

Notes on Parallel Computing

Nihar Shah

June 13, 2024

Contents

1	Introduction	4
2	Classification of Architectures - Flynn's Taxonomy	5
2.1	SIMD - Single Instruction Multiple Data	5
2.2	MISD - Multiple Instruction Single Data	7
2.3	MIMD - Mutliple Instruction Multiple Data	7
3	Classification based on Memoory	8
3.1	Shared Memory	8
3.1.1	Uniform Memory Access (UMA)	9
3.1.2	Non-Uniform Memory Access (NUMA)	10
3.2	Distributed Memory Architeture	11
3.3	Shared Memory Architectures: Cache Coherence	12
3.3.1	State Transition Diagram	13
3.4	Implementation of Cache Coherence Protocols	14
3.4.1	Snooping Protocol	14
3.4.2	Directory Based Protocol	14
3.5	False Sharing	15
4	Interconnection newtorks for a Parallel Computer	17
4.1	Network Topology	18
4.1.1	Bus-based networks	18
4.1.2	Cross-bar Networks	19
4.1.3	Multistage Network	20
4.1.4	Completely Connected Networks	25
4.1.5	Star Connected Network/Star Topology	26
4.1.6	Linear Arrays, Meshes and k-d Meshes	27
4.1.7	Tree Based Networks	29
4.2	Evaluating Static Networks	30
4.2.1	Diameter	30
4.2.2	Connectivity	31
4.2.3	Bisection Width and Bisection Bandwidth	31
4.2.4	Cost	32
5	Graphical Processing Units	33
5.1	GPU Architecture	34
5.2	CUDA Memory Spaces	36

6 Parallelization Principles	39
6.1 Evaluation of Parallel Programs	39
7 Parallel Programming Classification and Steps	44
7.1 Parallel Program Models	44
7.2 Programming Paradigms	44
7.3 Parallelizing a Program	44
7.4 Data Parallelism and Data Decomposition	45
7.5 Data Distributions	46
7.6 Task Parallelism	47
7.7 Orchestration	48

Here it has been assumed that as a prerequisite, the reader has a basic understanding of C programming language and familiarity with Basics of Computer Architecture.

1 Introduction

Moore's Law states

The number of transistors in a dense integrated circuit doubles approximately every two years. (1)

This law has been true for the past 50 years. This has led to the increase in the number of cores in a processor.

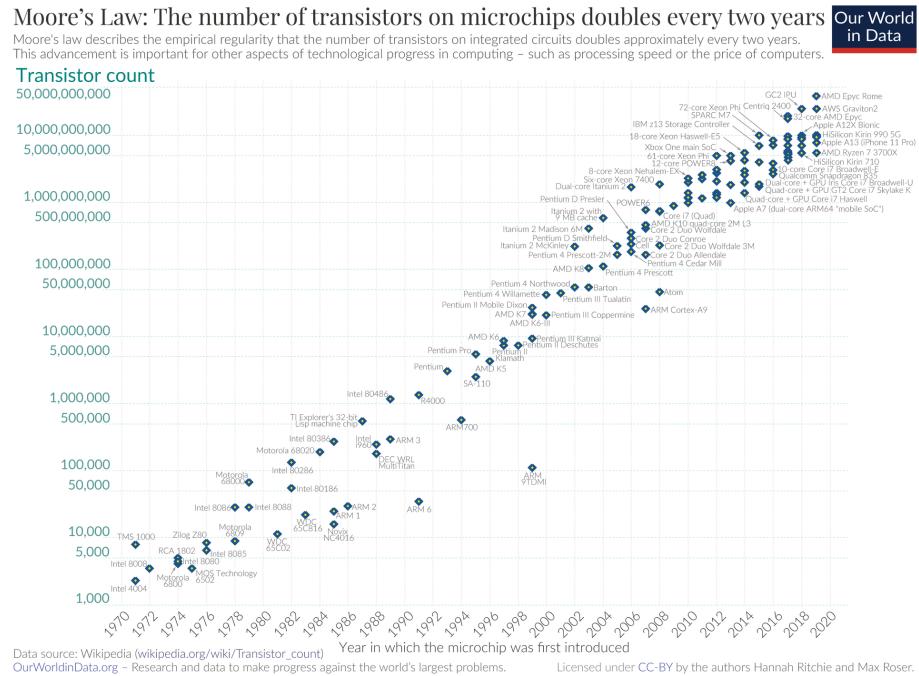


Figure 1: Moore's Law Graph

But computing power is saturating. The power consumption is increasing and the heat dissipation is becoming a problem. Moore's Law is not sustainable. The solution to this problem is parallel computing. Parallel Computing is the use of multiple

processors to perform a computation. It is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently. It leads to Faster execution times from days or months to hours or seconds. Example: Climate Modelling, Bioinformatics, Computational Fluid Dynamics, etc. The large amount of data can be processed in parallel. Example: Google, Facebook, process billions of requests per day. Parallelism cannot be achieved in all problems. It is not always possible to divide the problem into smaller problems. But it is more natural for certain kinds of problems. Example: Climate Modelling.

Because of the Computer Architecture trends the CPU Speeds have saturated and thus slow memory bandwidth is the bottleneck. Parallelism helps in data transfer Overlap.

2 Classification of Architectures - Flynn's Taxonomy

Flynn's Taxonomy is a classification of computer architectures based on the number of instruction streams and data streams.

- SISD - Single Instruction, Single Data (Serial Computers)
- SIMD - Single Instruction, Multiple Data
- MISD - Multiple Instruction, Single Data
- MIMD - Multiple Instruction, Multiple Data

2.1 SIMD - Single Instruction Multiple Data

It uses vector processors and processor array. In this, a single instruction is executed on multiple data elements simultaneously.

- Vector Processors - They are capable of executing a single instruction on multiple data elements simultaneously.
 - Processor Array - It is a collection of processors that work in parallel on different data elements.
-

For example, CM-2, Cray-90, Intel Xeon Phi, Intel -vector instructions (1028 bytes), etc.

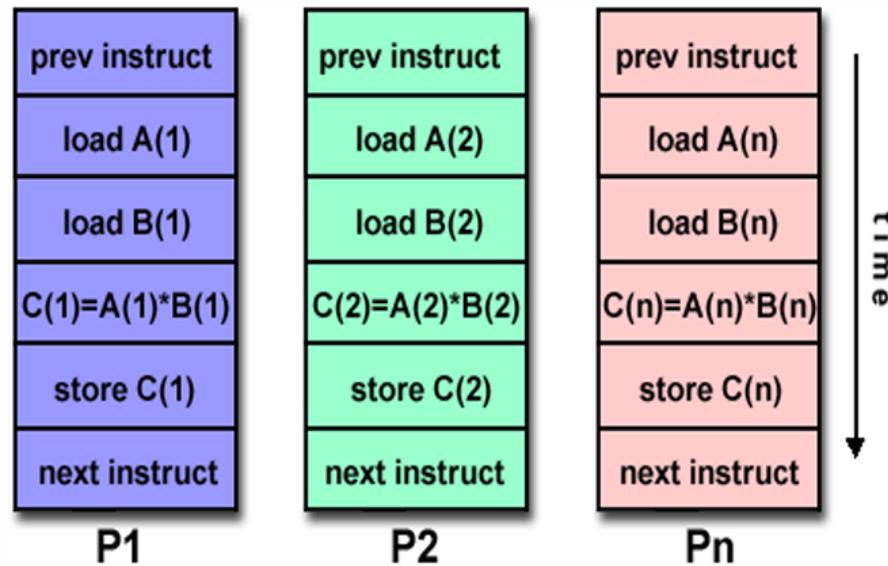


Figure 2: SIMD Architecture

[Link to Parallel Computing Tutorial](#) This link provided by LLNL provides a good tutorial on parallel computing for more information. Consider the figure 2. Say two vectors A and B are given of size n (i.e. n elements). The operation is to perform a element-wise multiplication between the two given vectors. We are given n processors (from P₁,...,P_n) Then, in case of SIMD architecture we will pass say ith element of the vector to the ith processor i.e. the first processor will get the first element of the vector, the second processor will get the second element of the vector and so on. Then the processor will simultaneously with time start executing the same instruction on the data element that it has been given. So as shown in the figure 2, all the processors will simultaneously load the data element i.e. ith processor will load the ith element of the vector A. P₁ will load A(1), P₂ will load A(2) and so on P_n will load A(n). Then they will simultaneously execute the next instruction to load the data element of the vector B. Thus, the ith processor will load the ith element of the vector B i.e. P₁ will load B(1), P₂ will load B(2) and so on P_n will load B(n). Then, once the data has been loaded, the processors will simultaneously execute the next instruction to multiply the data elements that they have loaded.

Thus, P1 will multiply A(1) and B(1), P2 will multiply A(2) and B(2) and so on Pn will multiply A(n) and B(n). Then, the processors will simultaneously execute the next instruction to store the element in the result vector C. The ith processor will store the result in the ith element of the vector C. Thus, P1 will store the result in C(1), P2 will store the result in C(2) and so on Pn will store the result in C(n). Thus, the SIMD architecture is used to perform the element-wise multiplication of two vectors in parallel. This is how the SIMD Architecture works.

2.2 MISD - Multiple Instruction Single Data

This is not very common. It is used in fault-tolerant systems. In this, multiple instructions are executed on the same data. Some of the examples are:

- Fault-tolerant systems
- Redundant systems
- N-modular redundancy
- Cryptography - Multiple algorithms to crack same data

2.3 MIMD - Multiple Instruction Multiple Data

In this, multiple instructions are executed on multiple data elements. It is the most common architecture. Some of the examples are IBM SP, most supercomputers, cluster, Computational grids, etc.. At a given timestep different processors can execute different instructions on different data elements.

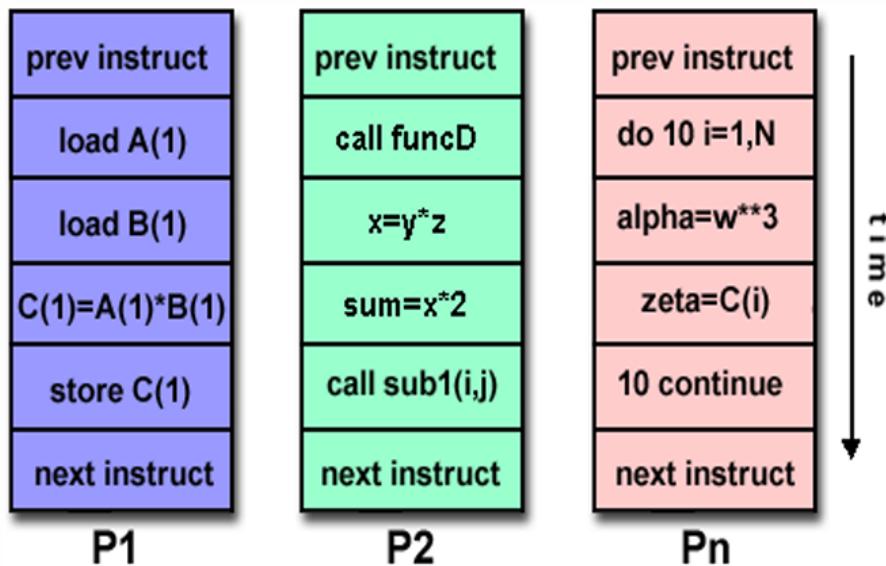


Figure 3: MIMD Architecture

As shown in figure 3, each of the processor P1, P2, ..., Pn can execute different instructions on different data elements simultaneously in time. P1 is doing element wise multiplication, P2 is running a completely different instruction simultaneously on completely different data elements and so on. Thus, MIMD architecture is used to perform different instructions on different data elements in parallel. This is how the MIMD Architecture works.

3 Classification based on Memoory

- Shared Memory
- Distributed Memory

3.1 Shared Memory

In this, all the processors share the same memory. It requires special computer architecture for memory management (explained later). It is easier to program.

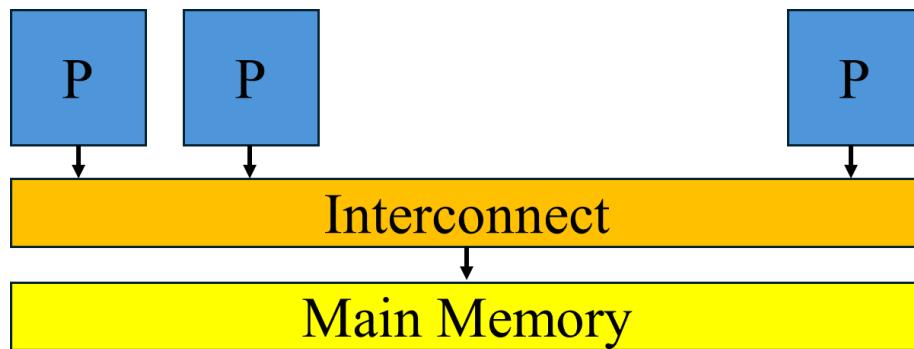


Figure 4: Shared Memory Architecture

As shown in figure 4, in this the n processors share the same physical address space i.e. say the address is 0x7FFF5FBFFD98 then, whenever a processor accesses the address 0x7FFF5FBFFD98 then it refers to the same physical address location in the memory for all the processors P_1, P_2, \dots, P_n i.e. thus address 0x7FFF5FBFFD98 is at the same physical location for all the processors. Thus, any of the processor can access the address 0x7FFF5FBFFD98 or the address 0x7FFF5FBFFD98 refers to the same physical location for all the processors. This is called sharing the same physical address space. It will be cleared further when you will see Distributed Memory Architecture. It will be explained in detail later how the shared memory is managed in the computer architecture and thus will clear the doubts about how the shared memory is managed. This leads to certain problems in Read and Write and Maintaining Cache Coherence (explained later) Thus, any communication between the processors can be done thorough this shared memroy. There are two types of shared memory:

- Uniform Memory Access (UMA) - All processors have equal access time to all memory locations.
- Non-Uniform Memory Access (NUMA) - Access time depends on the location of the memory.

In general, the time taken by the processor to access a memory location depends on the distance between the processor and the memory location.

3.1.1 Uniform Memory Access (UMA)

In this, all the processors have equal access time to all memory locations. It is easier to program.

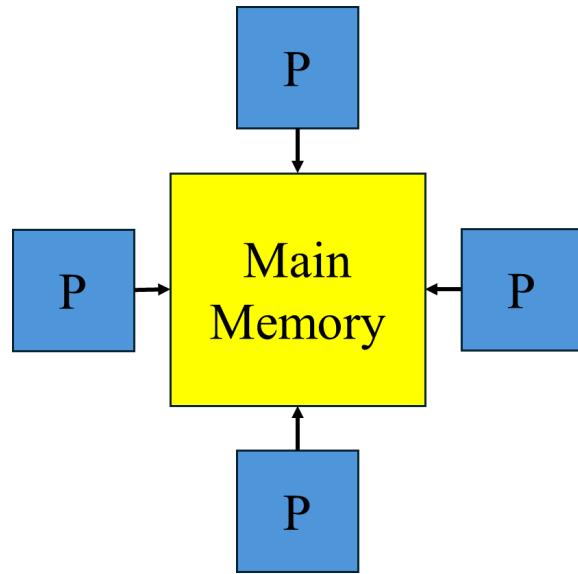


Figure 5: Uniform Memory Access Architecture

As shown in figure 5, all the processors have equal access time to all the memory locations i.e. the time taken by the processor to access a memory location is same for all the processors. All can be thought of as being at the same distance from the memory location. Thus, all the processors have equal access time to all the memory locations.

3.1.2 Non-Uniform Memory Access (NUMA)

In this, the access time depends on the location of the memory. It is difficult to program.

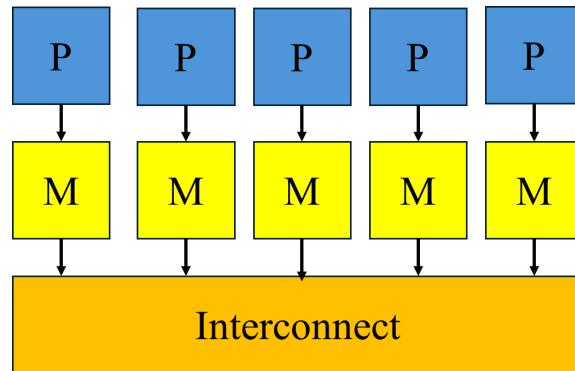


Figure 6: Non-Uniform Memory Access Architecture

As shown in figure 6, each processor might have some portion of shared physical address space that is physically close to it and therefore accessible to it in less time. Thus, the time taken by the processor to access a memory location depends on the location of the memory.

3.2 Distributed Memory Architcture

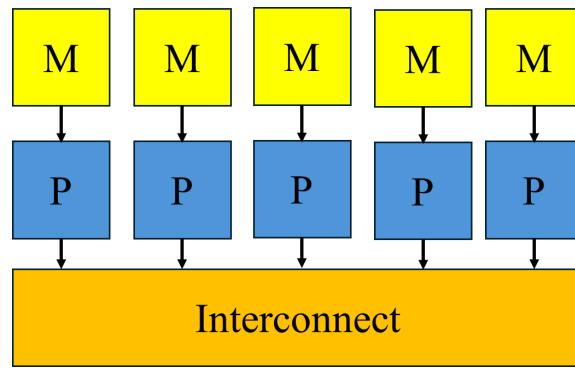


Figure 7: Distributed Memory Architecture

As shown in figure 7, If two processors access the same memory address say 0x7FFF5FBFFD98 then they will get different data since the physical address space is not shared. Thus, the address 0x7FFF5FBFFD98 refers to different physical locations for different processors. The only way to access data between processor is by sending and recieving data i.e. message passing. This is generally preferred when we wish to use very large number of cores. It is difficult to program.

3.3 Shared Memory Architectures: Cache Coherence

Now consider the figure 4. As shown in the figure the processors are connected to the Main Memory via Interconnect. Note that even in case of shared memory architecture individual processors have their own caches. Now suppose there is a variable X in the Main Memory and the value of X is 5. Now suppose the processor P1 tries to access the variable X, it will be a Cache miss and hence the value of X will be loaded into the cache of the processor P1. Now suppose the processor P2 tries to access the variable X, it will be a Cache miss and hence the value of X will be loaded into the cache of the processor P2. Now suppose the processor P1 changes the value of X to 10, it will be a Cache hit and hence the value of X will be updated in the cache of the processor P1. Now suppose the processor P2 tries to access the variable X, it will be a Cache hit and hence the value of X used will be the stale value 5. This is called Cache Coherence Problem. The value of X is not consistent across the caches of the processors. There are two ways to solve this problem:

- Write Update Protocol - Propagate Cache lines to other processors on every write to a processor as well as Global memory. Write operation in write update takes longer time because it has to send the data to all the processors. In Read operation, write update is better as the update data is there in local caches of the processor, hence its a faster read.
- Write Invalidate Protocol - Whenever one of the Processor updates the data it sends a signal to other processor to invalidate the data in the cache. Thus, next time if the processor requires that data it will be a cache miss and the data loads from the main memory. Each processor gets the latest data from the main memory. In update protocol, the write updates are faster as it is required to send only an invalidate signal to the processors. In write invalidate protocol, read operation will be a cache miss: thus requires fetching data from memory.

In write update protocol, if the data is present in another processor it will get updated regardless of whether that data is going to be used or not by that processor. This does not happen in write invalidate protocol. Thus, more traffic in write update protocol. This is why most of the systems use write invalidate protocol. NOTE: The specific steps of protocols vary from implementation to implementation.

3.3.1 State Transition Diagram

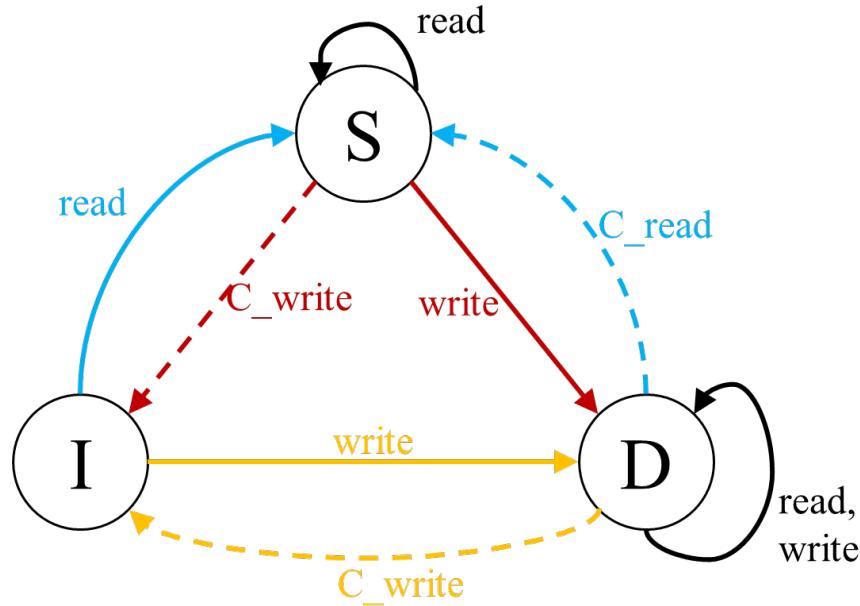


Figure 8: State Transition Diagram for invalidate protocol

A cache line can be in either of the three states -Shared (S) - a variable shared by 2 caches, Invalid (I) - another Processor has updated the data, Dirty (D) - the data has been updated by the processor. In the figure shown in 8, the state transition diagram for the invalidate protocol is shown. The solid lines indicate the Primary action and the dashed lines indicate the Cache coherence protocol action.

Now consider the figure 8. Suppose the cache line is in the shared state S i.e. the data is shared by at least two caches (i.e. in different processors). Then if it is read by either of the processors, it will remain in the shared state S indicated by the solid black line.

If it is written by one of the processors, it will go to the dirty state D from shared state S for the processor that writes the data as shown by the solid red line from S to D, and because of the cache coherence protocol action, it will go to the invalid state I for the other processor as shown by dashed red line in the figure 8 from S to I. If the cache line is in the dirty state D i.e. the data has been updated by the processor, then if it is read/write by the processor, it will remain in the dirty state D as shown by solid black line in the figure 8 from D to D.

If the cache line is in the invalid state I i.e. the data has been updated by another processor, then, if it is read by the processor, it will be a cache miss and hence the data will be loaded from the main memory and the cache line will go to the shared state S as shown by the solid blue line in the figure 8 going from I to S. Because of this read, the cache coherence protocol action will be to go from the Dirty state D to the shared state S for the other processor as shown by the dashed blue line in the figure 8 from D to S.

If the cache line is in the invalid state I i.e. the data has been updated by another processor, then, if it is written by the processor, it will go to the dirty state D as shown by the solid yellow line in the figure 8 from I to D. Thus, as a result the processor will send an invalidate signal to all the processors as part of the cache coherence protocol, the cache line which was in Dirty state D for the other processor will thus become invalid. Thus, the dirty state D will go to Invalid state I for the other processor as shown in figure 8 by a dashed yellow line from D to I.

3.4 Implementation of Cache Coherence Protocols

3.4.1 Snooping Protocol

It is generally used for Bus based architectures. Only one operation can be done at a time. All the memoery operations are proagated over the bus and snooped. The idea is that each processor has a specialised cache controller which always snoops onto the bus. Thus, any write operation in any of the cache is publicly broadcasted onto the bus and thus is known by all the other cache controllers of the processors hearing on the bus. Thus, any information related to the cache line present in their cache is invalidated. But this has a problem that only one operation can be done at a time thus, it is slow and not scalable. It requires specialised cache controllers for each processor which increases the cost of architecture. All the data write/read are costly because all the processors are continuously writing on bus and other processors are snooping on the bus. To overcome this problem, the directory based protocol is used.

3.4.2 Directory Based Protocol

It is generally used for Network based architectures. It is scalable and faster. It is used in large scale systems. In the snooping protocol all the other processors even which do not have that data are also snooping on the bus and are listening even though they are not concerned with data. Hence, a mechanism is required where the invalidate signals are sent only to certain processors. This is done by

maintaining a directory for each cache line which contains the information about which processors have the data. Thus, the invalidate signals are sent only to the processors which have the data. Thus, instead of broadcasting memory operations to all the processors, cache coherence operations are propagated only to the relevant processors. A centralised directory maintains states of all the cache lines and the processors which have the data. Consider it has M cache lines and P processors, thus a matrix of $M \times P$ is maintained where each row represents a cache line and each column represents a processor. The data about which processor contains that cache line is maintained by presence bits (1 and 0s) in the matrix. wherever the presence bit is 1, it means that the processor has the data and wherever the presence bit is 0, it means that the processor does not have the data. Thus, any invalidate signal will be sent only to the processor with presence bit as 1. The directory also maintains who is the current owner of the cache line. Note that this is called a *Full Bit Vector Scheme* where the number of presence bits is of the $\mathcal{O}(M \times P)$. Huge number of cache lines/processors requires huge amount of memory for storing directory and less memory for storing the actual data. Thus, requires lots of swaps to memory thus, losing the advantage of parallelism. Ideally, at all times only some of the cache lines will be of interest to only some of the processors. Thus, the matrix $M \times P$ is sparse. Thus, a *Sparse Bit Vector Scheme* is used where the number of presence bits is of the $\mathcal{O}(M + P)$. Here the number of cache lines (m) is much lesser than the total number of cache lines(M) ($M \ll m$) and the number of processors (p) is much less than the total number of Processors (P). ($p \ll P$). Thus, a limited number of cache lines and processors are of interest at any given time.

3.5 False Sharing

A cache is made up of multiple cache lines. All the cache coherence protocols, such as write invalidate, write update deal with the granularity of cache lines and not at the scale of individual variables. Thus, if a variable is in one of the cache lines of caches of two processors and if one of the processors updates some other variable belonging to the same cache line, then even though the variable in the other processor was not updated, since the entire cache line is invalidated, the variable in the other processor will be invalidated. This is called False Sharing.

Let us understand this with an example, Consider modern day caches, a modern day cache lines are of 64 bytes in size. Now consider the data to be stored be of double type which takes 16 bytes. Thus, we can store 4 double type variables in a cache line. Now consider two processors P1 and P2. Suppose a cache line is shared by the processors P1 and P2. Now, say P1 updates the first variable out of the four variables

stored in that cache line. Now consider a situation where P2 does not ever require to read or write the first variable but it requires to read or write the second variable. But, since P1 updates the first variable of the cache line and as the cache coherence protocols deal with granularity of cache lines and not of the individual variables, this will lead to the entire cache line being invalidated for the processor P2. Thus, even though the processor P1 deals only with the first variable and P2 deals with only the second variable, the entire cache line will be invalidated for P2. This is called False Sharing. This leads to unnecessary cache misses and thus, the performance of the system is degraded. Thus, even though these two processors are accessing different variables of the same cache line i.e. it is not true sharing, since processor P1 deals only with the first variable and processor P2 deals with only the second variable, but still any updating to either of the variable will lead to invalidating the entire cache line for the other processor. Thus, it is called False sharing. It unnecessarily introduces cache coherence protocols even though the processors are not sharing the same variable or the same element.

4 Interconnection networks for a Parallel Computer

Interconnection networks for a parallel computer provide mechanisms for data transfer between processing nodes or between processor and memory modules. A blackbox view of interconnection network consists of n inputs and m outputs. Interconnections networks are build using links and switches. **Links** are set of wires or fibres for carrying information. They limit the speed of propagation because of capacitive coupling, attenuation of signal strength which are a function of length of link. **Switch:** A switch maps input ports to output ports. Degree of a switch is the total number of ports on a switch. It supports internal buffering when output ports are busy and allows Routing (to alleviate congestion on Network) and multicast (same output on multiple ports). There are two type of networks:

- Static/Direct Networks - It is point to point communication links among the processing nodes Example: Hypercube, Mesh, Torus, etc.
- Dynamic/Indirect Networks - They are connected by switches linking processing nodes and memory banks. Example: Crossbar, Omega, etc.

The classification is as shown in figure 9. The diagram on the left shows a Static Network and the diagram on the right shows Indirect Network.

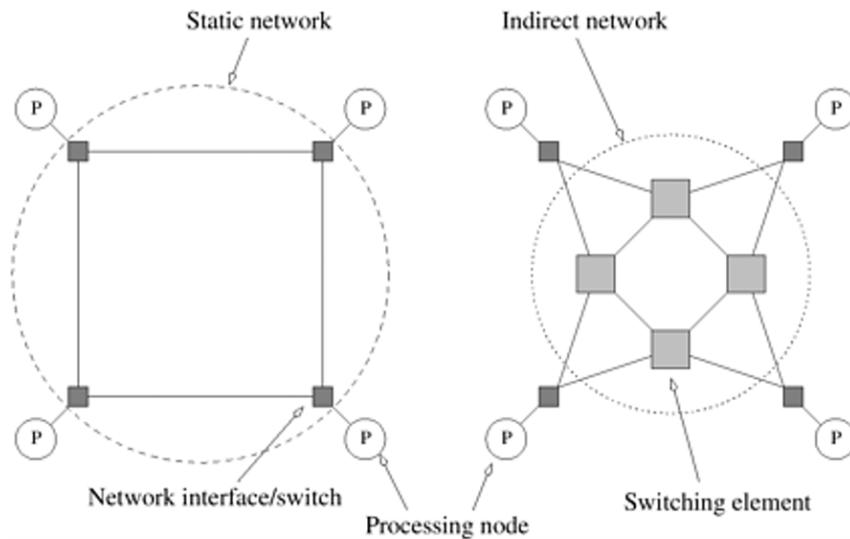


Figure 9: Classification of Interconnection Network

4.1 Network Topology

Network topology is evaluated broadly in terms of cost and scalability with performance.

4.1.1 Bus-based networks

It is a shared medium that is common to all node. They are scalable in cost but unscalable in terms of performance. The cost of the network is thus proportional to the number of nodes p , thus scales as $\mathcal{O}(p)$. The distance between any two nodes in the network is constant $\mathcal{O}(1)$. Since they broadcast information among nodes, there is a little overhead associated with broadcast compared to point to point message transfer. The bounded bandwidth places limitation on the overall performance of the network. Example: Intel Pentium and Sun Enterprise servers. The demands for bandwidth can be reduced by cache for each node i.e. Private data, thus, only remote data is to be accessed through the bus. This is as shown in figure 10.

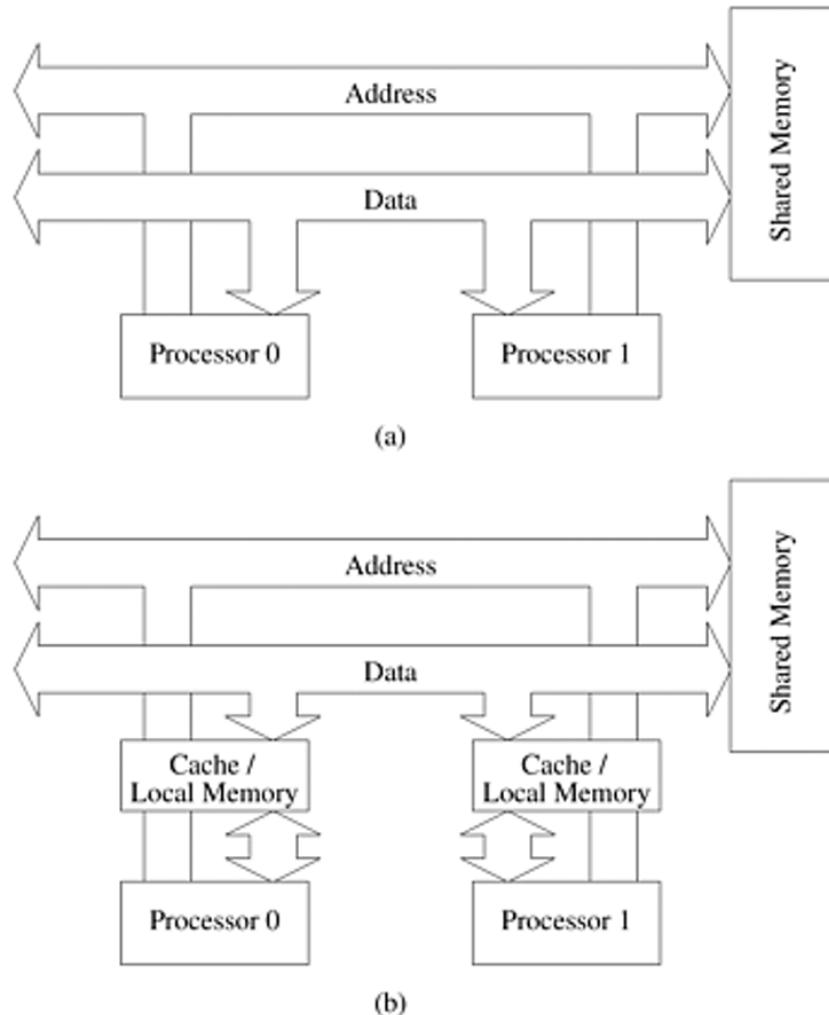


Figure 10: Bus-based Network

The figure at the top shows a bus-based interconnects with no local caches. The figure at the bottom shows a bus-based interconnects with local memory/caches.

4.1.2 Cross-bar Networks

Consider the figure 11. It connects p processors to b memory banks. It is a non-blocking network i.e. the connection of a processing node to a memory bank does not block the connection of any other processing nodes to other memory banks.

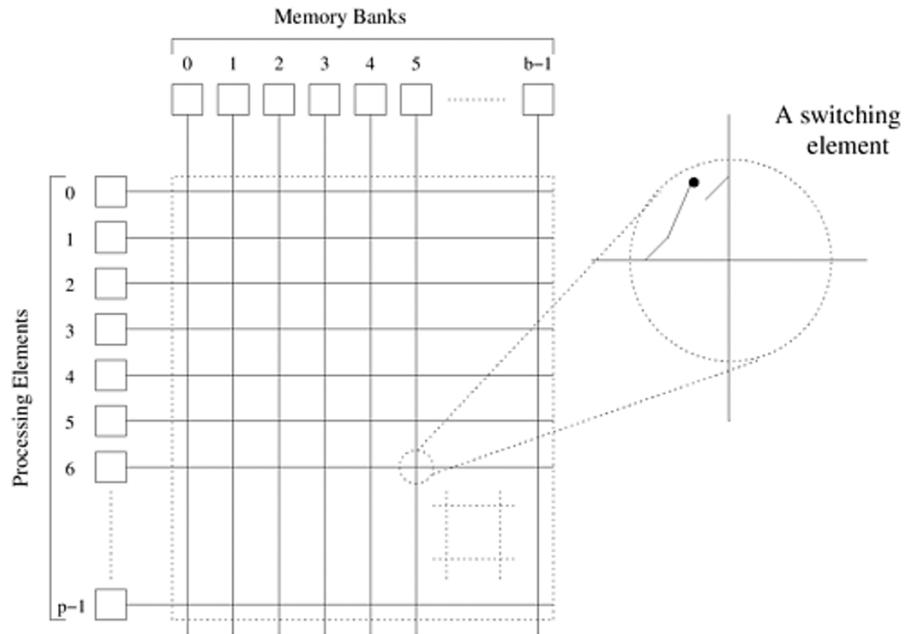


Figure 11: Cross-bar Network

The total number of switching nodes is $\Theta(pb)$. Generally, $b > p$ so that all the processors can access some memory bank. Thus as p is increased, the complexity of switching network grows as $\Omega(p^2)$. Thus, as number of processing nodes becomes large, switch complexity is difficult to realize at high data rates. Thus, Cross bar networks are not scalable in terms of cost but are scalable in terms of performance.

4.1.3 Multistage Network

Consider the figure 12. It lies between crossbar and bus network topology. It is more scalable than bus in terms of performance and more scalable than cross bar in terms of cost.

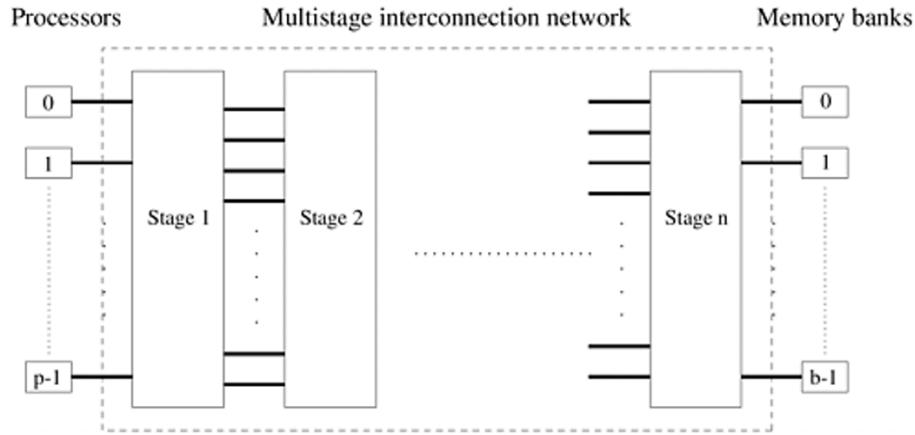


Figure 12: Multistage Network

Omega Network is a type of multistage network with p number of inputs (p processing node), p number of outputs (p memory banks) and $\log(p)$ number of stages each consisting of $p/2$ switches. At any consecutive intermediate stage, a link exists between input i and output j according to the following interconnection pattern:

$$j = \begin{cases} 2i & 0 \leq i \leq p/2 - 1 \\ 2i + 1 - p & p/2 \leq i \leq p - 1 \end{cases} \quad (2)$$

This is called a perfect shuffle i.e. left rotation on binary representation of i to obtain j . In order to understand, consider the example shown in the figure 13. The figure shows a perfect shuffle interconnection for eight inputs and outputs.

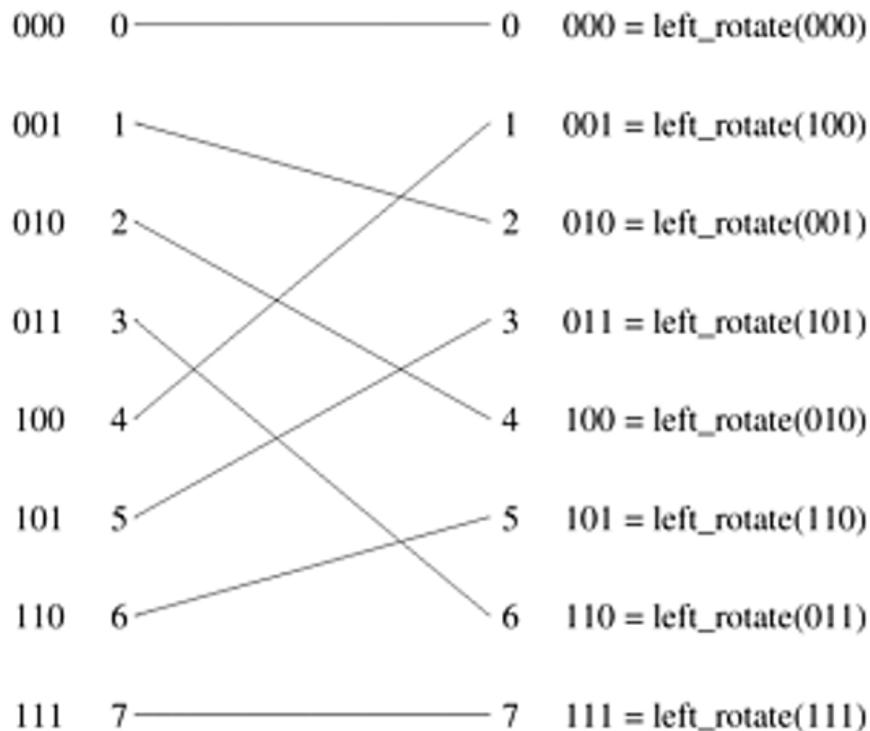


Figure 13: A perfect shuffle interconnection for eight inputs and outputs

Starting from the first input 000. It should be connected to the output obtained on left rotating the input i.e. 000. Thus, as can be clearly seen in the figure it is connected to 000. Similarly, the second input 001 should be connected to the output obtained on left rotating the input i.e. 010. Thus, as can be clearly seen in the figure it is connected to 010. Similarly, the third input 010 should be connected to the output obtained on left rotating the input i.e. 100. It can be seen that there is a connection for 010 to 100 and so on for all the inputs till 111 whose left rotation will be 111 and hence a connection between 111 to 111. Switching nodes can be in either of the two configurations pass-through or cross-over as shown in the figure



Figure 14: Pass-through and Cross-over configuration of Switching Nodes

In figure 14 the figure on the left shows pass-through configuration of a switching node in which the inputs are sent straight to outputs and the figure on the right shows cross-over configuration of a switching node in which the inputs are crossed over and sent out.

Now, the number of switches required in each stage is $p/2$ as a switching element takes two inputs and return two outputs. Thus, since there are p inputs, $p/2$ switches are required in each stage. Thus, the total number of switches required is $\log(p) * p/2 = \Theta(p \log(p))$. Recall, that the number of switches required in complete cross bar network scaled as $\Theta(p^2)$ and the number of switching nodes in case of a bus-based network scaled as $\Theta(p)$. Thus, $\Theta(p) < \Theta(p \log p) < \Theta(p^2)$. This, proves that the multistage network is more scalable than the cross bar network in terms of cost and more scalable than the bus network in terms of performance.

Omega Network

Consider the complete omega network as shown in the figure 15. We now understand the routing scheme of the omega network.

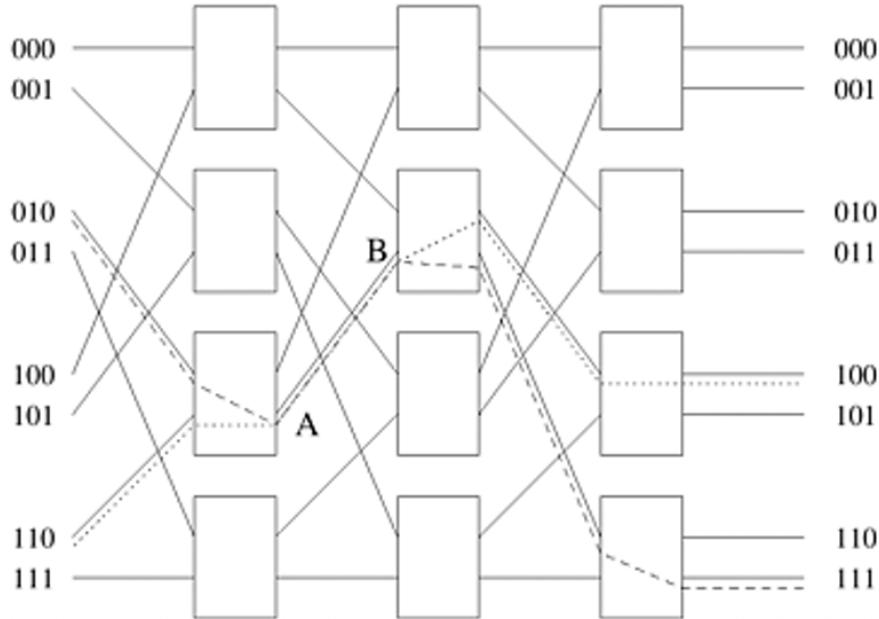


Figure 15: Omega Network

Let s be the binary representation of processors that need to write some data to bank t . Now the bits are written in the binary representation. The most significant bit is the left most bit and the least significant bit is the right most bit. Now the routing scheme works as follows, at the first switching node, if the most significant bits of s and t are the same, then the data is routed in pass-through mode. If bits are different than the data is routed through cross-over mode. At the second switching node, if the second most significant bits of s and t are the same, then the data is routed in pass-through mode. If bits are different than the data is routed through cross-over mode. This is done for all the bits of s and t , by repeating over next switching stage using the next most significant bit. In this way, it uses all $\log p$ bits in the binary representation of s and t .

For example, consider a message to be passed from 010 to 111. As shown in the figure 15 by dashed lines. As the message reaches the first switching elements because the most significant bits of 010 and 111 are different the message is routed through cross-over mode. As the message reaches the second switching elements because the second most significant bits of 010 and 111 are the same it is routed through pass-through mode. As the message reaches the third switching elements because the third most significant bits of 010 and 111 are different the message is routed through cross-over mode. Thus, the message is passed from 010 to 111.

Consider another example, consider a message to be passed from 110 to 100. As shown in the figure 15 by solid lines. As the message reaches the first switching elements because the most significant bits of 110 and 100 are the same the message is routed through pass-through mode. As the message reaches the second switching elements because the second most significant bits of 110 and 100 are different it is routed through cross-over mode. As the message reaches the third switching elements because the third most significant bits of 110 and 100 are the same the message is routed through pass-through mode. Thus, the message is passed from 110 to 100.

Now consider the case where both of the messages in the above examples were to be passed from 010 to 100 and 110 to 111 simultaneously. As shown in the figure 15 by solid and dashed lines. In the case when the processor two and six are communicating simultaneously then one may disallow access to another memory bank the another processor. This is because they have a common link in the routing scheme shown by AB in the figure 15. This property is called **blocking networks**.

4.1.4 Completely Connected Networks

In this, each node has a direct communication link to every other node in the network. A node can send a message to another node in a single step, since a communication link exists between them. It is a static counter part of crossbar switching networks as the connection between any input/output pair does not block communication between any other pair. For example, a completely connected network of eight nodes is as shown in the figure 16.

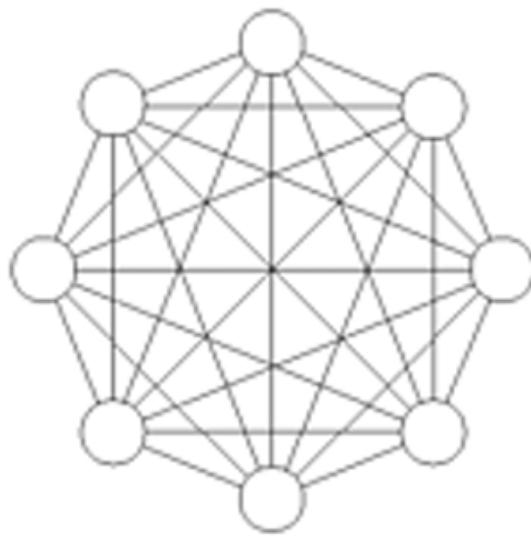


Figure 16: Completely Connected Network

4.1.5 Star Connected Network/Star Topology

In this, all the nodes are connected to a central node. The central node acts as a switch to connect the nodes.

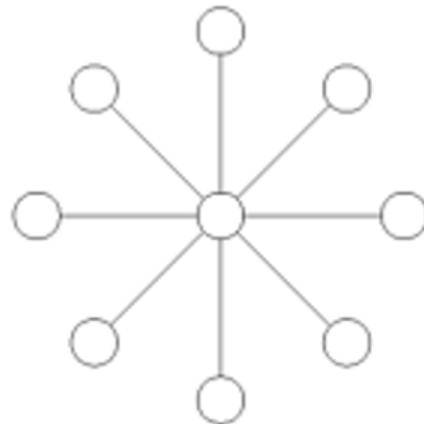


Figure 17: Star Connected Network

As shown in the figure 17. Every other processor has a communication link connecting it to this processor. It is thus similar to a shared bus. Here, the central

processor thus acts as a bottleneck.

4.1.6 Linear Arrays, Meshes and k-d Meshes

Linear arrays are as shown in figure 18. It is a one-dimensional array of processing nodes. Each node is connected to its immediate neighbours.



Figure 18: Linear Array

As shown in the figure 18 there are two possible cases, the left figure shows a linear array with no wraparound links. It is a static network in which each node (except the two nodes at the ends) has two neighbors, one each to its left and right. The right figure shows a linear array with wraparound links also called a 1D torus. The wraparound links connect the last node to the first node and the first node to the last node. The ring has a wraparound connection between the extremities of linear array. Each node has two neighbors in this case.

2D Mesh

It is a linear array extended to 2D as shown in the figure 19

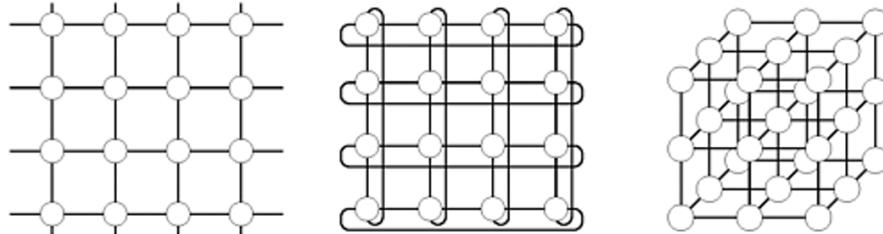


Figure 19: Mesh

It has \sqrt{p} nodes in each direction. As shown in the figure there are three possible cases for a 2D mesh. The leftmost figure shows a 2D mesh with no wraparound links. In this each (i,j) node is connected to $(i+1,j), (i,j+1), (i-1,j), (i,j-1)$. It can be laid out in 2D space. A variety of regulatory structure computations map naturally to 2D. 2D mesh were often used as interconnects in parallel machines.

The middle figure shows a 2D mesh with wraparound links. It is also called a 2D Tori.

The rightmost figure is a 3D Cube with no wraparound which is a generalization of

2D mesh to 3D. In this, each node is connected to six other nodes, two along each of the three dimensions. 3D simulations can be mapped naturally to 3D.

General Class of k-d meshes

It is a class of topologies with d dimensions and k nodes along each dimension. Thus, in total k^d nodes. A 1D linear arrays is a special case of a k-d mesh with $d=1$. A 2D mesh is a special case of a k-d mesh with $d=2$. A hypercube has two nodes along each dimension and has $\log_2 p$ dimensions. It is thus, written as 2-log mesh. A d -dimensional hypercube is constructed by connecting two $(d-1)$ dimensional hypercubes as shown in the figure 20.

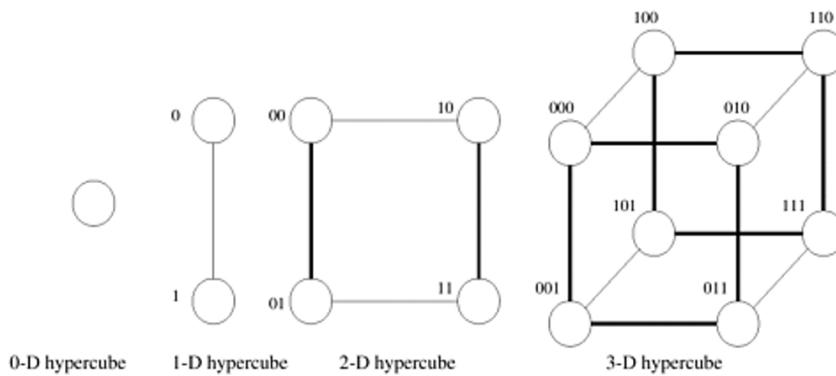


Figure 20: Hypercube

The hypercubes follow a numbering scheme which is useful later on for writing many parallel algorithms. Number the nodes then place the two $d-1$ dimensional ($p/2$ nodes) hypercubes side by side. Prefix one with 0 and the other with 1. **Property: The minimum distance between two nodes is given by the number of bits that are different in two labels.** For example, 0110 and 0101 are different in two bits. the third and fourth place bits. Thus, the minimum distance between them is two links apart. This property can be used for deriving parallel algorithms for hypercube architecture. A 4D hypercube with 16 nodes is as shown in the figure 21

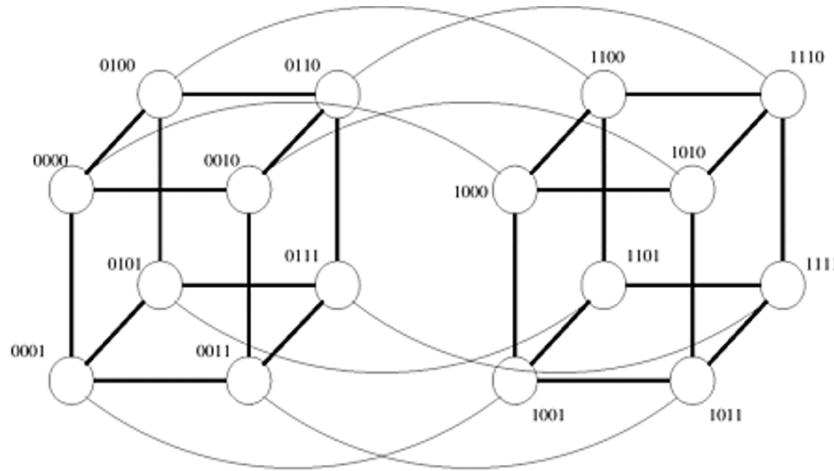


Figure 21: 4D Hypercube

4.1.7 Tree Based Networks

It is a hierarchical network. It is a tree with a root node and each node has a parent and children as shown in the figure 22.

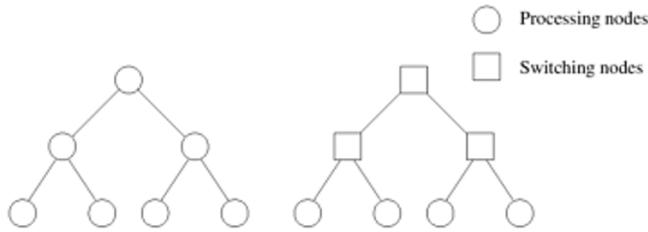


Figure 22: Tree Based Network

Note that there is only one path between any pair of nodes. IN the figure 22, the left figure shows a Static Tree Network where each processing element is at each node of the tree. The right figure shows a Dynamic Tree Network where the nodes at intermediate level are switching nodes, and the leaf nodes are processing elements.

Routing: Source sends the message up the tree until it reaches the node at the root of the smallest subtree containing both the source and destination nodes. Then the message is routed down the tree towards the destination node. At higher levels of tree, example, nodes in left subtree of a node communicate with nodes in right tree, root node must handle all the messages. This leads to communication bottleneck.

It is solved by dynamic tree called Fat Tree as shown in figure 23 by increasing the number of connection links and switching nodes close to the root.

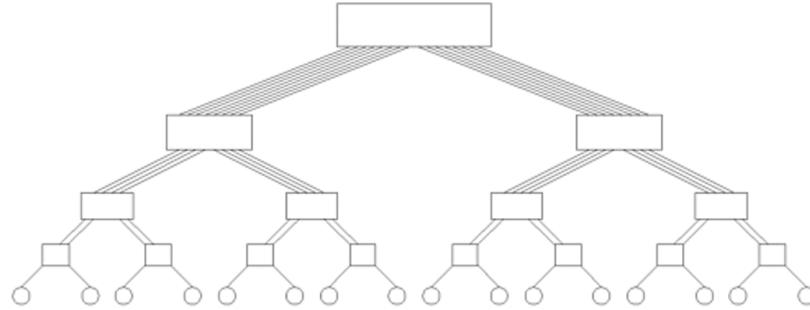


Figure 23: Fat Tree with 16 Processing Nodes

The processors are arranged in leaves and all the other nodes correspond to switches. Note that here the property that the number of links from a node to a children is equal to the number of links from the node to its parent. Thus, the edges become fatter as we traverse up. Now any pair of processors can communicate without contention : non-blocking network. It has a constant bisection bandwidth networks as the number of links crossing the bisection is constant. As shown in the example figure 23 a two level fat tree has a diameter of four.

4.2 Evaluating Static Networks

The performance of a static network is evaluated in terms of cost, performance and scalability.

4.2.1 Diameter

The diameter of a network is the maximum distance between any two processing nodes in the network. The distance between any two processing nodes is defined as the shortest path (in terms of number of links) between them. For example, the diameter of a completely connected network is 1 and of the star-connected network is two. The diameter of a ring network is $\lfloor p/2 \rfloor$. The diameter of a 2D mesh without wraparound connection is $2(\sqrt{p} - 1)$. The diameter of a 2D mesh with wraparound connection is $2\lfloor \sqrt{p}/2 \rfloor$. The diameter of a complete binary tree is $2 \log(\frac{p+1}{2})$ because two communicating nodes may be in separate subtrees of the root node and a message might have to travel all the way to the root and then down the other subtree.

4.2.2 Connectivity

The connectivity of a network is a measure of the multiplicity of paths between any two processing nodes. A network with high connectivity is desirable since it lowers latention of communication resources.

Arc Connectivity of the network: The minimum no. of arcs that must be removed from the network to break it into two disconnected network. For example, Arc connectivity of Linear array, tree star connected networks is 1. For rings, 2D meshes without wraparound is 2. It is 4 for 2D meshes with wraparound. It is d for d -dimensional hyper cubes.

4.2.3 Bisection Width and Bisection Bandwidth

Bisection Width: The bisection width of a network is defined as the minimum no. of communication links that must be remove to partition into two equal halves. For example, bisection width of ring=2. Bisection width of 2D p node mesh without wraparound = \sqrt{p} . For a 2D mesh with wraparound bisection width = $2\sqrt{p}$. Bisection width of a hypercube, but deconstructing the connection we did to get from $2(d - 1)$ dimensional hypercube to 1d dimensiona hypercube. Hence, $p/2$ nodes to be cut to seperate into two subcubes.

Channel Width: No. of bits that can be communicated simultaneously over a link connecting two nodes. Channel width = no. of physical wires in each communication link. **Channel rate:** The peak rate at which a single physical wire can deliver bits is called the channel rate.

Channel bandwidth: Peak rate at which data can be communicated between the ends of a communication link. Thus,

$$\text{Channel bandwidth} = \text{Channel Width} \times \text{Channel rate} \quad (3)$$

Bisection bandwidth: The minimum volume of communication allowed between any two halves of the network.

$$\text{Bisection bandwidth/Cross section bandwidth} = \text{Bisection Width} \times \text{Channel width} \quad (4)$$

It is also a measure of cost as it provides a lower bound on aread in 2D packing and volume in 3D. Say bisection width = w , lower bound on area in 2D packaging gives $\Theta(w^2)$ and for volume in 3D packaging is $\Theta(w^{3/2})$. According to it, the hypercubes and completely connected networks are more expensive.

4.2.4 Cost

Number of communication links or number of wires required by the network. For example, linear array = $p-1$ links to connect p nodes. For, d -dimensional wraparound mesh = dp links. For a hypercube, $\frac{p \log p}{2}$ links.

All of this has been summarise in the following table 1.

Network	Diameter	Bisection Width	Arc connectivity	Cost
Completely connected	1	$p^2/4$	$p-1$	$\frac{p(p-1)}{2}$
Star	2	1	1	$p-1$
Ring	$\lfloor p/2 \rfloor$	2	2	$p-1$
Complete Binary Tree	$2 \log(\frac{p+1}{2})$	1	1	$p-1$
Linear Array	$p-1$	1	1	$p-1$
2D mesh (no wraparound)	$2(\sqrt{p} - 1)$	\sqrt{p}	2	$2\sqrt{p}(\sqrt{p} - 1)$
2D mesh (wraparound)	$2\lfloor \sqrt{p}/2 \rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$\frac{p}{2}$	$\log p$	$\frac{p \log p}{2}$
Wraprround k-array dcube	$d\lfloor k/2 \rfloor$	$2k^{d-1}$	$2d$	dp

Table 1: Summary of Static Networks

5 Graphical Processing Units

A CPU has at max of 32 Cores and 8 threads per core. It is good for serial processing. Thus, they are called Multi-core architectures. But for parallel processing, we need more cores. Thus, we need to use GPUs. GPUs have 1000s of cores and are good for parallel processing. Thus, are called Many-Core Architectures. GPUs consist of a large number of light-weight cores which are not as powerfull as CPU Cores. In the early, days before 2014 NVIDIA GPUs were used for gaming and video applications for graphical displays. Theses can be done in stages with each stage consisting of independent computations like reading pixels, shading, etc. which can be done in highle parallel manner. Reading of one pixel is independent of reading of other pixel. Processing/rendering of pixels can happen simultaneously. Higher no. of cores of GPU helps in processing in real time. Thus, GPU many-core architectures consisting of light-weight cores as they can perform only simple computations and not very heavy computations are used. Typically GPU and CPU coexists in a hetrogeneous setting. GPUs do not exists in standalone setting. A program runs on a CPU(coarse-grain parallelism) and CPU offloads some of the computations to the GPU for light-weight computations (fine-grain parallelism). NVIDIA's GPU architectures consists of CUDA (Compute Unified Device Architecture) which is a parallel computing platform and application programming interface (API) model created by NVIDIA. CUDA programming model is based on C/C++ and is used to program NVIDIA GPUs. It is used to write parallel programs that run on the GPU.

GPUs were initially used for gaming (possible to divide up monitor pixels to process things in parallel). They are also efficient in Matrix calculations.

5.1 GPU Architecture



Figure 24: GPU Architecture

A GPU Architecture is shown in figure 24. The GPU Architecture consists of five main components:

- TPC - Texture Processing Clusters
- SMX - Streaming Multiprocessors
- SP - Single Precision (GPU Core)
- DRAM - Device RAM (GPU RAM)
- ROP - Render Output Unit

GPU cores are called Streaming Multiprocessors (SMX). SP cores are called Single Precision (GPU Core). Coarse level parallelism tasks execute on SMX which gives fine level parallel tasks to execute on SP. Thus, promotes coarse and fine level parallelism. For example, the latest NVIDIA Architecture Kepler consists of 15 SMX with each of 192 SP cores = 2880 SP cores in total. The speed of each of the processors is 745 MHz which is much less than that of a CPU. GPUs follow SIMD (Single Instruction Multiple Threads) model. NVIDIA GPU has L2 cache, common memory and individual caches. Each SMX has its own 32 bit registers for storing the context of GPU threads. Consider the figure 25 of one SMX, the GPU has 65536 registers,

192 SP cores - Single Precision Cores for single precision calculations, 64 DP cores - Double Precision Cores for double precision calculations, 32 Special Function Units (SFUs) - for certain special function, 32 load/store units for loading data from overall GPU memory to shared memory within each SMX, 16 Texture filtering units, Shared Memory of size 6 KB. CUDA gives the programmer freedom to use some of it as shared memory and some of it as L1 cache. Shared memory is controlled by the programmer and L1 cache is taken care by architecture.

Computations are organised as blocks. Blocks are given to SMX. A block consists of many threads and threads are given to SP cores.



Figure 25: SMX Architecture

5.2 CUDA Memory Spaces

Consider the figure 26 of CUDA Memory Spaces. CPU pushes the relevant data to Global memory to be executed on GPU. It spawns GPU kernels/function which are executed on SMX. Each thread in SMX loads data from gloabl memory/device

memory to shared memory.

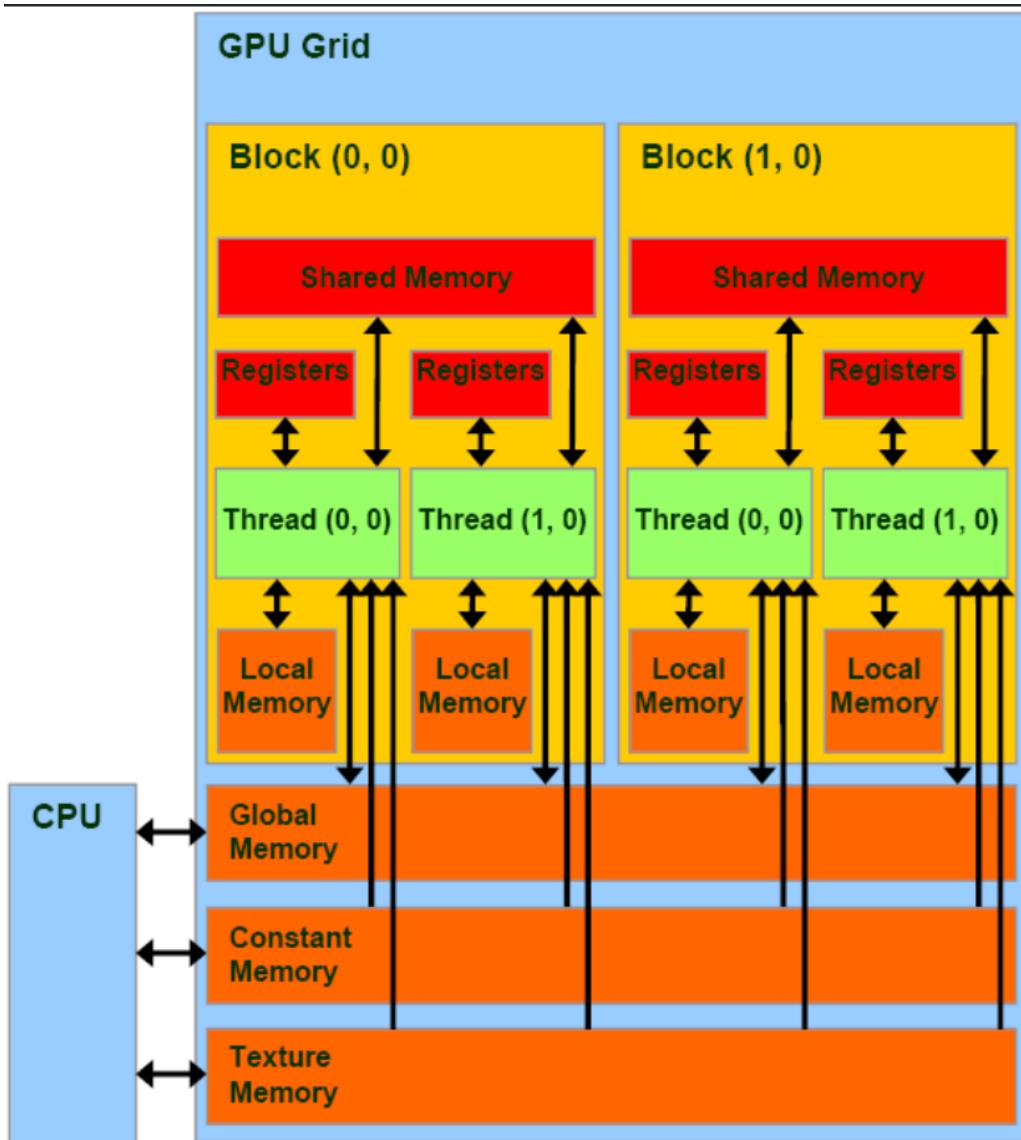


Figure 26: CUDA Memory Spaces

The memory spaces in CUDA are:

- Global Memory - It is the main memory of the GPU. It is accessible for read-/write by all the threads in all the SMXs. It can also be accessed by host -

CPU. for Kepler K40 it is around 12 GB. It has a latency of around 200-400 clock cycle (300 ns)

- Constant Memory - It is read only memory. It is used to store constants that are used by all the threads in all the blocks.
- Texture Memory - It can be accessed by all threads. It is used to store textures, used for texture mapping. It is used to improve performance of reads that exhibit spatial locality among the threads.
- Shared Memory - Each SMX has its own individual memory. It is the memory shared by all the threads in a block executing in a SMX. It is faster than global memory. Shared memory has a latency of 20-30 clock cycles (5 ns). The threads can read/write to this memory.
- Local Memory - It is the memory local to each thread, used to store temporary variables. Each thread has read/write access to this memory.
- Registers - They are present in each SMX. It is the memory local to each thread. It is used to store the context of the thread. Kepler K40 has 64K registers in each SMX.

The host can read/write global, constant and texture memory. GPUs prefer to access data from shared memory than device memory because of latency.

Difference between CPU and GPU Threads: There are a few differences between CPU and GPU threads regarding context switching. Context switching is much faster in GPUs because state of thread (thread block) stored in shared memory and threads stays till execution completion. Unlike in CPUs where in case of context switch memory related to that thread gets dumped to main memory or disk. Thus, loading the data for the thread next requires loading data from main memory or disk which is slow. Thus, context switching is faster in GPUs as there is no need to bring in any data as it already is loaded in the shared memory. Also, in case of GPUs the cache is explicitly managed by the programmer. The programmer will have to explicitly bring the frequently accessed data from the device to the shared memory. Unlike in CPUs where the cache is managed by the hardware.

6 Parallelization Principles

Parallel programs incur overheads which are not present in sequential programs. They are:

- Communication Overhead - Time taken to communicate between processors.
- Synchronization Overhead - Time taken to synchronize between processors.
- Idling Overhead - Time taken by a processor to wait for other processors.

A good parallel program tries to minimise these overheads.

6.1 Evaluation of Parallel Programs

The performance of a parallel program is evaluated using the following metrics:

- Execution Time (T_p) - Time taken by the parallel program to execute.
- Speedup (S) - It is the ratio of time taken by the *best* sequential program to the time taken by the parallel program.

$$S(p, n) = \frac{T(1, n)}{T(p, n)}$$

where, $T(1, n)$ is the time taken by the best sequential program and $T(p, n)$ is the time taken by the parallel program with p processors. Here, n indicates the data size. If we employ p processors we generally expect the speedup of p i.e. the time taken for execution in parallel to be reduced by p times. But, usually, $S(p, n) < p$. This is because of the overheads in parallel programs. The overheads are as discussed due to communication, synchronization and idling.

In some cases we may even get $S(p, n) > p$. This is called Superlinear Speedup. A large data in sequential program may not fit in cache, thus there will be a lot of cache misses. But in parallel program, the data is distributed among the processors and thus the data fits in the cache and there are less cache misses. Thus, decomposing the problem to p processor then each will require a small part of the data and thus the entire data could fit into cahces of all the individual processors and thus there will be less cache misses and thus the Speedup for that parallel program will be more than p . This is called Superlinear Speedup when the speedup is more than p (expected speedup).

- Efficiency - Now generally, we expect the speed up to be of p times. But, speedup alone does not tell the full picture since the same speedup can be achieved by using different number of processors (Say a speedup of 3 can be achieved by using 3 proessors or by using 100 processors). Thus, we normalize the speedup by the number of processors to get efficiency. Efficiency is the ratio of speedup to the number of processors. It gives the idea of how efficiently our program uses the processors. A low efficiency implies that the overheads are not handled properly and thus are dominating the execution time.

$$E(p, n) = \frac{S(p, n)}{p}$$

Generally, Efficiency is less than 1 but sometimes it can be more than 1 because of the super linear speedup.

- Scalability - It is the ability of the program to maintain efficiency as the number of processors increases. This is important because we want to be able to use more processors to solve larger problems. If the efficiency decreases as the number of processors increases then the program is not scalable. Thus, the analysis of how the program behaves with increasing the number of processors p or increasing the problem size n is called scalability analysis. Scalability thus tell limitatons of the program and how well the program scales in relation to number of processors and problem size.

Ideally, we would like the Speedup to be linear i.e. Speedup should increase linearly with increase in the number of processors. But, in reality, the speedup is sublinear. Similarly, we would like the efficiency to remain constant with

increase in the number of processors but, in practical it decreases with increase in the number of processors. This can be explained by Amdahl's Law.

Amdahl's Law: The performance improvement to be gained from using some faster mode of executing is limited by the fraction of the time the faster mode can be used. For the context of parallel programming, it states that the speedup of a program is limited by the fraction of the program that cannot be parallelized. Thus, overall speedup is given in terms of fractions of computation time with and without enhancement. Consider a sequential program which takes t_s time, which has f_s part of the program which cannot be parallelized and let f_p be the part of the program that can be parallelized. Thus, in an ideal case the time taken by the parallel program will be addition of $f_s t_s$ i.e. time taken by the fraction of the sequential program and $\frac{f_p t_s}{p}$ time taken by the parallelized part of the program. Hence, the total time taken by the parallel program is $f_s t_s + \frac{f_p t_s}{p}$. Thus, speedup is given by the ratio of the time taken by the sequential program to the time taken by parallel program. It is given by:

$$S(p) = \frac{t_s}{f_s t_s + \frac{f_p t_s}{p}} = \frac{1}{f_s + \frac{f_p}{p}}$$

where, f_s is the fraction of the program that cannot be parallelized i.e. serial and f_p is the fraction of the program that can be parallelized. Thus, the speedup is limited by the fraction of the program that cannot be parallelized. This is because not all the operations can be parallelized and some operations are dependent on the previous operations.

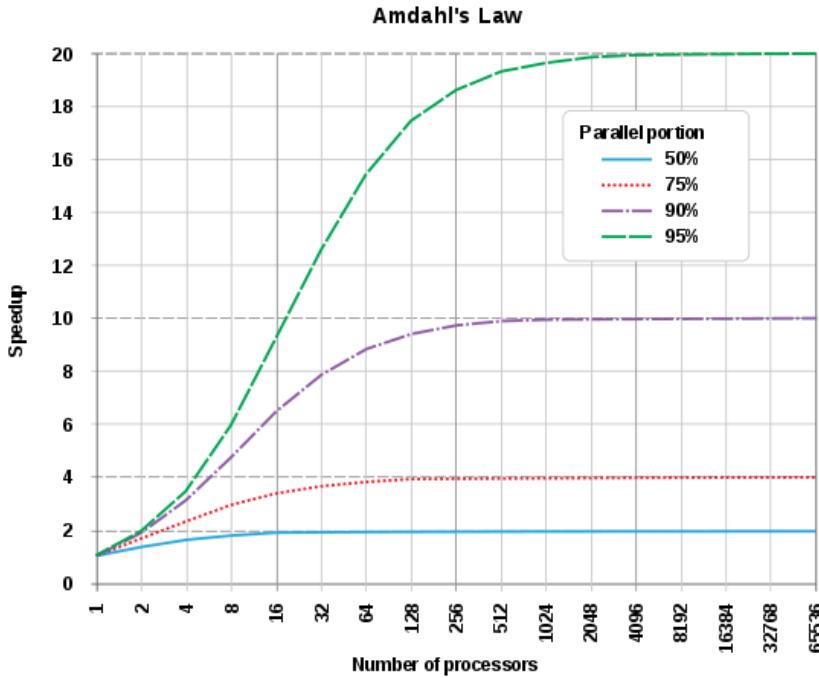


Figure 27: Amdahl's Law

It can be seen in the figure 27 that even for highly parallelizable programs ($f_p = 0.95$) the speedup curve starts to flatten out as the number of processors increases. Thus, it explains that the speedup curves starts flattening out as the number of processors increases.

The Amdahl's Law assumes that the problem size is fixed even on increasing the number of processors, thus the fractions f_s and f_p remain constant. But, in reality, the problem size increases with the number of processors. Thus, Gustafson's Law was proposed to address this issue.

Gustafson's Law: Increase the problem size proportionally so as to keep the overall time constant. Here the proportional increase means that the order of computations should increase proportionally to keep the overall time constant and not that the problem size should be increased linearly with the number of processors.

Strong Scaling: The scaling keeping the problem size constant (Amdahl's law) is called strong scaling.

Weak Scaling: The scaling due to increasing the problem size (Gustafson's

Law) is called weak scaling.

- Isoefficiency - Generally, efficiency decreases with increasing number of processors P and increases with increasing problem size N. Isoefficiency is the ability of the program to maintain efficiency as the number of processors increases and the problem size increases. For example, consider the formula for parallel Gaussian elimination in LAPACK package for solving Linear Algebra.

$$T_{par}(N, P) = \frac{2}{3} \frac{N^3}{P} t_f + \frac{(3 + 1/4 \log_2 P) N^2}{\sqrt{P}} t_v + (6 + \log_2 P) N t_m$$

$$T_{seq}(N) = \frac{2}{3} N^3 t_f$$

$$E = \frac{T_{seq}(N)}{P T_{par}(N, P)} = \left(1 + \frac{3 \sqrt{P} (3 + 1/4 \log_2 P)}{N} \frac{t_v}{t_f} + \frac{3 P (6 + \log_2 P)}{N^2} \frac{t_m}{t_f} \right)^{-1}$$

where, t_f is the time taken for a floating point operation, t_v is the time taken for a vector operation and t_m is the time taken for a memory operation. Thus, we can see that from the dominant term in the efficiency equation that as P is increased, N should be increased by $\mathcal{O}(\sqrt{P})$ to keep the efficiency constant. Now in Gaussian Elimination function the amount of computations to be carried out is $\mathcal{O}(N^3)$, thus the isoefficiency function is $\mathcal{O}(P\sqrt{P})$. The isoefficiency function is the ability of the program to maintain efficiency as the number of processors increases and the problem size increases. Thus upon increasing the problem size it is the number of processors that should be increased to keep the efficiency constant. Thus, the lesser the number of processors to increase with a large increase in the problem size the better the program isoefficiency. Thus, the isoefficiency should be as small as possible. Smaller isoefficiency functions imply higher scalability. Consider two parallel programs with isoefficiency functions $W1 = \mathcal{O}(P)$ and $W2 = \mathcal{O}(\sqrt{P})$. Then the second algorithm is considered to be more scalable since only small amount of processors need to be added. Similarly, an algorithm with an isoefficiency function of $\mathcal{O}(P)$ is highly scalable while an algorithm with quadratic or exponential isoefficiency function is poorly scalable.

7 Parallel Programming Classification and Steps

7.1 Parallel Program Models

Classification based on the way the program and Data is written and executed:

- Single Program Multiple Data (SPMD): We write a single program and run it on multiple processors. Each processor runs the same program but on different data using processor ids to execute certain parts of data. It is the most common model used in parallel programming.
- Multiple Program Multiple Data (MPMD): Different processors run different programs on different data. It is used in distributed memory systems. For example, used in climate modelling. It is not a very popular model.

7.2 Programming Paradigms

Classification based on the way the parallel program uses shared memory model or message passing:

- Shared memory Model - If there is a common global memory to access certain data. For example - Threads, OpenMP, CUDA
- message Passing model - If there is no global memory and the processes explicitly send data whenever required. For example, MPI (Message Passing Interface)

7.3 Parallelizing a Program

There is no general rule as it all depends on the application. Given a sequential program/algorithm, how to go about producing a parallel version. There are four steps to parallelize a program are:

- Decomposition - Identifying parallel tasks with large extent of possible concurrent activity; splitting the problem into tasks that can be executed in parallel.
 - Assignment - Assign the tasks to the processors and grouping the tasks into processes with best load balancing. i.e. amount of work performed by each of the processes must be equal. This is not necessarily done in Decomposition task. It specifies how to group tasks together for a process in order to have balance workload, reduce communication and management cost. This is done by
-

structured approaches such as code inspection (parallel loops) or understanding of application. Sometimes the grouping is done statically and sometimes dynamically (i.e. the grouping is done at runtime) based on the load.

Both decomposition and assignment steps are usually independent of architecture or programming model, but cost and complexity of using primitives may affect decisions.

- **Orchestration** - It is the process of managing the execution of the parallel program. It involves managing the synchronization and communication between the processes.
- **Mapping** - Mapping the processes to the processors. It is the process of mapping the processes to the processors. It is done in such a way that the processes are mapped to the processors in such a way that the communication overhead is minimized.

7.4 Data Parallelism and Data Decomposition

Data Parallelism: Given data is divided across the processing entities. Thus, each process owns and computes a portion of the data, it decides which data to communicate between other processes. This is called owner computes rule. The data is divided into chunks and each chunk is assigned to a process. This is called data parallelism where the data is divided across the processing entities and thus the parallelism is dictated by the data.

Domain Decomposition: Multi-dimensional domain in simulations divided into subdomains equal to processing entities is called domain decomposition. The given P processes are arranged in multi-dimensions forming a process grid. Consider the figure 28.

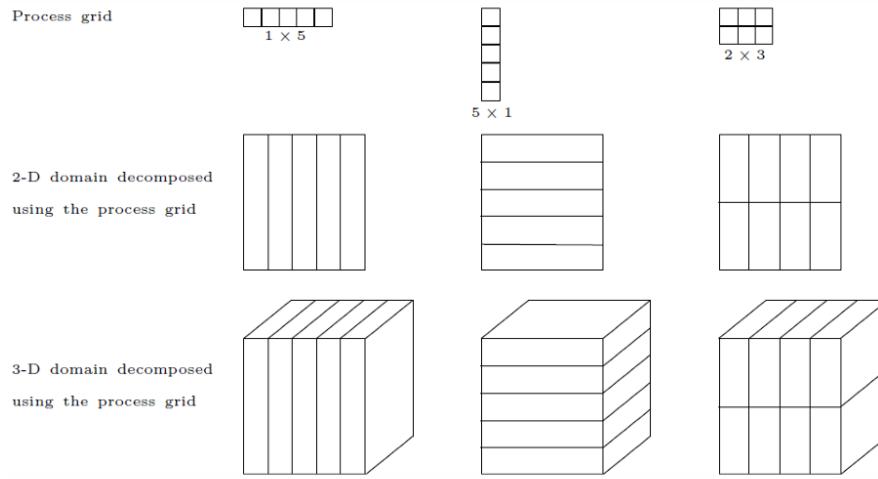


Figure 28: Domain Decomposition

It shows that if the process grid is arranged as a 1×5 grid than the 2D domain can be decomposed using the process grid as shown and the domain is divided into 5 equal parts. Each process is assigned a part of the domain. The domain decomposition is done in such a way that the communication between the processes is minimized. Similarly, same concept can be applied for a 3D domain as well as shown in the figure 28 shows many such ways for decomposing the domain (data) based on different arrangements of process grids. For a given process grid we decompose the domain so as to map the domain onto process grid such that the communication between the processes is minimized.

7.5 Data Distributions

For dividing the data in a dimension using the processes in a dimension, data distribution schemes are followed. Common data distributions are:

- Block Distribution - The data is divided into blocks and each block is assigned to a process. It is the simplest form of data distribution. It is used when the data is uniformly distributed.
- Cyclic Distribution - The data is distributed in a cyclic manner. It is used when the data is not uniformly distributed.
- Block-Cyclic Distribution - It is a combination of block and cyclic distribution. It is used when the data is not uniformly distributed.

For example, consider the figure 29.

	b_2													
b_1	↔													
0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5	3	4	5
0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
3	4	5	3	4	5	3	4	5	3	4	5	3	4	5

Figure 29: Data Distribution

Here, it can be seen that the process grid is arranged as a 2×3 grid and the blocks of data are distributed in a cyclic manner.

7.6 Task Parallelism

We decompose into tasks that can be executed in parallel. It is used when the data is not uniformly distributed. The independent tasks are identified and mapped to different processors. The tasks are grouped by a process called mapping. Here, we are trying to achieve two objectives - balance the groups and minimize inter-group dependencies. This is represented as a Task graph. For example, consider the task dependency graph shown in figure 30.

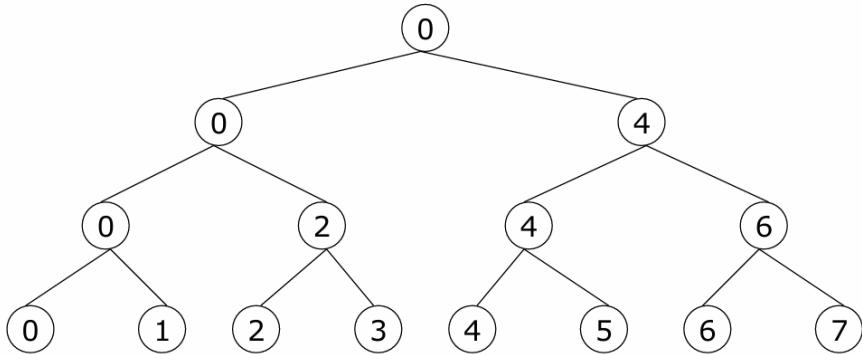


Figure 30: Task Dependency Graph

Each node shown represents a task. In general a task can be a statement or a block of statements in a program. Suppose, independent tasks follow a tree like parallelism as shown in figure 30. The 8 tasks at the bottom are independent and then they are merged to form 4 tasks and then 2 tasks and then 1 task. Suppose we wish to map this tasks to 8 processors. So, we can map the 8 tasks to 8 processors. Then to merge the tasks we assign it to one of the processor as shown in figure 30, this allows as one of the processor already has the data and the other processor can directly access the data from the processor which has the data. Thus, the communication overhead is minimized and similarly for the other tasks. In general, assigning a task graph to processors is an NP complete problem. Thus, we use heuristics to assign the tasks to the processors.

7.7 Orchestration

It is the process of managing the execution of the parallel program. It involves managing the synchronization and communication between the processes. **Maximizing data locality:** It is the process of maximizing data locality in a processor and avoid unnecessary communication. Thus, it is about minimizing the volume of data exchange. (like not communicating intermediate results) or Minimizing the frequency of interactions - packing. For example, say to compute parallel dot products. Say we have two vectors A and B and we wish to find dot product between two. It requires multiplying the corresponding elements of the two vectors and then summing them up. Thus, we can do the multiplication part in parallel across various processes or various threads. Now the communication part can be done in two ways - either we can communicate the intermediate results to the master processor and then sum

them up or we can sum them up in the individual processors and then communicate the final result to the master processor. The second approach is better as it minimizes the volume of data exchange. Thus, the orchestration is about minimizing the volume of data exchange and minimizing the frequency of interactions. Thus, structuring the communication in such a way that the communication overhead is minimized. Point to point communication between two processors involves latency and bandwidth. The latency is the time taken to establish the connection between the two processors and the bandwidth is the rate at which the data can be communicated between the two processors. By minimizing the frequency of interactions we can minimize the overall delay i.e. latency.

References

- [1] [Parallel Computing Tutorial](#) - Reference for Introduction, Classification of Architectures - Flynn's Taxonomy, and Classification based on Memory.
 - [2] [Mathew Jacob - Parallel Architecture Lectures](#) - Lecture 38 and Lecture 39 - Reference for Introduction, Classification of Architectures - Flynn's Taxonomy, and Classification based on Memory.
 - [3] Section 2.4 from Kumar, V., Grama, A., Gupta, A., Karypis, G. (1994). Introduction to parallel computing (Vol. 110). Redwood City, CA: Benjamin/Cummings for Network Topology, Shared Memory Architecture and Cache Coherence Protocols.
 - [4] Many-core - Google for NVIDIA A100 white paper
 - [5] Section 2.2 and 2.3 - Culler, D., Singh, J. P., Gupta, A. (1998). Parallel computer architecture: a hardware/software approach. Elsevier.
 - [6] [OpenMP tutorial](#) - OpenMP tutorial for parallel programming.
 - [7] [OpenMP Tutorials](#) - Youtube lectures by Intel for OpenMP.
-