**Scholars Journal of Engineering and Technology (SJET)** **ISSN 2321-435X**
**Sch. J. Eng. Tech., 2013; 1(4):218-225**
©Scholars Academic and Scientific Publisher
(An International Publisher for Academic and Scientific Resources)
www.saspublisher.com

## Review Article

# Review on Parallel and Distributed Computing

**Inderpal Singh**
Computer Science and Engineering Department, DAV Institute of Engineering and Technology, Jalandhar, India

***Corresponding author**
Inderpal Singh
Email: er.inderpal13@gmail.com

**Abstract:** Parallel and distributed computing is a complex and fast evolving research area. In its short 50-year history, the mainstream parallel computer architecture has evolved from Single Instruction Multiple Data stream (SIMD) to Multiple Instructions Multiple Data stream (MIMD), and further to loosely coupled computer cluster; now it is about to enter the Computational Grid era. The algorithm research has also changed accordingly over the years. However, the basic principles of parallel computing, such as inter-process and inter-processor communication schemes, parallelism methods and performance model, remain the same. In this paper, a short introduction of parallel and distributed computing will be given, which will cover the definition, motivation, various types of models for abstraction, and recent trend in mainstream parallel computing.

**Keywords:** Single Instruction Multiple Data stream (SIMD), Multiple Instructions Multiple Data stream (MIMD), inter-processor communication and loosely coupled

## INTRODUCTION

### Distributed And Parallel Computing

Distributed computing is the process of aggregating the power of several computing entities, which are logically distributed and may even be geologically distributed, to collaboratively run a single computational task in a transparent and coherent way, so that they appear as a single, centralized system.

Parallel computing is the simultaneous execution of the same task on multiple processors in order to obtain faster results. It is widely accepted that parallel computing is a branch of distributed computing, and puts the emphasis on generating large computing power by employing multiple processing entities simultaneously for a single computation task. These multiple processing entities can be a multiprocessor system, which consists of multiple processors in a single machine connected by bus or switch networks, or a multicomputer system, which consists of several independent computers interconnected by telecommunication networks or computer networks.

Besides in parallel computing, distributed computing has also gained significant development in enterprise computing. The main difference between enterprise distributed computing and parallel distributed computing is that the former mainly targets on integration of distributed resources to collaboratively finish some task, while the later targets on utilizing multiple processors simultaneously to finish a task as fast as possible. In this thesis, because we focus on high performance computing using parallel distributed computing, we will not cover enterprise distributed computing, and we will use the term "Parallel Computing".

### Motivation Of Parallel Computing

Parallel computing is widely used to reduce the computation time for complex tasks. Many industrial and scientific research and practice involve complex large- scale computation, which without parallel computers would take years and even tens of years to compute. It is more than desirable to have the results available as soon as possible, and for many applications, late results often imply useless results. A typical example is weather forecast, which features uncommonly complex computation and large dataset. It also has strict timing requirement, because of its forecast nature.
Parallel computers are also used in many areas to achieve larger problem scale. Take Computational Fluid Dynamics (CFD) for an example. While a serial computer can work on one unit area, a parallel computer with N processors can work on N units of area, or to achieve N times of resolution on the same unit area. In numeric simulation, larger resolution will help reduce errors, which are inevitable in floating point calculation; larger problem domain often means more analogy with realistic experiment and better simulation result.

As predicted by Moore's Law [1], the computing capability of single processor has experienced exponential increase. This has been shown in incredible advancement in microcomputers in the last few decades. Performance of a today desktop PC costing a few hundred dollars can easily surpass that of million-dollar parallel supercomputer built in the 1960s. It might be argued that parallel computer will phase out with this increase of single chip processing capability. However, 3 main factors have been pushing parallel computing technology into further development.

First, although some commentators have speculated that sooner or later serial computers will meet or exceed any conceivable need for computation, this is only true for some problems. There are others where exponential increases in processing power are matched or exceeded by exponential increases in complexity as the problem size increases. There are also new problems arising to challenge the extreme computing capacity. Parallel computers are still the widely used and often only solutions to tackle these problems.

Second, at least with current technologies, the exponential increase in serial computer performance cannot continue forever, because of physical limitations to the integration density of chips. In fact, the foreseeable physical limitations will be reached soon and there is already a sign of slow down in pace of single-chip performance growth. Major microprocessor venders have run out of room with most of their traditional approaches to boosting CPU performance-driving clock speeds and straight-line instruction throughput higher. Further improvement in performance will rely more on architecture innovation, including parallel processing. Intel and AMD have already incorporated hyperthreading and multicore architectures in their latest offering [2].

Finally, generating the same computing power, single-processor machine will always be much more expensive then parallel computer. The cost of single CPU grows faster than linearly with speed. With recent technology, hardware of parallel computers are easy to build with off-the-shelf components and processors, reducing the development time and cost. Thus parallel computers, especially those built from off-the-shelf components, can have their cost grow linearly with speed. It is also much easier to scale the processing power with parallel computer. Most recent technology even supports to use old computers and shared component to be part of parallel machine and further reduces the cost. With the further decrease in development cost of parallel computing software, the only impediment to fast adoption of parallel computing will be eliminated.

## Theoretical Model Of Parallel Computing

A machine model is an abstract of realistic machines ignoring some trivial issues, which usually differ from one machine to another. A proper theoretical model is important for algorithm design and analysis, because a model is a common platform to compare different algorithms and because algorithms can often be shared among many physical machines despite their architectural differences. In the parallel computing context, a model of parallel machine will allow algorithm designers and implementers to ignore issues such as synchronization and communication methods and to focus on exploitation of concurrency.

The widely-used theoretic model of parallel computers is Parallel Random Access Machine (PRAM). A simple PRAM capable of doing add and subtract operation is described in Fortune's paper [3]. A PRAM is an extension to traditional Random Access Machine (RAM) model used to serial computation. It includes a set of processors, each with its own PC counter and a local memory and can perform computation independently. All processors communicate via a shared global memory and processor activation mechanism similar to UNIX process forking. Initially only one processor is active, which will activate other processors; and these new processors will further activate more processors. The execution finishes when the root processor executes a HALT instruction. Readers are advised to read the original paper for a detailed description.

Such a theoretic machine, although far from complete from a practical perspective, provides most details needed for algorithm design and analysis. Each processor has its own local memory for computation, while a global memory is provided for inter-processor communication. Indirect addressing is supported to largely increase the flexibility. Using FORK instruction, a central root processor can recursively activate a hierarchical processor family; each newly created processor starts with a base built by its parent processor. Since each processor is able to read from the input registers, task division can be accomplished. Such a theoretical model inspires many realistic hardware and software systems, such as PVM [4] introduced later in this thesis.

## Architectural Models Of Parallel Computer

Despite a single standard theoretical model, there exist a number of architectures for parallel computer. Diversity of models is partially shown in Figure 1-1. This subsection will briefly cover the classification of parallel computers based on their hardware architectures. One classification scheme, based on memory architecture, classifies parallel machines into Shared Memory architecture and Distributed Memory architecture; another famous scheme, based on observation of instruction and data streams, classifies parallel machines according to Flynn's taxonomy.

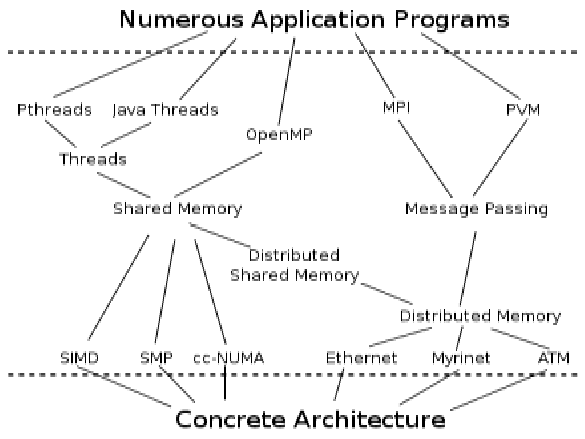

**Figure 1-1 A simplified view of the parallel computing model hierarchy**

**Shared Memory And Distributed Memory**

Shared Memory architecture features a central memory bank, with all processors and this memory bank inter-connected through high-speed network, as shown in Figure 1-2. Shared Memory shares a lot of properties with PRAM model, because of which it was favoured by early algorithm designers and programmers. Furthermore, because the memory organization is the same as in the sequential programming models and the programmers need not deal with data distribution and communication details, shared memory architecture has certain advantage in programmability. However, no realistic shared-memory high-performance machine have been built, because no one has yet designed a scalable shared memory that allows large number of processors to simultaneously access different locations in constant time. Having a centralized memory bank implies that no processor can access it with high speed.

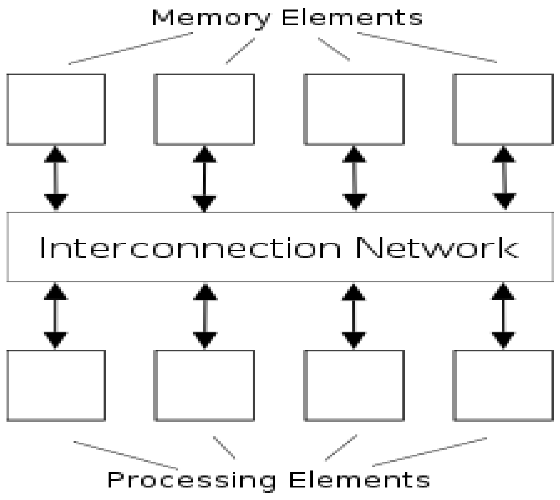

**Figure 1-2 Diagram illustration of shared-memory architecture**

In Distributed Memory architecture, every processor has its own memory component that it can access via very high speed, as shown in Figure 1-3. Accessing memory owned by other processor requires explicit communication with the owner processor. Distributed Memory architecture uses message-passing model for programming. Since it allows programs to be optimized to take advantage of locality, by putting frequently-used data in local memory and reducing remote memory access, programs can often acquire very high performance. However, it imposes a heavy burden on the programmers, who is responsible for managing all details of data distribution and task scheduling, as well as communication between tasks.

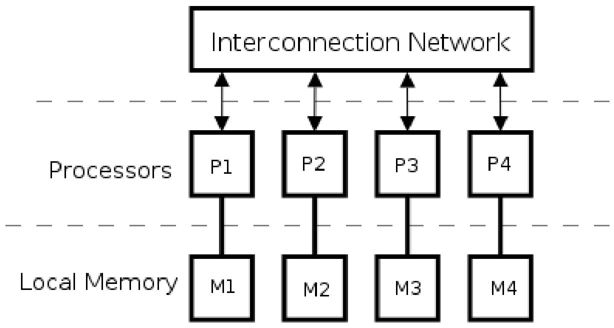

**Figure 1-3 Diagram illustration of distributed memory architecture**

To combine the performance advantage of Distributed Memory architecture to the ease of programming of Shared Memory architecture, Virtual Shared Memory, or Distributed Shared Memory (DSM) system, is built on top of Distributed Memory architecture and exposes a Shared Memory programming interface. DSM virtualizes the distributed memory as an integrated shared memory for upper layer applications. Mapping from remote memory access to message passing is done by communication library, and thus programmers are hidden from message communication details underneath. Nevertheless, for the foreseeable future, use of such paradigm is discouraged for efficiency-critical applications. Hiding locality of memory access away from programmers will lead to inefficient access to memory and poor performance until significant improvements have been gained in optimization.

The most common type of parallel computers, computer clusters, belongs to the distributed memory family. With different programming tools, the programmers might be exposed to a distributed memory system or a shared memory system. For example, using message passing programming paradigm, the programmers will have to do inter-process communication explicitly by sending and receiving message, andare based on the distributed memory architecture; but when a distributed shared memory library such as TreadMarks is used, the distributed memory nature will be hidden from the programmer. As discussed above, we would suggest the use of message passing over distributed shared memory, because communication overhead can be more significant in computer clusters. It is advantageous to allow the programmer to control the details of communication in a message passing system.

**Flynn's Taxonomy**

Another classification scheme is based on taxonomy of computer architecture firstly proposed by Michael Flynn [5] in 1966. Flynn differentiated parallel computer architectures with respect to number of data streams and that of instruction streams. According to Flynn, computer architectures can be classified into 4 categories, namely Single Instruction Single Data Stream (SISD), Single Instruction Multiple Data Stream (SIMD), Multiple Instruction Single Data Stream (MISD), and Multiple Instruction Multiple Data Stream (MIMD). This work was later referred to as Flynn's taxonomy.

In Flynn's taxonomy, normal sequential von Neumann architecture machine, which has dominated computing since its inception, is classified as SISD. MISD is a theoretical architecture with no realistic implementation. SIMD machine consists of a number of identical processors proceeding in a lock step synchronism, executing the same instruction on their own data. SIMD was the major type of parallel computer before 1980s, when the computing capability of asingle processor is very limited. Nowadays, SIMD computing is only seen inside general-purposeprocessors, as an extension to carry out vector computation commonly used, for example, in multimedia applications.

MIMD is the most commonly used parallel computers now, and covers a wide range of interconnection schemes, processor types, and architectures. The basic idea of MIMD is that each processor operates independent of the others, potentially running different programs and asynchronous progresses. MIMD may not necessarily mean writing multiple programs for multiple processors. The Single Program Multiple Data (SPMD) style of parallel computing is widely used in MIMD computers. Using SPMD, a single program is deployed to multiple processors on MIMD computers. Although these processors run the same program, they may not necessarily be synchronized at instruction level; and different environments and different data to work on may possibly result in instruction streams being carried out on different processors. Thus SPMD is simply a easy way to write programs for MIMD computers.

It is obvious that computer cluster is a type of MIMD computer. Most parallel programs on computer cluster are developed in the SPMD style. The same program image is used on each parallel processor, and each processor goes through a different execution path based on its unique processor ID.

A relevant topic is the concept of granularity of parallelism, which describes the size of a computational unit being a single "atom" of work assigned to a processor. In modern MIMD system, the granularity is much coarser, driven by the desire to reduce the relatively expensive communication.

**Performance Models Of Parallel Computing Systems Speedup, Efficiency And Scalability**

In order to demonstrate the effectiveness of parallel processing for a problem on some platform, several concepts have been defined. These concepts will be used in later chapters to evaluate the effectiveness of parallel programs. These include speedup, which describes performance improvement in terms of time savings, efficiency, which considers both benefit and cost, and scalability, which represents how well an algorithm or piece of hardware performs as more processors are added.

Speedup is a first-hand performance evaluation. However, it is a controversial concept, which can be defined in a variety of ways. Generally speaking, speedup describes performance achievement by comparing the time needed to solve the problem on N processors with the time needed on a single processor. This is shown as:

$S(n) = T(1) / T(n);$

where $S(n)$ is the speedup achieved with n processors, $T(1)$ is the time required on a single processor, and $T(n)$ is the time required on N processors. The discrepancies arise as to how the timings should be measured, and what algorithms to be used for different numbers of processors. A widely accepted method is to use optimal algorithms for any number of processors. However, in reality, optimal algorithm is hard to implement; even if it is implemented, the implementation may not performoptimally because of other machine-dependent and realistic factors, such as cache efficiency inside CPU.

A typical speedup curve for a fixed size problem is shown in Figure 1-4. As the number of processors increases, speedup also increases until a saturation point is reached. Beyond this point, adding more processors will not bring further performance gain. This is the combined result of 1) reduced computation on participating node, and 2) increased duplicate computation and synchronization and communication overhead.

The concept of efficiency is defined as
$E(n) = S(n) / n.$

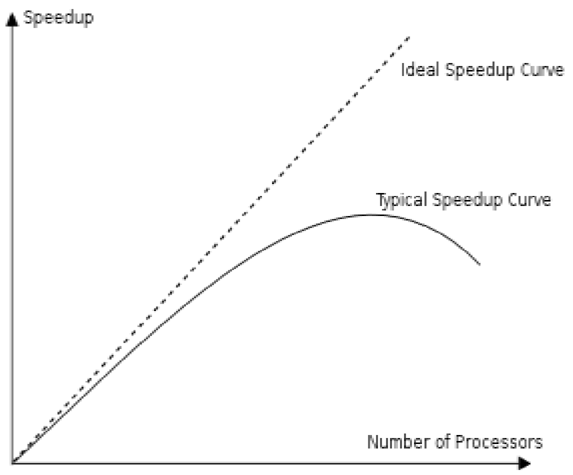

**Figure 1-4 Typical speedup curve**

It measures how much speedup is brought per additional processor. Based on the typical speedup curve shown in the figure above, it is evident that typically efficiency will be decreased upon increase in the number of processors.

The concept of scalability cannot be computed but evaluated. A parallel system is said to be scalable when the algorithm and/or the hardware can easily incorporate and take advantage of more processors. This term is viewed as nebulous [6], since it depends on the target problem, algorithm applied, hardware, current system load, and numerous other factors. Generally, programs and hardware are said to be scalable when they can take advantage of hundreds or even thousands of processors.

In practice, the computable speedup and efficiency can be much more complex. Both values are affected by many factors, which can be algorithmic and practical. Take superlinear speedup as an example. Superlinear speedup is defined as the speedup that exceeds the number of processors used. It is proved that superlinear speedup is not achievable in homogeneous parallel computers. However, when heterogeneous parallel computers are used, it is possible to achieve it [7]. An example of practical factors that may lead to superlinear speedup is cache performance: when a large number of processors are used, problem scale on a single node is largely reduced, which may result in higher cache hit ratio, fast execution, and finally probably superlinear speedup even if communication overhead is not negligible. When the parallel computer is not dedicated to a single parallel computing task, load difference among the computing nodes will imply heterogeneity and consequently the possibility of superlinear speedup. That is what we will encounter in later chapters.

## AMDAHL'S LAW

As shown in the previous subsection, efficiency gets reduced as more processors are added. This effect implies the limit of parallel performance: when the number of processors reaches some threshold, adding more processors will no longer generate further performance improvement and will even result in performance degradation, due to decrease in time saving brought by further division of task and increase in overhead of interprocess communication and duplicate computation. Gene Amdahl presents a fairly simple analysis on this [8], which is later referred to as Amdahl's Law.

Amdahl gave the speedup of a parallel program as:

$$S(n) = \frac{1}{s + \frac{p}{n}} < \frac{1}{s}.$$

where p is the fraction of code that is parallelizable, and s=1-p, is that requires serial execution. This inequality implies that superlinear speedup is not achievable and themaximal ideal speedup cannot exceed 1s, where s is the ratio of serial code (i.e., the code that requires serial execution) out of the whole program.

Amdahl's Law is a rough method to evaluate how parallel computing can be effective for a specific problem. Amdahl's Law has resulted in pessimistic view of parallel processing. For example, if 10% of the task must be computed using serialcomputation, the maximal ideal speedup is 10. Since 1967, Amdahl's Law was used as an argument against massively parallel processing (MPP).

Gustafson's discovery [9] on loophole of Amdahl's law has led the parallel computing field out of pessimism and skepticism. Since then, the so-called Gustafson's Law has been used to justify MPP. Amdahl assumed the problem size to be fixed as the number of processors changes, and thus s and p to be constants. In many scientific applications, problem size is flexible, and when more processors are available, problem size can be increased in order to achieve finer result such as higher resolution or higher precision. To quote Gustafson, "speedup should be measured by scaling the problem to the number of processors, not fixing problem size." When problem size is changed, s and p are no longer constants, and the limit set by Amdahl's Law is broken.

According to Gustafson's observation, the amount of work that can be done in parallel varies linearly with the number of processors and the amount of serial work, mostly vector startup, program loading, serial bottlenecks and I/O, does not grow with problem size. Use s' and p' to represent execution time associated with serial code and parallel code, rather than ratio, spent on the parallel system with n homogeneous processors, then if this task is to be computed on a single processor, the time needed can be represented as: T(1) = s' + np'

and the scaled speedup can be written as:

$$s'(n) = \frac{T(1)}{T(n)} = \frac{(s'+np')}{s'+p'} = n-(n-1)\cdot\frac{s'}{s'+p'} = n-(n-1)\cdot s'',$$

if s'' is defined as s'/(s'+p'). s'' is the ratio of serial code, but has different meaning from the ratio s in Amdahl's Law: s'' is the ratio of serial code with reference to whole program executed on one processor in a parallel execution, while s is with reference to all code in the whole program for the problem [10]. It must also be noted that s is a constant that is only relevant to the computation problem, under the precondition that problem scale is fixed; while s'' is a constant under the precondition of problem scale changes as Gustafson described. Under Gustafson's Law, the speedup can be linearly increased with the number of processors hired in the computation.

In the context of computational bioengineering, Gustafson's Law makes more sense than Amdahl's Law, because with larger computing capability, it is desirable to acquire better result, in terms of resolution in image processing and simulation and in terms of higher precision in many numerical applications. When the problem size is fixed, Amdahl's Law has told us to reduce the fraction of code that has to be executed in serial. Essentially, we have to reduce the fraction of code whose execution time cannot be reduced by introducing more processors. Since communication code has this feature, we will look into the techniques to optimize inter-processor communication.

**Interconnection Schemes Of Parallel Computing Systems**

Both Amdahl's Law and Gustafson's Law acknowledge the significance of serial code in affecting the parallel computer performance. Another important factor that is closely related to parallel program performance is inter-process communication and synchronization. Especially with modern technology, processing capability of single chip has been tremendously increased; however, inter-process communication has received relatively small improvement, and thus become the bottleneck of overall performance. That also explains the trend of coarser-granularity parallelism. High- performance parallel computers, especially those able to scale to thousands of processors, have been using sophisticated interconnection schemes. Here we cover the major interconnection schemes listed in Figure 1-5 in brief.

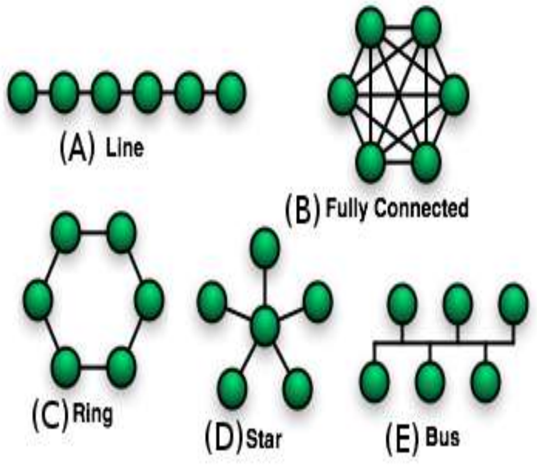

**Figure 1-5 Illustrations of Simple interconnection schemes**

Figure 1-5(A) illustrates the line scheme, which is the simplest connection scheme. In this illustration, circle represents a computing node and line represents direct communication channel between nodes. Computing nodes are arranged on and connected with a single line. Except for the nodes at the two ends, vertex degrees are all 2 and thus the implementation of network interface is simple; routing is simple and the topology can be viewed as recursive. However, communication between any two non-neighbor nodes needs the help of other nodes; the connectivity is only 1 and fault at any node will make the whole system break; and diameter of this corresponding graph is n-1, where n is the number of nodes, which implies that the latency could be very high. To summarize, this scheme is simple and low-cost, but will not be able to generate high performance or reliability; and as system scales, the performance degrades rapidly.

Figure 1-5(B) illustrates the ring scheme, which is an enhanced line topology, with an extra connection between the two ends of the line. This increases theconnectivity to 2 and decreases the diameter to half of the corresponding line topology. However, basic characteristics are still the same.

The other extreme is probably the fully-connected topology, in which there is a direct connection between any two computing nodes. Fully-connected topology is shown in Figure 1-5(C). The corresponding graph representation has an edge between any two vertices, and distance between any two vertices is 1. Thus the diameter is 1, and it generates the minimal communication latency, if the physical link implementation is fixed, as well as the maximal connectivity. However, the degree of nodes changes with the number of processors and thus the implementation of network interface must be very complex; and it is hard to be recursive, adding another layer of complexity of implementation and reducing the scalability. To summarize, this scheme will generate the highest performance possible, but due to the complexity and thus cost, it can hardly be scalable: with larger scale, although performance will not degradeatall, complexitywillclimbveryfastatthelevelofO(n2).

Similar to fully-connected network, bus network, illustrated in Figure 1-5(E), has direct connection between any two nodes. In fact, bus topology shares the same logical graph representation with fully-connected topology and. Consequently; static characteristics of bus topology are exactly the same as those of fully-connected topology. But connection between any pair of nodes is not dedicated but shared: interconnection is implemented via a shared bus. This reduces the complexity significantly. In fact, its complexity is similar to that of line and ring topology. However, the dynamic characteristics, such as data transfer speed, are more inferior to those of fully-connected counterpart. Although collective communication is now very easy to implement, this single shared bus prevents more than one pair of nodes tocarry out point-to-point communication. As a result, the system does not scale very well.

An intuitive improvement on bus network is to change the bus to eliminate the constraint that only two nodes can communicate at any time. The result is the star network, where a communication switch node is added to replace the shared bus, as shown in Figure 1-5(D). If we treat this switch node as a non-computing node and ignore it in the graph representation, then star network corresponds to the same fully- connected graph as bus network, while the implementation does not have the constraint of bus network; if switch node is viewed as normal computing node, then the corresponding graph has a diameter of 2, supports easy implementation of collective communication with the help of the central switch node, and allows recursive expansion. Except for the switch node, all other nodes have a constant vertex degree of 1. The critical disadvantage is that the connectivity is 1: failure at the switch node will cause the whole system to fail.

For computer clusters, most are built with a star structured interconnection network around a central switch. For better fault tolerance or easier setup, the other interconnection scheme might also be used. Parallel program using message passing might be rewritten to better adapt to different interconnection network.

There are other types of more sophisticated topology schemes, such as tree, mesh, and hypercube, which are widely used in parallel computers with thousands of processors or more. These schemes often scale better to larger scale network with good performance. Readers are advised to [11] for more information about this.

## Programming Models Of Parallel Computing Systems

Programming models are high-level abstract views of technologies and applications that hide most architectural details with programmers as the main audience. For MIMD machine like a computer cluster, the most important models include shared-memory model, message-passing model, and object-oriented model.

In the shared-memory model, multiple tasks run in parallel. These tasks communicate with one another by writing to and reading from a shared memory. Shared-memory programming is comfortable for the programmers, because the memory organization is similar as in the familiar sequential programming models, and programmers need not deal with data distribution or communication details. Popularity of this model was also promoted by its similarity to the theoretical PRAM model. Practice of programming on this model originated from concurrent programming on transparency of data and task placement determines that, besides the simplicity of programming, the performance cannot be predicted or controlled on hardware platform with Non-Uniform Memory Architecture (NUMA) or distributed memory. This performance problem is evident especially on large-scale multiprocessor systems, in which access time to memory at different locations varies significantly and thus memory locality plays critical role in determining overall system performance.

Message passing model is becoming the prevailing programming model for parallel computing system, thanks to the trend to large-scale multiprocessors systems,including computer clusters. In this model, several processes run in parallel and communicate and synchronize with one another by explicitly sending and receiving message. These processes do not have access to a shared memory or a global address space. Although harder to program compared to previous models, it allows programs to explicitly guide the execution by controlling data and task distribution details. Since everything is under the programmer’s control, the programmer can achieve close to optimum performance if enough effort has been spent on performance tuning. Besides this performance advantage, message passing model is also versatile. Being a relatively low-level model, it is capable of implementing many higher-level models. A typical example is the widely-used SPMD data model, which fits in with many naturally data-parallel scientific applications. Very low-level message passing systems, such as Active Message, are even used to implement shared-memory system by emulating a shared memory. These systems, while allowing easy high-level algorithm design with the help of more friendly high-level models, expose enough low-level details to support and encourage the programmers to manually control and tune the performance. Wide deployment of multicomputers and loosely-coupled computer clusters, which feature expensive communication, promotes the popularity of message passing systems. Message Passing Interface (MPI) [12] and Parallel Virtual Machine (PVM) [4] are the 2 major programming libraries used to build message passing system.

## CONCLUSION:

In this paper, we have reviewed some basic concepts in parallel computing systems. Parallel computing is the simultaneous execution of the same task onmultiple processors in order to obtain faster results. Computer cluster belongs to distributed memory MIMD computers.

## REFERENCE

1. Gordon Moore; Cramming more components onto integrated circuits. Electronics Magazine, 19 April 1965.
2. Herb Sutter; The free lunch is over: a fundamental turn toward concurrency in software. Dr. Dobb's Journal, 2005:30(3).
3. Steven F, Wyllie J; “Parallelism in random access machines,” in Proceedings of the tenth annual ACM symposium on Theory of computing, San Diego, California, United States, 1978:114-118.
4. Sunderam VS; PVM: a framework for parallel distributed computing. Concurrency: Practice and Experience,1990; 2(4):315-339.
5. Michael J. Flynn, “Very high-speed computing systems,” in Proceedings of the IEEE, 1966;54:1901-1909.
6. Lou Baker, Bradley J. Smith; Parallel Programming, New York, McGraw- Hill, 1996.
7. Donaldson V; Parallel speedup in heterogeneous computing network. Journal of Parallel Distributed Computing, 1994; 21:316-322.
8. Amdahl GM; “Validity of the single processor approach to achieving large scale computer capability” in Proceedings of AFIPS Spring Joint Computer Conference, pp. 30, Atlantic City, New Jersey, United States, 1967.
9. Gustafson JL; Reevaluating Amdahl’s law. Communications of ACM, 1988; 31(5):532-533.
10. Yuan Shi. Reevaluating Amdahl’s law and Gustafson’s law. Available:http://joda.cis.temple.edu/~shi/docs/amdahl/amdahl.html
11. David C, Singh JP, Anoop Gupta; Parallel Computer Architecture : A Hardware/Software Approach. Morgan Kaufmann, 1998.
12. Message Passing Interface Forum, MPI: A message-passing interface standard, May 1994.