A LOAD BALANCING STRATEGY FOR PARALLEL COMPUTING SYSTEM USING SMP SUN FIRE X4470

Dissertation submitted to Jawaharlal Nehru University
in partial fulfillment of the requirements
for the award of the degree of

MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE AND TECHNOLOGY

TAJ ALAM
ENROLLMENT NO. 10/10/MT/32

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110067
INDIA

2012

SCHOOL OF COMPUTER & SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI, 110067 (INDIA)

CERTIFICATE

This is to certify that the dissertation entitled **"A Load Balancing Strategy for Parallel Computing System using SMP Sun Fire X4470"** is being submitted by **Mr. Taj Alam** to **School of Computer and Systems Sciences, Jawaharlal Nehru University New Delhi-110067, India** in the partial fulfillment of the requirements for the award of the degree of **Master of Technology in Computer Science and Technology**. This work has been carried out by him in the School of Computer and Systems Sciences under the supervision of Dr. Zahid Raza. The matter personified in the dissertation has not been submitted for the award of any other degree or diploma.

DR. ZAHID RAZA	PROF. KARMESHU
(SUPERVISOR)	(DEAN)

DECLARATION

I hereby declare that the dissertation work entitled **“A Load Balancing Strategy for Parallel Computing System using SMP Sun Fire X4470”** in partial fulfillment for the requirements for the degree of **“Master of Technology in Computer Science and Technology”** and submitted to School of Computer & Systems Sciences, Jawaharlal Nehru University, New Delhi-110067, India, is the authentic record of my own work carried out during the time of Master of Technology under the supervision of Dr. Zahid Raza. This dissertation comprises only my original work. This dissertation is less than 100,000 words in length, exclusive tables, figures and bibliographies.

The matter personified in the dissertation has not been submitted for the award of any other degree or diploma.

Taj Alam
Enrollment No. 10/10/MT/32
M. Tech (2010-12)
SC&SS, JNU
New Delhi India -110067

ACKNOWLEDGEMENT

I am very glad to express my sincere gratitude and thanks to my supervisor **Dr. Zahid Raza** for his guidance. I would like to express special thanks to Dr. Zahid Raza for many helpful discussions and his intellectual input to make dissertation work worthy. His extensive and invaluable research experiences were very helpful in my dissertation and the most important thing was the helping nature of him that contributes an important share in fulfillment of this work. The mythology, philosophy and problem solving methods learned by him have been very beneficial in this work and would be afterward. I would like to express my thanks to Dr. D. P. Vidyarthi for his valuable help, suggestions and encouragement.

I would like to express my thanks to Dean SC&SS JNU, Prof. Karmeshu in support to pursue my work in the School. Also my thanks go to School administration and librarian of software library and main library for supporting me, in whatever way they can, to make dissertation a success. Their support has been a real emphasize in completing this dissertation.

I would like to accord my sincere thanks to Mr. Mohammad Sajid, Mr. Mohammad Shahid, Mr. D. P. Sahu and Mr. Kapil Gupta for their valuable suggestions for my dissertation work.

My parents and family members have been my strength through the long hours of study and research. I especially thank my father and mother for their patience, unconditional love and economical as well as moral support for completing this dissertation. Finally I would like to express thanks to each person & thing which is directly or indirectly related to my dissertation work.

TAJ ALAM

Dedicated to Almighty (Allah)
Who Created Everyone

Table of Contents

ABSTRACT

Parallel computing is an evolution of serial computing that attempts to imitate what has always been the state of affairs in the natural world: many complex, interrelated events happening at the same time, yet within a sequence, whether it is galaxy formation, planetary movement, weather and ocean patterns etc. Historically, parallel computing has been considered to be "the high end of computing", and has been used to model difficult problems in many areas of science and engineering. Today, commercial applications provide an equal or greater driving force in the development of faster computers. These applications require the processing of large amounts of data in sophisticated ways. Some of the examples could be databases, data mining, oil exploration, web search engines, web based business services etc. Main reasons for using parallel computing is that it saves time and money with the philosophy that if an application has modules that can run in parallel, deploying more computational resources will shorten it's time of completion, with potential cost savings. The development of parallel computers has seen an enormous growth, with the advancements in the area of chip fabrication. Thus parallel computers can now be built from cheap, commodity components. The use of parallel computers is primarily to solve large scale problems which are either impractical to solve on a single computer, especially given limited computer memory or can be solved more efficiently on a parallel machine owing to the inherent parallelism in the job.

Parallel systems main goal is to minimize turnaround time by parallel execution of the job(s) by distributing the entire workload on the available computational resources, thus allowing various modules of the job to run simultaneously. To meet this objective, parallel computing has to deal with a lot of issues which crop up while working with parallel code. These issues can result in bottleneck and restrict the behaviour of parallel program in attaining an aforesaid speedup suggested by Amdahl Gene. The most problematic issue that crops up is the distribution of workload in both the categories of parallel system viz. homogenous and heterogeneous system. In homogenous system the processor with maximum load overpowers the working of system resulting in poor job turnaround time whereas in heterogeneous system the slowest processor dominates the job turnaround time. Therefore, in parallel systems, distribution of workload could result into some nodes to be heavily loaded and some nodes to be under loaded. This situation demands an effective load balancing strategy to be in place which ensures a uniform distribution of load across the board. Load balancing mechanism could be treated as a software approach to redistribute system wide workload among the nodes of the system in order to reduce the mean job execution and hence the turnaround time. An efficient load balancing strategy must exhibit the features like creating little traffic overhead, low overhead for running the load balancing algorithm, must be fair enough so that heavily loaded node is balanced first with lightly loaded node, should utilize minimum CPU time to name a few.

This dissertation presents a model for the load balancing strategy for a multiprocessor system that aims to minimizing the turnaround time for a job(s) submitted for execution. The model is developed using Sun Fire X 4470 server as a test bed using OpenMP as a programming tool. Sun Fire X 4470 server is a multiprocessor system with four nodes each with eight cores. Since, each core can be treated as a node; it makes available thirty two nodes that can be programmed. OpenMP is used as a programming tool as it is suitable for the shared memory programming applications.

The proposed scheduler allocates the modules of the job(s) over the nodes in such a way that the desired objective of minimizing the turnaround time is met. The proposed model is based on centralized dynamic load balancing strategy using thresholds. The threshold values set helps in categorizing the nodes as heavily or lightly loaded nodes. The threshold values used here are adaptive in nature i.e. as the load on the system increases, threshold values are readjusted to suite the growing load on the system. The model works in such ways that the thresholds tend to converge the nodes load towards the mean of the workload. These values becomes approximately equal when the load becomes evenly distributed depicting the balanced state of the system. The model is centralized in nature and hence it results in little traffic overhead. Moreover, the load redistribution process is fair as load is first readjusted between heavily loaded node and lightly loaded node through the use of max priority queue and min priority queue. The balancing process utilizes minimum CPU time as redistribution is only carried out when lightly loaded and heavily loaded nodes are reported.

Simulation study has been carried out for the model to evaluate its performance under various test conditions. It has been found that the model works well in ensuring an even distribution of the workload.

List of Acronyms

CPU	Central Processing Unit
MPP	Massive Parallel Processing
RISC	Reduced Instruction Set Computing
IF	Instruction Fetch
ID	Instruction Decode
EX	Instruction Execute
MEM	Memory Access
WB	Write Back
NUMA	Non- Uniform Memory Access
UMA	Uniform Memory Access
SMP	Symmetric Multiprocessor
CMP	Chip Multiprocessor
RU	Rack Unit
IMC	Integrated Memory Controller
MB	Memory Buffer
SMI	Scalable Memory Interface
QPI	Quick Path Interconnect
HT	Hyper Threading
DIMM	Dual In-line Memory Module
DDR3	Double Data Rate Type Three
DIM	Dual Integrated Memory
GT/Sec	Giga Transaction per Second
GB/Sec	Gigabytes Transfer per second
SATA	Serial AT Attachment
USB	Universal Serial Bus
PCI	Peripheral Component Interconnect
OpenMP	Open Multiprocessing
MPI	Message Passing Interface

SPMD	Single Program Multiple Data
T_{under}	Lower threshold
FIFO	First In First Out
QoS	Quality of Service
I/O	Input / Output
TPS	Transaction Processed per Second
NP	Nondeterministic Polynomial
P	Polynomial
N_{task}	Number of Tasks
N_{proc}	Number of Processors
T_{lower}	Lower threshold
T_{upper}	Upper threshold
LHM	Lower half mean of l_i for the nodes sorted in ascending order
UHM	Upper half mean of l_i for the nodes sorted in ascending order
M	Mean of l_i for the nodes sorted in ascending order
L	Min priority queue containing nodes having load l_i below T_{lower}
H	Max priority queue containing nodes having load l_i above T_{upper}
X	Queue for nodes having load between T_{lower} & T_{upper}
LQ_i	Local queue of jobs for each processing element N_i
LQL_i	Length of the local queue for each processing element N_i
TAT	Turnaround Time
S	Speedup
ξ	Efficiency
DLB	Dynamic Load Balancing

List of Figures

List of Tables

Chapter 1

Introduction

Traditionally, software has been written for serial computation to be run on a single computer having a single Central Processing Unit (CPU) where only one instruction may execute at any moment in time. Parallel computing is an evolution of serial computing that attempts to imitate what has always been the state of affairs in the natural world. Initially, parallel computing was considered to be "the high end of computing", and was used to model difficult problems in many areas of science and engineering. Today commercial applications provide an equal or greater motivating force in the development of faster computers. Main reasons for using parallel computing are that it saves time and money. Throwing more resources at a task shortens it's time to completion, with potential cost savings. Parallel computers can be built from cheap, commodity components. It solves very large problems that are impractical to solve on a single computer, especially given limited computer memory. During the past twenty years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that parallelism is the future of computing. The chapter starts with discussion on parallel and distributed computing, the types of parallelism and various issues and challenges that crop in parallel computing. This is followed by parallel computer memory architecture, discussion on symmetric multiprocessor and various programming tools for designing parallel programs.

1.1 Parallel and Distributed Computing

Parallel computing [1, 2, 3] is a form of computation in which many calculations are carried out concurrently operating on the rule that large problems can often be divided into smaller ones, which are then solved in parallel. There are several different forms of parallelism: bit-level, instruction-level, task-level, data-level parallelism. Parallel computers can be roughly classified into multi-core and multi-processor computers having multiple processing elements within a single machine while clusters, MPPs, and grids use multiple computers connected via network to work on the same task. Parallel programs [4] are more difficult to write than sequential ones. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance. The maximum possible speed-up of a program is observed as Amdahl's law [9].

1.2 Types of Parallelism

Various types of parallelism have been defined at various levels such as bit level, data level, task level and instruction level. Some of these are discussed as follows.

1.2.1 Bit-level Parallelism

Increasing the word size reduces the number of instructions the processor must execute to perform an operation on variables whose size are greater than the length of the word. For example, where a 8-bit processor is required to add two 16-bit integers, the processor must first add the 8 lower-order bits from each integer using the standard addition instruction then add the 8 higher-order bits using an add-with-carry instruction and the carry bit from the lower order addition. Thus, an 8-bit processor requires two instructions to complete a single operation whereas a 16-bit processor requires just one instruction to complete the operation [1, 2, 3].

1.2.2 Instruction-level Parallelism

The possible overlap among instructions is called instruction level parallelism. A five-stage pipeline in a RISC machine has the following instruction parts, IF (Instruction Fetch), ID (Instruction Decode), EX (Execute), MEM (Memory Access), WB (Write Back). The instructions can be re-ordered and combined into groups which are then executed in parallel without changing the result of the program. This is known as instruction-level parallelism. Modern processors have multi-stage instruction pipelines. Each stage in the pipeline corresponds to a different action. The processor performs on that instruction in that stage. A five-stage pipelined superscalar processor, capable of issuing two instructions per cycle have two instructions in each stage of the pipeline, for a total of up to 10 instructions being simultaneously executed [1, 2, 3].

1.2.3 Data-level Parallelism

Data parallelism is a form of parallelization of computing across multiple processors in parallel environments. Data parallelism is the parallelism intrinsic in program loops which focuses on distributing the data across different computing nodes to be processed in parallel. Parallelizing loops often leads to similar operation sequences being performed on elements of a large data structure. Data parallelism focuses on distributing the data across different parallel computing nodes. In a multiprocessor system executing a single set of instructions, data parallelism is achieved when each processor performs the same task on different pieces of distributed data. In some situations, a single execution thread controls operations on all pieces of data. In others, different threads control the operation, but they execute the same code [1, 2, 3].

1.2.4 Task-level Parallelism

Task-level parallelism is a form of parallelization of computer code across multiple processors in parallel computing environments. Task parallelism is the characteristic of a parallel program that entirely different calculations can be performed on either the same or different sets of data. This contrasts with data parallelism where the same calculation is performed on the same or different sets of data. Task parallelism focuses on distributing execution processes (threads) across different parallel computing nodes. In a multiprocessor system, task parallelism is achieved when each processor executes a different thread (or process) on the same or different data. The threads may execute the same or different code. In the general case, different execution threads communicate with one another as they work. Communication takes place usually to pass data from one thread to the next as part of a workflow [1, 2, 3].

1.3 Issues in Parallel Computing

Parallel computing has to deal with lot of issues which crop up while working with parallel code. These issues result in bottlenecks and restrict the behavior of parallel program in attaining an aforesaid speedup given by Amdahl Gene [9]. Some of these issues are discussed below.

- **Load Balancing:** One of the main problems that need to be tackled by any system that attempts to provide efficient execution of parallel programs in distributed environments is load balancing. In order to be efficient, the system must distribute the workload among the different computing nodes in a way that guarantees optimal utilization of the available resources and in particular of the CPU [12].
- **Portability:** A portable high-performance program must be capable of adapting to the particular environment in which it is running. We call the technique for achieving this adaptation Two-Phase Adaptation. Firstly, an automatic study and examination of the underlying architecture environment is carried out. Secondly, an efficient matching between the application complexity and the environment complexity is completed [1, 2, 3].
- **Problem Size:** Applications are often classified according to how often their subtasks need to synchronize or communicate with each other. An application exhibits fine-grained parallelism if its subtasks must communicate many times per second. It exhibits coarse-grained parallelism if they do not communicate many times per second, and it is embarrassingly parallel if they rarely or never have to communicate [1, 3].
- **Communication:** Communication depends upon the size of the problem and how we are dividing our problem to get solved. More the granularity more will be the communication between them. However, greater parallelism is achieved but we have to compromise with the communication cost [1, 2, 3].
- **Scalability:** Scalability is the capability of a system, network, or process, to handle growing amounts of work in an elegant manner or its ability to be enlarged to accommodate that growth. For example, it can refer to the capability of a system to increase total throughput under an increased load when resources are added [1, 2, 3].
- **Resource Allocation:** Performing computing and communication tasks on parallel and distributed systems involves the coordinated use of different types of machines, networks, interfaces, and other resources. Resource allocation is used to assign the available resources in an economic way. Resource allocation is the scheduling of activities and the resources required by those activities while taking

into consideration both the resource availability and the project time. Resource allocation may be decided by using computer programs applied to a specific domain to automatically and dynamically distribute resources to applicants. It may be considered as a specialized case of automatic scheduling. [1, 2, 3]

- **Scheduling:** The problem of job scheduling is to determine how sharing of resources should be done in order to maximize the system's utility. Scheduling is the method by which threads, processes or data flows are given access to system resources (e.g. processor time, communications bandwidth). This is usually done to load balance a system effectively or achieve a target quality of service. The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (execute more than one process at a time) and multiplexing (transmit multiple flows simultaneously). [17, 18].

1.4 Challenges in Parallel Computing

There are many challenges which are hindrance towards the parallel computing which makes it a difficult task to parallelize a problem. These challenges are discussed below.

- **Concurrency:** Concurrency is a system property to execute multiple things simultaneously, operating on a principle that many instruction can be interleaved resulting in a minimized job turnaround time. Concurrent use of shared resources can be a source of indeterminacy leading to issues such as deadlock, and starvation. The design of concurrent systems often exhibit finding reliable techniques for coordinating their execution, data exchange, memory allocation, and execution scheduling to minimize turnaround time and maximize throughput [1].
- **Data Locality Problem:** In a distributed memory machine, if iterations are executed on the processors that initially have much of the data they need, then communication overhead and latency will be reduced, resulting in better execution time. Furthermore, if multiple iterations access the same data, communication requirement can be reduced by executing them on same processor. If however the data is migrated to another remote location then

reduction in execution time should over power the communication time. The study of this important issue is called data locality problem [1, 3, 16].

- **Scalability Support in Hardware:** Methods of adding more resources for a particular application fall into two broad categories, vertical scaling and horizontal scaling. In the past, the price difference between the two models has favored "scale out" computing for those applications that fit its archetype, but recent advances in virtualization technology have imprecise that advantage, since deploying a new virtual system over a hypervisor (where possible) is almost always less expensive than actually buying and installing a real one. Larger numbers of computers means increased management complexity, as well as a more complex programming model and issues such as throughput and latency between nodes; also, some applications do not lend themselves to a distributed computing model. Scalability support in hardware is limited by bandwidth and latencies to memory plus interconnects between processing elements [1, 3].
- **Synchronization Constructs:** Synchronization refers to one of two distinct but related concepts: synchronization of processes, and synchronization of data. Process synchronization refers to the idea that multiple processes are to coordinate at a certain point, so as to reach an agreement or commit to a certain sequence of action. Data synchronization refers to the idea of keeping multiple copies of a dataset in consistency with one another, or to maintain data integrity. Process synchronization primitives are commonly used to implement data synchronization. Synchronization constructs and protocols must be used very carefully such that programs are free from deadlock and race conditions [1, 3].
- **Software Engineering Practices:** Software engineering is the study of designing, implementing, and modifying software in order to ensure it is of high quality, affordable, maintainable, and fast to build. It is a systematic approach to software design, involving the application of engineering practices to software. Software engineering deals with the organizing and analyzing software to get the best out of them. It doesn't just deal with the creation or manufacture of new software, but its internal maintenance and arrangement. Appropriate software engineer

practices have to be adopted such as incremental parallelism or code reuse while designing parallel program code [1, 3].

- **Support for Portable Performance:** Portability in high-level computer programming is the usability of the same software in different environments. The pre-requirement for portability is the generalized abstraction between the application logic and system interfaces. When software with the same functionality is produced for several computing platforms, portability is the key issue for development cost reduction. The programmer has to adapt right models so that he can write code once and expect it to execute well on the important parallel platforms without much modification [1, 3].

1.5 Parallel Computer Memory Architecture

Main memory in a parallel computer is either shared memory or distributed memory [1, 5]. Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well. Distributed shared memory combines the two approaches. Accesses to local memory are typically faster than accesses to non-local.

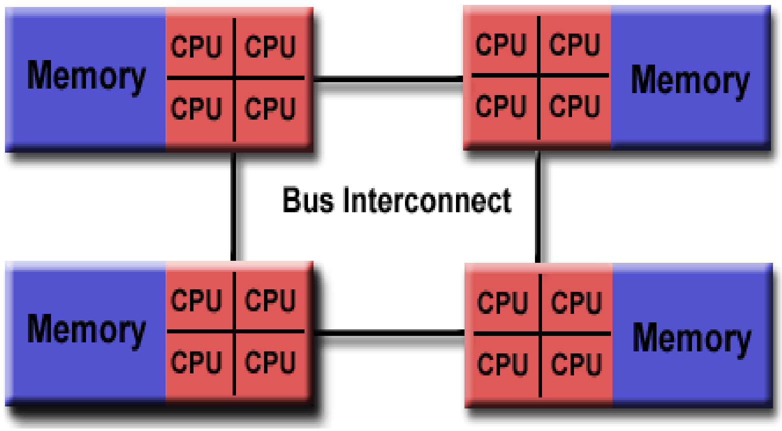


Figure 1.1– Non Uniform Memory Access (NUMA) Model

A logical view of Non-Uniform Memory Access (NUMA) architecture is shown above in Figure 1.1. Here the processors in one directory can access that directory's memory with less latency than they can access memory in the other directory's memory. The computer architectures in which each element of main memory can be accessed with equal latency are known as Uniform Memory Access (UMA) systems and are depicted in Figure 1.2.

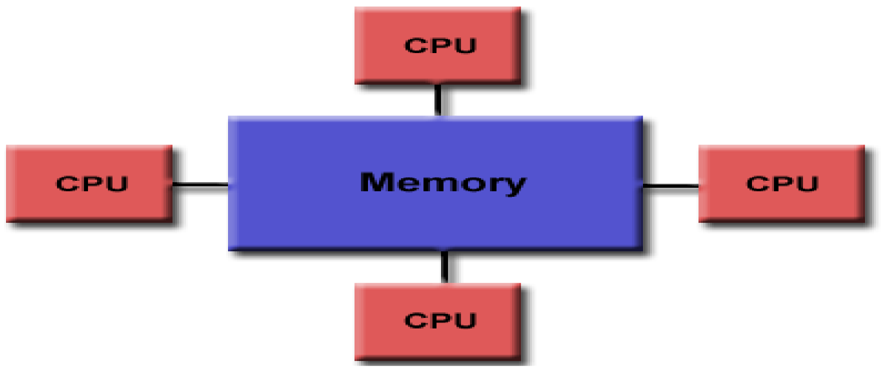


Figure 1.2 – Uniform Memory Access (UMA) Model

Typically, uniform access can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as Non-Uniform Memory Access (NUMA) architecture. Distributed memory systems have non-uniform memory access. Computer systems make use of caches—small, fast memories located close to the processor which store temporary copies of memory values. Parallel computer systems have difficulties with caches that may store the same value in more than one location, with the possibility of incorrect program execution. These computers require a cache coherency system, which keeps track of cached values and tactically access them, thus ensuring correct program execution. Bus snooping [1] is one of the most common methods for keeping track of which values are being accessed. Processor–processor and processor–memory communication can be implemented in hardware in several ways via shared memory, a crossbar switch, a shared bus or an interconnect network of various topologies including star, ring, tree, hypercube, or mesh.

1.6 Symmetric Multiprocessor

Symmetric Multiprocessor is among the class of system that come under parallel and distributed system. Parallel computers can be roughly classified into multi-core and multi-processor computers having multiple processing elements within a single machine while clusters, MPPs, and grids use multiple computers to work on the same task connected via inter connect network. Symmetric Multiprocessor also called SMP is a shared memory system with all the processors having access to the same memory. SMP follow the UMA class of memory architecture. Many SMP can even be grouped together to execute a parallel job(s) where the resulting system being known as cluster SMP. Cluster SMP comes under the class of NUMA architecture.

1.6.1 Symmetric Multiprocessing

In computing, symmetric multiprocessing (SMP) involves a multiprocessor computer hardware architecture where two or more identical processors are connected to a single shared main memory and are controlled by a single OS instance [19]. Processors may be interconnected using buses, crossbar switches or mesh networks. The bottleneck in the scalability of SMP using buses or crossbar switches is the bandwidth and power consumption of interconnection among the various processors, the memory, and the disk arrays. Mesh architectures avoid these bottlenecks, and provide nearly linear scalability to much higher processor counts. A computer system that uses symmetric multiprocessing is called a symmetric multiprocessor [1, 3, 5].

1.6.2 Symmetric Multiprocessing v/s Other Parallel Technologies

The parallel computing is a wide field which encompasses many technologies. Each technology explores the parallelism accordingly. The explanation of each of the technologies is handled below.

- **Multi Core Computing:** A multi-core processor is a single computing component with two or more independent actual processors (called "cores"), which are the units that read and execute program instructions. The instructions are ordinary CPU instructions such as add, move data, and branch. The presence of multiple cores facilitates the user to run multiple instructions at the same time, increasing overall speed for programs agreeable to parallel computing. Manufacturers typically integrate the cores onto a single integrated circuit die (known as a chip multiprocessor or CMP), or onto multiple dies in a single chip package. A multi core processor can issue multiple instructions per cycle from multiple instruction streams [1, 5].
- **Distributed Computing:** It is a field of computer science that studies distributed systems. A distributed system consists of multiple autonomous computers that

communicate each other through a computer network to carry out the processing. A computer program that runs in a distributed system is called a distributed program, and distributed programming is the process of writing such programs. In distributed computing, a problem is divided into many tasks, each of which is solved by one or more computers. Distributed computers are highly scalable [5].

- **Cluster Computing:** A cluster is a group of loosely coupled computers that work together closely, so that in some respects they can be regarded as a single computer. Clusters are composed of multiple standalone machines connected by a network. While machines in a cluster do not have to be symmetric, load balancing is more difficult if they are collection of heterogeneous systems [1, 3, 5].
- **Massive Parallel Processing:** A massively parallel processor (MPP) is a single computer with many networked processors. MPPs have many of the same characteristics as clusters, but MPPs have specialized interconnect networks whereas clusters use commodity hardware for networking. MPPs also tend to be larger than clusters, typically having far more than 100 processors. In MPP each CPU contains its own memory and copy of the operating system and application. Each subsystem communicates with the others via high-speed interconnect [1, 2].
- **Grid Computing:** Grid computing is the most distributed form of parallel computing. It makes use of computers communicating over the internet to work on a given problem. Because of the low bandwidth and extremely high latency available on the internet, grid computing typically deals only with embarrassingly parallel problems. Most grid computing applications use middleware, software that sits between the operating system and the application to manage network resources and standardize the software interface [1, 2, 5].

1.6.3 SMP Sun Fire X4470 Server

The Sun Fire X4470 server is a symmetric multiprocessor (SMP). It can provide the critical virtualization platform for consolidating web application servers and collaboration tools, virtualizing enterprise performance management applications, as well as batch processing. It is a compact and expandable enterprise class 4-socket x 86 servers, delivering an efficient performance, expandability, density and power efficiency in a 3 rack unit (RU) form factor. The server has following main characteristics [6].

- **Intel Xeon Processor:** This processor has some inherent features which makes it among the fastest available processors in the world. It has eight cores per die, 24 MB Level-3 shared inclusive cache, two Integrated Memory Controllers (IMCs) with two Intel Scalable Memory Interfaces (SMIs) each, four full-width, bidirectional Intel QPI buses within the Sun Fire X4470 server. Intel QPI technology provides high-speed, point-to-point interconnects between processors with Intel Hyper Threading (HT) technology between processors and I/O and Intel Turbo Boost Technology enabled within the system. Figure 1.3 presents an insight into the Intel Xeon Processor 7500 [6].

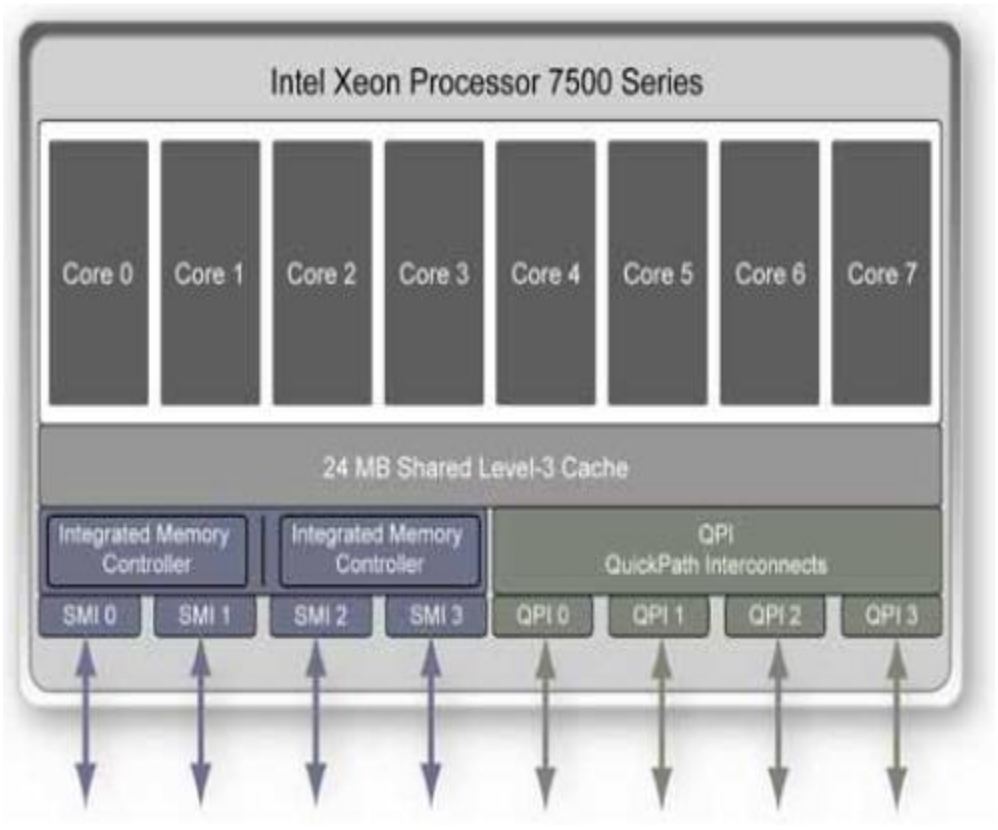

Intel Xeon Processor 7500 Series
Core 0
Core 1
Core 2
Core 3
Core 4
Core 5
Core 6
Core 7
24 MB Shared Level-3 Cache
Integrated Memory Controller
Integrated Memory Controller
QPI
QuickPath Interconnects
SMI 0
SMI 1
SMI 2
SMI 3
QPI 0
QPI 1
QPI 2
QPI 3

Figure 1.3- Intel Xeon Processor 7500

- **Intel 7500 Scalable Memory Architecture:** Each Intel Xeon Processor 7500 Series CPU provides two integrated memory controllers that each operates on a pair of interlocked memory channels. By default, memory is interleaved between the two memory controllers belonging to each processor. A pair of SMI links connects each integrated memory controller to the memory subsystem. To increase reliability and bandwidth, the SMI links that originate from the same memory controller operate in lock-step fashion to access memory DIMMs. Intel 7500 Scalable Memory Buffers (MBs) control SMI link access to the memory DIMMs. Each MB connects to one SMI link and up to four DIMMs using two DDR3 channels. Figure 1.4 presents the Scalable Memory Architecture for Intel 7500 [6].

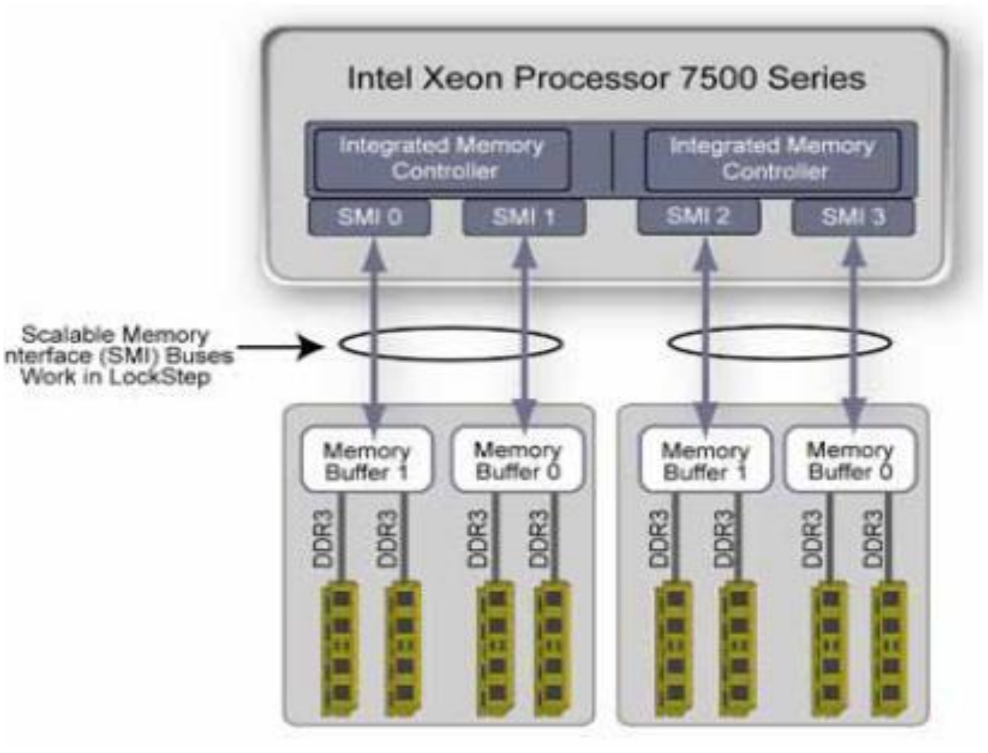

Intel Xeon Processor 7500 Series
Integrated Memory Controller
Integrated Memory Controller
SMI 0
SMI 1
SMI 2
SMI 3
Scalable Memory nterface (SMI) Buses Work in LockStep
Memory Buffer 1
Memory Buffer 0
Memory Buffer 1
Memory Buffer 0
DDR3
DDR3
DDR3
DDR3
DDR3
DDR3
DDR3
DDR3

Figure 1.4- Intel 7500 Scalable Memory Architecture

- **Motherboard configuration:** The design of the Sun Fire X4470 server supports the following system architecture features. There are Four-processor Intel Xeon Processor 7500 Series CPUs, Dual Integrated Memory (DIM) controller on each processor with Intel Quick Path Interconnect architecture, providing 6.4 GT/sec links, delivering up to 25 GB/sec of total bandwidth. The Intel 82801JB I/O

Controller Hub, supporting PCI, SATA, and USB connectivity. It has ten high-speed PCI Express 2.0 slots for high-performance I/O expansion. The architecture of the motherboard of Sun Fire X4470 server is shown in Figure 1.5 [6].

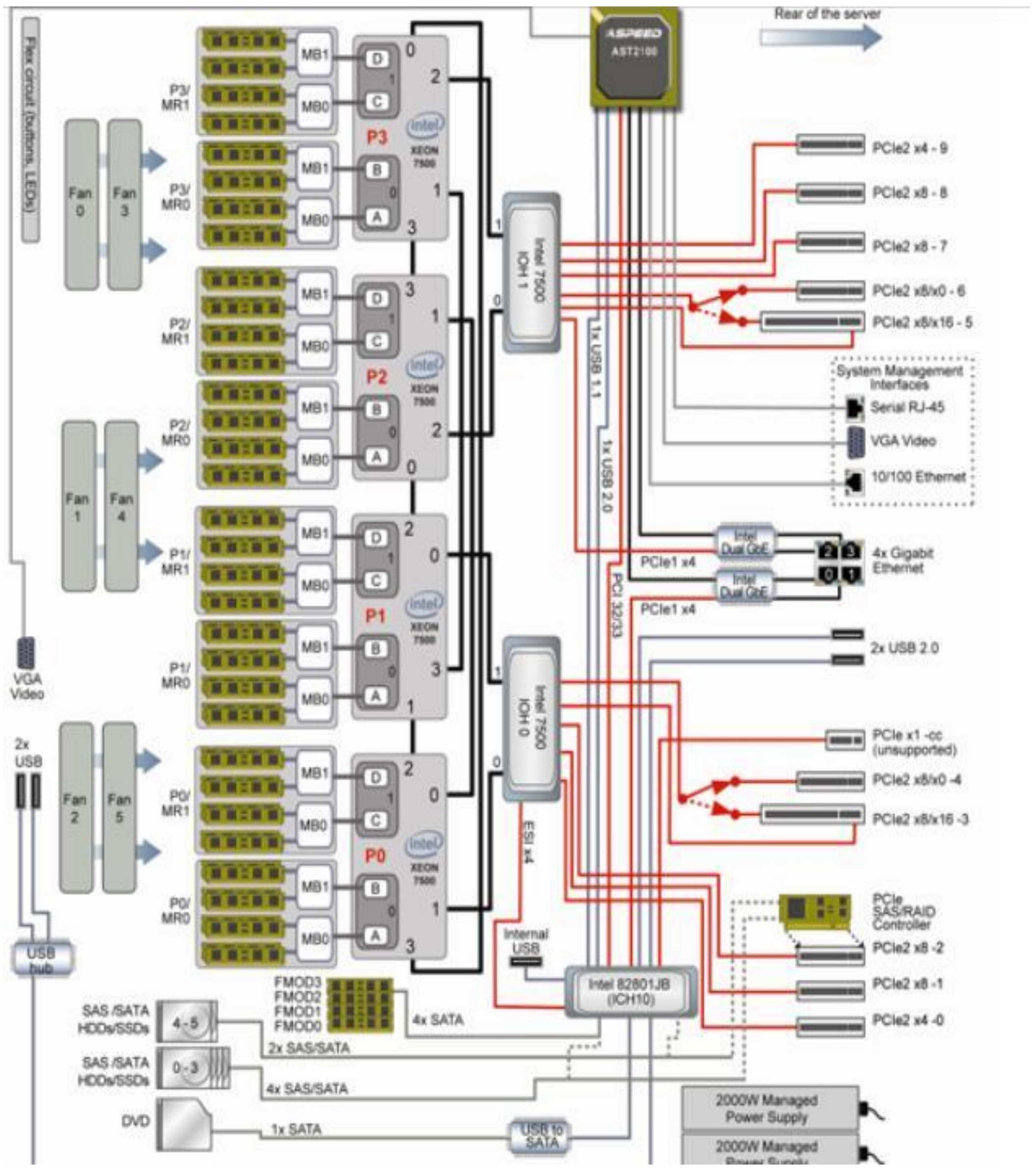


Figure 1.5 - Sun Fire X4470 Server Motherboard

1.7 Programming Tools for Parallel Computing

There are three approaches to parallel programming which are popular in research community. These approaches are defined for working with multithreading on shared memory systems and message passing for distributed memory system. Multithreading explores the task parallelism in a program whereas message passing explores the data parallelism in a program. The third is a hybrid approach which combines the both approaches to achieve the benefits of both programming methodologies. The programming tools for working on these methodologies are defined as OpenMP designed by OpenMP Architecture Review Board and the Message Passing Interface handled by MPI Forum along with a hybrid approach for mixed mode programming involving both OpenMP and MPI [7-8].

1.7.1 Message Passing Interface

The message passing programming model is a distributed memory model with explicit control parallelism. This uses an SPMD (Single Program Multiple Data) model. Processes are only able to read and write to their respective local memory. Data is copied across local memories by using the appropriate subroutine calls. The MPI standard defines a set of functions and procedures that implements the message passing model. MPI codes run on both distributed and shared memory architectures. It is adjustable to coarse grain parallelism. A large number of vendor optimized MPI libraries exist. Each process has its own local memory. Data is copied between local memories via messages which are sent and received via explicit subroutine calls [8].

1.7.2 OpenMP

OpenMP is an industry standard for shared memory programming. Based on a combination of compiler directives, library routines and environment variables, it is used to specify parallelism on shared memory machines. Directives are added to the code to tell the compiler of the presence of a region to be executed in parallel. This uses a fork-join model. The code will only run on shared memory machines. It is fairly portable. It permits both course grain and fine grain parallelism. Uses directives help the compiler parallelize the code. Each thread sees the same global memory, but has its own private memory [7].

1.7.3 Mixed Mode Programming

A mixed mode programming model should be able to take advantage of the benefits of both models. It allows us to make use of the explicit control data placement policies of MPI with the finer grain parallelism of OpenMP. The majority of mixed mode applications involve a hierarchical model with MPI parallelization occurring at the top level and OpenMP parallelization occurring below as shown in Figure 1.6.

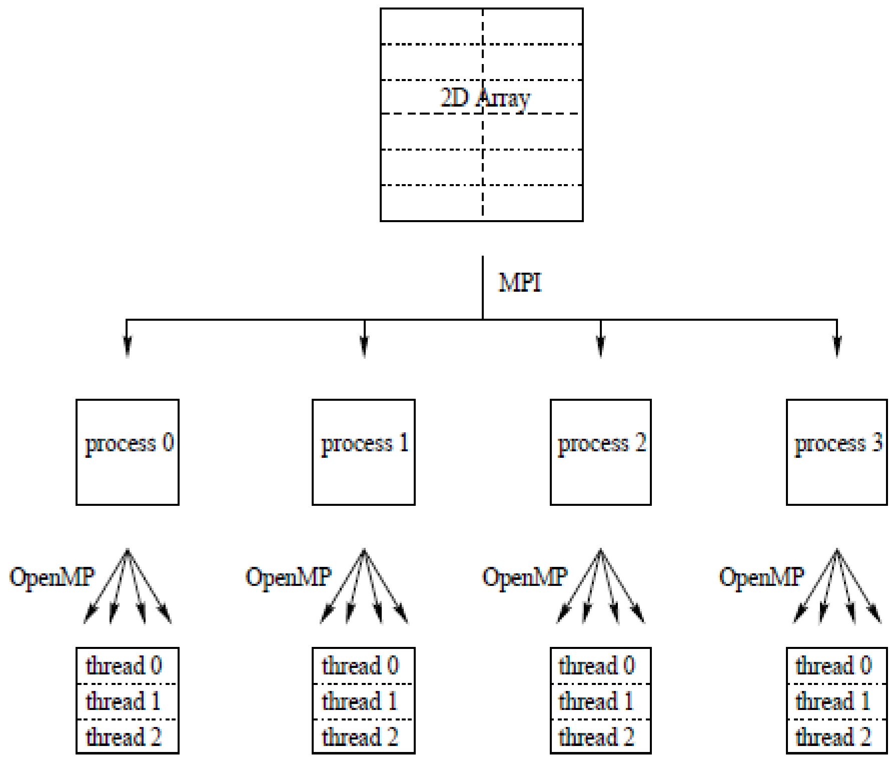


Figure 1.6- Hybrid Programming Model

To ensure that the code is portable, all MPI calls should be made within thread sequential regions of the code. This often creates little problem as the majority of codes involve the OpenMP parallelization occurring beneath the MPI parallelization and hence the majority of MPI calls occur outside the OpenMP parallel regions. MPI calls can occur within an OpenMP parallel region also but they should occur in restricted constructs only [7, 8].

Chapter 2

Load Balancing

Parallel and distributed systems are considered to be the future for scientific and engineering computing. The main goal of the parallel systems is to minimize job turnaround time by parallel execution of jobs. Parallel computing has to deal with lot of issues which crop up while working with parallel code. These issues result in many bottlenecks that restrict the behavior of the parallel program in attaining an aforesaid speedup given by Amdahl Gene. The most problematic issue that crops up is the distribution of workload in both the categories of parallel system viz. homogenous and heterogeneous systems. In homogenous system the processor with maximum load overpowers the working of system resulting in poor job turnaround time whereas in heterogeneous system the slowest processor dominates the job turnaround time. In parallel systems, distribution of workload could result into some nodes to be heavily loaded and some nodes to be under loaded. This situation demands an effective load balancing strategy to be in place which ensures a uniform distribution of load across the board. The chapter begins with discussion on load balancing, various issues and challenges that crop during balancing of workload along with classification of various load balancing strategies. This is followed by discussion on various load balancing algorithms with insight into some related work reported in the literature. Later, some important QoS parameters are discussed. The chapter concludes with NP Completeness of load balancing algorithms and some possible solutions to this problem.

2.1 Load Balancing

The problem of job scheduling is to determine how sharing of resources should be done in order to maximize the system's utility. Scheduling is the method by which threads, processes or data flows are given access to system resources for e.g. processor time, communications bandwidth [21]. This is usually done to load balance a system effectively or achieve a target quality of service. The need for a scheduling algorithm arises from the requirement for most modern systems to perform multitasking (execute more than one process at a time) and multiplexing (transmit multiple flows simultaneously). Scheduling of jobs should be done in such a way that each computing node has its proper share of work so that job turnaround time is minimized. Load Balancing can be treated as a subset of scheduling where such process is adopted. Load balancing is a methodology to distribute workload across multiple computers, network links, central processing units, disk drives, or other resources, to achieve optimal resource utilization, maximize throughput, minimize response time, and avoid overload [1, 10]. It is an allocation of system recourses to individual jobs for certain time periods to optimize an objective function(s). To effectively utilize the resources, the job should be scheduled in such a way that no resources are underutilized and that the turnaround time is minimized. Load balancing optimizes the way jobs are scheduled on the system so that these objective function(s) are met. In order to achieve above goal load balancing strategy must exhibit the following features:

(i) Must create little traffic overhead
(ii) Low overhead for running the load balancing algorithm
(iii) Must be fair so that heavily loaded node is balanced first with lightly loaded node
(iv) Load balancing should utilize minimum CPU time

2.2 Issues and Challenges in Load Balancing

Various issues and challenges turn up while load balancing. These have to be tackled so that effective load balancing is done on the system for realizing the objective function(s) [1, 3, 10-12].

- **Synchronization:** The load balancing leads to synchronization of jobs so that one does not lag behind the other during resource utilization. Synchronization refers to one of the two distinct but interrelated concepts: synchronization of processes, and synchronization of data. Process synchronization refers to the idea that multiple processes coordinate at a certain point, so as to reach an agreement or commit to a certain sequence of action. Data synchronization refers to the idea of keeping multiple copies of a dataset in consistency with one another, or to maintain data integrity. Process synchronization primitives are commonly used to implement data synchronization. Synchronization constructs and protocols must be used very carefully such that programs free from deadlock and race conditions.

- **Communication Overhead:** Communication refers to the interaction between the processes to reach a valid conclusion. During program execution in parallel, a lot of data is communicated so that a valid result is achieved. Load redistribution is handled between the processes via communication of messages only. A lot of messages are communicated to processes to make the system consistent. During balancing, if this combination overhead is more than benefits of transferring work then it is useless to redistribute the work. Hence this overhead is a factor that comes up while balancing of load. More communication means less computation and hence speedup will be affected.
- **Locality Principle:** In a distributed memory machine, if iterations are executed on the processors that initially have much of the data they need, then communication overhead and latency will be reduced, resulting in better execution time. Furthermore, if multiple iterations access the same data, communication requirement can be reduced by executing them on the same processor. Locality is an issue which must be weighed against load balancing. It has to be checked that process migration does not over power the computation time on local processor due to excess in data transfer time to remote location.
- **Scalability:** Larger numbers of computers results in an increased management complexity, a more complex programming model and issues such as throughput and latency between nodes. Also, some applications even do not lend themselves to a distributed computing model. Scalability support in hardware is bandwidth and latencies to memory plus interconnects between processing elements. More the numbers of nodes in a system mean more will be the load on the load balancing algorithm to effectively utilize the resources. As the number of nodes increases, the communication overhead also increases to validate the system state. This in turn affects the speedup that has to be achieved. In normal scenario the speedup attained is less in comparison to the speedup proposed by Amdahl Gene.
- **Reliability:** Reliability means features that help avoid and detect system faults. A reliable system should not silently continue and deliver results that include

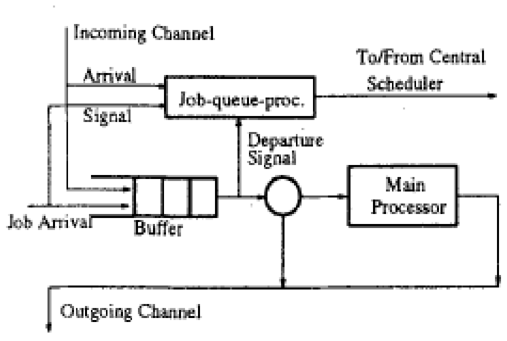
incorrect and corrupted data instead it should correct the corruption as and when possible. Further, the algorithmic approach should also be reliable. Centralized approach makes the system less reliable as the functioning of the system totally depends on the central node. If the central node fails, the whole system collapses. On the other hand, in the decentralized approach, the system is not dependent on one node with the control and decision power resting with multiple nodes. Thus, if any node fails, the system continues to work while producing correct results.

- **Excessive Page Migration:** Excessive page migration results in thrashing which is normally used to describe a computer whose virtual memory subsystem is in a constant state of paging. This is due to the rapidly exchanging data in memory for data on disk to the exclusion of most application-level processing. This causes the performance of the computer to degrade or collapse. The situation may not resolve itself quickly but can continue indefinitely until the underlying cause is addressed. Locality of data leads to excessive page migration. This further leads to thrashing as system involves in more paging than computation while giving a false impression that the processors are busy.

2.3 Classification of Load Balancing Approaches

There are many approaches to classify the load balancing strategies. Broadly, load balancing can be classified as centralized/ decentralized, static / dynamic, periodic / non periodic and with threshold / without threshold. Each of the above could be used either alone or in combination with others to provide effective load balancing [1, 11, 12].

- **Centralized v/s Distributed Load Balancing:** In centralized load balancing scheme the global load information is collected at a single node called central scheduler [12]. Local nodes send their load update messages to central scheduler. The central scheduler maintains three queues corresponding to lightly loaded nodes, medium loaded nodes and overloaded nodes. According to this information central scheduler balances the load from overloaded nodes to lightly loaded nodes. A typical model of local node and central node is presented in Figure 2.1 and Figure 2.2 respectively.



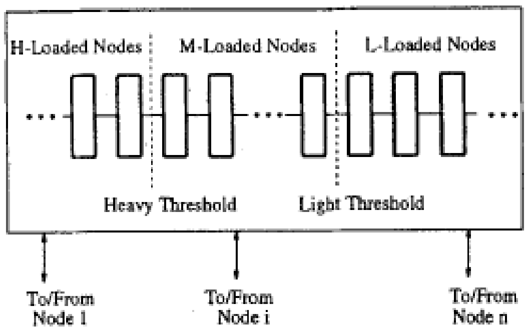


Figure 2.1- Model of Local Node **Figure 2.2- Model of Central Node**

In decentralized load balancing, each node broadcasts (periodically or instantaneously) its load information to other nodes to update their locally maintained load information table whenever its load state changes. According to this load information received the node with under loaded state requests for the jobs from overloaded nodes. The jobs are transferred to the requesting node if extra load is there on requesting node else the negative response is send. On receiving the negative response the requesting node searches for the other overloaded node in its information table. The process is continued till the node is successful in receiving jobs from other overloaded node. The model however incurs lot of cost during the whole process due to message overhead. A typical model of decentralized load balancing scheme is depicted in Figure 2.3 [11].

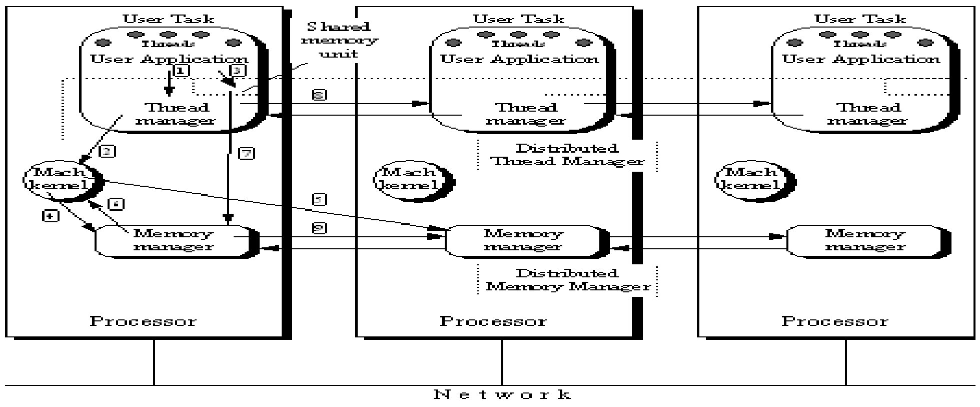


Figure 2.3 - Decentralized Load Balancing

- **Periodic v/s Non Periodic Load Balancing:** In periodic load balancing approach the load redistribution is carried out at a predefined interval of time. The CPU time is divided in computation time and communication time. During this communication time the procedure for load redistribution is carried out. The information for load status is maintained up during this phase only. As the uneven distribution of load is received the under loaded node requests for jobs from overloaded nodes (decentralized case) or central node asks the overloaded node to transfer extra jobs to its peer under loaded node (centralized case). Moreover unnecessary transfer of jobs is also restricted as the system is allowed to continue its execution until periodic time is reached. This scheme is usually adopted in combination with other schemes.

In non periodic load balancing approach the load redistribution is carried out instantaneously whenever the load state of nodes changes on arrival of new jobs. This process is carried out only when both under loaded node and over loaded node are reported. The load redistribution is initiated as this case appears else the nodes continue their processing until uneven state is reached. The process incurs less cost in comparison to messages communicated to update the state of nodes. As the uneven state is reached the system is brought to even state so that job turnaround time is minimized. Frequent load distribution however incurs lot of cost in comparison to periodic load balancing. The system in this approach is not allowed to stay in uneven state of load whereas in periodic the load redistribution is not initiated until the periodic time is reached [11, 12].

- **With Threshold v/s Without Threshold Load Balancing:** In threshold load balancing the workload is divided into three categories namely lightly loaded, medium loaded and overloaded. The parameter is assigned to check in which category of load state the node falls. Usually two parameters are set to account for the state of the node. T_{lower} checks the lower bound of load. The nodes reporting load below this value are treated as under loaded state. T_{upper} checks the upper bound of load on any node. The nodes whose load falls above this threshold value are treated as over loaded. The node whose load falls in between the threshold values are treated as normal loaded nodes. The parameters are predefined according to the state of system. Usually constant values are taken for both these parameters. However if the jobs arrival rate is quiet high then this constant allotment becomes useless. In such scenario adaptive threshold values are to be used to make system consistent towards the growing state of load.

In without threshold no threshold is maintained at individual nodes. Individual nodes maintained the load information table regarding the load on other nodes (decentralized case). Whenever the node becomes idle it searches its table for the requesting node for extra jobs. The request is send to the requesting node for the load redistribution. In centralized case central node maintains the information about the load state of individual nodes. It is the responsibility of the central node to ask the node with maximum load to transfer its jobs to idle node. This approach works on the process of work stealing as the load redistribution is only initiated when node becomes completely idle. This process incurs additional cost even in the case when jobs are not to be transferred but load redistribution is still carried out e.g. when one node is idle and other node just has two jobs to execute [10, 12].

- **Static v/s Dynamic Load Balancing:** In static load balancing the main systems parameters are predefined and jobs are dispatched according to the rules that are set a priori and are not affected by current state of the system. The jobs are allocated to individual nodes according to the predefined rules. If the job gets allocated initially on any node then it has to complete its execution on that node only. The process chooses the rules such that uneven state does not arises but as the system is dynamic there are situation when after careful allotment also the uneven sate is reached. This further decreases the assumed speedup. Various static load balancing algorithms are random, round robin, first come first serve etc.

  In dynamic load balancing approach the load is distributed in the system dynamically. Allocation of threads to processors is done during the run time. Thread migration is allowed from one processor to other during application execution. The jobs are allotted to nodes on their creation. During the execution the state of nodes are continuously checked according to the various parameters. If the system benefits in redistribution of jobs the load on the nodes is readjusted. In this approach no node is kept idle even after execution of jobs after initial allotment. The jobs are reassigned to it (periodically or instantaneously) as the case may be according to the algorithm definition. Various dynamic load balancing algorithms are central queue scheme, local queue scheme etc. [11, 12].

2.4 Load Balancing Algorithms

Load balancing algorithms could be of various types keeping in view the principles behind their working. These algorithms can be based on centralized model or decentralized model, static or dynamic model, with threshold or without threshold model, periodic or instantaneous model. The algorithms can be designed using more than one principle to have more effective approach. Each approach has its pros and cons, so before designing our model the designer has to keep in view the basic system requirements and architecture which the approach has to support. Primarily all load balancing algorithms are classified into static and dynamic with all the algorithms mainly falling under these two categories only. Some of the major static and dynamic algorithms are as follows.

- **Random Algorithm:** The random algorithm is the simplest load balancing algorithm supported by the system. It is static, selecting a host for a new thread when the thread is being created. The thread runs on this host during its entire execution. Here the host is selected at random from the set of processors participating in the application execution. There is no predefined rule except the random selection process. The random algorithm can produce even load distribution also and an uneven load distribution also. The process is simple and easy to design without any overhead in after initial allotment. An advantage of random algorithm is the absence of load balancing inter-process communication which increases message overhead. Hence, the scheme can even attain the best performance among all the load balancing algorithms for particular special parallel applications. Nevertheless, random algorithm is not expected to achieve good performance in the general case [11].
- **Round Robin Algorithm:** The round robin algorithm is a static load balancing scheme where new threads are divided evenly between all the processors. The threads are assigned to processors in a “round robin” order, i.e., each new thread is sent to the next processor. The order of thread allocation is maintained on each

processor locally, independent of allocation from remote processors. As the jobs are created each job is assigned to the processor and next job to the next processor. The algorithm does not assess the viability of job allotment on individual processor. There could be the case that some processors are fast enough to execute the jobs where other processors are still lagging behind (typical case of heterogeneous system). As the process is static the allotment is done according to round robin rule only. An advantage of round robin algorithm is the absence of load balancing inter-process communication which increases message overhead. Hence, the scheme can even attain the best performance among all the load balancing algorithms for particular special parallel applications. Nevertheless, round robin algorithm is not expected to achieve good performance in the general case. A typical model of round robin algorithm is shown in Figure 2.4 [11].

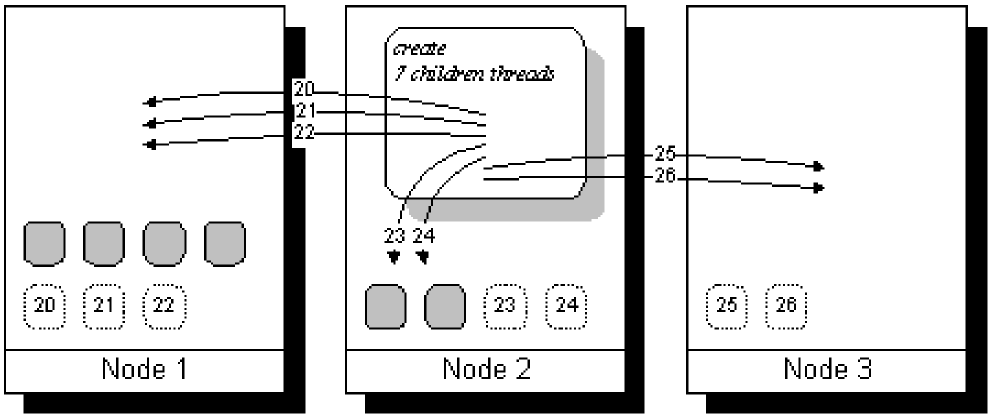


Figure 2.4-Round Robin Load Balancing

- **Central Load Manager Algorithm:** The central load manager algorithm is a static load balancing algorithm where a host for allocation of a new thread is selected by the central load manager [10, 11]. The thread is allocated to the minimally loaded host. The central load manager runs on the main host known to all remote load managers. All requests for host selection are sent to the central load manager. If a parent thread runs on the main host, then the central load manager is called directly without sending a message. Hosts for new threads are selected by the load manager so that the processor load after thread allocation is as uniform as possible. The central load manager reaches a decision based on the available information on the system load state. This information is updated by remote thread managers, which send a message each time the load on their nodes changes. The message overhead of the central load manager algorithm is one message for each change of load and two messages per thread allocation from remote hosts. The typical model of central load manager algorithm is depicted in Figure 2.5. A general disadvantage of all static schemes is that the final selection of a host for thread allocation is made when the thread is created, and cannot be changed during thread execution to accommodate changes in the system load. All the same, the central load manager scheme is expected to perform much better than the simpler schemes for parallel applications,

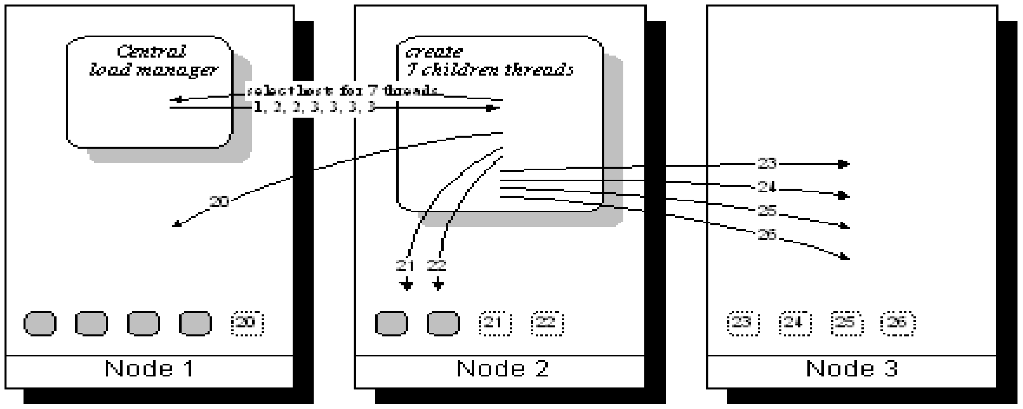


Figure 2.5- Central Load Manager Algorithm

- **Threshold Algorithm:** According to this algorithm, the threads are allocated immediately upon creation to hosts selected by the load manager. The load manager is distributed between the processors, and hosts are selected locally without sending remote messages. Each local load manager keeps a private copy of the system's load state. The load state of a processor is characterized by one of the following three levels: under loaded, medium and overloaded. These levels are defined by two threshold parameters, T_{under} and T_{upper}, which can be defined by the user: a processor is under loaded when load < T_{under}; medium when $T_{under} \le$ load $\le T_{upper}$ and overloaded when load > T_{upper}. Default values of $T_{under} = 2$

ready threads and T_{upper} = 4 ready threads. Initially, all the processors are considered to be under loaded. When the load state of a processor exceeds a load level boundary, the local load manager sends messages regarding the new load state to all remote load managers, constantly updating them as to the actual load state of the entire system. A host is selected for a new thread according to the following algorithm: if the local state is not overloaded then the thread is allocated locally; otherwise, a remote under loaded host is selected, and if no such host exists, the thread is also allocated locally. The message overhead of the algorithm is N-1 messages for every exceeding load level boundary on a processor, where N is the total number of processors. Among the advantages of the thresholds algorithm are relatively low inter process communication and a large number of local thread allocations. A disadvantage of the algorithm is that all threads are allocated locally when all remote processors are overloaded (their load is more than the constant parameter T_{upper}). A load on one overloaded processor can be much higher than on other overloaded processors, causing significant load imbalance, and increasing the execution time of an application. The typical model of threshold algorithm is depicted in Figure 2.6 [11].

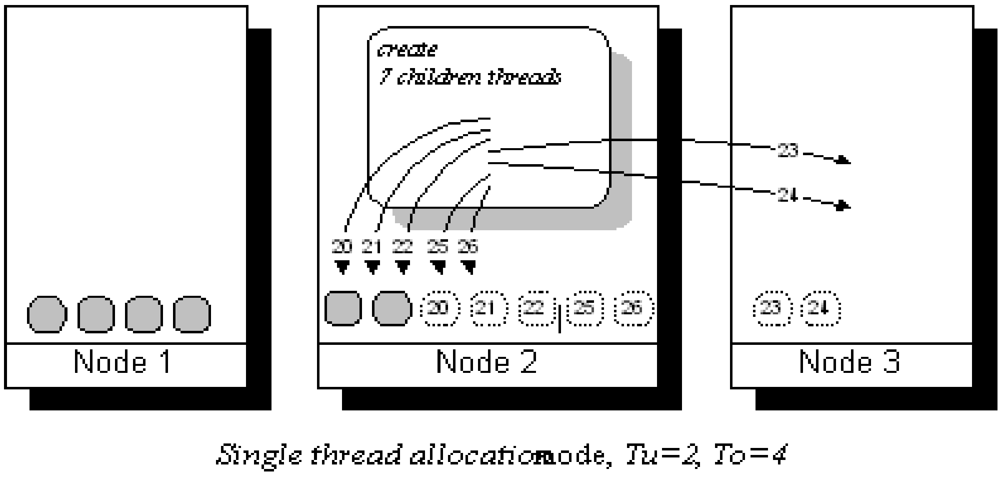


Figure 2.6- Model of Threshold Algorithm

- **Central Queue Algorithm:** The central queue algorithm is a dynamic load balancing algorithm where new parallel activities are not allocated immediately after creation. Instead they are buffered in the central thread-request queue on main host and allocated dynamically upon request from remote hosts. Queue is maintained by the central load manager running on main host. The purpose of the central thread-request queue is to store new activities and unfulfilled requests. It is structured as a cyclic FIFO queue on the main host. Each new activity arriving at the queue manager is inserted into the queue. Then, whenever a request for an activity is received by the queue manager, it removes the first activity from the queue and sends it to the requester. If there are no ready activities in the queue, the request is buffered until a new activity is available. If a new activity arrives at the queue manager while there are unanswered requests in the queue, the first such request is removed from the queue and the new activity is assigned to it. The central thread-request queue can contain in any given moment either new activities or unanswered requests; they cannot be interleaved in the queue. When a processor load falls beneath the threshold T_{lower}, the local load manager sends a request for a new activity to the central load manager [11].

The central load manager answers the request immediately if an activity is found in the thread-request queue, or queues the request until a new activity arrives. The parameter T_{lower} is user-defined as the minimal number of ready threads on each processor. Its default value is two ready threads. The central queue algorithm provides at least T_{lower} ready threads on each processor if a sufficient number of activities have been created. The message overhead of the central queue algorithm is three messages per parallel activity (one message transfers a new thread to the central load manager, another makes the request and the third is for thread allocation). The load manager running on the main host does not send any messages to the central load manager, but rather requests new activities directly from it, decreasing the overall message overhead of the algorithm.

The most important advantage of the central queue algorithm is dynamic distribution of threads. Unlike static algorithms, dynamic algorithms allocate threads dynamically when one of the processors becomes under loaded. The working of central request queue is depicted in Figure 2.7.

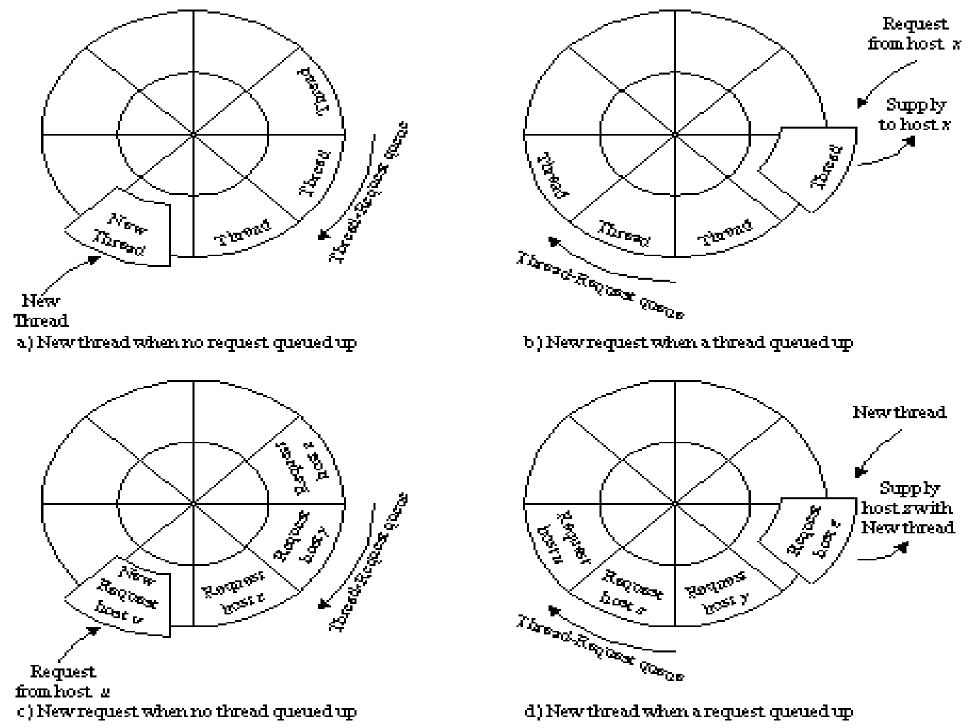


Figure 2.7- Central Thread Request Queue Working

- **Local Queue Algorithm:** Local queue algorithm is a dynamic load balancing algorithm where local queue is maintained at each node with new threads as entries. The basic idea of the local queue algorithm is static allocation of all new threads with thread migration initiated by a host when its load falls beneath a threshold T_{under} where T_{under} is a user-defined parameter of the algorithm with default value of 2. The parameter defines the minimal number of ready threads the load manager attempts to provide on each processor if at least one host with more than T_{under} ready threads exists. The local load manager attempts to get several threads from remote hosts. It randomly sends synchronous requests with the number of local ready threads to remote load managers. When a load manager receives such a request, it compares the local number of ready threads with the received number. If the former is greater than the latter, then some of the running threads are transferred to the requester and a positive confirmation with the number of threads transferred is returned. A negative reply is sent to the requester if the local number of ready threads is less than the number received. If the requester receives a negative reply, or if the number of threads received is not sufficient to reach the T_{under} threshold, the load balancing process is continued with another remote processor. If, after trying all remote processors, the T_{under} threshold is still not reached, the load balancing is periodically repeated until the threshold is met. All the hosts apart from the main one also initiate periodic load balancing at the beginning of an application execution, until the T_{under} threshold is achieved. The local queue load balancing algorithm is expected to achieve the best performance, as it is dynamic and can redistribute running threads during application execution. Static allocation of new activities decreases the overhead of remote thread allocations and the overhead of remote memory accesses, thus improving performance significantly. Another advantage of the algorithm is that its message overhead is relatively low; messages are sent only when a host becomes under loaded and thread redistribution is required. One apparent drawback of the algorithm is that it ignores the locality principle. A thread for transfer is selected randomly regardless of the threads running on the under loaded and local processors. This decreases the performance of parallel applications with massive data exchange between subsequent parallel iterations or blocks [11].

2.5 Review of Load Balancing Strategies

A dynamic load balancing mechanism for distributed system is proposed in [10] with adaptive threshold where central node is used for maintaining load state information and decision for balancing is taken at local nodes. Six load balancing strategies are studied in [11] with application on four problems. These schemes include random, round robin, central load manager, threshold, central queue and local queue. In [12] various strategies for dynamic load balancing are explored which include sender initiated diffusion, receiver initiated diffusion, hierarchical balancing method, gradient model, domain exchange method. Loop re-partitioning has been reported as a runtime load balancing function for data parallel applications [13]. Although dynamic and guided options of OpenMP can achieve load balancing to some extent a profiled clause is added to the schedule clause in OpenMP to optimize dynamic load balancing where schedule is given by Schedule (profiled [chunk_size]). In [14] the generic N+ 1 dimensional perfectly nested loop is parallelized across the outermost N dimensions, so as to perform sequential execution along the innermost dimension in a pipeline fashion, interleaving computation and communication phases. The parallelization of outermost loops is done according to the tiling transformation. A simple load balancing strategy for task allocation in parallel machine has been proposed in [15] where load balancing is decentralized and execution of load balancing is decided among processors using the local queue length of individual processor. The processor with minimum queue length is given task of executing the load balancing. A comparison of three approaches of guided self scheduling, irregular parallel programs and lazy task creation without taking data locality into consideration has been done in [16]. It employs dynamic load balancing scheme implementing central queue and local queue while considering data locality problem.

2.6 Quality of Service Parameters in Load Balancing

Load balancing is a method to ensure a uniform distribution of load over the constituent nodes. Load balancing can be used for improving the system performance considering various QoS parameters. Thus a load balancing strategy can be designed while considering either one or a combination of many QoS parameters. Some of the QoS parameters are listed below.

- **Throughput:** The amount of work performed by a computer within a given time. It is a combination of internal processing speed, peripheral speeds (I/O) and the efficiency of the operating system, other system software and applications all working together. Transactions processed per second (TPS) is one metric commonly used to gauge throughput.
- **System Utilization:** It is to keep system as busy as possible so that no resource is ever kept idle and it has work to execute.
- **Turnaround time:** It is estimated as the time taken by the job from its submission to the final execution. Thus, it is always expected from a scheduler to allocate the job to those resources which results in the faster overall execution of the job i.e. with minimum turnaround time.

- **Waiting time:** It is the amount of time spends to wait by a particular job in system for getting a resource. In other words waiting time for a job is estimated as the time taken by the job from its submission to the get system for execution. The waiting time depend on the parameters similar as turnaround time.
- **Response time:** It is the amount of time to get first response in time sharing system. The response time depend on the parameters similar as turnaround time.
- **Fairness:** It is defined as the time taken by each system in load balancing environment is same. Fairness deals with fair utilization of each available resource such that no resource is over utilized and no resource is underutilized.
- **Reliability:** It is the ability of a system to perform failure free operation under stated conditions for a specified period of time.

2.7 Load Balancing NP-Complete Optimization Problem

Computation problems broadly can be classifies as two class of problems, P class and NP class. The types of problem which can be solved by exact methods in polynomial time are the polynomial time solvable problems referred to as class P problem. An algorithm is said to be polynomial or a polynomial-time algorithm, if it’s running time is bounded by a polynomial in input size. The other class of optimization problems is known as NP-hard (NP-complete) problems. For such problems, no polynomial-time algorithms are known and it is generally believed that these problems cannot be solved in polynomial time. If a problem is NP-complete it is likely that it does not admit a polynomial-time algorithm, and should be treated by some other means [1].

Another class of problem is decision problem. A problem is called a decision problem if the output range is decidable. P is the class of decision problems which are polynomial time solvable. NP is the class of decision problems with the property that for each “yes”-answer, a certificate exists which can be used to verify the “yes”-answer in polynomial time. For two decision problems R and Q, we say that R reduces to Q (denoted by $R \alpha Q$) if there exists a polynomial-time computable function g that transforms inputs for R into inputs for Q such that x is a “yes”-input for R if and only if g(x) is a “yes”-input for Q. If R and Q are decision problems and $R \alpha Q$ then $Q \in R$

implies R $\in$ R (and, equivalently, R $\notin$ R implies Q $\notin$ R). A transitive relationship exists between decision problems. If Q, R, S be decision problems and R α Q, Q α S, then R α S. A decision problem Q is called NP-complete if Q $\in$ NP and for all other decision problems R $\in$ NP, we have R α Q. If any single NP-complete decision problem Q could be solved in polynomial time then we would have P = NP. To prove that a decision problem R is NP-complete it is sufficient to prove the following two properties:

i. R $\in$ NP, and

ii. There exists an NP- complete problem Q with Q α R.

An optimization problem is NP- hard if its decision version is NP-complete. For such problems, no polynomial-time algorithms are known and it is generally believed that these problems cannot be solved in polynomial time. Most of the scheduling problems (including load balancing) are optimization problems, i.e., a schedule that optimizes a certain objective function. Load balancing is an NP-Complete problem owing to the large number of resources and jobs along with their heterogeneous nature demanding scheduling. The input size of a typical balancing problem is bounded by the number of jobs 'n', the number of machines 'm' [1].

Load-balancing problem falls into the ``easy class" of NP-complete optimization problems [20]. Computational complexity theory provides a mathematical framework that explains why some problems are easier to solve than the others. It is accepted that more computational complexity means problem is harder and vice versa with their computational complexity depending on their input size and the constraints imposed on it. Irregular loosely synchronous problems consist of a collection of heterogeneous tasks communicating with each other at the synchronization point, is the characteristic of this problem class. Both the execution time per task and amount and pattern of communication can differ from task to task. It is noted that formally this is a very hard-so-called NP-complete-optimization problem. With $\boldsymbol{N_{task}}$ tasks running on $\boldsymbol{N_{proc}}$ processors we cannot afford to examine every one of the $N_{task}C_{N_{proc}}$ assignments of tasks to processors. This problem is easier as one does not require the exactly optimal assignment. Rather, a solution whose execution time is within 10% of the optimal value can be quite acceptable. The physical optimization methods and more problems specific heuristics have shown themselves very suitable for this class of approximate optimization problems [20].

2.7.1 Solution to NP Class Problems

At present, all known algorithms for NP-complete optimization problems require time that is super polynomial in the input size, and it is unknown whether there are any faster algorithms. Small sized problems can be solved by mixed integer, linear programming, dynamic programming, branch and bound methods. To find a "good" solution within an acceptable amount of time for problems of larger size, two types of algorithms can be used:

- **Approximation Algorithms:** Approximation algorithms are algorithms used to find approximate solutions to optimization problems. An algorithm is called an approximation algorithm if it is possible to establish analytically how close the generated solution is to the optimum. Approximation algorithms are often associated with NP-hard problems as it is implausible that there can be efficient polynomial time exact algorithms solving NP-hard problems, so one has to settle for polynomial time sub-optimal solutions. Ideally, the approximation is optimal up to a small constant factor (for instance within 5% of the optimal solution). Approximation algorithms are increasingly being used for problems where exact polynomial-time algorithms are known but are too expensive due to the input size [1].

- **Heuristics**: A heuristic is a rule of thumb for solving NP Complete problems. Heuristics are often used to improve efficiency or effectiveness of optimization algorithms, either by finding an approximate answer when the optimal answer is prohibitively difficult or to make an algorithm faster. Heuristics do not guarantee that an optimal solution to the problem is always found however results about NP-hardness in theoretical computer science make heuristics the only viable alternative for many complex optimization problems which are significant in the real world. The performance of a heuristic algorithm is usually analyzed experimentally, through a number of runs using either generated instances or known benchmark instances [1].

Chapter 3

The Proposed Model

Scheduling is the problem of mapping the job(s) on the system resources keeping in mind that all the nodes in the system get equal share of work resulting in minimizing the turnaround time. The proposed model presents a centralized load balancing strategy for job(s) submitted for execution on a symmetric multiprocessor system Sun Fire X 4470 server with the aim of minimizing the turnaround time. The model presented follows a centralized dynamic load balancing with threshold which is used to study the system load at a given moment of time. The threshold values corresponds to the minimum and maximum workload assigned to the nodes and are adaptive in nature being adjusted periodically as the load on the system increases for even distribution of the load. When the load is evenly distributed these values becomes approximately equal depicting the balanced state of the system. The load redistribution process is fair as it is periodically adjusted between most heavily loaded nodes and most lightly loaded nodes through the use of max priority queue and min priority queue. The balancing process utilizes minimum CPU time as redistribution is only carried out when lightly loaded and heavily loaded nodes are reported. The chapter starts with presentation of the scheduler while discussing the parameters and fitness functions considered for design of the model and the scheduling algorithm used. This is followed by an example to illustrate the working of the model. The chapter concludes with simulation study and their analysis.

3.1 The Scheduler

The aim of parallel and distributed system is to primarily minimize the turnaround time of the job submitted for execution. The parallelism for the system can be considered from both the hardware and the software angle. Hardware parallelism refers to the presence and availability of multiple computational resources while parallelism at the software level refers to the parallelism inherent in the application in the form of individual grains of program that can be executed simultaneously. Thus, if the application is scheduled on the available nodes in such a way that maximum software parallelism is exploited can result in the best performance. Symmetric multiprocessor system provides a hardware platform with identical processors which can be used for scheduling any parallel application. A mere presence of multiple computational resources does not ensure maximum performance if the scheduling strategy is incapable of spreading the parallel executable modules evenly on these resources. Scheduling on a multiprocessor system can be done to optimize any Quality of Service (QoS) parameters like Turnaround Time, Reliability, Throughput etc. or could be a combination of a few QoS parameters.

The proposed model presents a centralized dynamic load balancing strategy which continuously keeps a track of the load on the nodes using threshold with the aim of minimizing the turnaround time of the jobs submitted for execution. The centralized scheduling approach is adopted by the model to ensure minimum traffic overhead in comparison to the distributed approach in which a lot of messages are to be exchanged to update the locally maintained load information table. The model uses Sun Fire X 4470 server as the test bed providing the users with a maximum of 32 cores for the job execution. These cores act as the processing elements on which jobs can be submitted for execution. Since the model follows centralized job scheduling approach, of the available cores, one core is taken as central scheduler on which job has got submitted and is eventually used for dispatching the independent job modules to the other cores (processing elements). Each processing element has a local queue where the allotted jobs are queued up and are taken up for execution one by one in the order of their arrival. The scheduler used in the work is as shown in Figure 3.1.

The job submitted for execution can be considered to be comprising of sub-modules which can run in parallel and are independent in nature so that there is no order in job execution and any job/sub-module can finish its execution irrespective of job number or order of precedence in the job. The process starts by randomly allocating these sub-modules of the job(s) to the processing elements. This random allocation results in a possible scenario in which few of the nodes gets a large number of sub-jobs to execute while some may get very less or no module to execute. This result in an imbalanced state with few processing elements heavily loaded and few are remaining idle thus demanding

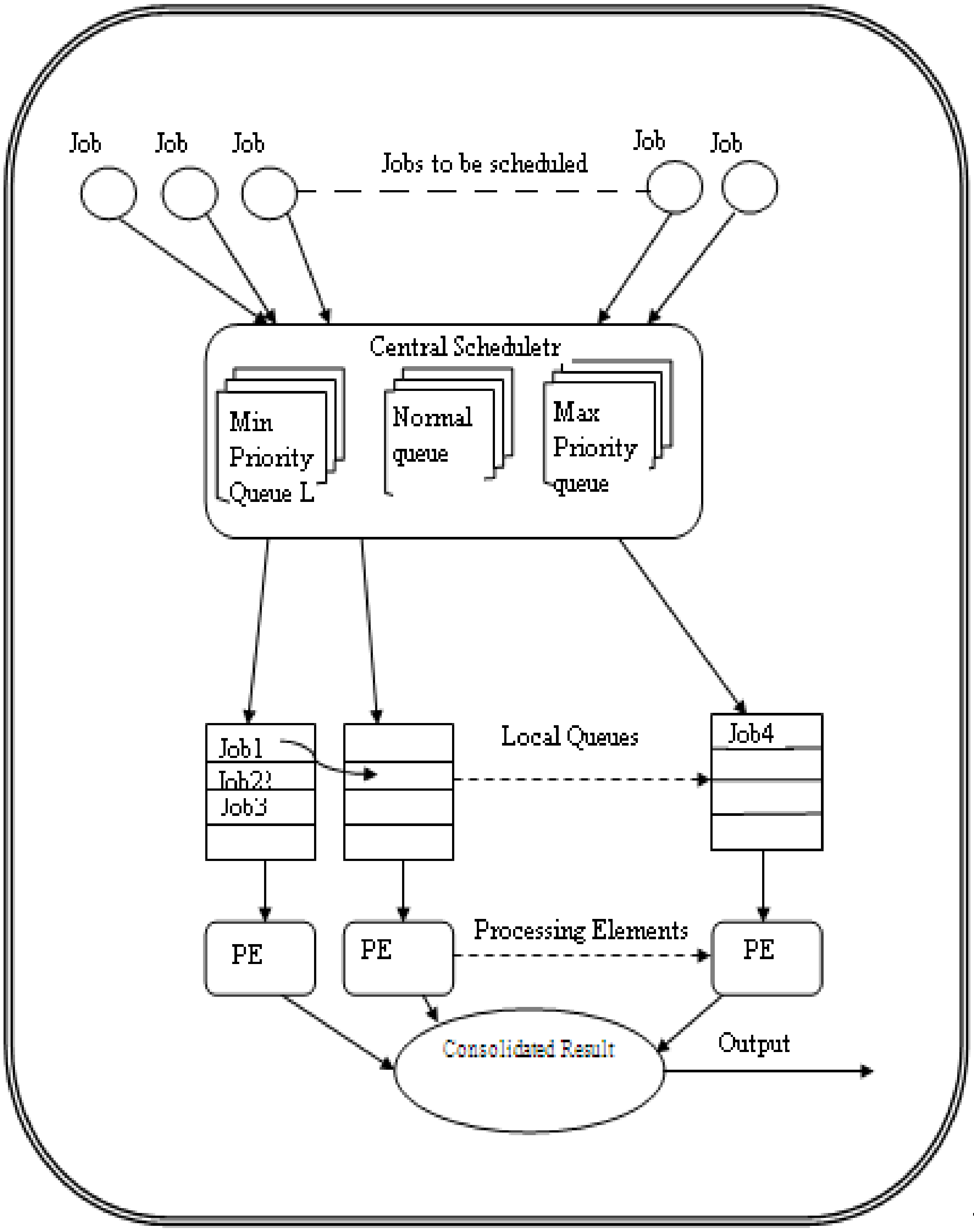


Figure 3.1 – Model of Central Scheduler

load balancing which becomes the additional responsibility of the central scheduler. Whenever, the model experiences an uneven distribution of load, a readjustment of load is initiated to evenly distribute the load over the nodes till a balanced state is reached. The load on the nodes is evaluated by using two threshold values viz. Lower Threshold (T_{lower}) and Upper Threshold (T_{upper}) values which are adaptive by nature. T_{lower} corresponds to threshold value indicating the minimum workload that is allowed on a node while T_{upper} representing the maximum value of the workload accepted on any node. As a node is assigned a workload, the same enters its job execution queue. The global queues are maintained by the central scheduler only and are implemented as maximum priority queue for heavily loaded nodes and minimum priority queue for lightly loaded nodes. As the load on the system changes these thresholds are adjusted to suite the changing load on the system making the threshold selection adaptive i.e. the threshold values increases with increasing load and vice versa. As the average number of jobs in local queues of processing element increases, the threshold values are readjusted and so the global queues regarding the normal loaded nodes, lightly loaded nodes and heavily loaded nodes are adjusted. The process is continuously repeated till the load is evenly distributed on the computing nodes with T_{lower} and T_{upper} becoming approximately equal depicting a balance state of the system. The load balancing process is instantaneous. As soon as the heavily loaded nodes and lightly loaded nodes are reported, the central scheduler starts load balancing between the nodes responsible for the imbalance. The load balancing strategy is fair as the load is adjusted first between most heavily loaded node and the most lightly loaded node with the process being repeated for the next most heavily loaded nodes and the next most lightly loaded nodes using max priority queue and min priority queue. Further, the balancing process utilizes minimum CPU time as redistribution is only carried out when lightly loaded and heavily loaded nodes are reported. The scheduling strategy has been explained in detail in the next section.

3.1.1 Scheduling Strategy and Algorithm Used

The scheduler aims in minimizing the turnaround time for the job(s) submitted for execution by effectively load balancing the jobs on various computing nodes available. This further, adds towards better utilization of the computational resources as well. The various parameters used in the model are presented in Table 3.1 along with their description.

Table 3.1-Parameter Used in the Model

Parameters	**Description**
K	Number of nodes
J	Number of jobs
J_i	Job identifier where 1 <= i <= J
N_i	Node identifier where 0 <= i <= K-1
l_i	Workload on each node N_i
T_{lower}	Lower threshold
T_{upper}	Upper threshold
LHM	Lower half mean of l_i for the nodes sorted in ascending order
UHM	Upper half mean of l_i for the nodes sorted in ascending order
M	Mean of l_i for the nodes sorted in ascending order
L	Min priority queue containing node identifier for nodes having load l_i below T_{lower}
H	Max priority queue containing node identifier for nodes having load l_i above T_{upper}
X	Queue for nodes having load between T_{lower} & T_{upper}
LQ_i	Local queue of jobs for each processing element N_i
LQL_i	Length of the local queue for each processing element N_i

The scheduler uses the Sun Fire X4470 Server as a test bed which comprises of 4 processors each with 8 cores. Therefore, the maximum number of nodes available to the scheduler becomes 32 represented by K. Since the test bed for the scheduler is Sun Fire X4470 Server, the processing elements are homogenous. The individual node under consideration has been represented by N_i where 0 <= i <= K-1. The load on each node is given by l_i. The model uses the centralized approach for load balancing. Thus, out of the nodes selected for job execution, one node is used as central scheduler who serves two objectives viz. dispatching the jobs to the remaining nodes and making load balancing decisions depending on the system state. The remaining nodes simply act as processing elements for jobs execution. Each processing element has a local queue where jobs can be queued. The central dispatcher/scheduler node maintains the load information of each processing element by maintaining the information about the nodes with high load, low load and normal load using threshold values. This is done using the priority queues being H for heavily loaded nodes, L for lightly loaded nodes and X for normal loaded nodes. If L and H queue is non empty then jobs from node in H are transferred to node in L. If any of the queues L or H is empty, load balancing will discontinue as this is the stopping condition for balancing.

Since the scheduler load balances the workload using thresholds, these values for under loaded nodes and overloaded nodes are considered as T_{lower} and T_{upper} which are calculated using Lower Half Mean (LHM), Upper Half Mean (UHM), and Mean M values. The nodes are sorted in ascending order of their workloads before calculating LHM, UHM and M such that $l_i >= l_{i-1.}$ LHM, UHM and M are calculated using equations (i) – (iii).

$$LHM = 1/((K-1)/2)\sum_{i=1}^{i=(K-1)/2} l_i \quad ---------------------- (i)$$

$$UHM = 1/((K-1)/2)\sum_{i=(K-1)/2+1}^{i=K-1} l_i \quad ---------------------- (ii)$$

$$M = 1/(K-1)\sum_{i=1}^{i=K-1} l_i \quad ---------------------- (iii)$$

The values of T_{upper} and T_{lower} are calculated using Upper Half Mean, Lower Half Mean and Mean of load of all nodes sorted according to workload. These values are readjusted as load on individual node changes. We want maximum number of nodes whose load is normal. In our definition normal load is the load which is approximately equal to average load of the system. Numbers of nodes whose load is normal falls under the range of T_{upper} and T_{lower}. The model has been implemented taking work offloading as basic load redistribution strategy. Before the node becomes completely idle it receives a share of work from other heavily loaded nodes where as in work stealing the node asks for share of load from heavily loaded nodes when it completely becomes idle. So no node is idle if extra load is there on any node in a system. Moreover the load redistribution criteria makes system resistive towards imbalance as same node does not result in imbalance after load redistribution. Here the heaviest loaded node is balanced with lightest loaded node first making balancing process fair. Using equations (i) – (iii), T_{lower} and T_{upper} can be calculated as equations (iv)-(v).

$$T_{lower} = \begin{cases} LHM: LHM >= 0.9M \\ 0.9\,M: LHM < 0.9M \\ 1: LHM < 1, 0.9M < 1 \end{cases} \quad \text{------------------------------------} \; (iv)$$

$$T_{upper} = \begin{cases} UHM: UHM <= 1.1M \\ 1.1M: UHM > 1.1M \\ 2: UHM < 2, 1.1\,M < 2 \end{cases} \quad \text{------------------------------------} \; (v)$$

In proposed model both T_{upper} and T_{lower} are adaptive in nature. LHM and UHM provide us the reference points using which T_{lower} and T_{upper} are set. The scheduler works with the intention of bringing that state of the system in which both LHM and UHM (and hence T_{lower} and T_{upper}) ranges between ±10% of the mean M resulting in a load balanced state. If LHM and UHM are outside this range T_{lower} and T_{upper} are set to be 90% and 110% of M respectively. Thus the scheduler continues to load balance the system to bring the average workload between ±10percent of the mean M.

Initially the values of thresholds T_{lower} and T_{upper} are taken as 1 and 2 respectively and are gradually adjusted using the node's workload sorted in the ascending order. Now, as the load on a node increases the value of thresholds are readjusted and accordingly the number of nodes in L, H and X keeps on changing. Sorting the nodes in terms of their workload enables the scheduler to have an idea about the nodes and their workloads. Further, in this way, the nodes gets divided into under loaded, overloaded and normal loaded which are handled via minimum priority queue L, maximum priority queue H and queue X which are the workload queues for lightly loaded, heavily loaded and medium loaded nodes. The scheduler then tries to converge the workload of these nodes towards the mean value M. Nodes belonging to L, H and X can be decided using equation (vi), (vii) and (viii) respectively.

$$N_i \in L \text{ if } l_i < T_{lower} \quad \text{------------------------------------} \; (vi)$$

$$N_i \in H \text{ if } l_i > T_{upper} \quad \text{------------------------------------} \; (vii)$$

$$N_i \in X \text{ if } l_i >= T_{lower} \,\&\, l_i <= T_{upper} \quad \text{------------------------------------} \; (viii)$$

As the values of LHM and UHM approaches M the system approaches balanced state with even distribution of workload. The load balancing process is instantaneous in nature. As the heavily loaded and lightly loaded node are reported, the central scheduler assumes its job of load redistribution deciding on the nodes for job transfer and the number of jobs that needs to be transferred. Since the nodes are in ascending order of workload, the job transfer is done in such a way that it is done between most heavily loaded node (last in the order) to the lightest node (first in the order). The process of load redistribution continues for remaining number of nodes in L and H, reporting lightly loaded and heavily loaded status until either of the queue L or H becomes empty. This way, the load between these two nodes in which load distribution has taken place, becomes approximately equal. Simultaneously, the threshold values are also adjusted with the changing queue lengths thereby changing the values in H, L and X as well. It is necessary that once the load has got redistributed the same node doesn't become imbalanced quiet frequently. Accordingly, the number of jobs that are transferred from a heavily loaded node to the lightly loaded node is governed by equation (ix).

$$\textit{Number of jobs to be transferred} = (l_{i \in H} - l_{j \in L})/2 \quad \text{----------------------------} \; (ix)$$

The model aims to minimize the turnaround time for the jobs submitted for execution. Initially, the jobs are submitted randomly to individual processing elements by the central scheduler and afterwards, load balancing is initiated as discussed above. Accordingly, the load balancing strategy on an average results in the total number of jobs executed by individual processing elements to be near the average value of the workload. Since, this result in an even distribution of load on all the processing elements, the turnaround time for the job is minimized. The processing element with maximum number of jobs for executed decides the overall turnaround time. At any moment of time, the scheduler ensures the load on individual nodes to be around the average workload of the jobs submitted for execution.

This model is best worked around in situation where jobs arrival rate is very high in comparison to service rate. So, if the jobs are heavily fired, resulting in changing load on the nodes, the scheduler changes the threshold values to adapt to such situation. If the workload remains constant or very small, the system results in unnecessary initiation of load balancing thereby resulting in thrashing. Therefore, the model is best suited for the job execution scenarios with heavy workloads. The algorithm for the load balancing strategy is presented in the box.

```
Load_Balancer ()
{   Submit Jobs                      // Submit the jobs for execution
    Initialize ()                    // Select $N_0$ as central dispatcher and scheduler
                                     // $T_{lower} = 1$, $T_{upper} = 2$, $LQL_i = 0$
                                     // Move all nodes to minimum priority queue L

    For Processor $N_0$
      {
        Do
          {
            Allocate ($N_i$, $J_i$)          //randomly allocates jobs to nodes
            $LQL_i = LQL_i + 1$              // Update queue length with each allocation
            Sort ()                          // Sort nodes in ascending order as per their LQL
            Calculate LHM, UHM & M
            Calculate $T_{lower}$, $T_{upper}$
              Update ()                      // Update L, H and X
          }
        If ($H \neq NULL$ and $L \neq NULL$)
          {
            Extract ($N_i$)          // Extract the heavily loaded node $N_i$ from H
            Extract ($N_j$)          // Extract the lightly loaded node $N_j$ from L
            Transfer ($N_i$, $N_j$)  // Transfer jobs from node $N_i$ to node $N_j$
            Execute ($J_j$)          // Execute the jobs allocated
          }
        } while ($LQ_i \neq NULL$)
      Calculate TAT     //  TAT =Time taken by the processing element assigned with
                        //  maximum number of jobs
}
```

The algorithm starts with submission of the jobs demanding execution on the multiprocessor system in the format as discussed in Section 3.1. Therefore, a job is considered to be comprising of sub-modules which can run in parallel. Node N_0 is selected as central dispatcher and scheduler with the remaining nodes simply acting as processing elements for the job execution. Once the jobs are submitted, LQL_i for each processing element/ node is initialized to 0 with T_{lower} and T_{upper} being assigned the values of 1 and 2 respectively. All the nodes using node identifiers are then moved to minimum priority queue L. The central scheduler N_0 is then assigned the jobs that need to be executed by dispatching them randomly to the individual processing elements.

As soon as the jobs are assigned to the processing elements, the parallel execution can start. The jobs assigned to each processing elements are first allocated to their local queue LQ_i with the local queue length LQL_i updated accordingly. The processing elements are sorted simultaneously according to the queue length LQL_i to calculate LHM, UHM and M as per equations (i)-(iii). The central scheduler then modifies the threshold values T_{lower} and T_{upper} as per the changed values of LHM, UHM and M as per equations (iv)–(v). According to the new threshold values, minimum priority queue L, maximum priority queue H and queue X gets updated as per equations (vi)-(viii) to group the nodes as lightly loaded, heavily loaded and medium loaded nodes respectively. If L and H are not empty, the central scheduler starts load balancing the workload by transferring jobs from most heavily loaded node to the most lightly loaded node as per equation (ix). This process continues till either minimum priority queue L or maximum priority queue H becomes empty. While the central node N_0 is busy accepting new jobs, dispatching them and load balancing, the processing elements N_i (where $i \neq 0$), continue extracting the jobs assigned to them from their local queues and executing them.

The job execution continues till there is no job to execute and each local queue of individual processing element becomes empty. The turnaround time for the jobs submitted depends on which processing element is taking the maximum time in execution. Therefore, the TAT for the jobs submitted becomes equal to the time taken by that processing element which has executed the maximum number of jobs. Effectively, for the job submitted, each processing element gets the number of jobs nearly equal to the average of the workload of the system. Thus, the algorithm ensures a uniform distribution of load resulting in effective resource utilization.

3.2 Illustrative Example

To better understand the model, an example is illustrated in this section to present the basic working of the model in terms of the turnaround time computation. The example considers that no new job is added to local queue of a node and no job is taken away from the queue until the load balancing is done making it static whereas in practice, the model performs load balancing on the workload dynamically. In other words, the model considers the job service rate to be less than job arrival rate leading to removal of no job from the queue until the allotment has been done.

The example considers a scenario with a total number of available nodes for execution as 11. As per the scheduling strategy presented in Section 3.1.1, N_0 acts as the central node and N_1 to N_{10} acting as the processing elements for job execution. Load on each node N_i is represented by $l_{i.}$ Initially $T_{lower\ and}$ T_{upper} are assumed to be 1 and 2 respectively. Total 134 jobs are assumed to be submitted to the system for execution and the allotment after random distribution of the load is as shown in the Table 3.2. Therefore, each entry in the table opposite to the node identifier indicates the number of jobs assigned to a node. The allotment clearly suggests an unbalanced state of the system thus prompting the scheduler to take corrective measures.

Table 3.2-Initial Allocation of Load

N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}
1	2	3	4	6	9	15	22	25	47

Initially the nodes are sorted in ascending order of their workload. In this case, the allotment is already in the sorted form. The process starts with the calculation of LHM, UHM and M. Using equation (i), LHM is calculated as the mean of workload on N_1, N_2 N_3, N_4 and N_5 which are the nodes in the lower half of the table sorted in ascending order and is calculated as

LHM = (1+2+3+4+6)/5

= 3.2.

Similarly, UHM is also calculated as per equation (ii) which is the mean of load on N_6, N_7, N_8, N_9, and N_{10} and is calculated as

$$\begin{aligned} UHM &= (9+15+22+25+47)/5 \\ &= 23.6. \end{aligned}$$

The value of M is then calculated as per equation (iii) and is mean of load on N_1, N_2 N_3, N_4, N_5, N_6, N_7, N_8, N_9, and N_{10} which is calculated as

$$\begin{aligned} M &= (1+2+3+4+6+9+15+22+25+47)/10 \\ &= 13.4. \end{aligned}$$

It can be seen from the example till now that the system started with the threshold values T_{lower} and T_{upper} as 1 and 2 respectively. Here, the mean M of the workload is 13.4, requiring the T_{lower} and T_{upper} values to be modified to move the bias towards M which acts the average workload of the system. Since, the difference between LHM (3.2) and UHM (23.06) from M (13.4) is very large, it indicates that there are many nodes which are under loaded and overloaded necessitating the load balancing to continue. Accordingly using equations (iv) – (v), the new value of T_{lower} and $T_{upper\ can}$ be calculated as

$$\begin{aligned} T_{lower} &= \max(\max(LHM, 0.9M), 1) \\ &= \max(\max(3.2, 12.06), 1) \\ &= 12.06. \end{aligned}$$

$$\begin{aligned} T_{upper} &= \max(\min(UHM, 1.1M), 2) \\ &= \max(\min(23.6, 14.74), 2) \\ &= 14.74. \end{aligned}$$

The nodes that come under L and H as per equation (vi) – (vii) becomes

L (N_1, N_2 N_3, N_4, N_5, N_6)

H (N_{10}, N_9, N_8, N_7)

It can be seen that node N_1 is the most lightly loaded node with N_{10} being the most heavily loaded node. Thus node N_1 is workload balanced with N_{10} as per equation (ix) by transferring some jobs from $N_{10\ to}$ N_1. Similarly, N_2 is balanced with N_9, N_3 is balanced with N_8 and N_4 is balanced with N_7. This results in emptying the queue H. Therefore, the scheduler stops the load balancing for the moment. The resultant load on each node after redistribution is shown in Table 3.3.

Table 3.3: Load on Nodes after Balancing

N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}
24	13	12	9	6	9	10	13	14	24

The resultant nodes after load balancing are again sorted to calculate the new values of thresholds T_{lower} and T_{upper}. The nodes with new resultant load in sorted order are shown in Table 3.4. In practice, this process is dynamic as the jobs are added and removed from the queue simultaneously. However, the example just illustrates the working of model till load is balanced without considering the addition and removal of new jobs for the sake of simplicity.

Table 3.4- Sorted Nodes According to Load of Table 3.3

N_5	N_4	N_6	N_7	N_3	N_2	N_8	N_9	N_1	N_{10}
6	9	9	10	12	13	13	14	24	24

In the way as illustrated till now, the new values of LHM, UHM and M now becomes 9.2, 17.6 and 13.4 respectively. It can be seen now that difference between LHM and UHM with M has reduced considerably indicating some load balancing which can be observed from Table 3.4 as well where the distribution of workload is more uniform as compared to the initial state. Similarly, the values of T_{lower} and T_{upper} are calculated as 12.06 and 14.74 respectively. The nodes that come under L are N_5, N_4, N_6 and N_7 and with H are N_{10} and N_1.

Table 3.5-Load Redistribution of Nodes of Table 3.4

N_5	N_4	N_6	N_7	N_3	N_2	N_8	N_9	N_1	N_{10}
15	16	9	10	12	13	13	14	17	15

The load is again balanced and the result is shown in Table 3.5 with Table 3.6 presenting the same in the sorted order.

Table 3.6-Nodes in Sorted Order of Table 3.5

N_6	N_7	N_3	N_2	N_8	N_9	N_5	N_{10}	N_4	N_1
9	10	12	13	13	14	15	15	16	17

Again, the values of LHM, UHM and M are calculated and found as 11.4, 15.4 and 13.4 respectively with the values of T_{lower} and T_{upper} being 12.06 and 14.74 respectively. Now, the nodes that come under L are N_6, N_7 and N_3. The same for H becomes N_1, N_4, N_{10} and N_5. Table 3.7 presents the load balanced system after this step with Table 3.8 presenting the nodes in a sorted order according to their workload

Table 3.7- Load Redistribution of Nodes of Table 3.6

N_6	N_7	N_3	N_2	N_8	N_9	N_5	N_{10}	N_4	N_1
13	13	13	13	13	14	15	14	13	13

Table 3.8-Nodes in Sorted Order of Table 3.7

N_1	N_2	N_3	N_4	N_6	N_7	N_8	N_9	N_{10}	N_5
13	13	13	13	13	13	13	14	14	15

The new values of LHM, UHM and M are now calculated as 13, 13.8 and 13.4 respectively which are approximately equal. This is the driving condition which depicts the even distribution of load. The values of T_{lower} and T_{upper} are 13 and 13.8 respectively. Therefore, no node is found to be under L whereas nodes that come under H are N_5, N_{10} and N_9. The load is readjusted only when L and H is non empty. Since L has become empty, no load balancing is needed any further. The nodes will execute the jobs allocated to them till each node executes 13 jobs. The load status of nodes after execution of 13 jobs is shown in Table 3.9.

Table 3.9-Nodes after 13 Jobs Execution

N_1	N_2	N_3	N_4	N_6	N_7	N_8	N_9	N_{10}	N_5
0	0	0	0	0	0	0	1	1	2

The new values of LHM, UHM and M are 0, 0.8 and 0.4 respectively. So the values of T_{lower} and T_{upper} are 1 and 2 respectively. The nodes under L are N_1, N_2, N_3, N_4, N_6 and N_7 and there is no node under H. This state presents the other extreme in which L is non empty and H is empty again indicating the balanced state. Therefore, the nodes carry on execution till all local queues become empty. The final allocation of the workload to the nodes after complete load balancing is shown in Table 3.10 presenting the nodes with exact number of jobs allocated and hence executed. It can be seen that this value is near 13.4 which is the mean M of the workload.

Table 3.10-Nodes with Number of Jobs Allotted and Executed

Node	**N_1**	**N_2**	**N_3**	**N_4**	**N_5**	**N_6**	**N_7**	**N_8**	**N_9**	**N_{10}**
Allotted	1	2	3	4	6	9	15	22	25	47
Executed	13	13	13	13	13	13	13	14	14	15

In the above example, the number of jobs considered for execution is 134 with each job executing in 0.264 seconds. Therefore, the time taken by 134 jobs to execute sequentially on one processing elements becomes 35.376 seconds. However, for parallel execution, the total turnaround time (TAT) can be calculated as

*TAT= Max number of jobs executed on any node * Execution Time of a Single Job ----(x)*

In the given example, since the maximum number of jobs executed on any processing element is 15 as shown in Table 10, the TAT using equation (x) can be calculated as

*TAT= 15*0.264=3.96 seconds*

The speedup for such a system can be calculated as the ratio of the time taken T_{seq} by the job when executed sequentially on a node to the time taken for parallel execution T_{par}

Speedup 'S'= T_{seq} / T_{par} -- *(xi)*

= 35.376 /3.96

= *8.93*

As can be seen, the speedup obtained is 8.9 indicating approximately 900 % faster execution of the job. Similarly, the normalized speedup can be stated as

Efficiency ξ *=Speedup/Number of nodes* -------------------------------------- *(xii)*

= 8.93/10=0.89

Thus, the system is resulting in an efficiency of 89% which can be treated as fairly good.

3.3 Simulation Study

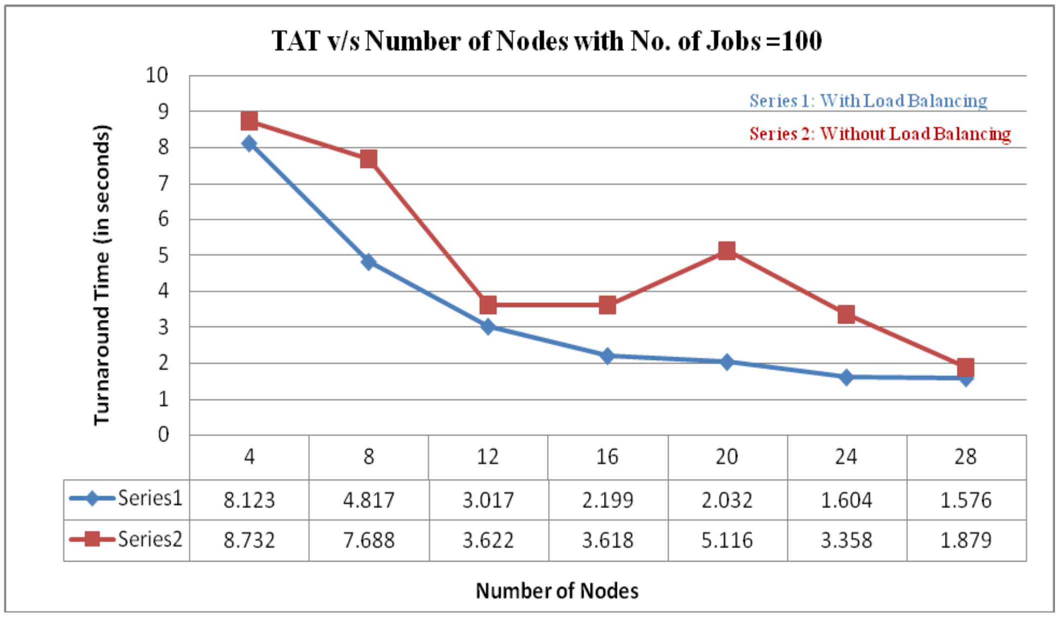
To evaluate the performance of the model, simulation study was performed. The SMP used for the study was Sun Fire X4470 Server using Linux operating system. OpenMP was used as the tool for programming as it is suitable for SMP. The jobs submitted are independent to each other and can be scheduled and executed in any order. The composition of the individual job considered here is such that it has single instruction being executed in a nested loop with 10^8 iterations further divided in 10^4 and 10^4 iteration of each for loop. The simulations has been done on Sun Fire X4470 with each job found to be executing in 0.264 seconds on any individual node out of the available 32 nodes.

The experiments were designed to observe the change in total turnaround time of job(s) submitted to the system by varying the number of nodes available for execution with the provision of even varying the workload by changing the number of jobs demanding execution. Table 3.11 summarizes some of these results keeping the numbers of jobs fixed and increasing the numbers of nodes for the system with and without dynamic load balancing. Further, it has been assumed that the communication cost for the threads migrating from central dispatcher to the processing elements and between the processing elements is the same for all the jobs and processing elements and has been considered negligible as compared to the overall turnaround time. In addition the time taken by the scheduler in making the scheduling decisions is also very small and do not affect the turnaround time of the job significantly. For each set of experiments the arrival rate and service rate of jobs of nodes are considered to be the same.

Table 3.11-Comparitive Study of the System With and Without Load Balancing

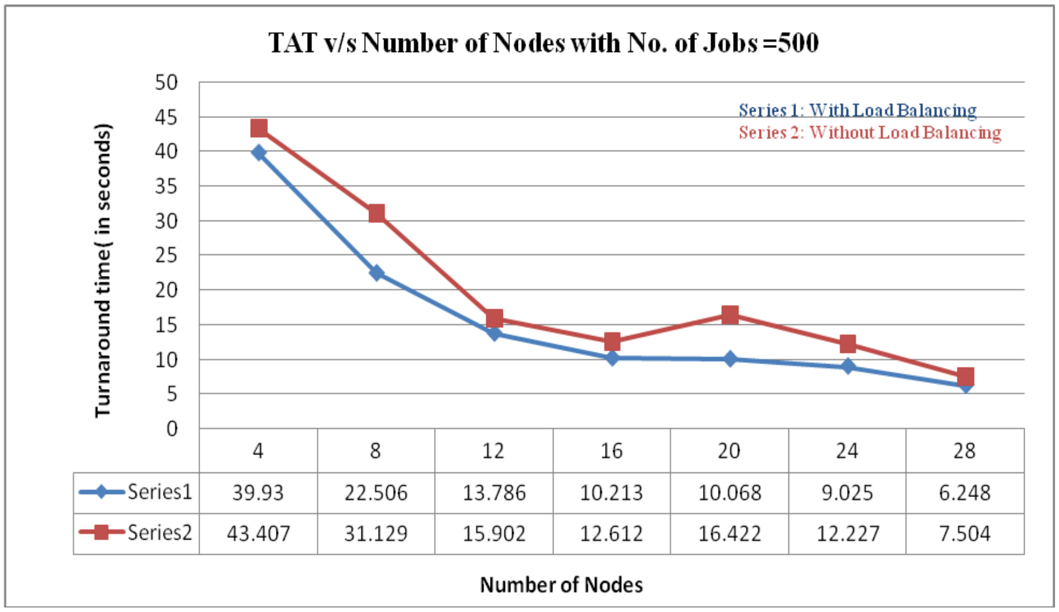
	Number of Jobs 100		**Number of Jobs 500**		**Number of Jobs 1000**	
Number of Nodes	**TAT With Load balancing (in Seconds)**	**TAT Without Load Balancing (in Seconds)**	**TAT With Load balancing (in seconds)**	**TAT Without Load Balancing (in Seconds**	**TAT With Load balancing (in seconds)**	**TAT Without Load Balancing (in Seconds**
4	8.123	8.732	39.93	43.407	86.22	86.502
8	4.817	7.688	22.506	31.129	45.282	62.659
12	3.017	3.622	13.786	15.902	27.285	30.948
16	2.199	3.618	10.213	12.612	20.763	21.932
20	2.532	5.116	10.868	16.422	18.929	28.115
24	1.604	3.358	9.025	12.227	17.449	24.237
28	1.576	1.879	6.248	7.504	12.68	13.775

Figures 3.2 to 3.4 illustrate the above results with the number of jobs being fixed to 100, 500 and 1000 respectively. Further, Figure 3.5 summarizes the results reported in Figures 3.2 – 3.4 using Dynamic Load Balancing (DLB). In the figures, X-axis represents the number of nodes in the system and Y-axis the turnaround time of jobs execution measured in seconds.



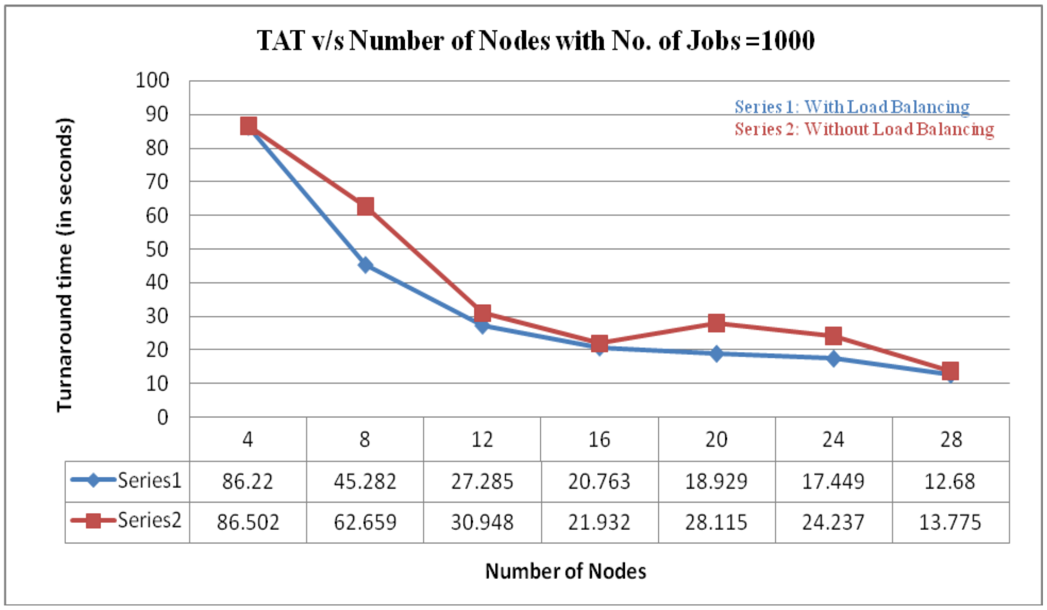
	4	8	12	16	20	24	28
Series1	8.123	4.817	3.017	2.199	2.032	1.604	1.576
Series2	8.732	7.688	3.622	3.618	5.116	3.358	1.879

Figure 3.2-TAT v/s Number of Nodes with Number of Jobs = 100



	4	8	12	16	20	24	28
Series1	39.93	22.506	13.786	10.213	10.068	9.025	6.248
Series2	43.407	31.129	15.902	12.612	16.422	12.227	7.504

Figure 3.3-TAT v/s Number of Nodes with Number of Jobs = 500



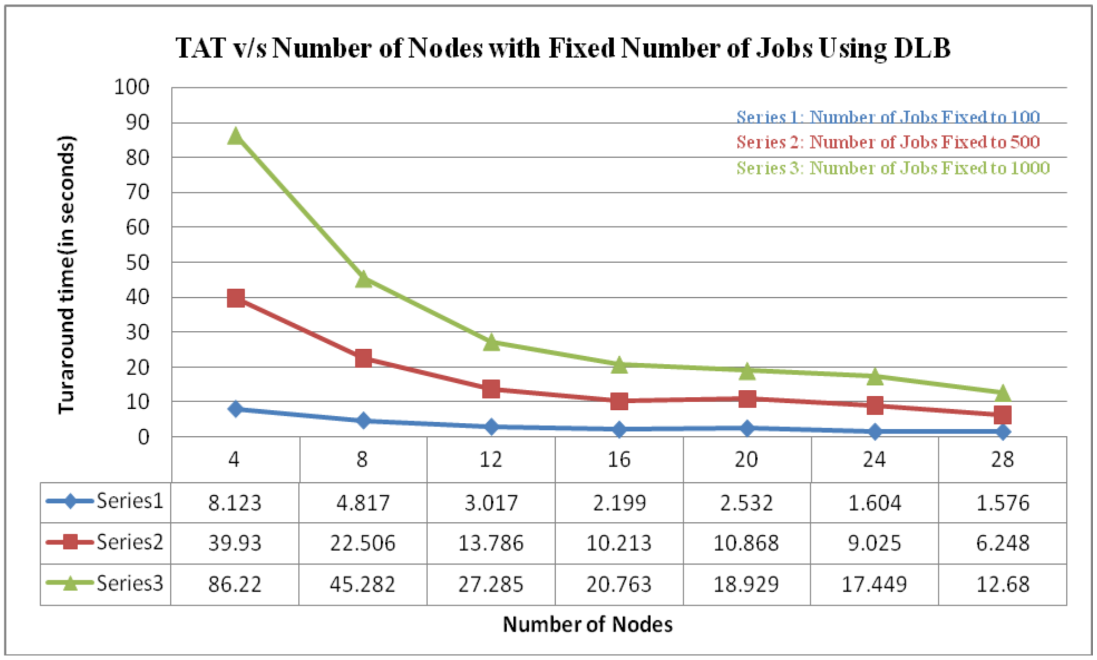
	4	8	12	16	20	24	28
Series1	86.22	45.282	27.285	20.763	18.929	17.449	12.68
Series2	86.502	62.659	30.948	21.932	28.115	24.237	13.775

Figure 3.4-TAT v/s Number of Nodes with Number of Jobs = 1000

Observations

- As the number of nodes increases, the turnaround time decreases for both the strategies involving load balancing and without load balancing as a general trend.
- The decrease in turnaround time for the load balanced system is smooth in comparison to the system without load balancing which is abrupt at many times.
- The system is observed to be fairly scalable with the increase in the number of jobs as well as computational resources. As the resources are added, the growing amount of load is handled efficiently.
- The model exhibits an even distribution of load leading to effective utilization of resources. Considering the arrival rate as same in both the scenarios of with and without load balancing, the former has even distribution of load.
- The adaptive nature of threshold parameters makes system robust to the growing amount of load. The nodes are idle only when there is no extra load on any node in the system.
- The model conforms to the Amdahl’s law [9]. This is evident from the observations reported in Figure 3.2 where the effect is more noticeable being the case of small workload. Here, an increase in the number of nodes does not

translate into an equal gain in performance, which becomes steady after a certain point, if the workload remains the same.



	4	8	12	16	20	24	28
Series1	8.123	4.817	3.017	2.199	2.532	1.604	1.576
Series2	39.93	22.506	13.786	10.213	10.868	9.025	6.248
Series3	86.22	45.282	27.285	20.763	18.929	17.449	12.68

Figure 3.5- TAT v/s Number of Nodes with Fixed Number of Jobs Using DLB

Figure 3.5 illustrates the summarizes the results of Figure 3.2, Figure 3.3, and Figure 3.4 with graphs presenting the results using Dynamic Load Balancing (DLB) used by the model. It can be observed that the system works well even if the workload is continuously increased with the efficient utilization of resources.

Chapter 4

Conclusion and Future Scope

It is always desired from a computing system that it should execute the job in the fastest possible way. Several measures were undertaken to achieve this goal leading to the use of parallel and distributed systems. Parallel computing systems are one of them that aim to minimize the task execution time. Distributed systems have led to multi-computer systems with various computing nodes communicating with each other, moving towards concurrent and cooperative engineering. Parallel computing has to deal with lot of issues which crop up while working with parallel code. These issues result in bottleneck and restrict the behavior of parallel program in attaining an aforesaid speedup given by Amdahl Gene. The most problematic issue that crops up is the distribution of workload in both the categories of parallel system viz. homogenous and heterogeneous systems. In homogenous system the processor with maximum load overpowers the working of system resulting in poor job response time whereas in heterogeneous system the slowest processor dominates the job response time. Therefore, in parallel systems, distribution of workload could result into some nodes to be heavily loaded and some nodes to be heavily under loaded. This situation demands an effective load balancing strategy to be in place which ensures a uniform distribution of load across the board. Load balancing mechanism is a software approach to redistribute system wide workload among the nodes of the system in order to reduce the mean job execution and hence the turnaround time. An efficient load balancing strategy must exhibit the features like creating little traffic overhead, low overhead for running the load balancing algorithm, must be fair enough so that heavily loaded node is balanced first with lightly loaded node, should utilize minimum CPU time to name a few. Load balancing is not only an issue in distributed memory system but also for shared memory system. In such systems the processing power of elements can only be utilized when efficient scheduling of jobs is done. Dynamic Load Balancing (DLB) is very useful as it helps the system to adapt to the changing workload and can be implemented using adaptive threshold values. This creates the scenario where no node is ever idle even if there is extra workload on any node in the system.

Load balancing on a parallel system has been established as an NP Complete problem. Therefore, heuristic approach is considered as the best way to deal as no exact solution can be established for such problems. The model proposed in this literature is a heuristic approach towards an optimized solution to the load balancing problem. This dissertation presents a model for the load balancing strategy for a multiprocessor system that aims to minimizing the turnaround time for a job(s) submitted for execution. The model is developed using Sun Fire X 4470 server as a test bed using OpenMP as a programming tool. Sun Fire X 4470 server is a multiprocessor system with four nodes each with eight cores. Since, each core can be treated as a node; it makes available thirty two nodes that can be programmed. OpenMp is used as a programming tool as it is suitable for the shared memory programming applications.

The proposed scheduler allocates the modules of the job(s) over the nodes in such a way that the desired objective of minimizing the turnaround time is met. The proposed model is based on centralized dynamic load balancing strategy using thresholds. The threshold values set helps in categorizing the nodes as heavily or lightly loaded nodes. The threshold values used here are adaptive in nature i.e. as the load on the system increases, threshold values are readjusted to suite the growing load on the system. The model works in such a way that the thresholds tend to converges the load towards the mean of the workload. These values becomes approximately equal when the load becomes evenly distributed depicting the balanced state of the system. The model is centralized in nature as it results in little traffic overhead. Moreover, the load redistribution process is fair as load is first readjusted between heavily loaded node and lightly loaded node through the use of max priority queue and min priority queue. The balancing process utilizes minimum CPU time as redistribution is only carried out when lightly loaded and heavily loaded nodes are reported.

Simulation study has been carried out for the model to evaluate its performance under various test conditions. It has been found that the model works well in ensuring an even distribution of the workload.

In the present work, it has been assumed that if average of the workload is distributed and hence executed by the processing elements, best results can be realized in terms of the turnaround time. Even better solution can be obtained if the model is made more realistic by considering other issues related to load balancing like communication cost and data locality.

References

1. http://en.wikipedia.org/wiki.
2. Tanenbaum, A., “Parallel and Distributed Systems”, PHI, 2002.
3. Barney, Blaise, "Introduction to Parallel Computing", Lawrence Livermore National Laboratory.
4. Foster, Ian, “Designing and Building Parallel Programs", Addison-Wesley Longman Publishing Co., 1995.
5. Steen, A.J. Van Der, Dongarra Jack, "Overview of Recent Supercomputers": http://www.phys.uu.nl/~steen/web03/contents.html.
6. www.oracle.com/us/products/.../sun-fire-x4470-server-077286.html
7. Chapman, Barbara, Jost, Gabriele, Pas, Ruud van der, Kuck, David J., "Using Open MP: Portable Shared Memory Parallel Programming", The MIT Press, 2008.
8. Gropp, William, Lusk, Ewing, Thakur, Rajeev, “Using MPI-2”, MIT Press, 1999.
9. Amdahl, Gene, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", AFIPS Joint Computer Conference, 1967, pp 1-4.
10. Youran, Lan, "A Dynamic Load Balancing Mechanism for Distributed Systems” J. of Computer Science & Technology, May 1996, Vol. 11, No. 3, pp 1-13.
11. Dubrovsky, Alexander, Friedman, Roy, Schuster, Assaf, “Load Balancing in a Distributed Shared Memory System”, IBM, 2005.
12. Willebeek, Marc, Reeves, Anthony P.,” Strategies for Dynamic Load Balancing on Highly Parallel Computers”, IEEE Transaction on Parallel systems Software, 1999, pp 51-59.
13. Sakae, Yoshiaki et al, “Preliminary Evaluation of Dynamic Load Balancing Using Loop Re-partitioning on Omni/SCASH”, Third IEEE International Symposium on Cluster Computing and the Grid, May 12-May 15 2000.
14. Drosinos, Nikolaos, Koziris, Nectarios,”Load Balancing Hybrid Programming Models for SMP Clusters and Fully Permutable Loops”, Supercomputing IEEE 2000 Conference, 04-10 Nov. 2000, pp 10-10.

15. Rudolph, Larry et al, “A simple load balancing scheme for task allocation in parallel machines”, Proceedings of the third annual ACM symposium on Parallel algorithms and architectures, p.237-245, July 21-24, 1991
16. Keckler, W. Stephen, “The Importance of Locality in. Scheduling and Load Balancing for. Multiprocessors”, IEEE Transactions on Software Engineering, May 1986, pp 675-680.
17. Michael, Pinado, “Scheduling: Theory, Algorithms, and Systems”, http://www.amazon.com/Scheduling-Theory-Algorithms-Systems-2nd/dp/0130281387
18. Baker, K.R., "Introduction to Sequencing and Scheduling”, John Wiley, 1974.
19. Tanenbaum, A., “Operating Systems”, PHI, 2004.
20. http://www.netlib.org/utk/lsi/pcwLSI/text/node248.html
21. Brucker, P., “Scheduling Algorithms”, Fifth edition, Springer, Heidelberg, 2007.