Sheffield Hallam University

Defining the selective mechanism of problem solving in a distributed system.

MASHHADI, Tahereh Yaghoobi.

Available from Sheffield Hallam University Research Archive (SHURA) at:

http://shura.shu.ac.uk/20022/

This document is the author deposited version. You are advised to consult the publisher's version if you wish to cite from it.

Published version

MASHHADI, Tahereh Yaghoobi. (2001). Defining the selective mechanism of problem solving in a distributed system. Doctoral, Sheffield Hallam University (United Kingdom)..

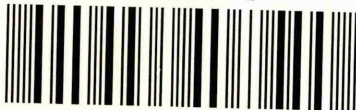
Copyright and re-use policy

See http://shura.shu.ac.uk/information.html

Sheffield Hallam University Research Archive
http://shura.shu.ac.uk

CITY CAMPUS, POND STREET,
SHEFFIELD S1 1WB.

101 667 486 4



Fines [illegible] charged at 50p per hour

REFERENCE

ProQuest Number: 10697329

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 10697329

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Defining the Selective Mechanism of Problem Solving in a Distributed System

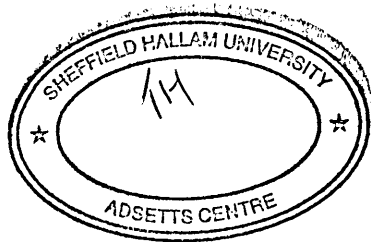
Tahereh Yaghoobi Mashhadi

A thesis submitted in partial fulfilment of the requirements of

Sheffield Hallam University

for the degree of Doctor of Philosophy

March 2001


SHEFFIELD HALLAM UNIVERSITY
ADSETTS CENTRE

Abstract

Distribution and parallelism are historically important approaches for the implementation of artificial intelligence systems. Research in distributed problem solving considers the approach of solving a particular problem by sharing the problem across a number of cooperatively acting processing agents. Communicating problem solvers can cooperate by exchanging partial solutions to converge on global results.

The purpose of this research programme is to make a contribution to the field of Artificial Intelligence by developing a knowledge representation language. The project has attempted to create a computational model using an underlying theory of cognition to address the problem of finding clusters of relevant problem solving agents to provide appropriate partial solutions, which when put together provide the overall solution for a given complex problem. To prove the validity of this approach to problem solving, a model of a distributed production system has been created.

A model of a supporting parallel architecture for the proposed distributed production problem solving system (DPSS) is described, along with the mechanism for inference processing. The architecture should offer sufficient computing power to cope with the larger search space required by the knowledge representation, and the required faster methods of processing. The inference engine mechanism, which is a combination of task sharing and result sharing perspectives, is distinguished into three phases of initialising, clustering and integrating. Based on a fitness measure derived to balance the communication and computation for the clusters, new clusters are assembled using genetic operators. The algorithm is also guided by the knowledge expert. A cost model for fitness values has been used, parameterised by computation ration and communication performance.

Following the establishment of this knowledge representation scheme and identification of a supporting parallel architecture, a simulation of the array of PEs has been developed to emulate the behaviour of such a system. The thesis reports on findings from a series of tests used to assess its potential gains.

The performance of the DPSS has been evaluated to verify the validity of this approach by measuring the gain in speed of execution in a parallel environment as compared with serial processing. The evaluation of test results shows the validity of the proposed approach in constructing large knowledge based systems.

Acknowledgements

I would like to thank my supervisor R. A. Steele for his help and guidance throughout the period of this work. I am also grateful to the staff of the School of Computing and Management Science for their friendship and help.

I would also like to express my deepest appreciation to my dear parents and husband for their patient and encouragement.

My special thanks to my dear brother, Ali-Akbar, for his valuable supports and encouragement during the period of my education.

I am also indebted to my two little sons for being always source of delight and hope, and others from whom I have gained knowledge.

Contents

1. Introduction **1**

1.1 Objective of the thesis 7

1.2 Outline of the thesis 7

2. Parallelism in Production Systems **9**

2.1 Introduction 9

2.2 The architecture of production systems 11

2.3 Parallel architectures 14

2.3.1 DADO architecture 16

2.4 Parallel rule matching 16

2.5 Parallel production systems 18

2.5.1 Rule interdependencies 20

2.5.2 Compatibility problem 20

2.5.3 Convergence problem 22

2.5.4 Rule distribution 23

2.5.5 Parallel rule firing models 25

2.6 Distributed production systems 27

2.7 Multiagent production systems........32
2.8 Appropriate parallel architecture........33
2.9 Summary........35
3. Genetic Algorithms........36
3.1 Introduction........36
3.2 Advantages and disadvantages of genetic algorithms........37
3.3 Traditional genetic algorithm........38
3.3.1 Parameters of the GA........41
3.3.2 Objective and fitness functions........44
3.3.3 Encoding........45
3.3.4 Selection........47
3.3.5 Crossover and mutation operators........50
3.4. Hybrid genetic algorithms........52
3.5 Parallel genetic algorithms........52
3.5.1 Global model........53
3.5.2 Island model........53
3.5.3 Fine grained model........55
3.6 Applications of genetic algorithms........56
3.6.1 Task allocation........56
3.6.2 Clustering........58
3.6.3 Automated theorem proving........59
3.7 Summary........61
4. A Distributed Problem Solving System........63
4.1 Introduction........63
4.2 Goals of the system........65
4.3 Architecture of the system........66

4.4 Problem solving methodology 68

4.4.1 Negotiation strategy 73

4.5 Perspectives on the system 74

4.6 The knowledge representation paradigm 76

4.7 Structure of the knowledge base 78

4.8 Behaviour of Problem Solving agents 81

4.9 Implementation of genetic algorithm 84

4.9.1 Chromosome representation 86

4.9.2 Fitness evaluation 87

4.9.3 Reproduction scheme 88

4.9.4 Genetic operators 89

4.9.5. Population size and operator rates 89

4.9.6. Hill climbing 89

4.10 The overall behaviour of the system 90

4.11 Design of the system 93

4.11.1 The structure chart 93

4.11.2 The knowledge diagram 95

4.12 Summary 97

5. Testing and Evaluation 99

5.1 Introduction 99

5.2 Structure of the simulation program 100

5.3 Verification 102

5.3.1 Verification of the knowledge base 102

5.4 Validation 106

5.4.1 Validation methodology 107

5.4.2 Validation criteria 107

5.4.2.1 Calculating speedup in terms of production cycles 110
5.4.2.2 Calculating speedup in terms of CPU times 111
5.5 Evaluation of the results 113
5.5.1 Comparison against known results 114
5.5.2 Performance improvement 114
5.6 Summary 122
6. Conclusion 124
6.1 Introduction 124
6.2 Restatement of the project aims 124
6.3 Summary of the thesis 125
6.4 Discussion of the results 128
6.5 Summary 131
6.6 Further work 132
References 134
Appendix A: The Simulation Program 1-A
Appendix B: The KB 26-B
Appendix C: Test Runs 36-C

List of Figures

Figure 3.1. The outline of a genetic algorithm. 41

Figure 3.2. Binary encoding. 45

Figure 3.3. Permutation encoding. 46

Figure 3.4. Value encoding. 46

Figure 4.1. Schematic representation of the system. 66

Figure 4.2. Three levels of the distributed problem solving system. 75

Figure 4.3. The hierarchical model of the KB 79

Figure 4.4. The computational model of a problem solving cycle. 83

Figure 4.5. The computational model of the HPGA. 86

Figure 4.6. An instantiation of chromosome representation. 87

Figure 4.7. The computation model of the DPSS. 92

Figure 4.8. The modified structure chart describing the simulation program. 94

Figure 4.9. The knowledge diagram 95

Figure 4.10. Graphical representation of a simple rule. 96

Figure 4.11. Graphical representation of a conjunctive rule. 96

Figure 4.12. Graphical representation of a disjunctive rule. 97

Figure 5.1. Performance of the DPSS. 116

Figure 5.2. Processing times against number of PEs. 117

Figure 5.3. Comparison between sequential and parallel processing times............. 118

Figure 5.4. Variances in parallel processing time. 118

Figure 5.5. Comparison between sequential and parallel processing times............. 119

Figure 5.6. Comparison between execution times for different phases of the inference methodology. 120

Figure 5.7. Speedups against the number of production rules. 121

List of Tables

Table 5.1. Representation of rule-sets. 106

Table 5.2. Simulation results using production cycles. 115

Table 5.3. Simulation results using execution times. 116

1

Introduction

Distribution and parallelism are important aspects of the history of Artificial Intelligence (AI). Indeed the history of AI goes back to when the question 'Can machines think?' was first posed. This question put forward the idea that a specific form of behaviour, which is associated with human beings, may also be associated with machines. At that time (the early 1950s), computation was considered as a form of machine behaviour [Boden 1990]. The idea of machines behaving like humans released the question of whether computational capability of digital computers could expand enough to enable them to exhibit intelligent behaviour. In other words, whether computers could act more than just number crunchiers.

Two different visions, each viewing the computer from different perspectives, began to develop. The first of these visions is about creating artificial intelligence by modelling the mind's symbolic representation of the world, and is called strong AI [Searle 1990]. It views the computer as a system for manipulating mental symbols, through which intelligent tasks can be implemented. This idea takes problem solving as its paradigm of intelligence, and utilises logic. According to this vision the computer is not only a tool in the study of the mind, but also an appropriately programmed computer, is in fact, a mind. This means that computers that are given the right

programs can be said to understand and have other cognitive states. The main idea here is that intelligent tasks can be implemented by reasoning processes operating on symbolic internal models.

The second vision is about creating artificial intelligence by modelling the brain rather than the mind's symbolic representation of the world, and is called weak AI [Searle 1990]. It considers the computer as a tool for modelling the brain as the only truly intelligent system. It uses a computer as a tool for simulating the interactions of neurones, and draws its roots not only from philosophy but also from neuroscience. This idea takes learning as its paradigm of intelligence, and utilises statistics. According to this vision, the principal value of the computer in the study of the mind is that it provides a very powerful tool for scientists to test hypotheses about brain function in a more precise way. Here the basic idea is that intelligence appears from the interactions of large numbers of simple processing units. This idea was directly inspired by the work of D. O. Hebb [1949]. He suggested that neurones, which connected by synapses, could learn when are simultaneously excited. This excitation increases the strength of the connections between them, leading to a process of learning. Rosenblatt [1962] followed Hebb's notion of synaptic modification [Pfeifer et al 1989] and studied a simple neural model, called a 'perceptron', for many years [Minsky and Papert 1969]. He intended to build a physical device, or to simulate such a device on a digital computer, that could then generate its own abilities. He was able to train the perceptron to classify certain types of patterns. Connectionism or neural network modelling refers to a class of models in which information is stored in the states of connections between computing elements. These models are frequently described as neural-like or brain-like modelling by similarity of them on computation in networks of simple processing elements, which designed in some ways to resemble neurones and their links.

Following publication of the book called 'Parallel Distributed Processing' by Rumelhart and Zipser [1985], interest in connectionism was increased. In the beginning of the 1960s, in addition to symbolic AI, connectionism was one of the two paradigms for AI-research [Pollack 1989]. Connectionist networks with their inherent parallelism could be the best answer for reducing the time of computation.

About 1955 Newell and Simon [1972] concluded that strings of bits, which are manipulated by a digital computer, could represent numbers and also features of the real world. Moreover, they concluded that programs could be used as rules to represent the relations between physical symbols. They assumed that the human brain and the digital computer, while being totally different in structure and mechanism, have a common functional description. Both may be seen as two different instances of a single device that exhibits intelligent behaviour by manipulating symbols using formal rules. They stated that 'intelligence is the work of symbol systems, and symbols lie at the root of intelligent action'. Newell and Simon stated their view as a general scientific hypothesis, called the *physical symbol system hypotheses*. According to this hypothesis, a physical symbol system can be intelligent, and any intelligent system should be implemented via a physical symbol system. A physical symbol system is an instance of a universal machine that uses physical symbols. This hypothesis specifies a general class of those systems, which are capable of intelligent action, and implies that intelligence would be exhibited by a universal computer.

One of the strongest claims of the cited hypothesis is that the symbolic behaviour of a human arises because it has the characteristics of a physical symbol system [Newell and Simon 1990]. It explains human intelligent behaviour in terms of symbol systems.

Today, the physical symbol system hypothesis is a dominant view of behaviour in symbolic AI. In fact the roots of this hypothesis go back to the program of Whitehead and Russell [1910-1913] for formalising logic [Searle 1990]. The sense of formalising logic is avoiding from what seems relevant to meaning and human symbols. Whitehead and Russell's efforts to treat all mathematics as the purely formal derivation of theorems from basic axioms, are indeed the roots of the oldest branch of AI, automatic theorem proving [Bledsoe and Loveland 1984]. Due to the generality of logic, a wide variety of problems can be attacked by this field. The problem description and relevant background information is represented as logical axioms and problem instances are treated as theorems to be solved.

One of the inefficiency of early automated theorem provers was that they proved large number of irrelevant theorems before achieving the correct one. The reason is that

purely formal, syntactic methods of guiding search are inherently incapable of dealing with a large space. Many argued that the only alternative relies on the formal, ad hoc strategies that humans use in solving problems. This is the approach underlying the development of expert system, and it has proved to be a fruitful one [Luger and Stubbefield 1989].

In 1943, Post introduced production systems to model intelligence [Franklin 1995]. Newell and Simon [1972] were the first to employ production systems. Since then, production systems have played a major role in several symbolic AI systems, such as SOAR [Laird et al 1987], and they are used to model cognitive processes. Production systems provide a model of encoding human expertise in the form of rules and designing pattern-driven search algorithms, tasks that are central to the design of the rule based expert systems. Production systems while apply an AI programming language, provide the theoretical foundations of rule based expert systems [Buchanan and Shortliffe 1984]. Rules in rule based symbolic systems are in the form of 'if A then B' and traditional AI programs typically rely on the von Neumann type computer as hardware.

After several decades of research on AI, rule based production systems have emerged as one of the most important and widely employed tools for the implementation of expert systems and other AI programs. The proof for this claim can be the increasing numbers of programs, starting with game playing, and early expert systems such as MYCIN [Shortliffe 1976] and DENDERAL [Buchanan et al 1969] for problem solving. DENDERAL is a program that infers the structure of an unknown chemical compound from various data about this compound. MYCIN is a program that diagnoses certain blood infections and prescribes courses of antibiotics.

Early rule based problem-solving systems clearly suffered from an insufficient knowledge base, and tended to fail when tested outside a narrow problem domain. Some people believed that insufficient knowledge was the most important deficiency of rule based systems, and attempted to improve it by constructing larger collections of everyday facts, which are expressed as formal production rules. The other disadvantage of using production systems is that the matching involved in the match-select-execute is

an inherently inefficient computational process, and about 90% of run time processing is wasted in this stage [Gupta 1985]. Another drawback is that they cannot be used for partial matches between facts and rules. In most rule-based systems, the search strategy is embedded in the code. This limits the range of applications for which the system can be used. The other is concerned with the format of knowledge as a set of production rules, which is not always as easily understood as it would be expressed in some other forms.

Interesting properties of rule based production systems are the modularity and the simplicity of modification and augmentation, which offer potential advantages in the execution of many large-scale AI tasks. On the other hand, the use of these systems in such applications is very expensive in execution time. For example, OPS5 production system programs, using the LISP based interpreter, execute at a speed of only 8-40 working memory element changes per second (WME-changes/sec) on a VAX-11/780 [Gupta 1987]. The slow speed of execution poses particular problems in the case of real time applications, such as real-time speech recognition, that require the program be able to execute at a rate of about 200,000 WME-changes/sec.

Researchers have investigated several different methods to remove the limitations and speeding up the execution of production systems such as the use of faster technology, the use of better architectures, the use of better algorithms, and the use of parallelism. The aim of this thesis is to focus on the use of parallelism for the following reasons:

(1) Production systems appear to be more capable of utilising large amounts of parallelism. The parallelism in rule based systems was motivated by the advantages of separating the knowledge, stored in the knowledge base, from inference, and of modularising knowledge into identifiable and meaningful parts.

(2) It is always possible to use parallel processing techniques to achieve higher execution speeds.

(3) Conventional von Neumann computers are not suitable for real world AI applications. Indeed the von Neumann bottleneck leads directly to a hardware crisis. A

von Neumann architecture, with a centralised control, presents a processor/memory bottleneck to intensive and irregular memory access patterns.

(4) Although the advancement of VLSI technology has improved the performance of single-processor machines, these machines are still not fast enough to perform certain applications within a reasonable time period, such as real time pattern recognition, or real-time speech recognition. There are physical limitations in VLSI technology and we should accept the fact that physical laws limit the maximum speed of the processor's clock, which governs how quickly instructions can be executed [Zargham 1996].

The possibility of using parallel hardware to reduce the execution time of production systems is thus of considerable interest. Over time, serial processing problem solving methods were generalised into parallel symbol processing models. Blackboard systems [Ensor and Gabbe 1985, Nii 1986] are examples of problem solving systems, which evolved from the applying parallel/distributed techniques to the production system approach. In the late 1970s, works on systems such as Contract Net Protocol [Smith 1980] began, which can be considered as the first phase of research into the Distributed Artificial Intelligence. The roots of DAI are lodged in biological and social metaphors and are related to the brain theory, where different neuronal subsystems combine and exchange signals to work together.

Today, some of the earliest disadvantages of production systems, such as the ability to automatically modify or add to the rule base, or dealing with approximate reasoning and uncertainties have been improved by systems such as classifier systems [Kitano et al 1991] and fuzzy production systems [Kandel 1992]. Yet applying parallel processing techniques on production rule systems, parallel production systems [Kuo and Moldovan 1992] and developing parallel match algorithms, such as RETE algorithm [Forgy 1982] and parallel search techniques [Lamont and Shakley 1988] have had great value in reducing some of the drawbacks of production systems. On the other hand, developing neural networks offers partial match and flexibility. Further development of the production systems paradigm can be investigated by combining both symbolic production systems and fuzzy production systems on a connectionist architecture; a connectionist fuzzy production system [Kandel 1992].

1.1 Objective of the thesis

The aim of this research programme is to develop a self-organising knowledge representation language and a corresponding investigation of a suitable parallel architecture to support it. The mechanism of self-organisation is a selection operating on populations of production rule-sets, inspired by the role of neuronal groups in the theory of neuronal group selection (TNGS) [Edelman 1989, 1992].

The theory suggests that the primary mechanism of self-organisation in the brain is a selection operating on populations of synapses, generated with a high degree of a pre-existing variance during the development of the brain. The mechanism of selection is proposed to be changes in the strengths of synapses connecting neural circuits. This mechanism changes the relative contribution of each such circuit, or neuronal group, to behaviour, depending on the evaluation of that behaviour by innate mechanisms, known as value systems [Reeke and Sporns 1993].

After the establishment of the knowledge representation scheme, and the identification of a supporting parallel architecture, a simulation will be developed to emulate the behaviour of such a system, which will then undergo a series of tests to assess its potential benefits. The architecture should offer sufficient computing power to cope with a larger search space provided by the knowledge representation and suitably fast methods of processing. After the completion of the simulation, a test-bed will be defined. The benchmark of the tests to be performed is to assess the suitability of the language developed and its corresponding parallel architecture. The results will then be evaluated, and overall conclusions presented.

1.2 Outline of the thesis

Two major research areas of this project are concurrent production systems and genetic algorithms. The first field provides a background of related work done by other researchers in the area of parallel/distributed production systems. These are related to the development of the knowledge representation within this research work. The second

area, genetic algorithms, has been studied to establish a good understanding of the theme, and consequently adopting a supporting selectionist mechanism for problem solving.

The outline of this thesis is as follows. Chapter 2 provides a discussion of recent research in the field of parallel/distributed production systems. We discuss two popular methods for improving the slow execution of the production systems, i.e., match level parallelism and multiple rule firing. Whilst the implementation of match level parallelism causes no problem for the system, in multiple-rule firing the serialisation problem may lead to incorrect results. This chapter also presents discussions on concurrent production rule systems, including parallel production systems, distributed production systems, and multiple production systems. In Chapter 3, we look at the Genetic Algorithms as heuristic search methods which have been widely used in many applications [De Jong and Spears 1989, Matwin et al 1991, Mansour and Fox 1992, Miller and Todd 1989]. We begin this chapter by highlighting advantages and disadvantages of the algorithm and present a brief discussion on traditional genetic algorithm. Important parameters in design of the algorithm are discussed in more detail. Various implementations including hybridisation and parallelisation, and applications of the algorithm in task allocation, clustering and automated theorem proving are also introduced. Chapter 4 introduces the model of inference and problem solving. The knowledge representation scheme, the inference mechanism for manipulation of the knowledge, and the structure of knowledge base are also discussed along with a description of the design of the system. Verification and validation of the simulated system, testing methodology, criteria for evaluating the behaviour of the system, and results evaluation are presented in Chapter 5. Chapter 6 is the concluding chapter, in which we discuss overall conclusions of the project and provide suggestions and conditions for achieving improved results.

2

Parallelism in Production Systems

2.1 Introduction

Production rule representation was initially intended as a model of human information processing, and it continues to occupy a prominent place in artificial intelligence [Kuo and Moldovan 1992]. Production systems have computationally high costs and consume significant time for their knowledge processing. It has also been discovered that as the scale of rule based expert systems increases, the efficiency of production systems significantly decrease. These drawbacks have restricted the applicability of production systems to large-scale applications and real time AI tasks [Moldovan and Parisi-Presicce 1986, Morgan 1988]. Consequently, researchers have attempted to introduce parallelism into large production systems to extend their usefulness in practical applications. Several multiple processor architectures and algorithms have been developed to improve the efficiency of production systems. Parallel execution of production systems can be investigated in the following ways:

- Parallelism within a rule, which is simultaneous processing of elements forming a rule, including condition level parallelism in which all of the conditions of a single production are matched in parallel, and action level parallelism in which all of the actions of a single production are executed in parallel.

- Parallelism between rules; two kinds of parallel algorithms have been developed to more effectively utilise the parallel processing hardware. One is simultaneous matching (parallel rule matching) of production rules, which aims to improve the performance of the match phase of production cycles that consumes up to 90% of total processing time. Another kind of parallel algorithm is simultaneous firing (parallel rule firing) of production rules, which intends to remove the sequential bottleneck made by conflict resolution strategies that fire a single rule instantiation per cycle, by executing multiple matched rules simultaneously on a multiple processor system. Production systems, which implement the synchronous parallel rule firing approaches are known to parallel production systems, where multiple rules are fired in parallel and globally synchronised in each production cycle.
- Task level parallelism, in which a number of cooperating but loosely coupled production system tasks execute in parallel. This approach is another way for scaling up production system technology and deals with developing interaction mechanisms that allow multiple production systems to work together to solve a single problem. Research in this area can be classified into two categories: First, asynchronous parallel production system or distributed production systems, where multiple rules are fired in parallel without global synchronisation. Second, multiagent production systems, where multiple production system programs compete or cooperate to solve a single problem or multiple problems.

Other sources of parallelism in production systems are potentially available, such as overlapping between phases of the production system cycle. Hierarchies of production systems may be considered for the purpose of operating at several levels of abstraction in large knowledge bases. Also, parallelism may be implemented in the control portion of production systems [Moldovan 1993].

Parallelism in rule matching does not create any problem, in terms of the correctness of the final solution, whilst parallelism in multiple rule firing may. Only independent rules can be applied in parallel. These sets of rules will result from the study of the

interdependencies between rules. Parallelism and distribution in production systems have been studied by many authors, such as Gupta [1985, 1987], Xu and Hwang [1991], Dixit and Moldovan [1990], Neiman [1994], Ishida et al [1994, 1992, 1990], Kuo and Moldovan [1991, 1992], Schmolze [1991], and Stolfo et al [1991].

This chapter provides the descriptions of the work done by others in the area of parallel execution of production systems. It focuses on recent research efforts, ranging from research on parallel processing techniques to distributed processing techniques, for improving the performance of production systems.

Parallel processing techniques, which concern with developing parallel computer architectures, languages, and algorithms, are directed toward solving the performance problems of production systems. Distributed systems ranges from tightly coupled systems, in which there is a completely centralised control mechanism and a shared knowledge base, to ones in which both control and knowledge are fully distributed.

We begin by discussing the architecture of production systems in the next section. Section 2.3 describes the structure of various parallel processing systems and presents a detailed description of one of the early parallel architectures to support the parallel execution of production rule representation. Section 2.4 introduces the concept of parallel rule matching in production systems. Section 2.5 is devoted to the topic of parallel rule firing, including rule interdependencies, compatibility and convergence problems, which are normally discussed in the parallel rule firing processing. Section 2.6 focuses on distributed production systems and overviews some distributed production systems. Section 2.7 describes multiagent production systems. The final section briefly introduces the proposed architecture of the system developed in this research.

2.2 The architecture of production systems

Production rules are knowledge representation languages used for modelling human problem solving, and are particularly useful in classification problems [Ringland and

Duce 1989]. In production rule systems, knowledge is represented by propositions in the form of working memory elements and rules that are closely related to logical implications. The architecture of a production rule system includes three components.

The first component is called working memory. It contains a set of working memory elements representing facts and information about the current state of the problem being solved. Data in working memory comes from the reasoning that the system has performed so far, or from the user of the system.

The second component is the rule base that holds general knowledge about a domain. Rules contain permanent information that applies to all problems that the system may be asked to solve. A production rule is a condition-action pair $C \Rightarrow A$, where C is a set of conditions, and A is a sequence of actions. The production rules can be expressed in equivalent pseudo-code IF THEN formats. If all conditions in a production rule are satisfied, then the sequence of actions is executed.

The inference engine or interpreter is the third component in a production rule system, and is utilised to execute the rules. It must determine which rule(s) should be executed on each cycle, because working memory may match the patterns in more than one production rule. The inference engine may be described as consisting of three action states, match-rules, select-rule, and execute-rule. Two global control decisions made by the inference engine are forward and backward chaining. Forward chaining is the most natural global method for production rule systems [Barr et al 1989]. Forward chaining is breadth-first search facilitated and backward chaining is depth-first search facilitated. EMYCIN [van Melle 1981] uses a rule based knowledge representation scheme with backward chaining mechanism. OPS5 [Browrston et al 1985] is a general-purpose knowledge engineering language, which uses a rule based representation scheme that works using forward chaining as its inference method.

In the match-rules state, the inference engine matches the data in working memory against the conditions of the rule set. Those rules whose conditions match are put into the conflict set. In the second state, select-rule, one of the rules in the conflict set is selected (conflict resolution) for execution. Various strategies have been suggested for

conflict resolution. For example, a simple strategy is to take the first rule in the rule base that matches. Others include exploring all the applicable rules in parallel, choosing a rule arbitrarily, selecting a different rule in successive stages, and firing rules according to their priorities. McDermott and Forgy [1978] have concluded that a simple conflict resolution strategy cannot be completely satisfactory. Different systems use different combinations of the conflict resolution methods. Conflict resolution strategies, such as the OPS5 LEX and MEA strategies, uniquely determine the execution path for a given production system program in a depth-first fashion based on the recency of working memory time tags and the specificity of the rules.

In the execute-rule state, the action part of the selected rule is performed. This leads to the addition of new elements to working memory and the deletion or modification of existing facts.

The advantages of using production rules lie in their naturalness for expressing the type of heuristic knowledge, or rules of thumb. Production rule systems are very modular. Permanent knowledge stored in a rule base is distinguished from the temporary knowledge represented by working memory elements. The inference engine is independent of the knowledge encoded in the rule base and working memory. Each production rule represents a specific piece of knowledge that is completely independent of other rules. Thus, updating the knowledge base becomes much easier. The modularity of the knowledge base simplifies constructing and maintaining rules. The only way in which rules in a rule base can communicate with each other is through the working memory. Production rule systems can have explanation facilities, which are used for debugging or learning purposes.

One of the disadvantages of using production systems is that the matching involved in the match-select-execute is an inherently inefficient computational process, and nearly all of run time processing is wasted in this stage [Gupta 1985].

Before discussing parallel processing of production systems, it seems to be appropriate to first review several classes of parallel architectures.

2.3 Parallel architectures

Within the four categories of Flynn's [1972] classification of computer architectures, three architectures identify the parallel processing systems: SIMD (single instruction stream, multiple data stream), MISD (multiple instruction stream, single data stream), and MIMD (multiple instruction stream, multiple data stream) architectures. In a SIMD architecture, many identical processors simultaneously execute the same instructions on different data. Processors can also transfer data amongst themselves through an interconnection network. In a MISD architecture, several instructions would apply to a single piece of data. This class of machines has been considered as impractical or impossible [Trew and Wilson 1991]. An alternative is to consider a class of machines in which the data flows through a series of processing units [Zargham 1996]. These types of machines are considered as pipeline architectures. Pipeline architectures perform a series of stages, each of which performs a particular function during a specified period of time, producing an intermediate result. In a MIMD architecture, several independent processing units intercommunicate over an interconnection network. MIMD machines have the ability to perform multiple tasks in the same time, concurrent to each other.

Most proposed taxonomies consider the SIMD and MIMD classes of Flynn's classification as two general classes of parallel processing [Hord 1993]. However, advances in computer technologies have created new architectures, such as hybrid architectures and artificial neural networks [Maren et al 1990] that cannot be clearly defined by Flynn's taxonomy.

The parallel systems of the MIMD class fall into two categories: Shared memory multiprocessor (tightly coupled) and message passing multicomputers (loosely coupled). A tightly coupled multiprocessor can be considered as a parallel computer consisting of several interconnected processors that share a memory system. The processors in a multiprocessor system communicate with each other through shared variables in a common memory. The interconnection network connects each processor to some subset of the memory modules. Direct interprocessor communications are supported by optional interprocessor communication networks, instead of through the shared memory. Each processor can execute a different part of a program or they can all

execute different programs simultaneously. A transfer instruction causes data to be moved from each processor to the memory to which it is connected. Distributed memory multicomputers correspond to loosely coupled MIMD systems with distributed local memories attached to each processing element. A multicomputer system can be viewed as a parallel machine in which each processor has its own local memory. In multicomputers the main memory is distributed among the processors. This means that a processor only has direct access to its local memory and cannot access the local memories of other processors. The interconnection network connects each processing node, consisting of a processor and a local memory, to some subset of the other processing nodes. Message passing is the most popular communication method used to transfer data between the computing nodes in a multicomputer system.

Hwang et al [1987] have presented a taxonomy of computer architectures for artificial intelligence processing. One of the AI machines in this taxonomy are knowledge-based machines, which support particular knowledge representations such as semantic networks, production rules, or frames. In particular, production systems are a type of knowledge-based machine, supporting production rule representation. To improve the efficiency of production systems, several parallel architectures have been investigated. For instance, the parallel architecture for executing production rules has been implemented by Stolfo and Miranker [1986] for DADO, Shaw [1985] for NON-VAN machine, Moldovan [1989] for RUBIC, and others. Other parallel systems such as PARS [Winston 1984], PSM [Acharya and Tambe 1989] and CREL [Miranker et al 1990] have been simulated. A survey of parallel architectures for executing production systems can be found in [Kuo and Moldovan 1992].

Although there have been attempts to implement production systems on SIMD machines [Bahr et al 1991], MIMD architectures better match the needs of production system algorithms [Gupta and Tambe 1988]. In the following section we review the DADO machine, which is designed as a production rule based system, using a database of productions to derive conclusions and a database of facts with a method of inference.

2.3.1 DADO architecture

DADO2 [Stolfo 1987] is the second implementation of a class of machine architectures, termed DADO [Stolfo et al 1981] and uses a forward-chaining strategy. The DADO class of machines is characterised by a binary tree topology, fast broadcast from root to leaf and the ability to segment the tree into sub-trees working in multiple SIMD fashion. Different implementations differ in the number of processing elements (PEs) and the granularity, memory capacity, and processing power of each PE. DADO2 is a medium grained tightly coupled parallel machine with 1023 nodes arranged in a 10-level binary tree. Each node consists of an 8-bit microprocessor, 64K of RAM, and a switch. The switches can broadcast 1 byte from root to leaf in one instruction cycle. DADO2 can divide itself into sub-trees that can then work independently on matching productions. Each node can operate in either MIMD or SIMD mode. In MIMD mode, a node will ignore all instructions from nodes higher up the tree and execute instructions stored in local RAM independently of the other PEs. In SIMD mode, it executes instructions from higher nodes. The upper levels of the tree perform the select and fire phases, together with synchronisation. The middle level of the tree operates in MIMD mode during the match phase, and the leaves store the facts and carry out the lower level pattern matching. When the leaves complete their tasks, the results of successful matches are sent back up the tree. In effect DADO2 can function as a multiple-SIMD machine. A single conventional host processor adjacent to the root of the DADO tree controls the operation of the entire set of PEs.

2.4 Parallel rule matching

Investigations into speeding up the match-state of production systems can be divided into two categories: Speeding up the match phase by faster sequential match algorithms, and speeding up the match phase by parallel processing techniques.

Moldovan [1993] has divided the match algorithms into two types of state-saving and non-state-saving algorithms. In a non-state-saving algorithm, each rule is matched

with every working memory element in each match-select-execute cycle. As a result, the performance for these kinds of algorithms is very slow. The basic idea behind a state-saving match algorithm is that the state of the match is stored. Then, by updating the changes to the match state caused by the changes to the working memory, a new set of eligible rules is easily determined without redundant matching. The state-saving algorithms are faster sequential match algorithms. The RETE match algorithm [Forgy 1982], used by uniprocessor implementations of OPS5 [Browrston et al 1985] and SOAR [Laird et al 1987], is an example of a state-saving algorithm. The RETE algorithm uses a special kind of a data-flow network compiled from the left-hand sides of productions to perform a match. Because only a small fraction of the working memory component changes in each cycle, the algorithm stores results of match from previous cycles and uses them in subsequent cycles. The RETE algorithm also utilises the similarity between condition elements of productions and then performs common tests only once. These two features combined together make the RETE a very efficient algorithm for match. Other improved sequential algorithms such as YES/RETE [Highland and Iwaskiw 1989] have been proposed. Miranker [1987] has proposed the TREAT match algorithm, which was designed for fine-grained parallel processor systems. The RETE and TREAT algorithms are the best known state saving algorithms, which avoid recomputations of comparisons done in previous match-select-execute cycles. Both algorithms map the patterns of the LHSs of the rules to nodes of a network. Speedups of 4 to 10-fold were reported for a number of production systems when faster match algorithms were used.

Another approach to improve the match phase is to apply parallel processing techniques. The parallel match systems enhance the performance of production programs by parallelising the match phase. Many efforts have tried to parallelise the match state, while leaving the system to continue executing only one rule at a time. Research on parallel rule matching started with the DADO project [Stolfo et al 1981] followed by the PSM project [Gupta et al 1986]. Stolfo and Miranker [1986] offered parallel match algorithms for DADO. Since the parallel rule matching does not affect the semantics of the production system programs, most of the research has focused on partitioning the RETE network and mapping the partitions to parallel processors. Forgy,

who first proposed the RETE algorithm for speeding up the matching operations in uniprocessor production systems, then adopted it for parallel environments [Harvey et al 1991].

Parallel match algorithms have been investigated on shared memory multiprocessors and message passing multicomputers. The PSM machine, on which Gupta implemented the paralleled RETE algorithm, is a shared memory multiprocessor and exploits very fine-grained parallelism. Paralleled RETE algorithm has also been developed by Gupta and his colleagues and simulated on the Nectar simulator, a message-passing computer [Kuo and Moldovan 1992]. For most of these works, maximum speedups realised by these approaches are about 10 times the fastest sequential version of OPS5, independent of the number of processors used [Gupta 1987]. However, there are some exceptional cases such as DRETE [Kelly and Serviro 1989], where speedups of 23-fold for match algorithms are reported. One reason for limited speedups for parallel match systems is that only a small number of changes to working memory occur in each cycle. These investigations have shown that the total speed up available from this source is insufficient to improve the problem of slow execution in large-scale production system implementations [Xu and Hwang 1991]. Then continuing to speed up the match phase alone will not provide significant overall performance improvement for production systems [Kuo and Moldovan 1992]. This problem can be further improved by allowing multiple rules to fire concurrently.

2.5 Parallel production systems

Parallel production systems are the results of efforts that have aimed to reduce the total number of sequential production cycles by executing multiple matching rules simultaneously on multiprocessor architectures. In a parallel rule firing system, many rule instantiations are fired in parallel to reduce the total number of sequential cycles and these are globally synchronised at the select phase. In a parallel firing model, it is not assumed that only one rule is chosen in the select phase, rather the firing of multiple rules simultaneously on multiple processors will be proposed. Firing multiple rules in

parallel produces a larger number of changes to the working memory in one cycle, where the whole conflict resolution set is fired in each cycle. The SOAR production system language takes the parallel rule firing approach [Gupta 1985]. In SOAR there is no conflict resolution phase and multiple productions can fire in parallel. The SOAR production system programs can also improve their performance over time by adding new productions at run time.

Synchronous production firing systems require synchronisation in the select phase of the match-execute cycle. All the productions can be matched in parallel but synchronisation must occur in the resolve phase in order to select the production or set of productions to fire next. In the execute phase, the selected productions can be fired in parallel.

The parallel rule firing mechanism was first investigated by Ishida and Stolfo [1985], followed by Moldovan [1986]. Later Schmolze [1991], Kuo and Moldovan [1992], Kuo, Miranker, and Browne [1991] addressed this problem. Parallel rule firing systems can be coupled with parallel match algorithms to obtain further performance improvement in addition to the speedups from the match phase, by firing multiple rule instantiations in a production cycle.

Since the parallel firing model allows multiple rules to be fired in parallel, programmers are thus not required to prioritise rules or assume any particular conflict resolution strategy [Ishida 1994]. Unlike parallel rule matching, results from parallel rule firing may differ from the sequential firing of those rules in an arbitrary order. To solve this problem interference analysis techniques should be applied and detected in the select phase. Parallel rule firing explores medium grain parallelism between rules. This method is often implemented on a message passing multicomputer with distributed memories [Xu and Hwang 1991].

Understanding the parallel rule firing approach requires discussion on the following topics.

2.5.1 Rule interdependencies

Rule dependencies were first proposed independently by Ishida and Stolfo [1985], and Tenorio and Moldovan [1985]. The main results were the formulation of dependencies between production rules. The graph grammar theory [Ehrig 1979] provided the mathematical foundation for studying rule interdependencies and subsequent parallelism.

For a pair of rules three types of dependencies can be defined: Inhibit dependence, output dependence, and enable dependence [Moldovan 1993]. Inhibit dependence occurs when the firing of one production rule deletes or adds working memory elements such that the condition part of another production rule is no longer satisfied. Output dependence occurs when the working memory elements that are added by the firing of one production rule are deleted by the firing of another production rule. Enable dependence occurs when the firing of one production rule deletes or adds working memory elements such that the condition part of another production rule may then be satisfied. Two rules are said to be compatible if they are neither inhibit-dependence nor output-dependence. It has been proved that for any two compatible rules the result of sequential and parallel processing of rules is equivalent. The above discussion is also valid for inter rule instantiation dependencies [Kuo and Moldovan 1992]. Data dependence analysis is sufficient to determine compatible rules.

2.5.2 Compatibility problem

In a serial production system, a conflict resolution strategy selects one instantiation to execute in each cycle, while in multiple rule production systems, a subset of matching instantiations is selected to fire. This can yield results that are impossible for a serial system to produce, leading to erroneous behaviours. This problem can be addressed by the compatibility and convergence problems. The problem of finding a set of compatible rules that are allowed to fire simultaneously, i.e., the resulting working memories are the same in both sequential and parallel is called the compatibility

problem by Moldovan [1986], serialisation problem by Schmolze [1991], and synchronisation problem by Ishida and Stolfo [1985].

Most work in this area is based on data dependency graphs to detect interference among rules. Early works regarding this problem are based on compile time analysis to determine which rules may be fired in parallel from the conflict set. Ishida and Stolfo [1985] first identified the compatibility problem underlying parallel rule execution. They defined synchronisation as a solution for the serialisation problem by synchronising certain pairs of instantiations, i.e., prohibiting certain pairs from co-execution. These authors produced, at compile time, a compatibility matrix. Rules that could potentially interfere with each other were labelled as incompatible. At run time, incompatible rules were prevented from being executed simultaneously. The select phase makes sure that no two instantiations are executed simultaneously for rules that cannot coexecute, i.e., a pair of rules that cannot simultaneously execute are synchronised. This approach, which is based solely on compile time analysis does not guarantee the serialisable behaviour in the system.

Later Schmolze [1989] and Ishida [1990] independently improved this framework. Their work was based on an additional run time analysis to detect interference amongst instantiations, rather than amongst rules, by combining compile time and run time analysis. Therefore, if interference could be roughly analysed at compile time, then run time analysis process would not take much time. Schmolze developed a number of algorithms, with various increasing degrees of precision and cost. These algorithms require obtaining the binding of variables by tests that have to be executed at run time. He synchronises instantiations not only rules whose instantiations might interference. One instantiation disables another if executing the first would cause the second to match no longer. This occurs if the first one deletes (adds) a WME that the second one matched positively (negatively). Run time analysis consumes considerably more time than compile time analysis. The trade-off is to perform less work at run time and more work at compile time. Combining compile and run time analysis is a formal solution to the problem of guaranteeing serialisable behaviour in synchronous parallel production systems that executes many rules simultaneously. Ishida's model of a parallel firing system [1991] guaranteed serialisability by combining the compile and run time

analysis techniques based on data dependency analysis to permit efficient and accurate interference detection. His model detects interference in the select phase, choosing as many instantiations as possible as long as interference does not occur among selected instantiations, firing rules according to the selected instantiations simultaneously.

Most parallel rule firing systems reported in the literature employ some form of rule interaction analysis [Kuo et al 1991, Kuo and Moldovan 1992, Stolfo et al 1991].

The advantage of compile time analysis is that the computations are done before processing. The disadvantage is that only partial data dependence analysis is possible, due to the presence of variables in production systems. These ambiguities can be resolved at run time when the variables are explicitly bound. Compile time analysis checks the dependencies between rules and then introduces less parallelism. This checking process alone cannot definitively determine whether two rule instantiations are in conflict, but rather it can determine that they may be in conflict. This approach usually refuses concurrent execution of two rule instantiations that believed to be in conflict if they are actually not.

Run time analysis is more complete than compile time analysis. At run time all variables are bound and it is possible to analyse the data dependencies between rule instantiations instead of just between rules. This increase in parallelism comes at the cost of increasing computation time. As a result, most researchers propose a mixture of compile time and run time analysis. The idea is to use the compile time result for most of the compatibility checks and to use the run time analysis only when the compile time analysis does not give a precise answer. Run time analysis approaches suffer from synchronisation, and testing of possible rule conflicts is expensive in time and decreasing parallel performance.

2.5.3 Convergence problem

In a parallel rule firing system, finding a solution for the compatibility problem is not sufficient to guarantee the correctness of the final solution. Correct solutions are not

always reached when multiple compatible rules are fired [Moldovan 1993]. This is because not all compatible rules in an inference cycle should be fired if they are to produce correct solutions. Kuo and Moldovan [1992] first pursued the convergence problem. Their motivation is to ensure the correctness of the parallel rule firing. The convergence problem is concerned with firing only rules that achieve correct solutions. Correctness of parallel rule firing in parallel production systems is called the 'convergence problem' [Kuo and Moldovan 1992]. The convergence problem is caused by violation of the problem solving strategy. Firing compatible rules arbitrarily without taking into consideration the problem solving strategy can result in incorrect solutions. The convergence problem can be resolved by applying the idea of nondeterministic execution system or by developing a parallel conflict resolution strategy.

In a nondeterministic production system, a rule instantiation is chosen arbitrarily and fired without the help of conflict resolution strategy. The correctness of a nondeterministic program is guaranteed by proving that all possible execution sequences reach the correct solution. Since all execution sequences reach the correct solution, firing a set of compatible rule instantiations in a nondeterministic production program is guaranteed to reach the correct solution [Kuo et al 1991].

One solution for the convergence problem is developing a parallel conflict resolution strategy in the format of a set of meta rules. The inputs to the meta rules are the matched rule instantiations [Stolfo et al 1991]. A set of noninterfering rule instantiations can be selected by meta rules and fired.

2.5.4 Rule distribution

Parallel rule firing explores medium-grain parallelism between rules. This method is often implemented on a message passing multicomputer architecture with distributed memory. The execution of a production system on a multicomputer involves the problem of partitioning the rules among the processors, where each production set has its own conflict set. The production system program is partitioned into several parts so that the processing required by productions in each partition is almost the same, and

then each partition is allocated to one processor. Each processor works from its local memory and communication between processors is required. The aim is to partition the rules to processors such that the total cost of communication is minimised. In the extreme case, the number of partitions equals the number of productions in the program, so that the matching for each production in the program is performed in parallel. In general, the number of rules in an expert system is often much greater than the number of nodes in a multicomputer system. Thus many rules may be allocated to a single node. The assignment of production rules to nodes has a great impact on the overall system performance. For example, if two rules have to communicate with each other, assigning them to different nodes will take longer to communicate. The task of partitioning is difficult because good models are not available for estimating the processing required by productions, and also because the processing required by productions varies over time. Another difficulty is to find partitions of the production system that require the same amount of processing, i.e., assigning a balanced workload to all nodes. To find an optimal allocation of the production rules has been proven as an NP-complete problem [Harvey et al 1991].

This distribution could be done automatically or with the help of the user [Miranker et al 1990]. A decomposition algorithm is proposed to find an optimal partition or distribution of given production rules onto multiple processor elements, so that the gain of parallel rule firing increases as much as possible. Ishida and Stolfo [1985] have outlined a method of decomposing production system based on the data dependency graph. Dixit and Moldovan [1990] have proposed a method to solve the allocation problem using a heuristic method, the A* algorithm. However, their method introduces an overhead, which increases rapidly with the number of rules in a system. Xu and Hwang [1991] have presented a simulated annealing method to achieve a balanced processing of rule-based systems on a multicomputer. Their main purpose is to minimise communication costs in message passing among nodes. Their method is an improvement over previous approaches. A cost function is introduced to minimise load imbalance and expected communication cost.

2.5.5 Parallel rule firing models

Several parallel rule-firing models have been proposed, but few simulation and implementation results have been presented [Kuo and Moldovan 1992]. This is because the problems associated with parallel rule firing have not been thoroughly investigated. Problems to be considered are how to select a set of rules which do not interfere with each other, and how to ensure that the parallel execution reaches the goal states. Implementations of parallel rule firing on hardware systems include PARULEL [Stolfo et al 1991], CREL [Kuo et al 1991] and RUBIC [Moldovan 1989]. Moldovan designed RUBIC and used synchronisation at the rule level to guarantee serialisability, allowing two rules to execute simultaneously if, and only if, neither can disable the other. CREL (Concurrent Rule Execution Language) has been developed by Miranker et al [1990]. CREL addresses the compatibility problem by analysing the inter rule data dependencies at compile time. Rules are divided into clusters such that rule instantiations for rules belonging to different clusters can be executed in parallel. Run time analysis is needed to select and fire instantiations in the same cluster simultaneously. The convergence problem is resolved by ensuring that all eligible serial execution paths reach correct goals. It is the programmer's responsibility to write a CREL program and to ensure that it is correct.

PARULEL [Stolfo et al 1991] focus on rule instantiations and not simply the rules themselves. It does not perform compatibility or convergence checks, but allows these requirements to be encoded as the meta-reaction rules. The meta rules give the programmer finer control of production program execution. First, all rules that are matched and whose instantiations enter the conflict set are intended to be fired in parallel. It uses a set of meta rules, called reaction meta rules, to eliminate rule instantiations from the conflict set prior to parallel firing. Meta rules constitute a programmable conflict resolution scheme. The programmer provides meta-level rules that dictate the conditions when two rule instantiations are in conflict, and which instantiations need to be deleted from the conflict set. It is the programmer's responsibility to write correct meta rules such that the selected rule instantiations do not interfere with each other. The selected rule instantiations are fired in parallel. The

advantage of this approach is that there is no need to add run time synchronisation control, eliminating the associated overhead. The disadvantage of this approach is that a programmer may not specify a complete set of correct meta-rules, thus developing a program that is not conflict-free.

Kuo and Moldovan [1992] present three multiple-rule firing models: The rule dependence model (RDM), the single context-multiple rules model (SCMR), and the multiple-contexts-multiple-rules model (MCMR). The RDM model, fires a set of noninterfering rules by analysing the inter rule data dependencies. This model provides speed advantages, but sometimes fails to reach the correct solution as provided by sequential firing. The SCMR model addresses the convergence problem by restricting the number of parallel rules fired. A production program is divided into its constituent tasks, called contexts. In the SCMR model, the contexts are fired one at a time. Parallel rules within a context may fire simultaneously. The MCMR model improves the parallelism over the SCMR model by executing multiple contexts at a time. The RDM model addresses the compatibility problem using inter rule data dependence analysis. The SCMR and MCMR models address both the compatibility (caused by interfering rules) and convergence problems (correctness of parallel solutions). The MCMR model has been simulated on the RUBIC simulator. The MCMR model addresses the compatibility and convergence problems at two levels, the context and program levels. Production rules can be divided into rule sets (contexts) according to the tasks that they perform. A control flow diagram is derived by analysing the interactions between different contexts. At context level, the conflict resolution can be eliminated and compatible rule instantiations can be fired concurrently for converging contexts. For a sequential context, rule instantiations are executed serially. At the program level only compatible contexts can be activated. Two contexts are compatible if they do not contain rule instantiations that have data dependence with rule instantiations in the other context, and executing the contexts concurrently does not violate the control flow.

Parallel production model proposed by Ishida [1990] resolves the compatibility problem by data dependence analysis coupled with a selection algorithm and the convergence problem by grouping rules into parallel and sequential rule sets. It is the

programmer's responsibility to ensure that rule instantiations in a parallel rule set do not interfere with each other.

A number of other works have concentrated on multiple rule execution production systems, not all of which guarantee serialisability. For example, BLITZ [Morgan 1988] provides mechanisms to control which rules can simultaneously execute but leaves this responsibility to the programmer.

The evaluation results of several production system applications show that the degree of concurrency can be increased by a factor of 2 to 9 by introducing parallel rule firing [Ishida et al 1992].

2.6 Distributed production systems

Distributed production systems (asynchronous production systems) are a special case of distributed problem solving systems. Research in distributed problem solving systems considers how the work of solving a particular problem can be divided among a number of nodes that cooperate and share knowledge about the problem and about the developing solutions. There are many reasons for distributing intelligence. First, many AI applications are inherently distributed, especially functionally or temporally. Second, to increase the speed of the system or efficiency and achieving faster problem solving by exploiting parallelism. Third, desire to distribute control to reduce the degree of tail. Forth, to simplify the development and design of systems by building these as collections of separate, but interacting parts (modular design and implementation).

A distributed production system is composed of a set of separate modules or production system agents and a set of communication paths between them. Each module is usually expected to behave as a problem solving entity in its own right. To perform domain problem solving by multiple production system agents however, each agent needs knowledge that represents the necessary interactions amongst production system agents. Each production system agent is a production system capable of interacting with

other agents. It consists of an interpreter, a part of the domain knowledge and domain data. Each production agent has its own conflict set.

For asynchronous execution of production systems, the theory of parallel rule firing is extended to distributed rule firing, where problems are solved by a society of production system agents using distributed control. In asynchronous production firing systems, rules are distributed amongst multiple processors and fired in parallel without global synchronisation in production systems. Thus, these systems do not have distinct match, select, and execute phases across the parallel system. Distributed production systems achieve speedup over synchronous parallel production systems by eliminating synchronisation bottlenecks [Schmolze and Goel 1990]. In a collection of distributed production system agents, rules are asynchronously fired by distributed agents. Advances in hardware technology make it possible to connect large numbers of processing units that execute asynchronously.

As already described in previous sections, interference exists among rule instantiations when the result of parallel execution of rules is different from the results of sequential execution applied in any order. For distributed production systems, only compile time analysis has benefit because run time analysis requires global synchronisation, which is too expensive in distribute situations.

Schmolze and Goel [1990], Ishida [1994], Ishida et al [1990], and Neiman [1994] have studied distributed rule firing to eliminate synchronisation overheads in parallel rule firing. Schmolze [1990] has proposed an asynchronous distributed production system, called PARS. Production rules and WMEs are distributed amongst the processors, and each processor executes the inference cycle asynchronously. The inter rule data dependence are analysed at compile time. To ensure that incompatible rule instantiations are not fired simultaneously, disable messages are sent between processors. Each processor selects a rule instantiation from its locally matched rule instantiations. Before firing a rule instantiation in a processor, the set of rules that are incompatible must be disabled. Disabled messages are only sent to processors that contain incompatible rules with higher priority. To prevent a cyclic disable message between production rules, they are prioritised and higher priority rules may fire as soon

as they matched. The selected rule instantiations are fired in the act phase and the changes to the working memory are updated by message passing. One way to improve the performance of the system is to allow each processor to select and fire multiple rule instantiations in a cycle. In addition, the PARS system also needs to address the convergence problem.

Neiman [1994] presents an asynchronous rule firing approach that allows rules to execute as soon as they become instantiated, therefore avoiding synchronisation delays. In his model, eligible rules are placed in an eligibility set, undergo processing or working memory locking scheme to ensure that they do not conflict with other eligible rules, and then placed on an execution queue. A single dedicated processor is responsible for the conflict resolution/locking/scheduling phases. The rate at which rules can be processed depends on the rate at which this scheduling process can place rules into the execution queue. The working memory-locking scheme presents a number of advantages. The overhead of maintaining correctness during rule firing is low. It is not necessary to compare each eligible rule against all others, so rules may fire asynchronously and the synchronisation bottleneck is removed. No compile time analysis is required. The primary disadvantage of the working memory-locking scheme is that it does not guarantee serialisable problem. Serialisable executions will result by prohibiting cyclical disabling relationships from rule instantiations. However, a locking scheme cannot prevent rule instantiations due to negated condition elements in rules. The programmer is responsible to ensure the correctness of the program.

Ishida [1994] defined a production cycle model of distributed production system agents by extending the conventional match-select-execute cycle to solve inter agent data transfers and synchronisation. Inter agent inconsistency caused by distribution is locally resolved by using temporary synchronisation via rule deactivation. His distributed firing protocol for each agent is as follows: Each agent may receive/send a synchronisation request, a synchronisation release and a MW modification message from another agent. When an agent receives a synchronisation request message from some agent, it deactivates the corresponding rule until receiving a synchronisation release message from the same agent. When receiving a WM modification message, it updates the local working memory to reflect the change made in another agent's WM.

For each rule, match the conditions with the current WM. Choose one instantiation of a rule that is not deactivated. Using interference knowledge, send synchronisation request message to the agents requiring synchronisation and handle all WM modification messages that have arrived during synchronisation. If the selected instantiation is deactivated send synchronisation release messages and restart the production cycle. Otherwise, fire the selected rule instantiation. Using the data dependency knowledge, inform dependent agents with WM modification messages. Send synchronisation release message to all synchronised agents.

The organisation self design approach proposed by Ishida et al [1992], is a technique for building problem solving production systems capable of adapting to changing environmental conditions. This approach can complement other approaches currently developed for real time expert systems, such as approximate processing techniques and adaptive intelligent system [Hayes-Roth et al 1989]. Ishida's approach is based on reorganisation of a collection of problem solvers, where problems solved by a society of distributed problem solving agents, with the aim to achieve adaptive real time performance. Authors introduced new reorganisation primitives, composition and decomposition of agents, and reported an actual performance of reorganisation in real-time. Organisation self design is performed in the following way: Initially, only one agent knows about all domain and knowledge about relationships among agents (data dependencies, interference, and locations of domain rules), exists in the organisation. The organisation knowledge for the initial agent is prepared by analysing its domain knowledge before execution. For effective reorganisation, agents implement the decomposition or composition primitives for each situation. Decomposition divides one agent into two, to increase parallelism. Composition combines two agents into one, to save computing resources for cost-effective problem solving or to reduce response time (when communication overheads cannot be ignored). Composition and decomposition, which change the population of agents and the distribution of knowledge, can be performed simultaneously in different parts of the organisation. These primitives are made solely on the basis of firing ratios, and the choice of rules to transfer is made arbitrarily. Moreover, partial knowledge transfer among existing agents can be

combined with the composition and decomposition to provide a flexible and distributed task-sharing system.

Previous research on reorganisation typically aimed at changing agent roles or inter agent task ordering, such as the contract net [Davis and Smith 1983] and [Durfee and Lesser 1987]. The contract net framework utilises the negotiation as a fundamental mechanism for interaction. Blackboard systems are organisation self design systems that inherently decomposable and consist of independent knowledge components.

The blackboard architecture is a problem solving approach initially developed for the HEARSAY-II [Fennel and Lesser 1981] speech-understanding systems and applied to the variety of other problem domains, such as vision [Hayes-Roth 1985]. It considers problem solving as an incremental process of assembling a satisfactory configuration of partial solutions. It is based on three basic assumptions: (1) All partial solutions generated during problem solving are recorded in a global database called the blackboard. (2) Partial solutions are generated and recorded on the blackboard by independent processes called knowledge sources (KSs). (3) On each problem solving cycle a control mechanism chooses a single KS activation record to execute its action. The common mechanism is a sophisticated scheduler that uses a variety of criteria such as expected value of the knowledge sources action.

In a blackboard system, the rule base is divided into different KSs, each of which can be regarded as a representation of the knowledge relevant to a particular subsystem within the larger problem that the system is trying to solve. Each knowledge source has its own interpreter. Knowledge sources communicate with each other through the shared blackboard. The blackboard thus has a similar function to working memory in a traditional production system. In computational terms, the blackboard is a shared memory and each KS is a modular piece of software, an expert on processing hypotheses that satisfy a particular pattern. When a hypothesis matches a particular pattern in the KS, the KS does some amount of processing and then sends new hypotheses to the blackboard. This will then cause another KS to be invoked. Detecting and responding to a suitable solution can be the task of a dedicated KS that compares hypothetical solutions against thresholds for completeness and confidence, or it can be

itself be a distributed process. The blackboard controlled by having a prioritised agenda of potential KS executes, each rated by how well the KS expects to perform on the current data. On each cycle, each knowledge source notifies a global interpreter where there is new information on its partition of the blackboard that it can use to derive yet more information. The global interpreter then makes a control decision about which knowledge source to execute. The global interpreter selects one of the knowledge sources for execution.

The blackboard architecture has the advantage of providing an asynchronous communication link (the blackboard) among independent KS modules. Expertise can be added to the overall system incrementally by introducing more and more KSs. Because KSs are involved based on patterns of entries on the blackboard, KSs never have to be aware of other KSs.

2.7 Multiagent production systems

Multiagent systems have been considered as another arena of DAI [Bond and Gasser 1988]. Research on multiagent systems is concerned with coordinating intelligent behaviour among a collection of autonomous intelligent agents, how they can coordinate their knowledge, goals, skills and plans jointing to take action or solve problems. Agents in a multiagent system may be working on a single task or separate individual tasks that interact. Ishida [1994] first extended the theory of parallel and distributed production systems to create a basis for multiagent production systems. Similarly, in a multiagent production system multiple production system agents compete or cooperate to solve a single problem or multiple problems. According to the distribution of production rules and/or working memory over the processors, multiagent systems can be classified into shared memory and distributed memory multiagent production systems [Ishida 1994]. In a shared memory multiagent production system, production system interpreters access to their own production memories and to one shared working memory. A transaction manager executes transactions concurrently produced by the multiple production system interpreters. In a distributed memory

multiagent production system, production system interpreters access to their own production memories and working memories. Shared information is distributed among different working memories. In this case, multiple transaction managers, corresponding to production system interpreters, provide an integrated view of distributed working memories. Then, the consistency of distributed working memories is maintained through cooperation between multiple transaction managers. The main issue, in both models, is how to provide a transaction model that guarantees the consistency of shared working memory or shared information distributed among multiple production system interpreters. Ishida has proposed a transaction model based on a shared working memory information architecture. An efficient concurrency control protocol called the lazy lock protocol is introduced that guarantees inter agent consistency. A new problem in multiagent production systems is the inter-transaction consistency problem. A solution to overcome this problem is presented. However, it needs to guarantee inter-transaction consistency.

At the end of this chapter, it is appropriate to review the aim of this research more technically and introduce the parallel architecture suitable for this model of problem solving. Research in this project has focused on a distributed problem solving architecture and considers how the work of solving a particular problem can be divided among a number of production systems that cooperate and share knowledge about the developing solutions. The domain knowledge, in the format of production rules, is distributed over the processors. Shared information is distributed among clusters of different processors. Clusters' supervisors cooperate with each other to provide an integrated view of distributed shared information among the clusters.

2.8 Appropriate parallel architecture

The parallel architecture suitable for this model of knowledge representation, which could be used as the basis for developing a parallel language, can be suggested by treating several major considerations. Theses include network topology, timing method,

switching methodology and a control strategy [Almasi 1985]. Static topologies are appropriate for problems whose communication patterns can be predicted reasonably well, whereas dynamic topologies (switching networks) are suitable for a wider class of problems.

Timing methods are available in synchronous and asynchronous modes. When the network requires a synchronous stream of instruction or data, a synchronous communication system is required. Synchronous communication seems an appropriate choice for SIMD machines, because most SIMD machines transmit data at the same time. When connection for an interconnection network is proposed dynamically, an asynchronous communication system is required. In MIMD machines, processors are free to execute separate instructions and this means they are inherently asynchronous.

The three main types of switching methodology are circuit switching, packet switching, and integrated switching. Circuit switching establishes a complete path between source and destination and holds this path for the entire transmission. It is best suited for transmitting large amounts of continuous data. In packet switching, data items are partitioned into fixed sized packets. Packet switching has no dedicated physical connection setup. It is most useful for transmitting small amounts of data. Most SIMD machines use circuit switching, while packet switching is most suited for MIMD machines. The combination of circuit and packet switching is called integrated switching.

All dynamic networks are composed of switch boxes connected together through a sequence of links. The states of the switches of a network can be set by a central controller or by each individual switch. The first state is a centralised control system, while the second state is a distributed control system. Thus we have two modes of control strategy, centralised and distributed.

Following this short discussion on criteria for choosing an appropriate parallel architecture, we consider the structure of the proposed system in this research programme as a two dimensional mesh-based message-passing multicomputer (MIMD), using a physically distributed memory. Thus the system offers higher memory

bandwidth and potentially higher scalability than a shared memory architecture. Two-dimensional meshes provide a good structure for applications that require strong local communication. The major components of the system are the user interface, secondary storage, switching network, controller, and the array of processing elements.

2.9 Summary

This chapter has attempted to familiarise the reader with research on the field of parallel/distributed production systems. While production systems have the advantages of modularity and ease of enhancement; they suffer from slow execution, especially in large-scale AI applications and real time tasks. Researchers have consequently investigated systems, using parallel processing techniques or distributed problem solving techniques that gain more speedups. These researches can be divided to parallel rule matching systems, parallel rule firing systems, distributed production systems, and multiagent production systems. The chapter starts by describing the architecture of production systems and has a glance of the field of parallel processing architectures. It continues with a discussion on the speeding up of the match phase of production systems in two directions; applying faster sequential match algorithms and parallel processing techniques. Parallel production systems are described in the next section. There exists the case where the result of parallel execution of rules is different from the results of sequential executions, leading to erroneous behaviours. This problem can be addressed by the compatibility and convergence problems. The next section concentrates on distributed production systems, where there is no requirement for synchronisation in the select phase across processors. To perform domain problem solving by multiple production system agents however, each agent needs knowledge that represents the necessary interactions between production systems agents. The theory of multiagent production systems is briefly discussed. An appropriate parallel architecture for the proposed system in this research programme is introduced.

3

Genetic Algorithms

3.1 Introduction

Genetic algorithms, evolutionary strategies, and evolutionary programming are three main branches of evolutionary computation [Fogel 1995]. These techniques are generally similar and apply to a set of possible solutions of the problem. The differences between these methods are concerned with the evolution being modelled. A genetic algorithm models evolution as the changes in gene frequencies, while evolutionary strategies and evolutionary programming model evolution as a process of adaptive behaviour of individuals or species [Back 1995]. Genetic algorithms are more familiar than other evolutionary methods.

Genetic algorithms (GAs) were first introduced by Holland [1975] and have been widely applied as a search technique to find some suitable solution (not necessarily the best solution) for various problems. They have been used in searching for better solutions in function optimisation problems or discovering better solutions in machine learning problems.

With recent advancements in technology and the utility of various parallel architectures, there have been several studies on parallel version of GAs. The

motivations for exploring parallel genetic algorithms are to improve the speed and efficiency of the algorithm by employing parallel architectures, to apply the GA for larger problems, and to follow the biological metaphor more closely by introducing structure and geographic locations into the population [Chipperfield and Fleming 1994].

Many researchers, notably Goldberg [1988], Davis [1989, 1991], [Koza 1992], Michalewicz [1996], Whitley [1989] and Grefenstette [1986, 1989] among others, have studied genetic algorithms and its applications. The aim of this chapter is to provide background information about the algorithm and its application, especially in the field of NP-complete problems, such as task allocation and clustering. These two applications are the targets to be used in the proposed system in this research. Section 3.2 highlights some of the advantages and drawbacks of genetic algorithms. Section 3.3 overviews traditional genetic algorithms and outlines the pseudo-code to show the steps of the algorithm. Discussion about determining the parameter values required for running a genetic algorithm, encoding representations, selection methods, crossover and mutation operators is provided. Section 3.4 describes the hybrid genetic algorithms. Section 3.5 presents a different model for implementing parallel genetic algorithms, including the global model, the island model and the fine-grained model. We have devoted the next section to discussion on the application of genetic algorithms in the field of task allocation and clustering, which are relevant to what we have done in this research.

3.2 Advantages and disadvantages of genetic algorithms

Many advantages are associated with genetic algorithms, including:

- They traverse a search space with more than a single point in parallel. The need to efficiently exploit this parallelism becomes fundamental for systems applied to real world tasks [Kitano 1993].

- GAs operate on several solutions simultaneously. Their ability to maintain multiple solutions concurrently makes them less likely to be influenced by the problems of local optimum and noise [Buckles et al 1990].

- GAs work with encoding of the parameter set (genotype) rather than the parameter set itself (phenotype), which are also easy to implement.
- GAs provide a set of efficient domain-independent search heuristics, which are a significant improvement over traditional search methods, without the need for incorporating highly domain-specific knowledge [De Jong 1989]. Only the objective function and fitness values influence the directions of search. GAs consistently outperform both gradient techniques and various forms of random search on more difficult problems, such as optimisation problems.
- GAs use genetic operators based on probabilistic rules rather than deterministic ones.
- GAs optimise the trade-off between exploring new points in the search space and exploiting the information discovered thus far.

Disadvantage of GAs is in their computational times, which can be implement slower than some other methods. On the other hand, choosing encoding and appropriate fitness functions could be difficult. Other problems are premature convergence and convergence to the local optimum rather than to global optimum. During the following sections, we note to some adopted techniques for improving these disadvantages.

3.3 Traditional genetic algorithm

Genetic algorithms transform a population of possible solutions, each similar to chromosomes associated with a fitness value, into a new population. This is motivated by the expectation that the new population will generate new solutions with increased average fitness. Therefore, solutions will be improved over successive generations. This improvement will be obtained by using operations of reproduction and survival of the fittest and other genetic operations that might naturally occur.

Holland [1975] first proposed the use of GA as a search technique in adaptive systems. He designed a procedure to utilise the operators crossover and inversion. The

crossover operator defined by Holland is known as the one-point crossover. It takes two sequence of strings as parent individuals and exchanges portions of their internal representations, made by a randomly chosen position, among the two parents. Inversion operation was defined as taking a random section of a sequence and reversing its indices. Inversion has been suggested as an important genetic operator by Holland, although it has not been in general found to be useful in practice [Davis 1991]. Later, Fraser proposed an algorithm similar to Holland's genetic algorithm, which operated based on operations crossover, inversion and mutation for operating genetic changes [Fogel 1995].

A typical genetic algorithm starts with a population consisting of randomly created individuals. Three operations usually used in GAs are reproduction, crossover and mutation. Each individual in the population is assigned with a fitness value to use as a measure for evaluation of individual performance in the population, according to an objective function inherent in the problem. The interpretation of fitness in the natural world is the ability of an individual to survive in its environment. Thus, the objective function provides a criterion for selection of those individuals, with higher fitness values more often for the reproduction stage.

During the reproduction phase, each individual is assigned a fitness value, used to direct the selection of individuals towards more fits for doing genetic recombination operators. Each fitness value is a probability, showing the individual's likelihood of being selected, proportional to its fitness relative to other chromosomes in the population. The fittest individuals in the population have a higher probability to being selected and less fit individuals have a low probability. According to the assigned probabilities of reproduction, for each individual, a new population of chromosomes is probabilistically generated by choosing individuals from the current population. The chosen individuals generate offspring by means of genetic recombination operators. The most important recombination operator is called crossover.

The aim of using the crossover operator is to exchange genetic information between pairs or larger groups of individuals. Crossover is a reproduction technique that takes two parent individuals and mates them to produce two child chromosome. The simplest

recombination operator is the one-point crossover, however other recombination techniques are also available [Section 3.3.5]. The crossover operator need not to be performed on all individuals in the population. Instead, it is applied with a frequency specified with the probability of P_c to generate new offspring.

The mutation operation occasionally changes single bits, in a chromosome style representation of individuals, with a small probability P_m. The aim of mutation operation is increasing the genetic diversity of the population by creating new individuals. Mutation is considered to ensure that the probability of searching a particular subspace of the problem space is never zero. The influence of mutation operation on the algorithm is to inhibit the possibility of converging to a local optimum, instead to the global optimum.

Briefly, in each generation, each individual in the population is evaluated in order to determine its fitness in the environment. Individuals for reproduction are selected based on fitness information. The GA then iteratively performs the operations of reproduction, crossover and mutation with the frequencies specified by the respective probability parameters P_r, P_c and P_m on each generation of individuals to produce new populations of individuals until some termination criteria are satisfied.

A typical implementation of a GA is performed for tens or hundreds of generations. Because the GA is a stochastic search method, it is not generally possible to formally specify termination criteria. Depending on the nature of the problem, there exist several stopping criteria for the algorithm. For problems where a perfect solution can be recognised once it is encountered, the algorithm can terminate when such an individual is found. Another common method is to specify the termination criterion in terms of a maximum number of generations to be completed. However, determination of the maximum number of generations does not follow any specific rule. Then, the best individual in the current generation of the population is considered as the result of the genetic algorithm. If the result of the GA is not acceptable in comparison to the problem definition, the GA may be restarted or a fresh search initiated. Termination of the GA may also be triggered when all individuals in a generation are identical, when fitness

values for all individuals are the same, when the difference between any two fitness values is less than some small value, or some other application dependent criterion.

The following pseudo-code outline illustrates the basic components of a simple genetic algorithm.

1. Initialisation: Generate randomly initial population of individuals.
2. Evaluation: Evaluate fitness of individuals in the initial population.
3. Reproduce a new population by applying the recombination operators.
4. Evaluate fitness of individuals in the new population.
5. Preserve the fittest individual so far.
6. Go to step 3 until the algorithm terminates.

Figure 3.1. The outline of a genetic algorithm.

3.3.1 Parameters of the GA

Running a genetic algorithm requires the correct specification of values for a number of parameters, such as operator probabilities, which control the frequencies of recombination operators, and the population size. The performance of a GA can be impacted by poor settings of parameters. Finding settings that work well on a problem is not a trivial task due to limited resources with which one has to solve the problem [Davis 1989].

Davis asserts determining robust parameter settings for population size and operator probabilities, valid for all representations schemes, for all range of possible operators, and for a number of different problems of different types is a hard problem for two reasons: First, we are asking for good values for the parameters, but measuring how good such values are obtained is not trivial. Second, the GAs are stochastic and the same parameter settings used on the same problems by the same genetic algorithm generally yield different results, due to a noisy evaluation function. Consequently, it could take longer to derive parameter values designed for one's problem than the time available for solving the problem itself.

Several strategies have been employed by researchers in the genetic algorithm field to find out good operator probabilities. Davis [1991] classified the techniques for setting good parameters values for GAs into four classes: Carrying out hand optimisation [De Jong 1985], using the genetic algorithm [Grefenstette 1986], carrying out brute force search [Schaffer at al 1989], and adapting parameter settings [Davis 1989, 1991].

De Jong [1985] first solved this problem by choosing a test suite for function optimisation problems to stand in for the full range of possible domains. De Jong, due to noisy evaluation function, derived parameter values for single-point crossover and mutation by hand. He considered constant size populations, at 100 individuals.

Grefenstette [1986] later derived new values for the parameters by using a genetic algorithm, as a metalevel optimisation technique. The result was a new set of parameter settings that outperformed De Jong's settings. Grefenstette's new settings have become a default standard for researchers using binary representation.

Schaffer et al [1989] sampled the possible parameter settings for a range of values, using the De Jong test bed. Their new parameter settings were well for the range of problems, which they had considered. Another important result was that the optimal parameter settings depends on the problem, also population size, the representation technique, the selection procedure, and the method of crossover and mutation operators.

Davis [1989] introduced a technique for adapting the probabilities of genetic operators during the running of a genetic algorithm. The problem of adapting genetic

algorithm parameters in general was simplified, in that the only chromosomal representation technique used was the bit string, and only two operators were considered, mutation and crossover. The background GA differed from the one that Grefenstette used. Davis's technique for setting the parameters involves adapting the operator probabilities, based on their observed performances, as the run proceeds. Periodically, operator fitnesses are adjusted so that the fitness reflects recent operator performance. Using an adaptive mechanism leads to performance slightly inferior to that of carefully derived interpolated parameter settings on the test problems reported in [Davis 1989]. The adaptive technique finds good parameter values quickly and effectively [Davis 1991].

Premature convergence is an important concern in genetic algorithms. This occurs when the population of chromosomes reaches a configuration such that crossover no longer produces offspring that can outperform their parents. Under such circumstances, all standard forms of crossover regenerate the current parents. A suggestion for premature convergence is offered by Schradolph and Belew [1992]. Their method is based on dynamically resizing the range of each parameter. When a heuristic suggests that the population has converged, the minimum and maximum values for the range are resized to a smaller window and the process is iterated. In this manner, dynamic parameter encoding, DPE, can zoom in on solutions that are closer to the global optimum than provided by the initial precision.

The population size affects the performance and efficiency of GAs. Choosing a suitable population size is a key decision for developers of genetic algorithms. If a small population size is selected, the GA will converge too quickly with insufficient processing of search. On the other hand, a large population is more likely to contain representatives from a large number of search spaces. Hence, the GAs can perform a more informed search. As a result, a large population discourages premature convergence to sub-optimal solutions. On the other hand, a large population requires more evaluations per generation, possibly resulting in unacceptably long waiting times for significant improvement. Some research shows that best population size depends on encoding, and on size of encoded string [Goldberg 1989]. Goldberg's computations

have suggested that relatively small populations are appropriate for serial implementations, and large populations are appropriate for parallel GAs.

The crossover rate controls the frequency with which the crossover operator is applied. In each new population, (P_c * population-size) of individuals undergo crossover. The higher crossover rates result in introducing more quickly new individuals into the population. Therefore, if the crossover rate is too high, high performance structures are discarded faster than selection can produce improvements. If the crossover rate is too low, the search may become inactive due to the lower exploration rate [Goldberg 1988]. Crossover rates generally should be high.

After selection, each bit position of each individual in the new population undergoes a random change with a probability equal to the mutation rate P_m. Consequently, approximately (P_m* population-size * length-individual) mutations occur per generation. A low level of mutation prevents any given bit position of individuals from remaining forever to a single value in the entire population. A high level of mutation yields an essentially random search. The probability of mutation should be set at a very small number [Goldberg 1988, Davis 1991].

3.3.2 Objective and fitness functions

The objective function, f, is used to provide a measure of the individuals' performance in the problem domain. If the problem at hand is a maximisation problem, the most fit individuals will have the highest numerical value of the corresponding objective function. In the case of the minimisation problem, the most fit individuals will have the lowest numerical value. The objective function is often inherent in a problem and must be capable of evaluating every individual that it encounters. In many real applications the objective function is likely to be a heuristic that simply indicates which individuals are better than others. In most cases it may not be realistic to use the value generated by the objective function to judge relative differences in fitness, because most objective functions do not yield an exact measure of fitness [Whitley 1989]. The fitness function is then used to assign a fitness value to each possible individual in the

population. The fitness function, F, maps the value of the objective function, for each individual, to a nonnegative number showing the relative fitness. This mapping is always necessary when the objective function is to be minimised or when the objective function can take on negative values [Grefenstette and Baker 1989]. A commonly used fitness function is the proportional fitness assignment. The individual fitness, $F(x_i)$, is computed as the individual's raw performance, $f(x_i)$, relative to the whole population:

$$F(x_i) = f(x_i) / \sum_i^n f(x_i) \qquad (3.1)$$

3.3.3 Encoding

Chromosome representation should contain information about solution, which it represents and highly depends on the problem. It is a mapping that expresses each possible point in the search space of the problem as a character string. Binary encoding is the most used method of encoding, mainly because first works about genetic algorithm used this type of encoding. In binary encoding [Koza 1992] every chromosome is a string of bits, 0 or 1.

Chromosome	11010011010001

Figure 3.2. Binary encoding.

The use of binary strings is not accepted in the genetic algorithm by all researchers, because this encoding is often not natural for many problem. For some problems the binary encoding is in fact deceptive in that it obscures the nature of the search. For some problems integer representation provides a natural way of expressing the mapping from representation to problem domain. For instance, Michalewicz [1996] indicated that for real valued numerical optimisation problems, floating-point representations outperform binary representations.

Permutation encoding can be used in ordering problems, such as the travelling salesman problem [Michalewicz 1996]. In permutation encoding, every chromosome is a string of numbers, which represents numbers in a sequence. One of the possible application domains of this encoding is solving clustering problems by genetic algorithm, genetic clustering.

Chromosome	1 4 3 2 8 6 5 7 9

Figure 3.3. Permutation encoding.

Value encoding can be used in problems where some complicated values, such as real numbers, are used. In value encoding, every chromosome is a string of some values. Values can be anything connected to problem, such as numbers, real numbers or characters to some complicated objects. Value encoding is very good in design of neural network topologies, and finding weights for their connections. Value encoding is applicable to the information retrieval problems.

Chromosome	1.23 0.45 2.95 3.33 5.87 0.67 2.61 1.67
Chromosome	ACBDIJDAEFLICDPLMNDSAIHJC
Chromosome	(back), (back), (left), (forward), (right)

Figure 3.4. Value encoding.

3.3.4 Selection

The individuals participating in the crossover operation are chosen according to a selection method. Selecting individuals based on fitness value is a major factor in the strength of GAs as search algorithms. Grefenstette and Baker [1989] partitioned the selection phase into two distinct processes: The selection algorithm and the sampling algorithm. The selection algorithm assigns to each individual a non-negative number, called the target sampling rate, which indicates the expected number of offspring to be generated from that individual. The sampling algorithm produces a new population by creating copies of individuals based on the target sampling rates. The first part is concerned with the fitness assignments (fitness function). The second part is the probabilistic selection of individuals for reproduction based on the fitness of individuals relative to one another.

There are many methods for selecting the best individuals, for example roulette wheel selection [Davis 1991], elitist selection [Grefenstette 1986], tournament selection [Michalewicz 1996], rank selection [Backer 1985], steady state selection [Whitley 1989] and others.

Roulette wheel selection is a popular approach for implementing selection [Davis 1991] which randomly copies individuals from the current generation into the next generation with a probability proportional to their fitness, $F(x_i) = f(x_i) / \sum_i^n f(x_i)$. In other words, each individual in the current population is reproduced a number of times proportional to its performance. The better individuals have more chances to be selected more times. A parent is then randomly selected based on this probability. The roulette wheel selection method is stochastic sampling with replacement.

A second method, called deterministic sampling, assigns to each individual, i, a value C_i = RND (F_i * pop-size) + 1 (RND means round to integer and pop-size means population size). The selection operator then assures that each individual participates as a parent exactly C_i times [Buckles et al 1990].

One problem with selection proportional to fitness is that this method cannot ensure convergence to a global optimum [Rudolph 1994]. Grefenstette [1986] employed a heuristic, known as elitist selection, to resolve the drawback of selecting parents in proportional selection. Elitist selection always copies the individual with the best performance or a few of the best individuals unaltered from the current generation into the next generation, created by crossover and mutation. This procedure guarantees asymptotic convergence but at various rates depending on the problem [Grefenstette 1986]. In the absence of such a strategy, it is possible that the best individual in the population lost, due to sampling errors, or applying crossover and mutation operators. Elitism can very rapidly increase performance of the GA, because it prevents the disappearance of the best found solution.

Another approach to implementing the selection operator is ranking. Backer [1985] first reported experiments where reproductive trails were allocated according to the rank of individuals in the population rather than by the individual fitness relative to the population average. In ranking, the population is sorted in increasing order by fitness values. That is, the worst individual is assigned to Min, the best is assigned to Max, and the target sampling rate for each of the other individuals is interpolated according to its rank. The main idea behind introducing the ranking is that the premature convergence is caused by the presence of super individuals, which are much better than the average fitness of the population. Such super individuals have a large number of offspring and due to the constant size of the population, prevent other individuals from contributing any offspring in subsequent generations. Hence, in a few generations a super individual can eliminate desirable chromosomal material and cause a rapid convergence to possibly local optimum [Michalewicz 1996]. The primary advantage of ranking over proportional selection is that the algorithm is less affected by the premature convergence caused by individuals that are far above average.

Whitley [1989] also used a rank based genetic algorithm, GENITOR, and argued that allocating reproductive trails according to rank is superior to fitness proportional reproduction. He states that selective pressure and population diversity should be controlled as directly as possible. Calculating fitness as a function of rank is an effective way to obtain a greater degree of control over selection pressure. Allocating

reproductive trails according to rank can be used to speed up genetic searching. Allocating reproductive trails according to rank prevents scaling problems (lacking the selective pressure and premature convergence), since ranking introduces a uniform scaling across the population. Ranking acts as an objective function transformation that assigns a new fitness value to an individual based on its performance relative to other genotypes.

Steady state selection is a reproduction technique, which replaces only one or two individuals at a time rather than all individuals in the population. Some of the low fitness valued individuals are removed and the new offspring, which are generated from the high fitness valued of parents, are put in their place. The rest of the population survives to a new generation. This technique was first described by Whitley [1989], as a one-at-a-time reproduction scheme used in his GENITOR algorithm. GENITOR is a rank based genetic algorithm which the generational approach and reproduces new individuals on an individual basis. GENITOR only produces one new individual at a time. This reproduction scheme has also been described in Syswerda [1989], where it is called steady state reproduction. The generational replacement is the special case of steady state reproduction, in which the number of new individuals created equals the size of the population [Davis 1991]. The main difference between steady state GAs and generational replacement algorithms is that within each generation, only a few members of the population are changed [Syswerda 1989]. Some researchers have found that steady state reproduction is inferior to generational replacement [Davis 1991]. Another technique that leads to improved performance is called steady state without duplicates, Davis [1991]. However, this technique is not worthwhile considering for real time applications due to the additional execution time spent by the algorithm using this technique.

An additional selection method is called tournament selection [Michalewicz 1996]. This method, in a single iteration selects some number of individuals and selects the best one from this set of elements into the next generation. This process is repeated for the size of population times.

3.3.5 Crossover and mutation operators

The crossover operation exchanges sub-strings between two individuals with crossover probability p_c for creating new offspring, with fitness values higher than their two parents. New offspring are indeed new points in the search space that is to be tested. Crossover is a basic operator in GA. We quote the following sentences from [Davis 1991] in supporting the importance of crossover operation. 'Crossover is an extremely important component of a genetic algorithm. Many genetic algorithm practitioners believe that if we delete the crossover operator from a genetic algorithm the result is no longer a genetic algorithm. Genetic algorithm researchers have shown that when crossover is deleted from a genetic algorithm, performance is degraded on a variety of problems.'

There are many ways to do crossover for problems, especially for problems using the binary encoding, such as one-point crossover, two-point crossover, multi-point crossover and uniform crossover. There are three principal choices for permutation problems: Order crossover, PMX (partially matched) crossover and cycle crossover [Buckles et al 1990].

Crossover methods can be categorised according to two kinds of biases that affect the exploratory power: Positional bias, and distributional bias [Eshelman et al 1989]. One-point crossover selects a position along the length of individual at random with a uniform distribution and exchanges the segments to the right of this position between two parents to produce two offspring. One-point traditional crossover has a strong positional bias but it is unbiased with respect to the distribution of material exchanged.

Two-point crossover considers the chromosome string as a ring. It selects two unique points at random along the length of the string and divides the ring into two segments. The segments are exchanged between the parents to produce two offspring. Traditional two-point crossover reduces positional bias without introducing distributional bias. Two-point crossover has been shown to be consistently better then one-point crossover.

Uniform crossover [Syswerda 1989] exchanges bits rather than segments. For each bit position in the string, the bits from two parents are exchanged with fixed probability.

Uniform crossover has not positional bias but has a strong distributional bias. It has been shown that in almost all cases of function optimisation problem, uniform crossover is more effective than either one or two-point crossover. However, there are some exceptions [Eshelman et al 1989].

Multi-point crossover is the natural extension of two-point crossover. Increasing the number of crossover points reduces positional bias but it also introduces some distributional bias. Experimental results of Eshelman et al [1989] indicate that it is better to have more distributional bias toward higher disruption rates and less positional bias in crossover operations.

The mutation operator offers the opportunity for new genetic material to be introduced into a population. The new genetic material does not originate from the parents and is not introduced into the child by crossover. Rather, it occurs a small percentage of the time after crossover. Mutation is implemented by one of two methods [Fogel 1995]: By considering each element of each individual in the population and a certain probability, either flipping it to the complementary value or replacing it by a randomly chosen element with a certain probability. Both methods of mutation implementation can be made equivalent by adjusting the mutation rate. Mutation is employed to ensure that all bits have some probability of entering the population. It assures that, given any population, the entire search space is connected.

Inversion operates on a single individual. It changes the order of elements between two randomly chosen points in an individual without changing their meaning. Inversion attempts to reduce the error rate and increases the combination rate of crossover. With uniform crossover, use of inversion is not necessary and will not have any effect [Syswerda 1989]. Davis claims inversion has not been found to be useful in practice [Davis 1991]. However, some researchers believe inversion is a useful operator for longer individuals.

3.4. Hybrid genetic algorithms

One of the problems that traditional genetic algorithms are involved with is the premature convergence to local optima, otherwise a long time may be required for the GA to find an optimal or near to optimal solution. Some methods for overcoming this problem and increasing the efficiency of GAs, which depend on the selection algorithm or parameter settings have been discussed in previous sections. Another method is based on the incorporation of problem-specific knowledge to direct the blind search of the GA to the fruitful regions of the search space for improving the efficiency. The resulting algorithm is referred to as a hybrid genetic algorithm [Goldberg 1988, Davis 1991]. Hybridising the GA with other domain-based heuristics, such as hill climbing, and simulated annealing can produce an algorithm better than the GA. There are many reports in the literature on using hybrid GAs for various problems. For example, Mansour [1992, 1991], Booker [1987], Grefenstette [1987], Schoneveld et al [1997], Muhlenbein [1989] have generated a number of successful optimisation algorithms.

3.5 Parallel genetic algorithms

GAs are inherently parallelisable and many researchers have studied techniques for implementing them on multiprocessor machines. These studies suggest that even if we are not using a parallel architecture, we may enhance the performance of GAs if we simulate parallel runs by maintaining multiple populations and controlling the interactions among them carefully [Davis 1991]. Several approaches have been proposed for distributing the required computations in GAs that are classified around three approaches: The global model [Chipperfield and Fleming 1994], the island (migration) model, and the fine grained (diffusion) model [Stender 1993]. Most of the major differences are encountered in the population structure and the method of selecting individuals for reproduction. These categories reflect the different ways in which parallelism is exploited in the GA and the nature of the population structure and recombination mechanisms used.

3.5.1 Global model

The global model [Chipperfield and Fleming 1994] treats the entire population as a single breeding unit and aims to exploit the inherent parallelism in the GA. A dedicated processing node, called farmer, initialises and holds the entire population, performs selection and assigns fitness to individuals. Other nodes, workers, apply the crossover and mutation operations and evaluate the objective function for the resulting offspring. In computational terms, a number of identical processors independently perform genetic operators and objective function evaluations on a population stored in a shared memory. This method, in cases where the objective function is computationally expensive, is a useful method of reducing the execution time of the GA, because most of the time spent by a genetic algorithm is spent in function evaluation [Davis 1991]. In particular, when the objective function is of low computational cost, then there is potentially a bottleneck at the farmer while fitness assignment and selection are performed. Goldberg [1989] describes a similar scheme, synchronous master-slave using a hybrid GA. He also suggests two alternative models to overcome this drawback: Semi-synchronous master-slave and asynchronous, concurrent GA [Goldberg 1989].

3.5.2 Island model

In the island model, a large population is distributed into many smaller isolated sub-populations evolved in parallel, and periodically each sub-population swaps its worst individual(s) with the best one(s) of its neighbours. Each sub-population uses local selection and reproduction rules to locally evolve the species. The pattern of migration of individuals limits how much genetic diversity can occur in the global population. Additional routines are included to exchange individuals between sub-populations according to the communications topology employed. This model is based on the hypothesis that several competing sub-populations could be more search-effective than a wider one in which all the members were held together. The island model is utilised on MIMD machines [Stender 1993].

Grosso, in 1985, first introduced an island model and founded improvement in the performance of the GA in terms of the quality of solution [Chipperfield and Fleming 1994]. He also asserted that limited migration of individuals between sub-populations was more effective than either complete sub-population interdependence or independence.

Later, Petty et al [1987] presented a migration GA influenced by earlier work done by others. Pettey's parallel genetic algorithm (PGA), consists of a group of identical nodal GAs (NGA), one per node of the Intel iPSC message passing multiprocessor system, with ncube interconnection topology. An NGA communicates with its neighbouring NGAs in each generation by sending the best individual in the local population to each neighbour's population. The insertion of new individuals from neighbours into the local population can be done by replacing randomly chosen individuals, the worst individuals, or the individuals most like the incoming individuals (i.e., the smallest Hamming distance) strategies. They tested the worst individual replacement policy. Petty et al found a significant performance improvement for numerical optimisation problems selected from De Jong's test-bed functions. They reason that in a sequential GA an individual is selected based on its performance in the whole population but in a PGA an individual is selected based on its performance in its local sub-population. This difference in selection could result in premature convergence. On the other hand, this difference might lead to a slow down of the convergence and ultimately produce better results. Another result was that the population size for a PGA should be set with the optimal population size in mind.

[Tanese 1987, 1989] also worked on an island GA model, where each processor runs the GA on its own sub-population and periodically replaces bad individuals in its own sub-population with neighbour's good individuals. The neighbour with which this exchange takes place will vary over time: Each exchange will take place along a different dimension of a hypercube topology. He used two new parameters to specify the frequency of exchange, at which generation migration should take place, and the migration rate, which is the number of individuals exchanged between sub-populations.

In [Tanese 1987], it is assumed that individuals are probabilistically selected for migration from the subset of the sub-population, whose fitness was at least equal to the

average fitness of the sub-population. Likewise, individuals were selected for replacement by immigrants probabilistically from the subset of individuals whose fitness was no greater than the average for that sub-population.

Later, Tanese [1989] developed a parallel genetic algorithm, called the distributed genetic algorithm, and adopted a new strategy. At a migration generation, each sub-population produces extra offspring than the current sub-population size. The migrants were then uniformly selected from the offspring and removed from the sub-population, to maintain the correct sub-population size. When a sub-population receives migrants from another sub-population, it selects individuals to be replaced by incoming migrants selected randomly from its sub-population. The results presented by Tanese showed a near linear speed up when compared against a sequential GA with a population size equal to the sum of the individual's sub-populations. The distributed GA achieves higher average fitness for the entire population, without detracting from its ability to find filter individuals than the traditional genetic algorithm.

3.5.3 Fine grained model

An alternative model of a distributed GA is the diffusion GA. In fine grained (or diffusion) model, a single population evolves, each individual of which is placed in a cell of a planar grid. Selection and crossover are applied only between neighbouring individuals on the grid, according to a pre-defined neighbouring structure. The use of local selection and reproduction rules lead to a continuous diffusion of individuals over the population. The diffusion GA is usually utilised on SIMD parallel machines and array processors.

Spiessens and Manderick [1989] have proposed a fined-grained parallel genetic algorithm, FG-algorithm, on DAP [Trew and Wilson 1991], a massively parallel computer. An extra parameter denotes the size of the neighbourhood. In the FG-algorithm, the individuals of the current population are put on a plannar grid. Each individual is assigned with a fixed-size neighbourhood, by an extra parameter. For instance, if range equals one then the neighbourhood of a given individual consists of

nine individuals (including the individual itself). The next generation is calculated as follows: First, replace each individual by an individual selected from its neighbourhood on the grid. Second, do the crossover operation for each individual with a randomly selected mate from its neighbourhood with a certain probability, and third, do the mutation operation for each individual depending on the mutation rate. The neighbourhood experiments indicate that FG is able to obtain satisfactory results with moderate sized neighbourhoods.

Spiessens and Manderick [1989] also compared the performance of the FG-algorithm to Robertson's algorithm [1987] for the Connection Machine [Hills 1985]. The Robertson algorithm is also fine-grained. Each individual is put on a separate processor but it uses global selection and random mating. Consequently, Robertson's algorithm has to make heavy use of communication, whilst the FG-algorithm requires local communication.

3.6 Applications of genetic algorithms

Genetic algorithms have been successfully applied to many problems, such as practical optimisation problems, task allocation, clustering, neuronal network design, classifier systems, and machine learning [Carbonell 1989]. Their usefulness has been proved not only by comparison with other traditional search methods, but also by a large number of practical applications in domains such as biology, chemistry, physics, medicine and computer aided design [Back 1995]. In the following subsections, we briefly review task allocation problem and clustering for the purpose of relevancy to work in this research.

3.6.1 Task allocation

The task allocation problem consists of partitioning a problem into tasks (sub-problems), and allocating these tasks into processors of a given distributed memory machine so that an objective function is minimised, and the optimal allocation of tasks

is found. The quality of an allocation is measured by the execution time of the application, which depends on communication and calculation components. Finding the global optimum is a NP-hard problem. This problem can be solved statically or dynamically [Shirazi et al 1995]. Several cost models can be found in the literature [Chu et al 1980, Efe 1982]. Many researchers have used genetic algorithms as a heuristic to solve this problem. For example, Mansour and Fox [1991, 1992] introduced a hybrid genetic algorithm for task allocation problem in distributed memory multicomputers, which used the following cost function as its objective function.

$$f = \gamma \sum_n W^2(n) + \beta R \sum_k \sum_l C(k, l) \qquad (3.2)$$

Where R is a machine-dependent communication to calculation time ratio, t_{comm}/ t_{float}. γ and β are scaling factors expressing the importance of the calculation term and the communication term, respectively. W(n) is the number of data elements allocated to processor n. C(k, l) is the Hamming distance between processors k and l.

This function can be rewritten as the following function [Schoneveld et al 1997]. In this function, the metric for deciding on the quality of a task allocation is the execution time.

$$H = \beta \sum_i W_i^2 + (1 - \beta) \sum_{k>l} J_{kl} (1 - \delta_{kl}) \qquad (3.3)$$

Where, $i \in \{1, ..., P\}$, and P is the number of processors. J_{kl} is the communication time between processors k and l. W_i is the total execution time on the processor i. The parameter $\beta \in [0, 1]$ is used to tune the amount of competition between the calculation and the communication terms.

The fitness of an individual is evaluated by the objective function and represents the cost of task allocation, and required to be minimised.

An allocation of tasks to processors is coded as a sequence, where each letter in the sequence corresponds to a task, while the letter in the sequence corresponds to the processor allocation number of the given task.

3.6.2 Clustering

The primary objective of cluster analysis is to partition a given set of objects into a fixed number of clusters such that objects within a cluster are more similar to each other than objects belonging to different clusters. Clustering is very problem-oriented in the sense that cluster algorithms that can deal with all situations are not yet available. The best known and most widely used algorithm is the K-means algorithm or the Isodata algorithm. Lately, neural networks [Simpson 1992], for example competitive-learning networks [Intrator 1995], self-organising feature maps [Kohonen 1990], and adaptive resonance theory (ART) networks [Carpenter and Grossberg 1988] also have been used to cluster data. In order to mathematically identify clusters in data, it is usually necessary to first define a measure of similarity or proximity, which will establish a rule for assigning patterns to the domain of a particular cluster. The measure of similarity is problem dependent. Generally, a proximity is defined either by a similarity measure or a dissimilarity measure. The former increases in value when two data points resemble each other, whereas the latter increases in value when two data points differs from each other.

Solving this problem is difficult because the number of partitions of N objects into K clusters increases with K^N [Hartigan 1975]. In special cases, exact solutions are possible in polynomial time, but in general near-optimal solutions must be found using heuristics. GAs have been used as a technique for solving partitioning problem by many researchers [Maulik and Bandyopadhyay 2000]. To solve this problem with GAs, one must encode partitions in a way that allows their manipulation by genetic operators. Jones and Beltramo [1991] considered three encoding methods. The first encodes a partition as an N-string whose ith element is the group number assigned to object i. The second method encodes a partition as a permutation of N objects and K-1 group separators. The third method encodes a partition as a permutation of the N objects and decodes this permutation with a greedy adding heuristic. The greedy heuristic uses the first K objects in the permutation to initialise K groups. The remaining objects are then added to these groups in the order they appear in the permutation, always adding the object to the group that yields the best objective value. This method requires more

knowledge about the objective function. In their experiments, the permutation encoding with greedy decoding and PMX crossover performs better than the others, because it incorporates additional problem knowledge.

GAs can outperform some standard heuristics on some problems. Jones and Beltramo compared GA to switching heuristic [Hartigan 1975]. They found that a GA outperforms the switching clustering heuristic.

3.6.3 Automated theorem proving

Automated theorem proving (ATP) aims to use computer technology to automatically determine whether or not a given query is a logical consequence of a set of axioms [Bledsoe and Loveland 1984]. Attempting to prove the validity of such results typically requires traversing in a very large search space. Therefore, it is said that ATP essentially amounts to solving search problem: The set of all objects which can be derived from an input problem by means of the inference rules forms the search space [Ibens 1999]. Automated theorem provers, like many of the developed artificial intelligence techniques, rely on heuristic search through large space of possible inferences. Advances in parallel and distributed computing can potentially offer improvements in performance of various search strategies including depth-first, breadth-first, and A* search [Cook and Hannon 1998]. Parallelisation of different approaches in theorem proving can be generally divided into two categories: AND-parallelism and OR-parallelism [Stenz and Wolf 1999a].

Different search strategies may behave significantly different on a given problem. In general, it cannot be decided which strategy is the best for a given problem. Therefore, one is encountered with the difficult problem of selecting a good strategy that increases the effectiveness of the reasoning program for a given task.

Stenz and Wolf [1999a, 1999b, 2000] have suggested a method for strategy selection problem, called strategy parallelism. Strategy parallelism, where a proof task is performed in parallel by a set of competitive agents with different strategies and

distributed resources, is a concept for applying parallelism to automated theorem proving. They have investigated a competition approach, where different strategies are applied to the same problem and the first successful strategy stops all others. Different, competitive proof agents traverse the same search space in parallel via different, ideally non-overlapping paths. They introduced a hybrid genetic algorithm, genetic gradient approach, that can be used to optimise a prover performance. The optimisation problem, strategy allocation problem, can be formulated as follows:

Given a set of training problems, a set of usable strategies, a time limit, and a number of processors. The problem is the determination an optimal distribution of resources such as time and processors to each strategy, i.e., a combination of strategies which solves a maximal number of problems from the training set within the given resources.

In their approach, the individuals are represented by prover schedules [Stenz and Wolf 1999a]. The initial generation is created randomly and the fitness measure is the number of problems in the training set solved by an individual. They used the survival of the most successful individuals of each generation (elite selection). Based on a kill-off-rate parameter a predetermined portion of each generation is died before reproduction and the survivors make up the first part of the succeeding generation. The selection strategy is followed by filling the remaining slots applying the crossover operator to the survivors. Only new generated individuals by crossover operation are subject to mutation with a certain probability. The number of generations is given by the user.

The gradient method in combination with the genetic algorithm can be used to eliminate runaway strategies from schedules by transferring their time resources to more successful ones. In their approach, the best resulting schedule founded by the genetic algorithm is then refined by a gradient optimisation technique.

They compared the results of the gradient method and the genetic algorithm. The genetic algorithm performs better. Their experimental results show only a poor scalability for the prover system. The attributes of the initial generation that are selected

at random strongly influence the overall results of the experiment. The deficiencies of an unfit initial generation can not be wholly remedied by the subsequent optimisation.

In [Stenz and Wolf 1999b] they combined the approaches of genetic algorithm and feature based problem analysis in a three-phased process: In the first phase, all given problems are divided into a small number of classes according to some simple discrimination criteria. In the second phase, for each of these classes, a set of schedules is evaluated by the genetic algorithm. The best of these schedules are selected for refinement in the third phase by applying the gradient method.

In order to evaluate the potential of the method, they have implemented the strategy parallel theorem prover e-SETHEO [Stenz and Wolf 1999b]. 545 problems are considered as the training data set and 91 different strategies as the strategy set. They extracted all these strategies and ran each strategy on all problems. The successful results of all runs were collected in a single list that became the database for the genetic gradient algorithm. Then they ran the genetic gradient algorithm on the collected data.

All the results indicated that the genetic gradient approach is extremely useful for automatically configuring a strategy parallel theorem prover. The results obtained by their automatic prover configuration can actually be considered a success.

3.7 Summary

In this chapter, we discussed the genetic algorithm as a stochastic search method, which is applicable to a broad spectrum of activities. Many of the variations on the original GA have been discussed, such as different representation and selection strategies, hybrid GAs and parallel implementation of GAs. Some of these techniques have aimed to minimise the likelihood of premature convergence. They are based on selection schemes, such as ranking, for reducing the stochastic sampling errors, and other techniques have been incorporated into the reproduction scheme to control the level of competition among individuals and to maintain diversity. The use of two point

crossover operators has been suggested for enhancing exploration and improving the search. Other techniques have been developed to increase the efficiency of the GAs.

The application of GAs to the task allocation problem in distributed memory MIMD systems, clustering and strategy parallelism in automated theorem proving has been described in more detail.

4

A Distributed Problem Solving System

4.1 Introduction

Within distributed AI, a typical system is based on parallel and distributed processing and different subsystems work together to form groups that achieve more than the individual entities can achieve. Indeed the roots of distributed AI are actually not far from brain theory [Durfee 1995]. The main difference is that distributed AI has focused on coarser grain collections of agents or problem solving nodes, whereas in the brain finer grained levels of intelligence and connectionist neural computing approaches are evidenced.

In AI applications, rule based expert systems have emerged as an important tool in problem solving systems. Many problems are naturally formulated in terms of rules and rule based systems have the capability of rapid modification [Morgan 1988]. Expert systems are generally designed to be expert in one problem domain. One way of scaling expert system technology to more complex problem domains is to develop co-operative interaction mechanisms that allow multiple expert systems to work together to solve a common problem. This approach is motivated by the operation of the brain, where

different neural subsystems combine and exchange signals to work together to yield an overall intelligent nervous system.

Furthermore, every knowledge based system must rely on search techniques and virtually all systems use search techniques for problem solving. Indeed problem solving is the generation of conclusions that are not explicitly represented, but are implied by the knowledge that is explicitly represented. In the search paradigm, each problem solving step requires an evaluator to select the next state. Recently, genetic algorithms, which use principles of evolution through natural selection to solve problems, have been established as powerful searching and optimisation techniques. Genetic algorithms conduct search by stepwise modification of a population of individuals, being candidate solutions of the given problem. The search space of a problem can be metaphorically considered as a fitness landscape on which genetic operators navigate the search space. A fitness landscape is defined by the finite set of candidate solutions, a fitness function which assigns a value to each solution, and a neighbourhood relation specified by a modification operator [Slavov and Nikolaev 1999].

In the following sections, we describe a proposed *distributed problem solving system* (DPSS) in terms of its behaviour and the structure of a parallel machine suitable for its implementation. The DPSS can be viewed as a collection of distributed agents which cooperate to perform automated reasoning about the domain knowledge. Many concurrent reasoning tasks may proceed simultaneously. Since the domain knowledge is distributed, there is no global view for agents, and each agent has its own knowledge of the domain expressed in production rule representation. When a problem solving task is assigned to the collection of agents, each agent begins working towards a solution based on its local knowledge. However, since reasoning tasks are distributed amongst the agents, each agent has incomplete knowledge and cannot solve the reasoning task without acquiring further knowledge through communication with appropriate agents.

The system proposed here is in fact a knowledge-based system, which consists of two components: A knowledge base for storing information about the appropriate domains, and an inference mechanism for the manipulation of its knowledge. We initially explain the mechanism of inference in Section 4.4 and then discuss the

structure of the knowledge base in the following sections. Three stages of initialising, clustering, and integrating are identified during the inference process. Pseudo-code, which describes the run time processing cycle for each problem-solving agent in the system, is presented. The detail of the developed parallel hybrid genetic algorithm, used to compose relevant problem solving nodes into clusters, is discussed in Section 4.9. Section 4.10 presents the computational model that describes the behaviour of the system. The final section is devoted to the design activity of the proposed system and presents the developed structure chart and knowledge diagram.

4.2 Goals of the system

We consider the proposed system as a multicomputer with distributed memories (MIMD) such that processors in the network work in parallel on parts of large and complex problems in parallel to quickly solve a given problem. The processing nodes communicate with each other through message passing. Due to advances in message passing computers, which have reduced the communication and message passing overheads, these computers have become feasible candidates for implementing production systems. The suitability of message-passing architectures for production systems has been shown by [Gupta and Tambe 1988]. Several goals have been assumed for the system:

- Increasing the variety of solutions by allowing processing nodes to form local solutions.
- Increasing the solution creation rate by forming sub-solutions in parallel.
- Improving the use of individual node expertise by allowing processing nodes to exchange their partial solutions.
- Reducing the communication overheads by allowing nodes to exchange their partial solutions within determined clusters.

4.3 Architecture of the system

In a loosely coupled distributed system, an agent spends most of its time in computation as opposed to communication. Since problem solving is inherently computation intensive, we have chosen a loosely coupled implementation for our distributed problem solver. Each problem solving agent spends most of its time performing match-select-fire cycles. Therefore a suitable architecture should support an increase in processor speeds to shorten the execution times required for the implementations of parallel production systems. The architecture also needs to be scalable for new and complex applications to increase the available parallelism in production system. Figure 4.1 illustrates the architecture of the DPSS as a block diagram, showing the major subsystems and the interconnections between them. There are several major subsystems, including the user interface, secondary storage, controller, switching network, and an array of processing elements (PEs). Information flow between the subsystems is shown by the arrow lines connecting these systems.

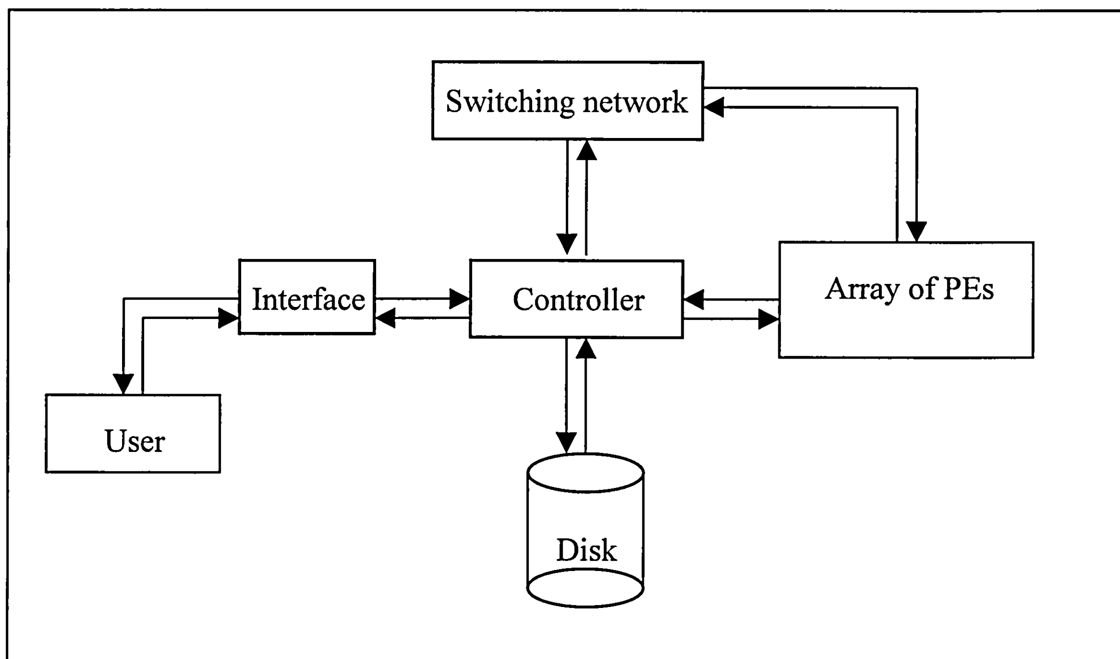


Figure 4.1. Schematic representation of the system.

The user interface is a tool and provides facilities for interaction, enabling the user to query, modify, and display the knowledge base. Queries are the means used for information retrieval in database systems, whilst in AI oriented systems they can cause some modifications, including additions and deletions of content of a knowledge base. In commercial database systems, there are facilities to provide data manipulation, which involve insertion, deletion, and retrieval of data. These facilities are embedded in languages like SQL (structured query language) [Denn 1985]. Similar facilities are offered by knowledge based tools. For example, Knowble [Mylopoulos et al 1993] provides essentially three basic functions: TELL to create a fact, UNTELL to remove it, and RETELL to update the knowledge base.

Secondary storage is a separate entity of the parallel architecture and is used to store the large knowledge base of the system. We consider the system as a general problem solving system that should be able to deal with different problems. Furthermore, a complex problem concerns different problem domains, and since a problem requires domain specific knowledge, hence the system must be provided with a global knowledge base, including the various domains' knowledge. Storing the large knowledge bases in distributed memories is almost impossible, because local memories are small in size and expensive per byte. Thus, it is necessary to store the knowledge base on disks as an auxiliary memory (secondary storage), which have more capacity.

The controller, as a management system, does the following jobs in the proposed system:

- It has overall control of operations related to the disk unit. These operations include identify, retrieval, transfer, and mapping of the relevant portions of the knowledge base from the disk to the array of PEs.

- It accepts a request from the user via the interface and tries to discover the appropriate knowledge from the collection of data stored on the disk unit.

- It transfers the relevant portions to the switching network. The switching network is set by the controller to assist in the mapping operation. It is assumed that the mapping operation is based on allocating a rule-set to a PE (the structure of the knowledge base will be discussed later).
- It keeps the knowledge base up-to-date by accepting new knowledge, or replacing the old knowledge with the new knowledge.
- It controls and initialises the parallel propagation of messages (data packets).
- It has overall control over the inference mechanism and knowledge manipulation.
- It integrates the results and returns them to the user.

Processing nodes are set in a scalable two dimensional mesh network topology and controlled in hardware by the control unit. Each PE has direct connection to its four neighbours. A system bus will be employed to provide the communication paths between PEs themselves and between the controller and PEs. The PEs are the heart of the system, providing intermediate results that are stored in their local memories. Each local memory will be large enough to accommodate data and the appropriate software for the inference mechanism. Registers provide temporary storage for data. Random Access Memory (RAM) provides longer term primary storage for data. Each PE also contains the CPU logic and an input/output (I/O) interface for exchanging data. I/O ports provide access to peripheral devices, such as disk storage. The ALU carries out arithmetic and logic operations (utilised for knowledge manipulation) at times specified by the control unit.

4.4 Problem solving methodology

The adopted methodology for solving problems in DPSS is based on co-operation of individual problem solving agents, whose local processing results should be exchanged to produce an integrated solution. Each agent in the DPSS is assumed to be a production rule based system, called a Problem Solver (PS), which can access a subset of the

appropriate portions of the knowledge base. Consequently, each PS can infer an intermediate result from the production rules and the facts that it knows. As they progress in their problem solving, their initially identical working memories information will begin to diverse. Thus, nodes must exchange locally generated partial results with nodes that could be working toward related results. The exchange of information introduces communication overheads. We have applied a selective approach, based on the implementation of a genetic algorithm, to find clusters of relevant nodes with an optimised value of the objective function. The objective function provides estimation for the parallel computing overheads. After determination of clusters' configurations, nodes communicate with each other only within relevant clusters.

Three stages of knowledge processing in the network can be distinguished during problem solving; including initialising, clustering, and integrating.

Initialising: After issuing a request, and according to the given information by the query, the appropriate portions of knowledge is identified and retrieved from the knowledge base on the disk unit. We assume that the identified knowledge is already partitioned into distinct sub-sets (rule-sets) and distributed, based on mapping 'a rule-set onto a processor' in the array.

The controller performs the operation of broadcasting the query to the array of processing elements i.e. multiple copies of the input are made and distributed across all processing nodes. Hence, all working memories of PSs initially contain identical elements.

After performing the operations of query broadcasting and knowledge mapping, PSs are invoked and begin to process their information to find out partial results at the finest level of granularity for the given complex problem. In this stage, each PS asynchronously performs the match-select-fire cycles on its own data until no rules can be matched and fired with existing data in its working memory. No message passing communication is assumed during this stage.

Clustering: Following completion of the first stage and producing partial results by PSs, the clustering stage begins to perform with the aim of grouping those PSs that are

relevant to each other in some sense. This stage indeed comprises of two sub-stages. First, performing the clustering process to specify an optimised configuration for clusters of PSs. Second, executing communication-computation cycles within specified clusters.

The relevant nodes within a cluster are called neighbouring nodes. After determination of processor clustering structures, neighbouring nodes in a cluster then exchange the local partial solutions to other neighbouring nodes in that cluster. This process is performed in parallel for other clusters. The main objective of cluster analysis is to search for an optimised configuration of groups of PSs such that each group can provide appropriate results for some aspect of the given problem, and at the same time the cost of communication overhead in the system is minimised. We have considered an objective function as a similarity measure in clustering process, which refers to the cost of communication and computation overheads resulted from processing nodes' cooperations. The reason for choosing such an objective function is that the efficient utilisation of the computational power of distributed memory parallel computers is highly dependent on how the composite calculation-communication workload is distributed amongst the processing nodes.

We have also assumed that the human expert (system designer) knowledge can be used as a heuristic search method to guide the searching process of clustering so that each cluster eventually can provide an appropriate solution in some region of the state space, and combination of these approximations can be sufficient to yield a globally satisfactory solution.

We have proposed a parallel hybrid genetic algorithm (PHGA) to perform cluster analysis. Each PS in the system deals with its own population and independently performs the genetic operations of selection, crossover, mutation, and inversion on individuals in its own sub-population. The relevancy of any pair of rule-sets is defined to be judged by the human expert (system designer). An expert is a person who is sufficiently experienced to make consistent meaningful judgements of relevancy in the context of a given domain. The expert knowledge consists of the data dependency relationships between agents and interference between rules in different rule-sets, which

has been analysed at compile time. In general, rule interactions in distributed production systems are resolved at compile time, because run time analysis is expensive. We assume a prior knowledge is not available for large knowledge based systems.

The role of natural selection is to selectively form the best configuration of PS clusters, such that the processors having appropriate relations clustered to groups and simultaneously inter node communications are minimised. Therefore, at the end of this stage, we expect to have clusters of PSs, which are made by process of selection and vary functionally. In general, we are not interested in finding in the absolute similarity values of problem solvers, but in discovering a configuration that best preserves an acceptable balance between the relative ordering of relevancy indicated by the expert, and the communication and computation overheads in the network. The number of clusters depends on important features of the input and can be determined by the expert. Performing the PHGA leads to a specification of the structure of clusters and each PS in each cluster communicate with their neighbouring PSs. Therefore, in each cluster the knowledge processing continues sequentially through the communication-computation cycles, until no further processing is possible in each cluster.

PSs clustering in this particular problem can be interpreted as a task allocation problem in the system. Each cluster plays the role of a complex agent, with the ability to produce coarser grain solutions, and clustering of PSs can be viewed as the process of allocating PS-tasks to these complex agents.

Integrating stage: In our distributed problem solving network, each computing agent is a semi-autonomous problem solving node that can communicate with a subset of other nodes. To improve their local information, nodes must share subproblem solutions. In a distributed system, integration of partial results to achieve an overall solution is one of the grand challenges of AI applications [Kitano et al 1993]. In the clustering stage, PSs interact with neighbouring nodes to find solutions for different aspects of the problem. The solutions are constructed through the computation-communication cycles by incremental agreement of mutually constraining or reinforcing partial solutions in clusters. Herein, due to assumption that each cluster of relevant agents can respond well to a feature of the complex problem, an overall

solution can be integrated by combination of the approximations obtained from the clusters. Combination is related to the distributed database technology and is performed by the operation of UNION on clusters' partial results.

So far we addressed the problem of a strategy for a network of PS co-ordination. Corkill and Lesser [1983] state three conditions for any network co-ordination policy to be successful:

(1) Coverage: Any given portion of the overall problem must be included in the activities of at least one node.

(2) Connectivity: Nodes must interact in a manner which permits the covering activities to be developed and integrated into an overall solution.

(3) Capability: Coverage and connectivity must be achievable within the communication and computation resource limitations of the network.

The clustering approach specifies a possible coverage and connectivity pattern that can potentially satisfy the capability condition. Genetic clustering provides an adaptive framework, which is strongly modified by feedback from the environment (expert). The advantages of this approach are described as follow.

The process of formation a cluster takes place in parallel with other clusters, and then PSs work concurrently in clusters to generate an overall solution. Therefore, we obtain an overall solution faster. Problem solving nodes share their results with neighbouring nodes within specified clusters, which are smaller subsets of the entire set of all active nodes in the network. Therefore, we expect to have a greater reduction in communication delays.

Multi-agent systems often rely on preplanned, domain-specific co-ordination to satisfy flexibility demands in complex domains for co-ordination and communication. Firstly, it is difficult to anticipate and preplan for all possible co-ordination failures, particularly in scaling up to complex domains. Secondly, given domain specificity, reusability co-ordination has to be redesigned for each new domain. In contrast, the self-

organised approach implemented in the DPSS relies less on preplanned co-ordination of problem solvers and as a result it suffers less of the above difficulties.

4.4.1 Negotiation strategy

The subject of negotiation [Davis and Smith 1983] has been of continuing interest in the distributed artificial intelligence community. The operation of co-operating, intelligent agents would be greatly enhanced if they were able to communicate their information to reach mutually beneficial agreements. There are some techniques for helping nodes to resolve their sub-problem interactions, improve their agreement, and reduce their inconsistency [Barr et al 1989]. Firstly, the problem can be initially decomposed such that each node knows how its partial result should fit into complete solutions. Secondly, a co-ordinating node can be considered such that it builds the global view through its communications with every node. Thirdly, all the information from each node can be exchanged with every other node. Fourthly, to give a node the ability to predict what results it might form and need, by exchanging their plans. By this manner nodes can co-ordinate their individual actions to more effectively form compatible results. Fifthly, allowing nodes to selectively exchange a small subset of the data they form if that data is indicative of the node's eventual partial result.

The first approach cannot work when problems are inherently distributed. Therefore, nodes do not have a view of the overall problem. In the second approach, this node might be a bottleneck and a reliability hazard. The third suggestion is not acceptable because, in general, nodes cannot provide the communication or computation to share all their data. In the fourth and fifth approach, each node needs a sophisticated local control which allows a node to understand the implications of its planned problem solving, and communication actions on other nodes' goals and plans.

On the other hand, co-ordination is not achieved just through exchanging information. An alternative for co-ordination, employed in our proposed system, is to allow nodes to compose clusters of relevant nodes, by a process of self-organisation and to exchange their local partial solutions to each other within those clusters. Two sources

of information are considered to cluster the relevant nodes: Inter-node correlation, which are represented in the structure of the knowledge base, and a value function, which represents the amount of communications between nodes. A possible way to implement the process of self-organisation of clusters is by applying a hybrid genetic algorithm. The algorithm uses the value function to minimise the communication overheads and runs a simple hill-climbing procedure to find the inter-node relations.

Co-operation among the agents in the network is guided by a heuristic strategy that specifies in a general way the information and relationships among nodes. This heuristic, which comes from the expert knowledge, is used to provide each node with a high level view of problem solving in the network. Clustering specifies node interaction patterns. The fundamental motivation that drives us to use GAs in supporting negotiation is the view that the processing of negotiation can be viewed as negotiators jointly searching a multi-dimensional space and then agreeing on a single point in the space.

To determine which rules can be fired in parallel, the rule-dependency graph is examined to discover those rules which may interfere with each other. Rules from distinct rule-sets can be fired asynchronously without violating the serializability requirements. Within each rule-set, rules are fired sequentially.

4.5 Perspectives on the system

The overall mechanism for solving problems in DPSS is based on co-operation and co-ordination of multi problem solving agents. In general, two perspectives on co-operative problem solving are viewed; result sharing and task sharing [Durfee 1995]. The task sharing perspective, such as Contract Net protocol [Smith 1980] considers the co-ordination problem as associating tasks to be done with the right agents to do them. In the result sharing perspective, such as the blackboard architecture [Nii 1986], problem solvers would individually solve their local sub-problems and share their partial results. Over time, increasingly complete results would be formed and the system would solve the entire problem.

Task sharing can be coupled with result sharing. One way is through the assignment of problem solving roles to the nodes such that each node becomes a specialist in terms of a different aspect of the problem. Then, nodes co-operatively exchange and integrate partial results to construct a consistent and complete solution. The other possible way of coupling result sharing and task sharing, which we employ in our proposed system, originated from the formation of neuronal groups in the theory of neuronal group selection. In this approach, initially problem solving nodes co-operatively exchange and integrate their fine-grained partial results within selectively determined clusters of PS nodes to make each cluster specialist in some aspect of the given problem. Establishment of clusters can be interpreted as a task sharing perspective in the DPSS, because it is postulated that each cluster can satisfactorily respond to a feature of the problem.

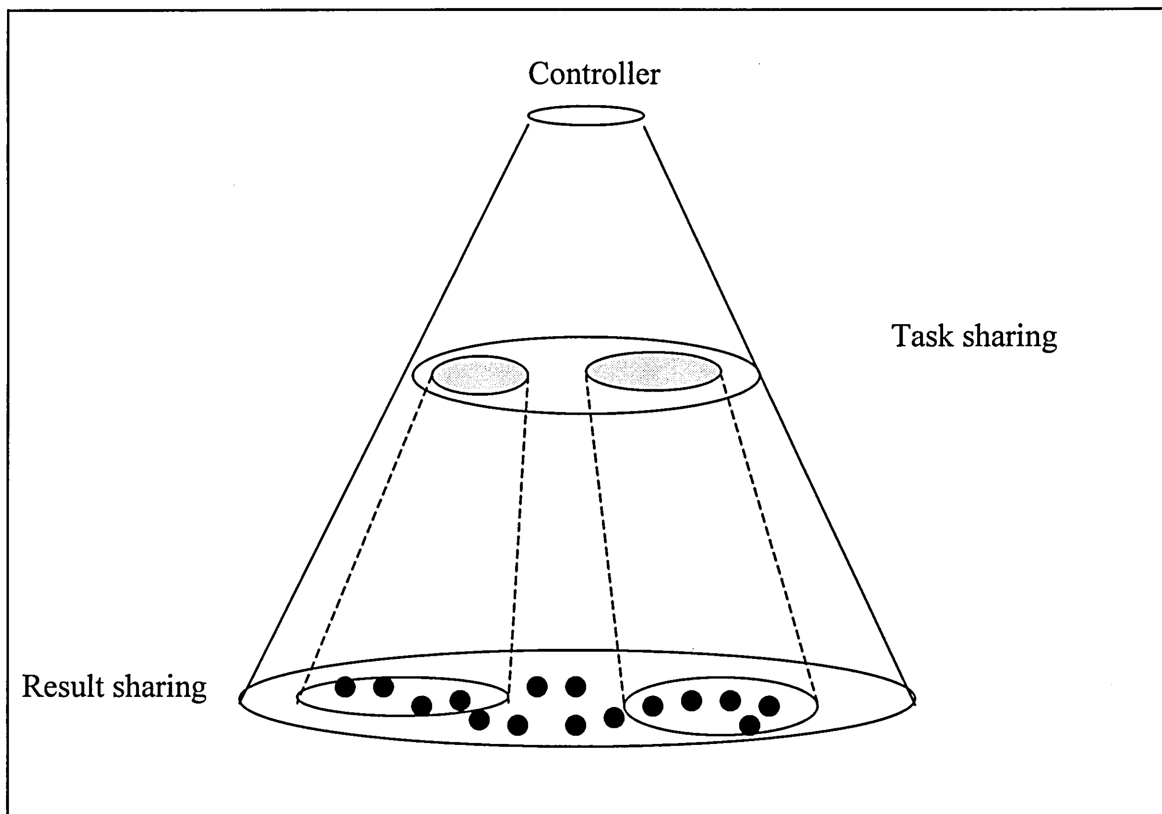


Figure 4.2. Three levels of the distributed problem solving system.

> A node in the lowest level is an agent with the ability to solve a portion of a complex problem. They share their results in a result sharing perspective, to form clusters of relevant nodes. At the higher level, clusters in a task sharing perspective find more complete solutions.

Then, the coarse-grained partial results, obtained from the clusters, are considered as the approximate solutions which have been given by the right agents. Formation of clusters, corresponding to a neuronal group [Edelman 1992], is based on result sharing. Instead of sharing results in the entire network of all active nodes, the PSs exchange their partial solutions only with their neighbouring nodes within specified clusters. We use the term 'distributed result sharing' for this kind of sharing results. Associating tasks to the right clusters is viewed as a task sharing perspective. The final result can be obtained by combining the approximations, which would be a globally satisfactory solution rather than an optimal result.

Each knowledge-based system relies on two principles. One is the appropriate representation of the application domain knowledge, and another is the control of this knowledge. These two principles are embodied in the two components of the knowledge based system architecture; a knowledge base (KB), and an appropriate inference mechanism for manipulation of its knowledge. In the previous section, we discussed the inference mechanism of the DPSS. In the following sections, we first discuss the knowledge representation paradigm and then present a model for the structure of the KB.

4.6 The knowledge representation paradigm

The proposed DPSS can be considered as a knowledge-based system that represents and uses knowledge. Rules are a popular paradigm for representing knowledge. We select production rules representation as an appropriate knowledge representation. The choice of knowledge representation lies essentially between the two broad classes of

symbolic and connectionist-style representations. The symbolic knowledge representation includes logic, production rules, frames, and semantic networks. According to the similarity between logic representations and rules on one hand, and frames and semantic networks on the other hand, we can have a choice between two general types of symbolic knowledge representations, rules and structures. The reasons for preferring production rules over connectionist and structure representations are as follows:

- Our view of Artificial Intelligence discipline is a top-down approach. By this we mean that cognition models of brain function, such as the theory of neuronal group selection, can be used as a tool for building intelligent systems, which help us better solve some of the problems. Thus, biological realism does not constrain our choice.

- There is a distinction between knowledge based tasks and behaviour based tasks, although we cannot say that all tasks executed by humans are only knowledge based or behaviour based. In behaviour based tasks such as speech, locomotion, and perception, the problem solver is not able to explain how the problem is being solved. In contrast in knowledge based tasks, such as diagnosis, which involve choice and reasoning, the problem solver is required to explain the reasoning. It seems that the connectionist model is better suited to behaviour based tasks [Pfeifer et al 1989].

- Production rules can resemble the structure of neurones in a connectionist structure. Both rules and neurones conceptually carry out the same operation. Rules resemble stimulus-response pairs in which the left-hand side can represent a stimulus of arbitrary complexity, and the right-hand side can represent a response sequence.

- Neural computing is suitable for problems where solutions cannot be explicitly described by a set of rules or algorithms [Tarassenko 1998].

- The implementation of features of knowledge based problem solving, such as explanation, task verbalisation, and the declarative description of knowledge, are limitations of a connectionist structure.

- The structure paradigm is suitable for domains consisting of a set of objects having interrelationships that need to be explicitly modelled, whilst rules are suitable for large and ill-structured problems.

- As a consequence of the limited and indirect interactions between rules, production systems are well suited to loosely coupled problems that are decomposable into relatively independent sub-problems or behaviours.

- Rule based systems show that the idea of using symbolic structures and symbol manipulation coupled to theories on heuristic problem solving and logical inference is a valid way to approach knowledge based tasks [Steels 1989]. One of the interesting properties of rules is the property of tractability, which means that the path of inference can be followed, unlike as in neuronal networks.

- Rule based systems are more suitable in problem solving and classifications tasks, although they have their own drawbacks. One primary drawback is that facts and rules cannot efficiently represent the structures and relationships within even simple concepts or objects.

- Sometimes it becomes intuitively obvious to the developer that knowledge in the domain can be better represented by rules or by using other paradigms.

We consider rules as consisting of a left-hand side, which is a sequence of one or more and-ed condition elements, and a right-hand side which is an action as illustrated:

IF condition-1 **and** condition-2, ..., **and** condition-n **THEN** action.

Or-ed conditions are considered as separate rules.

4.7 Structure of the knowledge base

Studies have shown that domain knowledge is the key to building problem solving systems, although methods of reasoning are important [Giarratano and Riley 1989]. In

fact, reasoning may play a minor role in an expert's problem solving system. The KB is one of the components of the system, stored on disk units and representing two types of knowledge. One is general knowledge relevant to the problem domain and the other one is specific knowledge relevant to a specific application of the system [Frost 1986]. Therefore the contents of the KB includes domain specific facts and relations, generic facts, and general rules about these facts. Facts and relations are used to represent characteristics of the entities and their attributes (binary relationships) in the application. In other words, specific facts describe a class member's properties and general rules describe class properties. All knowledge in the KB is represented in the form of production rules as described in the previous section.

Each large knowledge base includes the various domains of knowledge. To effectively use knowledge, the system must be able to extract only the necessary knowledge from the KB according to the type and domain of the problem to be solved. For this purpose, the KB must be well organised. It seems that the hierarchical model, as an appropriate database structure for the storage of knowledge in a very large knowledge base system, is necessary [Frost 1986, Barr et al 1989].

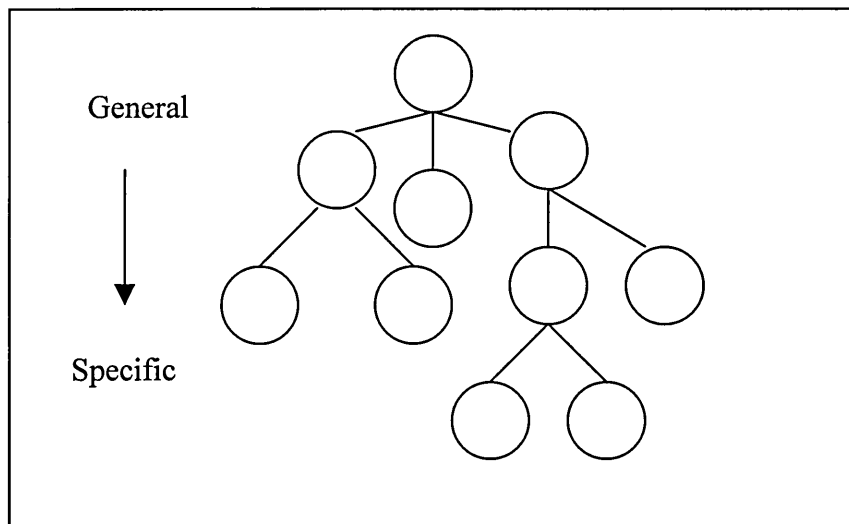


Figure 4.3. The hierarchical model of the KB.

The possible domain is divided among sub-domains.

In a hierarchical model, the global knowledge base that comprises of information about all possible sub-domains is replaced with a tree of distinct databases which correspond by related sub-domains (Figure 4.3). Each sub-domain is itself a KB. Actually, most of the information in a given sub-domain's database will be the same as the local database just above it (parent). To save space, only the differences are stored. The root of the tree, which represents the most general knowledge about the application domain, must be a full database. The hierarchical model of the KB provides a very efficient method of retrieving knowledge. Each sub-domain is retrieved via the other related sub-domains.

Production rules in each sub-domain of the KB are partitioned into several rule-sets. In general, the number of rules, representing domain and application problem, in a rule based system is often much greater than the number of processors in an array of PEs. Thus, many rules may be allocated to a single processor node. On the other hand, it is desirable to assign a balanced workload to all nodes. If two rules have to communicate with each other, assigning them to different nodes will take longer to communicate. If a number of rules are assigned to the same processor node, they cannot be fired in parallel. Performance of the system may be degraded by dependency between rules. Every rule might be assigned to only one rule-set. Rarely, some duplication of necessary information is allowed. As an example, suppose one of the sub-domains in the KB contains a simple domain knowledge comprising of three rules:

Rule 1: **IF** x and y **THEN** z.

Rule 2: **IF** a and b and c **THEN** x.

Rule 3: **IF** d and e **THEN** y.

These rules could be partitioned into two rule-sets: G_1 and G_2 as follows, and allocated to different processor nodes.

G_1: **IF** x and y **THEN** z.

IF a and b and c **THEN** x.

G_2: **IF** x and y **THEN** z.

IF d and e **THEN** y.

The overlapping subsets of information among the nodes make the network more robust. Dividing rules into rule-sets (rule level parallelism) is an important feature of rule based representation languages. It introduces the parallelism between production rules.

4.8 Behaviour of Problem Solving agents

The PSs are production rule based systems, which can formulate partial solutions using only their own knowledge. Each PS contains and fires some of the problem solving rules of the overall system, and consists of three components: Inference engine, working memory (WM), and production memory (PM).

A problem solving agent's PM contains a set of rules (rule-set), which represents parts of the problem solving knowledge. For simplicity, we assume no overlap between PMs in different agents, and assume the union of all PMs in the network is sufficient to solve the given problem. We also assume that each processor has the capacity to store one rule-set. According to the effects of rule-set distribution, a PS has incomplete knowledge to locally process certain data. The PMs are stored in agents' local memories as local knowledge bases of each PS. PSs infer with their current information, but the solutions to their local sub-problems are only partial.

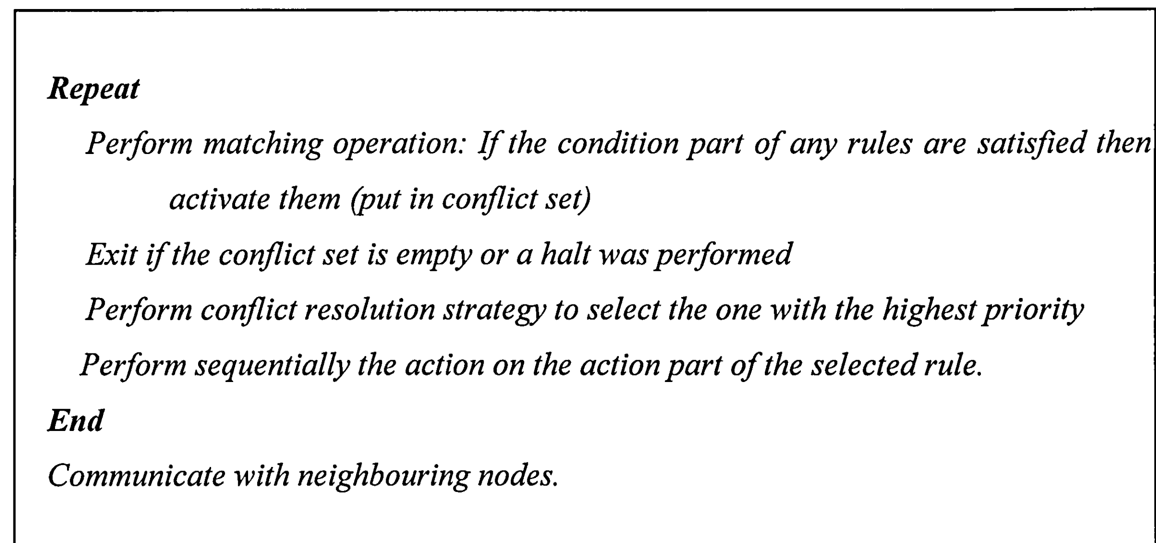
Each agent's WM contains a set of working memory facts regarding the status of the problem. The contents of each agent's WM, at the beginning of processing, is the information gained from the user and broadcast to the problem solving agents by the controller. As they progress in their problem solving, their initially identical working memories information will begin to vary. Each PS in the network deals with three kinds of data: The data given to the PS by a query, data generated by the PS itself, and data received from another PS in the form of messages. The WMs in different PSs may overlap, but the union of WMs in the system logically represents all the facts necessary to solve the given problem.

Each PS has an inference engine that asynchronously executes a serial model of production systems [Chapter 2]. The inference engine decides which rule stored in an agent's PM that could be applied to the current local WM should be invoked. The inference engine works in a forward chaining (data driven) manner, and executes the loop of match-select-fire until no matches are found in the match cycle. The intermediate conclusions, are new pieces of data which are placed in the agent's WM and copied to neighbouring agents. The choice of inference engine depends on the type of problem. We applied forward chaining because it does better in problem solving applications. The rule of inference used in problem solving is modus ponens. The modus ponens is a logical rule that states ' if A implies B, and you know A, then infer B'.

Each agent inference engine decides which rules' conditional parts are satisfied by WMEs, prioritises the satisfied rules, and executes exactly one rule with the highest priority in each cycle. The matching operation sequentially checks conditional parts of each rule against working memory facts. A rule whose patterns are all satisfied is activated. A rule may have many instantiations. Multiple activated rules may be on the conflict set at the same time. According to an adopted conflict resolution strategy, activated rules in the conflict set are prioritised. Prioritising is based on the combination of operations of recency, refraction, specificity and random selection of activated rules, respectively. To perform recency, each working memory element is associated with an integer, referred to as a time tag. This integer indicates when the element was first entered into the working memory. The inference engine uses time tags, at conflict resolution, to execute the rule matching the most recent working memory facts. The main idea of applying recency is to prefer rules that match against the most recent working memory facts. Refraction states that the same rule should not be applied twice to the same data. Thus, after a rule has been executed once, it should not be executed again. The main objective behind refraction is to prevent looping of those rules that do not modify any of their own conditions. The idea behind specificity is to prefer the more specific rules. The specificity strategy is used to make sure that exceptions to general rules are considered before the general rule is applied. This may be especially useful in dealing with default reasoning [Reichgelt 1991]. The main idea behind random

selection is that if all refraction, recency and specificity attempts to find a single dominant instantiation fail and more than one instantiation remains in the conflict set, then an arbitrary instantiation is selected. Consequently, if the match step produces a conflict set containing more than one rule instantiations, then the conflict resolution strategy selects one instantiation with the highest priority for firing. This leads to the operation of the action specified in the action part of the fired rule.

The following pseudo-code is a computational model, which summarised a problem solving cycle of distributed production system agents (PSs) by extending the conventional match-select-fire cycle to accommodate inter-agent data transfers. Each agent uses the same execution model. Processors send messages to neighbouring processors at the end of their knowledge processing and a select step is performed using both the local data and data from neighbouring agents. However, another possibility is that processors accept messages from neighbouring processors at the start of the match step (end of each cycle). Processing termination occurs after the match state, due to an halt action or once there are no more rule instantiations in the conflict set.


Repeat

Perform matching operation: If the condition part of any rules are satisfied then activate them (put in conflict set)

Exit if the conflict set is empty or a halt was performed

Perform conflict resolution strategy to select the one with the highest priority

Perform sequentially the action on the action part of the selected rule.

End

Communicate with neighbouring nodes.

Figure 4.4. The computational model of a problem solving cycle.

The inference engine of each PS searches through possible consequences of the data, and repeatedly executes a group of actions until no rules can be matched and fired with

existing data in its WM. Then, PSs send messages to neighbouring agents. Another possible option for termination the processing is that the user sets an explicit limit on the number of rule firings.

4.9 Implementation of genetic algorithm

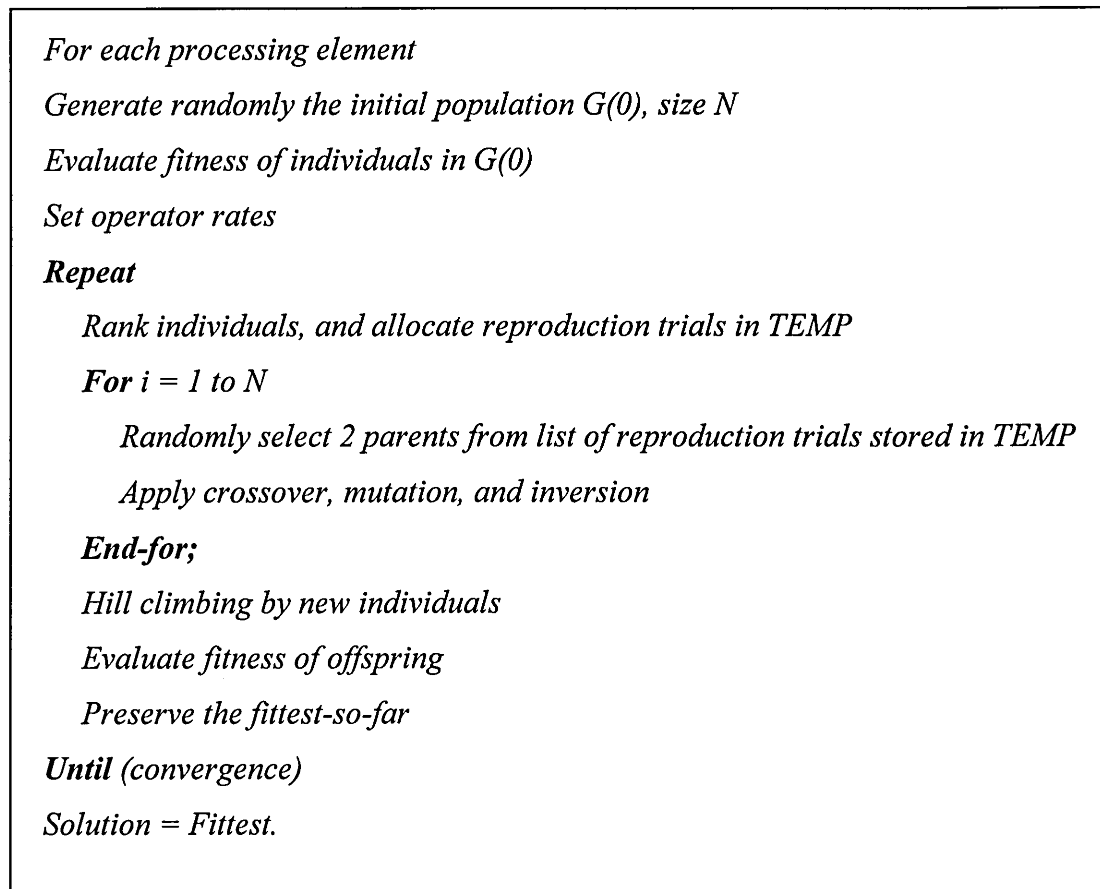
Each PS in the DPSS performs processing on its sub-problem and then co-operates and communicates with neighbouring nodes to produce an overall solution, because none of them has sufficient information to solve the entire problem. A composition scheme of PSs into clusters of relevant nodes is performed with the hope of improving local performance and reducing co-ordination overheads. In our system, search for finding the best configuration of neighbouring nodes is implemented by the PHGA. Since the initial population for the standard GA must contain many individuals to give good results, hence generates a huge amount of computation load. The availability of a parallel environment and the quantity of independent processing (computation of fitness values) required for each individual suggest that the algorithm is performed in parallel.

In our implementation of the PHGA, the population is distributed in several subpopulations on the network of problem solving nodes, so that each PS is in charge of one island. The islands (PSs) are positioned on a two-dimensional mesh network topology. These subpopulations increase the chances for the algorithm to find good solutions by having many standard genetic algorithms competing, each of them operating on small subpopulations. Distributing the individuals across several processors could thus decrease the computation time by competing in more search spaces evolved in parallel.

Each problem solving node in the network deals with its own population and applies the genetic operators of reproduction, crossover, mutation, and inversion according to their control rates, on its population to produce new generations. In this approach populations are isolated from each other and no individual migration is considered between them.

Another possibility is that the fittest individuals are allowed to migrate from one island to another. In this manner, a copy of the best individual is send to the next island on the vertical or horizontal lines. Each island can receive a new individual that replaces a randomly selected local individual. According to the results obtained by Calegari et al [1997] it seems that the number of islands influences significantly the quality of the results.

At the end of processing, PEs have candidates as their fittest individuals. The fittest individual among these candidates will be selected as the potential solution. In order to direct the blind search, the fittest individual is judged by the designer knowledge, which indicates some primitive inter-relationships between PEs. If the quality of a potential solution is acceptable, it is then considered as the solution of the PHGA, otherwise, its elements are distributed to the array of PEs. For each element, the structure of the appropriate cluster is changed if it can drop value of the objective function. In this case, the solution of the PHGA is the fittest individual among the individuals of the second stage of PEs processing. We have utilised the domain knowledge for increasing the efficiency of search. The incorporation of problem-specific knowledge to the fruitful regions of the search space, which directs the blind GA search, is referred to as hybrid GA [Goldberg 1989]. In our system, we have added a simple problem-specific hill climbing procedure for improving the efficiency of the search. We used two-point crossover and ranking selection to reduce the likelihood of premature convergence in our developed hybrid GA. The techniques and the procedures, which we have used, are outlined below.



```
For each processing element
Generate randomly the initial population G(0), size N
Evaluate fitness of individuals in G(0)
Set operator rates
Repeat
    Rank individuals, and allocate reproduction trials in TEMP
    For i = 1 to N
        Randomly select 2 parents from list of reproduction trials stored in TEMP
        Apply crossover, mutation, and inversion
    End-for;
    Hill climbing by new individuals
    Evaluate fitness of offspring
    Preserve the fittest-so-far
Until (convergence)
Solution = Fittest.
```

Figure 4.5. The computational model of the HPGA.

4.9.1 Chromosome representation

An individual, i.e. a cluster instance, is encoded as a string of integers, where an integer refers to a cluster and its position in the string representing the processor. The length of these strings is equal to the number of PEs, required with each request. For example, the genotype (1, 1, 2, 1, 2, 2, 3, 3, 3) indicates that processors 1, 2, and 4 are allocated to clusters 1, processors 3, 5, and 6 are allocated to cluster 2, and processors 7, 8, 9 are allocated to cluster 3 (Figure 4.6.).

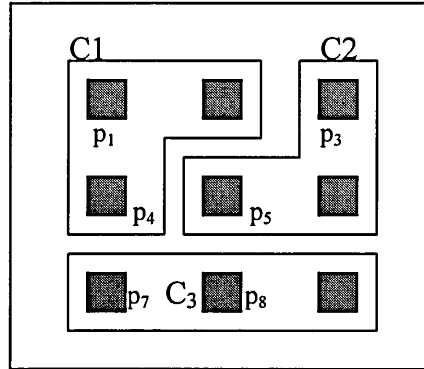


Figure 4.6. An instantiation of chromosome representation.

4.9.2 Fitness evaluation

In general, efficient utilisation of the computational power of distributed memory parallel computers is highly dependent on how the composite calculation-communication workload is distributed amongst the processing nodes. Efficient exploitation of the inherent parallelism of production systems however requires suitable algorithms for load balancing with simultaneously increasing communication overhead. To cope with this problem, the fitness of an individual is evaluated by the objective function in expression (4.1).

$$f = \beta \sum_i W_i^2 + (1 - \beta) \sum_i \sum_k D(i, k) \qquad (4.1)$$

Where, $i \in \{1, \ldots, P\}$ and P is the number of clusters. W_i is the total execution time on the cluster i. The parameter $\beta = 0.5 \in [0, 1]$ is used to denote the amount of competition between the calculation and the communication terms. D(i, k) is the distance between processors i and k, positioned on a two-dimensional array, defined by the following formula.

$$D(i, k) = |X_i - X_j| + |Y_i - Y_j|;\ i = (X_i, Y_i), j = (X_j, Y_j) \qquad (4.2)$$

This function is a combination of two equations (3.2) and (3.3), which represents the cost of parallel computing overheads in a multicomputer system, and required to be minimised for clustering of PEs. We use Eq. 4.1 as the objective function for the optimisation of clustering instances.

4.9.3 Reproduction scheme

The reproduction scheme is based on ranking, where the individuals are sorted by their fitness values and a number of reproduction trials according to predetermined scale of values is assigned. The expected number of offspring to be generated from the individual x at time t is indicated by the function tsr(x, t), and called target sampling rate. The sampling algorithm produces a new population by creating copies of individuals based on the target sampling rates.

$$tsr(x, t) = Min + (Max - Min)\,(rank(x, t) - 1)/\,N - 1, \qquad (4.3)$$

$$\text{with conditions: } 0 \leq tsr(x, t), \text{ and } \sum tsr(x, t) = N$$

Where the function rank(x, t) is the index of x, when P(t) is sorted in increasing order in terms of fitness values. Two conditions $0 \leq tsr(x, t)$, and $\sum tsr(x, t) = N$ require that $1 \leq Max \leq 2$, and $Min = 2 - Max$. The population P(t) is assumed to be of constant size N.

After ranking, pairs of parents are randomly chosen from the list of reproduction trials. The ranks assigned to the fittest and the least fit individuals are 1.2 and 0.8, respectively. First, single copies of individuals with ranks of more than 1 are assigned. Then, the fractional part of their ranks, and the ranks less than 1 are used as probabilities for assignment of copies.

4.9.4 Genetic operators

The genetic operators employed in the GA are two-point crossover, mutation, and inversion. Mutation refers to a random reallocation of a rule-set. Inversion inverts the allocation configuration of a segment of data in an individual. The two-point crossover is used because it offers less potential bias than the one-point standard crossover [Eshelman et al 1989].

4.9.5. Population size and operator rates

In the problem at hand, the population size is considered equal to the number of active PEs. We have considered fixed rates for the genetic operators of 0.6 for crossover, 0.002 for mutation, and 0.02 for inversion.

4.9.6. Hill climbing

Knowledge about the application can direct the blind genetic search to more profitable regions in the adaptive space. Herein, individuals carry out a simple problem-specific improvement strategy, known as hill climbing procedure that can increase their fitness. Hill climbing requires little memory and computational effort [Barto 1995]. The hill-climbing procedure can be expressed using a function which, for any feasible point x in search space, returns either an improved feasible point, or an indication that local improvement is not possible. Local hill climbing of the individuals is done with a simple exchange. The placement of two PEs in a cluster is exchanged. The exchange is accepted if it decreases the objective function that is to be minimised. This is repeated until no improvement occurs.

Another more direct example of domain knowledge involves the choice of the initial population used to start the search process. Although we have described the initial

population as randomly selected, if one can permit seeding the initial population with individuals known to have certain performance levels.

4.10 The overall behaviour of the system

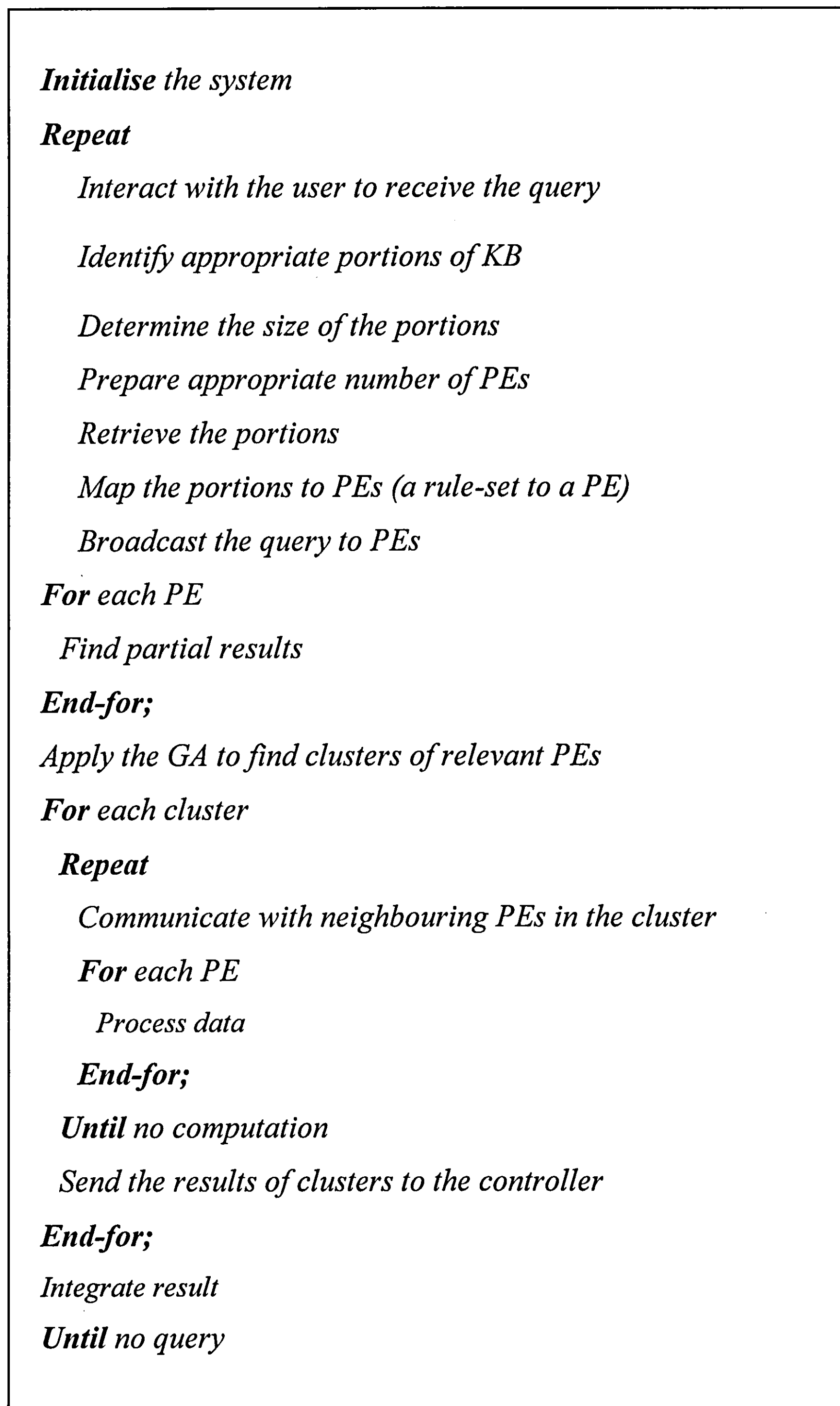
We have considered a network of loosely coupled asynchronous powerful problem solving nodes. Each PS is a production rule system capable of interacting with other processors. Each PS system comprises of the components, interpreter, which repeats the production cycle, domain knowledge contained in the knowledge base, which represents a part of domain, and working memory, which represents domain data. In the collection of distributed PS systems, rules are asynchronously fired by PSs. It was generally assumed that overlap between knowledge bases in different PSs does not exist, except rarely, and that the union of all knowledge bases in the system is sufficient to solve the given problem.

The PSs are interconnected so that each PS can communicate with every other node by sending messages and they do not share the memory. To perform domain problem solving by multiple production systems however, each PS needs knowledge that represents the necessary interactions among PS systems. A PS that has a relationship with a particular PS is called agent's neighbour. Some of the relationships between PSs are initially obtained by analysing domain knowledge at compile time. Others are selectively determined during run time. Composition of PSs to clusters of neighbouring nodes, by a process of self-organisation that could be working toward related results, is the main feature of the system. Composition changes the size of the PS population and may reduce response time and improve local performance by reducing co-ordination overheads.

A PS in each cluster reports a partial result to its neighbours. They use that result to confirm or deny competing results for their sub-problems, or to aggregate partial results of their own with the sender's result to produce a result that is relevant to the sender's sub-problem as well as their own. PSs within each cluster will repeatedly execute a group of tasks. The process continues within each cluster until all clusters reach their

saturation points. Thus, the results of processing in clusters are more complete than the results of processing in each PS. At the end of this stage, clusters of relevant nodes are formed which are specialists in different aspects of the problem.

For obtaining a complete and consistent solution, the clusters must integrate partial results. The results from clusters are transmitted to the controller. This approach ensures that the network will generate an overall solution as rapidly as possible, typically at the cost of some unnecessary communication of partial results and computations. The controller integrates clusters' results according to the concept of 'union' in the database systems. The following pseudo-code describes the general behaviour of the system.



```
Initialise the system
Repeat
    Interact with the user to receive the query
    Identify appropriate portions of KB
    Determine the size of the portions
    Prepare appropriate number of PEs
    Retrieve the portions
    Map the portions to PEs (a rule-set to a PE)
    Broadcast the query to PEs
For each PE
  Find partial results
End-for;
Apply the GA to find clusters of relevant PEs
For each cluster
  Repeat
    Communicate with neighbouring PEs in the cluster
    For each PE
      Process data
    End-for;
  Until no computation
  Send the results of clusters to the controller
End-for;
Integrate result
Until no query
```

Figure 4.7. The computation model of the DPSS.

4.11 Design of the system

The purpose of system design is to produce a solution to the problem. During this phase, the designer employs a variety of tools, such as data flow diagrams, and plans how data will flow through the system. In the case of a conceptual system, a simulation program can be used to describe and analyse the behaviour of the system. Then, the system is coded into an operational model, which is recognisable for the computer. This operational form is a representation of the proposed system. Simulating the system provides a means of analysing the behaviour of the real system by observing the behaviour of a model representing the system [Banks 1998].

In this section, which is devoted to design of the DPSS, we first draw a model of the simulated system using a hierarchical structure chart.

4.11.1 The structure chart

While design is a creative phase in a system life cycle, design methods can provide overall plans that will help to determine which choice may be most appropriate in a given situation [Budgen 1994]. The structure chart is a method of design that uses a graphical treelike representation to describe the way in which a program is assembled from a set of procedures. A method of design is a systematic approach for creating a design. Structure charts and data flow diagrams, due to their procedural nature, are not applicable to knowledge based systems [Gonzalez and Douglas 1993]. We used a method for representing the design of the system: Modified structure chart. It is similar to a conventional structure chart, but does not follow structure charts requirements e.g. one-entry-one-exit and sequential flow of control. Figure 4.8 represents a structure chart of the simulated program. This chart consists of boxes, which denote procedures, and arcs, which denote invocations.

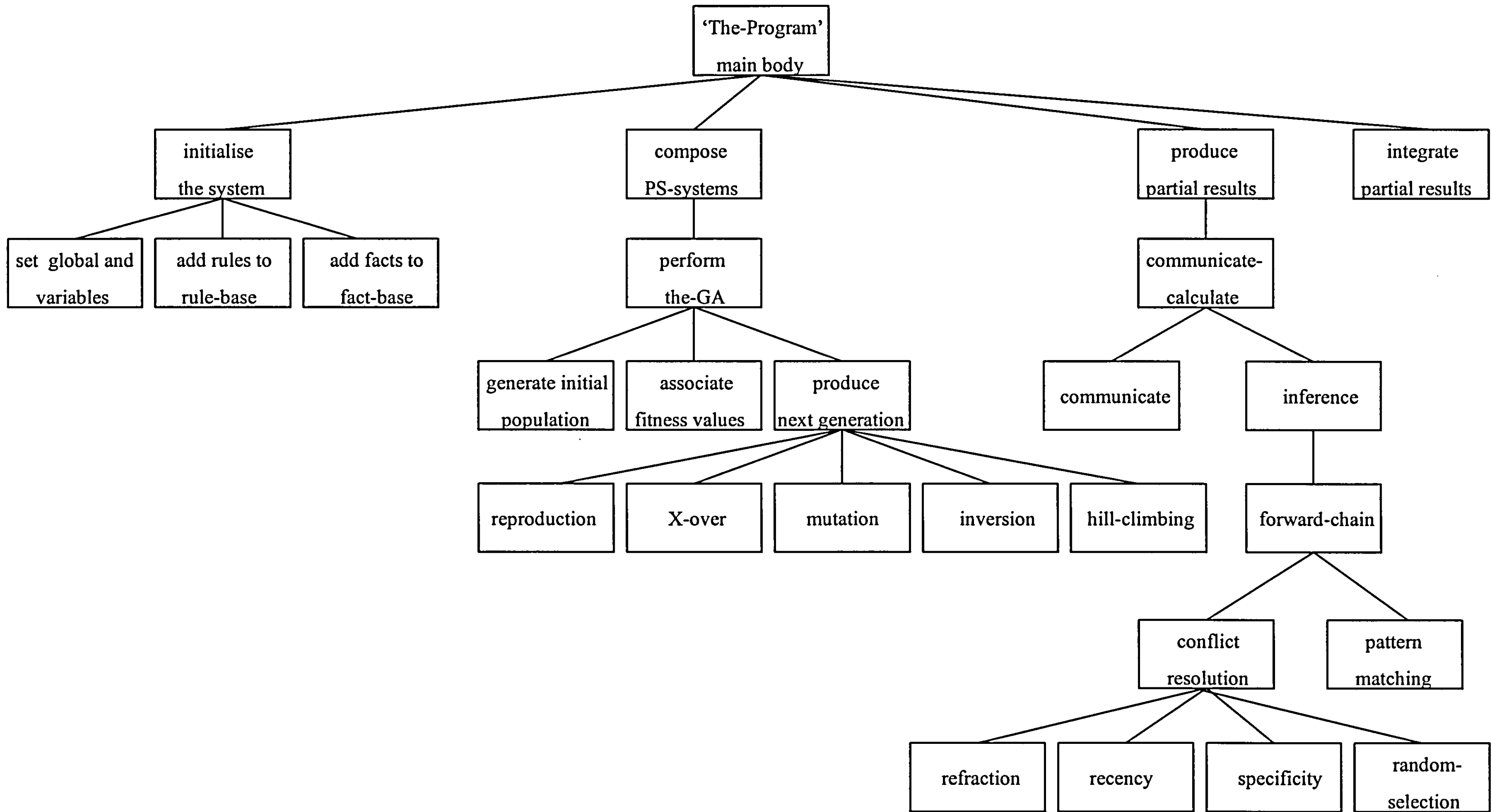


Figure 4.8. The modified structure chart describing the simulation program

4.11.2 The knowledge diagram

The knowledge diagram is a graphical representation for the knowledge. For a rule based system, the knowledge diagram details how rules connect facts together. This is detailed as a network of interconnected nodes where each node represents a single fact. Figure 4.9 represents a knowledge diagram for a portion of the developed KB.

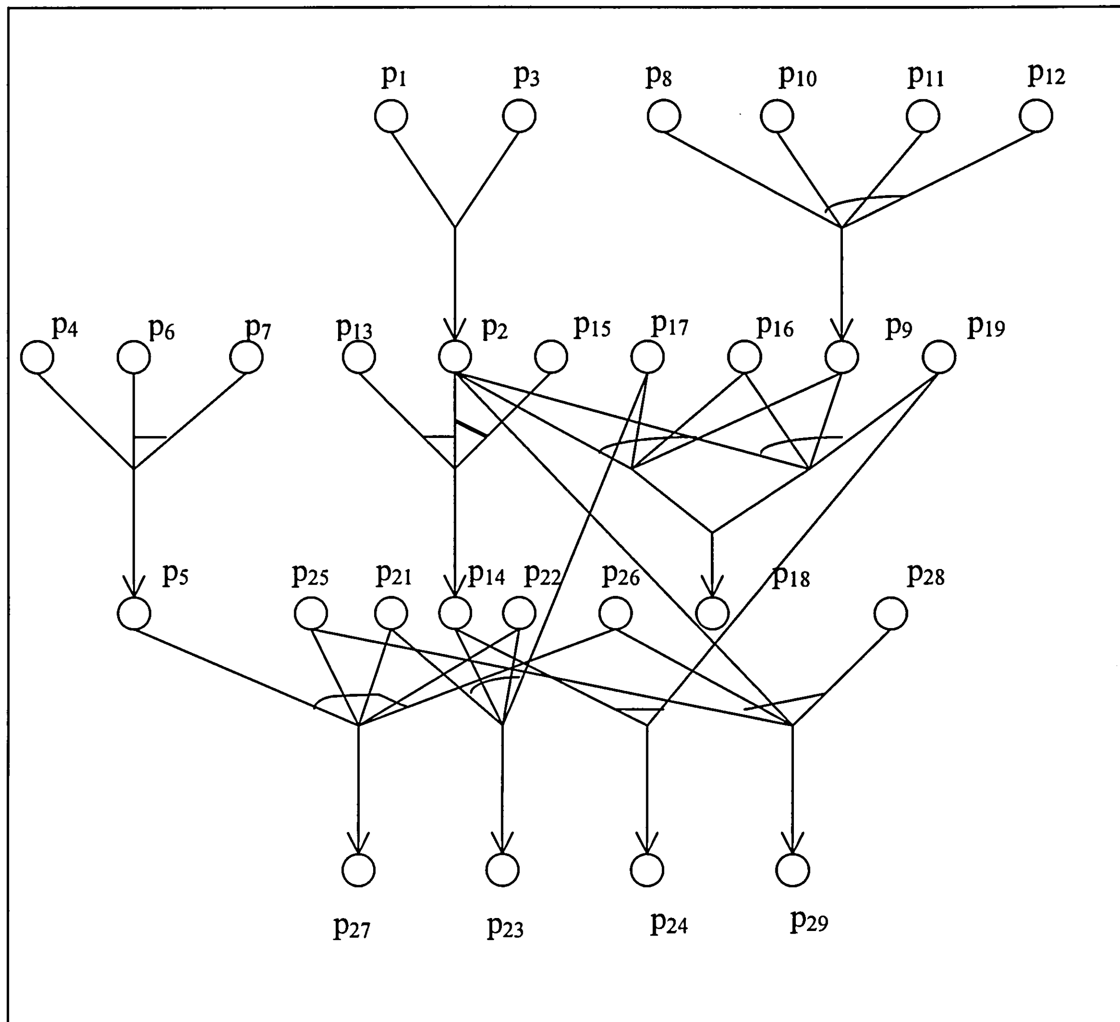


Figure 4.9. The knowledge diagram.

The directed arcs denote connections between nodes. Nodes at the higher level identify the condition parts of rules, and nodes at the lower level identify the action parts of rules. The disadvantage of this representation is that the diagram becomes complex as the size of the KB becomes larger.

As an example, the rule R_1: IF p_1 THEN p_2, is represented by the following figure.

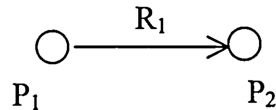


Figure 4.10. Graphical representation of a simple rule.

The conjunctive rule R_2: IF p_3 and p_4 and p_5 THEN p_6, is illustrated by Figure 4.11.

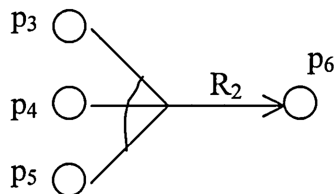


Figure 4.11. Graphical representation of a conjunctive rule.

The disjunctive rule R_3: IF q_1 or (q_2 and q_3) THEN q_4 is represented by the Figure 4.12.

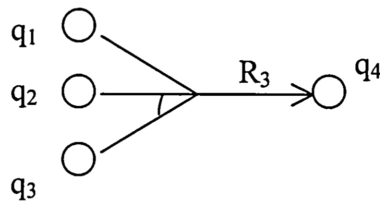


Figure 4.12. Graphical representation of a disjunctive rule.

4.12 Summary

This chapter has presented an approach which attempts to exploit the benefits of distributed processing by organising problem solvers into clusters of relevant agents with the aim of solving complex problems. The approach described is summarised as follows: Given a complex problem, compose production systems agents into clusters of relevant agents by process of selection, so that each cluster provides an appropriate solution in some region of the state space. Then, combine these approximations to yield a globally satisfactory solution rather than extract the best solution.

The supporting architecture was considered as a loosely coupled multicomputer with distributed memory. The central purpose in the design of this system was to increase the solution creation rates by forming sub-solutions in parallel and gaining the many potential benefits of distributed/co-operative problem solver systems. Co-operative problem solvers offer advantages of speed, reliability and modularity of the system in applications where information and expertise are naturally distributed, or where we can distribute them. Distributed memory technology is also less expensive and more accessible than shared memory technology.

The mechanism of inference in the proposed knowledge based system is a coupling of task sharing and result-sharing. The genetic algorithm was considered as a means for finding appropriate and relevant problem solvers. The expert knowledge is used to guide the GA-clustering search.

A hierarchical model was considered as an appropriate structure for the storage of knowledge. Production rule representation was chosen as a suitable knowledge representation. Where the number of rules in the knowledge base is much greater than the number of nodes in the mulicomputer, then many rules are allocated to a single node. This utilises the production level parallelism in our system. The assignment of production rules to nodes has a great impact on the overall system performance. To find an optimal allocation of the production rules has been proven as a NP-complete problem [Xu et al 1991].

The specifications of the PHGA were discussed in more detail. Pseudo-code that describes the behaviour of the system was presented.

The final section of this chapter presents a description of the design of the distributed production system, proposed in this project. A modified structure chart for representing the simulated system design has been used.

Finally, a knowledge diagram illustrating a graphical representation of the rule base and how the rules connect facts together has also been included.

5

Testing and Evaluation

5.1 Introduction

Knowledge based systems have development life cycles similar to those of conventional software systems. They also have verification and validation processes which resemble those for conventional software program testing [Ayel and Laurent 1991]. The basic motivation behind system testing is to control and check performance, efficiency, and quality of the knowledge base. For computer programs, system testing is the process of comparing the program's results to its original objectives, i.e. what the program should do and how well the program should do it. The objective of the verification process is to ensure that the system performs the functions defined in the objective specification and that it is free of errors. In contrast, expert system validation ensures that the system meets the expert's expectations and that the output of the system is correct. In the case of a conceptual system, a simulation program can be used to describe and analyse the behaviour of the system. Simulating the system provides a means of analysing the behaviour of the real system by observing the behaviour of a model representing that system [Banks 1998]. The system is encoded into an operational model which can be processed on a computer, provides a representation of

the proposed system. We have developed such a simulation program for modelling those aspects of the proposed system which are of particular interest.

In general, during system testing, several aspects of the system need to be tested. However, all categories are not necessarily applicable to every system [Myers 1979]. Our purposes of simulation of the system include analysis, cost effectiveness and performance evaluation of the system prior to its implementation.

In this chapter, we first describe the structure of the simulation program and then the activities related to verification of the simulated distributed problem solving system. These include the verification the inference engine and the knowledge base. The validation process, which comprises of the determination testing methodology and validation criteria, is then discussed in Section 5.4. Several performance measurements are considered in order to evaluate various aspects of the system. The results of benchmarking the simulated system and discussion about comparison of genetic-based clustering and random-based clustering are presented in Section 5.5.

5.2 Structure of the simulation program

The basic structure of the simulation program, which has been written in Lisp [Winston 1989] functional format, contains Global definitions (global variables), data structures and macros, the abstract program, defining the various operations performed on the data structures.

Global variables contain the information used for various operations performed by the program. For each PE, working memory or fact base FB, rule base RB, and conflict resolution set or agenda AA should be visible to the entire program, considered as global variables. The structure of fact base and agenda has been considered as the list structures within the program. The choice was based on the simplicity of the supporting functions for list structures. Since rules in the system are referred to in a sequential fashion, it is better to provide a means for referring to each rule by name. We have placed the rule base into a hash table. A hash table provides a means of mapping a set of

objects into a set of hash codes with the ability of retrieving the original objects. The aim is to decrease the average time to search through a linear list [Harrison 1990].

We have developed a frame-like structure for each production rule's structure with three slots, being: Rule, lhs, and rhs, because it has a flexible structure that can be easily modified. The slot rule takes the rule name. The slot lhs has the conditions that support the condition part of the rule. The slot rhs has the action part of the rule as its value. Condition elements of a rule have been considered as and-ed together, and or-ed components have been considered as separate rules.

The abstract simulation program, which is presented in Appendix A, contains various functions and sub-functions that execute pattern matching, conflict resolution, and the genetic algorithm implementation. The top-level function 'The-Program' is a representation of the three stages of processing in the theory: Initialising the system, finding clusters of relevant of PEs by running an appropriate genetic algorithm, and integrating partial results obtained from the clusters.

Initialisation is implemented by a loop structure function initialise, which sets the global variables, adds the user inputs (facts) from a file to the fact base of each PE (broadcasting of facts), adds domain knowledge (a rule-set) to the rule base of each PE, and performs the forward-chaining inference for finding the initial partial results.

The parallel hybrid genetic algorithm is performed by the generation of initial-random population and associating the fitness values to individuals. It applies the genetic operations of reproduction, crossover, mutation, and inversion, respectively, for each sub-population. In order to use the domain knowledge and bias the search of GA, the individuals perform a simple hill-climbing heuristic procedure. The genetic algorithm is performed by the iterative function find-clusters. The output of this function is a list, which represents the clusters' configuration.

Getting results from the clusters is performed by the function get-partial-results-from-clusters. This function takes the value of the find-clusters function, which represents the structure of clusters, and for each cluster, in a loop structure, performs the communication-computation cycles until no matching rule is available for each problem

solving node. Integration of partial results is performed by the function integrate-partial-results on the base of union of the existing results.

5.3 Verification

To ensure the correct performance of an expert system two major components, the knowledge base and the reasoning as processed by the inference engine need to be verified [Gonzaleaz and Douglas 1993]. Verification essentially addresses the syntactic aspects of the knowledge base. Many researchers have discussed verification and validation issues of knowledge based systems, particularly, rule based expert systems, such as Jafar and Bahill [1993] and O'Keefe and O'Leary [1993]. On the other hand, there is a lack of research software and benchmarks in the field of rule based systems [Miranker 1991]. Therefore, we have developed a set of software and test cases unique to the DPSS system to test our ideas. We have applied conventional software verification techniques for our own developed inference engine component. These include static test techniques for checking wrongful use of data, faults in declarations such as the use of undeclared variables, and faults in control flow such as infinite loops in the program source code.

5.3.1 Verification of the knowledge base

Verification is a white-box process that analyses the rules for sequence, structure, and specifications. This phase of testing is usually done in a static mode. Many available tools have verification features that detect common errors in rule redundancy or syntax. CONKRET (Control Knowledge Refinement Tool) [Lopez 1991], and MELODIA (Logical Methods For Checking K-Bases) [Charles and Dubois 1991] are examples of these tools. Schmolze and Synder [1997] have presented a formal definition of production rule redundancy and an algorithm for detecting redundant production rules. However, due to inadequacy of automatic verification tools [Awad

1996] we performed the verification of the knowledge base manually, although this compounds the complexity of the verification process.

The knowledge base represents the problem domain in the form of production rules. The basic syntax used for representing rules, has the grammar (r (lhs) (rhs)), where r refers to the name of a rule, lhs is a sequence of the form (C_1 C_2 … C_n), and rhs takes the form (A). The C's refer to conditions and A is the action that is implemented when all C's are true. The syntax of a condition element C and the action A is ((? x) P), where (? x) represents a variable and P is an arbitrarily property, representing a value for the matching attribute.

The syntax grammar of the language can be expressed by the BNF notation as follows:

terminals = { ( ), ?}

wff	:: = (symbolic-atom (lhs) (rhs))
lhs	:: = (variable lhs-value)*
lhs-value	:: = ⟨constant-symbol number⟩
variable	:: = (? character)
character	:: = x
	y
rhs	:: = (variable rhs-value)
rhs-value	:: = ⟨constant-symbol number⟩
number	:: = integer

The symbol " * " indicates that the preceding item is repeated zero or more times. A symbolic atom is any sequence of characters that is not a number and the system will treat as a single unit. The angled brackets have been used to enclose non-terminal symbols.

We considered a medium size of the KB containing between 250 and 1000 production rules. The maximum number of clauses in the lhs of each rule is equal to 5, and the rhs part of a rule takes one clause. Multiple predicates on the lhs have the form of conjuncted predicates. A large production rule is broken into smaller and independent rules.

In our system testing, the knowledge base verification comprised of checking for the implementation of the rule based knowledge representation paradigm and employment of the proper reasoning technique specified in the previous chapter. Another aspect for verification of the knowledge base was to check for semantic and syntactic errors that might have occurred during the development of the knowledge base, including:

- Redundant rules: Two rules are considered redundant if they have identical premises and reach identical conclusions. Redundant rules offer different approaches to the same problem, causing a duplication of knowledge. Absence of redundancy provides greater specificity to the notion of completeness.

- Conflict rules: Two rules are considered as conflict rules if they have identical premises yet their conclusions contradict.

- Subsumed rules: One rule is subsumed by another if it has more constraints in the condition statements while they have identical conclusions. If one rule is true, one knows the second rule is always true.

- Missing rules: These rules are characterised by facts that are not used within the inference process.

- Circular rules, which are rules causing an infinite loop of rule firings, checked by the refraction strategy during the conflict resolution.

These error checks involve the consistency and completeness of the knowledge base.

We have divided production rules in the knowledge base into rule-sets such that rule instantiations for rules belonging to different rule-sets can be executed in parallel. The assignment of production rules to nodes has a great impact on the overall system performance. For example, if a number of rules are assigned to the same processor node, they cannot be fired in parallel. Performance of the system may be degraded by dependency between rules. If two rules have to communicate with each other, assigning them to different nodes will take longer to communicate. To find an optimal allocation of the production rules has been proven as a NP-complete problem [Xu and Hwang 1991]. Obviously, each production rule must be assigned to some processor. In order to avoid trivial solutions and load imbalances, such as all rules being assigned to just one processor, we assumed that each processor is capable of handling only a few productions. Thus a processor cannot handle more rules than its production capacity.

We used compile time analysis to detect interference among multiple rules by analysing the inter-rules data dependencies. Run time analysis requires global synchronisation, which is too expensive in distributed systems. The compatibility problem of rules within rule-sets, and clusters of rule-sets is also solved at run time by sequential execution cycles of reasoning (local synchronisation). Compile time analysis can be performed automatically or by hand. For instance, CHECK [Nguyen et al 1985] is a program that verifies the consistency and completeness of the rule based expert system knowledge bases, and also generates a dependency chart which shows the dependencies among the rules and between the rules and the goals.

If a set of rule instantiations has a data dependence cycle then they are incompatible. Interference can be identified when multiple rules destroy other rules' preconditions in a cyclic fashion. If incompatible rules are distributed to different rule-sets then the result of parallel execution of the rules might be different from the results of sequential executions. This problem is caused by the elimination of the sequential conflict resolution strategy in the whole of the production system.

We assumed it is the programmer's responsibility to write correct programs and to ensure that the parallel production system reaches the correct solution. A program is correct if and only if all eligible serial execution paths reach correct solution. Obviously, the disadvantage of this method is that as the size of system increases, the programmer's job becomes more complicated.

In a test run example, 344 production rules [Appendix B] have been divided into 10 rule-sets according to tasks that they can perform. Table 5.1 shows a distribution of rule sets onto processors.

Processor	Rules
# 0	R1-R20, R327- R344
# 1	R21- R40
# 2	R41-R58, R307- R326
# 3	R59-R77
# 4	R78-R95, R247-R266
# 5	R96-R112, R267-R286
# 6	R113-R132, R192-R210
# 7	R133-R151, R211-R228
# 8	R152-R171, R229-R246
# 9	R172-R191, R287-R306

Table 5.1. Representation of rule-sets.

5.4 Validation

Validation of a knowledge-based system can be accomplished by validating the intermediate and final results of the system. Module testing was performed to validate the intermediate results of the system. To perform a module test the specification, which defines the module's input and output parameters and its function, is compared to the

function of the module. After writing source codes for each module of the simulation program, testing of individual units has been tested by using the simple test data taken from Winston [1989].

5.4.1 Validation methodology

Verification and validation of knowledge based systems call for informal, subjective, and often arbitrary execution of test cases. Many methods can be employed in the validation of knowledge based systems [Gonzalez and Douglas 1993]. One of the most popular validation techniques is validation by testing. Validation by testing involves the processing of prepared test cases by the knowledge based system and comparing results with those of an expert for agreement. This method is a black-box (functional) approach where only the system inputs and outputs are significant. In this method, the test data has to be generated by the expert or automatic test data generating programs. Testing a knowledge based system based on test cases has its own limitations. Generating test cases is time consuming, and the credibility of the validation process can be undermined if the domain expert who writes the test cases is also the tester. We have applied program testing as the validation methodology. Test case generation was done in an ad hoc manner. We note that expert systems with so many possible states or conditions make exhaustive testing impossible.

5.4.2 Validation criteria

The characteristics desirable in the system are referred to as validation criteria. Determining the appropriate validation criteria is an important consideration for the system. Generally, system testing is impossible if the project has not produced a set of measurable objectives for its product [Culler et al 1999]. According to the purposes and goals of the system, the validation criteria may vary. As a general rule, all systems are valid against certain criteria and invalid against others.

For the proposed distributed production system in this project, we considered several criteria according to two major aspects of the system. Firstly, it falls within the knowledge based systems category and secondly, it is a parallel/distributed system. The adopted criteria for testing our system include the quality of the system's answers, the correctness of the reasoning used, the system's efficiency and its cost-effectiveness. Validation of knowledge based systems indeed includes the determining of factor(s) for deciding whether the system is performing acceptably, rather than perfectly. The definition of acceptable performance may vary depending on the particular application. One possible definition for the acceptability of performance in the DPSS is defined as follow: A solution of parallel/distributed execution of rules is acceptable if it agrees with solutions proposed by an expert faced with the same problem. Consequently, we obtain the criterion of comparison against expert performance. However, most knowledge based systems are designed to model the designer's knowledge and thus we should not expect high accuracy. On the other hand, since comparison against expert performance may include mistakes which made by the expert, it is not a precise and highly beneficial criteria. Accuracy of the knowledge based systems also varies depending on the conflict resolution strategy. For example, a rule based system which executes all the activation on the agenda on each cycle would be impossible to control for any real task, since the execution of an activation can prevent the execution of some other activation. Therefore, we ignored this criterion in evaluating the DPSS performance.

Another possible definition for acceptable performance is concerned with the database serialisation problem related to the parallel execution of production systems. A parallel execution of a set of production rules is serialisable if there exists a sequential execution of those production rules such that both the parallel and serial execution result in the same final database state [Ullman 1982]. Thus, another validation criterion for evaluating the DPSS was chosen as: A solution of parallel/distributed execution of rules is acceptable if it agrees with solutions of sequential execution of the same problem. According to this definition we tested the system against the following criteria: Comparison against known results.

Another possible performance measurement for evaluating the system is related to the main objective of applying parallel processing techniques to production rule systems to improve the problem of slowness of execution time. Execution time is a useful metric for testing of the DPSS system. In general, performance improvement due to parallelism (speedup) is the most common performance metric in parallel computing. This is the ratio of the run time of the serial solution over the run time of the concurrent software running on n processing nodes. Thus we have:

Speedup = execution time (1 processor)/ execution time (n processor) (5.1)

$$1 \leq \text{Speedup} \leq n.$$

The execution time for a given program is the total time taken to execute the program, and should be measured in terms of total consumed time for the workload including all activities between, and not just the CPU time. In the case of a simulated system, the execution time is measured in terms of the CPU time spent to implement the program.

If speedup is equal to n, then the application is said to be perfectly parallel or has a linear speed. Often the speedup trends towards linear speedup, but does not reach it, due to the communications overhead between the processing elements. If speedup is greater than n, then it is called super linear speedup. Super linear speedups have been seen in limited and special situations. They obtain when the application is too large for a serial system and causes certain overheads, but when implemented on a many processors system, none of the pieces is large enough to incur the same overhead.

Speedup of a production system depends on several other parameters, which need to be considered when evaluating the system. These parameters include the number of production rules, the number of working memory elements or facts, the composition of rules, including the number of clauses in the lhs and rhs of the rule, and the average number of rules eligible for selection on a given cycle. Other characteristics depend on the system, inference engine, and method of conflict resolution being implemented.

Therefore, we measured another metric to evaluate the performance of the simulated system: Performance improvement due to applying parallel processing techniques. To test the system against this criterion, two different approaches have been examined.

- Calculating speedup in terms of production cycles
- Calculating speedup in terms of the CPU times

5.4.2.1 Calculating speedup in terms of production cycles

As mentioned in previous chapters, the main objective of applying parallel rule processing techniques in production systems is to improve the bottleneck of sequential execution of rules by firing multiple rules simultaneously. Therefore, it is a reasonable assumption to estimate the execution time of a parallel or serial production system by counting the total number of production cycles performed.

Since in our proposed system, genetic clustering of problem solving nodes is performed at run time therefore it is needed to estimate, in some way, the execution time of the algorithm. However, in real-world search and optimisation problems, the overall time to run a GA is more or less dominated by the time consumed by the objective function evaluations [Deb and Agrawal 1999]. With the assumption that the time consumed by each objective function evaluation roughly equals the time needed for execution of a match-select-fire cycle in the production system, we counted the total number of function evaluations during the run of the PHGA as an estimation for the execution time. Counting the total number of function evaluations has been suggested by Professor Fleming at the Department of Automatic Control and Systems Engineering, University of Sheffield, in a private conversation, in an attempt to calculate the time required for the execution of the PHGA independent of the implementation and the computer system.

Let S be the total number of function evaluations in a genetic algorithm with a population size of N, and T the number of generations that GA must be run, then

$$S = T.N, \qquad (5.2)$$

because in each generation N function are evaluated. We also do not take any special care to not count an individual appearing more than once either in one single population or in subsequent generations. In a parallel implementation of the genetic algorithm, where the initial population is divided into M sub-population, the total number of function evaluations is then reduced to

$$S = T. N / M, \qquad (5.3)$$

because in each generation, M sub-populations simultaneously work in parallel. In our implementation of PHGA, we calculated the total number of objective function evaluations according to the Eq. 5.5.

5.4.2.2 Calculating speedup in terms of CPU times

LISP has a number of functions, which are useful for finding information about the current LISP process. For example, the function TIME returns the amount of CPU time and the amount of real time the evaluation required as its value. The exact output of TIME is implementation-dependent. The other function is GET-INTERNAL-RUN-TIME, which returns the amount of time the LISP process has consumed so far. We inserted codes in terms of this function in appropriate places of the simulation program. The difference between two calls of this function is the amount of internal time units, which are expended during the computations between these calls. Internal time units are implementation-dependent fractions of a second. In this implementation, an internal time unit is a thousandth of second.

The execution time of a program is tied to a specific computer system and depends on many factors [Hwang and Xu 1998]. When the program is complex or the workload varies with different input data, such as sorting and searching, the workload can be measured by running the application on a specific machine. Input-dependent programs take different times on different input data, because control flow of the program could

change with different inputs. When using execution time as the workload for such input dependent programs, we need to use the longest processing time.

The overall processing time (in seconds) for a given test case, in the case of parallel execution, has been calculated according to the following formula:

$$T_{comp} = T_{initial} + T_{GA} + T_{cluster} \qquad (5.4)$$

Where $T_{initial}$, T_{GA}, and $T_{cluster}$ are the times needed to execute the first stage of processing, the GA for clustering of relevant PEs, and the processing in clusters, respectively. We inserted some additional codes at appropriate places of the simulation program for the purpose of collecting information about model behaviour, in terms of the execution and communication times and the amount of information communicated between PEs. The program model should then be executed and the output analysed and the dynamic model behaviour evaluated. The test program was executed more than 3 times, with timing starting on the second iteration to exclude the warm-up effect. The minimal, maximal, and the mean time for all PEs are collected. To interpret the results we focussed on the maximum execution time, not the average time over all PEs, because execution of a parallel program is not finished until the last process has been terminated.

Calculation of communication overheads

Communication was assumed to be performed in packet routing mode; a message along with its destination address is placed in a packet that moves through the nodes of the interconnection network. In the store-forward packet routing model, each node has a store/send capacity, it sends a packet to its neighbour and waits for an acknowledge message from the receiver. The transmission of a packet from one network node to any of its neighbours takes the communication time (in μs)

$$T_{comm}(m) = t_0 + m / r_\infty. \qquad (5.5)$$

T_{comm} is the communication overhead and is a linear function of the message length m (in bytes), t_0 is the start-up time in μs, and r_∞ is the asymptotic bandwidth in Mbyte/s.

In the simulation program, each packet contains a few working memory elements, resulting from the processing of information by each PE. The size of each packet was calculated by the function FILE-LENGTH, which returns the number of characters in the packet (file). We assumed each character is equivalent to a byte, and 2.7 Mbyte/s is a reasonable assumption for the transfer rate [Ishida 1994]. This amount is related to iPSC/2 [Bomans and Roose 1989], a typical message-passing multicomputer, with start-up time 700 μs [Culler et al 1999].

The impact of communication is best estimated not by the absolute amount of communication but by a quantity called the communication-to-computation ratio. This measurement gives a good indication of the overhead associated with the parallel process. This is defined as the communication time divided by the computation time;

$$T_{comm}/T_{comp} \qquad (5.6)$$

In general, if the ratio value is small, higher speedups can be achieved with some performance tuning. With large values, the parallel algorithm should be redesigned. The lower values of this ratio mean that the communication mechanism is relatively more effective than the raw computation power of the parallel system.

5.5 Evaluation of the results

To evaluate the effectiveness of the proposed approach, a simulation environment was implemented and the generic program executed. The complete program, which includes 344 rules, is included in Appendix B. The test cases, which are various aspects of the program in terms of different number of rules, number of facts, and number of PEs, have provided a range of benchmarks on a sequential compatible IBM PC, running a LISP based environment. Knowledge processing is based on matching, selecting, and firing of the rules. Each PE is assumed to be capable of performing those operations

(knowledge processing). Each PE adds the results of its processing to a file or a packet and sends the packet to its determined neighbours in the clusters. The process of passing packets from one PE to its neighbour is based on the concept of message passing, and hence provides communications between PEs. During execution, processors run the same algorithm on their own sub-problem (task), and then communicate with their relevant processors to exchange information and repeat a computation-communication cycle until they reach their saturation points. Thus, in the model of computation, calculation and communication do not overlap. This model of computation is repeated asynchronously for other parallel clusters.

5.5.1 Comparison against known results

To evaluate the results of the distributed production system against known results, we compared the result of the DPSS with the results of sequential processing of the same programs. In the most cases of test programs the parallel firing results of execution rules were compatible with the results of sequential executions in any order of applying those rules. However, since the process of formation of clusters in the system was a dynamic process implemented at run time, we obtained various instantiations of clusters for different runs of the same program. This results in that for a given program, after several times of running, there were cases that the conclusions of different runs were not the same set. This problem is related to the application of sound deduction rules in the field of automated theorem proving, which needs to be resolved.

5.5.2 Performance improvement

The simulation results for the test programs under the DPSS model, using the production cycle counts, are summarised in the Table 5.2.

Measurements	Production programs			
	Test-1	Test-2	Test-3	Test-4
# of production rules	51	344	286	286
# of working memory facts	14	71	43	50
# of sequential cycles = α	16	36	15	29
# of parallel cycles = β	3	9	7	8
# of function evaluations = γ	8	10	10	16
Speedup = $\alpha/(\beta+\gamma)$	1.45	1.89	0.88	1.20
# of rule-sets (PEs)	8	10	15	15

Table 5.2. Simulation results using production cycles.

From Table 5.2, we observe that the number of parallel cycles has been sensibly reduced in comparison to the sequential cycles, specially in the cases of large production rule systems with large amount of working memory elements. This indicates the potential parallelism available in production systems. However, speedups up to 2-fold have been obtained for the DPSS. This is mainly related to the high computational costs of genetic clustering. Speedup values depend on the number of generations which should be run for the genetic algorithm, the number of PEs, and the population size. For a fixed population size, higher speedups can be obtained if the number of PEs is increased or the number of generations is reduced.

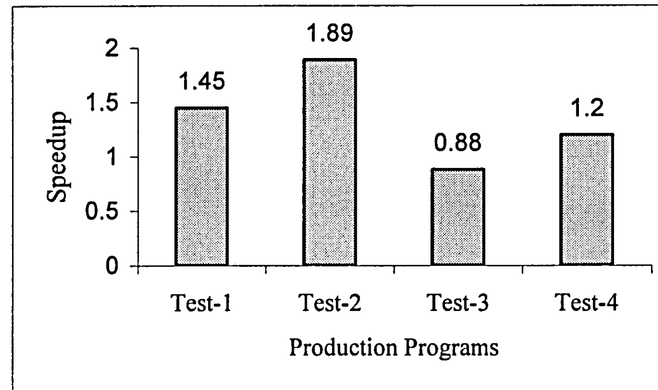


Figure 5.1. Performance of the DPSS.

Figure 5.1 provides a histogram showing the achieved speedup rates for each of the production programs. As can be seen, Test-2 resulted in the highest rates of 1.89. Test-3 provided a rate of only 0.88.

Measurements	Production Programs						
	A	B	C	D	E	F	G
# of production rules	286	326	344	286	286	286	344
# of working memory facts	40	43	46	40	40	40	52
Sequential times (s)	6.5	9.1	15.8	6.5	6.5	6.5	19.4
DPSS time (s)	21.3	13.6	13.3	16.2	30.8	12.5	18.3
# of rule-sets (PEs)	15	10	10	10	17	8	10

Table 5.3. Simulation results using execution times.

Table 5.3 summarises the simulation results, using the execution times. From this table, the following facts can be derived, which are evaluated by various charts (Figures 5.2- 5.7). In these figures, communication overhead is ignored.

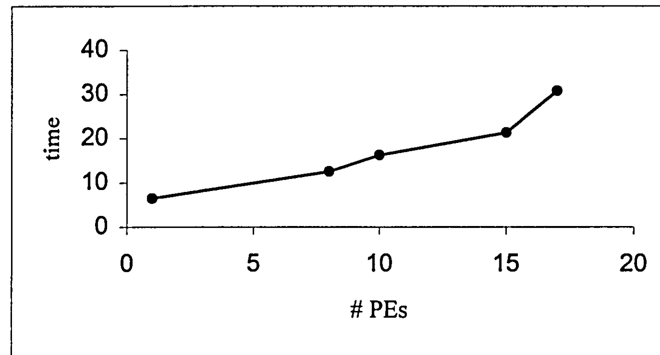


Figure 5.2. Processing times against number of PEs.

Figure 5.2 shows a plot of time (seconds) against the number of processing elements used to implement cases where the number of rules and facts are 286 and 40 respectively. The major outcome of these computations is that the addition of more processors on a fixed and small sized knowledge base does not decrease the processing time of the DPSS. The reason can be explained by the fact that in our implementation, as the number of PEs is increased, the population size becomes bigger, and consequently the time associated with execution of genetic algorithm is also increased. Another result is that the sequential execution of small production systems is more efficient than the DPSS approach for processing them. Thus, to improve the efficiency of the system employing rule-level parallelism, production rules should be divided into larger sets of rule-sets to reduce the interactions between PEs as much as possible. This emphasises the coarser grain level of parallelism of the production system.

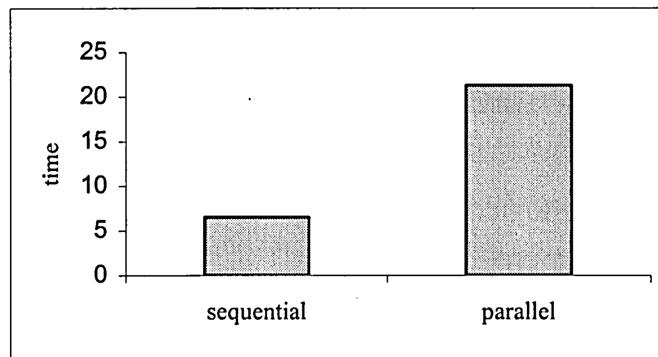


Figure 5.3. Comparison between sequential and parallel processing times.

Figure 5.3 shows the results obtained using the same number of rules and facts as in Figure 5.2, but with the number of PEs fixed at 15. Under these conditions, the sequential approach can be seen to be more efficient than the parallel one.

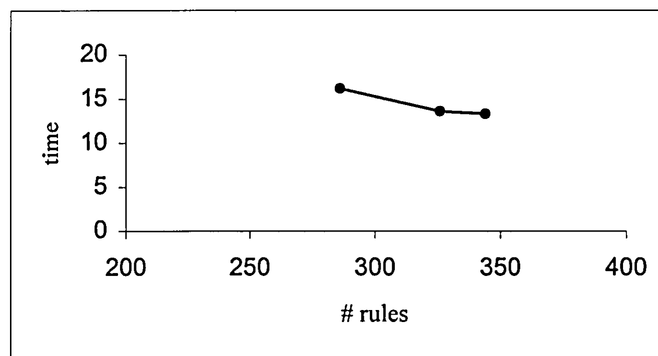


Figure 5.4. Variances in parallel processing time.

Figure 5.4 illustrates the time variances where processing has been carried out on cases with the number of PEs fixed at 10, and where the size of the knowledge base varies. The processing time is decreased as the number of rules increased. The reason is that with a fixed number of PEs, the time related to execution of the genetic algorithm will be constant, and as the number of rules increases, the sequential execution of knowledge base takes more time than the multiple rule firing of production rules in parallel. This emphasises on larger sized of knowledge bases to increase the efficiency of the DPSS system. Thus, the appropriate type of production system for the proposed approach in this project is a medium or large sized knowledge base.

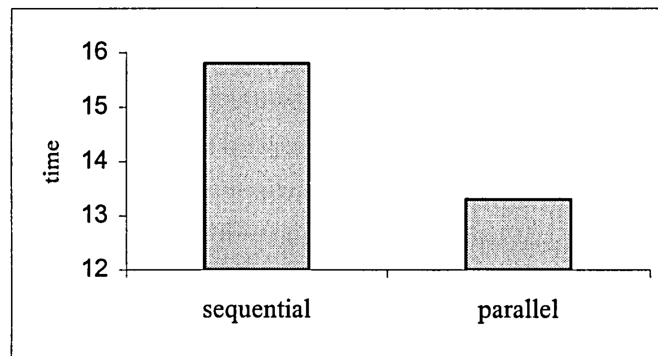


Figure 5.5. Comparison between sequential and parallel processing times.

In Figure 5.5, the number of rules, facts, and PEs are 344, 64 and 10 respectively. In this case, the parallel execution time is reduced in comparison with the sequential execution time of the same data.

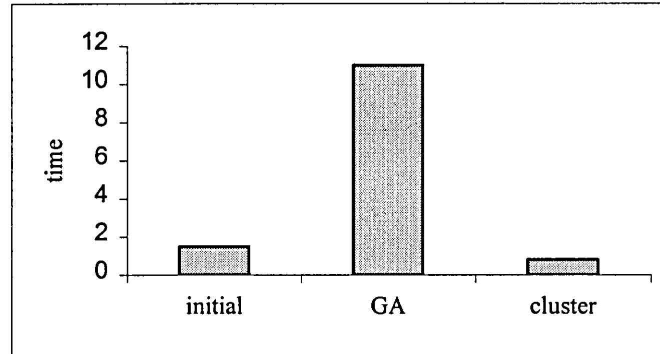


Figure 5.6. Comparison between execution times for different phases of the inference methodology.

Figure 5.6 analyses the parallel execution time from the same conditions as given in Figure 5.5. It indicates that the most time consuming stage in the proposed approach is the formation of clusters of relevant PEs by the implementation of the GA. Thus, using evolutionary computing at run time processing is not cost effective, due to the large amount of time consumed for processing of the genetic operations on populations of individuals. For systems where the execution time of a given program is an important factor, off-line evolutionary computation seems more appropriate.

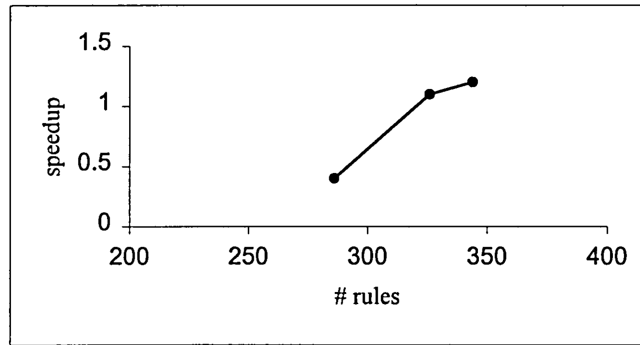


Figure 5.7. Speedups against the number of production rules.

Figure 5.7, under the same conditions as given in Figure 5.3, denotes as the size of the knowledge base increases the values of speedup also increases in the DPSS.

Communication overhead

The overall communication overhead was very small in the system. For example, under the same situation conditions given in Figure 5.5, the total number of transferred characters is 980 bytes. Hence, according to equation 5.4, the overall time spent to transfer this number of bytes equals to T (980) = 700 (μs) + 980 (byte) / 2.7 (Mbyte/s) = 0.001 (s).

The main reason for this can be explained by the fact that the number of changes made to working memory per cycle is very small in production systems. On the other hand, development of message passing machines has reduced the communication overhead by an order of magnitude, and the communication overhead is not a problem in current message passing machines.

Comparison between genetic-based and random-based clustering

In our implementation, the genetic algorithm starts with a random-generated initial population and, based on fitness measures derived, optimises the problem of communication and computation overheads in the network of problem solvers. The algorithm is also strongly directed by the knowledge of the system designer to reach to the more beneficial states of the search space. We tested the proposed approach by using a random-based algorithm to determine the patterns of communication between problem solving nodes in the network. Due to elimination of the γ factor in calculation speedups, $\alpha/(\beta+\gamma)$, we obtained more speedups of 2.1 to 5.3-fold. On the other hand, the role of expert knowledge in formation of clusters, in a random-based clustering, has been removed. The influence of this absence was as follows. The number of times that the results of test cases and also the number of facts were different from the sequential processing increased. The amount of communication and computation overheads were also more than the values obtained by running a GA. Therefore, a random clustering has more chances to produce results which are not the same as serial processing. Also, a random clustering has more speedups at the price of more computational costs.

5.6 Summary

The central objective of this chapter was to describe the testing, along with verification and validation of the DPSS, and evaluate the performance of the distributed production system proposed in this thesis. Section 5.2 briefly described the structure of the simulation program. Section 5.3 addressed the verification of the system for syntactic aspects of the knowledge base. The validation process, composed of a determination of a validation methodology and some appropriate criteria for evaluating the system, is discussed in Section 5.4. Two different approaches for calculating the speedups have been adopted. One is based on calculating the production cycles and another on the basis of measuring the CPU times for execution of the benchmark programs. However, since the execution time of a program is highly dependent on the specific computer hardware and also language implementation, then applying this

method on a computer PC has been subject to criticism by some researchers, although there is some support for this method. Hence, in evaluation of the result we concentrate on the first method. The simulation results indicate that the proposed approach is a valid way to developing knowledge based systems, and the appropriate size of production systems for the proposed approach is large sized knowledge bases with large amounts of working memory elements. The results also indicate that genetic algorithms as a method to determine the patterns of communication between problem solving agents are computationally expensive to implement at run time.

6

Conclusion

6.1 Introduction

In previous chapters we have briefly reviewed parallel execution of production systems and genetic algorithms, which form the introductory topics for this research work. We have also discussed a distributed production system (DPSS), with its problem solving strategy inspired from the concept of self-organisation in the brain, presented by TNGS. In this chapter, we first restate the aims of this research project, and then briefly review the thesis. Discussions on the evaluation of simulation results and direction for further work are the topics of the following sections of this chapter.

6.2 Restatement of the project aims

The main objective of this research programme was to develop a knowledge representation language along with a corresponding investigation of a suitable parallel architecture to support it. The proposed system introduced a co-operative mechanism of problem solving based on a selection mechanism. The main concern in TNGS is related to the role associated with groups of neurones, which selectively form and vary

functionality. Accordingly, the problem solving system forms clusters of relevant Problem Solvers (PS) by a process of selection, such that each cluster can provide an appropriate solution in terms of some aspects of the given complex problem. These approximations are then combined to yield a globally satisfactory solution. A parallel genetic algorithm, which uses the principles of evolution through natural selection, was developed as a search and optimisation technique to find the patterns of activities between appropriate PSs. After the establishment of the knowledge representation scheme and identification of the supporting parallel architecture, a simulation was developed to emulate the behaviour of such a machine, which was then subjected to a series of tests to assess the validity of our approach and potential benefits of the system, including efficiency and speedups.

6.3 Summary of the thesis

In Chapter 1, we reviewed the history of AI at a glance and pointed out some attempts which have been made during the last few decades to construct intelligent systems. Attempts to create artificial intelligence can be categorised into two broad classes of symbolic AI and connectionism. Although both approaches model the mind as an information processor, there are several differences between them. In the symbolic approach, cognition is defined as the transformation of symbols according to rules, by any device that can manipulate the symbols. In contrast, cognition in connectionism is defined as the emergence of global states in a network of simple components, through local rules for individual operation and rules for changes in the connectivity among the elements. Some attempts to combine symbolic AI techniques and connectionism [Shavilk 1994], to gain more advantages have been undertaken.

After a long period of research activity, rule based production systems still occupy a prominent place in AI, particularly in building programs simulating human problem solving. We also reviewed advantages and arguments against the use of production rules. A main disadvantage of production systems is their inherent inefficiency in the computational process. We also pointed out that researchers have investigated several

different methods to reduce the limitations and increase the speed of execution of production systems. One of these methods is based on applying parallel processing techniques to these systems. A review of these attempts formed the central subject of Chapter 2.

Chapter 2 provides the background knowledge in the field of parallel execution of production systems with the aim of improving the problem of slowness in execution of large production systems. While production systems have the advantages of modularity and ease of enhancement, they suffer from slow execution, especially in large-scale AI applications and real time tasks. Researchers have consequently investigated systems, using parallel processing techniques or distributed problem solving techniques that gain more speedups. These researches can be divided to parallel rule matching systems, parallel rule firing systems, distributed production systems, and multiagent production systems. This chapter briefly reviewed these attempts and concluded with a discussion on the suitable architecture supporting the proposed system in this work. We considered the structure of the system as a mesh-based medium-grain message-passing multicomputer (MIMD) using a physically distributed memory. The system offers higher memory bandwidth and potentially higher scalability than a shared memory architecture.

In Chapter 3, we discussed the genetic algorithm. Genetic algorithms represent a class of adaptive search methods that have been widely applied in various problems. Much of the interest in genetic algorithms is due to the fact that they provide a set of efficient domain independent search heuristics, which offer significant improvement over traditional methods without the need for incorporating intensive domain specific knowledge. This chapter has provided a brief overview of various implementations of the genetic algorithm and some of its applications relevant to work in this programme.

Chapter 4 introduced a distributed production system with the aim of addressing this problem: Given a complex problem, compose production systems agents into clusters of relevant agents by a process of selection so that each cluster provides an appropriate solution in some region of the state space. Then, combine these approximations to yield a globally satisfactory solution rather than extract the best solution. The central purpose

in the design of this system was to increase the solution creation rates by forming sub-solutions in parallel, thus gaining many of the potential benefits of distributed/co-operative problem solver systems. The mechanism of inference in the proposed knowledge based system was a coupling of task sharing and result sharing. The genetic algorithm, which is based on the idea of natural selection, was considered as an adaptive search technique for the problem of exploring a large state space for finding relevant problem solvers. A hierarchical model was considered as an appropriate structure for the storage of knowledge. Production rule representation was chosen as a suitable knowledge representation. De Jong [1988] states one reason that production systems have emerged as a favourite programming paradigm in both the expert system and machine learning communities is that they provide a knowledge representation approach that can simultaneously support two kinds of activities:

- Treating knowledge as data to be manipulated as part of a knowledge-acquisition and refinement process,
- Treating knowledge as an executable entity to be used in performing a particular task.

We introduced a parallel hybrid genetic algorithm that can be used to find an optimal solution, based on certain cost criteria, for configuration of clusters. The specification of the developed GA was discussed in more details. Pseudo-code describing the behaviour of the system was presented. The last section of the chapter can be regarded as the design stage of the distributed production system life cycle, proposed in this project. A modified structure chart for representing the simulated system design was used. The knowledge diagram, which is a graphical representation of a portion of the rule base and details how the rules connect facts together, was presented.

The central theme of Chapter 5 was testing and evaluating the proposed system. The structure of the simulation program in terms of global variables and data structures was described. Testing, along with verification and validation of the proposed system including verification of the inference engine and knowledge base, determination of testing methodology, and specification of the validation criteria were discussed. We tested the simulated system to verify the validity of the adopted approach. The

performance of the system was also evaluated by calculating the performance improvements of a set of benchmark programs on a compatible IBM PC. We gave some experimental data, which indicates speedups up to 2-fold have been obtained for test programs.

6.4 Discussion of the results

As aforementioned the system proposed in this thesis was considered as a distributed production system, with its main inference mechanism motivated by the theory of neuronal group selection. According to this theory, synaptic connections, which are established during development, are selectively strengthened or weakened to make circuits with strengthened synapses, called neuronal groups. Neuronal groups are local collections of hundreds or thousands of neurones that are strongly interconnected and vary functionally. We attempted to make a computational model of this mechanism by the DPSS.

On the other hand production rule based systems have emerged as an important tool for artificial intelligence researchers. Many problems are naturally formulated in terms of rules and rule based systems are amenable to rapid modification. We used rule-based representations to resemble neuronal groups in the brain. We developed a distributed production system with a self-organising co-operative interaction mechanism that allows multiple expert systems to work together to solve a complex problem. This is similar to the brain, where different neural subsystems combine and exchange signals to work together to yield an overall intelligent nervous system.

Distributed production systems attempt to reduce the processing time by executing different inference phases concurrently. To ensure that the parallel solution is equivalent to the sequential solution, two problems should be addressed: The compatibility and convergence problems. The compatibility problem is caused by elimination of the sequential conflict resolution strategy in the whole of the production system. Then there is a chance of firing a set of rules, which prevent other instantiations from being fired. Firing a set of compatible rule instantiations concurrently would reach the firing rules in

a sequential order. On the other hand, firing a set of compatible rules does not guarantee the correctness of the final solutions, because the right rule instantiation in a cycle may be wrong in the following cycles. As a result, the final parallel solution does not agree with serial solution of the program. We assumed the programmer writes programs in which all possible execution sequences reach the correct solution.

In this thesis, we showed the applicability of genetic algorithm methods to distributed production systems, as a search technique to cluster the relevant problem solvers, especially in cases where the space is large and poorly understood. For many practical domains of application, it is very difficult to construct strong domain theories to guide the process of structural change. Particularly, if the search space is large, a good deal of time and effort can be spent in developing domain-specific heuristics. We exploited the key feature of GAs, i.e., the ability to exploit accumulating information about an initially large complex, and unknown search space, in order to bias subsequent searches into useful subspaces. In the problem of searching for a solution in a large space, the computations tend to be unpredictably structured and have multiple solutions. The desired solution is usually specified by certain properties, and any state satisfying these properties is an acceptable solution. GA focuses on parallel exploration of the search space to find optimal solutions, based on certain cost measures, that is specially important in real-life applications. GA starts from a random generated set of possible solutions and applies the genetic operators to transform a state of the search space to another. Therefore, for a given problem the genetic algorithm may explore different solutions at different runs. The influence of these variations is that for a given problem, there are cases where the final solutions founded by the DPSS are not the same set. This problem is caused by the elimination of communication patterns between clusters, i.e. a processor node may request to communicate with other node(s) in another cluster. In our implementations, the GA is guided by applying a hill-climbing procedure which reduces differences in solutions.

Several other implications can be pointed out from the observation of the simulation results. Firstly, the proposed approach can not perform efficiently on production system with small sized knowledge bases. In general small production systems acceptably work in terms of speed of execution, in a sequential approach. Secondly, we should not

expect the addition of more processors on a fixed size knowledge base system improves the speedup of the parallel execution of rules. Also when applying rule-level parallelism in the system, we should concentrate on larger sized rule-sets. Thus, fine-grained rule-level parallelism is not suitable for the system. Thirdly, using stochastic search methods, such as genetic algorithms, are not cost effective enough to implement at run time processing. Fourthly, communication overhead is too small in the system. Fifthly, although the test results show that the increase in size of the knowledge base leads to improve the speedup of the system, in general there is no reason to expect that larger production systems will necessarily exhibit more speedups. The parallel speedups were less than 2-fold, dependent on the implementation of genetic algorithm. The main reasons can be explained by the following facts.

The main objective of the testing the system was to see whether the proposed problem solving system works or not. Then in our implementation, we did not use techniques to parallelise the match phase, which constitutes 90% of the run time and is a bottleneck that needs to be improved. Generally, it is the inherent nature of the computation involved in implementing production systems that prevents significant speedup for parallelism. A production system, which employs multiple rule firing techniques on the top of match level parallelism can obtain more speedups. In the evaluation of the production systems we should also consider the method of conflict resolution.

Although production systems appear to be highly parallel-able, in practice the speedup obtainable from parallelism is quite limited. We quote the conclusion of Rice [Neiman 1994]: 'The blackboard model, an appealing cognitive model for concurrent problem solving, does not necessarily work as well in practice as intuition might lead one to expect. A consequence of this is that, although we hoped to deliver many orders of magnitude of speedup due to parallelism, we have only been able to show off the order of one order of magnitude with some indications that this might scale to about two orders of magnitude.'

Another reason concerns with the formation of clusters of relevant processing elements based on implementation of the genetic algorithm. The implementation of a

genetic algorithm depends on several parameters, such as population size, choice of GA operators, operator probabilities, representation of variables, and method of execution of the algorithm, which interact in a complex way. Parallel execution of the algorithm highly improves the execution time, however in our implementation larger population sizes increase the execution times. This is the price to be paid for searching poorly understood spaces.

Aside from the performance measurements, it is also important to evaluate the complexity involved in requiring the programmer to guarantee the correctness of production programs. In addition to the above theoretical problems, there is a lack of research software and benchmarks in the field of production systems [Miranker 1991]. This makes it difficult to compare different research results. Because the number of available benchmark production systems is very small, research results may not reflect true behaviour of production systems. Even more critical is the fact that accurate research results are difficult to obtain because most of the benchmarks are sequential in nature.

6.5 Summary

The fields of knowledge representation and parallel processing have been the major areas of research in the history of AI. Different knowledge representation formalism and architectural approaches to build parallel system have been developed. Production rule systems show that the idea of using symbolic structures and symbol manipulation coupled to theories on heuristic problem solving and logical inference is a valid approach to knowledge based tasks. Parallel/distributed production systems are supporting examples of this claim. On the other hand, cognitive psychology has an important role to play. Despite the success of AI in certain areas, humans are still the most advanced intelligent systems that we know. Better understanding of human intelligence and knowledge representation methodologies will provide important insights to build intelligent systems, only when it is looked for the right means not just mimicking the brain function.

6.6 Further work

There are some points deserving our attention in future work. First of all is the role of knowledge in a distributed problem solving system [Conry et al 1990]. Successful problem solving requires that agents be able to cooperate with other agents based on their knowledge. When a reasoning task is assigned to a group of agents, each agent has incomplete knowledge. Therefore, it is possible for each agent to perform forever, due to incomplete knowledge, or the system can not prove a theorem that sequential systems can determine its validity. Consequently, a distributed automated reasoning system needs decision making intelligence heuristics to allow agents communicate to each other based on their local knowledge.

For the DPSS, an alternative is the development of other strategies, rather than using GA, to perform automated reasoning.

Another alternative is that we allow clusters communicate to each other, for example in a pipelined approach, to produce the final solution rather than combining the clusters' solutions. Therefore, the integration stage of problem solving methodology can be implemented in following way:

After running the PHGA to specify the clusters' configurations, PSs communicate to neighbouring nodes in the same cluster to find approximations for a given complex problem. As each cluster stops its computations, a corresponding cluster manager is created. Managers, whom are production rule systems with combination of facts and rules that each cluster member knows, communicate to each other. The process of communication-computation between clusters' managers continues until there is no any matching rule for each manager. The final result is the result produced by the clusters' mangers.

Moreover, several possible research directions can be suggested to improve the efficiency of the system.

- Representing knowledge by other methods, such as combining symbolic and neural networks or developing hybrid symbolic knowledge representations.

- Applying an efficient and parallel matching algorithm. The performance of the production system may be improved by parallelising not only the act phase but also the match phase.

- Applying other sources of parallelism, such as data parallelism which is the distribution of working memory elements over the processors.

- Applying competitive interaction mechanisms rather than co-operative mechanisms between problem solvers.

References

Acharya, A., and Tambe, M. (1989). 'Production systems on message passing computers: Simulation results and analysis'. *International Conference on Parallel Processing (ICPP-89), 246-254.*

Almasi, G.S. (1985). 'Overview of parallel processing'. *Parallel Computing, vol.2, no.3, 191-203.*

Ankenbrandt, C.A. (1991). 'An extension to the theory of convergence and a proof of the time complexity of genetic algorithms'. In *Foundation of Genetic Algorithms* (Rawlins, G.E. ed.).

Awad, E.M. (1996). *Building Expert Systems: Principles, Procedures, and Applications.* West Publishing Company.

Ayel, M., and Laurent, J.P. (eds.) (1991). *Validation, Verification and Testing of knowledge-Based Systems.* John Wiley.

Back, T. (1995). 'Artificial evolution'. *Lecture Notes in Computer Science (LNCS), vol.1063, 3-20.*

Backer, J.E. (1985). 'Adaptive selection methods for genetic algorithms'. *International Conference on Genetic Algorithm (ICGA-85), 101-111.*

Bahr, E., Barachini, F., and Doppelbauer, J. (1991). 'A parallel production system architecture'. *Journal of Parallel and Distributed Computing, 13, 4, 456-462.*

Banks, J. (ed.) (1998). *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice.* John Wiley.

Barr, A., Cohen, P.R., and Feigenbaum, E.A. (1989). *The Handbook of Artificial Intelligence.* vols.II, IV. Addison- Wesley.

Barto, A.G. (1995). 'Learning as hill climbing in weight space'. In *The Handbook of Brain Theory and Neural Networks* (Arbib, M.A. ed.), MIT Press.

Berra, P.B., Mitkas, P.A. (1988). 'An initial design of a very large knowledge base architecture'. In *Information Technology for Organisational* Systems (Bullinger, H.J. et al eds.), Elsevier Science.

Bhuyan, J.N., Raghavan, V.V., and Elayavalli, V.K. (1991). 'Genetic algorithm for clustering with an ordered representation'. *International Conference on Genetic Algorithms (ICGA-91), 408-415.*

Bledsoe, W.W., and Loveland, D.W. (eds.) (1984). *Automated Theorem Proving: After 25 years.* American Mathematical Society.

Boden, M.A. (ed.) (1990). *The Philosophy of Artificial Intelligence.* Oxford University Press.

Bomans, L., and Roose, D. (1989). 'Benchmarking the iPSC/2 hypercube multiprocessor'. *Concurrency: Practice and Experience, vol.1, 3-18.*

Bond, A., and Gasser, L. (1988). *Readings in Distributed Artificial Intelligence.* Morgan Kaufmann.

Booker, L. (1987). 'Improving search in genetic algorithms'. In *Genetic Algorithms and Simulated Annealing* (Davis, L. ed.), 61-73, Morgan Kaufmann.

Browrston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley.

Buchanan, B., Sutherland, G., and Feigenbaum, E. (1969). 'Heuristic DENDARL: A program for generating explanatory hypothesis in organic chemistry'. In *Machine Intelligence 4* (Meltzer, B., and Michie, D. eds.).

Buchanan, B.G., and Shortliffe, E.H. (1984). *Rule based Expert Systems.* Addison-Wesley.

Buckles, B.P., Petry, F.E., and Kuester, R.L. (1990). 'Schema survival rates and heuristic search in genetic algorithms'. *Proceedings of Tools for Artificial Intelligence, 322-327.*

Budgen, D. (1994). *Software Design*. Addison-Wesley.

Calegari, P., Guidec, F., Kuonen, P., and Kobler, D. (1997). 'Parallel island-based genetic algorithm for radio network design'. *Journal of Parallel and Distributed Computing, 47, 86-90.*

Carbonell, J.G. (1989). 'Introduction: Paradigms for machine learning'. *Artificial Intelligence, 40, 1-9.*

Carpenter, G.A., and Grossberg, S. (1988). 'The ART of adaptive pattern recognition by self-organising neural networks'. *Computer, Mar, 77-88.*

Charles, E., and Dubois, O. (1991). 'MELODIA: Logical methods for checking K-bases'. In *Validation, Verification and Test of Knowledge Based Systems* (Ayel, M., and Laurent, J.P. eds.), 95-105, John Wiley.

Chipperfield, A., and Fleming, P. (1994). *Parallel Genetic Algorithms: A Survey.*

Chu, W.W., Holloway, L.J., Lan, M.T., and Efe, K. (1980). 'Task allocation in distributed data processing'. *Computer, November, 57-69.*

Clark, A. (1995). 'Philosophical issues in brain theory and connectionism'. In *The Handbook of Brain Theory and Neural Networks* (Arbib, M.A. ed.), MIT Press.

Cohoon, J.P., Martin, W.N., and Richards, D.S. (1991). 'A multi-population genetic algorithm for solving the K-partition problem on hypercubes'. *International Conference on Genetic Algorithms (ICGA-91), 244-248.*

Conry, S.E., MacIntosh, D.J., and Meyer, R.A. (1990). 'DARES: A distributed automated reasoning system'. *National Conference on Artificial Intelligence (AAAI-90), 78-85.*

Cook, D.J., and Hannon, C. (1999). 'Adaptive parallel search for theorem proving'. *International Florida AI Research Society Conference.*

Corkill, D.D., and Lesser, V.R. (1983). 'The use of meta-level control for coordination in a distributed problem solving network'. *International Joint Conference on Artificial Intelligence (IJCAI-83), 748-756.*

Cremonesi, P., Rosti, E., Serazzi, G., and Smirni, E. (1999). 'Performance evaluation of parallel systems'. *Parallel Computing, vol.25, no.13-14, 1677-1698.*

Culler, D.E., Jaswinder, P.S. with Gupta, A. (1999). *Parallel Computer Architecture: A Hardware/ Software Approach.* Morgan Kaufmann.

Davis, L. (1989). 'Adapting operator probabilities in genetic algorithms'. *International Conference on Genetic Algorithms (ICGA-89), 61-69.*

Davis, L. (ed.) (1991). *Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

Davis, R., and Smith, G. (1983). 'Negotiation as a metaphor for distributed problem solving'. *Artificial Intelligence, vol. 20, 63-109.*

De Jong, K. (1988). 'Learning with genetic algorithms: An overview'. *Machine Learning 3, 121-138.*

De Jong, K.A., and Spears, W.M. (1989). 'Using genetic algorithms to solve NP-complete problems'. *International Conference on Genetic Algorithms (ICGA-89), 124-132.*

Deb, K., and Agrawal, S. (1999). 'Understanding interactions among genetic algorithm parameters'. In *Foundation of Genetic Algorithms.5* (Banzhaf, W., and Reeues, C. eds.).

Deen, S.M. (1985). *Principles and Practice of Database Systems.* Macmillan Publishers.

Dixit, V.V., and Moldovan, D.I. (1990). 'The allocation problem in parallel production systems'. *Journal of Parallel and Distributed Computing, 8, 20-29.*

Doran, J. (1992). 'Distributed AI and its applications'. *Lecture Notes in Artificial Intelligence (LNAI), vol. 617, 368-372.*

Durfee, E.H. (1995). 'Distributed artificial intelligence'. In *The Handbook of Brain Theory and Neural Networks* (Arbib, M.A. ed.), MIT Press.

Durfee, E.H., and Lesser, V.R. (1987). 'Using partial global plans to coordinated distributed problem solvers'. *International Joint Conference on Artificial Intelligence (IJCAI-87), 875-883.*

Durfee, E.H., Lesser, V.R., and Corkill, D.D. (1987). 'Coherent cooperation among communicating problem solvers'. *IEEE Transactions on Computers, vol.36, 1275-1291.*

Edelman, G.M. (1989). *The Remembered Present: A Biological Theory of Consciousness*. Basic Books, New York.

Edelman, G.M. (1992). *Bright Air, Brilliant Fire: On the Matter of the Mind.* Basic Books, New York.

Efe, K. (1982). 'Heuristic models of task assignment scheduling in distributed systems'. *Computer, June, 50-56.*

Ehrig, H. (1978). 'Introduction to algebraic theory of graph grammars'. *Lecture Notes in Computer Science (LNCS), vol.73.*

Ensor, J.R., and Gabbe, J.D. (1985). 'Transactional blackboard'. *International Joint Conference on Artificial Intelligence (IJCAI-85), 340-344.*

Eshelman, L.J., Caruana, R.A., and Schaffer, J.D. (1989). 'Biases in the crossover landscape'. *International Conference on Genetic Algorithms (ICGA-89), 10-19.*

Fennel, R.D., and Lesser, V.R. (1981). 'Parallelism in artificial intelligence problem solving: A case study of Hearsay-II'. *IEEE Transactions on Computers, vol. 26, no. 2, 98-211.*

Flynn, M.J. (1972). 'Some computer organisations and their effectiveness'. *IEEE Transactions on Computers, 21(9), 948-960.*

Fogel, D.B. (1995). *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press.

Fogel, D.B., and Fogel, L.J. (1995). 'Artificial evolution'. *Lecture Notes in Computer Science (LNCS), vol.1063, 21-33.*

Forgy, C.L. (1982). 'RETE: A fast algorithm for the many pattern/ many object pattern match problem'. *Artificial Intelligence, no. 19, 17-37.*

Franklin, S. (1995). *Artificial Minds*. MIT Press.

Frost, R.A. (1986). *Introduction to Knowledge Base Systems*. Collins.

Georgeff, M.P. (1982). 'Procedural control in production systems'. *Artificial Intelligence, vol.18, 175-201.*

Giarratano, J., and Riley, G. (1989). *Expert Systems: Principles and Programming.* PWS-KENT.

Goldberg, D.E. (1988). *Genetic Algorithms in Search, Optimisation, and Machine Learning.* Addison-Wesley.

Goldberg, D.E. (1989). 'Sizing populations for serial and parallel genetic algorithms'. *International Conference on Genetic Algorithms (ICGA-89), 70-79.*

Gonzalez, A., and Douglas, D.D. (1993). *The Engineering of Knowledge-Based Systems: Theory and Practice.* Prentice-Hall.

Grefenstette, J.J. (1986). 'Optimisation of control parameters for genetic algorithms'. *IEEE Transactions on Systems, Man, and Cybernetics, vol. 16, no. 1, 122-128.*

Grefenstette, J.J. (1987). 'Incorporating problem specific knowledge into genetic algorithms'. In *Genetic Algorithms and Simulated Annealing* (Davis, L. ed.), 42-60, Morgan Kaufmann.

Grefenstette, J.J. (1989). 'A system for learning control strategies with genetic algorithms'. *International Conference on Genetic Algorithms (ICGA-89), 183-190.*

Grefenstette, J.J., and Baker, J.E. (1989). 'How genetic algorithms work: A critical look at implicit parallelism'. *International Conference on Genetic Algorithms (ICGA-89), 20-27.*

Gupta, A. (1985). 'Parallelism in production systems: The sources and the expected speedup'. *International Workshop on Expert Systems and Their Applications, 25-57.*

Gupta, A. (1987). *Parallelism in Production Systems.* Morgan Kaufmann.

Gupta, A., and Tambe, M. (1988). 'Suitability of message passing computers for implementing production systems'. *National Conference on Artificial Intelligence (AAAI-88).*

Gupta, A., Forgy, C., Newell, A., and Wedig, R. (1986). 'Parallel algorithms and architectures for rule based systems'. *International Symposium on Computer Architecture.*

Harrison, P.R. (1990). *Common Lisp and Artificial Intelligence.* Prentice Hall.

Hartigan, J.A. (1975). *Clustering Algorithms.* John Wiley.

Harvey, W., Kalp, D., Tambe, D., Mckeown, D., and Newell, A. (1991). 'The effectiveness of task-level parallelism for production systems'. *Special Issue on the Parallel Execution of Rule Systems, Journal of Parallel and Distributed Computing, vol.13, 395-411.*

Hayes-Roth, B. (1985). 'A blackboard architecture for control'. *Artificial Intelligence, vol. 26, 251-321.*

Hayes-Roth, F., Waterman, D.A., and Lenat, D.B. (1983). *Building Expert Systems.* Addison-Wesley.

Hebb, D.O. (1949). *The Organisation of Behaviour: A Neuropsychological Theory.* Wiley.

Highland, F.D., and Iwaskiw, C.T. (1989). 'Knowledge base compilation'. *International Joint Conference on Artificial Intelligence (IJCAI-89).*

Hills, W.D. (1985). *The Connection Machine.* MIT press.

Holland, J. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press.

Hord, R.M. (1993). *Parallel Supercomputing in MIMD Architectures*. CRC Press.

Hwang, K., and Degroot, D. (eds.) (1989). *Parallel Processing for Supercomputers and Artificial Intelligence*. McGraw-Hill.

Hwang, K., and Xu, Z. (1998). *Scalable Parallel Computing*. McGraw- Hill.

Hwang, K., Ghosh, J., and Chowkwanyun, R. (1987). 'Computer architectures for artificial intelligence processing'. *Computer, vol.20, no.1, 19-29.*

Ibens, O. (1999). 'Automated theorem proving with disjunctive constraints'. *Lecture Notes in Computer Science (LNCS), VOL.1713.*

Intrator, N. (1995). 'Competitive learning'. In *The Handbook of Brain Theory and Neural Networks* (Arbib, M.A. ed.), MIT Press.

Ishida, T. (1990). 'Methods and effectiveness of parallel rule firing'. *Conference on Artificial intelligence Applications.*

Ishida, T. (1991). 'Parallel rule firing in production systems'. *IEEE Transactions on Knowledge and Data Engineering,, vol.3, no.1, 11-17.*

Ishida, T. (1994). *Parallel, Distributed, and Multiagent Production Systems.* Springer-Verlag.

Ishida, T., and Stolfo, S.J. (1985). 'Towards the parallel execution of rules in production system programs'. *International Conference on Parallel Processing.*

Ishida, T., Gasser, L., and Yokoo, M. (1992). 'Organisation self-design of distributed production systems'. *IEEE Transactions on Knowledge and Data Engineering, vol. 4, no. 2, 123-134.*

Ishida, T., Sasaki, Y., Nakata, K., and Fukuhara, Y. (1994). 'A meta-level control architecture for productions systems'. *IEEE Transactions on Knowledge and Data Engineering,.*

Ishida, T., Yokoo, M., and Gasser, L. (1990). 'An organisational approach to adaptive production systems'. *National Conference on Artificial Intelligence (AAAI-90), 52-58.*

Jafar, M., and Bahil, A.T. (1993). 'Interactive verification of knowledge-based systems'. *IEEE-Expert, vol.8, no.1, 25-32.*

Janikow, C.Z., and Michalewicz, Z. (1991). 'An experimental comparison of binary and floating point representations in genetic algorithms'. *International Conference on Genetic Algorithm (IGCA-91), 31-36.*

Jones, D.R., and Beltramo, M.A. (1991). 'Solving partitioning problems with genetic algorithms'. *International Conference on Genetic Algorithms (ICGA-91), 442-449.*

Kandel, A. (ed.) (1992). *Fuzzy Expert Systems*. Boca Raton.

Kelly, M., and Seviora, R. (1989). 'An evaluation of DRete on CUPID for OPS5'. *International Joint Conference on Artificial Intelligence (IJCAI-89), 84-90.*

Kitano, H. (1990). 'Designing neural networks using genetic algorithms with graph generation system'. *Complex Systems, 4, 461-476.*

Kitano, H. (1993). 'Challenges of massive parallelism'. *International Joint Conference on Artificial Intelligence (IJCAI), 813-834.*

Kitano, H., Hunter, L., Wah, B., v.Hahn, W., Oka, R., and Yokoi, T. (1993). 'Grand challenge AI applications'. *International Joint Conference on Artificial Intelligence (IJCAI), 1677-1683.*

Kitano, H., Smith, S. F., and Higuchi, T. (1991). 'A parallel associative memory processor for rule learning with genetic algorithms'. *International Conference on Genetic Algorithms (ICGA-91), 311-317.*

Kohonen, T. (1990). 'The self-organisation map'. *Proc. IEEE, vol. 78, 1464-1480.*

Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.

Kuo, C.M., Miranker, D.P., and Browne, J.C. (1991). 'On the performance of the CREL system'. *Journal of Parallel and Distributed Computing, 13, 424-441.*

Kuo, S., and Moldovan, D. (1991). 'Implementation of multiple rule firing production systems on hypercube'. *Special Issue on the Parallel Execution of Rule Systems, Journal of Parallel and Distributed Computing,* vol. *13, 383-394.*

Kuo, S., and Moldovan, D. (1992). 'The state of the art in parallel production systems'. *Journal of Parallel and Distributed Computing, vol.15, 1-26.*

Laird, J.E., Rosenbloom, P.S., and Newell, A. (1987). 'Soar: An architecture for general intelligence'. *Artificial Intelligence, vol.33, 1-64.*

Lamont, G.B., and Shakley, D.J. (1988). 'Parallel expert system search techniques for a real-time application'. *Communication of the ACM, 1352-1359.*

Lopez, B. (1991). 'CONKRET: A control knowledge refinement tool'. In *Validation, Verification and Test of Knowledge Based Systems* (Ayel, M., and Laurent, J.P. eds.), 191-206, John Wiley.

Luger, G.F., and Stubbefield, W.A. (1989). *Artificial Intelligence and the Design of Expert Systems.* Benjamin/Cummings.

Manderick, B., and Spiessens, P. (1989). 'Fine-grained parallel genetic algorithms'. *International Conference on Genetic Algorithms (ICGA-89), 428-433.*

Manner, R., and Manderick, B. (eds.) (1992). *Parallel Problem Solving from Nature, 2.* North-Holland.

Mansour, N., and Fox, G.C. (1991). 'A hybrid genetic algorithm for task allocation in multicomputers'. *International Conference on Genetic Algorithms (ICGA-91), 466-473.*

Mansour, N., and Fox, G.C. (1992). 'Allocating data to multicomputer nodes by physical optimisation algorithms for loosely synchronous computations'. *Concurrency: Practice and Experience, 4(7), 557-574.*

Maren, A., Harston, C., and Pap, R. (1990). *Handbook of Neural Computing Applications.* Academic Press.

Matwin, S., Szapiro, T., and Haigh, K. (1991). 'Genetic algorithms approach to a negotiation support systems'. *IEEE Transactions on Systems, Man, and Cybernetics, vol.21, no.1. 102-114.*

Maulik, U., and Bandyopadhyay, S. (2000). 'Genetic algorithm-based clustering technique'. *Pattern Recognition, vol.33, no.9, 1455-1465.*

McDormett, D., and Forgy, C. (1978). 'Production system conflict resolution strategies'. In *Pattern Directed Inference Systems* (Waterman, D., and Hayes-Roth, F. eds.), Academic Press.

Michalewicz, Z. (1996). *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag.

Miller, G.F., and Todd, P.M. (1989). 'Designing neural networks using genetic algorithms'. *International Conference on Genetic Algorithms (ICGA-89), 379-384.*

Minsky, M., and Papert, S. (1969). *Perceptron: An Introduction to Computational Geometry.* MIT Press.

Miranker, D.P. (1987). 'TREAT: A better match algorithm for AI production systems'. *National Conference on Artificial Intelligence (AAAI-87), 42-47.*

Miranker, D.P. (1991). 'Special issue on the parallel execution of rule systems: Guset editor's introduction'. *Journal of Parallel and Distributed Computing, 13, 345-347.*

Miranker, D.P., Kuo, C.M., and Browne, J.C. (1990). 'Parallelising compilation of rule-based programs'. *International Conference on Parallel Processing, 247-251.*

Moldovan, D. (1985). 'SNAP: A VLSI architecture for artificial intelligence processing'. *Journal of Parallel and Distributed Computing, vol.2, no.2, 109-131.*

Moldovan, D. (1986). 'A model for parallel processing of production systems'. *IEEE International Conference on Systems, Man, and Cybernetics, 568-573.*

Moldovan, D., and Parisi-Presicce, F. (1986). 'Parallelism analysis in rule-based system using graph grammars'. *International Workshop on Graph-Grammars and Their Application to Computer Science, 427-439.*

Moldovan, D.I. (1989). 'RUBIC: A multiprocessor for rule-based systems'. *IEEE Transactions on systems, Man, and Cybernetics, vol.19, no.4, 699-706.*

Moldovan, D.I. (1993). *Parallel Processing: From Applications to Systems.* Morgan Kauffmann.

Morgan, K. (1988). 'BLITZ: A rule-based system for massively parallel architectures'. *Conference on Hypercube Multiprocessors.*

Muhlenbein, H. (1989). 'Parallel genetic algorithms, population genetics, and combinatorial optimisation'. *International Conference on Genetic Algorithms (ICGA-89), 416- 421.*

Myers, G.J. (1979). *The Art of Software Testing*. John Wiley.

Mylopoulos, J., Wang, H., and Kraner, B. (1993). 'Knowbel: A hybrid tool for building expert systems'. *IEEE- Expert: Intelligent Systems and Their Applications, vol.8, no.1, 17-24.*

Neiman, D.E. (1994). 'Issues in the design and control of parallel rule-firing production systems'. *Journal of Parallel and Distributed Computing, 23, 338-363.*

Newell, A., and Simon, H.A. (1972). *Human Problem Solving.* Prentice- Hall.

Newell, A., and Simon, H.A. (1990). 'Computer science as empirical enquiry: Symbols and search'. In *The Philosophy of Artificial Intelligence* (Boden, M.A. ed.), Oxford University Press.

Nguyen, T.A., Perkins, W.A., Laffery, T.J., and Pecora, D. (1985). 'Checking an expert system knowledge base for consistency and completeness'. *International Joint Conference on Artificial Intelligence (IJCAI-85), 375-378.*

Nii, P.H. (1986). 'Blackboard systems, blackboard application systems, blackboard systems from a knowledge engineering perspective'. *AI Magazine, August, 82-106.*

O'Keefe, K.M., and O'Leary, D.E. (1993). 'Expert system verification and validation: A survey and tutorial'. *Artificial Intelligence Review, vol.7, 3-42.*

Petty, C.P., Leuze, M.R., and Grefenstette, J.J. (1987). 'A parallel genetic algorithm'. *International Conference on Genetic Algorithms (ICGA-87), 155- 161.*

Pfeifer, R., Schreter, Z., Fogelman-Soulie, F., and Steels, L. (1989). 'Putting connectionism into perspective'. In *Connectionism in Perspective* (Pfeifer, R., Schreter, Z., Fogelman-Soulie, F., and Steels, L. eds.), xi-xxi, Elsevier Science.

Plander, I. (ed.) (1989). 'Parallel computer architectures for knowledge information processing systems'. In *Artificial Intelligence and Information-Control Systems of Robots-89.* Elsevier Science.

Pollack, J. B. (1989). 'Connectionism: Past, present, and future'. *Artificial Intelligence Review, 3,3-20.*

Ralston, A., Reilly, E.D. (eds.) (1993). *Encyclopaedia of Computer Science.* Chapman & Hall.

Reeke, G.N., JR., and Sporns, O. (1993). 'Behaviourally based modelling and computational approaches to neuroscience'. *Ann. Rev. Neurosci.16, 597-623.*

Refenes, A.N. (1989). 'Parallelism in knowledge-based machines'. *The Knowledge Engineering Review, vol. 4, no. 1, 53-71.*

Reichgelt, H. (1991). *Knowledge Representation: An AI Perspective.* Ablex Publishing Corporation.

Ringland, G.A., and Duce, D.A. (1989). *Approaches to Knowledge Representation: An Introduction.* John Wiley.

Robertson, G, (1987). 'Parallel implementation of genetic algorithms in classifier systems'. In *Genetic Algorithms and Simulated Annealing* (Davis, L. ed.), 129-140, Morgan Kaufmann.

Ronald, S. (1997). 'Robust encodings in genetic algorithms: A survey of encoding issues'. *IEEE International Conference on Evolutionary Computation, 43-48.*

Rosenblatt, F. (1962). *Principles of Neurodynamics.* Spartan Books.

Rudolph, G. (1994). 'Convergence properties of canonical genetic algorithms'. *IEEE Transactions on Neural Networks, vol.5, no.1, 96-101.*

Rumelhart, D.E., and Zipser, D. (1985). 'Feature discovery by competitive learning'. *Cognitive Science, 9, 75-112.*

Schaffer, J.D., Caruana, R.A. Eshelman, L.J., and Das, R. (1989). 'A study of control parameters affecting online performance of genetic algorithms for function optimisation'. *International Conference on Genetic Algorithms (ICGA-89), 51-60.*

Schmolze, J. (1990). 'A parallel asynchronous distributed production system'. *National Conference on Artificial Intelligence (AAAI-90), 65-71.*

Schmolze, J.g. (1989). 'Guaranteeing serializable results in synchronous parallel production systems'. *International Joint Conference on Artificial Intelligence.*

Schmolze, J.G. (1991). 'Guaranteeing serialisable results in synchronous parallel production systems'. *Special Issue on the Parallel Execution of Rule Systems, Journal of Parallel and Distributed Computing, vol.13, 348-365.*

Schmolze, J.G., and Goel, S. (1990). 'A parallel asynchronous distributed production system'. *National Conference on Artificial Intelligence (AAAI-90), 65-71.*

Schmolze, J.G., and Synder, W. (1997). 'Detecting redundant production rules'. *Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference (AAAI-IAAI), 417-423.*

Schoneveld, A., de Ronde, J.F., and Sloot, P.M.A. (1997). 'Task allocation by parallel evolutionary computing'. *Journal of Parallel and Distributed Computing, 47, 91-97.*

Schraudolph, N.N, and Belew, R.K. (1992). 'Dynamic parameter encoding for genetic algorithms'. *Machine Learning, vol.9, no.1, 9-21.*

Searle, J.R. (1990). 'Minds, brains, and programs'. In *The Philosophy of Artificial Intelligence* (Boden, M.A. ed.), Oxford University Press.

Shavlik, W. (1994). 'Combining symbolic and neural learning'. *Machine Learning, 14, 321-331.*

Shaw, D.E. (1985). 'NON-VON's applicability of three AI task areas'. *International Joint Conference on Artificial Intelligence (IJCAI-85), 61-72.*

Shirazi, B.A., Hurson, A.R., and Kavi, K.M. (eds.) (1995). *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Computer Society Press.

Shortliffe, E. (1976). *Computer-Based Medical Consultation: MYCIN.* American Elsevier.

Simpson, P.K. (1992). 'Foundations of neural networks'. In *Artificial Neural Networks: Paradigms, Applications, and Hardware Implementations* (Sanchez-Sinencio, E., and Lau, C. eds.). IEEE Press.

Slavov, V., and Nikolaev, N.I. (1999). 'Genetic algorithms, fitness sublandscapes and subpopulations'. In *Foundation of Genetic Algorithms.5* (Banzhaf, W., and Reeues, C. eds.).

Smith, R. (1980). 'The contract net protocol: High-level communication and control in a distributed problem solver'. *IEEE Transactions on Computers, 29, 1104-1113.*

Smith, R.G., and Davis, R. (1980). 'Frameworks for cooperation in distributed problem solving'. In *Readings in Distributed Artificial Intelligence* (Bond, A.H., and Gasser, L. eds.), Morgan Kaufmann.

Spiessens, P., and Manderick, B. (1991). 'A massively parallel genetic algorithm: Implementation and first analysis'. *International Conference on Genetic Algorithms (ICGA-91), 279- 285.*

Stender, J. (ed.) (1993). *Parallel Genetic Algorithms: Theory and Applications*. IOS Press.

Stenz,G., and Wolf, A. (1999a). 'Strategy selection by genetic programming'. *International Florida AI Research Society Conference, 346-50.*

Stenz,G., and Wolf, A. (1999b). 'E-SETHEO: Design, configuration and use of a parallel automated theorem prover'. *Lecture Notes in Artificial Intelligence (LNAI), vol.1747, 231-243.*

Stenz,G., and Wolf, A. (2000). 'Scheduling methods for parallel automated theorem proving'. *Lecture Notes in Artificial Intelligence (LNAI), vol.1822, 254-266.*

Stolfo, S.J. (1987). 'Initial performance of the DADO2 prototype'. *Computer, vol.20, no.1, 75-83.*

Stolfo, S.J., and Miranker, D.P. (1986). 'The DADO production system machine'. *Journal of Parallel and Distributed Computing, 3, 269-296.*

Stolfo, S.J., Miranker, D., and Shaw, D.E. (1981). 'Architecture and applications of DADO: A large-scale parallel computer for artificial intelligence'. *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-81).*

Stolfo, S.J., Wolfson, O. Chan, P.K., Dewan, H.M., Woodbury, L., Glazier, J.S., and Ohsie, D.A. (1991). 'PARULEL: Parallel rule processing using meta-rules for redactions'. *Journal of Parallel and Distributed Computing, 13, 366-382.*

Syswerda, G. (1989). 'Uniform crossover in genetic algorithms'. *International Conference on Genetic Algorithms (ICGA-89).*

Tanese, R. (1987). 'Parallel genetic algorithm for a hypercube'. *International Conference on Genetic Algorithms (ICGA-87), 177- 183.*

Tanese, R. (1989). 'Distributed genetic algorithms'. *International Conference on Genetic Algorithms (ICGA-89), 434-439.*

Tarassenko, L. (1998). *A Guide to Neural Computing Applications.* Wiley.

Tenorio, F.M., and Moldovan, D.I. (1985). 'Mapping production systems into multiprocessors'. *International Conference on Parallel Processing.*

Trew, A., and Wilson, G. (eds.) (1991). *Past, Present, Parallel: A Survey of Available Parallel Computer Systems*. Springer- Verlag.

Ullman, J.D. (1982). *Principles of Database Systems.* Pitman.

Van Melle, W.J. (1981). *System Aids in Constructing Consultation Programs.* Ann Arbor MI: UMI Research Press.

Wah, B.W., and Ramamoorthy, C.V. (eds.) (1990). *Computers for Artificial Intelligence Processing.* John Wiley.

Waterman, D.A. (1986). *A Guide to Expert Systems.* Addison-Wesley.

Whitehead, A.N., and Russell, B. (1910-1913). *Principa Mathematica.* Cambridge University Press.

Whitley, D. (1989). 'The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trails is best'. *International Conference on Genetic Algorithms (ICGA-89), 116-121.*

Winston, P. (1984). *Artificial Intelligence.* Addison-Wesley.

Winston, P. (1989). L*isp.* Third edition. Addison-Wesley.

Xu, J., and Hwang, K. (1991). 'Mapping rule-based systems onto multicomputers using simulated annealing'. *Journal of Parallel and Distributed Computing, 13, 442-455.*

Zargham, M.R. (1996*). Computer Architecture: Single and Parallel Systems*. Prentice-Hall.

Appendix A: The Simulation Program

A.1 Globals for the simulation program

```
-----------------------------------------------------------------------------------------------------------
DATA STRUCTURES
-----------------------------------------------------------------------------------------------------------
(setf rectangular-array (make-hash-table))
(defstruct PE rule-base fact-base agenda)
(defstruct rule lhs rhs)
-------------------------------------------------------------------------------------------------------------
MACROS THAT SUPPORT DATA STRUCTURES
-------------------------------------------------------------------------------------------------------------


(defmacro build-PE (name rule-base fact-base agenda)
  `(setf ,name (make-PE
               :rule-base ',rule-base
               :fact-base ',fact-base
               :agenda ',agenda)))
(defmacro build-array
(name rule-base fact-base agenda &optional rectangular-array)
  `(setf (gethash ',name rectangular-array)
    (make-PE
         :rule-base ',rule-base
```

```
        :fact-base ',fact-base
        :agenda ',agenda)))

(defmacro add-object (object variable)
  `(unless (member ,object (mapcar #'rest ,variable)
             :test #'equal)
     (setf ,variable (append ,variable
                       (cons (cons (add-tag) ,object)'())))))

(let ((time-tag 0))
  (defun add-tag ()
    (setf time-tag (1+ time-tag))))
```

REQUIERD PROCESSING ELEMENTS

```
(build-PE PE1 rule-base-1 fact-base-1 agenda-1)
(build-PE PE2 rule-base-2 fact-base-2 agenda-2)
(build-PE PE3 rule-base-3 fact-base-3 agenda-3)
(build-PE PE4 rule-base-4 fact-base-4 agenda-4)
(build-PE PE5 rule-base-5 fact-base-5 agenda-5)
(build-PE PE6 rule-base-6 fact-base-6 agenda-6)
(build-PE PE7 rule-base-7 fact-base-7 agenda-7)
(build-PE PE8 rule-base-8 fact-base-8 agenda-8)
(build-PE PE9 rule-base-9 fact-base-9 agenda-9)
(build-PE PE10 rule-base-10 fact-base-10 agenda-10)

(build-array PE1 rule-base-1 fact-base-1 agenda-1)
(build-array PE2 rule-base-2 fact-base-2 agenda-2)
(build-array PE3 rule-base-3 fact-base-3 agenda-3)
(build-array PE4 rule-base-4 fact-base-4 agenda-4)
(build-array PE5 rule-base-5 fact-base-5 agenda-5)
(build-array PE6 rule-base-6 fact-base-6 agenda-6)
(build-array PE7 rule-base-7 fact-base-7 agenda-7)
(build-array PE8 rule-base-8 fact-base-8 agenda-8)
(build-array PE9 rule-base-9 fact-base-9 agenda-9)
(build-array PE10 rule-base-10 fact-base-10 agenda-10)
```

```
(defvar FB (list (PE-fact-base PE1)
           (PE-fact-base PE2)
           (PE-fact-base PE3)
           (PE-fact-base PE4)
           (PE-fact-base PE5)
           (PE-fact-base PE6)
           (PE-fact-base PE7)
           (PE-fact-base PE8)
           (PE-fact-base PE9)
           (PE-fact-base PE10)))

(defvar RB (list (setf (PE-rule-base PE1) (make-hash-table))
           (setf (PE-rule-base PE2) (make-hash-table))
           (setf (PE-rule-base PE3) (make-hash-table))
           (setf (PE-rule-base PE4) (make-hash-table))
           (setf (PE-rule-base PE5) (make-hash-table))
           (setf (PE-rule-base PE6) (make-hash-table))
           (setf (PE-rule-base PE7) (make-hash-table))
           (setf (PE-rule-base PE8) (make-hash-table))
           (setf (PE-rule-base PE9) (make-hash-table))
           (setf (PE-rule-base PE10) (make-hash-table))))

(defvar AA (list (PE-agenda PE1)
           (PE-agenda PE2)
           (PE-agenda PE3)
           (PE-agenda PE4)
           (PE-agenda PE5)
           (PE-agenda PE6)
           (PE-agenda PE7)
           (PE-agenda PE8)
           (PE-agenda PE9)
           (PE-agenda PE10)))

(setf DR (list
         "f:\\simulati\\testing\\rset11.lsp"
         "f:\\simulati\\testing\\rset12.lsp"
```

```
        "f:\\simulati\\testing\\rset13.lsp"
        "f:\\simulati\\testing\\rset14.lsp"
        "f:\\simulati\\testing\\rset15.lsp"
        "f:\\simulati\\testing\\rset21.lsp"
        "f:\\simulati\\testing\\rset22.lsp"
        "f:\\simulati\\testing\\rset23.lsp"
        "f:\\simulati\\testing\\rset24.lsp"
        "f:\\simulati\\testing\\rset25.lsp"))

(setf files-list '("f:\\simulati\\testing\\h11.lsp"
            "f:\\simulati\\testing\\h12.lsp"
            "f:\\simulati\\testing\\h13.lsp"
            "f:\\simulati\\testing\\h14.lsp"
            "f:\\simulati\\testing\\h15.lsp"
            "f:\\simulati\\testing\\h21.lsp"
            "f:\\simulati\\testing\\h22.lsp"
            "f:\\simulati\\testing\\h23.lsp"
            "f:\\simulati\\testing\\h24.lsp"
            "f:\\simulati\\testing\\h25.lsp"))
------------------------------------------------------------------------------------------------
GLOBALS FOR THE GA
------------------------------------------------------------------------------------------------


(defvar *population-size*)
(defvar *number-of-PEs*)
(defvar *number-of-clusters*)
(defvar *KB*)
------------------------------------------------------------------------------------------------
PARAMETERS
------------------------------------------------------------------------------------------------


(setf *number-of-PEs* 10)
(setf *population-size* 7)
(setf *number-of-clusters* 5)
(setf *KB* '(1 1 4 2 3 3 4 0 2 1))
```

```
-----------------------------------------------------------------------
OPERATOR RATES
-----------------------------------------------------------------------

(setf Pc 0.6)
(setf Pm 0.002)
(setf Pinversion 0.02)

-----------------------------------------------------------------------
PROCESSING ELEMENTS COORDINATES IN THE ARRAY
-----------------------------------------------------------------------

(setf PE-coordinates '((0 0) (1 0) (2 0)
                       (0 1) (1 1) (2 1)
                       (0 2) (1 2) (2 2)
                       (0 3)))
```

A.2 The program main body

```
(defun the-program ()
  (initialise)
  (print (integrate-partial-result
   (get-partial-results-from-clusters
    (find-clusters)))))
-----------------------------------------------------------------------
INITIALISING THE SYSTEM
-----------------------------------------------------------------------

(defun initialise ()
  (setf time-list nil)
  (let ((i 0))
    (loop
      (when (= i *number-of-PEs*)
        (return-from initialise 'END-OF-THE-FIRST-STAGE))
      (setf (nth i FB) nil)
      (clrhash (nth i RB))
      (setf (nth i AA) nil)
```

```
    (delete-file (nth i files-list))
    (with-open-file
      (str (nth i files-list) :direction :output
        :if-does-not-exist :create))
    (setf (nth i FB) (add-fact-from-file
      "f:\\simulati\\testing\\fact.lsp" (nth i FB)))
    (add-rule-from-file (nth i DR) (nth i RB))
    (let ((t0 (get-internal-real-time)))
      (inference-engine i)
      (reverse (push (- (get-internal-real-time) t0)
              time-list)))
    (incf i))))
```

INTEGRATION OF PARTIAL RESULTS

```
(defun integrate-partial-result (list-of-partial-results)
  (delete-duplicates (my-concatenate list-of-partial-results)
    :test #'equal))
```

PRODUCEING PARTIAL RESULTS

```
(defun get-partial-results-from-clusters (list-of-clusters)
  (let ((result nil))
    (dolist (element list-of-clusters result)
      (push (comm-comp element)
        result))
    (reverse result)))

(defun comm-comp (a-cluster-list)
  (loop
    (let ((copy-secondary
          (copy-list (communicate a-cluster-list))))
      (dolist (i a-cluster-list)
        (inference-engine i))
      (when (equal (communicate a-cluster-list)
```

```
            copy-secondary) (return copy-secondary)))))

(defun communicate (a-cluster-list)
  (let ((secondary-list nil))
    (dolist (element a-cluster-list secondary-list)
      (setf secondary-list
          (add-fact-to-secondary
            (nth element files-list) secondary-list)))
    (dolist (element a-cluster-list secondary-list)
      (dolist (fact-element secondary-list)
        (add-object fact-element (nth element FB))))))
```

INFERENCE-ENGINE

```
(defun inference-engine (i)
  (loop (forward-chain i)
    (if (endp (forward-chain i))
      (return-from inference-engine 'end-of-the-cycle))))

(defun forward-chain (i)
  (let* ((rule-table (nth i RB))
        (fact-list (nth i FB))
        (B (use-rules rule-table fact-list)))
    (dolist (element B result)
        (setf result (fire-rule element rule-table i)))))
(defun fire-rule (a-list rule-table i)
  (let ((result (instantiate-variables
              (rule-rhs
                (gethash (first a-list)  rule-table))
              (rest (remove-if #' numberp a-list)))))
    (print-to-file result (nth i files-list))
    (print i)
    (print result)
    (add-object result (nth i FB))))
```

FORWARD CHAIN

```
(defun use-rules (rule-table fact-list)
  (setf current-agenda nil)
  (maphash #'(lambda (key value)
             (let ((binding-list
                    (apply-filters
                     (rule-lhs value) '(nil) fact-list)))
               (cond ((endp binding-list) current-agenda)
                     ((= 1 (length binding-list))
                      (push (apply #' cons key binding-list)
                        current-agenda))
                     (t (dolist (element binding-list)
                          (push (cons key element)
                            current-agenda))))))
    rule-table)
  (values current-agenda))

(defun apply-filters (patterns initial-input-list fact-list)
  (if (endp patterns) initial-input-list
    (apply-filters (rest patterns)
     (filter-binding (first patterns)
      initial-input-list fact-list) fact-list)))

(defun filter-binding (pattern a-list fact-list)
  (my-concatenate (mapcar #'(lambda (bindings)
                              (match-pattern-to-facts
                               pattern bindings fact-list))
                   a-list)))

(defun match-pattern-to-facts (pattern bindings fact-list)
  (delete-if #' endp (mapcar #'(lambda (fact)
                                 (try-fact
                                  pattern fact bindings))
                      fact-list)))
```

```
(defun try-fact (pattern fact bindings)
  (let ((result  (match pattern (rest fact) bindings)))
    (if (eq 'fail result) '() (cons (first fact) result))))

(defun my-concatenate (lists)
  (if (endp lists) 'nil
    (if (endp (first lists))
      (my-concatenate (rest lists))
      (cons (first (first lists))
        (my-concatenate
         (cons (rest (first lists)) (rest lists)))))))

(defun instantiate-variables (pattern a-list)
  (cond ((atom pattern) pattern)
        ((eq '? (first pattern))
         (extract-value (find-binding pattern a-list)))
        (t (cons (instantiate-variables (first pattern) a-list)
             (instantiate-variables (rest pattern) a-list)))))
```

PATTERN MATCHING

```
(defun match (pattern1 pattern2 &optional  bindings)
  (cond ((and (atom pattern1) (atom pattern2))
         (match-atoms pattern1 pattern2 bindings))
        ((and (listp pattern1) (eq '? (first pattern1)))
         (match-variable pattern1 pattern2 bindings))
        ((and (listp pattern1) (listp pattern2))
         (match-pieces pattern1 pattern2 bindings))
        (t 'fail)))

(defun match-atoms (p q bindings)
  (if (eql p q) bindings 'fail))
(defun match-variable (p q bindings)
  (let ((binding (find-binding p bindings)))
    (if binding (match (extract-value binding) q bindings)
```

```
        (add-bindings p q bindings))))

(defun match-pieces (p q bindings)
  (let ((result (match (first p) (first q) bindings)))
    (if (eq 'fail result) 'fail
      (match (rest p) (rest q) result))))

(defun find-binding (pattern-variable-expression binding)
  (unless (eq '- (extract-variable pattern-variable-expression))
    (assoc (extract-variable pattern-variable-expression) binding)))

(defun make-binding (variable datum)
  (list variable datum))

(defun extract-variable (pattern-variable-expression)
  (second pattern-variable-expression))

(defun add-bindings (pattern-variable-expression datum bindings)
  (if (eq '- (extract-variable pattern-variable-expression))
    bindings
    (cons (make-binding
           (extract-variable pattern-variable-expression)
           datum)
      bindings)))

(defun extract-value (binding)
  (second binding))
```

CONFLICT-RESOLUTION

```
(defun conflict-resolution (agenda-list current-agenda)
  (random-selection
   (specificity
    (recency
     (refraction agenda-list current-agenda)))))
```

REFRACTION

```
(defun refraction (main-list current-list)
  (if (endp main-list)  current-list
    (refraction (rest main-list)
     (compare (first main-list) current-list))))

(defun compare (list c-list)
  (delete-if #' endp
    (mapcar #' (lambda (a-list)
              (if (equal list a-list) nil a-list)) c-list)))
```

RECENCY

```
(defun recency (all-lists)
  (cond ((null (cdr all-lists)) all-lists)
       (t (remove-if #' endp
           (mapcar #'(lambda (a-list)
                    (membership (find-max all-lists)
                     a-list)) all-lists)))))

(defun membership (arg list)
   (when (member arg list) list))

(defun collect-tags (a-list)
  (mapcar #' (lambda (element)
             (remove-if-not #' numberp element)) a-list))

(defun find-max (a-list)
  (apply #'max (my-concatenate (collect-tags a-list))))
```

SPECIFICITY

```
(defun specificity (a-list)
  (cond ((null (cdr a-list)) a-list)
        (t (remove-if #'endp
             (mapcar #' (lambda (element)
                          (when (= (count-conditions element)
                                   (max-conditions a-list))
                            element ))a-list)))))

(defun max-conditions (a-list)
  (apply #' max
    (mapcar #' (lambda (element)
                 (length
                   (remove-if-not #' listp element)))
      a-list)))
(defun count-conditions (element)
  (length (remove-if-not #' listp element)))
```

RANDOM-SELECTION

```
(defun random-selection (a-list)
  (cond ((null (cdr a-list)) a-list)
        (t (list (nth (random (length a-list)) a-list)))))
```

ADDING FACTS TO FACT-BASE FROM A FILE

```
(defun add-fact-from-file (file fact-list)
  (if (probe-file file)
    (with-open-file (str file :direction :input)
      (read-add-loop str fact-list))))

(defun read-add-loop (stream fact-list &optional (eof (gensym)))
```

```
  (flet ((eof-p (obj)
           (eq obj eof)))
    (loop
      (let ((exp (read stream () eof ())))
        (if (eof-p exp) (return fact-list))
        (add-object exp fact-list)))))
```

ADDING RULES TO RULE-BASE FROM A FILE

```
(defun add-rule-from-file (file rule-table)
  (if (probe-file file)
    (with-open-file (str file :direction :input)
      (read-add-hash str rule-table))))

(defun read-add-hash (stream rule-table &optional (eof (gensym)))
  (flet ((eof-p (obj)
           (eq obj eof)))
    (loop
      (let ((exp (read stream () eof ())))
        (if (eof-p exp) (return t))
        (adding-rule exp rule-table)))))

(defun adding-rule (exp rule-table)
  (setf (gethash (first exp) rule-table)
      (make-rule :lhs (second exp) :rhs (third exp))))
```

ADDING FACTS TO SECONDARY STORAGE FROM FILE-RESULTS

```
(defun add-fact-to-secondary (file secondary)
  (if (probe-file file)
    (with-open-file (str file :direction :input)
      (read-add2-loop str secondary))))

(defun read-add2-loop (stream secondary &optional (eof (gensym)))
  (flet ((eof-p (obj)
```

```
       (eq obj eof)))
    (loop
      (let ((exp (read stream () eof ()))
        (if (eof-p exp) (return (reverse secondary)))
        (pushnew exp secondary :test #'equal)))))
```

PRINTING TO A FILE

```
(defun print-to-file (exp file)
  (with-open-file
    (str file :direction :output :if-exists :append
      :if-does-not-exist :create)
    (print exp str)))
```

THE PARALLEL GENETIC ALGORITHM

```
(defun PGA ()
  (let* ((temp nil)
         (i 0)
         (j 0)
         (temp2 nil))
    (loop
      (let* ((next-generation
               (IMCR (fitness-of-initial-population)))
             (fittest-so-far
               (fittest-individual
                 (associate-fitness-to-individuals
                  next-generation))))
        (push (car fittest-so-far) temp))
      (incf i)
      (when (= i *number-of-PEs*) (return (print temp))))
    (let ((fit1 (car (fittest-individual temp))))
      (cond ((< (car fit1) (objective-function *KB*)) fit1)
            (t (loop
                 (push (apply-parallel-hill-climbing j fit1)
```

```
        temp2)
       (incf j)
       (when (= j *number-of-PEs*)
        (return
         (car (fittest-individual temp2))))
       ))))))

(defun IMCR (pop)
  (apply-inversion (apply-mutation (apply-xover
                    (reproduction pop)))))
```

RANDOM GENERATION OF INITIAL POPULATION

```
(defun create-initial-population ()
  (let ((initial-population nil))
    (do ((counter 1 (+ 1 counter)))
       ((> counter *population-size*) initial-population)
      (if (not (member (generate-chromosom)
              initial-population))
        (push (generate-chromosom) initial-population)
        (setf counter (- 1 counter))))))
(defun generate-chromosom ()
  (let ((chromosom-string nil))
    (dotimes (counter *number-of-PEs* chromosom-string)
      (push (random *number-of-clusters*) chromosom-string))))
```

ASSOCIATION OF FITNESS-VALUES TO INDIVIDUALS

```
(defun fitness-of-initial-population ()
  (associate-fitness-to-individuals (create-initial-population)))

(defun associate-fitness-to-individuals (a-population)
  (mapcar #' (lambda (element)
          (cons (objective-function element)
              element)) a-population))
```

```
(defun objective-function (individual)
  (let ((calculation-factor 0.5)
        (communication-factor 0.5))
    (+ (* calculation-factor
          (calculation-time individual))
       (* communication-factor
          (communication-time individual)))))
```

CALCULATION OF COMPUTATION-TIME OF INDIVIDUALS

```
(defun calculation-time (individual)
  (let ((result 0))
    (dotimes (i *number-of-clusters* result)
      (setf result
            (+ (square (calculation-time-cluster
                        (get-cluster i individual)))
               result)))))

(defun get-cluster (number individual)
  (let ((cluster-list nil)
        (counter 0))
    (dolist (element individual cluster-list)
      (when (= element number)(push counter cluster-list))
      (setf counter (1+ counter)))
    (reverse cluster-list)))

(defun calculation-time-cluster (a-list)
  (let ((result 0))
    (dolist (element a-list result)
      (setf result
            (+ result
               (file-length (nth element DR))
               (file-length (nth element files-list)))))))
```

CALCULATION OF COMMUNICATION-TIME OF INDIVIDUALS

```
(defun communication-time (individual)
  (let ((result 0))
    (dotimes (i *number-of-clusters* result)
      (setf result (+ (communication-time-cluster
                      (get-cluster i individual))
                     result)))))

(defun communication-time-cluster (a-list)
  (let ((result 0))
    (dolist (i a-list result)
      (dolist (j a-list)
        (unless (>= i j)
         (setf result (+ result (communication-pes i j))))))))

(defun communication-pes (PEi PEj)
  (let ((t-comm 0.1))
    (* t-comm
      (information-amount PEi PEj)
      (distance PEi PEj))))

(defun information-amount (PEi PEj)
  (+ (file-length (nth PEi files-list))
    (file-length (nth PEj files-list))))

(defun distance (PEi PEj)
  (let ((xi (first (nth PEi pe-coordinates)))
       (yi (second (nth PEi pe-coordinates)))
       (xj (first (nth PEj pe-coordinates)))
       (yj (second (nth PEj pe-coordinates))))
  (+ (abs (- xi xj)) (abs (- yi yj)))))
```

FITTEST INDIVIDUALS

```
(defun fittest-individual (value-population)
  (let ((fittest-value
```

```
      (apply #'min (fitness-list value-population)))
    (fittest-individual nil))
  (dolist (element value-population fittest-individual)
    (when (= fittest-value (car element))
      (push  element fittest-individual)))
  (if (= 1 (length fittest-individual)) fittest-individual
   (cons (car fittest-individual) ()))))
```

FLIP-FUNCTION

```
(defun flip (probablity)
  (> probablity (random 1.0)))
```

RANKING REPRODUCTION SCHEME

```
(defun reproduction (value-population)
  (let ((alist
         (sort-by-reproduction-probablities value-population)))
    (loop
      (when (= (length *mates*) *population-size*)
        (return *mates*))
      (push (cdar  alist) *mates*)
      (setf alist (rest alist)))))

(defun sort-by-reproduction-probablities (population)
   (apply-sorting (reproduction-probablities population)))

(defun apply-sorting (list-of-lists)
  (let ((firsts
         (delete-duplicates
           (sorted-list-of-first-elements list-of-lists))))
    (my-concatenate
      (mapcar #'(lambda (element)
               (find-all element list-of-lists))
        firsts))))
```

```
(defun sorted-list-of-first-elements (a-list)
  (sort (copy-list (fitness-list a-list)) #'>))

(defun fitness-list (population)
  (mapcar #'(lambda (element)(car element)) population))

(defun find-all (no list-of-lists)
  (let ((result nil))
    (dolist (element list-of-lists result)
      (when (equal (car element) no)
        (push element result)))))

(defun reproduction-probablities (population)
  (setf *mates* nil)
  (mapcar #' (lambda (element)
           (let(( result (linear-rank
                       (fitness-list population)
                       (car element))))
             (cond ((<= result 1)
                  (cons result (rest element)))
                  (t (push (cdr element) *mates*)
                    (cons (- result 1) (rest element))))))
    population))

(defun linear-rank (list-of-fitness number)
  (let ((min 0.8) (max 1.2))
    (+ min (* (- max min)
          (/ (- (find-index list-of-fitness number) 1)
            (- *population-size* 1))))))

(defun find-index (list-of-numbers number)
  (let ((rank 0)
      (list-of-fitness
        (remove-duplicates
          (sort
            (copy-list list-of-numbers) #'<))))
```

```
    (if (not (member number list-of-fitness)) 0
      (dolist (element list-of-fitness rank)
        (when (<= element number)
          (setf rank (1+ rank))))))
```

TWO-POINTS CROSS OVER

```
(defun apply-xover (a-list)
  (let ((next-population nil)
        (mates (copy-list a-list)))
    (loop
      (cond ((= (length mates) 1)
             (push (first mates) next-population)
             (return next-population))
            ((= (length mates) 0) (return next-population))
            (t (let* ((parent-1
                        (nth (random (length mates)) mates))
                      (mate-1
                        (delete parent-1 mates :test #'equal))
                      (parent-2
                        (nth (random (length mate-1)) mate-1))
                      (mate-2
                        (delete parent-2 mate-1 :test #'equal))
                      (result (children parent-1 parent-2)))
                 (push (first result) next-population)
                 (push (second result) next-population)
                 (setf mates mate-2)))))))

(defun children (parent-1 parent-2)
  (cond ((flip Pc)
         (let  ((child-1 (copy-list parent-1))
                (child-2 (copy-list parent-2))
                (n (random (length parent-1)))
                (m (random (length parent-2))))
           (list
             (replace child-1 parent-2 :start1 (min n m)
```

```
            :end1 (+ 1 (max n m)) :start2 (min n m))
          (replace child-2 parent-1 :start1 (min n m)
            :end1 (+ 1 (max n m)):start2 (min n m)))))
      (t (list parent-1 parent-2))))
---------------------------------------------------------------------------------------------------------
MUTATION
---------------------------------------------------------------------------------------------------------

(defun apply-mutation (next-population)
  (mapcar #'(lambda (element)
             (mutation element)) next-population))

(defun  mutation (individual)
  (if (zerop Pm) individual
    (do ((i 0 (+ 1 i)))
       ((= i *number-of-PEs*) individual)
      (setf (nth i individual)
          (gene-mutation (nth i individual))))))

(defun gene-mutation (gene)
  (if (flip Pm) (setf gene (random *number-of-clusters*)) gene))
---------------------------------------------------------------------------------------------------------
INVERSION
---------------------------------------------------------------------------------------------------------

(defun apply-inversion (population)
  (mapcar #'(lambda (element)
             (inversion element)) population))
(defun inversion (individual)
  (if (flip pinversion) (invert individual) individual))

(defun invert (individual)
  (let* ((n (random *number-of-PEs*))
       (m (random *number-of-PEs*))
       (result-1 nil)
       (result-2 nil)
       (result-3 nil)
```

```
        (i 0)
        (j (min n m))
        (k (max n m)))
    (concatenate-lists
      (loop
        (if (= i (min n m)) (return (reverse result-1)))
        (setf result-1 (cons (nth i individual) result-1))
        (incf i))
      (loop
        (if (= j (max n m)) (return  result-2))
        (setf result-2 (cons (nth j individual) result-2))
        (incf j))
      (loop
        (if (= k (length individual))
          (return (reverse result-3)))
        (setf result-3 (cons (nth k individual) result-3))
        (incf k)))))
```

PARALLEL HILL CLIMBING

```
(defun apply-parallel-hill-climbing (i ind)
  (parallel-hill-climbing i ind *KB*))

(defun parallel-hill-climbing (i current-state goal-state)
  (let ((n (nth i goal-state)))
    (cond ((< (count n goal-state) 2) current-state)
          (t (change-state
               i (get-cluster n goal-state) current-state)))))

(defun change-state (i n-list ind)
  (let* ((ind2 (copy-list (rest ind)))
         (a (nth i ind2))
         (new-list (remove i n-list)))
    (loop
      (cond ((endp new-list) (return ind2))
            ((= 1 (length new-list)) (setf b (car new-list)))
```

```
        (t (setf b (cadr new-list))))
      (if (= a (nth b ind2)) nil (setf (nth b ind2) a))
      (setf new-list (cddr new-list)))
    (let ((of2 (objective-function ind2)))
      (if (< of2 (first ind)) (cons of2 ind2) ind))))
```

APPROPRIATE PE-CLUSTERS

```
(defun find-clusters ()
  (let ((best-individual (cdr (PGA)))
        (result-list nil))
    (dotimes (i *number-of-clusters* result-list)
      (push (get-cluster i best-individual)
        result-list))
    (reverse result-list)))
```

A.3 Serial genetic algorithm

```
(defun find-clusters ()
  (let ((best-individuals (SERI-GA))
        (result-list nil))
    (if (= 1 (length best-individuals))
      (setf fittest-individual (cdar best-individuals))
      (setf fittest-individual
          (cdar (nth (random (length best-individuals))
                best-individuals))))
    (dotimes (i *number-of-clusters* result-list)
      (push (get-cluster i fittest-individual)
        result-list))
    (reverse result-list)))

(defun SERI-GA ()
  (setf generation-0 (fitness-of-initial-population))
  (format t
    "~2%The initial-generation with associated values is:~2%~a."
    generation-0)
```

```
(let ((next-generation (apply-genetic-operators generation-0))
      (i 1))
   (loop
     (let* ((value-population
              (associate-fitness-to-individuals
               next-generation))
            (fittest-so-far
              (fittest-individuals value-population)))
       (format t
         "~2%The fittest individual(s) in generation ~a is:~%~a."
         i fittest-so-far)
       (incf i)
       (when (> i *maximum-generations*)
         (return fittest-so-far))
       (setf next-generation
           (apply-genetic-operators value-population))))))

(defun apply-genetic-operators (population)
  (apply-hill-climbing
     (apply-inversion
       (apply-mutation
         (apply-xover
           (reproduction population))))))
```

HILL-CLIMBING

```
(defun apply-hill-climbing (ind)
  (hill-climbing *KB* (rest ind)))

(defun hill-climbing (goal-state current-state)
  (cond ((< (objective-function current-state)
            (objective-function goal-state)) current-state)
        ((endp goal-state) current-state)
        (t (hill-climbing (remove (first goal-state) goal-state)
             (new-state goal-state current-state)))))
```

```
(defun new-state (s-space ind)
  (cond ((< (count (first s-space) *KB*) 2) ind)
        (t (let ((neighbours
                   (get-cluster (first s-space) *KB*)))
             (change-elements neighbours ind)))))

(defun change-elements (n-list ind)
  (let ((ind2 (copy-list ind))
        (a (nth (first n-list) ind)))
    (loop
      (cond ((endp n-list) (return ind))
            ((= 1 (length n-list)) (setf b (car n-list)))
            (t (setf b (cadr n-list))))
      (if (= a (nth b ind2)) nil
        (setf (nth b ind2) a))
      (setf n-list (cddr n-list)))
    (if (< (objective-function ind2)
           (objective-function ind))
      (setf ind ind2) ind)))
```

Appendix B: The KB

In this appendix, the developed knowledge base for the simulation is presented. The information in this knowledge base is in abstract form, in an effort to capture the complexities of a real-world knowledge base.

(R1 (((? x) p1)) ((? x) P2))
(R2 (((? x) p3)) ((? x) P2))
(R3 (((? x) p4)) ((? x) P5))
(R4 (((? x) p6) ((? x) p7)) ((? x) P5))
(R5 (((? x) p8)) ((? x) P9))
(R6 (((? x) p10) ((? x) p11) ((? x) p12)) ((? x) P9))
(R7 (((? x) p2) ((? x) p13)) ((? x) P14))
(R8 (((? x) p2) ((? x) p15)) ((? x) P14))
(R9 (((? x) p2) ((? x) p9) ((? x) p16) ((? x) p17)) ((? x) P18))
(R10 (((? x) p2) ((? x) p9) ((? x) p16) ((? x) p19)) ((? x) P18))
(R11 (((? x) p14) ((? x) p21) ((? x) p22) ((? x) p17)) ((? x) P23))
(R12 (((? x) p14) ((? x) p19)) ((? x) P24))
(R13 (((? x) p5) ((? x) p25) ((? x) p21) ((? x) p22) ((? x) p26))((? x) P27))
(R14 (((? x) p2) ((? x) p25) ((? x) p28) ((? x) p26)) ((? x) P29))
(R15 (((? x) p2) ((? x) p30)) ((? x) P31))
(R16 (((? x) p31) ((? y) p32)) ((? x) p17))
(R17 (((? x) p2)) ((? x) s85))(R18 (((? x) p14) ((? x) p38)) ((? x) p34))
(R19 (((? x) p30) ((? x) p25) ((? x) q66)) ((? x) q77))

(R20 (((? x) p31) ((? y) q32)) ((? x) p17))

(R21 (((? x) p33)) ((? x) P34))

(R22 (((? x) p35)) ((? x) P34))

(R23 (((? x) p36)) ((? x) P37))

(R24 (((? x) p38) ((? x) p39)) ((? x) P37))

(R25 (((? x) p40)) ((? x) P41))

(R26 (((? x) p42) ((? x) p43) ((? x) p44)) ((? x) P41))

(R27 (((? x) p34) ((? x) p45)) ((? x) P46))

(R28 (((? x) p34) ((? x) p47)) ((? x) P46))

(R29 (((? x) p34) ((? x) p41) ((? x) p48) ((? x) p49)) ((? x) P50))

(R30 (((? x) p34) ((? x) p41) ((? x) p48) ((? x) p51)) ((? x) P52))

(R31 (((? x) p46) ((? x) p53) ((? x) p54) ((? x) p49)) ((? x) P55))

(R32 (((? x) p46) ((? x) p51)) ((? x) P56))

(R33 (((? x) p37) ((? x) p57) ((? x) p53) ((? x) p54) ((? x) p58)) ((? x) P59))

(R34 (((? x) p34) ((? x) p57) ((? x) p60) ((? x) p58)) ((? x) P61))

(R35 (((? x) p34) ((? x) p62)) ((? x) P63))

(R36 (((? x) p63) ((? y) p64)) ((? x) p40))

(R37 (((? x) p6) ((? y) p10) ((? y) p14)) ((? y) P57))

(R38 (((? x) p35)) ((? x) P12))

(R39 (((? x) p1)) ((? x) P34))

(R40 (((? x) p2) ((? x) p63)) ((? x) P60))

(R41 (((? x) p65)) ((? x) P66))

(R42 (((? x) p67)) ((? x) P66))

(R43 (((? x) p68)) ((? x) P69))

(R44 (((? x) p70) ((? x) p71)) ((? x) P69))

(R45 (((? x) p72)) ((? x) P73))

(R46 (((? x) p74) ((? x) p75) ((? x) p76)) ((? x) P73))

(R47 (((? x) p66) ((? x) p77)) ((? x) P78))

(R48 (((? x) p66) ((? x) p79)) ((? x) P78))

(R49 (((? x) p66) ((? x) p73) ((? x) p80) ((? x) p81)) ((? x) P82))

(R50 (((? x) p66) ((? x) p73) ((? x) p80) ((? x) p83)) ((? x) P84))

(R51 (((? x) p78) ((? x) p85) ((? x) p86) ((? x) p81)) ((? x) P87))

(R52 (((? x) p78) ((? x) p83)) ((? x) P88))

(R53 (((? x) p69) ((? x) p89) ((? x) p85) ((? x) p86) ((? x) p90)) ((? x) P91))

(R54 (((? x) p66) ((? x) p89) ((? x) p92) ((? x) p90)) ((? x) P93))

(R55 (((? x) p66) ((? x) p94)) ((? x) P95))

(R56 (((? x) p95) ((? y) p96)) ((? x) p81))

(R57 (((? x) p65)) ((? x) q38))

(R58 (((? x) p66) ((? x) p79)) ((? x) q40))

(R59 (((? x) p97)) ((? x) P98))

(R60 (((? x) p99)) ((? x) P98))

(R61 (((? x) p100)) ((? x) P101))

(R62 (((? x) p102) ((? x) p103)) ((? x) P101))

(R63 (((? x) p104)) ((? x) P105))

(R64 (((? x) p106) ((? x) p107) ((? x) p108)) ((? x) P105))

(R65 (((? x) p98) ((? x) p111)) ((? x) P110))

(R66 (((? x) p98) ((? x) p111)) ((? x) P110))

(R67 (((? x) p98) ((? x) p105) ((? x) p112) ((? x) p113)) ((? x) P114))

(R68 (((? x) p98) ((? x) p105) ((? x) p112) ((? x) p115)) ((? x) P116))

(R69 (((? x) p110) ((? x) p117) ((? x) p118) ((? x) p113)) ((? x) P119))

(R70 (((? x) p110) ((? x) p115)) ((? x) P120))

(R71 (((? x) p101) ((? x) p121) ((? x) p117) ((? x) p118) ((? x) p122)) ((? x) P123))

(R72 (((? x) p98) ((? x) p121) ((? x) p124) ((? x) p122)) ((? x) P125))

(R73 (((? x) p98) ((? x) p126)) ((? x) P127))

(R74 (((? x) p127) ((? y) p128)) ((? x) p113))

(R75 (((? x) p97)) ((? x) q97))

(R76 (((? x) p109)) ((? x) q150))

(R77 (((? x) q81) ((? x) q129)) ((? x) P128))

(R78 (((? x) p129)) ((? x) P130))

(R79 (((? x) p131)) ((? x) P130))

(R80 (((? x) p132)) ((? x) P133))

(R81 (((? x) p134) ((? x) p135)) ((? x) P133))

(R82 (((? x) p136)) ((? x) P137))

(R83 (((? x) p138) ((? x) p139) ((? x) p140)) ((? x) P137))

(R84 (((? x) p130) ((? x) p141)) ((? x) P142))

(R85 (((? x) p130) ((? x) p143)) ((? x) P142))

(R86 (((? x) p130) ((? x) p137) ((? x) p144) ((? x) p145)) ((? x) P146))

(R87 (((? x) p130) ((? x) p137) ((? x) p144) ((? x) p147)) ((? x) P148))

(R88 (((? x) p142) ((? x) p149) ((? x) p150) ((? x) p145)) ((? x) P151))

(R89 (((? x) p142) ((? x) p147)) ((? x) P152))

(R90 (((? x) p133) ((? x) p153) ((? x) p149) ((? x) p150) ((? x) p154)) ((? x) P155))

(R91 (((? x) p130) ((? x) p153) ((? x) p156) ((? x) p154)) ((? x) P157))

(R92 (((? x) p130) ((? x) p158)) ((? x) P159))

(R93 (((? x) p159) ((? y) p160)) ((? x) p145))

(R94 (((? x) p133) ((? x) p157)) ((? x) P121))

(R95 (((? x) s98) ((? x) p141)) ((? x) s120))

(R96 (((? x) q1)) ((? x) q2))

(R97 (((? x) q3)) ((? x) q2))

(R98 (((? x) q4)) ((? x) q5))

(R99 (((? x) q6) ((? x) q7)) ((? x) q5))

(R100 (((? x) q8)) ((? x) q9))

(R101 (((? x) q10) ((? x) q11) ((? x) q12)) ((? x) q9))

(R102 (((? x) q2) ((? x) q13)) ((? x) q14))

(R103 (((? x) q2) ((? x) q15)) ((? x) q14))

(R104 (((? x) q2) ((? x) q9) ((? x) q16) ((? x) q17)) ((? x) q18))

(R105 (((? x) q2) ((? x) q9) ((? x) q16) ((? x) q19)) ((? x) q18))

(R106 (((? x) q14) ((? x) q21) ((? x) q22) ((? x) q17)) ((? x) q23))

(R107 (((? x) q14) ((? x) q19)) ((? x) q24))

(R108 (((? x) q5) ((? x) q25) ((? x) q21) ((? x) q22) ((? x) q26)) ((? x) q27))

(R109 (((? x) q2) ((? x) q25) ((? x) q28) ((? x) q26)) ((? x) q29))

(R110 (((? x) q2) ((? x) q30)) ((? x) q31))

(R111 (((? x) q31) ((? y) q32)) ((? x) q17))

(R112 (((? x) q24) ((? x) q12)) ((? x) p31))

(R113 (((? x) q33)) ((? x) q34))

(R114 (((? x) q35)) ((? x) q34))

(R115 (((? x) q36)) ((? x) q37))

(R116 (((? x) q38) ((? x) q39)) ((? x) q37))

(R117 (((? x) q40)) ((? x) q41))

(R118 (((? x) q42) ((? x) q43) ((? x) q44)) ((? x) q41))

(R119 (((? x) q34) ((? x) q45)) ((? x) q46))

(R120 (((? x) q34) ((? x) q47)) ((? x) q46))

(R121 (((? x) q34) ((? x) q41) ((? x) q48) ((? x) q49)) ((? x) q50))

(R122 (((? x) q34) ((? x) q41) ((? x) q48) ((? x) q51)) ((? x) q52))

(R123 (((? x) q46) ((? x) q53) ((? x) q54) ((? x) q49)) ((? x) q55))

(R124 (((? x) q46) ((? x) q51)) ((? x) q56))

(R125 (((? x) q37) ((? x) q57) ((? x) q53) ((? x) q54) ((? x) q58)) ((? x) q59))

(R126 (((? x) q34) ((? x) q57) ((? x) q60) ((? x) q58)) ((? x) q61))

(R127 (((? x) q34) ((? x) q62)) ((? x) q63))

(R128 (((? x) q63) ((? y) q64)) ((? x) q40))

(R129 (((? x) q52) ((? x) q40)) ((? x) p82))

(R130 (((? x) q47)) ((? x) q78))

(R131 (((? x) q57) ((? y) q68)) ((? x) q62))

(R132 (((? x) q58)) ((? x) p65))

(R133 (((? x) q65)) ((? x) q66))

(R134 (((? x) q67)) ((? x) q66))

(R135 (((? x) q68)) ((? x) q69))

(R136 (((? x) q70) ((? x) q71)) ((? x) q69))

(R137 (((? x) q72)) ((? x) q73))

(R138 (((? x) q74) ((? x) q75) ((? x) q76)) ((? x) q73))

(R139 (((? x) q66) ((? x) q77)) ((? x) q78))

(R140 (((? x) q66) ((? x) q79)) ((? x) q78))

(R141 (((? x) q66) ((? x) q73) ((? x) q80) ((? x) q81)) ((? x) q82))

(R142 (((? x) q66) ((? x) q73) ((? x) q80) ((? x) q83)) ((? x) q84))

(R143 (((? x) q78) ((? x) q85) ((? x) q86) ((? x) q81)) ((? x) q87))

(R144 (((? x) q78) ((? x) q83)) ((? x) q88))

(R145 (((? x) q69) ((? x) q89) ((? x) q85) ((? x) q86) ((? x) q90)) ((? x) q91))

(R146 (((? x) q66) ((? x) q89) ((? x) q92) ((? x) q90)) ((? x) q93))

(R147 (((? x) q66) ((? x) q94)) ((? x) q95))

(R148 (((? x) q95) ((? y) q96)) ((? x) q81))

(R149 (((? x) q85)) ((? x) q109))

(R150 (((? x) q71)) ((? x) s46))

(R151 (((? x) q66)) ((? x) s94))

(R152 (((? x) q97)) ((? x) q98))

(R153 (((? x) q99)) ((? x) q98))

(R154 (((? x) q100)) ((? x) q101))

(R155 (((? x) q102) ((? x) q103)) ((? x) q101))

(R156 (((? x) q104)) ((? x) q105))

(R157 (((? x) q106) ((? x) q107) ((? x) q108)) ((? x) q105))

(R158 (((? x) q98) ((? x) q111)) ((? x) q110))

(R159 (((? x) q98) ((? x) q111)) ((? x) q110))

(R160 (((? x) q98) ((? x) q105) ((? x) q112) ((? x) q113)) ((? x) q114))

(R161 (((? x) q98) ((? x) q105) ((? x) q112) ((? x) q115)) ((? x) q116))

(R162 (((? x) q110) ((? x) q117) ((? x) q118) ((? x) q113)) ((? x) q119))

(R163 (((? x) q110) ((? x) q115)) ((? x) q120))

(R164 (((? x) q101) ((? x) q121) ((? x) q117) ((? x) q118) ((? x) q122)) ((? x) q123))

(R165 (((? x) q98) ((? x) q121) ((? x) q124) ((? x) q122)) ((? x) q125))

(R166 (((? x) q98) ((? x) q126)) ((? x) q127))

(R167 (((? x) q127) ((? y) q128)) ((? x) q113))

(R168 (((? x) q106) ((? x) q97)) ((? x) q80))

(R169 (((? x) q122)) ((? x) p122))

(R170 (((? x) p121) ((? x) p104)) ((? x) q113))

(R171 (((? x) q115) ((? x) q116) ((? x) q103)) ((? x) q72))

(R172 (((? x) q129)) ((? x) q130))

(R173 (((? x) q131)) ((? x) q130))

(R174 (((? x) q132)) ((? x) q133))

(R175 (((? x) q134) ((? x) q135)) ((? x) q133))

(R176 (((? x) q136)) ((? x) q137))

(R177 (((? x) q138) ((? x) q139) ((? x) q140)) ((? x) q137))

(R178 (((? x) q130) ((? x) q141)) ((? x) q142))

(R179 (((? x) q130) ((? x) q143)) ((? x) q142))

(R180 (((? x) q130) ((? x) q137) ((? x) q144) ((? x) q145)) ((? x) q146))

(R181 (((? x) q130) ((? x) q137) ((? x) q144) ((? x) q147)) ((? x) q148))

(R182 (((? x) q142) ((? x) q149) ((? x) q150) ((? x) q145)) ((? x) q151))

(R183 (((? x) q142) ((? x) q147)) ((? x) q152))

(R184 (((? x) q133) ((? x) q153) ((? x) q149) ((? x) q150) ((? x) q154)) ((? x) q155))

(R185 (((? x) q130) ((? x) q153) ((? x) q156) ((? x) q154)) ((? x) q157))

(R186 (((? x) q130) ((? x) q158)) ((? x) q159))

(R187 (((? x) q159) ((? y) q160)) ((? x) q145))

(R188 (((? x) q134) ((? x) q149)) ((? x) s124))

(R189 (((? x) q100) ((? x) q115)) ((? x) s125))

(R190 (((? x) q130) ((? x) q141)) ((? x) s150))

(R191 (((? x) s129)) ((? x) q130))

(R192 (((? x) s1)) ((? x) s2))

(R193 (((? x) s3)) ((? x) s2))

(R194 (((? x) s4)) ((? x) s5))

(R195 (((? x) s6) ((? x) s7)) ((? x) s5))

(R196 (((? x) s8)) ((? x) s9))

(R197 (((? x) s10) ((? x) s11) ((? x) s12)) ((? x) s9))

(R198 (((? x) s2) ((? x) s13)) ((? x) s14))

(R199 (((? x) s2) ((? x) s15)) ((? x) s14))

(R200 (((? x) s2) ((? x) s9) ((? x) s16) ((? x) s17)) ((? x) s18))

(R201 (((? x) s2) ((? x) s9) ((? x) s16) ((? x) s19)) ((? x) s18))

(R202 (((? x) s14) ((? x) s21) ((? x) s22) ((? x) s17)) ((? x) s23))

(R203 (((? x) s14) ((? x) s19)) ((? x) s24))

(R204 (((? x) s5) ((? x) s25) ((? x) s21) ((? x) s22) ((? x) s26)) ((? x) s27))

(R205 (((? x) s2) ((? x) s25) ((? x) s28) ((? x) s26)) ((? x) s29))

(R206 (((? x) s2) ((? x) s30)) ((? x) s31))

(R207 (((? x) s31) ((? y) s32)) ((? x) s17))

(R208 (((? x) s9)) ((? x) s134))

(R209 (((? x) s159) ((? x) s31)) ((? x) s15))

(R210 (((? x) s153) ((? x) q123)) ((? x) p78))

(R211 (((? x) s33)) ((? x) s34))

(R212 (((? x) p60)) ((? x) s34))

(R213 (((? x) s36)) ((? x) s37))

(R214 (((? x) s38) ((? x) s39)) ((? x) s37))

(R215 (((? x) s40)) ((? x) s41))

(R216 (((? x) s42) ((? x) s43) ((? x) s44)) ((? x) s41))

(R217 (((? x) s34) ((? x) s45)) ((? x) s46))

(R218 (((? x) s34) ((? x) s47)) ((? x) s46))

(R219 (((? x) s34) ((? x) s41) ((? x) s48) ((? x) s49)) ((? x) s50))

(R220 (((? x) s34) ((? x) s41) ((? x) s48) ((? x) s51)) ((? x) s52))

(R221 (((? x) s46) ((? x) s53) ((? x) s54) ((? x) s49)) ((? x) s55))

(R222 (((? x) s46) ((? x) s51)) ((? x) s56))

(R223 (((? x) s37) ((? x) s57) ((? x) s53) ((? x) s54) ((? x) s58)) ((? x) s59))

(R224 (((? x) s34) ((? x) s57) ((? x) s60) ((? x) s58)) ((? x) s61))

(R225 (((? x) s34) ((? x) s62)) ((? x) s63))

(R226 (((? x) s63) ((? y) s64)) ((? x) s40))

(R227 (((? x) s34) ((? x) s41) ((? x) q94)) ((? x) q78))

(R228 (((? x) s37) ((? x) s40) ((? x) s63) ((? x) s78)) ((? x) s68))

(R229 (((? x) s65)) ((? x) s66))

(R230 (((? x) s67)) ((? x) s66))

(R231 (((? x) s68)) ((? x) s69))

(R232 (((? x) s70) ((? x) s71)) ((? x) s69))

(R233 (((? x) s72)) ((? x) s73))

(R234 (((? x) s74) ((? x) s75) ((? x) s76)) ((? x) s73))

(R235 (((? x) s66) ((? x) s77)) ((? x) s78))

(R236 (((? x) s66) ((? x) s79)) ((? x) s78))

(R237 (((? x) s66) ((? x) s73) ((? x) s80) ((? x) s81)) ((? x) s82))

(R238 (((? x) s66) ((? x) s73) ((? x) s80) ((? x) s83)) ((? x) s84))

(R239 (((? x) s78) ((? x) s85) ((? x) s86) ((? x) s81)) ((? x) s87))

(R240 (((? x) s78) ((? x) s83)) ((? x) s88))

(R241 (((? x) s69) ((? x) s89) ((? x) s85) ((? x) s86) ((? x) s90)) ((? x) s91))

(R242 (((? x) s66) ((? x) s89) ((? x) s92) ((? x) s90)) ((? x) s93))

(R243 (((? x) s66) ((? x) s94)) ((? x) s95))

(R244 (((? x) s95) ((? y) s96)) ((? x) s81))

(R245 (((? x) s70) ((? x) s66)) ((? x) q95))

(R246 (((? x) s95) ((? y) q72)) ((? x) q73))

(R247 (((? x) s97)) ((? x) s98))

(R248 (((? x) s99)) ((? x) s98))

(R249 (((? x) s100)) ((? x) s101))

(R250 (((? x) s102) ((? x) s103)) ((? x) s101))

(R251 (((? x) s104)) ((? x) s105))

(R252 (((? x) s106) ((? x) s107) ((? x) s108)) ((? x) s105))

(R253 (((? x) s98) ((? x) s111)) ((? x) s110))

(R254 (((? x) s98) ((? x) s111)) ((? x) s110))

(R255 (((? x) s98) ((? x) s105) ((? x) s112) ((? x) s113)) ((? x) s114))

(R256 (((? x) s98) ((? x) s105) ((? x) s112) ((? x) s115)) ((? x) s116))

(R257 (((? x) s110) ((? x) s117) ((? x) s118) ((? x) s113)) ((? x) s119))

(R258 (((? x) s110) ((? x) s115)) ((? x) s120))

(R259 (((? x) s101) ((? x) s121) ((? x) s117) ((? x) s118) ((? x) s122)) ((? x) s123))

(R260 (((? x) s98) ((? x) s121) ((? x) s124) ((? x) s122)) ((? x) s125))

(R261 (((? x) s98) ((? x) s126)) ((? x) s127))

(R262 (((? x) s127) ((? y) p128)) ((? x) s113))

(R263 (((? x) s127)) ((? x) p144))

(R264 (((? x) s98)) ((? x) q153))

(R265 (((? x) q130) ((? x) q131)) ((? x) s119))

(R266 (((? x) q129) ((? x) q137)) ((? x) s121))

(R267 (((? x) s129)) ((? x) s130))

(R268 (((? x) s131)) ((? x) s130))

(R269 (((? x) s132)) ((? x) s133))

(R270 (((? x) s134) ((? x) s135)) ((? x) s133))

(R271 (((? x) s136)) ((? x) s137))

(R272 (((? x) s138) ((? x) s139) ((? x) s140)) ((? x) s137))

(R273 (((? x) s130) ((? x) s141)) ((? x) s142))

(R274 (((? x) s130) ((? x) s143)) ((? x) s142))

(R275 (((? x) s130) ((? x) s137) ((? x) s144) ((? x) s145)) ((? x) s146))

(R276 (((? x) s130) ((? x) s137) ((? x) s144) ((? x) s147)) ((? x) s148))

(R277 (((? x) s142) ((? x) s149) ((? x) s150) ((? x) s145)) ((? x) s151))

(R278 (((? x) s142) ((? x) s147)) ((? x) s152))

(R279 (((? x) s133) ((? x) s153) ((? x) s149) ((? x) s150) ((? x) s154)) ((? x) s155))
(R280 (((? x) s130) ((? x) s153) ((? x) s156) ((? x) s154)) ((? x) s157))
(R581 (((? x) s130) ((? x) s158)) ((? x) s159))
(R282 (((? x) s159) ((? y) s160)) ((? x) s145))
(R283 (((? x) s132)) ((? x) s15))
(R284 (((? x) s17) ((? x) s22)) ((? x) s154))
(R285 (((? x) s160)) ((? x) q142))
(R286 (((? x) q142) ((? x) q136) ((? x) s129)) ((? x) s150))
(R287 (((? x) u33)) ((? x) u34))
(R288 (((? x) u35)) ((? x) u34))
(R289 (((? x) u36)) ((? x) u37))
(R290 (((? x) u38) ((? x) u39)) ((? x) u37))
(R291 (((? x) u40)) ((? x) u41))
(R292 (((? x) u42) ((? x) u43) ((? x) u44)) ((? x) u41))
(R293 (((? x) s34) ((? x) u45)) ((? x) u46))
(R294 (((? x) s34) ((? x) u47)) ((? x) u46))
(R295 (((? x) u34) ((? x) u41) ((? x) u48) ((? x) u49)) ((? x) u50))
(R296 (((? x) u34) ((? x) u41) ((? x) u48) ((? x) u51)) ((? x) u52))
(R297 (((? x) u46) ((? x) u53) ((? x) u54) ((? x) u49)) ((? x) u55))
(R298 (((? x) u46) ((? x) u51)) ((? x) u56))
(R299 (((? x) u37) ((? x) u57) ((? x) u53) ((? x) u54) ((? x) u58)) ((? x) u59))
(R300 (((? x) u34) ((? x) u57) ((? x) u60) ((? x) u58)) ((? x) u61))
(R301 (((? x) u34) ((? x) u62)) ((? x) u63))
(R302 (((? x) u63) ((? y) u64)) ((? x) u40))
(R303 (((? x) u52) ((? x) u40)) ((? x) p82))
(R304 (((? x) u47)) ((? x) u78))
(R305 (((? x) u57) ((? y) u68)) ((? x) u62))
(R306 (((? x) u58)) ((? x) t138))
(R307 (((? x) t129)) ((? x) t130))
(R308 (((? x) t131)) ((? x) t130))
(R309 (((? x) t132)) ((? x) t133))
(R310 (((? x) t134) ((? x) t135)) ((? x) t133))
(R311 (((? x) t136)) ((? x) t137))
(R312 (((? x) t138) ((? x) t139) ((? x) t140)) ((? x) t137))
(R313 (((? x) t130) ((? x) t141)) ((? x) t142))
(R314 (((? x) t130) ((? x) t143)) ((? x) t142))
(R315 (((? x) t130) ((? x) t137) ((? x) t144) ((? x) t145)) ((? x) t146))

(R316 (((? x) t130) ((? x) t137) ((? x) t144) ((? x) t147)) ((? x) t148))

(R317 (((? x) t142) ((? x) t149) ((? x) t150) ((? x) t145)) ((? x) t151))

(R318 (((? x) t142) ((? x) t147)) ((? x) t152))

(R319 (((? x) t133) ((? x) t153) ((? x) t149) ((? x) t150) ((? x) t154)) ((? x) t155))

(R320 (((? x) t130) ((? x) t153) ((? x) t156) ((? x) t154)) ((? x) t157))

(R321 (((? x) t130) ((? x) t158)) ((? x) t159))

(R322 (((? x) t159) ((? y) t160)) ((? x) t145))

(R323 (((? x) t132)) ((? x) t15))

(R324 (((? x) t17) ((? x) t22)) ((? x) t154))

(R325 (((? x) t160)) ((? x) q142))

(R326 (((? x) p142) ((? x) s136) ((? x) t129)) ((? x) t150))

(R327 (((? x) M65)) ((? x) M66))

(R328 (((? x) M67)) ((? x) M66))

(R329 (((? x) M68)) ((? x) M69))

(R330 (((? x) M70) ((? x) M71)) ((? x) M69))

(R331 (((? x) M72)) ((? x) M73))

(R332 (((? x) M74) ((? x) M75) ((? x) M76)) ((? x) M73))

(R333 (((? x) M66) ((? x) M77)) ((? x) M78))

(R334 (((? x) M66) ((? x) M79)) ((? x) M78))

(R335 (((? x) M66) ((? x) M73) ((? x) M80) ((? x) M81)) ((? x) M82))

(R336 (((? x) M66) ((? x) M73) ((? x) M80) ((? x) M83)) ((? x) M84))

(R337 (((? x) M78) ((? x) M85) ((? x) M86) ((? x) M81)) ((? x) M87))

(R338 (((? x) M78) ((? x) M83)) ((? x) M88))

(R339 (((? x) M69) ((? x) M89) ((? x) M85) ((? x) M86) ((? x) M90)) ((? x) M91))

(R340 (((? x) M66) ((? x) M89) ((? x) M92) ((? x) M90)) ((? x) M93))

(R341 (((? x) M66) ((? x) t138)) ((? x) u95))

(R342 (((? x) M95) ((? y) M96)) ((? x) M81))

(R343 (((? x) M70) ((? x) M66)) ((? x) q95))

(R344 (((? x) M95) ((? y) q72)) ((? x) q73))

Appendix C: Test Runs

In this appendix, we describe the testing process of the system with greater detail in a test run example.

C.1 Sequential processing case

-The rule base consists of 344 production rules R1-R344, as presented in Appendix C.

- The fact base consists of 46 elements as follow:

(object1 p11) (object1 p10) (object1 p12) (object7 p2) (object1 p15) (object1 p1) (object7 p25) (object1 P63) (object1 p160) (object1 p141) (object7 q95) (object1 q80) (object1 s33) (object1 s42) (object2 P6) (object2 q31) (object6 q32) (object2 q47) (object6 q67) (object2 q122) (object2 p11) (object2 p25) (object6 p98) (object2 p17) (object2 s111) (object2 s85) (object3 s14) (object3 s40) (object4 s78) (object4 q132) (object3 p10) (object5 p14) (object3 q21) (object3 q84) (object3 s128) (object3 p80) (object3 p129) (object3 s33) (object4 p159) (object4 s90) (object8 s90) (object8 q96) (object8 p111) (object9 p70) (object9 s156) (object9 q91) (object10 p1) (object10 p37) (object10 s116) (object11 s15) (object8 u156) (object8 t123) (object9 u40) (object7 u47) (object9 u58) (object10 t136) (object10 t143) (object10 p142) (object10 s127) (object11 q129) (object11 m89) (object11 m85) (object11 m86) (object11 m90)

-The implemented conflict-resolution strategy (Appendix B) is based on the combination of operations of receny, refraction, specificity and random selection. The result of processing is as follow:

(OBJECT7 Q130) (OBJECT7 P144) (OBJECT10 T137) (OBJECT9 T138) (OBJECT1 U78) (OBJECT9 U41) (OBJECT10 P34) (OBJECT10 P2) (OBJECT10 S85) (OBJECT7 Q81)

(OBJECT7 P128) (OBJECT7 S113) (OBJECT4 P145) (OBJECT3 S34) (OBJECT3 P130) (OBJECT4 Q133) (OBJECT3 S41) (OBJECT2 P122) (OBJECT6 Q66) (OBJECT6 S94) (OBJECT2 Q78) (OBJECT2 Q17) (OBJECT1 S34) (OBJECT1 U46) (OBJECT1 P2) (OBJECT1 S85) (OBJECT1 P14) (OBJECT1 P57) (OBJECT1 P60) (OBJECT1 S34)

The result of processing without applying a conflict resolution strategy is:
(OBJECT10 P2) (OBJECT1 P2) (OBJECT1 P9) (OBJECT7 S85) (OBJECT10 P34) (OBJECT1 P34) (OBJECT3 P130) (OBJECT4 P145) (OBJECT2 Q17) (OBJECT2 Q78) (OBJECT6 Q66) (OBJECT7 Q81) (OBJECT2 P122) (OBJECT7 Q130) (OBJECT4 Q133) (OBJECT3 S34) (OBJECT3 S41) (OBJECT7 P144) (OBJECT9 U41) (OBJECT1 U78) (OBJECT9 T138) (OBJECT10 T137) (OBJECT1 P14) (OBJECT1 S85) (OBJECT10 S85) (OBJECT1 P60) (OBJECT7 P128) (OBJECT6 S94) (OBJECT1 U46)

C.2 Parallel processing case

- In this case, the rule base was divided to 10 rule-sets:

RULE-SET1: R1-R20, R327-R344

RULE-SET2: R21-R40

RULE-SET3: R41-R58, R307-R326

RULE-SET4: R59-R77

RULE-SET5: R78-R95, R247-R266

RULE-SET6: R96-R112, R267-R286

RULE-SET7: R113-R132, R192- R210

RULE-SET8: R133-R151, R211- R228

RULE-SET9: R152- R171, R229- R246

RULE-SET10: R172-R191, R287- R306

-The number of PEs is 10 and equals to the number of rule-sets. The contents of working memories of PEs in initialising stage are as follow:

PE1:

(OBJECT10 P2) (OBJECT1 P2) (OBJECT1 P9) (OBJECT7 S85) (OBJECT1 P14)
(OBJECT1 S85) (OBJECT10 S85)

PE2:

(OBJECT10 P34) (OBJECT1 P34)

PE3:

(OBJECT10 T137)

PE4:

NIL

PE5:

(OBJECT3 P130) (OBJECT4 P145) (OBJECT7 P144)

PE6:

(OBJECT2 Q17)

PE7:

(OBJECT2 Q78)

PE8:

(OBJECT6 Q66) (OBJECT7 Q81) (OBJECT3 S34) (OBJECT1 S34) (OBJECT3 S41) (OBJECT6 S94)

PE9:

(OBJECT2 P122)

PE10:

(OBJECT7 Q130) (OBJECT4 Q133) (OBJECT9 U41) (OBJECT1 U78) (OBJECT9 T138)

After doing clustering according to this configuration

((0 1 7 9) (3 4 2) (5 6 8))

the contents of working memories of PEs changed as follows:

PE1:

(OBJECT10 P2) (OBJECT1 P2) (OBJECT1 P9) (OBJECT7 S85) (OBJECT1 S85) (OBJECT10 S85) (OBJECT1 P14)

PE2:

(OBJECT10 P34) (OBJECT1 P34) (OBJECT1 P57) (OBJECT1 P60)

PE3:

(OBJECT10 T137)

PE4:

NIL

PE5:

(OBJECT3 P130) (OBJECT4 P145) (OBJECT7 P144) (OBJECT3 P130)

PE6:

(OBJECT2 Q17)

PE7:

(OBJECT2 Q78)

PE8:

(OBJECT6 Q66) (OBJECT7 Q81) (OBJECT3 S34) (OBJECT1 S34) (OBJECT6 S94) (OBJECT3 S41)

PE9:

(OBJECT2 P122)

PE10:

(OBJECT7 Q130) (OBJECT4 Q133) (OBJECT9 U41) (OBJECT1 U78) (OBJECT9 T138) (OBJECT1 U46)

-The following is the final result, which is an integrated solution of the previous stage:

```
>(print (integrate-partial-result
   (get-partial-results-from-clusters
    '((0 1 7 9) (3 4 2) (5 6 8)))))
```

((OBJECT6 S94) (OBJECT3 S41) (OBJECT1 S34) (OBJECT3 S34) (OBJECT7 Q81) (OBJECT6 Q66) (OBJECT10 S85) (OBJECT1 S85) (OBJECT1 P14) (OBJECT7 S85) (OBJECT1 P9) (OBJECT1 P2) (OBJECT10 P2) (OBJECT10 P34) (OBJECT1 P34) (OBJECT1 P57) (OBJECT1 P60) (OBJECT7 Q130) (OBJECT4 Q133) (OBJECT9 U41) (OBJECT1 U78) (OBJECT9 T138) (OBJECT1 U46) (OBJECT7 P144) (OBJECT4 P145) (OBJECT3 P130) (OBJECT10 T137) (OBJECT2 Q78) (OBJECT2 Q17) (OBJECT2 P122))

C.3 The final results of some other configurations

>(find-clusters)

((3.776651845E7 0 3 2 3 1 0 4 3 2 1) (4.785796905E7 2 3 4 2 3 4 3 0 0 0) (4.91411326E7 1 3 4 4 4 1 3 1 3 0) (4.520797205E7 2 0 2 2 3 4 2 1 0 3) (4.954718015E7 0 3 2 3 4 2 2 1 2 0)

(4.123302345E7 4 3 0 3 2 4 0 4 3 1) (3.854217295E7 0 2 1 1 3 4 3 1 4 2) (4.07816309E7 3 1 4 3 4 0 1 4 0 2) (4.0560131E7 3 1 2 4 2 3 0 4 3 1) (4.230275755E7 1 0 1 0 4 3 2 3 4 1))
(0 4 2 3 1 0 4 3 2 1)
((0 5) (4 9) (2 8) (3 7) (1 6))

```
> (print (integrate-partial-result
   (get-partial-results-from-clusters
    '((0 5) (4 9) (2 8) (3 7) (1 6)))))
```

((OBJECT10 S85) (OBJECT1 S85) (OBJECT1 P14) (OBJECT7 S85) (OBJECT1 P9) (OBJECT1 P2) (OBJECT10 P2) (OBJECT2 Q17) (OBJECT7 P144) (OBJECT4 P145) (OBJECT3 P130) (OBJECT7 Q130) (OBJECT4 Q133) (OBJECT9 U41) (OBJECT1 U78) (OBJECT9 T138) (OBJECT10 T137) (OBJECT2 P122) (OBJECT7 P128) (OBJECT6 Q66) (OBJECT7 Q81) (OBJECT3 S34) (OBJECT1 S34) (OBJECT3 S41) (OBJECT6 S94) (OBJECT1 P34) (OBJECT10 P34) (OBJECT2 Q78))

```
> (print (integrate-partial-result
   (get-partial-results-from-clusters
    '((0 1 7 9) (3 4) (2 5 6 8)))))
```

((OBJECT6 S94) (OBJECT3 S41) (OBJECT1 S34) (OBJECT3 S34) (OBJECT7 Q81) (OBJECT6 Q66) (OBJECT10 S85) (OBJECT1 S85) (OBJECT1 P14) (OBJECT7 S85) (OBJECT1 P9) (OBJECT1 P2) (OBJECT10 P2) (OBJECT10 P34) (OBJECT1 P34) (OBJECT1 P57) (OBJECT1 P60) (OBJECT7 Q130) (OBJECT4 Q133) (OBJECT9 U41) (OBJECT1 U78) (OBJECT9 T138) (OBJECT1 U46) (OBJECT3 P130) (OBJECT4 P145) (OBJECT7 P144) (OBJECT2 Q78) (OBJECT10 T137) (OBJECT2 Q17) (OBJECT2 P122))

```
> (print (integrate-partial-result
   (get-partial-results-from-clusters
    '((0 1 9) (7 3 4) (2 5 6 8)))))
```

((OBJECT1 P60) (OBJECT1 P57) (OBJECT1 P34) (OBJECT10 P34) (OBJECT10 P2) (OBJECT1 P2) (OBJECT1 P9) (OBJECT7 S85) (OBJECT1 P14) (OBJECT1 S85) (OBJECT10 S85) (OBJECT7 Q130) (OBJECT4 Q133) (OBJECT9 U41) (OBJECT1 U78) (OBJECT9 T138) (OBJECT7 P128) (OBJECT6 Q66) (OBJECT7 Q81) (OBJECT3 S34) (OBJECT1 S34) (OBJECT3 S41) (OBJECT6 S94) (OBJECT3 P130) (OBJECT4 P145) (OBJECT7 P144) (OBJECT7 S113) (OBJECT2 Q78) (OBJECT10 T137) (OBJECT2 Q17) (OBJECT2 P122))

C.4 Calculation of time delays

-This code calculates the time spent for sequential processing of information and returns a time list.

```
(defun initialise ()
  (setf time-list nil)
  (let ((i 0))
    (loop
      (when (= i *number-of-PEs*)
        (return-from initialise 'END-OF-THE-FIRST-STAGE))
      (setf (nth i FB) nil)
      (clrhash (nth i RB))
      (setf (nth i AA) nil)
      (delete-file (nth i files-list))
       (setf (nth i FB) (add-fact-from-file
        "f:\\simulati\\testing\\fact.lsp" (nth i FB)))
      (add-rule-from-file (nth i DR) (nth i RB))
       (let ((t0 (get-internal-real-time)))
        (inference-engine i)
        (reverse (push (- (get-internal-real-time) t0)
                time-list)))
      (incf i))))
```

```
> (print time-list)
(13680)
```

The following time-lists are the result of several executions of the code.

(16503)

(11937)

(8823)

(25547)

-In the case of parallel processing, the code returns a time-list with the length equals to the number of PEs. The elements of the time-list are the time spent for each PE in the initialising stage. After several execution of the function we obtained the following lists.

(330 90 491 91 70 190 30 181 50 410), (max 491)

(300 90 481 90 81 190 20 140 50 471), (max 481)

(281 80 511 1803 100 180 20 170 230 380), (max 1803)

(311 90 471 100 90 180 60 181 70 381), (max 471)

(291 90 471 90 111 190 60 231 40 341), (max 471)

-The following code elements calculate the amount processing time delays for each PE, after each cycle of performing required communications.

```
(defun comm-comp (a-cluster-list)
  (loop
    (let ((copy-secondary
          (copy-list (communicate a-cluster-list))))
      (dolist (i a-cluster-list)
       (let ((t0 (get-internal-real-time))
         (inference-engine i)
         (print (- (get-internal-real-time) t0)))
        (print i))
        (when (equal (communicate a-cluster-list)
            copy-secondary) (return copy-secondary)))))
```

-This code calculates the amount of information being communicated within clusters.

```
(defun fire-rule (a-list rule-table i)
  (let ((result (instantiate-variables
              (rule-rhs
                (gethash (first a-list)  rule-table))
              (rest (remove-if #' numberp a-list)))))
    (print-to-file result (nth i files-list))
    (print i)
    (print result)
    (add-object result (nth i FB))))
```

-The following code calculates the execution time of the parallel genetic algorithm.

```
(let ((t0 (get-internal-run-time)))
      (fittest-individual
            (associate-fitness-to-individuals
             (IMCR (fitness-of-initial-population))))
      (- (get-internal-run-time) t0))
  (PGA)
  (let ((t1 (get-internal-run-time)))
      (apply-parallel-hill-climbing j temp)
      (- (get-internal-run-time) t1))
```

-This code calculates the time delays for finding appropriate clusters.

```
(let ((t0 (get-internal-run-time)))
     (find-clusters)
     (- (get-internal-run-time) t0)))
```

C.5 A test-run example of the GA

The initial-generation with associated values is:

((4.53630448E7 3 2 2 2 0 1 4 2 4 4) (4.534350005E7 4 0 2 0 4 2 3 0 3 3) (4.590192045E7 2 0 4 4 0 0 4 0 1 3) (5.84093639E7 0 0 3 0 0 1 1 2 1 0) (4.973217945E7 1 0 1 3 0 1 1 4 4 2) (4.081604755E7 4 1 3 1 3 2 1 2 3 0) (4.699808365E7 1 1 0 4 4 3 2 1 4 4)).

The fittest individual(s) in generation 1 is:

((4.081604755E7 4 1 3 1 3 2 1 2 3 0)).

The fittest individual(s) in generation 2 is:

((3.96187216E7 4 0 2 1 3 2 1 2 3 0)).

The fittest individual(s) in generation 3 is:

((3.55573191E7 1 0 1 3 0 3 2 4 4 2)).

The fittest individual(s) in generation 4 is:

((3.55573191E7 1 0 1 3 0 3 2 4 4 2)).

The fittest individual(s) in generation 5 is:

((3.55573191E7 1 0 1 3 0 3 2 4 4 2)).

The fittest individual(s) in generation 6 is:

((3.999341725E7 1 0 1 3 0 3 2 2 4 2)).

The fittest individual(s) in generation 7 is:

((3.999341725E7 1 0 1 3 0 3 2 2 4 2)).

The fittest individual(s) in generation 8 is:

((3.999341725E7 1 0 1 3 0 3 2 2 4 2)).

The fittest individual(s) in generation 9 is:

((3.76449917E7 1 1 3 0 0 0 4 2 4 2)).

((3.76449917E7 1 1 3 0 0 0 4 ...))

C.6 A test-run example of parallel genetic algorithm

In this example, the algorithm executes a parallel hill-climbing procedure.

The initial-generation with associated values is:

((6.29522149E7 0 0 4 1 4 4 4 0 0 2) (5.50660734E7 3 2 4 1 3 2 3 1 3 1) (4.648158685E7 2 3 1 0 3 1 0 2 2 3) (9.17882812E7 2 0 2 3 3 3 2 2 2 2) (4.012105935E7 0 2 2 1 3 0 3 4 1 3) (4.203946675E7 3 1 2 4 3 0 2 0 4 0) (6.719472345E7 2 0 0 0 1 2 3 0 0 0)).

The fittest individual(s) in generation 1 is:

((3.75836218E7 0 0 4 1 3 1 0 2 2 4)).

The fittest individual(s) in generation 2 is:

((3.740424175E7 0 2 2 1 3 0 2 1 3 4)).

The fittest individual(s) in generation 3 is:

((3.73754552E7 0 0 4 1 3 3 1 1 4 2)).

The fittest individual(s) in generation 4 is:

((3.542102995E7 0 3 3 4 2 0 2 1 1 4)).

The fittest individual(s) in generation 5 is:

((3.55856032E7 2 0 4 1 3 2 0 1 3 4)).

The fittest individual(s) in generation 6 is:

((3.759123925E7 0 2 1 0 3 0 2 4 3 1)).

The fittest individual(s) in generation 7 is:

((3.732481495E7 2 0 3 1 2 0 0 1 3 4)).

The fittest individual(s) in generation 8 is:

((3.755242515E7 3 0 2 1 3 0 0 1 4 2)).

The fittest individual(s) in generation 9 is:

((3.765146135E7 0 0 4 4 3 2 2 1 1 4)).

((3.765146135E7 0 0 4 4 3 2 2 ...)).