If the whole work was done by a single sequential program, it would have taken 5 minutes.

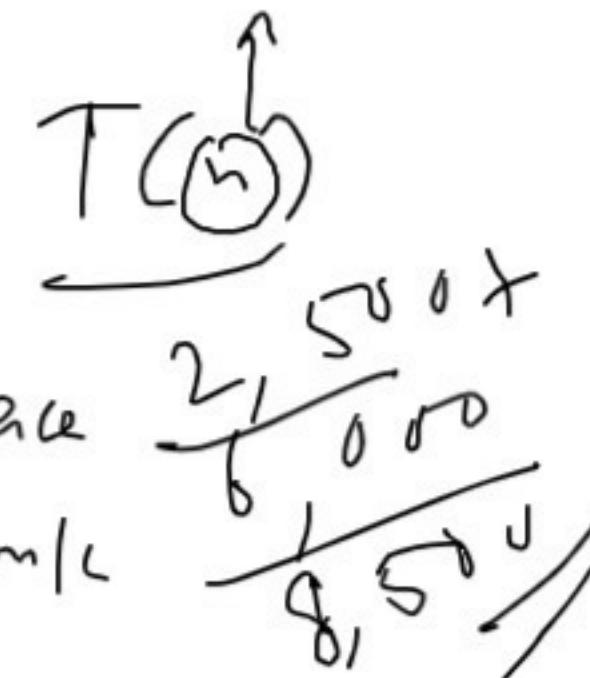
→ i. What is the speedup achieved?

6 ii. If the problem size is scaled up, what would be the speedup that can be achieved?

iii. If 1 minute of execution costs \$1000 and each additional device costs \$1000, how much cost benefit is achieved if we use 8 devices instead of 6 devices?

$$\frac{T_p}{T_s} = \frac{6+30}{6} = 2 \cdot 5 \text{ min}$$

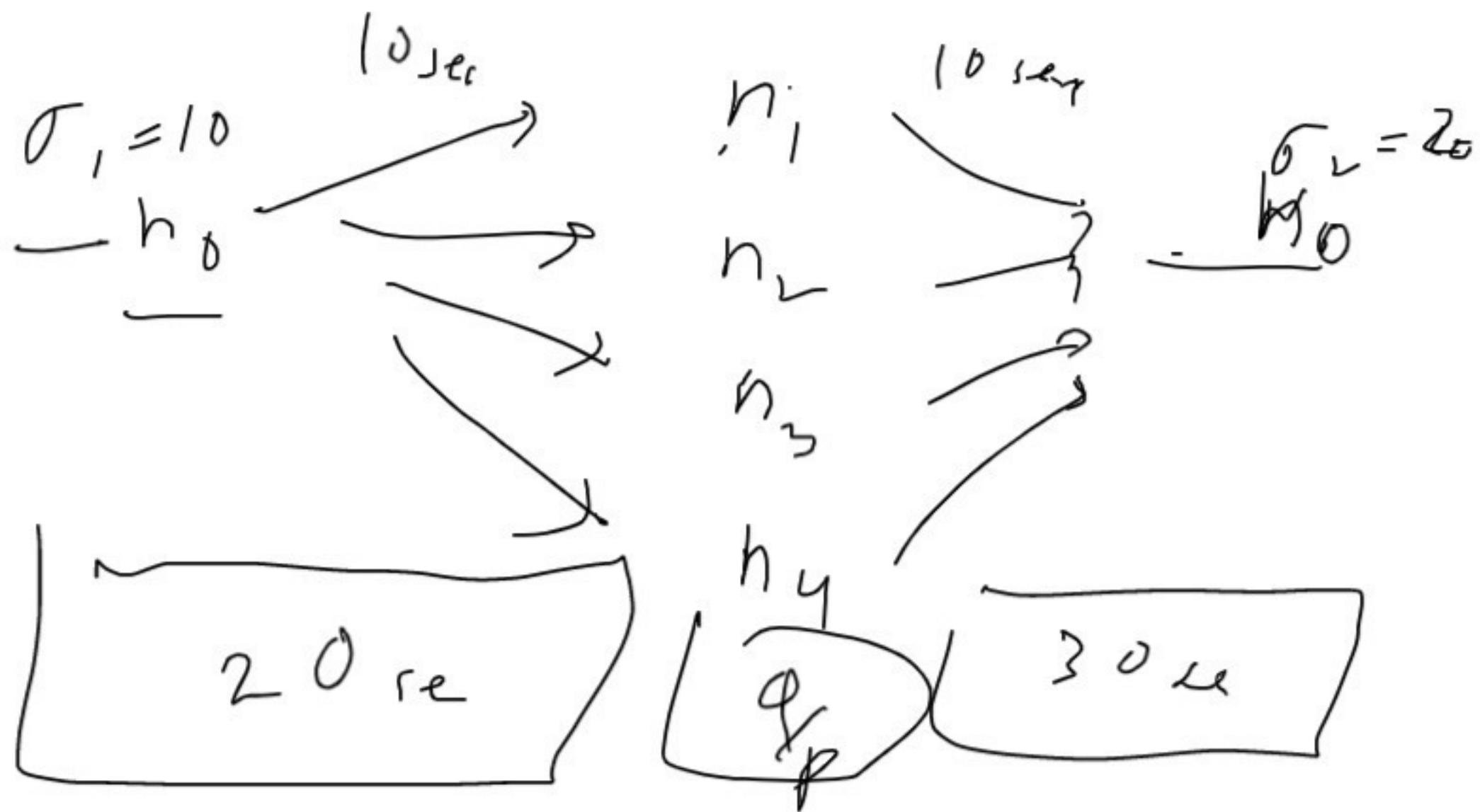
$$S = 2$$



$$\begin{array}{r}
 \$5000 + \$2,500 \\
 \hline
 \$6000
 \end{array}
 \approx 2,400 \quad \frac{P}{P} = \frac{180}{\$} = 22.5$$

$\approx 2,400$
 $\approx 2,400$
 $\boxed{1,400}$

(1)





Thank you !