CHAPTER 1
CONCURRENT PROGRAMMING - PROBLEMS AND OPPURTUNITIES

1.1 INTRODUCTION

'Do one thing at a time' has been the guiding philosophy behind every programming effort on sequential machines. Sequential Computing was thus a process of reducing the solution of a given problem into a sequence of steps that can be executed one after the other. A typical sequential machine architecture, built around a single processor unit, permits only one operation to be performed at a given time. Programming Languages serve as tools to express these steps as a sequence of language constructs or statements leading to a solution. A set of statements leading to the solution of a given problem is called a program. The quest for increased efficiency in the utilisation of the processor capabilities led to the introduction of specific purpose I/O Processors to control the I/O devices. The use of I/O processors freed the main processor from the mundane, lengthy and device dependent I/O operations. This technique brings in two new problems to solve. (Evans 1982)

a) The contention for the memory access

b) The communication between the Main and I/O Processors

The first is solved by using a switch through which the main processor and the I/O processors are connected to the memory. The communication between the main processor and the I/O processor occurs twice
i) The main processor initiates the I/O Processor

ii) The I/O processor reports to the main processor upon completion of the job.

The first communication can be implemented as a special instruction executed by the main processor. The main processor continues the execution of the program after initiating the I/O processor. The second communication is typically implemented as interrupts which enable the main processor to stop the execution of the program, handle the interrupt and resume the execution of the program as though nothing has happened. The interrupt mechanism introduces a certain degree of non-determinism hitherto unknown in computing systems. The non-determinism stems from the fact that one is not sure about the time of termination of an I/O job and the issue of an interrupt signal by the I/O processor. The main processor however, services the interrupt only after the current instruction is completely executed.

Eventhough the main process and the I/O jobs run simultaneosly, the I/O processor is very much dependent on the main processor for its operation. The onus of shielding these intricacies from the user falls on the operating system designer.

Extending this philosophy further leads to a more powerful concept of parallel processing wherein several processors execute user programs concurrently. Each processor can be as capable as the other and they function independently.

'Think Parallel' is the notion that would perhaps dominate the future course of computing. An attempt to solve problems more efficiently and more naturally led to the
development of novel computer architectures and programming methodologies. Concurrent programming, an attempt to do more than a single operation at a time is fast emerging as a viable answer to the evergrowing demands of the complex world. Even though Concurrent Programming is conceptually elegant, it poses several problems at the implementation phase. Some of the pertinent issues are (Muthukrishnan 1989):

- to build a reliable architecture using multiple processor units

- to provide programming languages that allow spawning of concurrent processes and the programmer to control the interprocessor communications.

- to design efficient and reliable operating systems for parallel processors.

Even though several processes are being executed concurrently, each individual process is inherently sequential in nature. Concurrent processing is simple when the processes do not interact with each other. The control initiates the parallel processes and upon termination of all processes the operation is said to be complete. The situation is more challenging when the processes have to interact with each other. In this case, whether a given operation of a program was executed before, after or simultaneously with a given operation of a parallel process assumes a great significance. Parallel programs are thus more prone to runtime time-dependant errors.

Parallelism can be exploited at two levels

a) the statement level

b) the program level
1.1.1 Statement Level Parallelism

Precedence graphs are popularly used to indicate the order of execution of statements within a program. Bernstein’s conditions are used to support data dependencies. The popular constructs provided in the programming languages to facilitate concurrent execution of statements are discussed below (Peterson 1983a).

a) Fork and Join Constructs

These constructs were the first specifications of concurrency. The fork L instruction spawns a process starting with the statement labelled L. This leads to two processes, one starting with the statement labelled L and the other starting with the statement immediately after the ‘fork’ statement.

The join <number> construct is used to signal the merging of a specified number of processes. A go-to is used to transfer control to the join statement wherever appropriate.

b) The Concurrent Statement

A higher level language construct
parbegin S1;S2;......;Sn parend
indicates the parallel execution of the single statements S1..Sn. The next statement in the program after the concurrent statement is executed only after the completion of all the statements S1..Sn. Concurrent statements can be nested.

A concurrent statement, is weaker than the fork-join constructs and cannot implement all precedence graphs. The concurrent statement used along with other
mechanisms such as semaphores matches the power of precedence graphs to a satisfactory extent.

1.1.2 Program Level Parallelism

Program level parallelism is more difficult to implement owing to the mutual interactions. In real life problems the parallel processes have to cooperate with each other to achieve a common goal. Several techniques outlined in Section 1.3 illustrate the typical problems encountered in implementing concurrent process execution.

1.2 HISTORICAL BACKGROUND

Concurrent programming has a long history with several pioneers contributing towards the furthering of this challenging field that has ushered a new era of computing. A chronological trace of landmark achievements in the field of concurrent programming is given below (Muthukrishnan 1989).

<table>
<thead>
<tr>
<th>Year</th>
<th>Name</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1962</td>
<td>Carl Adam Petri</td>
<td>Petri Net Theory</td>
</tr>
<tr>
<td>1963</td>
<td>Conway M. E</td>
<td>Fork and Join Constructs, Coroutines - A synchronisation concept</td>
</tr>
<tr>
<td>1965</td>
<td>E W Dijkstra</td>
<td>Concurrent Statement, Critical sections and Semaphores, Dekker's solution for two-process mutual exclusion problem, Solution for n process mutual-exclusion problem</td>
</tr>
<tr>
<td>Year</td>
<td>Author(s)</td>
<td>Title</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>1968</td>
<td>Dahl et al</td>
<td>Simula 67 - Class Concept</td>
</tr>
<tr>
<td>1968</td>
<td>E W Dijkstra</td>
<td>Discussion on Semaphores</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The THE System</td>
</tr>
<tr>
<td>1970</td>
<td>Brinch Hansen</td>
<td>Process concept and implementation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inter Process Communication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RC 400 System</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Messages for synchronisation</td>
</tr>
<tr>
<td>1971</td>
<td>E W Dijkstra</td>
<td>Discussion on Semaphores</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Secretary Concept</td>
</tr>
<tr>
<td>1971</td>
<td>Parnas</td>
<td>Abstract data types</td>
</tr>
<tr>
<td>1972</td>
<td>Brinch Hansen</td>
<td>Comparision of semaphore, critical region, conditional region</td>
</tr>
<tr>
<td>1972</td>
<td>C A R Hoare</td>
<td>Critical region concept</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conditional region concept</td>
</tr>
<tr>
<td>1973</td>
<td>Agerwala &amp; Flynn</td>
<td>A problem that cannot be solved by P and V operations</td>
</tr>
<tr>
<td>1973</td>
<td>Brinch Hansen</td>
<td>Monitor Concept</td>
</tr>
</tbody>
</table>
1973 | D.E. Knuth | Illustration of Coroutines
---|---|---
1973 | Horning & Randall | Formalisation of Process Concept
1974 | C A R Hoare | Complete description of Monitor
1975 | Brinch Hansen | Concurrent Pascal
1976 | Owicki & Gries | Correctness Proofs for parallel programs
1976 | Schmid | Efficient implementation of Conditional critical region
1977 | Carl Hewitt | Actor Model for concurrency
1977 | Kessels | Extended Monitor to allow automatic signalling
1977 | L. Lamport | Proof system for concurrent programs
1977 | N. Wirth | Modula
<table>
<thead>
<tr>
<th>Year</th>
<th>Author(s)</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1978</td>
<td>Brinch Hansen</td>
<td>Distributed Processes</td>
</tr>
<tr>
<td>1978</td>
<td>C A R Hoare</td>
<td>Communicating Sequential Processes</td>
</tr>
<tr>
<td>1978</td>
<td>Holt et al</td>
<td>CSP/k based on PL/1</td>
</tr>
<tr>
<td>1979</td>
<td>Cheriton</td>
<td>Thoth real time operating system</td>
</tr>
<tr>
<td>1979</td>
<td>Jean Ichbiah et al</td>
<td>Ada</td>
</tr>
<tr>
<td>1980</td>
<td>L. Lamport</td>
<td>Temporal logic of programs</td>
</tr>
<tr>
<td>1981</td>
<td>Peterson</td>
<td>Simpler solutions to two-process and n-process mutual exclusion problems.</td>
</tr>
<tr>
<td>1983</td>
<td>Clark and Gregory</td>
<td>PARLOG - PARllel LOGic Programming Language</td>
</tr>
<tr>
<td>1986</td>
<td>S Kostafian</td>
<td>Formal notion of an object</td>
</tr>
<tr>
<td></td>
<td>G Copeland</td>
<td></td>
</tr>
<tr>
<td>1986</td>
<td>J R Newport</td>
<td>Occam Language</td>
</tr>
<tr>
<td>1987</td>
<td>Bennet J K</td>
<td>Distributed Smalltalk</td>
</tr>
</tbody>
</table>

The subsequent years witnessed a rapid implementation of the key contributions.
Researchers across the globe have made extensions to several concepts resulting in a massive increase in the volume of contributions. Several new architectures were developed and reported. An annotated bibliography of various papers published in this area appears in literature (Zedan 1989).

1.3 CONCURRENT PROGRAMMING - PRIMITIVES

Concurrent programs need to share variables and/or resources which necessitate synchronisation. A detailed discussion of the primitives for program level concurrency are discussed below (Peterson 1983b).

1.3.1 Processes

A process is the basic unit of concurrency defined as a sequence of operations. Several processes running simultaneously towards a common goal form a set of cooperating sequential processes.

The earliest form of several processes running in parallel is the use of Coroutines. Introduced by Conway, coroutines obliterate the distinction between a main program and sub-programs. Thus main program can call sub-programs and the sub-programs can initiate a call to the main program. However, this form of concurrency is limited only to the transfer of control and thus is a very restricted version of parallel programming.

Concurrent processes are pictorially represented by Process Graphs. Process creation, execution and termination aspects present a host of problems to be handled by the operating system designer. Process creation and maintenance incurs a lot of system
overhead. Thread is a concept introduced to reduce this overhead thereby increasing the overall performance. Threads are a set of light-weight concurrent processes sharing a common set of data structures.

1.3.2 Interprocess Communication

Communication among several processes running concurrently is the crux of concurrent program implementation. A set of processes working simultaneously towards a common objective need effective and efficient communication facilities. Interprocess communication, as this issue is popularly described is implemented using three distinct techniques.

a) Shared memory systems

b) Message passing systems

c) Remote Procedure Call

1.3.2.1 Shared Memory Systems

Shared memory systems warrant the communicating processes to access a portion of physical memory. However, this type of implementation directly leads to critical regions to be discussed later. Another disadvantage of using shared variables is that the process accessing the shared variable must be close to the location of the variable, lest the latency in accessing the variable would be a bottleneck.

1.3.2.2 Message Passing Systems

A message system usually provides two communication primitives send and receive. A communication link has to be established between any two communicating
processes. The use of hardware links provides faster communication but an inflexible communication network. Providing logical or software links which can be mapped onto a basic hardware interconnection scheme gives the user a great deal of flexibility at a premium of communication overhead. Logical communication links are implemented in any one of the following ways (Peterson 1983a).

a) Direct Communication

Symmetric direct communication scheme requires the sender process to explicitly specify the name of the receiver process and the receiver process has to explicitly specify the name of the process from which it wants to receive a message. The logical link so established is bi-directional. The communication primitives provided are

- send \( (P, \text{message}) \) \{ Send to process \( P \) \}
- receive \( (Q, \text{message}) \) \{ Receive from process \( Q \) \}

Asymmetric direct communication permits a process to receive from any one of the other parallel processes. The sender process name is registered after the communication has taken place. The receive primitive will be modified as

- receive \( (\text{id}, \text{message}) \) \{ 'id' is replaced with the name of the process with which communication has taken place \}

The major disadvantage of direct communication is that to change the name of a process all the communication instructions involving this process have to be traced and changed accordingly, which thwarts the provision of separate compilation facilities.

b) Indirect Communication

In this strategy the processes communicate through the use of mailboxes or ports. The communication primitives are
Two processes can communicate if and only if they have a shared mailbox. Each mailbox has a unique identity. Between them, any two communicating processes can have many mailboxes each with a unique label. The communication link thus established can be either uni-directional or bi-directional depending on the implementation scheme of the mailboxes.

If the mailbox is declared within a process it becomes a private property of that process and no other process can receive messages from this mailbox. However, if the operating system is providing the mailbox then it must provide the facilities for

i) creation of a new mailbox

ii) sending and receiving messages through the mailbox

iii) destroying a mailbox

The process that creates a mailbox is its owner by default. However, the ownership may be relinquished by using the appropriate system calls. The child processes of the process that has created a mailbox acquire access rights to the mailbox.

The communication link established between two processes may have a capacity to store a number of queued messages. The capacity of the link could be

a) zero : The sender waits until the message is received

b) Bounded : The sender is delayed if the queue is full

c) Unbounded : The sender is never delayed
In case of links with a capacity to queue the messages there is a need for an explicit acknowledgement from the receiver to the sender.

A scheme that sends a reference to the messages rather than the original messages greatly helps the recovery of lost messages. Either the operating system or the sender process can detect and retransmit lost or scrambled messages. In some implementation schemes the operating system detects the loss of a message and indicates to the sender leaving the action to the sender’s discretion.

A common exception occurs when a receiver process terminates without processing the message. The sender waits eternally for an acknowledgement. In this case the sender process is notified by the operating system that the receiver is no more.

The communication can be made reliable by having the receiver transmitting an acknowledgement either after receiving a request from the sender or along with the reply.

1.3.2.3 Remote Procedure Call

A Remote Procedure Call mechanism can be implemented between a pair of concurrent processes. The procedure on the receiver is treated as a remote procedure and the sender marshalls the parameters in the form of a message. The parameter passing mechanisms and the message formats are to be standardised. More importantly the failure semantics have to be defined carefully.

It is to be noted that in a particular implementation all the above modes of communication can co-exist.
1.3.3 Critical Region

Concurrent processes co-operate with each other either by sharing data and/or by message passing. In the latter case, the programming languages provide communication primitives and it is up to the programmer to ensure safe communication between processes. The communication implementation protocols need not resolve deadlock problems as long as the processes are defined correctly. Message passing systems may also make use of network protocols to ensure mutual exclusion. A detailed discussion of these protocols is beyond the scope of this work.

The problem is not so simple when processes co-operate through shared data or shared resources. As only one process may be granted access at any given time, the shared data items and resources are termed as critical regions in process execution. There is now a need to ensure mutual exclusion of the processes in the critical regions. Any algorithm devised for this purpose must possess at least the following features (Pratt 1983).

a) It must ensure mutual exclusion

b) If the critical section is empty and there are several processes waiting to enter, one of them must be granted access in a finite time.

c) The mutual exclusion protocol is completely independent of any other system interactions

d) No process is privileged.

In case of shared data items the mutual dependence relations among them (integrity constraints) must also be considered to grant access to a process.
1.4 CONCURRENT PROGRAMMING LANGUAGES

A severe software crisis shook the computing world during 1970-80. This was a black decade that spelt doom to many programming languages existing until then. The next decade however witnessed several programming paradigms vying for the top place. The programming paradigms in the arena are given below (Muthukrishnan 1989).

a) Block Structured Paradigm
b) Object-Oriented Paradigm
c) Distributed Paradigm
d) Functional Programming Paradigm
e) Logic Programming Paradigm
f) Database Paradigm

The language paradigms discussed above are bifurcated into two distinct groups.

i) Imperative Paradigms

They view a program as a sequence of commands that transform a well defined set of inputs to a set of desired outputs. The block structured, Object-Oriented and distributed paradigms belong to this category.

ii) Declarative Paradigms

These paradigms emphasize what is to be computed rather than how it is computed. They support inherent concurrency and lazy evaluation which permits the need
based evaluation of sub-computations. The Functional Paradigm, Logic Paradigm and Database Paradigm are usually categorised as declarative paradigms.

Some of the popular block-structured concurrent programming languages are MODULA, OCCAM and ADA. ACORE is a concurrent programming language based on the Object-Oriented programming paradigm. CSP and DP are popular models for distributed languages and systems. OCCAM, PLITS are languages based on the distributed paradigm. PARLOG is a concurrent programming extension of the popular Logic Programming Language PROLOG. It is to be noted that a given language can be classified under more than a single paradigm of programming.

1.5 VERIFICATION OF PARALLEL PROGRAMS

A concern for checking computer programs for correctness was felt as early as 1961. By checking for correctness of a program we attempt to establish that the program upon execution leads to desired set of outputs for any given set of inputs. The need for verification of programs increased with the increase in the complexity of the problems on hand. Turing Machine is an abstract computing model used to verify correctness of sequential programs. Other techniques evolved to verify sequential programs are axiomatic approach, structured programming concepts etc. The programmer thus began to keep in view both verifiability as well as efficiency of the programs.

Concurrent programs, owing to the inherent unpredictable order of performing actions, are more difficult to be verified formally. A number of methods were suggested to verify parallel programs. Some of them are given below.

* Axiomatic Approach (Owicki 1975)
Any proof for correctness of a program is largely dependent on certain assumptions about the implementation, scheduling and execution of the program on hand. These assumptions are usually postulated as axioms describing the programming language and the system.

A correct program satisfies the following properties.

a) Safety Properties

Safety properties establish that nothing bad can happen by executing the given program. These properties provide proof for partial correctness of the program using invariant assertions and pre- and post-conditions. Invariant assertions are logical properties that must be satisfied by the system at all times during execution. Pre- and post-conditions for a given statement of the program relate the state of the program before the execution of that statement, to the state of the program after execution of that statement respectively.

b) Liveness Properties

Liveness Properties establish that the required good things happen upon the execution of the program. These properties confirm the termination of the program and recurrent properties in non-terminating programs (Operating Systems).
1.6 HARDWARE

Effective implementation of concurrent programming techniques needs revolutionary changes in the underlying machine architecture. The traditional sequential architectures are giving way to more powerful parallel architectures. The operations of every digital computer can be viewed as an interaction between Instruction Stream (a sequence of instructions) and Data Stream (a collection of data items namely inputs, intermediate results and outputs). The architecture of the machine decides the mode of interaction between these two streams. There are several ways proposed by Flynn, Feng and Wolfgang Handler to classify computer architectures. The most popular classification is due to Flynn and is mentioned below (Hwang 1986).

SISD : Single Instruction stream - Single Data stream

SIMD : Single Instruction stream - Multiple Data stream

MIMD : Multiple Instruction streams - Multiple Data streams

MISD : Multiple Instruction streams - Single Data stream

(This architecture is as rare as a hare’s horn)

Parallelism stems naturally from the two architectures SIMD and MIMD. Pipeline Architectures and Vector Processing are two popular techniques employed to support concurrency of operations in traditional SISD or Uniprocessor systems.

1.6.1 Multiprocessor Architectures

Multiprocessor systems have been in vogue due to two basic reasons

a) Increased system throughput

b) Fault-tolerance
Multiprocessor systems for concurrent execution of programs are of two types (Hockney 1981)

a) **Loosely Coupled Systems**

Several Computer Modules (a computer module consists of processor + memory + I/O + arbiter) are connected through a message transfer system.

Example: Carnegie Mellon University project Cm*

b) **Tightly Coupled Systems**

Several processor elements, shared memory modules, shared I/O facilities are all interconnected through one of the following methods (Gottlieb 1983)

i) Time shared Buses (Ex: Unibus, Multibus etc.)

ii) Crossbar Switch

iii) Multiport Memories.

These systems warrant efficient concurrent operating systems. Some popular multiprocessor systems are C.mmp (dismantled), Cm*, S-1, IBM 370/168MP, Univac 1100/8x, Univac 1100/9x, Tandem/16, Cray X-MP, Denelcor HEP, IBM 3081, IBM 3084, Cyber 170, Honeywell 60/66, PDP -10 etc.

### 1.6.2 Distributed Systems

Loosely coupled multi-processor systems wherein the individual computing modules are situated at distances ranging from a few metres to several thousand kilometers are called distributed systems. The message passing mechanisms in such systems is dependent on another fully developed domain called Computer Networking. A thorough
discussion on computer networks is beyond the scope of this work. Tightly-Coupled Systems are also known as the Processor Pool model of distributed systems (Colouris 1990), (Lelann 1977).

1.7 OPERATING SYSTEMS

The design and implementation of Operating Systems for the hardware platforms discussed above is proving to be a major challenge. The languages supporting concurrency also place a great demand on the underlying operating system. Some of the multiprocessor systems mentioned earlier support their own specific operating system. Popular operating systems such as UNIX do provide mechanisms for inter-process communication and synchronisation. In fact, UNIX can be readily ported onto multiprocessor architectures.

There are three emerging categories of Operating Systems which provide support for concurrency. The key features of these three categories for a N processor system are compared in Table 1.1 (Tanenbaum 1992).

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Network OS</th>
<th>Distributed OS</th>
<th>Multi-Processor OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vitual Uniprocessor</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Same Operating System on each</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>processor</td>
<td>N</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>No. of Copies of OS Communication</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>N/W Protocols req.</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Single run Queue</td>
<td>Usually NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Semantics for file Sharing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sharing</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distributed Operating Systems provide location transparency, migration transparency and replication transparency to the programmer. There has been an ever growing interest in the design of Distributed Operating Systems due to the promise they hold out to the concurrent programmer. However, the following problems are proving to be challenging and providing ample scope for intense research activity in the area (Tanenbaum 1992).

* Mutual Exclusion and Synchronisation
* Deadlocks
* Termination Detection
* Ordering of Events and Global State Computation for the purpose of verification (Lamport 1978)
* File Systems
* Fault Tolerance
* Elective Algorithms
* Load Balancing
* Security and Protection
* Performance Measurement

1.8 MOTIVATION, OBJECTIVES AND SCOPE OF THE WORK

One of the primary objectives of a typical Distributed Operating System is to ensure that atmost one process is using a share resources at any given time. This is the Mutual Exclusion problem. As mentioned in section 1.2, solutions for this problem are
seen as early as 1963. The solutions have attained more maturity over the past three decades and culminated in language level support for mutual exclusion and synchronisation. Such an abstraction mechanism at the language level has greatly simplified the task of the concurrent programmer.

An attempt to reduce the serialisation in these solutions provided the motivation to propose a new structuring concept called the concurrent monitor for distributed operating systems. The objectives of this thesis are:

- to propose a structuring concept to overcome the drawbacks discussed above while ensuring mutual exclusion

- to implement the model in a multi-processor environment

- to examine issues pertaining to load-balancing aspects.

The above work is detailed in the remaining six chapters. **Chapter 2** is a survey of various solutions proposed in the literature to solve the mutual exclusion problem.

**Chapter 3** is the description of the concurrent monitor. The salient futures of its implementation on both tightly coupled and loosely coupled machine architectures are mentioned. A new shared memory model is proposed. Solution for popular synchronisation problems using concurrent monitor are provided. A theoretical comparison between the Ada model and the concurrent monitor is presented to illustrate the performance edge of the latter.
The memory model proposed is compared with standard organizations such as monolithic, parallel banks and multiport memories in Chapter 4. The key issue has been to generate a trace with the distinction between the read and write requests.

Chapter 5 outlines the adaptation of concurrent monitor model to message passing systems operating on the principle of token ring architectures. An algorithm for achieving fault-tolerance in such a system is discussed.

The implementation of concurrent monitor with distributed control permits one to examine a variety of load balancing strategies. The results of simulation studies of some popular techniques are presented in Chapter 6. A load balancing algorithm is modified to considerably reduce the execution time by using additional storage space.

Chapter 7 reviews the objectives. The key contributions made are outlined. It also discusses future extensions to this work.