12-1984

Resolution Algorithms in a Parallel Processing Environment

Paula Beth Brandman Hencken

Western Michigan University

Follow this and additional works at: http://scholarworks.wmich.edu/masters_theses

Part of the Computer Sciences Commons

Recommended Citation
http://scholarworks.wmich.edu/masters_theses/1493

This Masters Thesis-Open Access is brought to you for free and open access by the Graduate College at ScholarWorks at WMU. It has been accepted for inclusion in Master's Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.
RESOLUTION ALGORITHMS
IN A
PARALLEL PROCESSING ENVIRONMENT

by

Paula Beth Brandman Hencken

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
December 1984
RESOLUTION ALGORITHMS
IN A
PARALLEL PROCESSING ENVIRONMENT

Paula Beth Brandman Hencken, M.S.
Western Michigan University, 1984

An ever increasing number of processors are becoming available to the multiprocessor user. For faster response and efficient utilization of parallel processing systems it is necessary to develop parallel algorithms to solve a single problem.

This paper evaluates the complexity of resolution algorithms run in a parallel processing environment. An overview of parallel processing is presented along with a discussion of general resolution and resolution-refutation systems. An analysis of the complexity required by different control strategies to perform resolution programming is presented using both uniprocessor and multiprocessor systems.
ACKNOWLEDGEMENTS

I would like to express my sincere appreciation and gratitude to my thesis advisor, Dr. Dionysios Kountanis, for his counsel and patience. I have gained much from his guidance, inspiring comments, suggestions and discussions not only on this thesis but throughout my studies at Western. I would also like to thank Dr. Dalia Motzkin for her general encouragement and warmth and for her review and helpful suggestions on this thesis.

A special thanks goes to Mr. Bruce Mull and Ms. Gloria Melara for their indispensable help.

I would also like to thank Mr. Robert Hencken for receiving the draft of this thesis and pointing out corrections, Ms. Eileen Sinicki for her tolerance and accurate typing, and Ms. Telma Lando for her superb drawings.

I will always be grateful for my friendship with Ms. Marcia Healey who is always willing to listen and help me keep my perspective.

My thanks also to all my friends in the Computer Science department for being there when I needed an ear and to those other people who have given their encouragement and support.

Finally to my children, Alex and Randy, who never
want to hear the word thesis again, I thank you with all my heart for the love we share.

Without the help of all these people, this thesis would never have been completed.

Paula Beth Brandman Hencken
INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
HENCKEN, PAULA BETH BRANDMAN

RESOLUTION ALGORITHMS IN A PARALLEL PROCESSING ENVIRONMENT

WESTERN MICHIGAN UNIVERSITY M.S. 1984

University Microfilms International 300 N. Zeeb Road, Ann Arbor, MI 48106

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark. 

1. Glossy photographs or pages 
2. Colored illustrations, paper or print 
3. Photographs with dark background 
4. Illustrations are poor copy

5. Pages with black marks, not original copy 
6. Print shows through as there is text on both sides of page 
7. Indistinct, broken or small print on several pages

8. Print exceeds margin requirements 
9. Tightly bound copy with print lost in spine 
10. Computer printout pages with indistinct print 

11. Page(s) _________ lacking when material received, and not available from school or author.

12. Page(s) _________ seem to be missing in numbering only as text follows.

13. Two pages numbered _________. Text follows.

14. Curling and wrinkled pages 

15. Other

University Microfilms International
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS ..................................................... ii
LIST OF TABLES ........................................................ v
LIST OF FIGURES ...................................................... vi

Chapter

I. INTRODUCTION ........................................ 1
II. AN OVERVIEW OF PARALLEL PROCESSING ........ 4
III. REVIEW OF LITERATURE ............................... 15
IV. RESOLUTION ............................................. 32
    Historical Background .................................... 32
    Usefulness of Resolution ................................ 34
    First Order Predicate Calculus ....................... 34
    Rules of Inference ...................................... 37
    Unification ............................................ 38
    Tautologies ........................................... 39
    Clauses ............................................... 40
V. RESOLUTION-REFUTATION SYSTEMS ................. 45
    Derivation Graph ..................................... 48
    Strategies ........................................... 48
    Breadth-first Strategy ............................... 50
    Set-of-support Strategy .............................. 50
    Unit-preference Strategy ............................ 52
    Linear-input Form Strategy ......................... 52
    Ancestry-filtered Form Strategy .................. 55

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
VI. COMPLEXITY OF PARALLEL RESOLUTION ALGORITHM . 58
Breadth-first Strategy .................... 59
Set-of-support .......................... 68
Unit-preference Strategy ............... 75
Linear-input Form Strategy ............. 79
Ancestry-filtered Form ................. 80
VII. CONCLUSION ........................... 88
BIBLIOGRAPHY ............................... 93
LIST OF TABLES

1. Breadth-first Complexity ................................ 66
2. Breadth-first Complexity with all Clauses Unique . 67
3. Set-of-support Complexity ............................. 74
4. Unit-preference Complexity ............................. 78
5. Linear-input Form ................................. 83
6. Ancestry Complexity ............................... 87
LIST OF FIGURES

1. A Resolution Refutation Tree .......................... 49
2. Breadth-first Strategy .................................. 51
3. Set-of-support Strategy ................................. 53
4. Unit-preference Strategy ................................. 54
5. Linear-input Form Strategy .............................. 56
6. Ancestry-filtered Form Strategy ......................... 57
7. Breadth-first on a Parallel Processor ................. 60
8. Breadth-first on a System with Two Processors ........ 63
9. Breadth-first on a Uniprocessor ......................... 69
10. Set-of-support on a Parallel Processor ............... 70
11. Set-of-support on a Uniprocessor ...................... 72
12. Unit-preference on a Parallel Processor .............. 76
13. Unit-preference on a Uniprocessor ..................... 77
14. Linear-input Form on a Parallel Processor ........... 81
15. Linear-input Form on a Uniprocessor ................. 82
16. Ancestry-filtered Form on Parallel Processor ......... 85
17. Ancestry-filtered Form on a Uniprocessor ............. 86
CHAPTER I

INTRODUCTION

Logical reasoning is the key to solving problems in mathematics, from designing and validating electronic circuits to answering many everyday questions. Computers can be programmed to reason logically and to assist in both practical and theoretical investigations.

A computer must receive explicit information. A large number of steps may be necessary to answer even a simple question.

A reasoning program has the ability to reason flawlessly and to consider more alternatives than would a person. But a reasoning program cannot be allowed to simply reason methodically and exhaustively because there are usually too many possibilities to be considered. Therefore, it is necessary to impose a strategy on the number of ways the program can reason and on the number of ways that reasoning can be controlled. To receive the maximum benefit from a reasoning program, good choices must be made about the ways that it can reason and the strategies that it can impose for controlling its reasoning.

Even with good strategies, a resolution program can take an enormous amount of time, as will be shown in this
paper. Any future improvement in speed depends upon being able to decompose a resolution algorithm and to run the program in parallel. Achieving high performance depends not only on efficient algorithms and on faster and more reliable hardware devices, but also upon major improvements in computer architecture and processing techniques.

Advanced computer architectures are centered around the concept of parallel processing. State-of-the-art parallel computer systems can be divided into three structural classes: pipelined computers, array processors and multiprocessor systems. The development and application of these computer systems require a broad knowledge of the underlying hardware and software structures and close interactions between parallel computing algorithms in order to achieve optimal allocation of machine resources. (Hwang and Briggs 1984)

This paper discusses effects on the complexity of resolution when it is run in a parallel processing environment. Chapter two gives a general overview of parallel processing. Basic concepts of parallel processing algorithms are discussed, such as synchronous and asynchronous, concurrency control, granularity and fluctuations in process speed.

Chapter three reviews the relevant literature of
parallel processing algorithms, the complexity of various algorithms, and specific architecture are discussed.

Chapter four examines the history of the development of resolution systems and how they operate. It also includes a brief presentation of first order predicate calculus and a procedure for converting any predicate calculus well formed formula ("wff") into a set of clauses which are used by the resolution.

Chapter five explains resolution-refutation systems. The following control strategies are discussed: breadth-first, set-of-support, unit preference, linear-input-form and ancestry filtered form.

Chapter six explores the use of resolution on a parallel processing system. A formula that measures time needed to run the resolution in comparison to the number of processors needed is developed for algorithms using the strategies presented in chapter five.

Chapter seven is a discussion of the findings and suggests areas for further research.
Computing speed of digital processors has been increasing in leaps and bounds ever since the first electronic computer was built almost four decades ago. While it does not appear that the future will witness the same rate of increase in the speed of computers the cost of hardware will be decreasing steadily, thanks to the technology of large scale integration. (Lakshmivarahan and Dhall and Miller 1982)

It has been recognized that the speed of a single computer is limited by technology. To increase the throughput further, multiple computers have to be connected together to form a parallel processor system.

Such systems are called multi-processor systems (MPS) because more than one processor share the memory and the computational load. Parallelism is an efficient method to increase the speed of computation in both hardware and software systems.

In these multiprocessor systems, as a rule, no single processor owns any subset of resources, e.g. memory, I/O devices, etc. Rather many processors share the resources belonging to a common pool. Each of the processors in an MPS may be simultaneously executing separate or related
programs without any kind of time sharing or multiprogramming technique.

Following Flynn's (1966) classification scheme, parallel computers are classified into SIMD (single-instruction stream-multiple data stream) machines and MIMD (multiple-instruction stream-multiple data stream) machines (Kung 1981). The essential computing process is the execution of a sequence of instructions on a set of data. The term stream is used to denote a sequence of items (instructions or data) as executed or operated upon by a single processor. An instruction stream is a sequence of instructions as executed by a machine; a data stream is a sequence of data (including input, partial, or temporary results), called for by the instruction stream (Hwang and Briggs 1984). With SIMD machines, one stream of instructions controls a number of synchronized processors, each operating upon its own memory. With MIMD machines, the processors have independent instruction counters, and operate in a speed independent manner on shared memories. A MIMD system is equivalent to a system of interconnected conventional processors, in the sense that at any given time each processor in the system executes one instruction, and the instruction operates on one datum. The system as a whole supports parallel operation in that the individual processors can execute
simultaneously. Complex programs are run by partitioning them into smaller serial programs that run on the individual processors. The programs on the several processors communicate with each other during execution to share information and to otherwise cooperate in the solution of the problem (Stone 1973). Parallel algorithms are tailor made for a specific architecture. Considerations in designing algorithms for SIMD machines and those for MIMD machines are quite different.

In a parallel algorithm, because more than one task module can be executed at a time, concurrency control is needed to ensure the correctness of the concurrent execution. The concurrency control enforces desired interactions among task modules so that the overall execution of the parallel algorithm will be correct (Kung 1981). Concurrency control can be centralized or decentralized, and synchronous or asynchronous. For example, algorithms for SIMD machines may be characterized by centralized, synchronous control. All the special purpose networks have distributed, synchronous control. For synchronous machines some of the basic problems are the choice of interconnection network and the pattern of data movement. Algorithms for MIMD machines could be either synchronous or asynchronous (Lakshmivarahan, Khall and Miller 1982).
Synchronized algorithms have some processes which have to wait on other processes. Since the execution time of a process is variable, depending on the input data and system interruptions, all the processes that have to synchronize at a given point wait for the slowest among them to be executed. This worst case computation speed is a basic weakness of synchronized algorithms and may result in worse than expected time and processor utilization. Asynchronous algorithms remedy the problems encountered by some synchronized parallel algorithms. In an asynchronous algorithm processes are not generally required to wait for each other and communication is achieved by reading dynamically updated global variables stored in shared memory. However, because of the concurrent memory accesses, conflicts occur which introduce some small delay in processes accessing common variables.

Another alternative approach to constructing parallel algorithms is macropipelining, which is applicable if the computation can be divided into parts, called stages, so that the output of one or several collected parts is the input for another part (Hwang and Briggs 1984).

To ensure that a parallel algorithm works correctly and uses parallelism effectively for solving a given problem, it is usually necessary to have interactions among the processes. Hence, in the program which controls
a process there may be some points where the process can communicate with other processes. These are called interaction points. The interaction points divide a process into stages. At the end of each stage a process may communicate with other processes before starting the next stage.

The module granularity of a parallel algorithm refers to the maximal amount of computation a typical task module can perform before having to communicate with other modules. The module granularity of a parallel algorithm reflects whether or not the algorithm tends to be communication intensive. For example, a parallel algorithm with a small module granularity will require frequent intermodule communication. Algorithms for a MIMD machine will, as a rule, have large module granularity, but those for special purpose networks have the least modular granularity (Lakshmivarahan, Dhall and Miller 1982).

There may be many fluctuations in process speed. The time taken by a fixed stage of a process is usually not a constant. The fluctuations may be due to both the multiprocessor system and the input to the stage.

The multiprocessor may consist of processors with different speeds. For example, the current configuration of C.mmp includes both PDP-11/20 and PDP-11/40 processors.
The latter processors are considerably faster than the former ones. A process may be run in a fast or slow manner, depending upon which processors are assigned to it during the stage.

From time to time the operating system on the multi-processor environment may assign certain processors to perform I/O, allocate processors to processes, switch a processor from one process to another, and so on. Hence during the period when a processor is carrying out a stage of a process, it could be interrupted by the operating system and start doing something else. In this case, a time consuming context swap is performed, and the stage is either taken over by another processor or suspended for an indefinite amount of time.

In a multiple user environment, the amount of resources allocated to a particular process at a given time is a variable, depending upon the number of processes the users have created and their priorities. Thus, the speed of a process may be influenced by the whole user community.

Also, as mentioned earlier, processors may be asynchronous and a process may be delayed due to memory conflicts.

The time taken by a parallel algorithm is defined to be the elapsed time of the process in the algorithm which
finishes last. The elapsed time of a process is the sum of the following three quantities: basic processing time, block time and execution time of synchronization primitives.

Recall that a process consists of consecutive stages and that the time taken by a stage is a random variable. The basic processing time of a process is the sum of the times taken by its stages.

A process may be blocked at the end of a stage because it waits for inputs in a synchronized algorithm, or for the entering of a critical section in an asynchronous algorithm. The blocking time of a process is the total time that the process is blocked.

Synchronization primitives are needed for synchronizing processes and implementing critical sections. The execution time of these primitives is often non-negligible in practice. (Kung 1981).

A widely accepted measure of the effectiveness of a parallel algorithm is the so called speed up ratio and it is defined to be the ratio of the time, measured in terms of the number of certain basic operations, required to solve the problem using the best uniprocessor algorithm to the time taken by a given parallel algorithm. Thus, speed up ration, \( S_k(n) = \frac{T_k(n)}{T_1(n)} \) where \( T_1(n) \) is the best-worst case time complexity on a problem size \( n \) using \( i \).
processor. Thus, \( S(n) \leq K \) and \( S(n) = K \) is the optimal speed up ratio. (Lakshmivarahan, Dhall, and Miller 1982)

New architectural features require new computer languages as well as new algorithms. In the case of parallel computers that are designed to work most efficiently on one or sometimes two-dimensional lists of data, this is most naturally achieved by introducing the mathematical concept of vector or matrix into the computer language and by analysing algorithms in terms of their suitability for execution on vectors of data. The principal ways of introducing parallelism into the architecture of computers may be summarised as follows:

1. Pipelining which is the application of assembly-line techniques to improve the performance of an arithmetic or control unit. To achieve pipelining, one must subdivide the input task (process) into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline. Successive tasks are streamed into the pipe and get executed in an overlapped fashion at the subtask level. The pipeline consists of a cascade of processing stages. The stages are pure combinational circuits performing arithmetic or logic operations over the data stream flowing through fast registers for holding the intermediate results between the stages. Information
flows between adjacent stages are under the control of a common clock applied to all the latches simultaneously.

(2) Functional which provides several independent units for performing different functions, such as logic, addition or multiplication, and which allows these to operate simultaneously on different data.

(3) Array which provides an array of identical processing elements (PES) under common control, all performing the same operation simultaneously but on different data stored in their private memories.

(4) Multiprocessing which is the provision of several processors each obeying its own instructions, and usually communicating via a common memory.

Individual designs may combine some or all of these parallel features. For example, a processor array may have pipelined arithmetic units as its PES, and one functional unit in a multi-unit computer might be a processor array.

Up to 1980, multiprocessor designs have been concerned primarily with methods of connecting several independent computers as to maximise the throughput of a computer installation. (Hockney & Jesshope 1981)

The rapid advent of very-large-scale integrated (VLSI) technology has created a new architectural horizon in implementing parallel algorithms directly in hardware.
Data flow and VLSI offer two mutually supportive approaches towards the design of future supercomputers. Data flow computers are based on the concept of data-driven computation, which is drastically different from the operation of a conventional Von Newmann machine. The fundamental difference is that instruction execution in a conventional computer is under program-flow control, whereas that in a data flow computer is driven by the data (operand) availability. Due to their strong appeal to parallelism, data flow techniques have attracted a great deal of attention in recent years.

Highly parallel computing structures promise to be a major application area for the million-transistor chips that will be possible in just a few years. This means that a VLSI chip will contain more than one million individual transistors. One such chip may contain more functions than one of today's large minicomputers. Such computing systems have structural properties that are suitable for VLSI implementation. Almost by definition, parallel structures imply a basic computational element repeated perhaps hundreds or thousands of times. This architectural style immediately reduces the design problem by similar orders of magnitude. The degree of concurrency in a VLSI computing structure is largely determined by the underlying algorithm. Massive parallelism can be achieved
if the algorithm is designed to introduce high degrees of pipelining and multiprocessing. When a large number of processing elements work simultaneously, coordination and communication become significant. Properly designed parallel structures that need to communicate only with their nearest neighbors will gain the most from very-large-scale integration. Precious time is lost when modules that are far apart must communicate. (Hwang and Briggs 1984)
Synchronized algorithms have some processes which have to wait on other processes. Since the execution time of a process is variable, depending on the input data and system interruptions, all the processes that have to synchronize at a given point wait for the slowest among them to be executed. This worst case computation speed is a basic weakness of synchronized algorithms and may result in worse than expected time and processor utilization. Asynchronous algorithms remedy the problems encountered by some synchronized parallel algorithms. In an asynchronous algorithm processes are not generally required to wait for each other and communication is achieved by reading dynamically updated global variables stored in shared memory. However, because of the concurrent memory accesses, conflicts occur which introduce some small delay in processes accessing common variables.

Another alternative approach to constructing parallel algorithms is macropipelining, which is applicable if the computation can be divided into parts, called stages, so that the output of one or several collected parts is the input for another part (Hwang and Briggs 1984).

To ensure that a parallel algorithm works correctly and uses parallelism effectively for solving a given problem, it is usually necessary to have interactions among the processes. Hence, in the program which controls
a process there may be some points where the process can communicate with other processes. These are called interaction points. The interaction points divide a process into stages. At the end of each stage a process may communicate with other processes before starting the next stage.

The module granularity of a parallel algorithm refers to the maximal amount of computation a typical task module can perform before having to communicate with other modules. The module granularity of a parallel algorithm reflects whether or not the algorithm tends to be communication intensive. For example, a parallel algorithm with a small module granularity will require frequent intermodule communication. Algorithms for a MIMD machine will, as a rule, have large module granularity, but those for special purpose networks have the least modular granularity (Lakshmivarahan, Dhall and Miller 1982).

There may be many fluctuations in process speed. The time taken by a fixed stage of a process is usually not a constant. The fluctuations may be due to both the multiprocessor system and the input to the stage.

The multiprocessor may consist of processors with different speeds. For example, the current configuration of C.mmp includes both PDP-11/20 and PDP-11/40 processors.
The latter processors are considerably faster than the former ones. A process may be run in a fast or slow manner, depending upon which processors are assigned to it during the stage.

From time to time the operating system on the multi-processor environment may assign certain processors to perform I/O, allocate processors to processes, switch a processor from one process to another, and so on. Hence during the period when a processor is carrying out a stage of a process, it could be interrupted by the operating system and start doing something else. In this case, a time consuming context swap is performed, and the stage is either taken over by another processor or suspended for an indefinite amount of time.

In a multiple user environment, the amount of resources allocated to a particular process at a given time is a variable, depending upon the number of processes the users have created and their priorities. Thus, the speed of a process may be influenced by the whole user community.

Also, as mentioned earlier, processors may be asynchronous and a process may be delayed due to memory conflicts.

The time taken by a parallel algorithm is defined to be the elapsed time of the process in the algorithm which
finishes last. The elapsed time of a process is the sum of the following three quantities: basic processing time, block time and execution time of synchronization primitives.

Recall that a process consists of consecutive stages and that the time taken by a stage is a random variable. The basic processing time of a process is the sum of the times taken by its stages.

A process may be blocked at the end of a stage because it waits for inputs in a synchronized algorithm, or for the entering of a critical section in an asynchronous algorithm. The blocking time of a process is the total time that the process is blocked.

Synchronization primitives are needed for synchronizing processes and implementing critical sections. The execution time of these primitives is often non-negligible in practice. (Kung 1981).

A widely accepted measure of the effectiveness of a parallel algorithm is the so called speed up ratio and it is defined to be the ratio of the time, measured in terms of the number of certain basic operations, required to solve the problem using the best uniprocessor algorithm to the time taken by a given parallel algorithm. Thus, speed up ration, $S_R(n) = T_R(n)/T_1(n)$ where $T_1(n)$ is the best-worst case time complexity on a problem size $n$ using i
Thus, \( S(n) \leq K \) and \( S(n) = K \) is the optimal speed up ratio. (Lakshmivarahan, Dhall, and Miller 1982)

New architectural features require new computer languages as well as new algorithms. In the case of parallel computers that are designed to work most efficiently on one or sometimes two-dimensional lists of data, this is most naturally achieved by introducing the mathematical concept of vector or matrix into the computer language and by analysing algorithms in terms of their suitability for execution on vectors of data. The principal ways of introducing parallelism into the architecture of computers may be summarised as follows:

1. Pipelining which is the application of assembly-line techniques to improve the performance of an arithmetic or control unit. To achieve pipelining, one must subdivide the input task (process) into a sequence of subtasks, each of which can be executed by a specialized hardware stage that operates concurrently with other stages in the pipeline. Successive tasks are streamed into the pipe and get executed in an overlapped fashion at the subtask level. The pipeline consists of a cascade of processing stages. The stages are pure combinational circuits performing arithmetic or logic operations over the data stream flowing through fast registers for holding the intermediate results between the stages. Information
flows between adjacent stages are under the control of a common clock applied to all the latches simultaneously.

(2) Functional which provides several independent units for performing different functions, such as logic, addition or multiplication, and which allows these to operate simultaneously on different data.

(3) Array which provides an array of identical processing elements (PES) under common control, all performing the same operation simultaneously but on different data stored in their private memories.

(4) Multiprocessing which is the provision of several processors each obeying its own instructions, and usually communicating via a common memory.

Individual designs may combine some or all of these parallel features. For example, a processor array may have pipelined arithmetic units as its PES, and one functional unit in a multi-unit computer might be a processor array.

Up to 1980, multiprocessor designs have been concerned primarily with methods of connecting several independent computers as to maximise the throughput of a computer installation. (Hockney & Jesshope 1981)

The rapid advent of very-large-scale integrated (VLSI) technology has created a new architectural horizon in implementing parallel algorithms directly in hardware.
Data flow and VLSI offer two mutually supportive approaches towards the design of future supercomputers. Data flow computers are based on the concept of data-driven computation, which is drastically different from the operation of a conventional Von Neumann machine. The fundamental difference is that instruction execution in a conventional computer is under program-flow control, whereas that in a data flow computer is driven by the data (operand) availability. Due to their strong appeal to parallelism, data flow techniques have attracted a great deal of attention in recent years.

Highly parallel computing structures promise to be a major application area for the million-transistor chips that will be possible in just a few years. This means that a VLSI chip will contain more than one million individual transistors. One such chip may contain more functions than one of today's large minicomputers. Such computing systems have structural properties that are suitable for VLSI implementation. Almost by definition, parallel structures imply a basic computational element repeated perhaps hundreds or thousands of times. This architectural style immediately reduces the design problem by similar orders of magnitude. The degree of concurrency in a VLSI computing structure is largely determined by the underlying algorithm. Massive parallelism can be achieved
if the algorithm is designed to introduce high degrees of pipelining and multiprocessing. When a large number of processing elements work simultaneously, coordination and communication become significant. Properly designed parallel structures that need to communicate only with their nearest neighbors will gain the most from very-large-scale integration. Precious time is lost when modules that are far apart must communicate. (Hwang and Briggs 1984)
CHAPTER III

REVIEW OF LITERATURE

Harold Stone (1973) states, according to current research in parallel computation, "that we cannot directly apply our knowledge of serial computation to parallel computation because efficient serial algorithms do not always lead to efficient parallel algorithms." (Stone p. 1) He shows that parallel computation is not a simple generalization of serial computation, but that it is an area with its own set of characteristic problems over and above any attributes inherited from serial computation. The indications raised in his paper suggest that the most efficient parallel algorithm for a computation is not necessarily a direct adaptation of the most efficient serial algorithm.

H. T. Kung (1980) presents many examples of parallel algorithms and studies them under a uniform framework. He identifies the important attributes of a parallel algorithm as concurrency control, module granularity, and communication geometry. His classification of parallel algorithms corresponds naturally to that of parallel architectures, SIMD, MIMD and "Sestolic". (A sestolic system consists of a set of interconnected cells, each capable of performing some simple operation. It optimizes
the hardware to fit major computation requirements used by
the machine.) Kung shows that issues concerning
algorithms for synchronous parallel computers are quite
different from those for asynchronous parallel computers.
For synchronous parallel computers, one is concerned with
algorithms defined on networks. Task modules of an
algorithm are simply computations associated with modes of
the underlying network. Communication geometry and data
movement are major parts of an algorithm. For chip
implementation it is essential that the communication
geometry be simple and regular, and that silicon area
rather than the number of gates alone be taken into
consideration. For asynchronous parallel computers, the
concern is with parallel algorithms whose task modules are
executed by asynchronous processes. The major issues are
the correctness and efficiency of an algorithm in the
presence of the asynchronous behavior of its processes.

H. Niemann and H. W. Hein (1981) present a system
concept which enables several independent processes, each
having specialized knowledge about one aspect of a complex
real world pattern, to cooperate in parallel, analyzing
and understanding this pattern automatically. They
introduce a software environment providing an experimenter
with tools to define different system configurations and
to develop reasonable control strategies for them. This
is intended to be usable just as the automatic understanding of continuously spoken human sentences.

Leon E. Winslow and Yuan-Chieh Chow (1981) discuss a unified approach for analyzing and classifying parallel sorting machines. Their approach accounts for a number of important factors that have significant impact on the measure of the size and complexity of parallel sorters. These factors include the sorter architectures, sequential/parallel impact, single/multiple passes and the base of the comparators.

Winslow and Chow show how sorters with sort times of $O(N\log_b N\log_b \text{MAX})$, $O(\log_b N\log_b \text{MAX})$, $O(1)$, $O(N\log_b N\log_b N)$, and $O(\log_b N)$ can be achieved where MAX is the largest value on the list. The radix sort and balanced tree sort are used as examples. The efficiency measure of the parallel sorters is proved to be useful in describing the behavior of the sorters. A new sorter with sort time of $O(c^t\log_b N)$ and efficiency $(1/c) > 0.5$ is proposed. This sorter could achieve near optimum efficiency with good software support. Some hardware for the implementation of the new sorter is discussed.

Sakti Pramanik (1981) describes several hardware algorithms for finding pattern matches in a linear text string stored on a circulating type device. The text is read as a sequence of substrings and search is parallel.
within each substring. Substring length can be arbitrarily chosen and this division is independent of the logical partitioning of the data. The algorithm developed is used in the design of a hardware associative search. Reduced complexity of the hardware is achieved by properly cascading many simple cells, where each cell can be a single character comparator.

S. C. Chen (1976) has developed parallel and direct computational algorithms that evaluate linear recurrence systems with constant coefficients. He shows that $O(\log_2 m \log_2 n)$ time steps and $O(mn)$ processors are sufficient to solve such a system. ($n$ is the size of the matrix and $m$ is the size of the bandwidth of a triangular matrix.) General recurrences with $m = n$ is computed with $O(\log_2 n)$ time steps with at most $O(n^2/4)$ processors. The algorithms are designed with easy data routing and simple machine control structures which can be easily implemented through software such as parallel compiler algorithms, numeric subroutines, or hardware control programs for future parallel or pipeline processors.

Siegel, Siegel, and Swain (1982) define and apply a number of performance measures for evaluating SIMD algorithms from the image processing problem domain. Their paper deals with the complexity of parallel algorithms which is a function of the machine size (number
of processors), problem size and type of interconnection network used to provide communication among the processors. They discuss measures which quantify the effect of changing the machine-size/problem and size/network-type relationships. The measures they discuss and compare include execution time, speed, parallel efficiency, overhead ratio, processor utilization, redundancy, cost effectiveness, speed-up of the parallel algorithm over the corresponding serial algorithm, and a price measure which assigned a weighted value to computations and processors. The cost effectiveness and price criteria attempt to provide an integrated picture of speed and resource utilization. From their study, one may conclude that no single measure is sufficient for all processing environments. Siegel, Siegel and Swain (1982) state that these measures could also be applied to algorithms for MIMD machines. However, due to the asynchronous nature of MIMD processing, analysis of algorithms is more difficult to perform.

Parkinson and Liddell (1983) find that there is no suitable yardstick for absolute performance estimation for a highly parallel system. However, it is possible to compare the performance of any two computers when solving the same problem. The results of this comparison can vary strongly with algorithm and size of problem. Thus a
single comparison does not form an adequate basis for estimating the relative performance of those computers on other applications or other problem sizes.

Parkinson & Liddell (1983) state that an algorithm developed for a serial machine is not usually suitable for parallel computation. Every computer has its "degree of parallelism" which must be compared with the parallelism of the problem to be solved. According to them, in certain cases which arise in the solution of large problems, the parallelism of the problem may greatly exceed that of the computer used for the solution, so a mixture of serial and parallel techniques should be employed.

Sameh (1981) presents an account of numerical algorithms suitable for parallel processing. The algorithms presented were originally designed for SIMD computers. Algorithms for evaluating arithmetic expressions and for solving standard eigenvalue problems and a comprehensive summary of direct linear system solvers are discussed. The speed up of a parallel algorithm is defined as the ratio of the corresponding sequential and parallel times. For example, if $T_k$ is the time required by an algorithm using $k$ processors, and $T_1$ is the minimum time required by the same algorithm using one processor then the speed up is given by $T_1/T_k \geq 1$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Another important parameter discussed is the redundancy of a parallel algorithm. If the parallel algorithm requires \( w \) arithmetic operations, then its redundancy is defined by \( p = \frac{w}{o} \geq 1 \); (\( w \) is the number of distinct nodes in the computational graph of the parallel algorithm and \( o \) is the minimum time required by the same algorithm using one processor.) The larger the redundancy of a parallel algorithm, the larger the probability that the parallel algorithm is not as numerically stable as its sequential counterpart.

Sameh (1981) is also concerned with the rounding errors in the sequential and parallel algorithms. He finds that the parallel algorithm has a lower absolute error bound than that of the sequential algorithm and generally yields more accurate results.

Stone (1978) gives timing comparisons for three sorting algorithms written on the CDC STAR computer. One algorithm is Hoare's Quicksort, which is the fastest (or nearly the fastest) sorting algorithm for most computers. A second algorithm is a vector version of Quicksort that takes advantage of STAR's vector operations. The third algorithm is an adaptation of Batcher's sorting algorithm, which makes especially good use of vector operations but has a complexity of \( N(\log N)^2 \) as compared to a complexity of \( N \log N \) for the Quicksort algorithm. In spite of its
greater complexity, Batcher's sorting algorithm is competitive with the serial version of Quicksort for vectors up to the largest that can be treated by STAR. Vector Quicksort outperforms the other two algorithms and it is generally preferred. These results indicate that unusual instruction sets can introduce biases in program execution time that counter results predicted by a worst-case asymptotic complexity analysis.

Bayer and Schkolnick (1981) examine the problem of concurrent access to B-trees. They discuss an accessing scheme which achieves a high degree of concurrency in using the index. They present a deadlock-free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters to satisfy these requirements. The solution presented uses simple locking protocols. They conclude that B-trees can be used advantageously in a multi-user environment.

Baer and Russel (1970) are concerned with the modeling of computations which can be divided into four general areas:

1. those general models concerned primarily with formal aspects of parallel processing without regard to actual programming considerations;

2. models that incorporate new programming languages designed to enhance parallel processing;
(3) models that incorporate extensions of existing sequential programming languages; and

(4) models that attempt to detect and represent parallelism in existing sequential languages.

In their paper, Baer and Russel (1970) review the U.C.L.A. approach to modeling computer programs for parallel processing systems. The main originality and the strongest feature of the U.C.L.A. modeling procedure is that, starting with already existing programs written in FORTRAN, one can automatically derive the model and predict the performance of the program if it runs on a system of K homogeneous parallel processors.

Baer and Russel further state that the type of parallel processing recognition demonstrated is by its very nature limited. The technique of starting from a program written for a single processor has natural deficiencies.

Berman and Synder (1984) are concerned with the mapping problem that arises when the interconnection structure of a parallel algorithm differs from the interconnection architecture of the intended parallel computer. This problem is intensified when the number of processes required by the algorithm exceeds the number of processors available in the computer. Their paper deals with a general methodology for the mapping problem when
there is topological and cardinality variation. They illustrate this methodology by presenting and evaluating a mapping for tree algorithms into shuffle-exchange architectures.

Stolfo and Miranker (1984) discuss DADO, a parallel, tree-structured machine designed to provide significant performance improvements in the execution of large expert systems implemented in production system form. A full-scale version of the DADO machine would comprise a large set of processing elements each containing its own processor, a small amount of local random access memory, and a specialized I/O switch. The processing elements would be interconnected to form a complete binary tree.

Their paper describes the application domain of the DADO machine and the rationale for its design. They then focus on the machine architecture and detail the hardware design of a moderately large prototype comprising 1023 microprocessors currently under development at Columbia University. They conclude with very encouraging performance statistics recently calculated from an analysis of extensive simulations of the system.

Knowledge-based expert systems are Artificial Intelligence problem-solving programs designed to operate in narrow "real-world" domains that perform tasks with the same competence as a skilled human expert. These systems
are illucidation of unknown chemical compounds and medical diagnosis. The heart of these systems is a knowledge base, a large collection of facts, definitions, procedures and heuristic "rules of thumb", acquired directly from a human expert.

Gupta (1984) uses DADO, a highly parallel tree-structured architecture designed to execute production systems at Columbia University to analyze the performance of it when executing production systems language programs. The analysis is based on the predicted performance of three different algorithms for implementing production systems on DADO. The analysis shows that the large-scale parallelism in DADO is not very effective for executing production systems. The reasons are:

(1) actions of production in the programs do not have global affects, but only affect a small number of other productions; and

(2) large-scale parallelism almost always implies that the individual processing elements are weak.

Since only a small number of productions are affected in every cycle, only a few of the large number of processing elements perform useful work. Furthermore, since the individual processing elements are weak, the performance is worse than if a small number of powerful processors are used. However, the tree-structured...
topology of the DADO architecture is not found to be a bottleneck.

Production systems are widely used in Artificial Intelligence to model learning, and to build expert systems.

Bhatt (1984) defines a sink tree as a minimum weight spanning tree having an optimum route to a common destination from every likely source. Computer networks employing routing tables, generated from sink trees, are said to use directory routing. For a complete directory routing all sink trees need to be generated. He presents an parallel algorithm to generate all sink trees. The algorithm does not have the restriction of non-negatively weighted edges and has an overall time complexity of O(Nd^3) where N is the number of nodes and d is the degree of node with highest incidence of edges.

When the 'greedy method' of augmenting nodes in non-decreasing path weight was employed, it failed to look-ahead effects of any negatively weighted edges. However, a parallel implementation resulted in partial path selections which are broadcasted and utilized to obtain look-ahead characteristics. Bhatt concludes that this not only leads to early revisions of path selections, but also yields minimal simple paths.

Quinn and Yoo (1984) describe the parallelization of
several familiar graph algorithms for the tightly-coupled MIMD model. In two cases—the farthest insertion heuristic for the traveling salesman program and Kruskal's minimum spanning tree algorithm—it was shown that an efficient parallel algorithm can be developed in a straightforward manner from the serial (sequential) algorithm. In the case of Moore-algorithms to solve the single-source shortest path problem, the queue, which works well in the serial case, is not well suited for use on a parallel computer. Alternative data structures are proposed, and speedup curves of algorithms based on these alternative data structures are presented. They conclude that the designer of data structures for parallel algorithms must carefully consider the trade-offs between the goals of reducing contention for shared data structures, limiting the number of process synchronizations, and balancing processor workloads, while keeping the amount of overhead added to the algorithm to a minimum.

Quinn and Yoo (1984) conclude that the development of an efficient parallel algorithm requires that several goals must be achieved simultaneously. The efficiency of a parallel algorithm is the ratio between the speedup and the number of processors used. Process synchronizations, serial code, and contention for shared data structures
must be limited. The workloads of the processors must be balanced. Finally, an efficient algorithm must be used. Trade-offs exist between these goals, but all of these goals must be met to at least a minimal degree if satisfactory speedup is to result.

They further generalize from their experiences with various algorithms that with partitioned algorithms it is easy to balance workloads and detect the halt condition. However, synchronization overhead limits the number of processes which can be used. After a while, processes spend more time synchronizing than doing useful work. Software pipelined algorithms also have easy-to-detect halting conditions, but incur an overhead in inter-segment coordination. Asynchronous algorithms have no synchronization costs, but detection of the halt condition is more difficult than with partitioned algorithms. In addition, it is hard to balance the workloads of the processes without adding to the complexity (or at least the constant of proportionality) of the underlying algorithm.

Kwan and Ruzzo (1984) study parallel algorithms for finding minimum spanning trees. Their main results are that in spite of obstacles of a greater need for communication, synchronization, and scheduling of processors to insure that each is assigned to a useful subproblem, the three best-known sequential minimum
spanning tree algorithms can all be efficiently parallelized to some degree. Their algorithms are edge density and processor adaptive, i.e., the worst case running time is a function of $e$, the number of edges, and $p$, the number of processors. For example, they parallelize Sollin's method to obtain an $O((e \log n)(p))$ algorithm, $p \leq e/\log n$, which is the best method known to us for concurrent read exclusive write shared memory models. They include a summary of parallel minimum spanning tree algorithms. Some results using parallel algorithms for sorting, heap operations, and the union/find problem were also given. In spite of processor assignment, communication and synchronization overheads, their parallel algorithms are efficient in that their processor-time products are not worse than their sequential counterparts. For many sparse graphs that arise in practice, these algorithms use processors very effectively to provide significant speed-up.

Carlson and Sugla (1984) present algorithms for solving linear recurrence equations of the form $x_i = a_i x_{i-1} + b_i$ on a network of parallel processors connected by the perfect shuffle. A perfect shuffle of a vector shuffles the processing elements of that vector in the same way that the cards in a deck of cards are shuffled when the deck is cut exactly in half and the halves are perfectly
interlaced. They discuss parallel algorithms for a wide class of recurrence equations and related problems that illustrate the trade-off which must be made between time and the number of processors. The algorithms are a result of illustrating that the binary tree associated with such a recurrence equation can be structured to match the given number of processors being employed and the pattern by which they are connected. By accepting inputs in a systolic fashion, the algorithms make efficient use of the resources of time and number of processors. They develop an algorithm that outputs every Pth term of the recurrence, and achieves an optimal speed up of O(n/P) when P processors are used to solve a recurrence of size n. They extend their methods to output all terms of the recurrence, and find that more processors are required, O(Plog₂P), to achieve the same time performance O(n/P). Thus, it follows that arbitrarily sized recurrences can be solved on fixed sized sets of tightly coupled processors in optimal time.

The search for solutions in a combinatorially large problem space is a major problem in artificial intelligence and operations research. Li and Wah (1984) have adapted and extended branch-and-bound algorithms for parallel processing. The computational efficiency of these algorithms depends on the allowance function, the
dominance relation, the data structure and the search strategies. Allowance is a function which specifies the allowable deviation of a suboptimal value from the exact optimal value. A dominance relation is a binary relation such that $P$ dominates $P'$ means that the subtree rooted at $P$ contains a solution node with a value no more than the minimum solution value of the subtree rooted at $P$.

Li and Wah (1984) find that anomalies owing to parallelism may happen and occur frequently when approximations or dominance tests are used. In their paper, sufficient conditions to guarantee that parallelism will not degrade the performance are developed. Necessary conditions for allowing parallelism to have a speedup greater than the number of processors is also presented. Theoretical analysis and simulations show that a best-first search is robust for parallel processing.
CHAPTER IV

RESOLUTION

Historical Background

Automatic deduction, or mechanical theorem-proving, has been a major concern of Artificial Intelligence since its earliest days. Newell and Simon (1956) discussed the Logic Theorist, a deduction system for propositional logic, at the first formal conference on AI, held at Dartmouth College in the summer of 1956. In 1960 Hao Wang (1960) at Harvard University produced the first implementation of a reasonably efficient, complete algorithm for proving theorems in propositional logic.

In 1965 an important step in the development of automatic deduction techniques was Robinson's (1965) description of a relatively simple, logically complete method for proving theorems in first-order predicate calculus. Robinson's procedure and those derived from it are usually referred to as resolution procedures.

Robinson's work had a major impact on research into commonsense reasoning and problem solving. His ideas in this area brought about a rather dramatic shift in attitudes toward automatic deduction. The resolution method seemed powerful enough to make it possible to build
a completely general problem-solver by describing problems in first-order logic and deducting solutions by a general proof procedure.

In 1969 Green (1969) carried out extensive experiments with a question-answering problem-solving system based on resolution. But the results of Green's experiments and several similar projects were disappointing. The difficulty was that, in the general case, the search space generated by the resolution method grew exponentially with the number of formulas used to describe a problem. Thus even problems of moderate complexity cannot be solved in a reasonable time. Several heuristics were proposed to deal with this issue, but they proved too weak to produce satisfactory results.

These failures resulted principally from two constraints the researchers had imposed upon themselves: They attempted to use only uniform, domain-independent proof procedures, and they tried to force all reasoning and problem-solving behavior into the framework of logical deduction. Despite the earlier disappointments, there has recently been a revival of interest in deduction-based approaches to commonsense reasoning. (Cohen and Feigenbaum 1982)
Usefulness of Resolution

The use of predicate logic is now firmly established in Artificial Intelligence as a useful way to represent declarative knowledge. Logical deduction is the process of generating conclusions that are not explicitly represented as logical statements themselves, but that are implied by the knowledge that is explicitly represented. Deductive methods are quite important in Artificial Intelligence approaches to database question-answering systems, to automatic planning, to program synthesis and verification, and to natural-language processing. (Webber and Nilsson 1981)

Resolution is both valid and complete. This means that the method only produces proofs of true theorems and that, if a theorem is true, the technique will produce a proof after a finite number of steps. (Hunt 1975)

First Order Predicate Calculus

In order to discuss resolution, first order predicate calculus must be summarized.

First order predicate calculus is a formal language in which a variety of statements can be expressed. The legitimate expressions of the predicate calculus are called well-formed formulas (wffs). The wffs are defined as follows:
The elementary components of the predicate calculus language are predicate symbols, variable symbols, function symbols and constant symbols set off by parenthesis, brackets, and commas. Predicate symbols are used to represent relations between elements in the domain.

A term is defined as follows:

1) variables and individual constants are terms
2) if \( f \) is a function symbol and \( t_1, \ldots, t_n \) are terms, then \( f(t_1, \ldots, t_n) \) is a term.

Atomic formulas are combined by the following connectives:

1) \( \neg \) not
2) \( \lor \) inclusive or
3) \( \land \) and
4) \( \Rightarrow \) implies:

1) Every atomic formula is a wff
2) If every \( A \) and \( B \) are wff's and \( y \) is a variable then \( (\neg A), (A \Rightarrow B) \) and \( ((\forall y)A) \) are wffs.

\( \forall \) is called the universal quantifier and means "for all". \( A \) is called the scope of \( \forall y \), i.e. the atomic formula enclosed in parenthesis along with \( \forall y \). The existential qualifier \( \exists y \) means "there exists".

The following conventions are used: upper-case-letter strings are used to name predicates and lower-case-letter strings are used to name functions. The context
determines the difference between predicates and constants.

An interpretation of a wff is obtained by assigning a correspondence between the elements of the formula and the predicates, functions, variables and constants in the domain considered. If the truth values of two wffs are the same regardless of their interpretation, then these wff's are called equivalent. A clause is a wff consisting of a disjunction of literals. The following equivalences are needed to convert wffs into clauses:

\[ \neg (\neg X_1) \text{ is equivalent to } X_1 \]
\[ X_1 \lor X_2 \text{ is equivalent to } \neg X_1 \to X_2 \]

DeMorgan's Laws:

\[ \neg (X_1 \land X_2) \text{ is equivalent to } \neg X_1 \lor \neg X_2 \]
\[ \neg (X_1 \lor X_2) \text{ is equivalent to } \neg X_1 \land \neg X_2 \]

Distribute Laws:

\[ X_1 \land (X_2 \lor X_3) \text{ is equivalent to } (X_1 \land X_2) \lor (X_1 \land X_3) \]
\[ X_1 \lor (X_2 \land X_3) \text{ is equivalent to } (X_1 \lor X_2) \land (X_1 \lor X_3) \]

Commutative Laws:

\[ X_1 \land X_2 \text{ is equivalent to } X_2 \land X_1 \]
\[ X_1 \lor X_2 \text{ is equivalent to } X_2 \lor X_1 \]

Associative Laws:

\[ (X_1 \land X_2) \land X_3 \text{ is equivalent to } X_1 \land (X_2 \land X_3) \]
\[ (X_1 \lor X_2) \lor X_3 \text{ is equivalent to } X_1 \lor (X_2 \lor X_3) \]

Contrapositive Law:

\[ X_1 \to X_2 \text{ is equivalent to } \neg X_2 \to \neg X_1 \]
\[ \neg (\exists x)P(x) \text{ is equivalent to } (\forall x)[\neg P(x)] \]
\[ \neg (\forall x)P(x) \text{ is equivalent to } (\exists x)[P(x)] \]
\[ (\forall x)[P(x) \land Q(x)] \text{ is equivalent to } (\forall x)P(x) \land (\forall y)Q(y) \]
$(\forall x)[P(x) \lor Q(x)]$ is equivalent to $(\forall x)P(x) \lor (\forall y)Q(y)$

$(\forall x)P(x)$ is equivalent to $(\forall y)P(y)$

$(\exists x)P(x)$ is equivalent to $(\exists y)P(y)$

The last two equivalences show that the bound variable in a quantified expression is a kind of "dummy" variable that can be arbitrarily replaced by any other variable symbol not already occurring in the expression.

(Nilsson 1980)

**Rules of Inference**

In the predicate calculus, there are rules of inference which transform wffs to new wffs. The following are some of the rules of inference:

**Modus Ponens:**

From the wffs of the form $W_1 \land W_2$ and $W_1 \rightarrow W_2$ modus ponens is the inference rule that produces $W_2$.

**Generalization:**

Generalization is the operation that produces $(\forall x)W$ from $W$.

**Universal Specification:**

Universal Specification produces the wff $W(A)$ from $(\forall x)W(x)$.

Combining modus ponens and universal specification produces $W(A)$ from the wffs $(\forall x)[W_1(x) \rightarrow W_2(x)]$ and $W_1(A)$.
In order to apply a rule of inference to a wff or to a set of wffs it is necessary to identify patterns on the wffs which enable the application of a rule of inference. Thus, it is necessary to match certain subexpressions. Finding substitutions of terms for variables to make expressions identical is called unification.

Substitutions are represented by a set of ordered pairs \( s = (t_1/v_1, t_2/v_2, \ldots, t_n/v_n) \). The pair \( t_i \) is substituted for the variable \( v_i \) throughout the set of wffs. Each occurrence of a variable must have the same term substituted for it and no variable can be replaced by a term containing the same variable. The following substitutions give four instances of \( P[x, f(y), B] \):

- \( S_1 = x/x, w/y \) results in \( P(z, f(w), B) \)
- \( S_2 = A/y \) results in \( P(x, f(A), B) \)
- \( S_3 = g(z)/x, A/y \) results in \( P(g(z), f(A), B) \)
- \( S_4 = c/x, A/y \) results in \( P(c, f(A), B) \)

Substitutions can be applied successively to an expression and they have the associative property, but in general they are not commutative.

If a substitution \( s \) is applied to every member of a set \( E \) of expressions, we denote the set of substitution instances by \( \{E_i^s\} \). A set of expressions \( \{E_i^s\} \) is
unifiable if there exists a substitution $s$ such that $E_1s = E_2s = E_3s = \ldots$. The substitution $s$ is called a unifier of $E_1$ since its use collapses the set to a singleton. For example: $S = \{A/x, B/y\}$ unifies $P[x, f(y), B], P[x, f(B), B]$, to yield $P[A, f(B), B]$. (Nilsson 1980)

Tautologies

A wff which is true for all possible interpretations is called a tautology or valid. When quantifiers occur, one cannot always compute whether or not a wff is valid. It has been proven that it is not possible to derive an algorithm that decides if a quantified expression is a tautology or not; therefore, the predicate calculus is said to be undecidable. But it has been shown that if a wff is valid, then a procedure exists for verifying the validity of the wff. This procedure applied to wffs that are not valid may never terminate.

A set of wffs is said to be satisfiable by an interpretation if, and only if, every wff in the set has the value true for this interpretation. A wff $x$ logically follows a set of wff if, and only if, every interpretation which satisfies the set of wff satisfies also $x$. For example, the wff $(\forall x)(\forall y)[A(x) \lor B(y)]$ logically follows from the set $(\forall x)(\forall y)[A(x) \lor B(y)], (\forall z)(C(z) \lor Q(A)]$. 

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Also the wff C(A) logically follows from (\(\forall x\))C(x).

Clauses

Resolution is applied to the class of wffs called clauses. A clause is a wff consisting of a disjunction of literals. The algorithm to follow can be used to convert any predicate calculus wff into a set of clauses. The algorithm is illustrated by applying the conversion process to the following wff:

\[(\forall x)(\forall y)[P(x,y) \land Q(y,z)] \land [Q(y,z) \lor S(x,y)]\]

The conversion process consists of the following steps:

1. Eliminate implication symbols. By using the substitution \(\neg x_1 \land x_2\) for \(x_1 \land x_2\) throughout the wff, all occurrences of the symbol in the wff are eliminated. In the example wff this substitution yields:

\[\neg (\forall x)(\forall y)[\neg P(x,y) \lor Q(y,z)] \land [\neg Q(y,z) \lor S(x,y)]\]

2. Reduce the scope of negation symbols. Each negation symbol, \(\neg\), must apply to only one atomic formula. This is done by repeated use of DeMorgan's laws and other equivalences mentioned earlier. Using these laws the example is changed as follows:

\[(\forall x)(\forall y)[\neg P(x,y) \lor \neg Q(y,z)] \land \neg S(x,y)\]

\[(\forall x)(\forall y)[\neg P(x,y) \lor S(x,y)]\]
3. Standardize variables. Each qualifier should have its own unique dummy variable. Standardizing the example wff yields:

\[(\forall x)(\forall y) [P(x,y) \land Q(y,z)] \lor [Q(y,z) \land \neg S(x,y)]\]

\[(\forall w)(\forall t) [\neg P(w,t) \lor S(w,t)]\]

4. Eliminate existential quantifiers. When an existential quantifier is within the scope of a universal quantifier, it is necessary to allow for the possibility that the existential variable, x, might depend on the value of the universal variable, y. If the dependency of x from y can be expressed by a function g, such that x = g(y) then g is called a Skolem function. By using the Skolem function in place of the x that exists, the existential quantifier can be completely eliminated. For example if \((\forall y)((\exists x) A(x,y))\) and x = g(y) then the \((\exists x)\) can be eliminated giving the result, \((\forall y)A(g(y), y)\).

The general rule for eliminating an existential quantifier from a wff is to replace each occurrence of its existentially quantified variable by a unique Skolem function whose arguments are those universally quantified variables that are bound by universal quantifiers whose scopes include the scope of the existential quantifier being eliminated. If the existential quantifier being eliminated is not within the scope of any universal quantifier, a Skolem function of no arguments, which is
just a unique constant is used. Thus, \((\exists x) Q(x)\) becomes \(Q(A)\) where the symbol \(A\) stands for an entity that is known to exist. Eliminating the existential quantifier in the example wff yields:

\[
(\forall y) [P(f(y), y) \land \neg Q(y, z)] \lor [Q(y, z) \land \neg S(f(y), y)]
\]

5. Obtain Prenex form. Since the variables have been standardized and there are no remaining existential quantifiers, all the universal quantifiers can be moved to the front of the wff. Thus the scope of each quantifier now includes the entirety of the wffs following it. The resulting wff is said to be in prenex form. A wff in prenex normal form consists of a string of quantifiers called a prefix followed by a quantifier-free formula called a matrix. The prenex form of the example wff is

\[
(\forall y)(\forall t) [P(f(y), y) \land \neg Q(y, z)] \lor [Q(y, z) \land \neg S(f(y), y)]
\]

6. Obtain conjunctive normal form. Any matrix can be written as the conjunction of a finite set of disjunctions of lituals (known as the conjunctive normal form) by repeatedly using the distributive rules. Obtaining the conjunctive normal form of the example wff yields

\[
(\forall y)(\forall t) [P(f(y), y) \land Q(y, z) \land \neg P(g(t), t) \lor S(g(t), t)] \land
[\neg Q(y, z) \land Q(y, z) \land \neg P(g(t), t) \lor S(g(t), t)] \land
[P(f(y), y) \land S(f(y), y) \land \neg P(g(t), t) \lor S(g(t), t)] \land
[\neg Q(y, z) \land \neg S(f(y), y) \land \neg P(g(t), t) \lor S(g(t), t)]
\]
7. Eliminate universal quantifiers. Since all of the variables in the wffs used must be bound, it is assured that all the variables remaining at this step are universally quantified. Furthermore, the order of universal quantification is unimportant, so the explicit occurrence of universal quantifiers are eliminated and it is assumed that all variables in the matrix are universally quantified. Thus the example wff becomes just a matrix in conjunctive normal form.

\[
\begin{align*}
[P(f(y),y)\lor Q(y,z)\lor \neg P(g(t),t)\lor S(g(t),t)] \\
[\neg Q(y,z)\lor Q(y,z)\lor \neg P(g(t),t)\lor S(g(t),t)] \\
[P(f(y),y)\lor \neg S(f(y),y)\lor \neg P(g(t),t)\lor S(g(t),t)] \\
[\neg Q(y,z)\lor \neg S(f(y),y)\lor \neg P(g(t),t)\lor S(g(t),t)]
\end{align*}
\]

8. Eliminate symbols. If a wff logically follows from \((x \land y)\) then it logically follows from \(x, y\). That is every interpretation which satisfies \((x \land y)\) also satisfies every interpretation of the set of wff \(\{x, y\}\) and vice versa. Thus, explicit occurances of symbols are eliminated by replacing expressions of the form \((x \land y)\) with a set of wffs \(x, y\). The result of repeated replacements is to obtain a finite set of wffs, each of which is a disjunction of literals. Any wff consisting solely of a disjunction of literals is called a clause. The example wff is transformed into the following set of clauses:

\[
\begin{align*}
P(f(y),y)\lor Q(y,z)\lor \neg P(g(t),t)\lor S(g(t),t) \\
\neg Q(y,z)\lor Q(y,z)\lor \neg P(g(t),t)\lor S(g(t),t) \\
P(f(y),y)\lor \neg S(f(y),y)\lor \neg P(g(t),t)\lor S(g(t),t) \\
\neg Q(y,z)\lor \neg S(f(y),y)\lor \neg P(g(t),t)\lor S(g(t),t)
\end{align*}
\]
9. Rename Variables. No variable symbol should appear in more than one clause. Variable symbols may be renamed by the rules presented earlier. Thus, the example now becomes:

\[ P(f(y_1), y_1) \lor Q(y_1, z_1) \lor \neg P(g(t_1), t_1) \lor S(g(t_1), t_1) \]
\[ Q(y_2, z_2) \lor Q(y_2, z_2) \lor \neg P(g(t_2), t_2) \lor S(g(t_2), t_2) \]
\[ P(f(y_3), y_3) \lor \neg S(f(y_3), y_3) \lor \neg P(g(t_3), t_3) \lor S(g(t_3), t_3) \]
\[ Q(y_4, z_4) \lor \neg S(f(y_4), y_4) \lor \neg P(g(t_4), t_4) \lor S(g(t_4), t_4) \]

( Nilsson 1980)
CHAPTER V

RESOLUTION-REFUTATION SYSTEMS

Resolution-based systems are designed to produce proofs by contradiction or refutations. The proof is usually presented in the following manner: assume that the theorem is false, and therefore that its negation is true. Then show that this assumption, taken together with the premises, leads to an impossible situation; therefore the theorem cannot be false, so it must be true. The contradiction is that, if the assumptions hold, some wff and its negation must be both true and false.

The basic resolution proof procedure consists of first replacing all the premises, and the negation of the theorem, by clauses which logically follow and are in conjunctive normal form. Then all the clauses are placed in one group. Two clauses are selected: one that contains some variable X and another that contains its negation \( \neg X \). If necessary the proper substitutions are made for the variables. A new clause is generated which contains all the ored elements of the two selected clauses except for X and \( \neg X \). The newly generated clause, called the resolvent, is added to the group of clauses and the process continues, until either a contradiction is deduced, thus proving the desired theorem, or no more new clauses
can be generated in which case the theorem cannot be proven.

The justification of the process of proof by refutation is given by the following argument. If a wff, $A$, logically follows from a set, $S$, of wffs then every interpretation satisfying $S$ also satisfies $A$. None of the interpretations will satisfy $\neg A$; therefore, no interpretation exists that can satisfy the union of $S$ and $\{\neg A\}$. A set of wffs that cannot be satisfied by any interpretation is called unsatisfiable. Thus, if $A$ logically follows from $S$, the set $S \cup \{\neg A\}$ is unsatisfiable.

If resolution is applied repeatedly to a set of unsatisfiable clauses, eventually the empty clause, NIL, will be produced. That is, if $A$ logically follows from $S$, then resolution will produce the empty clause from the clause representation of $S \cup \{\neg A\}$. (Nilsson 1980)

The following example demonstrates how resolution can be efficiently used to prove theorems.

Some patients like all doctors. No patient likes any quack. Therefore, no doctor is a quack. Denote

\begin{align*}
P(x) & : \text{x is a patient,} \\
D(x) & : \text{x is a doctor,} \\
Q(x) & : \text{x is a quack,} \\
L(x,y) & : \text{x likes y.}
\end{align*}
Then the facts and the conclusion may be symbolized as follows:

\[ F_1: (\exists x) (P(x) \land (\forall y) (D(y) \rightarrow L(x, y))) \]
\[ F_2: (\forall x) (P(x) \rightarrow (\forall y) (Q(y) \rightarrow \neg L(x, y))) \]
\[ G: (\forall x) (D(x) \rightarrow \neg Q(x)). \]

Transforming the premises into clauses, obtains

1. \( P(A) \)
2. \( \neg D(y) \lor L(A, y) \quad \text{from } F_1 \)
3. \( \neg P(x) \lor \neg Q(y) \lor \neg L(x, y) \quad \text{from } F_2 \)

where the variables have been standardized apart and where \( A \) is a Skolem constant. The negation of the conclusion to be proven, converted to a clause form, is

4. \( D(B) \)
5. \( Q(B) \quad \text{from } \neg G \)

(Since any unique constant can be used in the substitution of a given wff, \( B \) is chosen in the goal wff for clarification.)

To prove the theorem by resolution refutation involves generating resolvents from the set of clauses 1 through 5, adding these resolvents to the set, and continuing until the empty clause is produced. One proof produces the following sequence of resolvents.

6. \( L(A, B) \quad \text{a resolvent of (4) and (2)}\)
7. \( \neg Q(y) \lor \neg L(A, y) \quad \text{a resolvent of (3) and (1)}\)

substituting \( b \) for \( y \)
substituting \( a \) for \( y \)
Derivation Graph

A derivation graph is a useful control strategy structure to keep track of which resolutions have been selected and to avoid duplicated effort. A derivation graph is a way of structuring the global database at any stage of the production system process so that it indicates something about the history of rule applications. The nodes in a derivation graph are labeled by clauses. Initially, there is a node for every clause in the base set. When two clauses, $c_i$ and $c_j$, produce a resolvent $r_{ij}$, a new node is created, labeled $r_{ij}$ with edges linking it to both $c_i$ and $c_j$ nodes. $c_i$ and $c_j$ are considered the parents of $r_{ij}$ and $r_{ij}$ is considered a descendant of $c_i$ and $c_j$. This terminology deviates from the usual tree terminology since it is a tree upside down.

A refutation tree represents a resolution refutation having a root node labeled by Nil. Figure 1 shows a refutation tree for the example discussed previously.

Strategies

Although resolution is complete, it can be extremely
Figure 1. A resolution refutation tree
time-consuming. The control strategy searches for a refutation by growing a derivation graph until a tree is produced with a root node labeled by the empty clause, NIL. A control strategy for a refutation system is said to be complete if its use results in a contradiction wherever one exists.

**Breadth-first Strategy**

In the breadth-first strategy, all of the first-level resolvents are computed first, then the second-level resolvents, and so on. The breadth-first strategy is complete but it is grossly inefficient because the number of resolvents explodes exponentially and more paths are checked than necessary. Figure 2 shows the refutation graph produced by a breadth-first strategy for the example problem of the last section.

**Set-of-support Strategy**

When at least one parent of each resolvent is chosen from the negation of the theorem or from the set of clauses that are derived from it, a set-of-support strategy is being used. This strategy clearly restricts the number of clauses that can be resolved at any given time. It is usually more efficient than breadth-first search since less combinations of clauses are tested for a
Figure 2. Breadth-first strategy
resolvent resulting in fewer paths in the set-of-support strategy. Figure 3 shows the refutation graph for set-of-support.

Unit-preference Strategy

The unit-preference strategy is a modification of the set-of-support strategy in which, instead of filling out each level in a breadth-first fashion, it tries to select a single-literal clause (called a unit) to be a parent in a resolution. Every time units are used in resolution, the resolvents have fewer literals than does their parent. This process helps to focus the search toward producing the empty clause and, thus, typically increases efficiency. Figure 4 shows the refutation graph for unit-preference.

Linear-input Form Strategy

This strategy involves choosing resolvents so that each resolvent has at least one parent belonging to the base set, the set of original clauses. The first level resolvents are identical to those of the breadth-first strategy but at subsequent levels, the linear input form strategy does reduce the number of clauses produced. Therefore, it is more efficient than the previous strategies, but it is not complete, that is there are
cases in which it will not find a contradiction even if one exists. This strategy is often used because of its simplicity and efficiency. Figure 5 shows the refutation graph for linear-input form.

Ancestry-filtered Form Strategy

An ancestry-filtered form refutation is one in which each resolvent has a parent that is either in the base set or that is an ancestor of the other parent. Thus, ancestry-filtered form is very much like linear-input form, but the control strategy is complete. This strategy is still efficient as the number of clauses is restricted. However, it is not as simple as the linear-input form since the algorithm must keep track of all ancestors. Figure 6 shows the refutation graph for ancestry-filtered form. (Nilsson 1980)
Figure 5. Linear-input form strategy
Figure 6. Ancestry-filtered form strategy
CHAPTER VI

COMPLEXITY OF PARALLEL RESOLUTION ALGORITHM

On a uniprocessor environment, resolution can be very time consuming. In this chapter resolution algorithms for parallel processing are explored.

The algorithms are presented at the conceptual level and in the analysis of these algorithms the concentration is upon the comparison of complexity.

The following assumptions are made:

1) There is no limitation on the number of processors available.

2) The processors are interconnected in some way so messages can be exchanged between them.

3) Each processor has a local workspace.

4) There is no memory or data alignment time penalty.

The data for consideration consists of a set of clauses in the normal form. These clauses consist of a given set of axioms, and the denial of a theorem which is to be proven from these axioms (remember that the program finds proofs and not theorems). All variables that occur in the examples are treated as universally quantified; all existentially quantified variables have been replaced by constants or functions.
Breadth-first Strategy

As stated previously, in the breadth-first strategy all of the first level resolvents are computed first, then the second-level resolvents and so on. This algorithm is designed for synchronous multiprocessor systems.

In this parallel processing scheme all of the first level resolvents are processed simultaneously. The input to each processor is two clauses. There are as many processors available as necessary to have unique combinations for all the clauses. At any stage if the output from some processor is nil, a message is sent to the other processors and the algorithm halts, having proven the theorem true.

If there are no nil clauses then the second level resolvents are processed concurrently. The input at this level consists of all the new resolvents combined two at a time with new and old clauses, excluding the immediate parent clauses of a resolvent. If there are no nil clauses the process proceeds to the next level. Figure 7 shows how the example presented in the last chapter would be processed in parallel.

A variation of this algorithm would be checking at each stage (if there were no nil clauses) for duplicate clauses. This would have the effect of increasing the
Figure 7: Breadth-first on a parallel processor
time by the number of stages minus one, but it would decrease the number of needed processors.

The uniqueness of the clauses in the algorithm could be obtained, in principal, in one unit of time using \( r(r-1) \) processors where \( r \) is the number of clauses in the data base. Each clause \( a_i \ (i = 1 \text{ to } r) \) is compared with every other clause all in parallel. From the results of this comparison if \( a_i = a_j \) for \( i > j \) remove \( a_i \) from the data base. However, the number of literals in a clause will affect the time for comparison. Also the literals should be stored alphabetically and merged accordingly with each resolvent. This assumes that clauses can be compared in one unit of time regardless of the number of literals in a clause. This will increase the time but it will reduce the number of needed processors.

Now suppose there are a limited number of processors \( k \), such that \( k > 1 \). Using the breadth-first strategy the following algorithm will be used for analysis. The processors are ordered from 1 through \( k \). All the original \( n \) clauses are contained in a general data-base and as new resolvents are found they will be added to the data base. The first \( k \) clauses are given as input to the \( k \) processors. The clauses are counted from \( i = 1 \) to \( n \). Each processor compares all the combinations of clauses in the data base. That is, if the \( i^{th} \) clause is input to a
processor then that processor would compare the $i^{th}$ clause with $(i + 1)^{th}$ clause to the $n^{th}$ clause. Figure 8 shows how the example presented is processed with $k$ equal to 2. Thus the clause $P(A)$ is input to processor 1 and the clause $\lnot D(y) \lor L(A,y)$ is input to processor 2 and these clauses are compared.

After the first $k$ clauses have been processed the remainder of the clauses are assigned as input as a processor becomes available. Processor 2 will finish the necessary comparisons one cycle before processor 1. When processor 2 completes processing clause $\lnot D(y) \lor L(A,y)$, it starts to process the third clause which compares with the clauses that the third clause has not been compared with. Similarly when processor 1 has finished processing clause 1 it will start to process the fourth clause $D(b)$ which in this case only has to be compared to $Q(b)$. If at any point a resolvent yields nil, a message is sent to the other processors and the algorithm will halt, having proven the theorem. If a nil resolvent is not found the system proceeds to resolve the next level of resolvents. This time only the new resolvents would be initial input to the processors as only new combinations of resolvents will be compared. They are compared with the $(i + n)^{th}$ to $(n + j)^{th}$ clause where $j$ is the number of new clauses in the database. This $i^{th}$ clause would also be compared with
Figure 8. Breadth-first on a system with two processors
the first to the nth clause with the exception of its parent clauses. This process would continue until a nil clause is found or until the nth level of resolvents is reached. Figure 8 shows how the first level resolvents in the given example are compared with each other and with the original data base.

Step 1 starts with \( n = n_0 \) clauses. To get the number of first level resolvents it is necessary to compare all the combinations of the original clauses taken two at a time. Hence there are \( \binom{n}{2} = N_1 \) clauses generated at level one. These might not all be unique and in some cases no resolvents may be derivable. In deriving complexity, however, it is assumed that all resolvents are present and unique.

At step 2 each of the new clauses is compared to all old clauses, except parents \( [N_1, (N_0 - 2)] \) as well as all other new clauses \( [(N_1)] \). Hence there are \( N_1 = N_0 (N-2) + \binom{N_1}{2} \).

At step 3 the new clauses generated in step 2 are compared with all original clauses \( (N_2, N_0) \) and with resolvents of level 1, except parents, \( [N_2(N-2)] \) and with all other level 2 resolvents \( (N_2) \). Hence there are \( N_3 = N_2 n_0 + N_2(N_1-2) + \binom{N_2}{2} \) resolvents.

At the mth step the new resolvents are again compared to all original as well as all resolvents from levels 1
through \( m - 3 \sum_{i=0}^{m-1} N_i \) plus all resolvents at the \( m - 2 \) level, except parents; \( (N_{m-1}^{N_{m-2}-2}) \) and finally with all other \( m - 1 \) level resolvents \( \binom{N}{2^{m-1}} \).

Hence there are:

\[
N_n = N_{m-1}^N N_1 \cdots N_{m-1}^{N_1} \binom{N_{m-1}}{2^{m-2}} + \binom{N}{2^{m-1}}.
\]

In the worst possible case the order of new resolvents at the \( m \)th step is \( O(m^{m-1}) \). If there are \( k \) processors then the complexity is reduced by \( k \). If there is an unlimited amount of processors these algorithms will halt in linear time in a maximum of \( n \) steps either because a nil clause would have been found or because all possible combinations of the literals will have been found. However the number of processors required is \( O(n^{m^n}) \).

By using the associative law of disjunction, the number of new resolvents at step \( m \) becomes \( \binom{n}{m+1} \) when the algorithm checks for uniqueness. Thus, it follows that after the first step fewer resolvents are generated than in the previous algorithm.

The result of the complexity of these algorithms is shown in Tables 1 and 2. Both of these algorithms require exponential time. Thus the breadth-first strategy is reasonable for only a small number of clauses. The breadth-first complexity is \( O(n^{m^n}/k) \) where \( n \) represents the number of clauses, \( m \) represents the number of steps used and \( k \) represents the number of processors available. When
Table 1

Breadth-first Complexity

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of clauses generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$N_0 = n$</td>
</tr>
<tr>
<td>1</td>
<td>$N_1 = \binom{n}{0}$</td>
</tr>
<tr>
<td>2</td>
<td>$N_2 = N_1(N_0 - 2) + \binom{N_1}{N_2}$</td>
</tr>
<tr>
<td>3</td>
<td>$N_3 = N_2N_0 - N_2(N_1 - 2) + \binom{N_2}{2}$</td>
</tr>
<tr>
<td>4</td>
<td>$N_4 = N_3N_0 + N_3N_1 + N_3(N_2 - 2) + \binom{N_3}{2}$</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
<tr>
<td>m</td>
<td>$N_m = \sum_{i=0}^{m-3} N_{m-1}N_i + N_{m-1}(N_{m-2} - 2) + \binom{N_{m-1}}{2}$</td>
</tr>
</tbody>
</table>

$\approx \binom{n^{2m-1}}{m-1}$

Complexity with $k$ processors $\approx O(n^{2m-1}/k)$

$n$ - number of original clauses
$m$ - number of steps used
Table 2
Breadth-first Complexity with all Clauses Unique

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of new clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n</td>
</tr>
<tr>
<td>1</td>
<td>( \binom{n}{2} )</td>
</tr>
<tr>
<td>2</td>
<td>( \binom{n}{3} )</td>
</tr>
<tr>
<td>3</td>
<td>( \binom{n}{4} )</td>
</tr>
<tr>
<td>...</td>
<td>( \binom{n}{n} )</td>
</tr>
</tbody>
</table>

Complexity = \( O\left(\binom{n}{2}\right) = O\left(2^{n-1}\right) \)

Complexity with \( k \) processors = \( O\left(2^{n-1}/k\right) \)

\( n \) -- number of steps
\( k \) -- number of processors
duplicate clauses are eliminated from the database the complexity improves and is on the order of $O(2^{n-1})$.

By observing the Figures 7, 8 and 9 (which show how the example would be processed in parallel with an unlimited amount of processors, with two processors and with one processor respectively) it is apparent that there will be many combinations of clauses for which no resolvents can be found. Thus, most resolution problems will not actually have this worst case time for solving the problem.

Set-of-support

An algorithm which takes advantage of the set-of-support works as follows: in order to find the first level resolvents, the number of processors needed is equal to the number of clauses resulting from the contradiction of the goal. The clauses in each processor are compared to all the clauses in the database until a resolvent is found. The new resolvent would then be used as a parent and compared with the other clauses. If a nil clause is found, that information is communicated to the other processors and the algorithm then halts.

When there is only one clause in the negation of the theorem then the algorithm is essentially the same as the algorithm for a uniprocessor. Figure 10 shows how the
Figure 9. Breadth-first on a uniprocessor
Figure 10. Set-of-support on a parallel processor
given example would be performed in a parallel processing environment and figure 11 shows how this would be done on a uniprocessor.

The set-of-support strategy generally results in slower growth of the clause set since the growth at each time interval is limited to the number of clauses in the goal. This moderates the combinatorial explosion as seen in the breadth-first strategy. Time for this strategy is usually greater than for the breadth-first strategy depending upon the number of processors used.

The order of complexity is still exponential since the worst case must be considered for complexity. In general it is obvious that this method is more practical than breadth-first strategy as it needs fewer processors.

Step 1 uses the clauses generated from the negation of the goal (k) and compares them with all of the clauses of the premises and with each other. This gives

\[ N = k(n-k) + \binom{k}{2} \]

At step 2 the new resolvents are compared with all original clauses \(N_1(N_0 - 2)\), and with each other \(\binom{N_1}{2}\). Hence \(N_2 = N_1(N_0 - 2) + \binom{N_1}{2}\).

At step 3 the new resolvents are compared with all original clauses \(N_2N_0\) and with level 1 resolvents, except parents, \(N_1(N_0 - 2)\) and with each other \(\binom{N_2}{2}\). Hence \(N_3 = N_2N_0 + N_2(N_1 - 2) + \binom{N_2}{2}\).
Figure 11. Set-of-support on a uniprocessor
At the \( m \)th step the new resolvents are compared with all original clauses \((N_{m-1} N_0)\) and with levels 1 through level \( m - 2 \), except for immediate parents, \( (\sum_{i=1}^{m-2} N_{m-1} N_{i}) + N_{m-1}(N_{m-2} - 2) \) and with each other. Hence

\[
N_m = N_{m-1} N_0 + \sum_{i=1}^{m-2} N_{m-1} N_i + N_{m-1}(N_{m-2} - 2).
\]

Hence there are:

\[
N_m = N_{m-1} N_0 + N_{m-1} N_1 + \ldots + N_{m-1} N_{m-1} + N_{m-1}(N_{m-2}) + (N_{m-1}).
\]

In the worst possible case the order of new resolvents at the \( m \) step is \( O((nk(n-k)^2)^{m-2}) \). If there are \( k \) processors then the complexity is reduced by \( k \).

However, if the number of processors available is equal to all the combinations at a given level, that level of resolvent can be done in one time element with breadth-first strategy. Even if the maximum number of processors available is equal to the number of original clauses, the time for the breadth-first strategy would generally be faster than for the set-of-support as more combinations are compared in one time interval. As shown in Table 3, the order of complexity for the set-of-support algorithm is \( (nk(n-k)\sqrt{m-2}/k) \). If the number of processors is less than the square root of the number of original clauses (including the goal clauses) or when the premises are less than the square root of the original base clauses than the set-of-support is faster by a power of two. The worst case for the set-of-support is when \( k \) approaches \( n/2 \).
Table 3

Set-of-support Complexity

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of new clauses generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$N_0 = n$</td>
</tr>
<tr>
<td>1</td>
<td>$N_1 = k(n - k){k \choose 2}$</td>
</tr>
<tr>
<td>2</td>
<td>$N_2 = N_1(N_0 - 2) + {N_1 \choose 2}$</td>
</tr>
<tr>
<td>3</td>
<td>$N_3 = N_2N_0 + N_2(N_1 - 2) + {N_2 \choose 2}$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>$N_m = N_{m-1}N_0 + \sum_{i=1}^{m-3} N_{m-1}N_i + N_{m-1}(N_{m-2}^2) + {N_{m-1} \choose 2}$</td>
</tr>
<tr>
<td></td>
<td>$N_m = N_{m-1}N_0 + N_{m-1}N_1 + \ldots N_{m-1}N_{m-3} + N_{m-1}(N_{m-2}^2) + {N_{m-1} \choose 2}$</td>
</tr>
<tr>
<td></td>
<td>$\approx (nk(n - k)^{2m-2}$</td>
</tr>
</tbody>
</table>

Complexity with $k$ processors $\approx O\left(\frac{nk(n - k)^{2m-2}}{k}\right)$

$n$ - number of original clauses
$m$ - number of steps
$k$ - number of processors and number of clauses in goal
Unit-preference Strategy

To handle this strategy all the clauses in the data base are weighted according to the number of literals in a clause. Each literal is given a weight of one point so the weight of a clause is equal to the number of literals that clause contains. The processors will search for the unit clauses (which have a weight of one). This can be done in one unit of time if there are \( r \) clauses and \( r \) processors. The processors that find a unit clause then check for resolvents. All the new resolvents are given a weight and these resolvents are checked for unit clauses. If there are no unit clauses the processors check for the next lowest weight. But, after each stage, the new resolvents are checked for unity. If a unit clause is not found the clauses with the least weight are used. In this method there would be many idle processors, but if this is a time sharing environment, they are free to work on other jobs. Thus, this method is practical for a time sharing multiprocessor system. Figures 12 and 13 show how the strategy works on parallel and uniprocessor systems, respectively.

Table 4 shows the complexity of this algorithm to be \( O(n^{\log_2 l}/k) \) which is the same as the complexity for the breadth-first algorithm presented in Table 1. Since the
Figure 12. Unit-preference on a parallel processor
Figure 13. Unit-preference on a uniprocessor
Table 4

Unit-preference Complexity

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of clauses generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( N_0 = n )</td>
</tr>
<tr>
<td>1</td>
<td>( N_1 = \binom{N_0}{2} )</td>
</tr>
<tr>
<td>2</td>
<td>( N_2 = N_1(N_0 - 2) + \binom{N_1}{N_2} )</td>
</tr>
<tr>
<td>3</td>
<td>( N_3 = N_2N_0 + N_2(N_1 - 2) + \binom{N_2}{2} )</td>
</tr>
<tr>
<td>4</td>
<td>( N_4 = N_3N_0 + N_3N_1 + N_3(N_2 - 2) \binom{N_2}{2} )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>( \cdots )</td>
</tr>
<tr>
<td></td>
<td>( m ) ( N_m = \sum_{i=0}^{m-3} N_{m-1}N_i + N_{m-1}(N_{m-2} - 2) + \binom{N_{m-1}}{2^{m-1}} )</td>
</tr>
<tr>
<td></td>
<td>( = N_{m-1}N_1 + N_{m-1}N_2 + \cdots N_{m-1}N_{m-3} + N_{m-1}(N_{m-2} - 2) + \binom{N_{m-1}}{2^{m-1}} )</td>
</tr>
<tr>
<td></td>
<td>( \approx (n^{2^{m-1}}) )</td>
</tr>
</tbody>
</table>

Complexity with \( k \) processors \( \approx O(n^{2^{m-1}}/k) \)

\( n \) - number of original clauses
\( m \) - number of steps used
order of complexity deals with the worst case it is necessary to consider the situation where most of the clauses are all of the same weight and not unit clauses. However, if all the original clauses had only one literal then a nil clause would be found at the first level of resolvents or the algorithm would halt as there would be no new clauses generated. Simulation studies should show this algorithm to be much quicker than breadth-first as unit clauses are generally present and will also be generated. These will collapse the clauses and the algorithm might have an average time approaching that of the linear-input form strategy.

Linear-input Form Strategy

This algorithm starts off similarly to a breadth-first algorithm. There are as many processors as there are clauses. At the first level all the combinations of clauses are checked for resolvents. If a nil clause is not found then the original clauses are checked with the new resolvents. This is continued until there are no new resolvents or until a nil clause is found. Since this strategy is not complete, after n stages the clauses in the database are compared for uniqueness. After that, if no unique clauses are being generated, the algorithm halts. Since it becomes necessary to check for uniqueness
of clauses the time factor is increased as in the breadth-first algorithm. Figure 14 shows how this algorithm would work in parallel and Figure 15 shows the example on a uniprocessor.

Table 5 shows the complexity for this algorithm excluding the time to check for uniqueness. This has the best complexity \((O(n^{m+1})/k)\) of all the algorithms presented so far. This is because the new resolvents can only be compared to the initial clauses.

Ancestry-filtered Form

This is a very impractical strategy for a parallel processing environment as it involves too much communication between the processors in order to keep track of their predecessors. Also, the limitations that are placed on the comparisons cause many processors to be idle.

This form is best suited to a time sharing parallel processing system so that the processors not in use could be used for other jobs. There are as many processors as original clauses. All the combinations of clauses are compared. If a nil clause did not exist all the original clauses are compared with the new resolvents. This is repeated to get the second level of resolvents. After that the new resolvents are compared with the original
Table 5

Linear-input Form

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of new clauses generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$N_0 = n$</td>
</tr>
<tr>
<td>1</td>
<td>$N_1 = \binom{n}{2}$</td>
</tr>
<tr>
<td>2</td>
<td>$N_2 = N_1(N_0 - 2)$</td>
</tr>
<tr>
<td>3</td>
<td>$N_3 = N_2N_0$</td>
</tr>
<tr>
<td>\vdots</td>
<td>\vdots</td>
</tr>
<tr>
<td>m</td>
<td>$N_m = N_{m-1}N_0$</td>
</tr>
<tr>
<td></td>
<td>$= \binom{n}{2} + n\binom{n}{2} + n^2\binom{n}{2} + \ldots + n^{m-1}\binom{n}{2}$</td>
</tr>
<tr>
<td></td>
<td>$= \frac{(n^{m-1})(n - 1)(n - 2)}{2}$</td>
</tr>
<tr>
<td></td>
<td>$\approx O(n^{m+1})$</td>
</tr>
</tbody>
</table>

Complexity with k processors $O(n^{m+1}/k)$

- n - number of original clauses
- m - number of steps
- k - number of processors
clauses and with predecessors of the new resolvent. That process is repeated until a nil clause is found or until no new resolvents can be generated. Figures 16 and 17 show how the strategy will work on parallel and uniprocessor systems respectively.

Steps 1, 2 and 3 are the same for the ancestry-filtered-form as for the linear input form. Step 4 deviates since the new resolvents must also be compared to their ancestors found \(2N^3\) in the first level of resolvents along with the original clauses \(N^0\). Hence \(N = N^N + 2N^3\).

At step 5 the new resolvents are compared with the original clauses \((N_4 N_0)\) and with its ancestors at level one \((2^2 N^4)\) and level two \((2N^4)\). Hence \(N_5 = N_4 N_0 + 2N_4 + 4N_4\).

At step \(m\) the new resolvents are compared with the original clauses \((N_m N_0)\) and with its ancestors from level one through level \(m-3\) \(\left(\sum_{i=1}^{m-2} N_{m-1}^i\right)\). Hence \(N_m = N_{m-1} 0^i + \left(\sum_{i=1}^{m-3} N_{m-1}^i\right)^i\).

Table 6 shows the complexity for the ancestry-filtered strategy to be \(O((n^m /k + N^2)/k)\). Thus linear input form has the best complexity.
Figure 17. Ancestry-filtered form on a uniprocessor
### Table 6

Ancestry Complexity

<table>
<thead>
<tr>
<th>Step</th>
<th>Number of clauses generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( N_0 = n )</td>
</tr>
<tr>
<td>1</td>
<td>( N_1 = \binom{N_0}{2} )</td>
</tr>
<tr>
<td>2</td>
<td>( N_2 = N_1(N_0 - 2) )</td>
</tr>
<tr>
<td>3</td>
<td>( N_3 = N_2N_0 )</td>
</tr>
<tr>
<td>4</td>
<td>( N_4 = N_3N_0 + 2N_3 )</td>
</tr>
<tr>
<td>5</td>
<td>( N_5 = N_4N_0 + 2N_4 + (2)^2N_4 )</td>
</tr>
<tr>
<td>( m )</td>
<td>( N_m = N_{m-1}N_0 + \sum_{i=1}^{m-3} 2^i N_{m-1} )</td>
</tr>
<tr>
<td>( m+1 )</td>
<td>( \approx n^{m+1} + 2^m n^m )</td>
</tr>
</tbody>
</table>

Complexity with \( k \) processors \( \mathcal{O}\left(\frac{n^{m+1} + 2^m n^m}{k}\right) \)

- \( n \) - number of original clauses
- \( m \) - number of steps
- \( k \) - number of processors
The previous discussion of the time complexity of the algorithms proposed in this paper demonstrates that they have an exponential order of complexity. (Algorithms with this explosive complexity are referred to as NP-complete problems.) Resolution is an NP-complete problem. Parallel Processing allows more clauses to be processed in a reasonable period of time, but resolution-refutation systems remain NP-complete problems.

Although, with an unlimited number of processors, a resolution-refutation system can be performed in linear time, the NP-complete factor is transferred to the number of processors, as shown in the complexity analysis of the breadth-first strategy.

All methods presented have exponential order of complexity. Breadth-first and set-of-support are by far the worst and linear input form is the best. Complexity analysis is based on worst case behavior. Practical applications and empirical testing would show greater improvements using strategies other than breadth-first.

Not taken into consideration in the analysis of the complexity is the time for unification. The substitution of variables affects the time, regardless of the strategy used.
Other areas not addressed in this analysis include the problems of memory contention, memory space and actual control.

At present, very little is actually being tested in the area of resolution for parallel processing. The Artificial Intelligence community, in general, is not implementing algorithms for parallel processing because the availability of parallel processors is limited and is not readily available to computer scientists interested in Artificial Intelligence. In the next few years more work will be done in pattern recognition, vision and image processing on parallel processing systems.

Research in the area of resolution-refutation is mainly concerned with the types of problems which can be solved in a reasonable length of time and which class of problems is best for a particular kind of strategy. Looking at the complexity of any of the resolution strategies presented, it seems clear that classifying the types of problems and using heuristics that include a weighting scheme (such as after a clause has so many literals it should be eliminated) is a more practical approach.

Central to improvements in the future will be a "formula database machine" which performs at tremendous
speeds the substitution operations. While the algorithms presented in this paper use MIMD systems with globally shared memory as the best environment it seems reasonable that a formula database machine will feature many more loosely-coupled processors. Machines with up to 1,000 processors should exist within the next few years.

Obviously, there are a number of areas where further research needs to be done. Algorithms for parallel processing are in their infancy in general and in particular little has been done in the area of automatic reasoning. Constructing algorithms to convert WFF into clauses in parallel offers potential for extensions.

Empirical evaluations of the methods discussed might clarify which method is best for a particular class of problems. Complexity analysis is based on the worst case behavior. It is useful to know what the performance is on the average. This might be done with empirical testing on various types of problems suitable for resolution. Several parameters should be varied including the number of processors, the number of clauses and the number of literals per clause.

Another direction for research is in looking at applying the theorem proving algorithm to question-answering. It is obvious that given a question to answer, a question-answering system should not conduct an
exhaustive search among all the facts in order to answer the question. It should only use those relevant to the question. Data organization is critical to the success of a question-answering system and how best to use parallelism to gain efficiency.

The use of automated reasoning programs employs an unambiguous and exacting notation for representing information, precise inference rules for drawing conclusions, and carefully delineated strategies to control those inference rules. Unfortunately, precise rules do not as yet exist for using automated reasoning programs. For every well thought out suggestion of how to use an automated reasoning program an exception has been found. Guidelines for uniprocessor resolution-refutation methods have been developed based upon thousands of experiments with various automated reasoning programs. (Wos, Overbeck, Lusk, and Boyle 1984).

The art of translating natural language into first order predicate calculus remains one of the major problems of resolution-refutation systems. The difficulty rests in characterizing the concepts that are present in a given problem. Another related situation is with the representation of a problem, a reasoning program may solve it in a few seconds but given another, a few minutes might be required, or even hours, or the choice of
representation might even result in failure to reach any solution. Thus the choice of representation can play an essential role in its chance of succeeding.

In conclusion, this research contributes to the body of researched focused on automated reasoning programs. Studies have investigated many different strategies for resolution. The concern of resolution in a parallel processing environment, as focused upon here, is certainly a significant aspect of this problem for the next few years.
BIBLIOGRAPHY


