Simulated Performance and Task Scheduling Analysis of Multiprocessor in Parallel (Environment)

Gagandeep Singh
M.tech (ACET, ASR)

Chhailadeep Kaur
Assit Prof. (ACET, ASR)

Abstract:
Performance analysis of scheduling Algorithms in Simulated Parallel condition is deals with the optimal assignment of a set of tasks to the parallel multiprocessor system and commands to their execution for minimizing the total completion time. When we submit the Tasks to multiprocessors systems then we really want to know how well such tasks are performing. The actual experimentation on multiprocessor is still a costly and complex approach. These systems are still out of the reach of young researches for doing research in higher education institutes in developing countries. There have so many reasons for non-availabilities of these systems.
So all these shortcomings convinced us to switch towards simulation of multiprocessor environment for the performance measurement of processor.
This paper is an effort to provide a GUI based simulated multiprocessors environment for the performance measurement or analysis scheduling Algorithms in Simulated Parallel environment.

Keywords: 1) Multiprocessor Environment Parallelization, Simulated Framework performance evolution.

INTRODUCTION
In parallel processing, the parallel portion of the application can be accelerated according to the number of processors allocated to it. In a homogeneous architecture, where all processors are identical, the sequential portion of the application will have to be executed in one of the processors, considerably degrading the execution time of the application.
Two main distinguishing features of parallel versus sequential programming are program
Partitioning and task scheduling. Both techniques are essential to high- performance computing on both homogeneous and heterogeneous systems. The partitioning
problem deals with how to detect parallelism and determine
the best trade-off between parallelism and overhead, the scheduling problem deals with choosing the order in which a certain number of tasks may be performed and their assignment to processors in a parallel/distributed environment

Interconnection Networks
The interconnection network [6] plays a central role in determining the overall performance of a multicomputer system. If the network cannot provide adequate performance, for a particular application, nodes will frequently be forced to wait for data to arrive.
Some of the more important networks

- Fully connected or all-to-all
- Mesh
- Rings
- Hypercube
- X - Tree
- Shuffle Exchange
- Butterfly
- Cube Connected Cycles
- Fully connected or all-to-all

This is the most powerful interconnection network (topology): each node is directly connected to ALL other nodes.

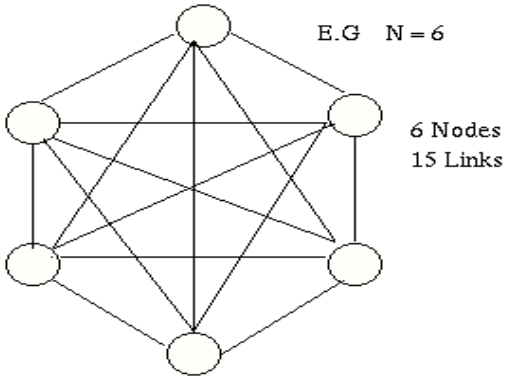

Fig.1 Fully connected
Each node has N-1 connections (N-1 nearest neighbors) giving a total of N(N-1) / 2 connections for the network. Even though this is the best network to have the high number of connections per node mean this network can only be implemented for small values of N. Therefore some form of **limited** interconnection network must be used.

- **Mesh (Torus)**

In a mesh network, the nodes are arranged in a **k dimensional lattice** of width w, giving a total of w^k nodes. [usually k=1 (linear array) or k=2 (2D array) e.g. ICL DAP] Communication is allowed only between neighbouring nodes. All interior nodes are connected to 2k other nodes.

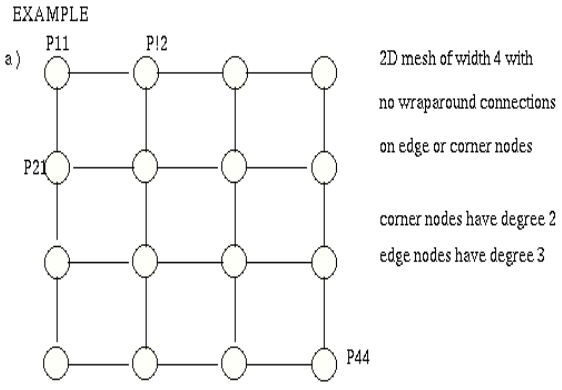

Fig. 1.1D mesh of width 4

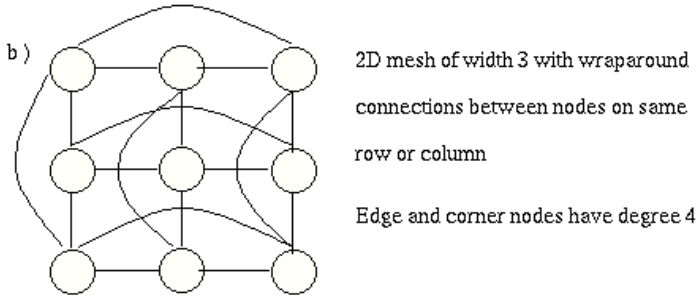

Fig.1.2D mesh of width 3

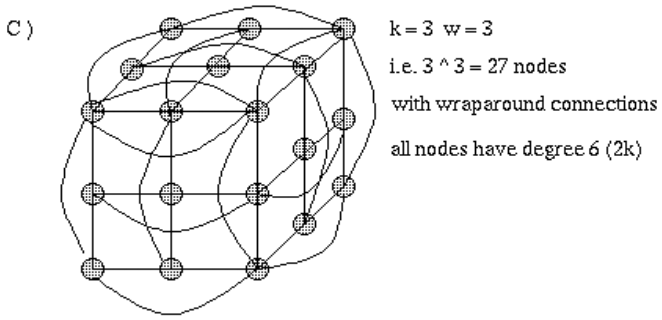

Fig.1.3 D mesh of width 3

- **Rings**

A **simple ring** is just a linear array with the end nodes linked.

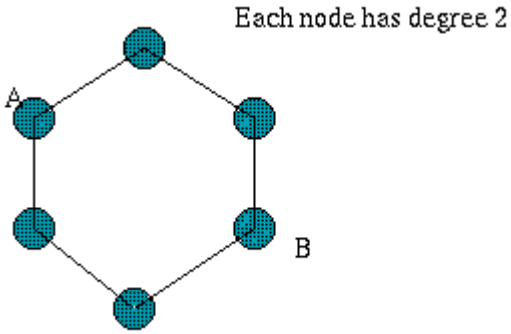

Fig.1.4 Ring

It is equivalent to a 1D mesh with wraparound connections. One drawback to this network is that some data transfers may require N/2 links to be traversed.

This can be reduced by using a **chordal ring** this is a simple ring with cross or chordal links between nodes on opposite sides.

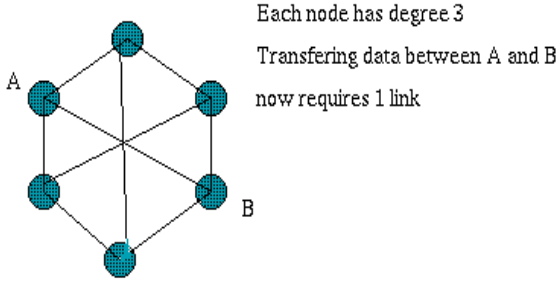

Fig.1.5 Chordal ring

- **Hypercube Connection (Binary n-Cube)**

Hypercube networks consist of N = 2^k nodes arranged in a k dimensional hypercube.

The nodes are numbered 0, 1...2^k -1 and two nodes are connected if their binary labels differ by exactly one bit.

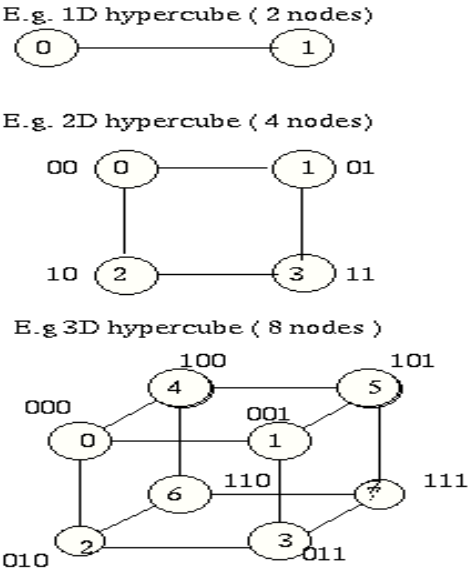

Fig.1.6 1D, 2D, 3D Hypercube Connections

4D Hypercube or Binary 4-Cube

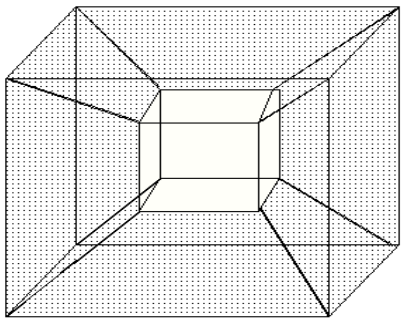

Fig.1.7 4D Hypercube

K dimensional hypercube is formed by combining two k-1 dimensional hypercube and connecting corresponding nodes i.e. hypercube are recursive. Each node is connected to k other nodes i.e. each is of degree k

The departmental NCUBE is based on this topology i.e. a 5 dimensional hypercube (64 nodes)

1. Metrics for Interconnection Networks

Metrics provide a framework to compare and evaluate interconnection networks [6]. The metrics we will use are:

1. Network connectivity
2. Network diameter
3. Narrowness
4. Network expansion increments

Network Connectivity

Network nodes and communication links sometimes fail and must be removed from service for repair. When components do fail the network should continue to function with reduced capacity.

Network connectivity measures the resiliency of a network and its ability to continue operation despite disabled components i.e.

connectivity is the minimum number of nodes or links that must fail to partition the network into two or more **disjoint networks**
The larger the connectivity for a network the better the network is able to cope with failures.

Network Diameter

The diameter of a network is the maximum internodes distance i.e. it is the maximum number of links that must be traversed to send a message to any node along a shortest path.
The lower the diameter of a network the shorter the time to send a message from one node to the node farthest away from it.

Narrowness

This is a measure of congestion in a network and is calculated as follows:
Partition the network into two groups of processors A and B where the number of processors in each group is Na and Nb and assume Nb < = Na. Now count the number of interconnections between A and B call this I. Find the maximum value of Nb / I for all partitionings of the network. This is the narrowness of the network.

The idea is that if the narrowness is high (Nb > I) then if the group B processors want to send messages to group A congestion in the network will be high (since there are fewer links than processors)

Network Expansion Increments

A network should be possible to create larger and more powerful multicomputer systems by simply adding more nodes to the network. E.g. an 8 node linear array can be expanded in increments of 1 node but a 3 dimensional hypercube can be expanded only by adding another 3D hypercube. (i.e. 8 nodes)

1.2 Parallel Environment (PE)

Parallel Environment is a high-function development and execution environment for parallel applications (distributed-memory, message-passing applications running across multiple nodes). It is designed to help organizations develop, test, debug, tune and run high-performance parallel applications written in C, C++ and Fortran on pSeries clusters. Parallel Environment runs on AIX® or Linux®. Parallel Environment includes the following components:

- The Parallel Operating Environment (POE) for submitting and managing jobs.
- IBM's MPI and LAPI libraries for communication between parallel tasks.
- PE Benchmarker, a suite of applications and utilities to analyze program performance.
- A parallel debugger (pdbx) for debugging parallel programs.
- Parallel utilities to ease file manipulation.

Overview of Scheduling Mechanism in Multiprocessor Systems

A **Scheduling** is a key concept in computer multitasking and multiprocessing operating system design, and in real-time operating system design. In modern operating systems, there are typically many more processes running than there are CPUs available to run them. **Scheduling** refers to the way processes are assigned to run on the available CPUs. This assignment is carried out by software known as a **scheduler**.

The **scheduler** is concerned mainly with:

- CPU utilization - to keep the CPU as busy as possible.
- Throughput - number of process that complete their execution per time unit.
- Turnaround - total time between submission of a process and its completion.
- Waiting time - amount of time a process has been waiting in the ready queue.
- Response time- amount of time it takes from when a request was submitted until the first response is produced.
- Fairness - Equal CPU time to each thread.

In real-time environments, such as mobile devices for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable.
A scheduling algorithm is the algorithm, which dictates how much CPU time is allocated to Processes and Threads. The goal of any scheduling algorithm is to fulfill a number of criteria:

- No task must be starved of resources - all tasks must get their chance at CPU time;
- If using priorities, a low-priority task must not hold up a high-priority task;
- The scheduler must scale well with a growing number of tasks, ideally being O (1). This has been done, for example, in the Linux kernel.

In order to decrease the impact of failures on an application, matching and scheduling algorithms must be devised which minimize not only the execution time but also the failure probability of the application. It is not possible to minimize both at the same time. Thus, the goal of this paper is to develop matching and scheduling algorithms, which account for both the execution time and the failure probability and can trade off execution time against the failure probability of the application. In order to attain these goals, a bi-objective scheduling problem is first formulated and then two different algorithms, the bi-objective dynamic level scheduling algorithm and the bi-objective genetic algorithm, are developed. The simulation results confirm that the proposed algorithms can be used for producing task assignments where the execution time is weighed against the failure probability.
Scheduling Algorithm is the method used to determine which of several processes, each of which can safely have a resource allocated to it, will actually be granted use of the resource.

First In First Out

FIFO is an acronym for **First In, First Out**, an abstraction in ways of organizing and manipulation of data relative to time and prioritization. This expression describes the principle of a queue processing technique or servicing conflicting demands by ordering process by first-come, first-served (FCFS) behavior: what comes in first is handled first, what comes in next waits until the first is finished, etc.
FCFS is also the shorthand name for the FIFO operating system scheduling algorithm, which gives every process CPU time in the order they come.
The simplest scheduling algorithm [10], FIFO simply queues processes in the order that they arrive in the ready queue.

- Since context switches only occur upon process termination, and no reorganization of the process queue is required, scheduling overhead is minimal.
- Throughput can be low, since long processes can hog the CPU
- Turnaround time, waiting time and response time can be low for the same reasons above
- No prioritization occurs, thus this system has trouble meeting process deadlines.
- The lack of prioritization does permit every process to eventually complete, hence no starvation.

This scheduling method is used on Batch-Systems, it is NON-PREEMPTIVE. It implements just one queue, which holds the tasks in order they come in. Its Strengths are its Simple and Fair but the Problems are Convoy Effect and Order of task arrival is very important for average Turnaround time.

1.2.2 Shortest Job First (SJF)

It selects the shortest Job/Process that is available in the run queue.
This scheduling algorithm assumes that run times are known in advance. It is NON-PREEMPTIVE
Strengths:
-Nearly optimal (Turnaround Time)
Problems:
-Only optimal if all jobs/process are available simultaneously
-Usually run times are not known ...

Shortest-Job-First (SJF)

- Another name is *Shortest Process Next algorithm*
- A better name may be *shortest next-CPU-burst first*
- Assumes we know the length of the next CPU burst of all ready processes
- The length of a CPU burst is the length of time a process would continue executing if given the processor and not preempted
- SJF estimates the length of the next burst based on the lengths of recent CPU bursts
- Starts with a default expected burst length for a new process
- Suppose that time intervals are numbered 1 for first CPU burst, 2 for second CPU burst, etc.
- Default length is e(1), the expected length of time for the first CPU burst
- Unlike other scheduling algorithms, this algorithm assumes that information about a process's burst length is stored between the times when it is ready.
- In keeping with the need for efficiency, only a small amount of info is stored and only a simple calculation is performed
- We can weight the previous expectation (representing all previous bursts) and the most recent burst with any two weights that add up to 1, e.g., say 0.5 and 0.5, or 0.9 for previous expectations and 0.1 for actual time for most recent CPU burst.

1.2.3 Round Robin

Round-robin (RR) is one of the simplest scheduling algorithms for processes in an operating system, which assigns time slices to each process in equal portions and in circular order, handling all processes without priority.
For example, if the timer runs at 100Hz, and a process' quantum is 10 ticks, it may run for 100 milliseconds (10/100 of a second).
In the Round Robin algorithm, each process is given an equal quantum; the big question how to choose the time quantum.
Advantages of Round Robin include its simplicity and strict "first come, first served" nature. Disadvantages include the absence of a priority system: lots of low privilege processes may starve one high privilege one.

Round-robin scheduling

The scheduler assigns a fixed time unit per process, and cycles through them.

- RR scheduling involves extensive overhead, especially with a small time unit.
- Balanced throughput between FCFS and SJN, shorter jobs are completed faster than in FCFS and longer processes are completed faster than in SJN.
- Fastest average response time, waiting time is dependent on number of processes, and not average process length.
- Because of high waiting times, deadlines are rarely met in a pure RR system.
- Starvation can never occur, since no priority is given. Order of time unit allocation is based upon process arrival time, similar to FCFS.

PROBLEM DEFINITION

A parallel algorithm, as opposed to a traditional sequential (or serial) algorithm, is an algorithm which can be executed a piece at a time on many different processing devices, and then put back together again at the end to get the correct result.
On a given problem, once the parallelism is extracted and the execution is described as a dynamic task graph, the problem is to schedule this task graph on the resources of the parallel architecture. This motivates theoretical studies: to design algorithms whose related task graphs can be scheduled on various architectures with proved bounds is a main research axis; to provide scheduling algorithms suited to machine models; to develop quantitative models of parallel executions.
Some algorithms are easy to divide up into pieces like this. For example, splitting up the job of checking all of the numbers from one to a hundred thousand to see which are primes could be done by assigning a subset of the numbers to each available processor, and then putting the list of positive results back together.
Most of the available algorithms to compute pi (π), on the other hand, cannot be easily split up into parallel portions. They require the results from a preceding step to effectively carry on with the next step. Such problems are called inherently serial problems. Iterative numerical methods, such as Newton's method or the three-body

problem, are also algorithms, which are inherently serial. Some problems are very difficult to parallelize, although they are recursive. One such example is the depth-first search of graphs.

Parallel algorithms are valuable because of substantial improvements in multiprocessing systems and the rise of multi-core processors. In general, it is easier to construct a computer with a single fast processor than one with many slow processors with the same throughput. But processor speed is increased primarily by shrinking the circuitry, and modern processors are pushing physical size and heat limits. These twin barriers have flipped the equation, making multiprocessing practical even for small systems.
The cost or complexity of serial algorithms is estimated in terms of the space (memory) and time (processor cycles) that they take. Parallel algorithms need to optimize one more resource, the communication between different processors. There are two ways parallel processors communicate, shared memory or message passing.
Shared memory processing needs additional locking for the data, imposes the overhead of additional processor and bus cycles, and also serializes some portion of the algorithm.
Message passing processing uses channels and message boxes but this communication adds transfer overhead on the bus, additional memory need for queues and message boxes and latency in the messages. Designs of parallel processors use special buses like crossbar so that the communication overhead will be small but it is the parallel algorithm that decides the volume of the traffic.
Parallel algorithm design is not easily reduced to simple recipes. Rather, it requires the sort of integrative thought that is commonly referred to as ``creativity." However, it *can* benefit from a methodical approach that maximizes the range of options considered, that provides mechanisms for evaluating alternatives, and that reduces the cost of backtracking from bad choices. Goal of this paper is to suggest a framework within which parallel algorithm design can be explored. In the process, we hope this will develop intuition as to what constitutes a good parallel algorithm. Various cost annotations, such as number of operations, Response time, Turnaround time, Waiting Time, Total Turnaround time have been introduced in order to lead to more realistic complexity analysis on distributed architectures.

1.2.4 Comparison between different Scheduling Algorithms

Scheduling algorithm	CPU Utilization	Throughput	Turnaround time	Response time	Deadline handling	Starvation free
First In First Out	Low	Low	High	High	No	Yes
Shortest remaining time	Medium	High	Medium	Medium	No	No
Round-robin scheduling	High	Medium	Medium	Low	No	Yes

Performance Metrics of Parallel Systems

Speedup: Speedup *Tp* is defined as the ratio of the serial runtime of the best sequential algorithm for solving a problem to the time taken by the parallel algorithm to solve the same problem on *p* processor. The *p* processors used by the parallel algorithm are assumed to be identical to the one used by the sequential algorithm.
Cost: Cost of solving a problem on a parallel system is the product of parallel runtime and the number of processors used $E = p.Sp$
Efficiency: Ratio of speedup to the number of processors.
Efficiency can also be expressed as the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on *p* processors. The cost of solving a problem on a single processor is the execution time of the known best sequential algorithm

Details about working of the Simulator

1.4.1 Simulator features

- **Training, evaluation and analysis system –** able to create and manage users and exercises. The instructor can assign and evaluate exercises checking its execution data.
- **GUI based –** Easy to use & understand.
- **Graph generator**--By just clicking on Graph Generator button in simulator it generates the performance graphs in Excel worksheet. With the help of VBA Macro coded in Excel worksheet, data from MS-Access is passed to excel worksheet and hence graphs are generated from this data.
- **Run-time performance measurement** --It measures the completion time as well as average completion time during run-time at regular intervals [15].

2 Steps for the working of simulator: -

1. Initially value for the jobs are assigned.
2. Then information about number of processors is fed to the simulator as shown in snapshot.
3. On the basis of number of process arrived it equally divides the load among different processors.
4. When jobs are divided among processors they started giving response and at run time choose the scheduling algorithm according to requirement.
5. Then show the simulator and start the simulation where jobs start running and we get the response time, Waiting time and Turnaround time. As information about Turnaround time of various jobs at any instant is available, Total turnaround time of the processor is generated.

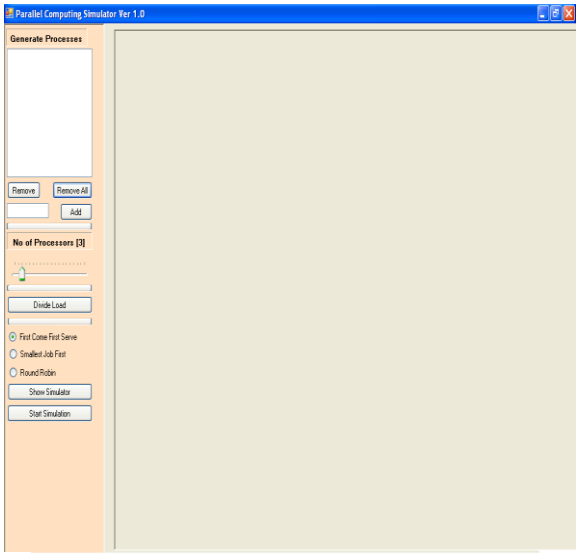

Fig.1 Simulator in Idle state.

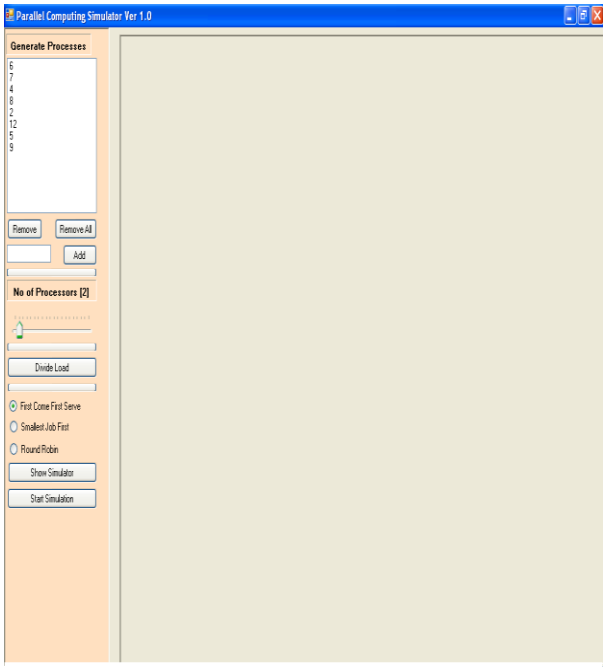

Fig.2 Processes have been added.

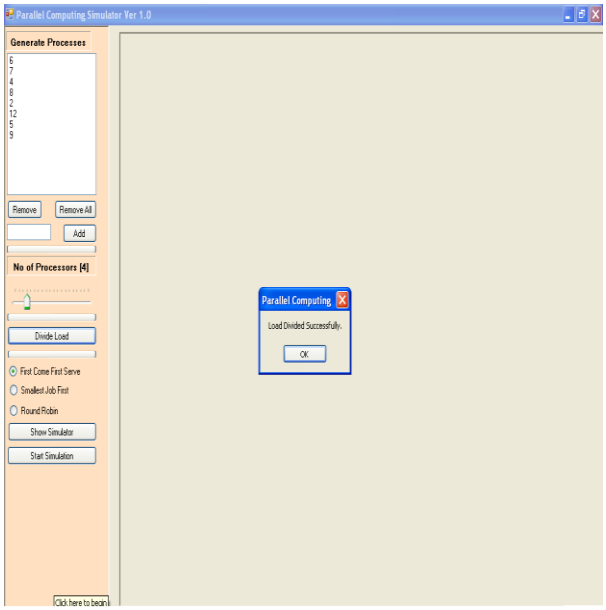

Fig.3 Number of processes is decided and load is divided.

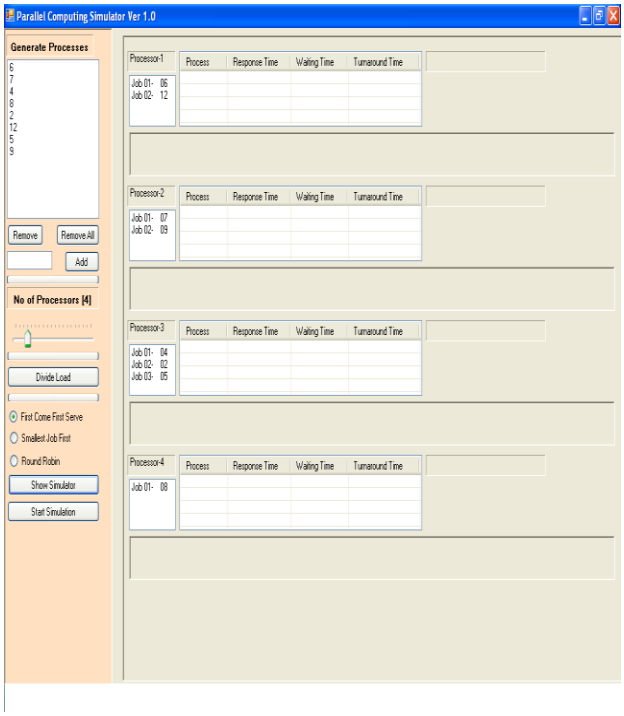

Fig.4 FCFS based strategy is choosed and simulator is showed.

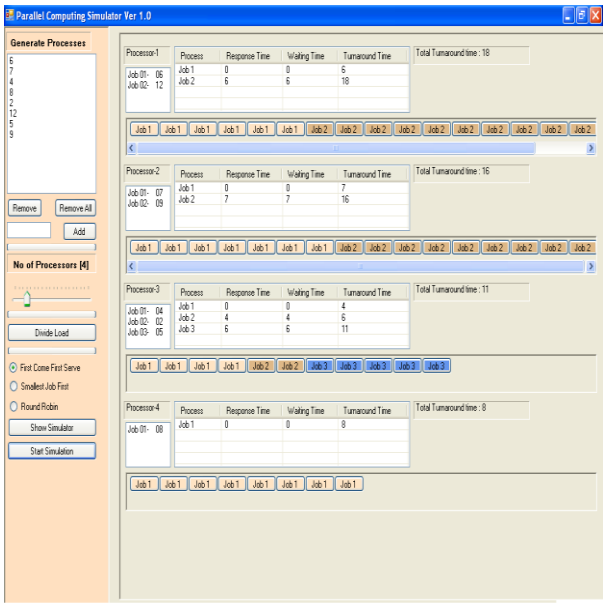

Fig.5 Simulator is started.

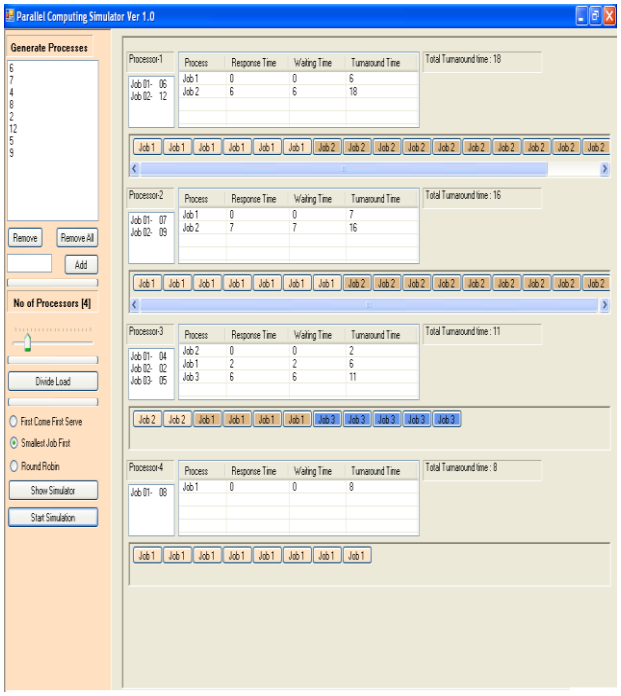

Fig.6 Smallest job first strategy has been choose & simulation is generated.

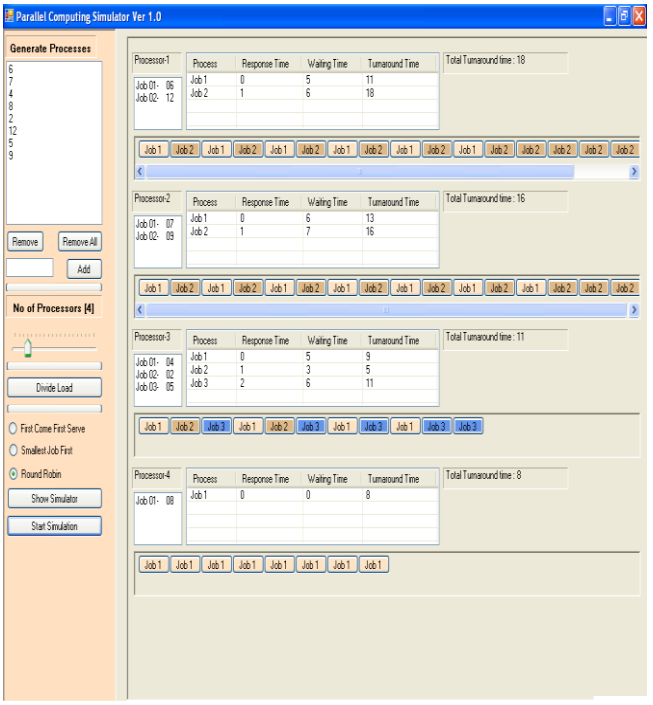

Fig.21 Round Robin strategy has been choosed and simulation is generated.

OBJECTIVE

In the design of scheduling algorithms for efficient parallel processing, there are four fundamental aspects for the design of scheduling algorithms: Performance, Time-complexity, Scalability and Applicability.

By high performance we mean the scheduling algorithms should produce high quality solutions. The algorithms must be robust so that they can be used under a wide range of input parameters. Scheduling algorithms should have low time-complexity. The time-complexity of an algorithm is an important factor so far as the quality of solution is not compromised. Parallel scheduling algorithms must be scalable. On the one hand, the problem should possess problem-size scalability, that is, the algorithms consistently give a good performance even for large input. On the other hand, the algorithms should possess processing-power scalability, that is, given more processors for a problem, the parallel scheduling algorithms produce solutions with almost the same quality in a shorter period of time. Scheduling algorithms could be used in practical environments. To achieve this goal one must take into account realistic assumptions about the program and multiprocessor models.

CONCLUSION & FUTURE DIRECTIONS

This Paper demonstrated the advantages of deploying a scheduling algorithm method in a parallel system. It had presented an scheduling algorithm method and demonstrated its favorable properties, both by theoretical means and by simulations.

The value of the proved minimal disruption property of the mapping adaptation has been demonstrated in the extensive set of simulations. Such a scheme is particularly useful in systems with many input ports and packets requiring large amounts of processing. With the proposed scheme, a kind of statistical multiplexing of the incoming traffic over the multiple processors is achieved, thus in effect transforming a network node into a parallel computer. The improvements of processor utilization decrease the total system cost and power consumption, as well as improve fault tolerance.

Work done in this Thesis was an effort to design and develop a simulated multiprocessor environment so as to virtualize the actual Scheduling system. This paper presents a simulation environment developed with the aim to facilitate the research of multiprocessor systems as well as performance measurement of scheduling algorithms in developing countries. A simulator program was coded in VB.net to fulfil this purpose. In future the work done in this thesis can be extended by modelling many more scheduling algorithms in the developed environment. Effort will be done in future to validate the data captured by simulator with actual experimental setup.

REFERENCES

1. Almasi, G.S. and A. Gottlieb (1989). *Highly Parallel Computing*. Benjamin-Cummings publishers, Redwood City, CA.
2. Hillis, W. Daniel and Steele, Guy L., *Data Parallel Algorithms* Communications of the ACM December 1986
3. Quinn Michael J, *Parallel Programming in C with MPI and OpenMP* McGraw-Hill Inc. 2004. ISBN 0-07-058201-7
4. IEEE Journal of Solid-State Circuits:"A Programmable 512 GOPS Stream Processor for Signal, Image, and Video Processing", Stanford University and Stream Processors, Inc.
5. Barney, Blaise. "Introduction to Parallel Computing". Lawrence Livermore National Laboratory. http://www.llnl.gov/computing/tutorials/parallel_comp/. Retrieved 2007-11-09
6. Bill Dally, Stanford University: Advanced Computer Organization: Interconnection Networks
7. Quinn Michael J, *Parallel Programming in C with MPI and OpenMP* McGraw-Hill Inc. 2004. ISBN 0-07-058201-7.
8. Albert Y.H. Zomaya, Parallel and distributed Computing Handbook, McGraw-Hill Series on Computing Engineering, New York (1996).
9. Ernst L. Leiss, Parallel and Vector Computing A practical Introduction, McGraw-Hill Series on Computer Engineering, New York (1995).
10. Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis, Introduction to Parallel Computing, Design and Analysis of Algorithms, Redwood City, CA, Benjmann / Cummings (1994).
11. X. Sun and J. Gustafson, "Toward a Better Parallel Performance Metric," Parallel Computing, Vol. 17, No. 12, Dec. 1991, pp. 1093-1109.
12. Bossel H., *Modeling & Simulation*, A. K. Peters Pub., 1994.
13. Ghosh S., and T. Lee, *Modeling & Asynchronous Distributed Simulation: Analyzing Complex Systems*, IEEE Publications, 2000.
14. Fishman G., *Discrete-Event Simulation: Modeling, Programming and Analysis*, Springer-Verlag, Berlin, 2001.
15. Woods R., and K. Lawrence, *Modeling and Simulation of Dynamic Systems*, Prentice Hall, 1997.
16. Rashmi Bajaj and Dharma P. Agrawal. Improving scheduling of tasks in a heterogeneous environment. IEEE Trans. Parallel Distrib. Syst., 15(2):107–118, 2004.