Distributed Algorithms

➔ **Some of the machine models for distributed computing** :

- ◆ Network Model
- ◆ Distributed Memory Model
- ◆ Message-Passing Model
- ◆ Communicating Sequential Processes Model

➔ **Message-Passing Model**

- ◆ Every node has a processor and a private memory(memory can't be accessed by any other node). The communication happens by passing of messages between different nodes, through the network.
- ◆ Rules :
  - Connected :
    - The network is connected i.e. there always exists a path from a node to another node in the network.
  - Bidirectional Links :
    - The transfer of messages can happen in both directions over the links.
  - <= 1 send + 1 receive at a time.
  - The cost to send and receive n-words :
    - $T(n) = \alpha + \beta n$
    - α = latency, units of time, it is a fixed cost no matter how large the message. It represents a fixed overhead that occurs while trying to send any message, and hence, it's independent of the message size.
    - β = inverse bandwidth, units of time/word, n = number of words.

- K-way congestion reduces bandwidth :
  - k= the number of messages competing for the same link
  - $T_{msg}(n) = \alpha + \beta nk$
  - This means that two messages (going in the same direction) that are competing for the same link will have a time that is closer to the serial time, than the parallel time for the two messages.

➔ **<u>There are 3 types of costs involved</u>** :

- ◆ Cost of Communication(α)
- ◆ Cost of message preparation(β)
- ◆ Cost of Computation(τ)
- ◆ $\tau << \beta << \alpha$

➔ Computation requires much less time than communication, so try to have as little communication as possible.

➔ Also, you can send ~1000 bytes in the same amount of time that it takes to prepare a message. It is better to send fewer larger messages than frequent short ones.

➔ **<u>Collective Operations</u>** :

- ◆ **All-to-one Reduce** :
  - All nodes participate to produce a final result on one node.
  - Consider a linked list, where the nodes contain numbers. We need to add these values and store the resultant in a single node. This is an example of All-to-one reduce, as all nodes are participating to produce a final result in one final node.
  - Have all odd indexed nodes send their data to the lower even indexed nodes. Only one word is sent in each

message, so the communication time is α + β. Repeat until only one result is left. The communication time is: **(# of steps) * (α + β) i.e (logP * (α + β))**.

- sendAsync(buf[1:n],dest) : This function is called every time a send operation is performed. It returns a handle. When the handle is returned, it doesn't mean that the message has been sent. It only means that a send request has been registered with the system.
- recvAsync(buf[1:n],source) : This function is called every time a receive operation is performed. It also returns a handle, but when a handle is returned, it doesn't mean that the data is available.
- To check the completion of the functions, we use handles. Wait is a blocking operation that takes these handles as arguments and pauses until all the corresponding operations are complete. The wait(*) operation waits for all the operations to finish.
- All-to-One Reduce Pseudocode :
    - let s = local value
    - bitmask ← 1
    - while bitmask < P do
    - PARTNER ← RANK ^ bitmask
    - if RANK & bitmask then
    - sendAsync (s → PARTNER)
    - wait(*)
    - break //one sent, the process can drop out
    - else if (PARTNER < P)
    - recvAsync(t ← PARTNER)
    - wait(*)
    - S ← S + t
    - bitmask ← (bitmask << 1)
    - if RANK = 0 then print(s)

- So, initially, all the nodes have some data. Performing a reduce, combines all the data to a single node. The node that stores the data is called the root.

◆ **One-to-all Broadcast** :

- One node has all the data, which it wants to broadcast to all other nodes. Since running broadcast is similar to running reduce in reverse, it is also called the “dual” of reduce. It has the same algorithm as that of reduce in reverse.

◆ **Scatter** :

- One node has all the data, which is scattered into pieces and sent to all the other nodes.

◆ **Gather** :

- Gather is the dual of Scatter. In this case, every node has a piece of data, which is finally collected into one node.

◆ **All-Gather** :

- Each process has a piece of data. The data is gathered and every process gets a copy of all the data. Equivalent to saying: Gather and broadcast.

◆ **Reduce-Scatter** :

- Each node contains a vector of data. Then, a reduce operation is performed and one node gets all the data. This data is then scattered. Equivalent to saying: Reduce and Scatter.

◆ Efficient Implementation of Gather and Scatter is done by divide and conquer technique.

➔ **<u>Parameters for Analysis</u>**:

◆ Based on Node and Links:

- Link bandwidth - Amount of data that can be put on the link per unit time.
- Latency

- Message preparation cost
- Computation cost

◆ Based on Topology:
- Number of Links - Gives upper bound on # messages that can be present in the network at a given instance of time.
- Diameter - Gives upper bound on # hops required
- Bisection Width - Min number of links required to be broken so as to divide the network into two equal parts.
- Bisection Bandwidth - Bandwidth available across bisection width.
- Congestion - Gives the max serialization that can occur on the link.

◆ Based on the algorithm:
- Time Complexity
- Space Complexity

➔ **<u>Network Topologies analyzed</u>**:

◆ Linear
◆ 2D Mesh
◆ kD Mesh
◆ Hypercube:
- 3D
- log(p) D

◆ Torus:
- Ring 1D
- Doughnut 2D
- kD Torus

◆ Fully Connected Graph
◆ Tree
◆ Fat Tree (Variation)

➔ **<u>Logical to Physical Mapping/Embedding</u>**:

- ◆ A logical network is the one we have used for the algorithm. The physical network is the one which is available to us as the hardware.
- ◆ Many-a-times it is not possible to create the required physical topology that caters to our algorithm as it could be expensive. For example, a torus (logical) can be mapped to ring (physical) topology which is easier to create. When more sophisticated networks are mapped to simpler networks congestion occurs.

➔ **<u>Congestion</u>**:

- ◆ Max number of messages that compete for the same link.
- ◆ Calculating congestion:
  B_L : Bisection width of the logical network
  B_P : Bisection width of the physical network
  Congestion = $\Omega(B_L/B_p)$

➔ **<u>Topology-based Parameters</u>**:
Assume p nodes (processors).

Topology	**Parameter**		
	# Links	*Diameter*	*Bisection Width*
Linear	$p-1$	$p-1$	1
2D Mesh	$2p-2\sqrt{p}$	$2(\sqrt{p}-1)$	$\sqrt{p}$
kD Mesh	$kp-kp^{(\frac{k-1}{k})}$	$kp^{\frac{1}{k}}$	$p^{(\frac{k-1}{k})}$
Ring	p	$p/2$	2
2D Torus (Doughnut)	$2p$	$\sqrt{p}-1$	$2\sqrt{p}$
kD Torus	kp	$\frac{k}{2}(p^{\frac{1}{k}}-1)$	$2p^{(\frac{k-1}{k})}$

$log(p)$ D Hypercube	$\frac{p\ log(p)}{2}$	$log(p)$	$p/2$
Binary Tree	$p-1$	$2log(p)$	1
Fully Connected	$\frac{p\ (p-1)}{2}$	1	$\frac{p^2}{4}$

➔ **Algorithmic Complexities based on various Topologies** :

Algorithm	Topology	Time Complexity
One-to-all Broadcast All-to-one Reduction	*Linear*	$(\alpha + \beta m)\ (p)$
	Ring	$(\alpha + \beta m)\ log(p)$
	Mesh	$(\alpha + \beta m)\ log(p)$
	Torus (2-D)	$(\alpha + \beta m)\ log(p)$
	Hypercube	$(\alpha + \beta m)\ log(p)$
	Tree	$(\alpha + \beta m)\ p$
	FCG	$(\alpha + \beta m)\ log(p)$
All-to-all Broadcast All-to-all Reduction	*Linear*	$(\alpha + \beta m)\ (p-1)$
	Ring	$(\alpha + \beta m)\ (p-1)$
	Mesh	$2\alpha(\sqrt{p}-1) + \beta m(p-1)$
	Torus (2-D)	$2\alpha(\sqrt{p}-1) + \beta m(p-1)$
	Hypercube	$\alpha log(p) + \beta m(p-1)$
	Tree	$(\alpha + \beta m)\ (p-1)$

	FCG	$(\alpha + \beta m)(p-1)$
All Reduce	*Linear*	$(\alpha + \beta m)\, log(p)$
	Ring	$(\alpha + \beta m)\, log(p)$
	Mesh	$(\alpha + \beta m)\, log(p)$
	Torus (2-D)	$(\alpha + \beta m)\, log(p)$
	Hypercube	$(\alpha + \beta m)\, log(p)$
	Tree	$(\alpha + \beta m)\, p$
	FCG	$(\alpha + \beta m)\, log(p)$
Scatter Gather	*Linear*	$\alpha log(p) + \beta m(p-1)$
	Ring	$\alpha log(p) + \beta m(p-1)$
	Mesh	$\alpha log(p) + \beta m(p-1)$
	Torus (2-D)	$\alpha log(p) + \beta m(p-1)$
	Hypercube	$\alpha log(p) + \beta m(p-1)$
	Tree	$\alpha log(p) + \beta m(p-1)$
	FCG	$\alpha log(p) + \beta m(p-1)$
Scatter-all Gather-all	*Linear*	$(\alpha + \beta m\frac{p}{2})(p-1)$
	Ring	$(\alpha + \beta m\frac{p}{2})(p-1)$
	Mesh	$(2\alpha + \beta mp)(\sqrt{p}-1)$
	Torus (2-D)	$(2\alpha + \beta mp)(\sqrt{p}-1)$

	Hypercube	$(\alpha + \beta m)(p-1)$
	Tree	$(\alpha + \beta m\frac{p}{2})(p-1)$
	FCG	$(\alpha + \beta m\frac{p}{2})(p-1)$

➔ **<u>Corresponding functions in MPI</u>** :

Operation	MPI function(s)
One-to-all broadcast	MPI_Bcast
All-to-one reduction	MPI_Reduce
All-to-all broadcast	MPI_Allgather[v]
All-to-all reduction	MPI_Reduce_scatter[_block]
All-reduce	MPI_Allreduce
Scatter	MPI_Scatter[v]
Gather	MPI_Gather[v]
All-to-all personalized	MPI_Alltoall[v|w]

➔ **<u>MPI</u>** :

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed by a group of researchers from academia and industry to function on a wide variety of parallel computing architectures.

MPI's send and receive calls operate in the following manner. First, process A decides a message needs to be sent to process B. Process A

then packs up all of its necessary data into a buffer for process B. These buffers are often referred to as envelopes since the data is being packed into a single message before transmission (similar to how letters are packed into envelopes before transmission to the post office). After the data is packed into a buffer, the communication device (which is often a network) is responsible for routing the message to the proper location. The location of the message is defined by the process's rank. Even though the message is routed to B, process B still has to acknowledge that it wants to receive A's data.

➔ **Simgrid**

SimGrid is a scientific instrument to study the behaviour of large-scale distributed systems such as Grids, Clouds, HPC or P2P systems. It can be used to evaluate heuristics, prototype applications or even assess legacy MPI applications. All this is available as free software.

Simgrid allows one to define the platform on which the algorithm has to be simulated via an XML file. XML files for some standard architectures can be found in the GitHub repository provided below.

We used the following flags to get the trace data from all the links and nodes :

- --cfg=tracing:yes
- --cfg=tracing/smpi:yes
- --cfg=tracing/smpi/internals:yes
- --cfg=tracing/uncategorized:yes

We used pajeng to visualize the trace data and get meaningful information out of the trace data produced. All automation scripts and data files can be

found in the GitHub repository. The following is a sample output after filtering the trace data produced by simgrid. The other graphs that were generated can be found in the presentation.

Source	Destination	Hop count
1	2	4
1	9	1
1	3	3
3	4	5
1	5	2
5	6	7
5	7	6
7	8	8

➔ **Github Repository** :
https://github.com/kanishkarj/Distributed-Systems

➔ **References**:

- https://www.cs.uky.edu/~jzhang/CS621/chapter5.pdf
- http://parallelcomp.uw.hu/ch04lev1sec1.html
- https://www8.cs.umu.se/kurser/5DV050/VT13/coll.pdf
- https://www8.cs.umu.se/kurser/5DV050/VT12/F1b.pdf
- https://s3.amazonaws.com/content.udacity-data.com/courses/gt-cse6220/Course+Notes/Lesson2-1+Basic+Model.pdf
- Udacity
- https://simgrid.org/tutorials/simgrid-smpi-101.pdf
- http://simgrid.gforge.inria.fr/simgrid/3.20/doc/getting_started.html
- http://simgrid.gforge.inria.fr/simgrid/3.20/doc/outcomes_vizu.html#instr_category_functions
- https://github.com/schnorr/pajeng