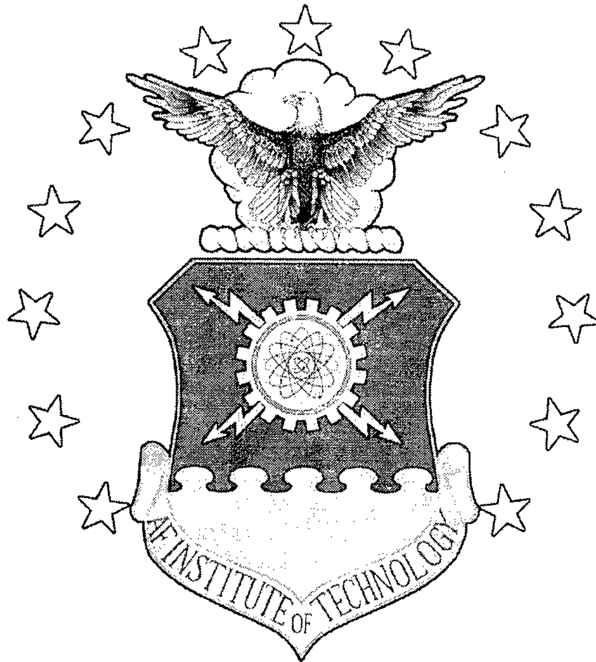


MISSION ROUTE PLANNING
WITH
MULTIPLE AIRCRAFT & TARGETS
USING
PARALLEL A* ALGORITHM

THESIS

Ergin Sezer, 1st Lt, TUAF

AFIT/GCE/ENG/00M-04

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

20000815 158

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

DTIC QUALITY INSPECTED 4

AFIT/GCE/ENG/00M-04

MISSION ROUTE PLANNING
WITH
MULTIPLE AIRCRAFT & TARGETS
USING
PARALLEL A* ALGORITHM

THESIS

Presented to the Faculty of the School of Engineering and Management

Of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of Master of Science in Computer Engineering

Ergin Sezer, B.S.

1st Lt, TUAF

21 March 2000

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCE/ENG/00M-04

MISSION ROUTE PLANNING

WITH

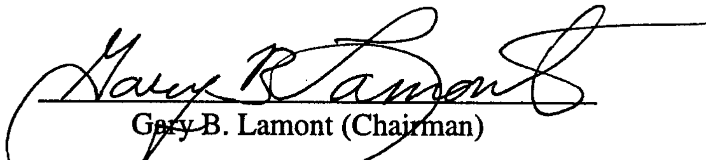
MULTIPLE AIRCRAFT & TARGETS

USING

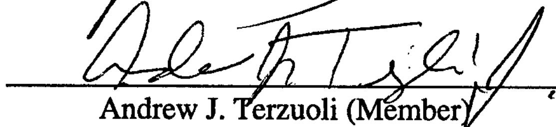
PARALLEL A* ALGORITHM

Ergin Sezer, B.S.
1st Lt, TUAF

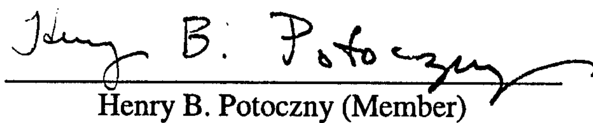
Approved:



Gary B. Lamont (Chairman) 9 Mar 00 date



Andrew J. Terzuoli (Member) 9 date



Henry B. Potoczny (Member) 9 MARCH, 2000 date

Acknowledgements

I would like to thank the members of my thesis committee, Dr. Gary B. Lamont, Dr. Andrew Terzuoli, and Dr. Henry Potoczny for their guide and assistance. They were always willing to answer all my questions with patience. Had they not encouraged me towards greater achievements I would not be able to build my scientific career as good as it is now. I am especially indebted to Dr. Gary B. Lamont, my thesis advisor, for his help until late nights during both weekdays and weekends.

I also would like to thank my other classmates for being there all together all the time. We sure burnt lots of midnight oil in labs together. I also thank my friends here for their friendship through which I could concentrate much better. I am thankful to them for sharing my happiness and sadness. I am thankful to my cousins and friends back home for constantly keeping in touch.

Most of all I owe a great deal of thanks to my family who has supported me throughout this experience at AFIT. They kept me in their minds the whole time. Thank you all.

Ergin SEZER

<u>*Table of Contents*</u>

ACKNOWLEDGEMENTS ..... **II**

TABLE OF CONTENTS ..... **III**

LIST OF TABLES ..... **VII**

ABSTRACT ..... **VIII**

1. INTRODUCTION **1-1**

1.1. BACKGROUND 1-1
1.2. PROBLEM STATEMENT 1-2
1.3. SCOPE 1-4
1.4. APPROACH 1-5
1.5. THESIS OVERVIEW 1-7

2. LITERATURE REVIEW OF MISSION ROUTING PROBLEM **2-1**

2.1. INTRODUCTION 2-1
2.2. PARALLEL PROCESSING 2-1
2.2.1. Communication costs in static interconnection networks 2-2
2.2.2. Performance Metrics for parallel Systems 2-4
2.2.3. Sources of Parallel Overhead 2-6
2.3. SEARCH TECHNIQUES 2-7
2.3.1. Basic graph searching notation 2-7
2.3.2. Approaches to Combinatorial Problems 2-9
*2.3.3. A Specialized Best-First Algorithm, A** 2-11
2.3.4. A Algorithm and its Path Planning Implementations* 2-14
2.4. MRP ALGORITHM IN DETAIL 2-16
2.4.1. Time Complexity of Bicriteria MRP Problem 2-18
2.4.2. Bicriteria MRP Implementation 2-22
2.5. MESSAGE PASSING INTERFACE 2-23
2.6. VISUALIZATION TECHNIQUES 2-24
2.7. SUMMARY 2-27

3. REQUIREMENTS AND HIGH LEVEL DESIGN OF MULTI TARGET MRP **3-1**

3.1. INTRODUCTION 3-1
3.2. MISSION ROUTING PROBLEM 3-1
3.2.1. Mission Parameters. 3-2
3.2.2. Representation of the world. 3-2
3.3. HIGH LEVEL DESIGN 3-9
3.3.1. Pseudocode 3-11
3.3.2. Heuristic 3-13
3.3.3. Parallel A Algorithm Considerations in MRP* 3-14
3.4. TESTING AND DESIGN OF EXPERIMENTS 3-18
3.5. SUMMARY 3-20

4. LOW LEVEL DESIGN AND IMPLEMENTATION OF MRP 4-1
4.1. INTRODUCTION 4-1
4.2. LOW LEVEL DESIGN 4-1
4.2.1. Parallel Architectures 4-4
4.2.2. The program Design 4-4
4.3. DESIGN ISSUES 4-9
4.3.1. Data Structures 4-9
4.3.2. Input Data Files 4-12
4.4. IMPLEMENTATION DETAILS 4-14
4.4.1. MPI and its Suitability 4-14
4.4.2. Multiple Aircraft against Multiple Target Case 4-17
4.5. SUMMARY 4-20
5. EXPERIMENTAL TESTING, RESULTS AND ANALYSIS 5-1
5.1. INTRODUCTION 5-1
5.2. DESIGN OF EXPERIMENTS 5-1
5.3. METRICS 5-2
5.4. INPUT DATA 5-5
5.5. EXPERIMENTAL TESTING 5-7
5.6. ANALYSIS OF PROGRAM EXECUTION 5-15
5.7. SUMMARY 5-19
6. CONCLUSION AND RECOMMENDATIONS 6-1
6.1. CONCLUSION 6-1
6.2. RECOMMENDATIONS FOR FUTURE RESEARCH 6-4
6.3. SUMMARY 6-7
7. APPENDIX A - PARALLEL PROCESSING 7-1
7.1. INTRODUCTION 7-1
7.2. PARALLEL PROCESSING 7-1
7.2.1. Models of Parallel Computers 7-2
7.2.2. Flynn's Taxonomy 7-2
7.2.3. An Idealized Parallel Computer Model Parallel Random Access Machine (PRAM) 7-3
7.2.4. Dynamic Interconnection Networks 7-3
7.2.5. Static Interconnection Networks 7-5
7.2.6. Evaluating Static Interconnections Networks 7-8
8. APPENDIX B - SEARCH TECHNIQUES 8-1
8.1. INTRODUCTION 8-1
8.2. SEARCH TECHNIQUES 8-1
8.2.1. Selecting a representation for the Search 8-1
8.2.2. Split and prune method 8-2
8.2.3. Exhaustive Graph Searching 8-3
8.2.4. State-space graph searching 8-3
8.2.5. Problem reduction representation and AND/OR Graph Searching 8-4
8.2.6. Stochastic methods and probabilistic algorithms 8-6

8.2.7. Deterministic Search Methods 8-9
8.2.8. Analysis of Heuristic Search Strategies 8-15
*8.2.8.1. The importance of Heuristic Functions Used in A** 8-17
*8.2.8.2. Properties of f** 8-17
8.2.8.3. A Search Algorithm* 8-19
8.2.8.4. A's convergence* 8-19
8.2.8.5. A's Admissibility-A guarantee for an optimal solution* 8-20
8.2.8.6. A's Complexity* 8-21

9. APPENDIX C - PORTING FROM NX TO MPI **9-1**

9.1. INTRODUCTION 9-1
9.2. PORTING OF INITIALIZATION AND SIMPLE SEND & RECEIVE INSTRUCTIONS 9-1
9.3. TRANSLATION OF IPROBE 9-6
9.4. TRANSLATION OF CSEND AS A BROADCAST MESSAGE 9-7
9.5. TRANSLATION OF REDUCTIONS 9-8
9.6. TRANSLATION OF TIME-RECORDING INSTRUCTIONS 9-9
9.7. CREATING SEPARATE COMMUNICATION GROUPS 9-9
9.8. MANIPULATION OF COMMUNICATION GROUPS IN THE MRP 9-11
9.9. SUMMARY 9-12

10. APPENDIX D – MATLAB CODE FOR VISUALIZATION **10-1**

BIBLIOGRAPHY **BIB-1**

VITA **VITA-1**

List of Figures

Figure 2.1. An A algorithm example showing how it progresses in a frontier fashion with f'.* 2-12
Figure 3.1. A pictorial representation of forward aircraft movement. 3-7
Figure 4.1. A dynamic model of the algorithm showing the execution hierarchy. 4-2
Figure 4.2. The initialization part of the program. 4-3
Figure 4.3. Software Function Hierarchy. Part a. 4-6
Figure 4.4. Software Function Hierarchy. Part b. 4-6
Figure 4.5: Representation of the groups. 4-18
Figure 5.1. Visualization of the terrain used for the test cases. 5-5
Figure 5.2. The location of the radar sites on the terrain. 5-6
Figure 5.3. The solution found when the code is first implemented without any modifications. 5-7
Figure 5.4. The solution path found by the new code. 5-8
Figure 5.5. Multiple Aircraft against Multiple Targets. 5-8
Figure 5.6. The Results of experiments on different platforms. 5-9
Figure 5.7. The speedup chart for different platforms. 5-10
Figure 5.8. The Efficiency Chart for different platforms. 5-11
Figure 5.9. The results of the experiments with different search sizes on Myrinet. 5-11
Figure 5.10. Multiple Targets experiment with the execution times of the first target. 5-12
Figure 5.11. Multiple Targets experiment with the execution times of the second target. 5-13
Figure 5.12. Multiple target case speedup chart. 5-14
Figure 5.13. The Efficiency for multiple target experiment. 5-15
Figure 5.14. The Distribution message passing (Red) compared to computation(Green). 5-17
Figure 5.15. The distribution of MPI commands countwise (on the left) and timewise (on the right). 5-18
Figure 5.16. A typical Vampir snapshot showing the message passing between processors. 5-19
Figure 7.1. The development of computers. 7-1
Figure 7.2. A representation of a crossbar switching network. 7-4
Figure 7.3. The representation of caching on a multiprocessor platform. 7-4
Figure 7.4. A representation of a multistage interconnection network. 7-5
Figure 7.5. Static Interconnection Networks. 7-6
Figure 7.6. A visual description of 2-D and 3-D Mesh architectures. 7-6
Figure 7.7. A static and a dynamic tree diagrams. 7-7
Figure 7.8. Hypercubes with different dimensions. 7-8
Figure 7.9. Programming Models commonly used in Parallel Computing. 7-15
Figure 8.1. An example of state-space graph. 8-4
Figure 8.2. An example of AND/OR Graph with arcs between branches representing AND. 8-6
Figure 8.3. Depth-first search with an example. 8-11
Figure 8.4. An example of Breadth-First Search. 8-12
Figure 8.5. Global search & optimization algorithms. 8-16

List of Tables

Table 2.1. Memory and time comparisons of parallel search algorithms. *2-10*
Table 9.1. Implementation of NX "csend" and "crecv" commands in MPI. *9-3*
Table 9.2. MPI Datatypes and their C language values. *9-4*
Table 9.3. "Path" data type as implemented in the code. *9-5*
Table 9.4. Translation of NX "iprobe" command into MPI. *9-6*
Table 9.5. Implementation of NX "csend" for Broadcasting a message in MPI. *9-8*
Table 9.6. MPI counterparts of the NX commands "gslow", "gihigh", and "gsync". *9-9*

AFIT/GCE/ENG/00M-04

Abstract

The general Mission Route Planning (MRP) Problem is the process of selecting an aircraft flight path in order to fly from a starting point through defended terrain to target(s), and return to a safe destination. MRP is a three-dimensional, multi-criteria path search. Planning of aircraft routes involves an elaborate search through numerous possibilities, which can severely task the resources of the system being used to compute the routes. Operational systems can take up to a day to arrive at a solution due to the combinatoric nature of the problem, which is not acceptable, because time is critical in aviation. Also, the information that the software is using to solve the MRP may become invalid during the computation. An effective and efficient way of solving the MRP with multiple aircraft and multiple targets is desired using parallel computing techniques. Processors find the optimal solution by exploring in parallel the MRP search space. With this distributed decomposition the time required for an optimal solution is reduced as compared to a sequential version. We have designed an effective and scalable MRP solution using a parallelized version of the A* search algorithm. Efficient implementation and extensive testing was done using MPI on clusters of workstations and PCs.

MISSION ROUTE PLANNING
WITH
MULTIPLE AIRCRAFT & TARGETS
USING
PARALLEL A* ALGORITHM

1. Introduction

1.1. Background

Pilots generally plan their missions by using traditional methods. Historically, they gather all the environmental information regarding the mission, such as threat condition, radar capabilities, missile range, fuel condition, evasion techniques, distance, day or night time, type of armament on the aircraft, number of planes in the sortie, presence of jammers, etc then, they plan their mission, resulting in subjective, time consuming and generally inconsistent results. The objective of an automated mission route planner is to select the route that minimizes the risk to the mission. For the general Mission Routing Problem (MRP) problem, a route planner typically must plan for multiple aircraft against multiple targets.

Computers are currently involved in contemporary mission planning process [1, 2, 3]. The majority of mission planning systems are interactive software programs, where the pilot selects the final route. In this research effort, a parallelized version of A* search algorithm is designed and implemented to provide an effective and efficient method for evaluating the various environmental elements. This algorithm is an extension of the work done by Gudaitis [6] using NX commands for communication on Paragon platforms. In this work the Message Passing Interface (MPI) is chosen as the message passing environment for interoperability reasons. The aircraft moves to and from the

target(s) through three-dimensional space avoiding solid obstacles (terrain), and avoiding threat areas or moving through them at increased risk to the mission. Path criteria are evaluated in order of importance to the mission, and a route is selected which satisfies the criteria. An optimal solution is one that optimizes the route criteria. Route optimality is determined by a weighted cost function of multiple criteria (objectives). The higher the cost, the greater the risk. In this study, distance and radar exposure are added up to a single cost function. The A* algorithm uses a distributed OPEN list, and a global CLOSED list.

1.2. Problem Statement

The goal of this investigation is to design, implement, test and analyze an effective and efficient MRP system that solves multiple aircraft against multiple targets in a timely manner.

In order to reach this goal some objectives must be achieved. Each objective achieved during the course of this research effort contributes to the overall goal. Gudaitis' work [6] is a basis for this research. Existing tools are used to the extent possible, and new algorithmic designs are included. Gudaitis designed an MRP algorithm using a parallel A* search algorithm with NX instructions on Intel Paragon. The first objective is to transform this NX parallel A* code to MPI. Having MRP implemented using MPI definitely improves its functionality, flexibility, and interoperability.

Another objective is to improve the parallel A* design and implementation in order to find a solution in a timely manner. Time is critical in aviation, especially in operational arena. A route that is safe and effective at planning time may be extremely dangerous and ineffective if it is not planned in a timely manner. The criteria that the planning is based

upon may change during the time of calculation. Thus, improving the execution time has a crucial impact on the efficiency of the mission.

Today's missions are not considered as stand alone missions. Planning a mission requires different criteria to be evaluated. Most of these criteria affect other ongoing missions as well. Therefore, planners have to take into consideration the fact that missions interact with each other. This research studies the feasibility and functionality of adding multiple aircraft against multiple targets into the MRP design and implementation. This improves the functionality of mission planning since it allows multiple missions in a region to be handled without interfering with one another. By adding multiple aircraft it is possible to find paths suitable for different types of aircraft. Also in real life scenarios there are multiple targets that the headquarters would want to assign different types of aircraft according to the nature of the mission and the characteristics of the target.

The capabilities of the aircraft, such as combat radius, maximum rate of climb and dive, and minimum turn radius differ from one aircraft type to the other. These data must be incorporated in order to have a multiple aircraft design and implementation.

Another objective is to design and implement a geocentric coordinate interface and integrate it with an existing visualization system. This feature is one more step towards a more real world MRP system. With that, it is possible to get real world elevation data and have the algorithm search on a real map data.

To have computers find the "best" route, the real world environment of the problem must be mapped into discrete mathematical models and data structures that a computer can manipulate to find a solution. There are many other criteria to consider while selecting the best route. The two criteria mentioned (distance and radar exposure) are

selected, because they are the most prominent ones, and many other criteria depend on these two. In past research [4], the radar threat was calculated before the program execution, which we call the static method. In the dynamic model, computing probability of detection at each node in the path is complicated. It depends on many factors such as the radar cross section of the aircraft, the orientation of the aircraft, the configuration, and the distance of the aircraft to the radar transmitter and receiver. Droddy [5] took the dynamic method. Since the computation complexity was very high he used a sphere model in which the radar cross-section is considered to be uniform, resulting in a great computation reduction.

When designing a parallel software system, the designer should consider the idle waiting time that the processors might have as a result of uneven load distribution among the processors. One of the objectives of this research effort is to have an effective load-balancing scheme that manages the distribution of the work evenly among the processors.

One last objective is to design a set of experiments that allows testing and evaluating the performance of the designed code. The performance metrics are selected in order to analyze the results that are gained from the conducted experiments. The achievement of the objectives and the underlying goal must be justified with the results of experiments.

1.3. Scope

This research has been built upon the work previously done at The Air Force Institute of Technology (AFIT). Grimm [4] proposed a solution to a multicriteria aircraft routing problem utilizing parallel search techniques. He used A* algorithm for his research, but the inefficiencies of his model caused the algorithm to take more time than a real time model would permit. Then, Olsan [21] implemented the code using genetic algorithms.

Droddy [5] added to Grimm's work a dynamic radar model and some improvements for the route evaluation technique which improved the performance of the code. Gudaitis [6] added radar cross section information for radar calculations which made the code more realistic. He also used A* with a global CLOSED list and eliminated redundant search efforts. The effect of a global CLOSED list is shown in this work as well. Gudaitis also improved the stopping criteria by circulating a message in a ring fashion. His work is taken as a starting point for this research. There are many areas to investigate, but only some of them are included in the agenda of this research and the rest is left for future research efforts. The areas of concentration are :

- The rehosting of the code in MPI.
- The design and implementation of multiple aircraft against multiple targets.
- Introduction of Geocentric coordinates format for the map.
- Implementation of an effective load-balancing scheme to improve algorithm efficiency.
- Analysis of the new implementation.

1.4. Approach

Initially, the problem domain is considered and models to represent the problem are determined. Possible MRP approaches and algorithms that are applicable to mission routing problem are then analyzed. Detailed information about the A* search technique, and its applicability for this case are given. It is a specialized best-first strategy where the partial path that appears to lead to the best solution is explored first [34].

A parallel version of A* algorithm is computationally designed and implemented to provide an effective and efficient method for evaluating the various environmental

models. A load balancing scheme is implemented is implemented in order to minimize idle processor waiting times. A dynamic load-balancing scheme is selected that manages the work distribution during the execution of the code as opposed to its static counterpart, which distributes the work before the code starts executing.

Specifically, this research investigation consists of the following tasks:

- Change the parallel NX code to utilize the MPICH implementation of the Message Passing Interface (MPI) as a means of communication between the processors, and implement it on AFIT COWs which comprises six machines interconnected by 1Gbps Myrinet and 10Mbps Ethernet connection using Unix (r) System V Release 4.0 and on AFIT Bimodal Cluster of PCs (ABCs) using 14 machines interconnected by 100 Mbps Ethernet connection using Redhat Linux Release 6.0.
- Design and implement a modified model, which allows implementation of multiple aircraft against multiple destinations.
- Change the coordinate system from a Cartesian format to a more realistic geocentric format. A coordinate is expressed in degree, minute, second notation with Latitude and Longitude values.
- Investigate new data structures for messages that minimize the communication wait time between processors by putting more information within a single message.
- Implement a dynamic load-balancing scheme in order to minimize processor idling time.
- Generate visualization models to determine if the results are effective.
- Analyze the performance of the program in detail by using a profiling tool, and then refine sections of the code that degrades the performance.

- Investigate performance metrics (speed, scalability, and efficiency) on above-mentioned computational platforms.

This research effort is about the design of experiments, results of the experiments, and analysis. Finally conclusions and recommendations concerning areas for continued research are presented.

1.5. Thesis Overview

This chapter gives us a description of the MRP domain, the scope of the thesis investigation, and the approach taken for solving the problem. The rest of the thesis consists of five chapters and four appendices. Chapter II is a literature review of the MRP problem, and presents an overview of the search strategies used in this field, and some information about parallel processing techniques. Chapter III provides information about A* search algorithm that is used, and a high-level design of MRP. Chapter IV is about low-level design and implementation issues. Chapter V is the design of experiments, testing, results, and analysis. Chapter VI includes conclusion remarks and recommendations for future effort. Appendices give information about parallel processing, search techniques, rehosting the existing NX code into MPI and Matlab code to visualize the results.

2. *Literature Review of Mission Routing Problem*

2.1. Introduction

A basic understanding of parallel processing systems, search strategies, and mission routing problem is necessary to conduct this research effort. The following sections discusses parallel processing systems, both hardware and software, search strategies including parallel techniques and heuristic search strategies, and MRP in general. The high level design issues are discussed.

2.2. Parallel Processing

As we might all have been witnessing, there are many investments and improvements in the parallel processing technology. Today massively parallel computers are available with hundreds and also thousands of processors. The advent of this parallel computing technology brought more power and more cost-effective machines into the computing arena. Parallel computers are well cheaper than massively expensive supercomputers and also they are more capable and more scalable. What seemed to be unsolvable couple of years ago has become an easy problem with the many orders of magnitude more computing power of parallel computers, if they are used effectively.

Parallel computing needs research in two fields, hardware and software. Having a hardware system with a very high capacity will be useless without the software to make good use of it. And unfortunately today software is still needed greatly to be implemented on different parallel platforms for different programs. Parallel algorithms need to be designed so that the effect of serial time of the algorithm and idling time of the processors can be reduced by making good use of all the processors and resources.

The fast improvement in the networking arena made it possible to build parallel architectures by reducing the communication latency between processors. Today parallel machines are widely used in many areas such as, weather prediction, biosphere modeling, pollution monitoring, discrete optimization, VLSI design, computer tomography, analysis of protein structures, speech recognition, neural networking, machine vision, and many more.

The MRP problem is studied in sequential machines as well. But the performance did not offered much since program execution to find an optimal path took more than hours. By applying parallel search techniques the work can be distributed among the processors participating in the search process leading to a more effective performance. The main issue is to design a parallel algorithm and be able to distribute work evenly with a static and/or dynamic load balancing schemes, using data locality principle to minimize the communication among the processors. Also when a multiple target case is included in the search process, the communication issues become more important because processors have to send and receive messages from different processor groups as well as from their own communication group. Below is a description of some terms related to parallel processing. It is important to understand these terms in order to realize how a parallel program executes, how we can evaluate a parallel program, where the problem areas are while developing a parallel design. There is more detailed information on parallel processing techniques in Appendix A.

2.2.1. Communication costs in static interconnection networks

The time taken to communicate a message between two processors is called *Communication Latency.* There are some other terms that we need to consider first in

order to be able to understand what constitutes the latency. Although these terms are not calculated in the analysis phase, they are the building blocks of understanding communication patterns between processors. System designers concentrate on these in order to lower the communication cost of the system.

Startup time(t_s) : The time required to handle a message at the sending processor. It includes the time to prepare the message (adding header, trailer, etc.), the time to execute the routing algorithm, and the time to establish an interface between the local processor and the router.

Per-Hop time (t_h) : The time taken by the header of a message to travel between two directly-connected processors is called the per-hop time.

Per-Word transfer time (t_w) : 1/channel bandwidth is called the per-word transfer time. It is the time taken to transfer one word of data from one processor to the other. There are two routing techniques; store-and-forward routing and cut-through routing.

Store-and-forward : Upon receiving, each node stores it and then forwards it to the next node.

Total comm time : $t_{comm}=t_s+(mt_w+t_h)l \approx t_s+mt_wl$ $\Theta(ml)$

Cut-through : A message is advanced from the incoming link to the outgoing link as it arrives. There is no need to store the entire message. So it uses less buffer space, and it is faster. If a link is in use then any message trying to use that link is blocked. That might lead to a deadlock. With certain routing techniques deadlocks can be avoided.

Total comm time : $t_{comm}=t_s+lt_h+ mt_w$ Θ (m+l)

2.2.2. Performance Metrics for parallel Systems

A sequential algorithm is generally evaluated by its execution time, and that is also a function of its input size. This is different in parallel algorithms, because the performance of parallel algorithms depends on the architecture of the parallel computer, and the number of processors as well. A parallel system is the combination of an algorithm and the parallel architecture on which it is implemented. Below are some of the metrics that are also incorporated in the analysis of this research.

Run Time

The parallel run time of a parallel program is not the time elapsed between the beginning and the end of its execution, as is the case for sequential programs. It is the time that elapses from the moment that a parallel computation starts to the moment that the last processor finishes its execution [10].

Speedup

When we implement a parallel system we want to know how much performance gain we had over its sequential version. Speedup is a measure that captures the relative benefit of solving a problem in parallel. It is the ratio of the time taken to solve a problem on a single processor with the fastest known algorithm to the time required to solve the same problem on a parallel computer with p identical processors [10].

$$S = \frac{T_s}{T_p}$$

Theoretically, speedup can never exceed the number of processors, p. A speedup of p is achieved on p processors if none of the processors spend more than $\frac{T_s}{p}$ time. In reality though, we can sometimes achieve a speedup greater than p. This superlinear speedup is

usually due to a nonoptimal sequential algorithm or to hardware characteristics, such as caching capabilities, that put the sequential algorithm at a disadvantage.

Efficiency

The efficiency of p is very difficult to achieve in practice, because processors cannot devote themselves to computation. They have to communicate with each other to solve the problem. Efficiency is a measure of the fraction of time for which a processor is usefully employed; it is defined as the ratio of speedup to the number of processors.

$$E = \frac{S}{p}$$

In an ideal system, speedup is equal to p and efficiency is equal to one. In practice, speedup is less than p and efficiency is between zero and one, depending on the degree of effectiveness with which the processors are utilized [10].

Cost

The cost, or processor time product of solving a problem on a parallel system is the product of parallel run time and the number of processors used.

$$C = p \times T_p$$

It reflects the sum of the time that each processor spends solving the problem [10]. Then we can have another definition for efficiency in terms of cost. It is the ratio of the execution time of the fastest known sequential algorithm for solving a problem to the cost of solving the same problem on p processors. If the cost of solving a problem on a parallel computer is proportional to the execution time of the fastest known sequential algorithm on a single processor, then that particular parallel system is said to be cost-optimal. We also know that efficiency is the ratio of sequential cost to parallel cost. So, a cost-optimal system has an efficiency of $\theta(1)$.

Scalability

As we know the upper limit for speedup is the number of processors. Speedup does not increase linearly as the number of processors increase. Amdahl's law states that the efficiency drops with an increasing number of processors [10]. Secondly, a larger instance of same problem yields higher speedup and efficiency for the same number of processors, although both speedup and efficiency continue to drop with increasing p. If, in a parallel system, we have the ability to keep efficiency at a fixed value by simultaneously changing the number of processors and the size of the problem then we call this system to be a scalable one. We want a parallel system to be scalable, because it is a measure of system's capacity to increase speedup in proportion to the number of processors. It means that the system can utilize increasing processing resources effectively.

2.2.3. Sources of Parallel Overhead

The overhead function of a parallel system is the difference between its cost and the serial run time of the fastest known algorithm for solving the same problem. An advantage with this definition is that we combine all the sources of performance degradation in a single expression, so that we can study their cumulative effect on the performance. The main sources of overhead in a parallel system are interprocessor communication, load imbalance, and extra computation.

There must be some kind of communication between processors to have a parallel system. This communication is the most significant source of overhead. Since it is impossible to predict the size of workload for each processor, we cannot divide the problem statically with a uniform workload for each processor. If the workload is not

uniformly distributed, then some processors will be idle while some others are working. This idling time will contribute to the overhead function. The best known sequential algorithm for a problem may be difficult or impossible to parallelize, forcing us to use a parallel algorithm based on a poorer but easily parallelizable sequential algorithm. In that algorithm we might need to do extra computation on different processors since we cannot use the results of computations done on other processors. That is also another source of total overhead which degrades the performance of the parallel system.

2.3. Search Techniques

There are many search strategies in the scientific engineering field that are designed to deal with different problems. Some of them may work very well for some applications, whereas perform very poor on other applications. It is important to get an overall idea of different search techniques, their characteristics, and their applicability for certain problem instances. That way it becomes more clear to realize the place of A* among all the other techniques, and why it is chosen for this specific study. A more detailed explanation of common search techniques and place of A* can be found in Appendix C.

2.3.1. Basic graph searching notation

A graph consists of a set of nodes (or vertices), which in our context represent encodings of subproblems. Every graph we will consider has a unique node s, which is the start node. Nodes are connected to each other by arcs or links. If there is an arc from *n* to n_1 then n_1 is called a successor (or a child or an offspring) of *n* and node *n* is called a parent or a father of n_1. The number of successors coming from a given node is the degree of that node, and normally is denoted by *b*.

A tree is a graph in which each node, except root, has only one parent. A node in a tree that has no successors is called a leaf, a tip, or a terminal node. A uniform tree of depth N is a tree in which every node at depth smaller than N has the same branching degree whereas all nodes at depth N are leaves. Often the arcs are assigned weights representing either costs or rewards associated with their inclusion in the final solution. If a path exists from n_1 to n_k, n_k is said to be a descendant of or accessible from n_1, and node n_k is called an ancestor of n_k. The cost of a path is normally understood to be the sum of the costs of all the arcs along the path.

The most elementary step of a graph searching that we consider is node generation, that is, computing the representation code of a node from that of its parent. The new successor is then said to be generated, and its parent is said to be explored. A coarser computational step of great importance is node expansion, which consists of generating all successors of a given parent node. The parent is then said to be expanded.

A search procedure, a policy, or a strategy is a prescription for determining the order in which nodes are to be generated. We distinguish between a blind, or uninformed, and an informed, guided, or directed search. In the former, the order in which nodes are expanded depends only on information gathered by the search but is unaffected by the character off the unexplored portion of the graph, not even by the goal criterion. The latter uses partial information about the problem domain and about the nature of the goal to help guide the search toward the more promising directions.

Naturally the set of nodes in the graph being searched can at any given time be divided into four disjoint subsets:

- Nodes that have been expanded.
- Nodes that have been explored but not yet expanded.
- Nodes that have been generated but not explored.
- Nodes that are still not generated.

Several of the search procedures discussed require a distinction between nodes of the first and third group. Nodes that were expanded are called closed, whereas nodes that were generated and are awaiting expansion are called open. Two separate lists called CLOSED and OPEN are used to keep track of these two sets of nodes.

2.3.2. Approaches to Combinatorial Problems

There are two broad approaches for solving combinatorial problems. Deterministic algorithms make all decisions in a deterministic manner, while probabilistic algorithms (stochastic) make some decisions based on probability. These algorithms may guarantee a best solution (or a solution within a specified tolerance of the best solution, but these algorithms also may take too long to solve a given problem [21]. Although probabilistic algorithms are not guaranteed to give us the best solution, they run in less time than deterministic methods. So, we have to have a compromise on the tradeoff for solution quality and solution time.

Deterministic methods can be general purpose branch and bound algorithms and more specialized methods such as greedy algorithms. The first class of algorithms work on general search problems, while the greedy algorithms work on any search problem. The branch and bound algorithm methodologically check the entire search space of the problem. If we apply heuristics then we can search some part of the search space implicitly, rather than explicitly, thus reducing the search time. Greedy algorithms

produce optimal solutions if the problem domain is such that local optima correspond to global optima [34]. Probabilistic algorithms produce solutions based on probabilistic selection of solutions. Sometimes it is a better idea to pick the next step at random, instead of determining the best step [34]. Algorithms using stochastic methods are called probabilistic algorithms. Random search, simulated annealing, and evolutionary strategies such as evolutionary algorithms and genetic algorithms are the most popular techniques. The search behavior can change with a given input based on the random probabilities of the method used. Deterministic search methods can yield optimal solutions for an objective function, whereas stochastic search methods may or may not give optimal answers [34]. Table 2.1 is the tabular representation of the optimization algorithms. Some advantages and disadvantages are also shown.

Table 2.1. Memory and time comparisons of parallel search algorithms.

Parallel Search Algorithms			
ALGORITHM	MEMORY	TIME	COMMENTS
Depth First	Requires little memory, Best feature	Varies greatly depending on where in search graph a solution is located. Can require prohibitive amount of time.	Branch and bound, and backtracking can greatly reduce time.
Breadth First	Can require prohibitive amounts of memory.	Varies greatly, depending on where in search graph a solution is located.	Branch and bound, and backtracking can greatly reduce time and memory required.
Best First	Can require prohibitive amounts of memory. Uses less than Breadth First.	Solutions are consistently quickest, but can be longer than Depth First depending on cost bound.	Characteristics depend on the variation used (AO*, A*, Z*, etc.).

Dynamic Programming	Same as Breadth First	Depends on Objective Function. May be comparable to A*.	Recursive Breadth First approach. Efficient because operations on same input are stored, not repeated.
A*	**Same as Best First.**	**Same as Best First.**	**If heuristic is admissible, and monotonic, A* dominates all other algorithms with access to the same heuristic.**
IDA*	Memory the same as Depth First.	Claimed to be the same or better than A*.	Still undecided issues on relative speed, especially in parallel version.

*2.3.3. A Specialized Best-First Algorithm, A**

There are different types of popular best-first strategies that are commonly used today. Among these, four of them are used for AND/OR graphs (General Best-First (GBF), GBF*, AO, and AO*) and five are limited to searching OR graphs (BF, BF*, Z, Z*, A*). A* can be used both for optimization problems and satisficing problems because the shortest path is the most natural choice for the small-is-quick principle.

A* is a best-first, branch-and-bound search process. A* is a deterministic type algorithm effectively used in many applications. It employs a heuristic estimate of the remaining cost of reaching the current location to determine the "fitness" of the currently examined location. The approach is inherently graph-oriented, rather than tree-oriented, since information about the path searched so far is maintained. Heuristic search is often useful in minimizing the combinatorial explosion common to search problems. When the operation to expand children states is nontrivial, as is the case with MRP (Mission Route

Planning, a problem on which A* search technique has already been applied), considerable improvement in execution time can be achieved by parallelizing that operation.

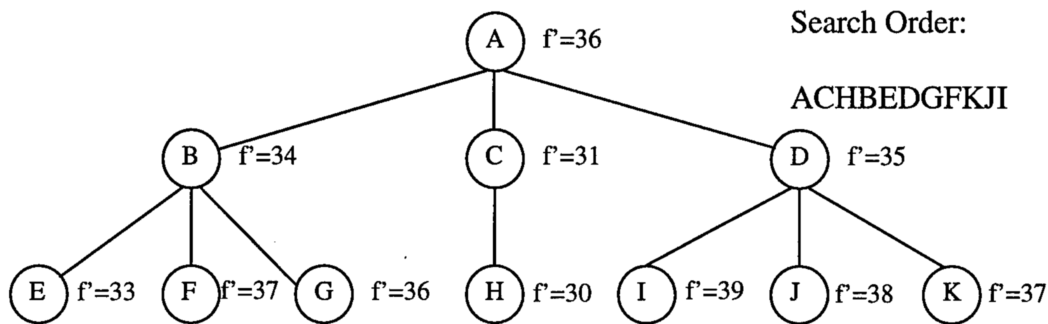


Figure 2.1. An A* algorithm example showing how it progresses in a frontier fashion with f'.

The A* algorithm contains two lists, OPEN and CLOSED, and operates on a graph G of N nodes. OPEN contains successor nodes that have not been evaluated. CLOSED contains all nodes that have been evaluated. A successor of node n, labeled n', is a node that is adjacent to n. c(n,n') is a cost function that evaluates the cost of moving from node n to its successor n'.

A* algorithm uses an additive cost heuristic to quickly home into the best solution. A* has gained wide acceptance because with proper choice of the heuristic, A* dominates (i.e. performs better than) all other algorithms with access to the same heuristic. A* algorithm was developed in 1968 by Hart, Nilsson, and Raphael. [28] The A in A* means that it uses an additive cost function. The '*' refers to optimality in that if certain conditions on f(n) are met, A* will always find a solution that optimizes the cost function f(n).

A* defines the 'best' node in terms of a function f' where f'=g+h'. The value of g is the 'cost' of the path from the initial node to the current node in question. The value of h' is an estimate of the 'cost' of the path from the current node in question to the goal node. The node with the smallest f' value is selected from all the nodes in consideration to be expanded on each cycle of the search. The h' value is called the heuristic and varies from one problem to the next. The optimality of the A* search can be analyzed based on the characteristics of the f' function. If f', the estimate of the cost of the path from the initial node to the goal node through the current node, is less than f, the actual cost, at each node in the graph, then the A* search will find the 'optimal' solution based on the f' function.

It is worth noting, though, that the breadth-first strategy is a special case of A* with h=0 and c(n, n')=1 for all successors. Similarly, the uniform-cost strategy is also a special case of A* with h=0. Depth-first strategies, on the other hand, can be obtained from A* by setting f(s)=0, f(b)=f(a)-1 when b is a direct successor of a.

A simple test on the type of information used by A* can guarantee that an optimal solution will eventually be found. Still another simple test guarantees that A* retains all its virtues even without appraising duplicate nodes on CLOSED. Also another test may reveal that one heuristic function is consistently better than another in that it always causes A* to expand a smaller number of nodes. In the A* search space is an OR graph, the task is to find a path s to some goal node, and a heuristic h(n) is computable for every node in the graph.

A* explores (as does every BF strategy) the state-space graph G using a traversal tree T. At any stage of the search, T is defined by the pointers that A* assigns to the nodes generated, with the branches in T directed opposite to the pointers. Whenever it is

desirable to stress the fact that a given path is part of T (at some phase of the search) it is denoted by this pointer-path by PP, e.g., $PPn_1\text{-}n_2 \in T$.

2.3.4. A Algorithm and its Path Planning Implementations*

In general, the A* algorithm is the most reported and most used algorithm for route planning problems [15]. The popularity of A* is supported by its good performance and applicability over a wide range of route planning problems. A small selection of those references that are most closely related to this research is discussed next.

Grimm [4] developed an optimal route planner based on the A* algorithm implemented on parallel processors for increased performance over conventional serial processors. An optimal route was generated for one aircraft against one target. Two criteria were used to plan the route-distance traveled and radar detection. Terrain was simulated by a three-dimensional mesh representation. The aircraft was modeled as a point mass with movement in one of 24 directions (one in each direction excluding backwards, and up to two upwards). The radar threat was modeled statically as a matrix overlaid on the terrain model. Up to five radar threats were represented in the threat matrix. The 'cost' associated with each route was a weighted combination of the distance traveled and the radar probability of detection. The optimal route was selected as the route with the lowest overall cost. Grimm's approach did not work consistently because a non-monotonic heuristic was used. The heuristic was a combination of both distance and radar threat, and could overestimate the actual cost of moving from a given node to the target. It "looked ahead" a specified distance and then estimated the rest of the way to the goal.

Droddy [5] continued the work begun by Grimm. The significant change he made to Grimm's work was the elimination of the static radar threat matrix and associated high memory storage cost. To make this change, Droddy modeled the aircraft as a spherical Radar Cross Section (RCS) and calculated radar probability of detection as needed. In this way, both bistatic and monostatic radar threats could be modeled. Droddy's code had faster execution times than Grimm, but Droddy's lack of a closed list for the A* algorithm made the code unreliable for many types of problems where the target is surrounded by radar threats. Elimination of the closed lit is valid for sequential algorithms if the heuristic function is monotonic. The heuristic used by Droddy was simply the straight-line distance from the current node to the target. Although this heuristic is monotonic, it is a gross underestimator of the actual cost when radar and terrain are in the path. In the parallel version of A* implemented by Droddy, this heuristic and lack of a Closed list caused the program to generate unproductive and duplicate nodes on the Open list. This, in turn, decreased the efficiency of the algorithm and caused erratic performance for most test cases.

In [18], A* is used to plan paths in three dimensions for an autonomous underwater vehicle (AUV). Details of the algorithm and the heuristic function are not given. The algorithm is implemented on a serial processor. A quadtree structure is used to reduce the number of search nodes. However, the penalty for this is that a suboptimal path may result because the path is constrained to go through the centers of quadrants [18]. The experiments showed that a small change in the number of path constraints resulted in a large change in execution time. This validates the need for parallel processors to handle large multicriteria 3-D path problems such as MRP.

In [20], Pal develops an A* approach to robot path planning in three dimensions. However, it can't really be called an A* approach because it seeks a 'good' solution rather than an 'optimal' solution. By sacrificing optimality, this method is able to provide collision avoidance solutions quickly for movement of a robot arm. Reported execution times were less than a minute to run on a PC/AT-386 for most experiments. However, the physical search area was constrained to a small fixed size. In addition, a simple polygon model was used for the search area rather than a standard grid. These design decisions reportedly worked well for the robot arm implementation but do not apply, in general, to MRP.

In [26], Teng incorporates time constraints, as well as distance and threat criteria, in his modified A* algorithm for robot path planning in two dimensions. This work is quite similar to the IBM work of Stiles reported in [27]. Both used a CM-2 massively parallel hypercube network and reported path planning experiments in two dimensions. However, Teng used a much larger grid size (512 x 512) than Stiles (100 x 100), and had more path constraints. Reported results showed that Teng's parallel A* approach performed faster than Stile's parallel dynamic programming approach. This lends support to our choice of the A* algorithm over dynamic programming for application to the 3-D aircraft route-planning problem.

2.4. MRP Algorithm in Detail

On the MRP problem, OPEN list is designed to be implemented by distributing the list among the processors. This way we can reduce the communication overhead, but processors would be finding only the local best which may not be the global best next point. This is not a problem in our case since in MRP local best also qualify for the global

best. CLOSED list is implemented using explicit global list not to duplicate the expanded nodes.

The distance and the radar exposure are taken as criteria to select the best route making the problem Bicriteria route problem (NP-complete). These two criteria are then reduced to one cost function making the problem $O(n^2)$ problem [6].

There is some research done in AFIT on MRP using different methods. Gudaitis' work is the most recent one. Droddy comes right before him. He implemented dynamic radar calculations, improved the route evaluation function, and modularized the software implementation. These enhancements provided significant performance gains over Grimm's work [4]. Gudaitis took Droddy's work and worked on it to improve the inefficiencies of the project preventing its real-time implementation. His results meet real-time objectives for all the scenarios tested. He introduced an explicit CLOSED list and eliminated manager/worker structure of the parallel algorithm. He also improved the radar cross-section estimation, thus making the algorithm closer to reality.

Although the general case of MRP problem is shown to be NP-Complete, the A* algorithm [6, 36, 16] reduced the number of n that must be searched explicitly, so that the time complexity is $O(l^2) \leq O(n^2)$, where l is the number of points in the optimal path [6].

The local best approach is used and eliminated the interprocessor synchronization required by the manager/worker "global best" approach used in previous research [4, 5]. Local best approach is found to be more effective than the global best approach [6]. In addition, a global CLOSED list was found to be essential to reducing duplicate work and thereby improving efficiency. A copy of the CLOSED list is stored at each processor and

kept globally current through interprocessor message passing. The speedup of the parallel algorithm is due to the use of multiple distributed OPEN lists.

The grid structure used for the terrain model was simple to implement and effective in providing dynamic calculations of radar exposure. The array implementation provided quick O(1) access to the information without a large burden on memory. Dynamic calculations for radar exposure is simpler and the most promising approach for a real-time mission route planner. [6]

2.4.1. Time Complexity of Bicriteria MRP Problem

NP-Complete problems are a class of hard-to-compute problems in terms of time and space requirements [38]. There is no known solution to these problems in less than exponential time, i.e. $O(c^n)$ worst case where c is a constant and n is the size of the problem space. It means that the worst case time complexity of the problem is an exponential function of the size of the problem. The NP-complete problems are characterized by two well-known features. First, the problem has to be combinatoric or have an exponential time complexity. Second, the data structure must be transformable to all other NP-complete problems in polynomial time, and vice versa. This second condition insures one of the most important characteristics of this class of problems. If any of the problems could be shown to be less than an exponential time complexity, then the class of these problems could be transformed and solved in less than exponential time.

This research effort tries to search into methods for parallelizing NP-complete problem solution algorithms and the amount of improvement that can be achieved by implementing one of them by reducing it with a single cost function. In real world

situations there are a good number of applications whose search space is huge and their solution time complexities are exponential.

NP-complete search techniques generally utilize partial state information along with a limiting function to improve the efficiency of the search algorithm [6]. There is another aspect of search algorithms that must be taken into account. It is the unpredictable nature of the search space. Since the size of the explicit search space is not known, having a dynamic load balancing technique that distributes parts of the search graph greatly affects the performance of the algorithm by reducing idling time of the processors.

MRP is a three-dimensional, multi-criteria path search, which is an NP-Complete problem [6]. The case of two criteria is also an NP-Complete problem. Two criteria considered in this research are exposure to radar and distance traveled. However, if the criteria are combined into a single cost function, the problem simplifies to a shortest path problem which can be solved in $O(n^2)$ time [6]. Dijkstra's algorithm is known to be a good algorithm to find a shortest path. A*'s superiority over Dijkstra's algorithm is that it uses a heuristic function that gives the likelihood of a node to be on an optimal path. A* algorithm does not have to search the entire search space as the Diskstra's algorithm does. With a good heuristic function the search space may be reduced, and only the paths with a better likelihood of being a solution are examined first. That way an optimal solution may be found earlier. Optimality is kept by an appropriate mapping of the search graph to satisfy the optimal path substructure requirement of shortest path algorithms. On this research the planning of a route for multiple aircraft to multiple destinations is considered.

In this research to calculate the time complexity of the problem, it is assumed that the multiple criteria for evaluating the path are combined into a single additive cost function that is bounded above by a constant, Cmax. It is also assumed that there are no negative values for the cost function. Only two criteria, distance and radar are considered. For each pair of points, the distance is the length l(e) of the edge connecting adjacent points in the path. The radar threat is the probability of detection $0 \leq P_d \leq 1$ times a radar multiplication factor k times the distance l(e). The maximum cost per edge is therefore l(e) + kl(e) = (k+1)l(e). Consider a function c(i, j) = (k+1)l(e_{ij}) which determines the arc cost of moving from one point i to point j. The most direct route is defined as the shortest distance obtained by summing the lengths of line segments connecting consecutive points in the path. Cmax is obtained by setting all criteria to their upper bound for a unit travel and multiplying by L. The cost of the optimal route from i to j is therefore bounded above by the maximum cost of a direct path between those points. In other words, the time complexity of finding an optimal solution does not depend on the total number of points in the search space. Instead, the search time depends on the maximum length for an optimum path. To calculate the maximum length we need to determine a priori the following:

- A constant $k = \frac{C_{\max}}{C_{\min}}$ representing the ratio of the cost of worst case travel (fully exposed to radar) to the cost of optimum travel (with no radar exposure), and
- L representing the length of a "direct" path. L may not be unique due to digitization bias.

The longest path Lmax with cost ≤ Cmax is one with no radar exposure. The longest path with no radar exposure must be equal to kL. Consequently, the length of the optimal

path is bounded above by kL = Lmax. The paths of length ≤ Lmax that have endpoints at start S and goal G are confined to the area of an ellipse drawn such that the sum of the radii is ≤ Lmax. Then since only the points within the ellipse should be checked, the total number of points N that must be considered in a search for an optimal solution is the area of an ellipse $O((kL)^2)$ for two dimensions, and the volume of an ellipsoid $O((kL)^3)$ for three dimensions. The maximum number of paths that must be generated is determined by the branching factor b, and the maximum number of points in the optimal path as follows:

$$\sum_{i=L}^{kL} b^i = (b^{kL} - b^L)b/(b-1)$$

The lower bound for the number of paths generated is readily determined if we assume that at each point, the successor is chosen such that it lies on the optimal path. Thus, MRP has lower bound $\Omega(bL) \leq MRP \leq O(b^{kL})$.

Typical values for b and k are 9 and 4 respectively, with L ≅ 100 or larger. The branching factor b is determined by the aircraft flight model, and represents the number of next points reachable by the current point. The variable k is the radar multiplication factor that is set according to the level of threat assigned to radar. The higher the value of k, the more the route planner tries to avoid radar in the route selection. With these values for b and k, even the largest, fastest computers in the world would not be much help in the worst case. Luckily, time complexity is closer to the lower bound in practice. This is the initiative behind this research for implementing parallel processing techniques with the A* algorithm. Another approach is to transform MRP into Shortest Path Problem [10, 8]. Then, the time complexity of the problem is reduced to $O(N^2)$, in a search space of N points. Therefore the upper bound is reduced to $O((kL)^5)$, which makes our complexity :

$$\Omega(bL) \leq MRP \leq O(k^5L^5) \leq O(N^2).$$

This reduction enables us to use this algorithm in real-time with larger input sets [6].

2.4.2. Bicriteria MRP Implementation

As mentioned above the most recent work in AFIT on MRP has been done by Gudaitis [6]. He implemented one aircraft against one target model on Paragon using NX commands. Setting up communication between processors using NX is easier than it is with MPI commands. With MPI, the user has all the responsibility of setting up the communication patterns. The user has to track which processor is sending what, to whom, when; and then make the receiving processor ready for the receive operation. NX also has more flexibility using the broadcast commands, whereas MPI has to use a set of send operations in an iterating fashion for that purpose (Although is has a bcast function). This definitely affects the performance because instead of one single operation, MPI has a number of send operations with their corresponding receive operations.

He added up the distance and the radar cost into a single cost function. Each node added to the best path on a processor added the cost associated with it. The optimal path then is the least cost path at the end of the execution. When the program started its execution, a controller would start the search process and send to each processor enough number of nodes to explore. That way each processor searches a different portion of the search space by using A* search in a frontier exploration fashion. Each processor has a local best path found so far, along with an OPEN list. They also have a CLOSED list to keep closed nodes. The processors update the list by sending messages when a change occurs.

The terrain is mapped to a grid structure, and is read from a terrain file. He used a Cartesian format for specifying locations on a terrain. As mentioned before the radar detection scheme is a dynamic model. It is not set up front before the execution of the program. The characteristics of the radar sites are read off of a file, and the threat calculations are made on the fly as the program is executed.

His experiments showed that optimal solutions are obtained for scenarios with 15 radars with execution times of less than ten minutes using 16 processors on the Intel Paragon. For some scenarios superlinear speedup values have been observed. The improvement in execution time for the parallel algorithm is due to the use of multiple distributed OPEN lists, with duplicates eliminated using a global CLOSED list. The algorithm scaled well on the Paragon with up to 20 processors. For larger numbers of processors, it was found that the problem size must be increased to achieve the same speedup.

2.5. Message Passing Interface

The voracious need for greater computing power in the science and technology arena cannot be satisfied by conventional, single processor architectures. There are many problem areas such as combinatoric problems that just cannot be solved with a single processor machine in a given amount of time. Parallel computing satisfies this need for especially time critical problems. What took days to calculate a few years ago takes only minutes with the application of this new power. Other than the expected high cost of these parallel architectures, the need for parallel software that enables users to exploit all the benefits of the multi processor systems is another problem with it. The development

of standards for programming parallel systems has accelerated the development of this architecture.

Message Passing Interface (MPI) is one of the most widely used standards. As one might think, it is not a programming language, but a library of subprograms that can be called from C and Fortran 77 programs [22]. MPI has different implementations designed for heterogeneous networks, such as MPICH, LAM, CHIMP, or MPI-2 [22, 23, 24]. As its name implies it is based on message passing, one of the most widely used paradigms for programming parallel systems. Programs written using MPI are portable and efficient, if used correctly. It is the interoperability of MPI on heterogeneous platforms that affected the decision of using MPI in this research effort. MPI is basically the function of sending a message from one processor to another. Although this approach makes the life easier for the user by keeping the user away from the programming details, it is a difficult and tedious work on the programmer side. The reason is that the programmer has to keep tack of the whole message traffic between the processors.

MPICH implementation of MPI is available on both Unix and Linux operating systems in AFIT. Appendix C explains the transformation of the MRP from NX into MPI with information about the restructuring of the data types to be manipulated by MPI.

2.6. Visualization Techniques

Data visualization is the art and science of turning complicated sets of data into visual insight. It enables people to easily make sense out of what otherwise would just be a set of meaningless numbers by using the one third of their brain devoted to visual processing and uses the power of the human eye and brain to discern relationships by presenting complex data as multi dimensional color images and animations. [39]

Industries ranging from financial services to telecommunications now use their computers to collect huge amounts of information every second. As a result they have databases and data warehouses full of incredibly valuable data. Unfortunately, in many cases, organizations fail to turn this data into insight that can lead to new discoveries that improve their competitive position. Now, with Data Visualization, this data can be rapidly turned into images that enable organizations to achieve true insight. Data Visualization is performed by computer programs that run on today's powerful desktop computers and workstations. These programs can reach across corporate networks to integrate data from many sources into images that can instantly lead to knowledgeable action. These computer programs are customized to maximize the ease of understanding in each data visualization situation. Data Visualization solves the problem of understanding multivariate data. When it captures data, it typically capture a set of data values for each record along with multiple independent variables such as time, location, and temperature. By looking at simple graphs of data values versus an independent variable it can easily miss complex interactions and trends. What appears as random data along any one axis can have discernable trends when viewed in a multi-dimensional format. Data Visualization uses techniques such as 3D imaging, colorization, animation, and spatial annotation to extract instant understanding from multivariate data. Well constructed Data Visualization applications not only display the data as meaningful images but also allow the user to interact with the data. Users are able to "drill-down" into the data to get an understanding at successively refined levels of detail. They are able to view 3D images from multiple aspects and "fly-through" their data on a journey of exploration. Data Visualization can enable users to play "What-if" games with their data

and to extrapolate future trends from past data. Visualization can also be integrated into application programs that generate large amounts of data. This enables users to change design parameters and rapidly visualize the result. It is very important to be able to identify and isolate some of the features among all the others. The aim in many disciplines is to study the evolution and essential dynamics of these amorphous regions or features and describe them for modified time periods, thus obtaining a deeper understanding of the observed phenomena or a reduced-and-simpler model of the original set of complex equations.

The aim of the visualization of massive scientific data sets is to devise algorithms and methods that transform numerical data into pictures and other graphic representations, thereby facilitating comprehension and interpretation. By coloring and connecting different segments of data (based upon a user-defined color map and threshold), the algorithms highlight regions of activity. Features are fundamental to the analysis/visualization process for many reasons:

Terrain visualization is one of the many application areas of scientific visualization. The objective of the Terrain Visualization is to integrate and demonstrate capabilities to collect and process high resolution digital terrain elevation data needed to accurately represent the 3-D image. By doing so it can solve complex geographical problems through computer analyses, design and implement graphic workstation networks, conduct research in graphic technologies and spatial algorithms, and provide services such as remote sensing, geo-database management, computer graphics, mapping, image processing, digitizing, and spatial modeling [39]. Still another goal is to integrate large

national geo-databases into a readily accessible computer library with efficient data management and maintenance systems.

In this research effort as well the place of visualization is very important. Parameters of the program such as terrain, radar sites, aircraft and the solution path can be visualized. It gives more perspective to the user about the effectiveness of the solution. Having an array of coordinates as the optimal solution would not make any sense to the user. But after putting it on a terrain with radar sites, and other aircraft in the scenario, and marking the place of starting base and the target locations, the planner is able to realize if the path meets required constraints. Matlab is used to visualize the results of the code. Following chapters give representations of the results using the Matlab code that is designed for that purpose.

2.7. Summary

This chapter presents the issues related to parallel processing techniques and architectures, and certain search algorithms. That way it is easier to realize the place of the problem among many categories, and also the place of the selected solution method. It also gives us reference to the previous work done in this field. Brief information about Message Passing Interface (MPI) is also included. The following chapters go into more detail about the MRP and its solution method used in this research. However, it is clear that this chapter provides us a basis that we can build models upon. Work done in other chapters makes great use of the information given here.

3. Requirements and High Level Design of Multi Target MRP

3.1. Introduction

This chapter discusses a detailed description of MRP problem, the search technique used on this research, i.e., A* search, and the high level design issues. The high level design explains the general approach taken, and some models created for manipulation with computers. Therefore, in the next sections Mission Route Planning is described, and then the methodology used in this research is discussed in a high level.

3.2. Mission Routing problem

As mentioned in previous chapters, MRP is a selection of an optimal path from a friendly base to a target through a hostile environment. There are many factors that affect the mission, such as the type of the aircraft, radar sites, threat conditions, weather conditions, refueling capability, terrain masking issues, distance, etc. A pilot has to consider all these factors and plan his flight accordingly to minimize risk to the mission. Automating this planning takes a lot burden on the people planning these missions. But, in order to be used in real life, automated tools must be designed to give consistent, effective, and efficient answers in a short time. There are some research efforts in this field or similar ones such as traveling salesman problem [29, 30], orienteering [31, 12], and aircraft routing [1, 2, 4, 5, 6, 21, 32, 33]. This study focuses on the issues of implementing multiple aircraft against multiple targets. For the simplicity, only two aircraft with two targets model is designed. Also the format of the terrain is designed to be expressed in geocentric format rather than Cartesian format. That way it is possible to import real world elevation data and implement the code on that terrain.

3.2.1. Mission Parameters.

In order for us to start planning a route, we need to have an Air Tasking Order (ATO) generated by a higher authority. The ATO assigns all the missions to specific forces. It has target locations and identifications, time on target information, type and number of aircraft in the mission and the type and amount of ammunition to be delivered, and the support fleet if there is any [1].

3.2.2. Representation of the world.

The mission routing problem requires discrete models that must be developed to represent the real world. The models that we need are as follows:

- Digital representation of the terrain (maps, grids, etc.).
- Radar detection and Radar Cross Section (RCS).
- Model of aircraft movement.
- Cost function to evaluate the criteria for selecting the optimal route.

3.2.2.1. Model of Terrain/Search Space

The search space is discretized such that the least-cost path can be computed by interconnecting subpaths from a starting point to a goal, and the "optimality" of the path evaluated when the goal is reached. A three dimensional model is required to represent the terrain and flight space for an aircraft. One of the models for the terrain is a standard three-dimensional grid of x, y, and z coordinates that represent latitude, longitude, and elevation, respectively. An advantage of this abstract representation is its simplicity. A 3-D array can be used as the implemented data structure. Each dimension of the array then represents a physical dimension of space. Therefore the entire search area is explicitly represented by discrete points. The discrete points are required to calculate radar exposure dynamically along the flight path.

A disadvantage of standard grid model is that the number of grid cells needed for a given resolution is $O(d^3)$, where d is one dimension of the terrain representation. Therefore doubling the resolution of the grid would multiply the number of grid cells required by $2^3=8$ times. However, if one of the dimensions is represented separately, e.g. the elevation, then the resolution in the plain of terrain can be changed with a reduced cost of $O(d^2)$. Also the mission may require elevation to be measured in feet or meters or the distance in miles or kilometers. Therefore, the grid representation is the model selected because of its simplicity and its advantage over other terrain models in dynamic calculations of radar detection with multiple radar cross-sections (RCSs). The example terrain grid used on this experiment is 100x100x25 with a scale of 100 feet in the XYZ plane. For the distance criteria, the total distance of the path must be within the aircraft's range in that configuration. A constant scale factor of 100 feet is used for the terrain space. Although the program can read the scale factor for the terrain that adjusts the resolution of the search space, it is set to the standard value so that one increment in the Cartesian format corresponds to one second increment in the geocentric format coordinates. Thus, it is easy to incorporate real world map data into the program.

The terrain file has the bottom right coordinates of the map data (which corresponds to 100x100) that comprise the search space. The latitude and longitude information is measured in degrees where one degree is composed of sixty minutes and one minute is composed of sixty seconds. The Latitudes and Longitudes are converted into a decimal seconds format for computational ease. Since one minute is approximately one Nautical Mile, i.e., six thousand feet, then one second is approximately 100 feet. That is also the

reason that the scale factor, or the resolution of the terrain space is selected to be a constant 100 feet.

The terrain is an artificially generated S-shaped mountain range for ease of evaluation and testing. The resolution and the shape of the terrain can easily be changed to the scenario desired by just changing the terrain file.

The representation of location points in that terrain space is done by giving latitude and longitude values. The bottom right coordinates of the sample map is given in the ATO file. The base and target locations are also given in latitudes and longitudes in the ATO file. The algorithm still uses the numbered values of the discrete points in the map. Therefore a translation is made after the ATO file is read. Each change in X or Y direction causes a 100 feet change, since that is the scale factor used. Also the representation format used is a degree, minute, second format. Since one minute is one Nautical Mile on a longitude and approximately one NM on a latitude, it is assumed that one second is 1/60 of a NM, i.e., 1/60 of 6000 feet. That corresponds to 100 feet. Then it can be concluded that a change of one in X or Y direction is equal to a 1 second change. Then it is easy to make the translation by using simple mathematical calculations.

3.2.2.2. Radar Model

Radar exposure is calculated using the standard radar range equation which is a function of the radar system characteristics, position of the aircraft with respect to the radar, and radar cross-section (RCS) of the aircraft. When a radar transmitter sends out waves, they travel out the same way as the water waves would do when a pebble is thrown into. The radar energy is equally distributed over all the points with a radius of *r*. But, we can point the radar antenna in a direction and have most of its power sent to that

direction. We call G_t, the ratio of power of the radar in the pointed direction to that of the other directions, as the gain of the antenna. This radar wavefront intercepts a target when the signals are reflected back. The amount of energy reflected depends on the radar cross section σ of the aircraft. The signal-to-noise ratio S/N is defined by the following equation [7] :

$$\frac{S}{N} = \frac{P_t G_t G_r \lambda^2 \sigma}{(4\pi)^3 K T_s B_n L_t L_r R_t^2 R_r^2}$$

S = received power (in watts)
P_t = power transmitted by the radar (in watts)
G_t = power gain of the transmitting antenna
G_r = power gain of the receiving antenna
λ = wavelength of the signal frequency (in meters)
σ = aircraft RCS (in square meters)
R_t = distance from the transmitter to the aircraft (in meters)
R_r = distance from the receiver to the aircraft (in meters)
N = noise power at the input to the receiver (in watts)
K = Boltzman's constant (1.38×10^{-23} joules'Kelvin)
T_s = receive system noise temperature (in degrees Kelvin)
B_n = noise bandwidth of the receiver (in Hertz)

The RCS of the aircraft is a characteristic of the aircraft and depends on the radar operating frequency, the shape and material of the aircraft, and the position of the aircraft relative to the radar transmitter and receiver. Since the RCS patterns are so complex, Grimm [4], and Droddy [5] treated the aircraft as a moving sphere with radar cross section to be the same from all the angles. In this research the RCS is modeled to reflect the difference for top, bottom, side, front, and rear views of the aircraft with a monostatic radar, in which the transmitting and receiving antenna are at the same location.

One of the criteria for route selection is probability of detection. It depends heavily on the strength of the signal reflected off the target aircraft. A target is detected when the reflected signal strength is over a certain threshold value. Probability of detection in this

research has been modeled such that $S/N \geq 15$ implies $p_d = 1$, $S/N \leq 5$ implies $p_d = 0$, and $5 < S/N \leq 15$ is a p_d value between 0 and 1 with the equation $p_d = \frac{(S/N)-5}{10}$. Since the S/N values differ according to the type of the aircraft, real world applications should take it into account as an input. The probability of detection can be calculated at every point along the path by using aircraft RCS, radar system characteristics, and the distance to the radar site. The total exposure to radar along the route can be calculated by adding up all the cost along the route.

In [4], the radar threat was pre-calculated and overlaid on each region of the search space. Orientation of the aircraft was not considered. The search tried to minimize flying through regions of radar coverage. This method is called the static method. In the dynamic method, computing probability of detection at each point in the route is very complicated. It depends on such factors as the radar cross-section of the aircraft, the aircraft's orientation (i.e. azimuth, elevation, and degree of bank or roll), and the distance of the aircraft to the radar transmitter and receiver. One of the most common simplifications is to treat the aircraft as a sphere so that the radar cross section appears the same from any angle. The approach taken in [5] used the sphere model for the aircraft radar cross-section.

In this research effort, the radar is calculated dynamically during program execution. Using a 3-D grid, threat is calculated only as each grid point is encountered. There are some advantages to this approach:

1. No additional memory is required. Only the location and the characteristics of the radar site are stored. Calculations are done as needed, and are not saved.
2. Calculations are only performed on potential route points. This can be a significant time savings over the static approach.

3. Updating the radar information involves only changing the information pertaining to the radar sites. Additions, deletions, and changes are updated in constant time.
4. Multiple RCS values can be used in the threat calculations. The aspect angle of the aircraft can be used to determine the appropriate RCS value and calculate the corresponding cost. This is especially important for planning routes for low-observable aircraft whose RCS is highly dependent on aspect angle, i.e., "stealth" characteristics.

3.2.2.3. Aircraft Model

The characteristics of multiple aircraft such as combat radius, maximum rate of climb and dive, and minimum turn radius are read from a plane file. By putting all the information necessary for aircraft calculations in to a file it is possible to have a scheme of implementing multiple aircraft for the path planning purposes. The aircraft RCS and its turn radius are two important features that should be taken into consideration while modeling the aircraft. By using the turn radius we can eliminate the paths with movements beyond the aerodynamic limits of the aircraft. There are multiple sets of aircraft characteristics for each type of aircraft, and they are implemented while checking the validity of moving from one node to the other. Therefore the code can be implemented for different aircraft types by just changing the aircraft characteristics list. The model used in this research is shown in Figure 3.1. In this model the movement of the aircraft is restricted to nine discrete forward directions. Descents and climbs are limited to 45° from horizontal.

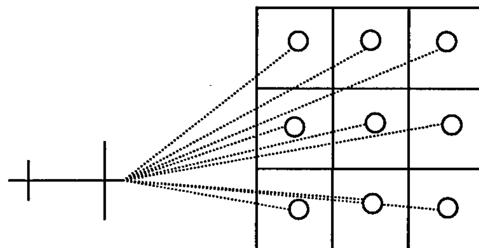


Figure 3.1. A pictorial representation of forward aircraft movement.

3.2.2.4. MRP Cost Model for Route Evaluation

Multiple factors affect the mission routing problem. To be able to solve the problem by using computers, and still be able to represent the real world conditions, the number of criteria selected must be decreased. Therefore only distance and radar exposure are selected on this research. These two criteria are selected because they can be used to calculate many of the other factors that affect the solution. In addition to probability of detection, it is important not only to know the total exposure to radar, but also the longest uninterruptible exposure to radar. For example, one 60-sec exposure is much worse than six 10-second exposures with no detection between the 10-second segments. The cost function that models these requirements follows a graph structure.

Given a set V of vertices, set E of edges, and a graph G=(V,E), let the length l_i of an edge e_i between two adjacent vertices u, v $\in$ V be defined by the function $l(u,v) = l(e_i) = (1-w_i)l_i$, $0<i\leq|E|$. Also, let a radar cost r_i for the same edge be defined by the function $r(u,v) = r(e_i) = w_i r_i$, where w_i is the weighing factor of the radar cost, and r_i is the cost of radar exposure for e_i. Let the total cost of traveling along edge e_i be $c_i = (1-w_i)l_i + w_i r_i$. Then the total cost C(s,g) of traveling along a path o N vertices from start s to goal g is simply the sum of the individual edge costs in the path as given by the following equation:

$$C(s,g) = \sum_{i=1}^{N-1} c_i = L(s,g) + R(s,g)$$

The length $l(u,v) = l_i$ is the Euclidean distance between u and v. The total length $L(s,g) = \sum(1-w_i)l_i$, and the total radar cost $R(s,g) = \sum w_i r_i$. The weighting factor w_i increases with each consecutive edge exposed to radar according to the following equations:

$$w_i = \frac{(k+1)W}{k+W}$$

where

$$k = \begin{cases} 0 & \text{if } r_i\text{-}1 = 0 \\ k+1 & \text{otherwise} \end{cases}$$

The weighting factor w_i increases from an initial value of 1 to a maximum value of W as consecutive route points exposed to radar are encountered. This has the effect of increasing the radar cost for continuous radar exposure along the route. The factor W is an initial constant that can be changed to give a "more or less" penalty for continuous radar exposure. The value of the weighting factor affects the route selection by causing the route planner to search for longer unexposed paths before reconsidering the shorter exposed path. The amount of radar coverage and the radar weighting factor determine the difficulty of this search problem. A larger factor W causes the algorithm to search more area than a smaller value for W. This value is parameterized to facilitate generating multiple routes using different weighting factors.

3.3. High Level Design

The development of a software system includes certain steps. These steps are required in order to have a scientific development. By approaching the problem scientifically, we can definitely get a better quality product. It might seem, at the beginning, that the development will take a very long time, but after going through the steps we can always backtrack, or change, or add to the code easily, whereas it would take days to modify any part of the code. Documentation is another tool that helps us. So, we should go through the scientific steps one by one, and document the achievements and failures.

Generally these steps can be divided into four phases, requirements, design, implementation, and testing [19, 13]. The requirements phase is the highest level of

abstraction, and consists of analysis and specification. At each level we should apply engineering principles and go into finer detail until we develop the implementation. Since the requirements, and high level design of the MRP are done previously by Gudaitis [6], it is mentioned briefly here.

The main purpose of this research is to find an *optimal* solution in a reasonably short amount of time. Parallel processing techniques are used to speed up the search process. The requirement in this research is to find the least cost flight path for two aircraft from a starting base to two different destination targets. The models used at this level are as follows:

V = {Vertices}
E = {Edges}
Earth = Graph(V, E)
Terrain $\subset$ Earth
Sky = Earth - Terrain
Path $p_i = [S, v_1, v_2, \ldots, v_k, G], v_k \in V, k = (|p_i| - 2)$

The models are kept general on purpose at this stage so that the formal high-level design is not dependent on the type of models selected. The refinements are made later at lower level design steps.

The primary objective of the high level design is that it provides a general view of the problem as a template. After designing the template, and selecting the algorithm around it, then the rest of the development is just making necessary refinements until the implementation is reached. A* algorithm is chosen to find the optimal solution, because with proper choice of heuristic, it performs better than all other algorithms with access to the same heuristic, and it does not have to search through the whole search space [36].

3.3.1. Pseudocode

The MRP algorithm is as follows:

1 Initialization
  (a) Read the contents of the input data/parameter files into internal data structures.
  (b) Setup different groups for different aircraft and targets.
  (c) If I am worker 0 of either search group, generate and send initial paths to other workers in that group;
  (d) Calculate an initial cost bound for the best path P_{best} to the target.
2. Perform A* search
  Begin Loop:
  (a) Remove path P_i from OPEN.
    - If P_i already on CLOSED, discard and remove another path from OPEN.
  (b) Put on CLOSED and broadcast to other workers.
  (c) Generate new paths P_j from P_i and put on OPEN.
    For each P_j do:
      i. Calculate actual distance cost to the current point.
      ii. Calculate actual radar cost to the current point.
      iii. Estimate remaining distance cost to the target.
  (d) If target reached, update P_{best} and broadcast this to other workers.
  (e) If received P_{best} info from other worker, update P_{best}.
  (f) If my OPEN list empty, request work from other workers.
  (g) Receive CLOSED path information from other workers and update my CLOSED list.
  (h) If work request received then
    - If my OPEN list not empty, send paths to requestor;
    - Else if my OPEN is empty and I am not the requestor, pass on the work request to next worker in the ring.
    - Else if I am requestor and my OPEN is empty, broadcast terminate message and exit loop; this last step occurs when the target has been reached, and all workers have emptied their OPEN lists.

End Loop.

As soon as the program starts execution, it sets up the processors that are assigned for the search process. All the processors are in MPI_COMM_WORLD communication group (There is detailed information about MPI in Appendix C). It then divides the processors into different groups depending on the number of targets. In this research the number of targets is chosen to be two. So the processors are divided into two groups, namely workers and workers2. Within these individual groups, the first processors are

assigned as the controller for their respective groups. They explore the first nodes starting from the base and distribute work evenly among the other workers. All the processors within the group including the controller itself conduct the search process.

The evaluation criteria of the nodes before they are entered into the queue are introduced, which reduces the time and space requirements. Previous work puts all the nodes into the queue first, and then compares them with the current best solution path found so far, and if they turn out to be worse, discards them. By this new scheme none of these tasks are required since the worse nodes are not even put on the queue. With this new improvement bad nodes are not inserted into the queue at all, saving O(n) time, and since they are not in the list they are not deleted from the queue as well saving O(1) time. That should have some impact on the performance.

Another issue is reading all the information required for the search. There is not much of a difference between Gudaitis' work and this one on that other than two improvements. First of all the aircraft file structure is updated to allow for multiple aircraft selection. There are two choices to achieve this objective. First one is to put different aircraft data into different files and make the program read it from there. Second method is to put all the data about different aircraft types for that mission, as specified in the ATO file, in one aircraft file and make the program read that one file. The terrain file structure is also updated to allow real world elevation data to be loaded on to the system. The home base and target locations are specified in a geocentric format using latitudes and longitudes. The code translates these values into numbered format by using the location and map starting point information. It also makes sure that the base and target locations are within the map coverage.

The algorithm increments the path length by one vertex starting with the start node and evaluating each successor vertex until the goal is reached. The cost of the path is determined by the cost function C(v). The new paths formed by adding vertices to the path are put on the OPEN list. List cost paths are taken off the OPEN list first. After we remove the paths and evaluate them we put them in the CLOSED list. It is the heuristic function that determines the order of the paths to be removed from the OPEN list. Since OPEN has the least cost path as the top entry, when the vertex equals the target then we have a least cost path from base to target. The good heuristics reduce the number of vertices to be evaluated before the target is reached.

3.3.2. Heuristic

To get the greatest efficiency, it is a must that the heuristic be monotonic in the A* algorithm [36]. As described before two criteria are used for the route evaluation, but only the distance is included in the calculation of the heuristic so that the monotonicitiy can be preserved. The distance heuristic is calculated with the Euclidean distance equation :

$$d = \sqrt{(X_P - X_T)^2 + (Y_P - Y_T)^2 + (Z_P - Z_T)^2}$$

T indicates the coordinates of the target, whereas P indicates the coordinates of the current point. This heuristic provides an exact estimate of the actual distance of paths along an axis or diagonal. The greatest error in the estimate is found in two and three dimension cases. For two dimensions, the greatest error lies somewhere between a diagonal and an axis. The heuristic estimate for two dimensions is within 8.24% of the actual shortest path on the discrete grid. Three dimensions heuristic never underestimates the actual distance by more than 12.81% [6].

3.3.3. Parallel A Algorithm Considerations in MRP*

The execution speed of serial machines is limited by the physical limitation of transmitting media. Many real world problems are too large to be handled by sequential machines for speed and memory considerations. When we parallelize an algorithm to run on parallel machines we need to consider some design issues. Those are as follows:

- Termination condition of the algorithm.
- Implementation of the OPEN list (distributed versus centralized)
- Implementation of the CLOSED list (explicit versus implicit)

3.3.3.1. Termination condition of the MRP algorithm

In the sequential version of the MRP algorithm, since the best solution found so far is at the top of the OPEN list, the optimal solution is achieved when the goal is reached. For the parallel version, on the other hand, there are p processors expanding points from the OPEN list, and each may be searching a different area of the search space. When a processor reaches a goal, there are still $p-1$ processors searching that may lead to better solutions. Therefore parallel version must delay termination until the global best solution is found. Termination occurs when processors empty their OPEN lists.

3.3.3.2. Implementation of the OPEN list (distributed versus centralized)

Each processor must access the OPEN list for each expansion of a point along the path. A centralized (global) OPEN list allows each processor access to the global best next point to expand, but may create a communication bottleneck, especially in message passing architectures. According to [25], a global OPEN list has limited scalability, and is best for shared memory systems. Distributing the OPEN list among the processors would reduce the communication overhead, but processors would be expanding only the local best that may not be the global best next point. It also has some additional problems with

consistency among Open lists, idling from uneven work distribution, and duplicate work among the processors. [25] A hybrid approach is used in [4, 5] with the best information from all processors managed by a Control Process, and the remaining information kept in local OPEN lists on each worker process. A distributed OPEN list is used in this research.

3.3.3.3. Implementation of the CLOSED list (explicit versus implicit)

In the sequential algorithm, a monotonic admissable heuristic function may prevent the need for an explicit CLOSED list. A point on OPEN that has been expanded does not need to be explicitly stored on a CLOSED list because it will never have a lower heuristic value due to its monotonicity of its heuristic function. But, it's not the case for the parallel implementation, because each processor is searching a different part of the search space. If the processors do not share information in some way, their search spaces may overlap causing some redundant work. An explicit CLOSED list may be used to reduce redundant searches between processors. In Grimm's and Droddy's work [4, 5] an explicit CLOSED list was not used. That reduces efficiency and causes execution time of the algorithm to be unacceptable for real time applications.

3.3.3.4. Load Balancing

In the previous designs, each processor was terminated one by one when it finished its searching. The old design did not have an efficient load balancing scheme for termination of the algorithm. In the new design, the work is "evenly distributed" among the processors so that there is no idle time before the target is reached. Once the target is reached, dynamic load balancing occurs when a worker empties its OPEN list. Work requests are passed along in a ring message passing design, and those workers that have paths remaining on their OPEN lists will send some of their paths to the requestors. When

a worker w_i empties its OPEN list, it requests work from its nearest logical neighbor w_{i+1}. If the neighbor w_{i+1} has paths on his OPEN list, it sends some of his paths to w_i; if not then the neighbor just passes along the request to his logical neighbor. If the request from w_i goes around the ring of processors without being replied, all OPEN lists must be empty, and the program terminates. This scheme allows each processor to work independently of the other. Synchronization among the processors is performed only when a worker receives its own work request, indicating that all the OPEN lists are empty, and the algorithm may be terminated.

*3.3.3.5. Variations of A**

Most of the variations of A* algorithm are derived to overcome the large memory requirement of the algorithm. Although A* is optimal and efficient with a monotonic heuristic, optimality is guaranteed only if all generated paths are stored on the OPEN list until they are evaluated. If interim paths are generated faster than they are evaluated, the OPEN list may grow quickly in size, and may require more memory than is available. In a recent research performed by M.Gudaitis [6], it is observed when less than four processors are used. Increasing the number of processors to eight reduced the growth of the OPEN list.

A with Iterative Deepening*

The idea for limiting storage requirements for depth first search (with backtracking) with iterative deepening can be useful in heuristic strategies. A* works in a way similar to breadth first, but the order of the nodes is not decided by their respective depth but the value of a heuristic function. A* with iterative deepening implements a depth first algorithm schema in which a value of a threshold is set at every iteration. This threshold

is used for heuristic function values. If the value of a heuristic function for a node is smaller than a given threshold, the algorithm backtracks.

In comparison with the original A* algorithm, A* with iterative deepening asymptotically examines the same number of nodes, the number of memory locations it uses is proportional to the length of a solution path, and for reducible heuristic functions it guarantees finding the optimal solution. Since only the current path is stored the total required memory is reduced. IDA* may perform as fast or faster than the A* algorithm, but it performs some redundant work across iterations, and this redundant work usually causes IDA* to have longer execution time. In the class of heuristic tree search algorithms with a reducible heuristic function, A* with iterative deepening is asymptotically optimal with regard to both the number of nodes examined and the number of memory locations used.

With parallel A* with iterative deepening, each processor searches in a disjoint search space, bounded by the cost function, according to the depth first strategy. When a processor ends searching through its part, the free part of some other processor's search space is analyzed and assigned to the idle processor. Since a search space is limited by a cost function, the first solution found by parallel A* with iterative deepening is the optimal one. The algorithm terminates if and only if the processors find a solution.

Real-Time A Algorithm*

A strategy controlling practical execution of a task is most often a real time A* algorithm (RTA*). For further analysis a node with a smaller total cost estimate is selected. RTA* is independent of the choice of strategy during the initial node of the full solution graph, but the cost of executing the strategy from a node initial at the current

phase of practical solving. The form of RTA* and A* are the same, only the cost estimate function is different.

3.4. Testing and Design of Experiments

Since this thesis research is built upon the work of Gudaitis [6], the design issues stayed similar. His work was implemented on a different platform, namely Intel Paragon, using NX communication commands. This research is implemented on a different platform using a different set of communication commands, MPI. After rehosting the code on this new platform using MPI, it is somewhat necessary to use similar test cases to be able to compare the performance of both implementations.

Gudaitis [6] conducted his experiments on the Paragon with tests of up to 20 processors. Near linear speedup, or better, was obtained for all test cases. For larger number of processors, it was found that the problem size must be increased to get the same level of performance. This shows us that the solution is scalable and that the algorithm may be implemented with larger problems which are representative of real mission scenarios.

This research has two different types of experiments. The first one is designed to be used right after rehosting the code to run on the platforms described above. This is basically the same code as the previous work with just a few small modifications. It is the code that finds the optimal route for one aircraft against one target. By having the same code run on the similar test conditions it becomes possible to evaluate the code behavior on different platforms with different communication tools. The second step is to build new features upon this rehosted code, such as incorporating real world map data, and

multiple target, multiple aircraft issues. Experiments for this version of the program are also run to evaluate the performance.

Gudaitis' work uncovered the effects of many factors. In his work, experiments are designed to measure the effects of radar against no radar scenarios; terrain versus no intervening terrain scenarios. Doing the same sets of experiments with one target version of the new code would be useless. For that reason only one of his test cases which had 15 radar sites is used to test the performance of the rehosted code. The terrain used for this test case is composed of 2x2, 4x4, 8x8, 16x16, 25x25, 32x32, 64x64, and 100x100 search space, in which there is an S-shaped mountainous terrain for the 100x100 case. By designing this scenario it is possible to determine how the code performs according to different search sizes. 15 radar sites are stationed on different parts of the terrain for 100x100 case. These experiments are good indication of the effectiveness and efficiency of the algorithm. The time values of each run of the experiments are used to calculate performance metrics that are to be used in the analysis phase. S-shaped terrain makes it easy for us to determine if the algorithm produces an effective solution. After visualizing the result with the terrain and radar data it is easy to decide if the algorithm found an optimal solution effectively by evading known threat areas, and reducing the distance as well. To evaluate if the code is efficient, execution times are used. For our case the algorithm is said to be efficient if it finds an optimal route in a reasonably short amount of time for every large input set.

The last set of experiments test the performance of last version of the code that uses two aircraft against two targets. Again the effectiveness of this code is determined by analyzing the solution against terrain and radar locations and the length of the route that

is selected as the optimal path. The base is selected to be the same for both aircraft, and different targets are assigned to each of them. Two test cases are run on this version. First one has a target that can be reached through a mountainous terrain threatened by radar sites. The other test case has a target that can be reached through a non-mountainous terrain without being exposed to radar sites. That is done on purpose to evaluate the lack of radar sites in the scenario.

3.5. Summary

In this chapter, the MRP is described in detail trying to incorporate its problem domain with a solution domain, because the first step in solving a problem is to have a thorough understanding of the problem. The methodology is discussed, which is centered around A* algorithm, and why A* is a suitable algorithm. Some of the implementation decisions are also discussed. We need to follow software engineering principles in designing a new system, because they offer a wide variety of tools that we can use to create an efficient, effective, and maintainable system. Following the introduction of the high level design, a more detailed description of the system is introduced, giving the data structure, and the algorithm details in the next chapter.

4. Low Level Design and Implementation of MRP

4.1. Introduction

This chapter goes into more detail on the issues covered in the previous chapter. Previous chapter discusses the software methodology and high level design issues of the code. The low level design and implementation of the parallelized A* algorithm from the previous chapter is the focus of this chapter. The low level design and implementation associated with it does not differ from the previous work done by Gudaitis [6]. The "C" programming language is used again, but this time instead of NX commands for communication MPI commands are used. Gudaitis implemented his code for single aircraft against single target. This research is to investigate the multiple aircraft against multiple target destinations. Another added feature is the terrain data used for the algorithm. In the old design the elevation data were in a Cartesian format, which did not reflected any real world information. In this research Geocentric format is used for location reference that allows real world map elevation data to be loaded on to the system.

4.2. Low Level Design

The low level design defines the structure of the program used to perform the MRP. In the older version there used to be a distinction between a host program and a node processors. The job of the host program was to get file names for data input/output, load node programs, get all the results, display the results and kill programs. The host was an interface between the user and the host processors. In the rest of the program execution host would stay idle waiting for the other processors finish their search. Since there are a limited number of processors in both the AFIT Bimodal Cluster of PCs (10 machines

usable on this research at the time of the research) and AFIT Cluster of Workstations (6 machines) the host program is also incorporated in the search process. Also having a host processor send this information to the node processors and then receive messages from them would increase the execution times. By having all the processors as worker processors it is possible to make each processor read its own set of data. This is also a useful feature while implementing the multiple aircraft, multiple target design. Below are the diagrams showing the dynamic model of the algorithm and initialization part of the program.

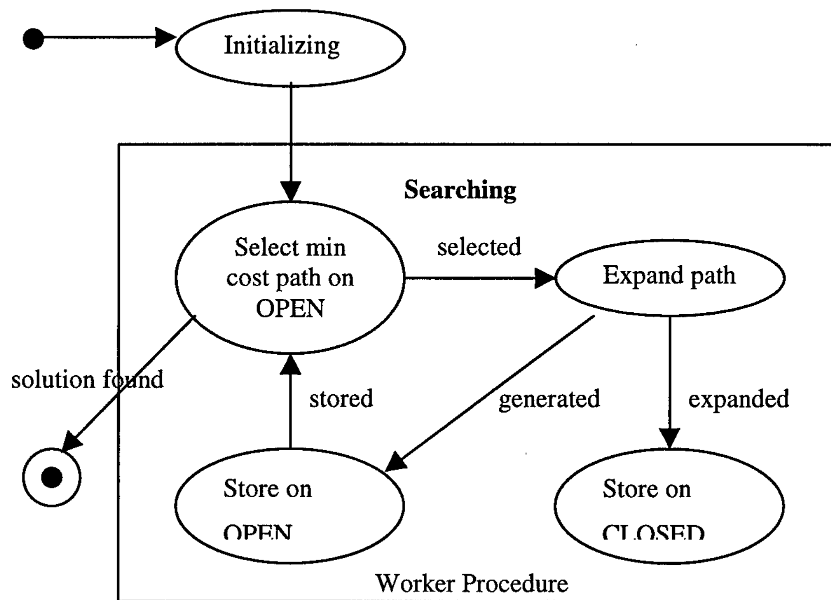


Figure 4.1. A dynamic model of the algorithm showing the execution hierarchy.

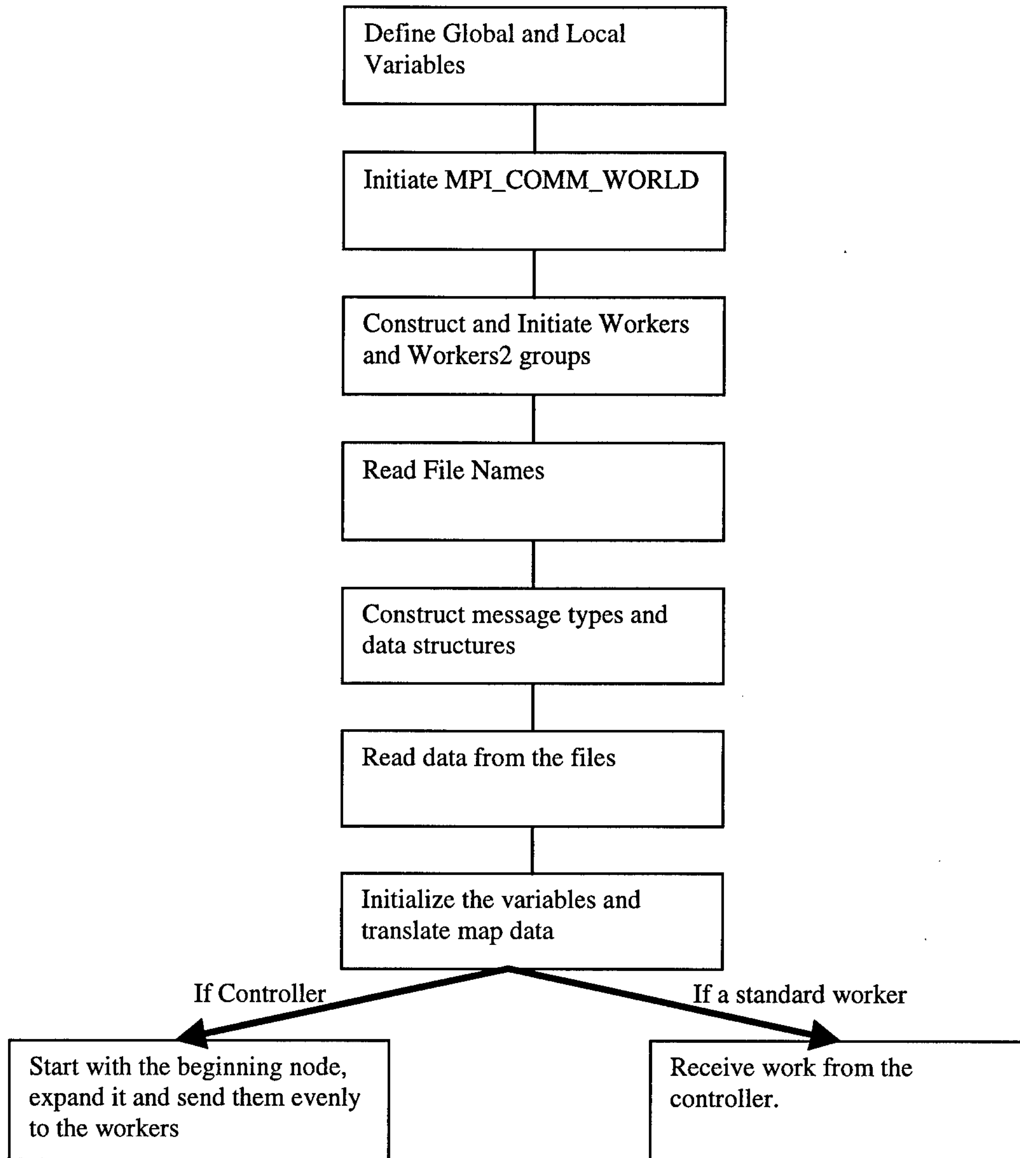


Figure 4.2. The initialization part of the program.

Still there is a controller processor that first initiates the search process by expanding the initial nodes on the path and sending even amount of work to all the processors and saving some for itself as well. As the program starts execution the processors are included in the MPI_COMM_WORLD communication group. Then the processors are divided into n groups where n is the number of targets to be found (n is 2 in this case).

The groups are workers and workers2, and each processor in one group can communicate with its group members using its group communicator and all the other processors using MPI_COMM_WORLD communicator. One processor cannot communicate with a member of any other group other than the global MPI_COMM_WORLD.

4.2.1. Parallel Architectures

There are two platforms that this research is designed upon. The AFIT Cluster of Workstations (COWs) comprises of 6 Sun Solaris machines at 170 and 200 MHz with two connection types. They can either connect using Myrinet connection at a speed of 1 Gbps, or Ethernet using 10 Mbps. the operating system on these machines is Unix. The other platform is the AFIT Bimodal Cluster of PCs (PPCs). They are Intel Pentium II PCs with processors ranging from 200 MHz to 400 MHz. Their connection type is also an Ethernet connection with a speed of 100 Mbps. They use Red Hat Linux as the operating system. The machines on both platforms are connected to each other, i.e., they can send and receive messages to and from each other.

4.2.2. The program Design

As discussed before in chapters II and III, criteria for distance travelled and radar exposure are combined into a single additive cost function for A* route evaluation. Each node along the path is given a cost value according to its terrain height, radar signal density, and distance. The A* search algorithm uses the additive cost function $f'=g+h'$, where g is the cost of reaching end of the route and h' is the projected cost of reaching the goal from the end of the route. At the end of the program the smallest cost path is selected as the optimal route.

A* explores the route which seems to be the best solution based on the cost function above. The OPEN list is a priority queue that contains all the routes, in ascending order of cost, which are under examination. When the program starts the OPEN list is initialized with the starting location. The top entry has the lowest projected cost and it is the most promising one to have the optimal solution. Then this entry is removed from the list and explored by finding all the allowable children of this entry. These children nodes are also saved in the OPEN list in order of increasing cost. This process is repeated until a solution is found. This research uses distributed local OPEN lists and a global CLOSED list.

A child is defined as those locations, in the three dimensional search space, which are adjacent to the given location. Adjacency is defined as being one grid location away, including a diagonal line. The directions directly above or below are excluded. Then for any given location there is a maximum of 24 neighboring locations (8 at the same altitude, 8 at the lower altitude, 8 at the higher altitude). Each child is first checked to be sure that it is within the boundaries of the terrain data, and also the aircraft's mission coverage area.

There are different functions and different procedures for each task. The function hierarchy is shown in the Figures 4.3 and 4.4 below.

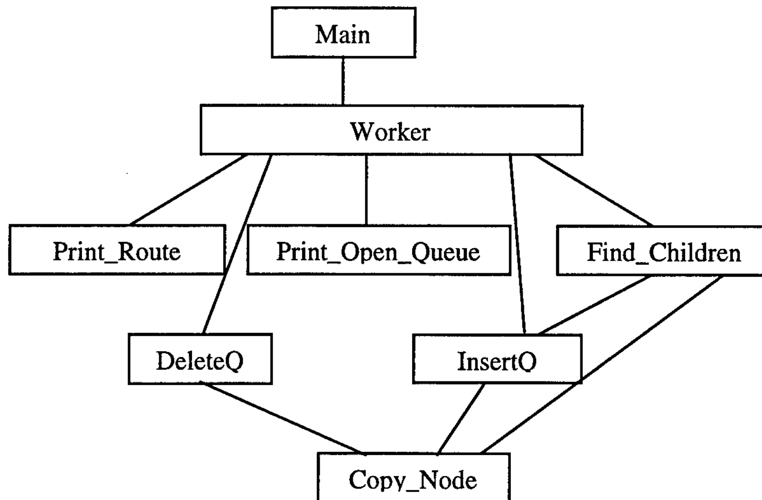


Figure 4.3. Software Function Hierarchy. Part a.

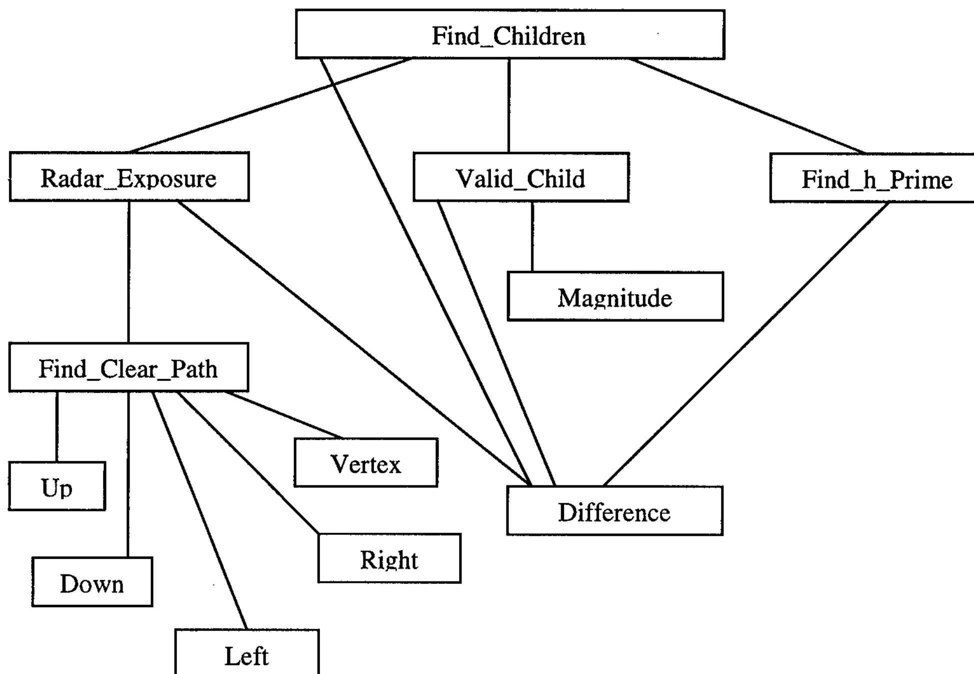


Figure 4.4. Software Function Hierarchy. Part b.

InsertQ and DeleteQ : These functions are used to put or remove generated paths from the OPEN queue. It has to search through the OPEN list until it finds a path with equal or greater cost value, and then insert the new path. This yields an O(k) order of work since the function has to search k <=maxQsize items before storing the new path. On the other hand, DeleteQ removes the least-cost path from the queue. Since the least-cost path is always at the top of the queue, this is an O(1) order of work.

Copy_node: This function is used to copy the contents of one path into the other. Each node represents a path implemented as an array of 200 gridpoints (X, Y, Z coordinates). This function copies only the non-empty entries. So, it takes O(k) order of time where k<=200.

Print_Open_Queue: This function is controlled by a flag called QUEUE_FLAG that is set at the beginning of the code. Its purpose is to help the user debug the code. In Normal operation the flag is set to False, but when it is turned on it displays the contents of the OPEN list at certain points.

Print_Route: It prints the path that is given as the parameter, along with its costs associated with each point in the path, and total distance and radar costs for the optimal path that is found.

Find_children: This function uses some other functions to find the children of a given path, and insert them into the OPEN list. The children of a path is found by concatenating its neighbors to the end of the path.

Valid_child: This function takes the path from the find_children function to determine if it is an acceptable child of that path. It basically checks the validity of some set of rules as indicated below:

1. If it is within the set X, Y, and Z limits of the terrain representation.
2. If it is within the altitude limitations set in the ATO file.
3. If it is within the specified aircraft combat radius.
4. If it is within the aircraft turning capability.

In case any of the above rules is not met, then the path is called invalid and discarded.

Find_h_prime: This function determines the h', the heuristic function which estimates the remaining cost of reaching the target from the current point. It is calculated using the straight distance to the target and the constant weighting of radar detection:

h_prime = (1.0 - WEIGHT_RADAR) * Dist;

Radar_Exposure: This function is used to calculate the radar exposure of the aircraft at a given point along the path. It is determined by the aircraft RCS, the radar location and characteristics, and the distance from the given point to the radar. The radar design has used the fact that all radar systems have their own minimum detectable angles. It is detected by the surrounding terrain and the physical construction of the radar. By knowing the minimum angle limitation of a radar site it is possible for MRP algorithm to find routes that pass below this specific angle to avoid exposure to the radar site. Since one other technique to evade radars is to use terrain masking it is important to find if the aircraft is visible to the radar, i.e., within line of sight and there is no terrain intervention. Find_clear_path is used to perform this function. If the aircraft is not within the line of sight to a radar then its exposure is zero, and does not need to be calculated. The overall radar exposure at one point is a cumulative function with respect to the number of radars, causing O(r) order of complexity.

Find_clear_path, Vertex, Up, Down, Left, Right : The first function is implemented by using a ray-tracing algorithm from Glassner [9]. The other five functions are designed in the previous work [6] and are not used in this application.

Distance: It calculates the straight line distance from one point to the other. It is used by other functions.

Magnitude: This function is used to calculate the vector magnitude of a given X, Y, Z components of a vector.

4.3. Design Issues

There are some issues that have to be observed while designing the code. These are discussed under this topic.

4.3.1. Data Structures

The design of data structures is an important task. There are many ways to represent data, but not all of them are as effective. Bad data structures may cause the application to take lots of memory in the first place. The second issue is that the related data should be together so that it makes sense to the user or the system developer. Still another issue is to organize the data structures accordingly to minimize the number of messaging between processors. This is especially important to reduce the time spent on communication. The header file compiled with the program has the constants that specify the maximum size of the array data structures. That way it is easy to manipulate these constants to fit the data structures into the available memory.

Terrain Representation.

The terrain is represented by latitude (X), longitude (Y), and elevation (Z) values in a three dimensional grid fashion. The scale factors for all the axes are set to 100 feet within

the program execution. This is new and different from the previous work. The reason for that is the introduction of geocentric location format instead of Cartesian format. Locations in the ATO file are given in latitude/longitude format, and since one second corresponds to approximately 100 feet, the scale factor is set to 100 feet. This gives us a nice resolution along with an ease of calculation and manipulation. That way the locations are stated up to second sensitivity, and then these values are translated into numerical format so that the designed algorithm can use them. For the design of experiments a 100 x 100 elevation data is stored on each processor as 4-byte integers. That takes up 100 x 100 x 4 =40000 bytes = 39 KB (1KB = 1024 bytes). This elevation data is used for path selection to meet the minimum altitude criteria, and for determining radar exposure on a given point.

Path Record:

The heart of the program is to find the optimal path from the base to the assigned targets. The paths generated during the course of program execution are stored on OPEN list. The data structure to keep these paths is given below. It is the same data structure used in Gudaitis' work.

```
typedef struct {
    int      number;                        /* Number of entries in the route     */
    US       x [MAX_PATH_LENGTH+1];         /* Vector of x locations              */
    US       y [MAX_PATH_LENGTH+1];         /* Vector of y locations              */
    US       z [MAX_PATH_LENGTH+1];         /* Vector of z locations              */
    int      VectorX;                       /* Direction vector in x direction    */
    int      VectorY;                       /* Direction vector in y direction    */
    int      VectorZ;                       /* Direction vector in z direction    */
    float    distance;                      /* Cumulative distance of the route*/
    float    radar;                         /* Cumulative radar detection cost */
    float    g;                             /* Cost of the given route            */
    float    cost;                          /* Calculated cost (f') of route      */
    int      link;                          /* Forward Links for  OPEN list       */
} PATH;
```

MAX_PATH_LENGTH is set to 200 in the previous research and is not changed. The link value is used to point to the next path stored in the OPEN list. Vector values show where the next movement direction would be. The x, y, z arrays contain point coordinates, and stored as unsigned short (US) integers that require two bytes. Since integer and float values require 4 bytes, the total storage required per path is (3 x 201 x 2) + (9 x 4) = 1242 bytes.

OPEN list: The OPEN list is a priority queue where all the generated paths are stored in an increasing cost order. The maximum size of OPEN list is 12000 paths which requires 14.2 MB of memory as an upper limit.

The insertQ operation is used to store generated paths in to the list in an increasing cost (f') order with the least cost paths to the front of the queue. Before putting the path into the list, insertQ compares it with the best solution found so far. If the path is worse then it is discarded without being inserted into the list. In the old design the paths are compared with the best found so far in deleteQ operation, which took extra space and time. The deleteQ removes the front of the list in O(1) time. DeleteQ also checks the path with the goal value to see if a solution is reached. If so, then best found solution is updated and this solution is sent to other processors as well.

CLOSED List: The CLOSED list implemented in this research is an explicit type as was the case for the older version. It is a 3-D array that keeps the cost values of given locations. It is shown before that keeping a global CLOSED list reduces the execution time of the program greatly by preventing re-exploration of the nodes [6]. The total memory requirement for 100 x 100 x 25 terrain elevation data is approximately 1 Mbyte. Updating the cost value of any grid point is O(1).

4.3.2. Input Data Files

In order to have a modular structure for the input data instead of scattering it inside the program, related data are put into the same files. This gives us flexibility to change the structure of any given data type. Especially this feature of the program made it easy for this research to change the code according to the objectives. To add a new aircraft type for multiple aircraft issue, the only thing to do is to change or add this extra information into the aircraft file. The rest is just to adjust the code in order to handle this new data file structure. Otherwise it would be much more difficult and complicated.

Terrain file: The terrain file is composed of four integer values plus the set of integer representation of elevation (mean sea level) for each of the X, Y vertices in meters. The first three integers in the file are x, y, z dimension values to set up the search space. The fourth one is the scale factor.

Radar file: This research used the same radar characteristics as the previous work. It contains the following information;

- Number of radar sites for this problem
- The transmitter power
- The transmitting antenna gain
- The receiving antenna gain
- The transmitted signal wavelength
- For each radar site:
    - ✓ The X coordinate
    - ✓ The Y coordinate
    - ✓ The Z coordinate
    - ✓ The minimum effective angle (azimuth) for the radar

This information is used by the radar exposure to calculate radar costs for each point along the path.

ATO File: This is a revised version of the previous one for multiple aircraft against multiple targets using geocentric format for locations. The following information is stored:

- The mission designator
- Coordinates of the lower right corner of the map
- Coordinates of the base location
- Coordinates of the first target location
- Altitudes of the base and the first target
- Minimum and maximum flight altitudes
- Altitude type
- Coordinates and altitude of the second target.

Plane file: This file has the characteristics of the aircraft to be used in the mission. Since there are two aircraft, there are also two sets of aircraft data in the plane file. The information kept in this file is as follows:

- Aircraft radar cross-section (RCS)
- Minimum turn angle
- Maximum altitude at which the aircraft can operate
- Maximum combat radius of the aircraft

For the scope of this research the RCS value is kept simple, whereas it is a very complex issue in reality. The simplification made here assumes the aircraft as a sphere for RCS purposes.

Algorithm file: There are two values kept in this file:

- Multiplication factor for radar cost.
- Weighting of radar detection.

The criteria weight used for distance traveled and heuristic calculation is one minus the weight assigned to radar.

4.4. Implementation Details

Some of the implementation issues such as coding standards, some file structures and the control structure, have been changed. MPI is used instead of NX commands for communication among the processors. This necessitates a change in the coding structure as well as the control of the program execution. Also the structure of location representation is changed from Cartesian to geocentric format. Necessary changes in the file structures have been made to accommodate this change. Multiple aircraft against multiple targets especially required a considerable change in the control structure of the code.

4.4.1. MPI and its Suitability

MPI was designed for high performance on massively parallel machines, workstation clusters, and heterogeneous networks [24]. MPI is primarily for SPMD/MIMD. The most common implementations available on line as well are LAM (Local Area Multicomputer), MPICH, and CHIMP. MPI-2 is also available with its added functionality and new features. Some of MPI-2's new features are dynamic process spawning, client/server functionality, one-sided communication, C++ bindings, and MPI-I/O. Log files that contain a history of a parallel computation can be very valuable in understanding a parallel program. The upshot and nupshot programs, provided in the MPICH and MPI-F implementations, may be used to view log files. Vampir, an MPI profiling tool is used to visualize the communication and computation time ratios of the code execution.

MPI is not a specific product or a compiler specification, but a message passing model for interprocess communication. Message passing is an approach that makes the

exchange of data cooperative [22, 23]. Data must both be explicitly sent and received. An advantage is that any change in the receiver's memory is made with the receiver's participation. Another operation type is one-sided which enables remote memory reads and writes. MPI is flexible and easy to use for simple applications. But it may get very complicated for bigger implementations with different data structures and communication patterns.

MPI enables us to have structured buffers and derived data types. This feature is greatly utilized in this research. A data type is derived for the path variables. This new data type made it very easy and compact to send and receive messages containing path information. Otherwise each field would have to be sent/received individually adding up to the execution time and complexity of the program. There are also different modes of sending/receiving messages, such as, normal (blocking and non-blocking), synchronous, ready (to allow access to fast protocols), and buffered modes. In this research two different versions of code are designed, one with normal blocking and the other with immediate asynchronous messaging.

There are many reasons why MPI is chosen over other interfaces like PVM (Parallel Virtual Machine):

MPI has more than one freely available, quality implementation.

There are at least LAM, MPICH, CHIMP, and MPI-2. The choice of development tools is not coupled to the programming interface which gives MPI the ability to operate on different platforms.

MPI has full asynchronous communication.

Immediate send and receive operations can fully overlap computation.

MPI defines a 3rd party profiling mechanism.

A tool builder can extract profile information from MPI applications by supplying the MPI standard profile interface in a separate library, without ever having access to the source code of the main implementation. This is the approach that Vampir profiling tool uses. A few sample experiments were run after Vampir has been linked. It gives a very detailed information about where most of the time is spent. That way the user can figure out the areas of bottlenecks in the code, and direct his/her attention towards that area to get a better performance.

MPI groups are solid, efficient, and deterministic.

Group membership is static. There are no race conditions caused by processes independently entering and leaving a group. New group formation is collective and group membership information is distributed, not centralized. This feature of MPI made it very easy to design multiple aircraft and multiple target case. After the code is initialized the processors form two different groups depending on their order in the MPI_COMM_WORLD. Then all the communication is held within the individual groups. Intergroup communication is not possible, but since each processor is also the member of the global communication group, they can communicate with all the other processors.

MPI is totally portable.

MPI can recompile and run on any implementation. With virtual topologies and efficient buffer management, for example, an application moving from a cluster to an MPP could even expect good performance.

MPI efficiently manages message buffers.

Messages are sent and received from user data structures, not from staging buffers within the communication library. Buffering may, in some cases, be totally avoided.

MPI synchronization protects the user from 3rd party software.

All communication within a particular group of processes is marked with an extra synchronization variable, allocated by the system. Independent software products within the same process do not have to worry about allocating message tags.

MPI can efficiently program MPP and clusters.

A virtual topology reflecting the communication pattern of the application can be associated with a group of processes. An MPP implementation of MPI could use that information to match processes to processors in a way that optimizes communication paths.

MPI is formally specified.

Implementations have to live up to a published document of precise semantics.

MPI is a standard.

Its features and behavior were arrived at by consensus in an open forum. It can change only by the same process.

4.4.2. Multiple Aircraft against Multiple Target Case

The necessary changes that must be made in the data and file structures have already been discussed above in 4.3.1. and 4.3.2. The code also has to be changed and redesigned to be able to manipulate these new data types with the algorithm being used. First of all, new communication groups must be created using MPI commands. This is shown in the Appendix C.

When the program starts executing, MPI_Init command initializes the MPI communication group. MPI commands have meaning only between the MPI_Init and MPI_Finalize commands. After initialization of the global communication group, a dummy communicator, world, is used to create the new groups. Before doing that, again a dummy group, world_group, is put to work. Within the two for loops, some of the processors are excluded from the world_group and the rest of the processors comprised the new worker_group. As a last step the groups are given their names with MPI_Comm_create commands, and the dummies are freed.

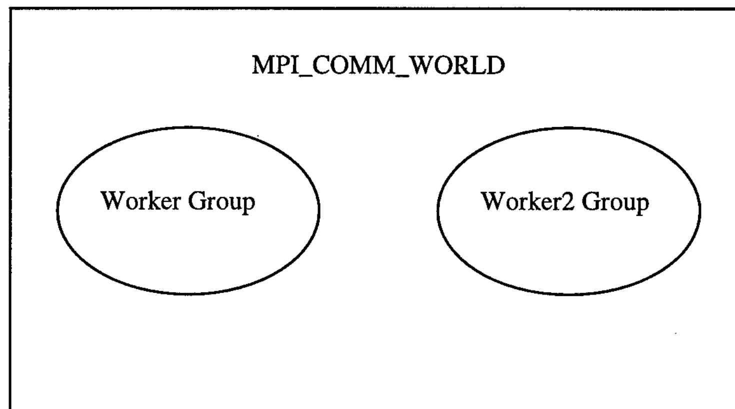


Figure 4.5: Representation of the groups.

Figure 4.5 shows the hierarchy of the groups. The processors in MPI_COMM_WORLD can communicate with each other all the time using MPI_COMM_WORLD as the communicator irrespective of their affliction with other groups. That is actually a good feature, and has great applicability in this research. The processors in either new groups may only communicate within the group using their new group communicators, worker or worker2.

For the MRP application for this research, after the new communicator groups are formed the work is divided into two. This is done by having processors read their own related data from the files. Within each group the processor with processor id zero is set to be the controller to start the search process by generating and sending initial paths evenly among all the processors in the group. After that the controller also becomes one of the workers and looses its controller feature for the rest of the code. When the code is in the data reading process, the distinction is made between the two groups according to their ids within the global communicator group. One of the files is the aircraft file where data about two different aircraft types are stored. The second file is the ATO file where two different target types are specified using geocentric location information. When the processors have their data they are ready to start the search process within their new groups. As mentioned before, since communicating with other processors is also available, processors can communicate with other group members using MPI_COMM_WORLD. That way it is possible to check if the two aircraft are too close to each other at one given time by exchanging path data. For this thesis research this conflict prevention is not handled in the code, but left as a future development project. The insertion of time can be implemented by having another field in the path entry to keep the time value. It can be calculated by putting into consideration the speed of the aircraft, wind speed, aircraft g, etc. Then at given times the processors exchange their data and check to see if the two paths get too close to each other and take proper action if that is a hazard for the optimal solution. In this work, instead of a conflict prevention scheme, a conflict detection scheme is developed by visualizing the terrain and the route data.

4.5. Summary

In this chapter some of the design issues are analyzed into more detail. The program design and the sequence of functions and their relations are shown. The functions are explained in detail. The architecture that the experiments are to be conducted is introduced. The data structures that parallel A* algorithm uses is shown in detail. Most of the data structures are untouched in this research, and left as it was in the previous work done by Gudaitis [6], and Droddy [5]. File formats, on the other hand, are changed in order to introduce multiple aircraft against multiple target version of the code. The OPEN and CLOSED lists are discussed; their role has been explained. MPI is the means of communication in this research, and some information is given about it, along with the reasons why it is suitable for this application. Also the technique to create two different groups within a global group with MPI commands is shown. This chapter covered the low level design issues related to the MRP. Next chapter is about experimental testing, the results, analysis of the results, conclusion and recommendations for future research efforts.

5. Experimental Testing, Results and Analysis

5.1. Introduction

This chapter defines a set of MRP experiments and examines the results. The experiments are designed to see if the design goals are met. The code is expected to run properly and produce effective and efficient results. The experiments are conducted to validate these two objectives:

- The efficiency is obtained if the code can produce results in a reasonably short amount of time for "large" data sets.
- Effectiveness is obtained when the solution of the code is found to be optimal.

The first set of experiments is conducted on the single target version of the rehosted code. After that, the multiple aircraft against multiple targets design is developed and tested. The following sections give a detailed description of the experiments. Before analyzing the performance, the design of experiments and performance metrics are described.

5.2. Design of Experiments

First set of experiments is designed to evaluate the performance of the rehosted code with a single aircraft against a single target. Different search sizes are evaluated starting from 16x16 terrain up to 100x100 terrain. The idea behind this design is to evaluate and determine the performance of the code with different search sizes as the number of processors is increased. Then the same code is tested with the 100x100 terrain several times on different platforms. This 100x100 terrain with an S-shaped mountainous area is used as a base terrain on which the test cases are built upon. The same type of aircraft is used for all test cases, as is the case for the radar sites. There are 15 radar sites and their locations are not changed. All the variables are kept unchanged from one iteration of the

code to the other in order to evaluate the algorithm performance by keeping other variables as static.

Second set of experiments is designed to be conducted with the new design of multiple aircraft against multiple targets scenario. Again the aircraft and radar characteristics are not changed. Both aircraft types are assumed to be the same. The base 100x100 terrain model is used. The only change is the addition of another target. This second target is selected at a place where there is an easy route without terrain intervention and less radar threats from the base to the target. The algorithm is expected to find an optimal solution for this target faster than it would for the original target.

5.3. Metrics

The metrics that are used in this research are the execution time, the speedup of parallel implementation over sequential version, the efficiency and the optimality of the selected route.

Execution Time

Execution time is one of the main metrics of evaluation. The speedup and efficiency metrics are gathered by using execution times. The main requirement is to reduce the overall execution time to get a reasonable time to implement this in real world scenarios. The factors affecting execution time are the computation complexity, the amount of communication among the processors, idle waiting times that must be cured by load balancing schemes, and the search space.

Speedup

Speedup is a measurement of the gain in performance by implementing an algorithm in parallel over its sequential version. The advantage of parallel systems are dividing the

same work to be executed by different processors, and possibly decreasing the overall execution time [10, 11]. It is defined by: $S_p = T_s / T_p$

where S_p is the achieved speedup by the parallel implementation with p processors. T_s is the time running the sequential version, and T_p is the time of the parallel implementation. There is more information about speedup in section 2.2.2.

Efficiency

Efficiency is a measure of the fraction of time for which a processor is usefully employed; it is defined as the ratio of speedup to the number of processors [10]. Efficiency shows the degree of processor utilization. In the ideal case speedup is equal to the number of processors p, and efficiency is equal to 1. In practice, speedup is less than p, and efficiency is between 0 and 1, and defined by: $E_p = S_p / p$

Optimality of Solution

One of the ways of determining optimality is to run experiments enough number of times and to check to see if the solution is the same for all. Some other metrics can be used to determine the optimality such as the number of paths generated, expanded, the number of times that the OPEN list gets full and has to delete 50 of its worst paths. The A* algorithm guarantees to find an optimal solution if it searches the whole search space. If the code discards nodes, then this may degrade the effectiveness of the algorithm. When trying to determine the optimality, the number of nodes expanded by different processors is a good indication of optimality. If the values are close to each other then the load balancing among the processors is also achieved, whereas difference in the number of nodes expanded is an indication that some processors did less work than the others leading to a badly implemented load balancing. Analyzing the OPEN list gives a very

good indication about the program execution. If the nodes are duplicated a lot within a processor, then there is definitely something wrong with the execution of the local search process. The use of a global CLOSED list should prevent that. When the queue gets full, 50 of the worst paths are discarded each time. There is a limit set to 12,000 for the maximum number of nodes that can be stored in the OPEN list. The reason for that is to be able to keep the memory need at a reasonable level. But, the A* search algorithm is guaranteed to give an optimal solution if all the paths in the list are evaluated [36]. Then this fact degrades the optimality.

Another problem with discarding these nodes is the greater possibility of cycles. When a node is discarded it can be found again and put into the queue, discarded again leading to a cycle that affects the execution time. The approaches that can be taken to reduce the number of paths to be stored in the OPEN list is either to use a good heuristic that guesses the cost very close to the real cost or eliminating the duplicates. Using a good heuristic directs the search process along the optimal route, and that way unnecessary paths are not put in the list at all. But there is a cost associated with that, i.e., the complexity of the computation. That is why a simple heuristic function is used in this research, and it leads the search process to a shorter distance path with less radar exposure. The other goal, eliminating the duplicates is done by a global CLOSED list. Gudaitis' work [6] showed that without a CLOSED list the average number of duplication was 50. Having a local CLOSED list prevented the duplicates within the OPEN lists of individual processors, but duplication was observed when the lists of different processors are compared. Although each processor starts its search process in a

different direction, the heuristic guides each processor in the same direction towards the target.

5.4. Input Data

This section discusses the input data that is used for code evaluation. The terrain data, radar data, and ATO data are presented. Plane data gives technical and performance data about the aircraft types selected and is chosen to be standard for all cases.

Terrain Data

The terrain data used in this research is the one used in Gudaitis' work. It gives the user to test different aspects of the program execution, such as if the algorithm is making an effort for terrain masking to evade radar exposure, if it is taking a route away from terrain and radar, if it is taking into consideration the distance, etc. It is an S-shaped mountainous terrain with canyons that an algorithm might choose to exploit. Figure 5.1 shows this terrain

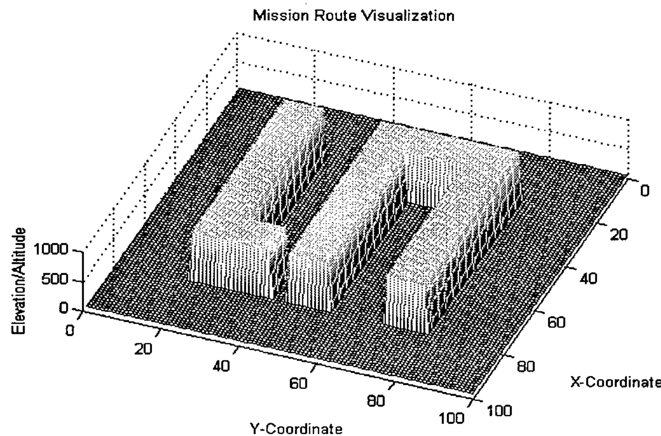


Figure 5.1. Visualization of the terrain used for the test cases.

Radar Data

The radar input has 15 radar sites distributed over the terrain described above. Their technical specifications are considered to be the same for all the sites. Since the effects of radar have been investigated in detail in [6] this research has only concentrated on the case with 15 radars. In Figure 5.2, the locations of the sites are visualized.

ATO Data

Mission related data is given here. On the first set of experiments as mentioned before, only the single aircraft against single target case is tested. Therefore mission data for this set of experiments include only the mission designator, location of the home base,

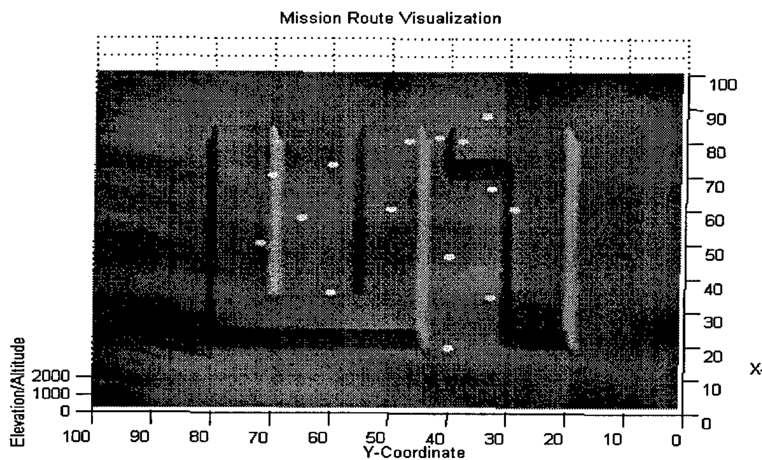


Figure 5.2. The location of the radar sites on the terrain.

and target are given along with altitude characteristics. The start point of the map is also given in geocentric format, which corresponds to 100 x 100 point in the original terrain file. Base altitude is 2*100(scale factor) = 200m and the target altitude is 3*100=300m. The start point for the map is chosen to be N10°10'10'', E20°20'20'' (100, 100 in the original). The base is locate at N10°10'20'', E20°20'48'' (72 90 2). The target is at a location N10°11'15'', E20°21'00'' (60 35 3). Minimum altitude is 100' and maximum

altitude is set to 4000'. Altitude types are MSL. On the second set of experiments the data for the second target is also added. It is located at N10°10'50'', E20°21'20'' (40 60 3).

5.5. Experimental Testing

After the inputs are given the experiments are run on different platforms. The experiments produced the results as explained in this section. When the single target case is tested, first couple of tests resulted in the solution shown in Figure 5.3, but then this result did not show up on any of the experiments on any of the platforms. This is the solution path found on Gudaitis' work as well. This solution path had a combined cost of 28K, whereas the new solution had only 19K. New solution path is shown in Figure 5.4.

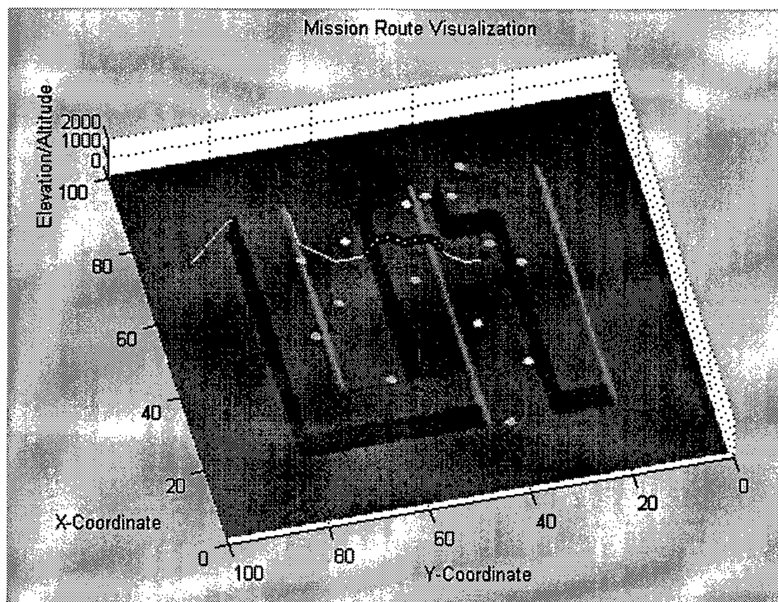


Figure 5.3. The solution found when the code is first implemented without any modifications. The light brown dots represent the radar sites, and the red dots along the path represent exposure to radar.

Since the results were mostly consistent, the solutions are regarded as optimal. More than 200 tests are run, and it showed up 3-4 times. The only reason that a better solution is found is the elimination of the paths before they are put into the queue, instead of eliminating them once they are put into the queue if they turn out to be useless.

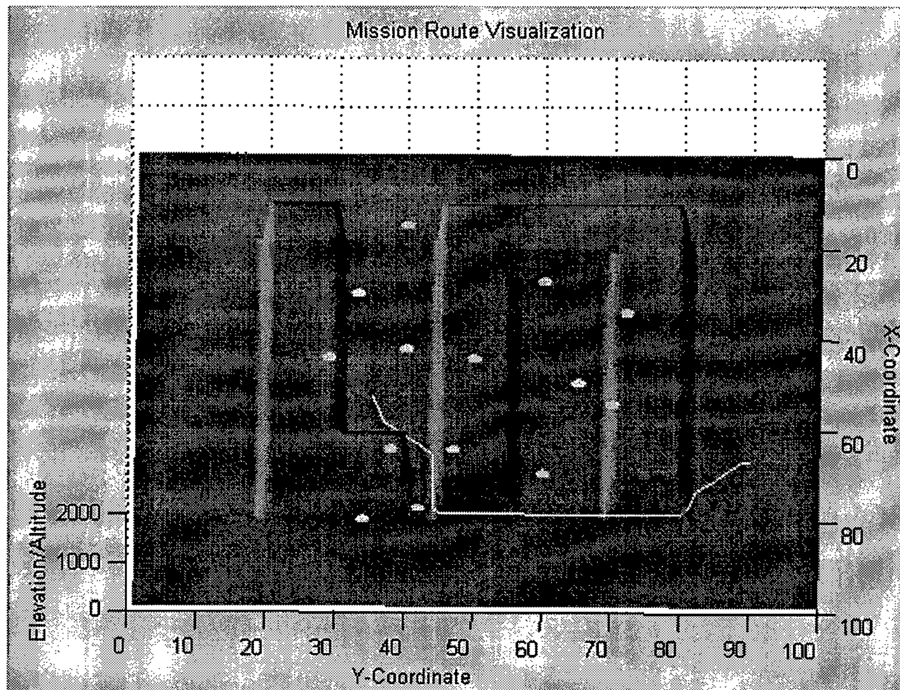


Figure 5.4. The solution path found by the new code.

Figure 5.5 is the solution found when the multiple targets case is experimented. On this multiple case the second target is in an area that can be found easily because it is not protected as much, and there are more choices to pick while searching the target.

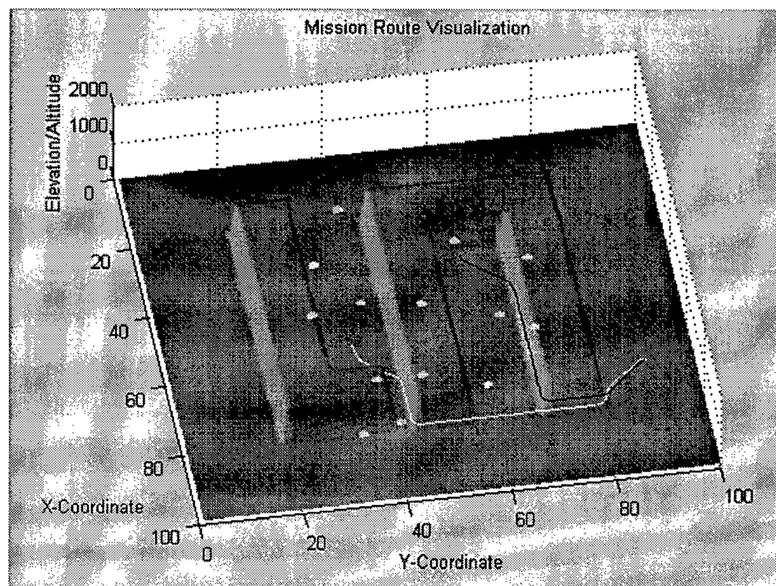


Figure 5.5. Multiple Aircraft against Multiple Targets.

The Matlab code to visualize the terrain, radar, and the path is in the Appendix D.

Results

The program is run several times on each platform to see if it would produce the same result each time, and to get a consistent execution time for each platform. The execution times did not differ from one iteration of the program to the other as long as the mission parameters were kept the same. For that reason statistics are not shown on the charts. On the ABCs platform only 8 real processors could be used, and the extra processors are duplicate processes on the same processor. The Sun Solaris system has 6 interconnected machines on the COWs. So, any addition to that number is a duplicate on any of the 6 processors. The duplicate processors share the resources of the processor, and for this reason when there are 8 processors, the execution sometimes gets smaller, but not as much as expected, and sometimes it increases.

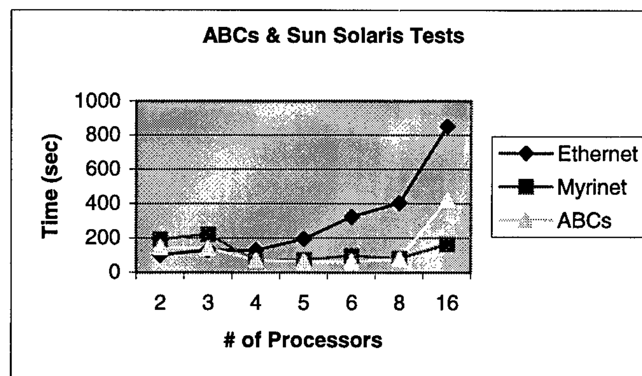


Figure 5.6. The Results of experiments on different platforms.

Figure 5.6 shows the results of the tests run on ABCs, and Sun Solaris system using both Ethernet and Myrinet connection types. Since Myrinet has the fastest bandwidth, it is expected to get smaller execution times. ABCs have faster processors and relatively fast Ethernet connection. When the figure is analyzed it is clear that the execution times of ABCs and Myrinet are going down, whereas that of Ethernet is going up. The fact that

the execution times are going up for all with 16 processors does not reflect the truth for the reason explained previously.

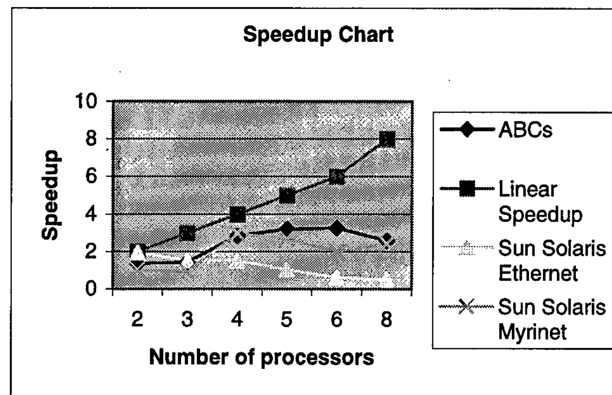


Figure 5.7. The speedup chart for different platforms.

The speedup diagram also shows that the best performance is realized with Myrinet. As expected the speedup of the Ethernet connection on Suns is decreasing as the number of processors increase. Also, speedup for all platforms is below the linear speedup. The reason for that is the fact that the increase in the number of processors brings with itself the increased cost of communication. Since the communication is very slow due to hardware restrictions in the Ethernet on Suns, 10 Mbps, it gets the performance down. For the other platforms the speed of communication is faster, and has a positive effect on the performance when we disregard the duplicate processor factor.

When the efficiency chart is analyzed in Figure 5.8, it is seen that the efficiency peaks at 4 processors for both the ABCs and the Myrinet connection at approximately 0.72. The highest efficiency for the Ethernet connection is achieved with two processors, since the communication latency is a bottleneck for that platform. After that value the efficiency is

in a dive as the number of processors increase. With a larger search space the computation outperforms the communication efficiency.

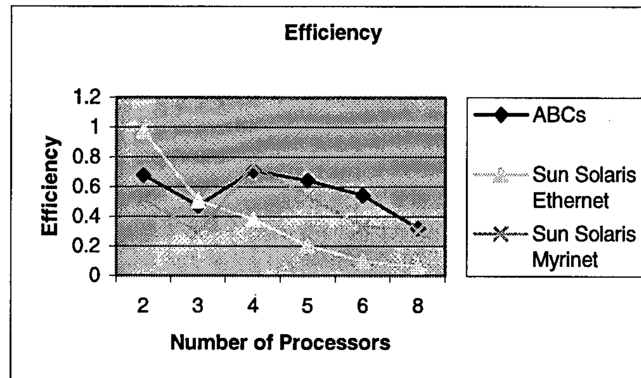


Figure 5.8. The Efficiency Chart for different platforms.

To better grasp the effect of the size of the search space a set of experiments is conducted on the Myrinet connection, since it turned out to give the "best" performance. The search space is increased from 16 x 16 up to 100 x 100, and the results are seen on Figure 5.9. To summarize the chart, up to 6 processors, the increase in the search space also increased the execution times.

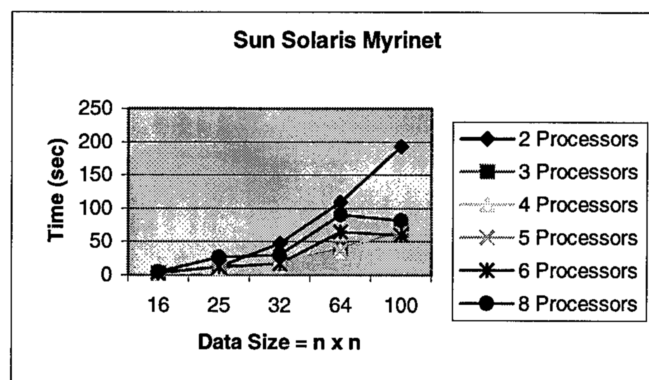


Figure 5.9. The results of the experiments with different search sizes on Myrinet.

But, for the 6 and 8 processors case, even if the 2 of 8 processors are duplicates, when the search space size increases from 64 x 64 to 100 x 100 the performance gets better. One of the reasons for that is the fact that there is a greater possibility of finding an

optimal solution with greater number of processors. The second reason is that as the search size increases, the computation complexity outperforms the communication complexity. When there are more processors this feature is a positive effect on the performance.

The time it takes for the algorithm to find an optimal solution depends on different factors. To show this effect in the multiple target design, one of the targets, the second target, is chosen as a location that is not protected by radars as much as the second target. The second target is also along a non-mountainous path, which makes it easy for the algorithm to find. When the execution times it takes for the new algorithm to find the optimal paths for both of the targets are compared by analyzing Figures 5.10 and 5.11, it is very intuitive to realize that finding the original target takes more than twice the time it takes for the other target. Even the slowest platform, Sun Solaris with Ethernet, has values less than half of the fastest execution time on Myrinet (150 versus 334 seconds).

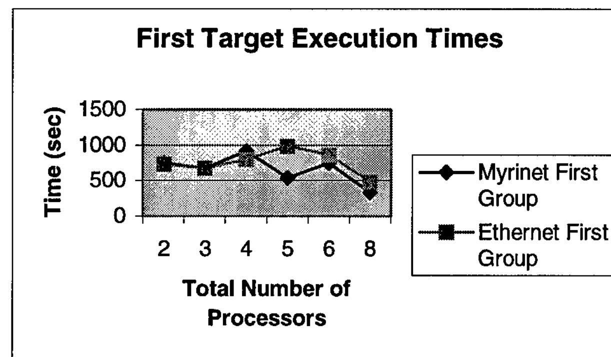


Figure 5.10. Multiple Targets experiment with the execution times of the first target.

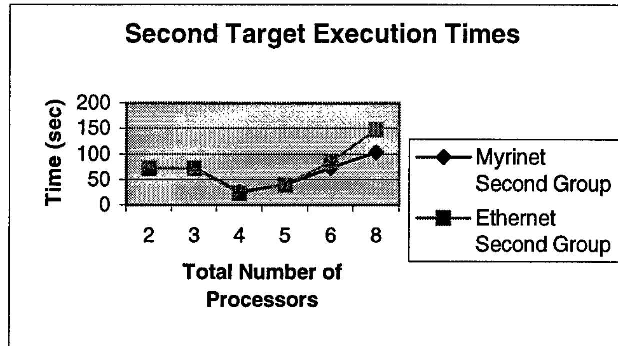


Figure 5.11. Multiple Targets experiment with the execution times of the second target.

There are some features of the program execution of the new design. First of all there is an extra work in the program execution to set up the new communication groups. When the program is run with *n* processors, at most *n/2* of them are assigned for each group. So, the number of processors working on the same search is half of the single target design. The peak performance is achieved with 4, 5 processors in the single target design. In order to achieve the same performance with the new design 4, 5 processors should work on the same target, which means that there must be at least 8,10 processors for two targets so that the algorithm can allocate 4, 5 processors to each target search. Another issue is there are two search processes going on at the same time. It means that the program overhead is doubled. Even if the processors are mostly independent from each other (as long as they do not make any inter group communication) there is some overhead setting up the communication, getting the initial nodes, expanding them sending them to the processors, waiting for them to respond that they are ready to execute, etc. Since there are two searches, this overhead is doubled. During the same execution two groups are using the same communication bandwidth. This feature also incurs some delays in the communication. These explain why finding the path from the same home base to the same target (first target) takes more time than that of the single target design.

The experiments also show that the computation complexity for the first target is more than its communication overhead. This is not the case for the second target. Since it doesn't take much to calculate an optimal path to the second target, the dominating factor is the communication latency for this case. These findings are supported on the charts of Figure 5.10 and 5.11. For the second target the execution times are best when the total number of processors is between 3 and 5; from then on the performance degrades as the number of processors are increased. This is not the case for the first target calculation. Its execution time stays almost stabile up to 6 processors. With 8 processors its performance gets better. If the 2 of 8 processors were not duplicates, the execution times would be less.

When the speedup chart in Figure 5.12 and the efficiency chart in Figure 5.13 for the first target are analyzed one can see that the speedup is below linear, and the efficiency is lower than 0.15 and going down as more processors are involved. But, after 5 processors the speedup and efficiency starts to go up. It is also clear that Myrinet gets better speedup and efficiency compared to the Ethernet when the processors are increased in number.

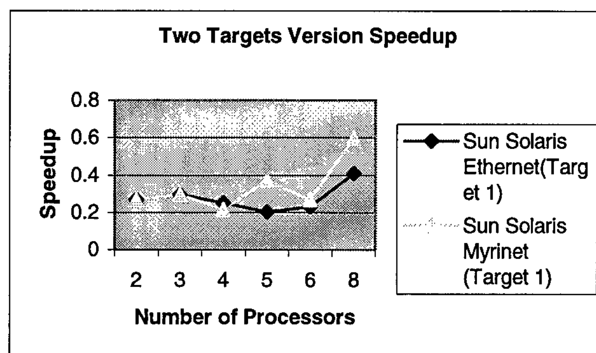


Figure 5.12. Multiple target case speedup chart.

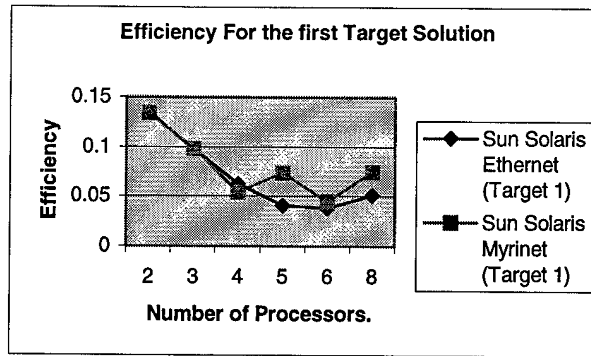


Figure 5.13. The Efficiency for multiple target experiment.

The load of idle waiting time and the communication time may degrade any benefit to partitioning the problem among *p* processors. Communication cost is getting higher as the number of processors increase, because each worker broadcasts its CLOSED cost value for each path evaluated, and receives (p-1) CLOSED values for *p* processors during each path evaluation cycle. With a very large search space this overhead may be overcome by the gain in the computation with more processors.

In the guidance of the results collected fro this research effort, it can be concluded that the program performs effectively and efficiently. That can also be asserted for large number of aircraft and targets as well, since the program only divides the work among the processors. If there are enough processors so that each search effort is conducted with multiple processors then the performance should turn out to be effective and efficient.

5.6. Analysis of Program Execution

The program is analyzed with an MPI profiling tool, Vampir, on AFIT Cluster of PCs. The makefile is changed accordingly and the libraries of Vampir are also included in the compilation process. There is no need to write extra code inside the program to manipulate the Vampir.

When the program starts its execution the Vampir creates a trace file and monitors all activity concentrating on the message passing activity more. When the program finishes its execution Vampir also stops monitoring and stores everything in a file. As mentioned before there is no need for extra commands within the code, but Vampir enables the users to have the option of manipulating the monitoring process by inserting some instructions within the code itself. All tracing activity starts after MPI_Init initializes the communication and finishes before MPI_Finalize terminates all message passing activities. Users can turn on or off the trace in this interval. Vampir has another good feature that it does not take up much from the execution time of the original code. The diagrams are of high utility to analyze the MRP code tested using Vampir. On none of these diagrams has tracing time become significant.

Having a profiling tool such as Vampir is vital while testing a parallel implementation. It shows the designer the problem areas. Then it is the designer's work to eliminate the bottleneck and improve the design to a better one. Vampir supports a graphical interface that the user can use and manipulate different views of the traced file. Below are some of the views collected for parallel MRP.

Figure 5.14 shows the general computation and MPI ratios of five processors. The one on the right shows the ratio of the count of instructions on five processors, and it is 50%. The one on the right shows the execution time ratios, and MPI takes 15% of total execution time. Tracing time is negligible compared to other execution times.

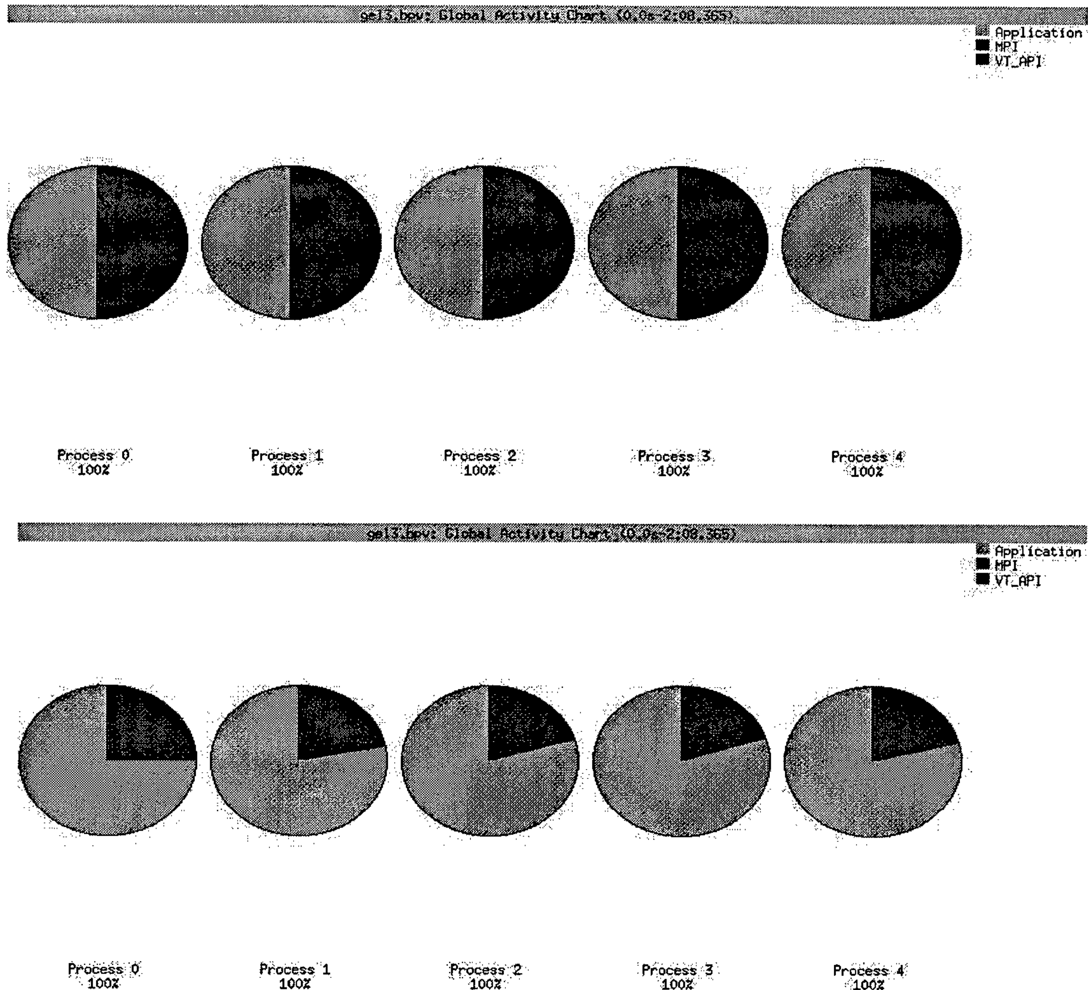


Figure 5.14. The Distribution message passing (Red) compared to computation(Green). The upper figure shows the instruction count (50% MPI), the lower one shows the time distribution (15% MPI).

The reason that MPI is only 15% of total execution time and still there are lots of message passing going on in the program is that the complexity of computation is very big for this problem case, and is outperforming the communication complexity. Experiments with smaller search spaces, or with targets that do not take too long to find showed that up to 90% of the total execution time was taken by MPI. But for either case, the parts that take the most are the same. MPI_Iprobe command takes the most in either case.

Figure 5.15 shows the count and execution times of all the MPI instructions. Other than the four most common instructions, the rest are negligible. Since there is an MPI_Wait, and MPI_Recv command associated with each MPI_Isend instruction, their counts are the same, whereas their execution times vary. MPI_Iprobe is the largest latency in the code taking 22% of total program. After that send operation is taking up most of the time. One of the reasons that this is so is the broadcast commands are implemented as an iteration of send commands, since they required a tag to be received and distinguished from the other messages being received.

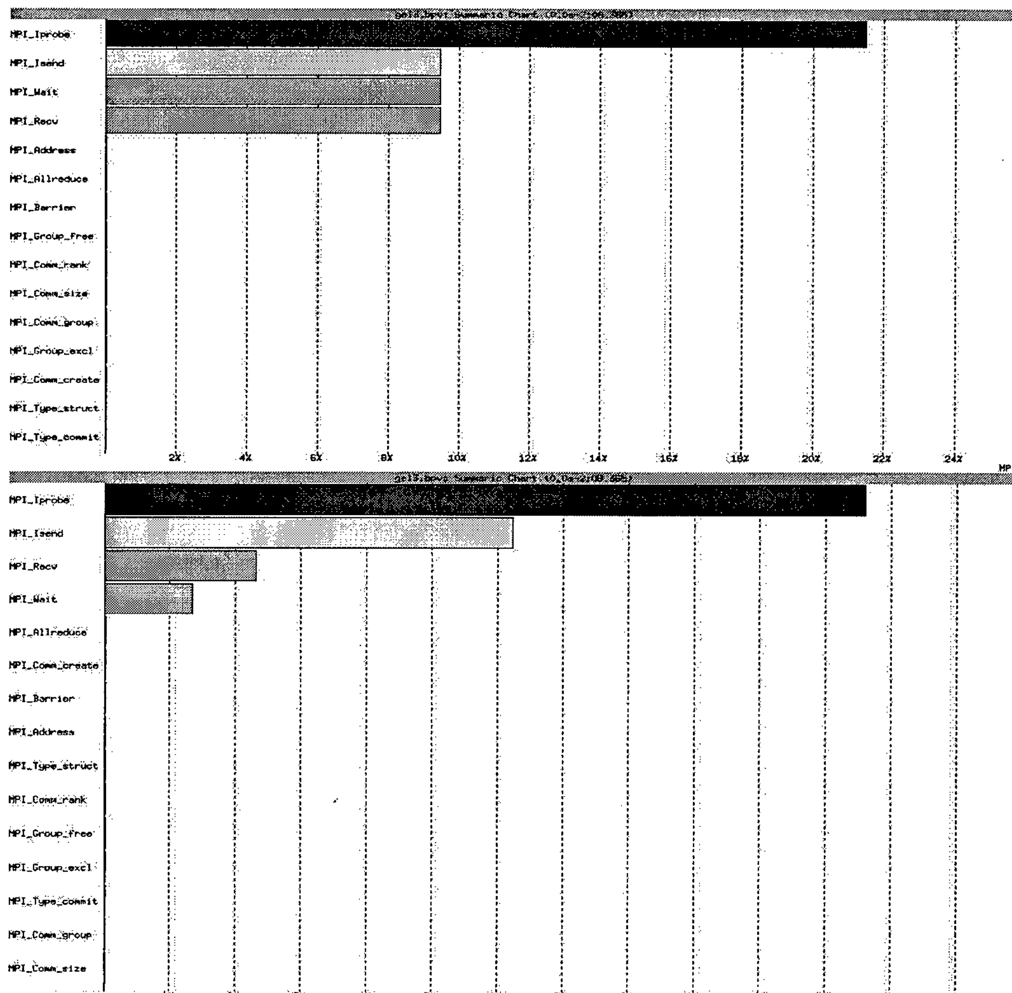


Figure 5.15. The distribution of MPI commands countwise (on the left) and timewise (on the right). The Red one is MPI_Iprobe, the second one is the MPI_Isend, Third one is MPI_Recv, and the last one is MPI_Wait.

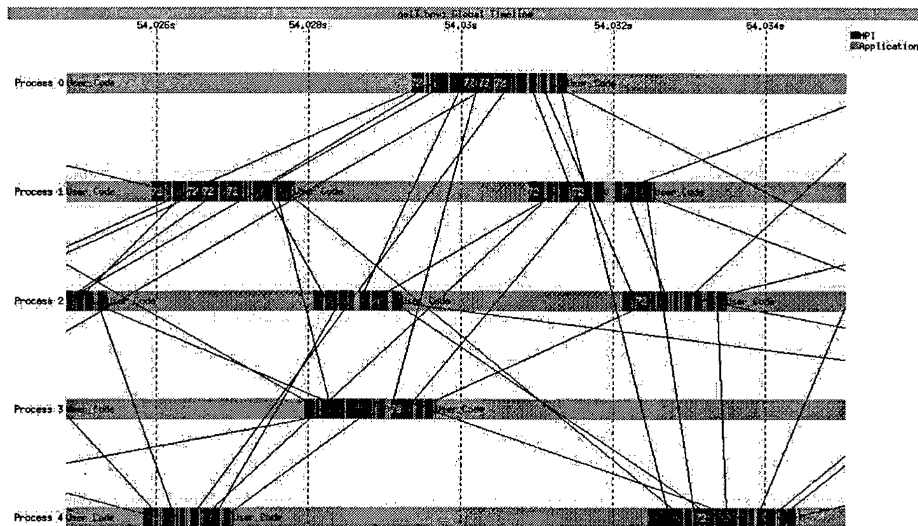


Figure 5.16. A typical Vampir snapshot showing the message passing between processors.

Figure 5.16 is a snapshot of an instance from Vampir. The green parts show the non-message passing activity, and the red parts are MPI activity with the code or name of the activity and the direction of the activity. The lines represent messages between processors. There are many other views and statistics of the whole course of program execution.

5.7. Summary

This chapter covered the design of experiments, the metrics used to evaluate the performance, the results of the experiments and the analysis of program execution. The results confirm that the goal is reached, i.e. an efficient and an effective MRP is developed. The objectives set forth are also achieved. The results show that the code finds optimal solution in a reasonably short amount of time. The analysis shows that MPI takes around 15% of total execution time. When the program is analyzed with a smaller search space MPI may take up to 90% of total execution time. Therefore, number of processors should be selected accordingly for the different search sizes. Next chapter gives conclusion remarks.

6. Conclusion and Recommendations

6.1. Conclusion

The goal of this research effort was to develop an effective and efficient MRP system. This goal is achieved through the design, implementation, extensive testing, and analysis of a functional MRP system. All of the objectives that were set in section 1.2 are achieved in this research. Objectives include rehosting the NX code with MPI and designing a new system that would calculate paths for multiple aircraft towards multiple targets. Selection of an optimal solution within a reasonably short amount of time is crucial to the efficiency of the system. A set of experiments is designed to gather performance metrics and evaluate the system designed.

The existing code is translated into an MPI code, and several experiments are run to evaluate these metrics. The parallel A* design and implementation is improved with the addition of Geocentric coordinates format. Also integrated is deletion of the nodes, before they are put into the OPEN list, if they have worse cost values than that of the solutions found so far. The specific rehosting of the code was successful. It was not a task of taking one NX instruction and substituting a counter part of it in MPI. It required the designer to completely have an understanding of the NX code and its data structure, as well as a thorough comprehension of both NX and MPI constructs. The fact that NX and MPI had different formats and different features makes it difficult to map the communication structure. A new communication and data structure is designed and implemented in order to accomplish the rehosting.

A set of appropriate experiments and metrics were defined. The results of this first set of experiments revealed that optimality is achieved. Each iteration of the test cases

resulted in the same path, as it should in a deterministic search algorithm. After visualizing the results on figures illustrating the terrain, radar sites, and the path, the calculated path is easy to visualize. Results show that the A* algorithm chose an optimal solution, considering both the terrain and the radar sites. This was expected since the exposure to radar would add to the total cost. In order for an aircraft not to be seen by radar sites it must be far from the radar site, or it must be below its minimum detection angle, or it must have an intervening terrain like a mountain between itself and the radar site. So the algorithm favored a longer route with the previously discussed features over a direct route with a higher radar cost. The code also favored a route over a plain terrain rather than over a mountain, to mask a radar site, since this would make it susceptible for exposure. Changing weights in the objective function can favor radar detection over distance or vice versa.

When designing such a system it is very important to lower the execution time in order to have a feasible real world MRP system. The results of first set of experiments showed that time criteria objective is achieved as well. The code produced results in an acceptable time limit with an acceptable speedup as the number of processors are increased. Speedups up to three are observed, and the reason that higher speedups are not observed is the high amount of execution time that message passing constitutes as the number of processors increase in the A* algorithm.

The above experiments are for a single target case with a standard search space (100 by 100 by 25). Another set of experiments are conducted to test the scalability of the program against different sizes of search space. It is seen that as the search space increases in size the experiments with up to 5 processors produced an increasing amount

of execution time. The increased size of the search space incurred more computation, and communication overhead added on top of that. But for 6 and 8 processors the computation gain well outperformed the communication loss and performed better execution times, compared to the theoretical performance, as the search space is increased. This validated the fact that new design performed better with more processors as the size of the search space is increased.

A MRP solution algorithm for multiple aircraft against multiple targets has been designed and implemented. The extended A* algorithm is executed and tested with a new set of experiments for the achievement of the objectives. Results for two aircraft against two targets showed that the code calculates two optimal routes for two targets effectively and efficiently. It found the routes within a very short amount of time, even though the processors were divided into two groups for different missions. The first target took as low as 23 seconds, and the second target took as low as 334 seconds to calculate. The difference comes from the distance between the base and the target, the radar coverage on the route, the terrain along the path etc. Since the resources and work is divided into two separate groups, the execution times for this multiple aircraft case took more than the single aircraft case to calculate the same target. It took 61 seconds for 5 processors to calculate the same target on the single target design, and stayed around that value with 6 and 8 processors. In the multiple target design in order to have 5 processors working on the same target at least 10 processors total would be needed. Since there were only 6 real processors, only 3 of them could work on the same target.

The code is analyzed in detail, on cluster of PCs, by using an MPI profiling tool, Vampir. Thus it is easy to determine what takes the most amount of the execution time. Then, there is a design of Matlab code (given in appendix D) that gives us a nice visualization of the terrain, the radar sites, and the optimal path that our code finds. There is also considerable effort on ways to have multiple aircraft with multiple targets.

The algorithm performs well with the new data format for the terrain location identification. The new design uses geocentric format with latitudes and longitudes to point to a location on a map. The interface of this format with an existing visualization tool using real world map data is left for future research efforts.

6.2. Recommendations for Future Research

Research is a continuous process. As seen with this research investigation as well, the work is built upon previous efforts which was built upon another. One of the improvements to the existing code could include incorporating a time concept. When the two or more aircraft going to their own targets get very close to each other either or both of them might have to take some action. This can be done only if it is known that two aircraft are close to each other at a given time. Neither aircraft has to take action if they are flying over a point at different times.

Another issue is to add multi distributed agents into the A* algorithm design. Distributed Agents are highly utilized in many areas, and can be implemented in this search process as well. This feature may also increase the efficiency of the code by decreasing the size of the explicit search space.

The radar model used in this research is a simple one, and is not a comprehensive model for a real time system. The distinction between monostatic, and bistatic radars can

be studied and an improved model can be included in the search process. Also probability of failure, probability of being hit by an overflying aircraft, probability of being jammed can be added. The Radar Cross Section (RCS) of the aircraft can be made more realistic. Instead of a spherical model in which the aircraft has the same RCS value on each side, more realistic approaches can be investigated. When making these improvements it has to be kept in mind that each of the improvements adds up to the complexity of the code. So, there has to be some kind of tradeoff between complexity and performance.

Aircraft data is currently quite simple. Characteristics of certain aircraft may be simulated with greater accuracy. Presently the aircraft data consists of its RCS value, its maximum turn angle, maximum combat radius and maximum ceiling. In order to have a more realistic model, its thrust/lift ratio can be added to its maneuverability calculations. This value can also be used to statistically determine probability that an aircraft can survive a missile shot with its maneuverability. The aircraft could also have different RCS values for different configurations, and for its different position in reference to the radar site.

The heuristic used favors the distance against radar detection. Different weights could be employed. More realistic heuristics can be used. Since heuristic affects the direction of the search process this might lead the processors to a better direction reducing the execution time. The algorithm should also take into consideration the fact that there are aircraft types that have very low radar reflections. A route that is not suitable for one aircraft type might be very suitable for this type of aircraft. Another issue is the addition of multiple objectives. A new design may be developed to enhance the existing code with multi-objective feature.

The algorithm can be improved. The implementation of OPEN list, the CLOSED list, and the A* algorithm that is used are all open areas. Applying some decomposition schemes into the algorithm can also reduce the communication time. Also as stated in earlier researches, instead of giving one optimal solution, the algorithm may be changed to give a set of optimal solutions. That approach gives flexibility to the pilot according to some other dictating criteria. Some other criteria can also be added in the evaluation of the routes such as, weather or fuel constraints. Some statistics can be added in the program. For example a target might have been destroyed before with some probability, and then the program might take actions and find the second target on its mission from current position.

A graphical interface can be added. The user can act with the program execution by clicking the icons or pull down menus. If the program is improved to allow changing parameters during the program execution, the user can change some of the parameters such as the target, or the radar location, and tests the course of action that the program takes. The real world map elevation data can be visualized with an existing tool. Some of the existing visualization tools that may be used are Advanced Visual Systems (AVS), EnSight, Geographic Resources Analysis Support System (GRASS), Noesys, FalconView, TerraVision, IBM Open Visualization Data Explorer (OpenDX).

The above presented areas of future research are the least, but not the last.

6.3. Summary

This research continues the previous work of Grimm [4], Droddy [5], and Gudaitis [6]. The models designed by them have been the basis for this research, and a new extended design has been generated. Multiple aircraft against multiple targets are added into the model. Having this feature enables the utilization of interaction between multiple missions. Introduction of the geocentric data format is a feature that brings the code one step closer to a real world MRP. There are still many areas of research and investigation to be worked upon. The ultimate goal of this research is a real world operational system that can be used to plan a mission route or set of routes either in an operations and planning headquarters or in a cockpit.

7. Appendix A - Parallel Processing

7.1. Introduction

In this section parallel processing and some important issues, such as different models, and different architectures are covered.

7.2. Parallel Processing

The following sections gives you an insight into parallel processing and gets you familiar with definition, scope, network topologies, metrics, communication, and routing techniques for parallel computers.

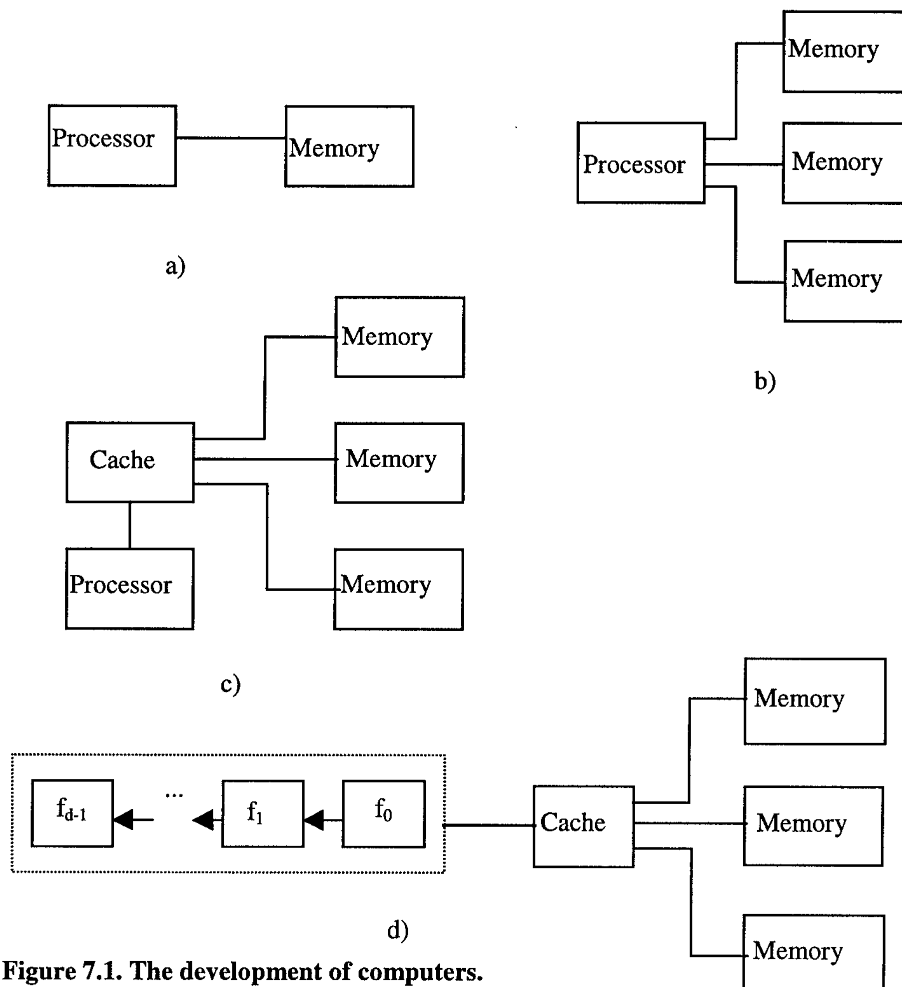


Figure 7.1. The development of computers.

a) A simple sequential computer. b) With memory interleaving. c) Memory interleaving and cache. d) A pipelined processor with d stages.

7.2.1. Models of Parallel Computers

Parallel-computing models are developed at different stages, as seen in Figure 6.1. The first machine model is called Von Neumann model, and comprised of a processor, memory unit and a connection between the two. It does not take lots of engineering skills to guess that the bottleneck was at this connection, memory-to-processor bottleneck. To compensate for that memory interleaving techniques are developed. This increased the performance, but still it was not fast. So, memory-caching techniques are developed. This on-chip or off-chip caches were actually memory units with very high access rates, parallel with its cost. Pipelining technology followed right after that in which different instruction of a program can be pipelined to be processed concurrently. For example, in this model the processor may fetch the next instruction while something is being read or written onto the memory.

7.2.2. Flynn's Taxonomy

Michael Flynn introduced a model for machine types in 1966 based on different types of data and instruction sets used. SISD is a single instruction stream, single data stream model and uses uniprocessor. SIMD is a single instruction stream, multiple data streams. They are used for special purposes, mostly in data parallel applications. They have specially designed CPUs for their specific functions. MISD is multiple instruction streams, single data stream, and none exist at this present time. MIMD is multiple instruction streams, multiple data streams. They are widely in use as a general-purpose parallel architecture today, and are very easy to scale. SPMD is single program , multiple data. Each processor executes the same program. They can execute same or different instructions on same or different data based on processor ID.

7.2.3. An Idealized Parallel Computer Model Parallel Random Access Machine (PRAM)

There are p processors with communication latency of zero. This model also has a global memory of unbound size. Memory accesses are uniform to all processors and the latency is regarded as zero. It is basically a synchronous shared-memory MIMD computer with unlimited bandwidth. There are four different types of PRAM according to how they handle reads and writes:

- EREW(Exclusive-Read, Exclusive-Write)

  Weakest PRAM, minimum concurrency
- CREW(Concurrent-Read, Exclusive-Write)

  Close to reality
- ERCW (Exclusive-Read, Concurrent-Write)

  Non-sensible.
- CRCW (Concurrent-Read, Concurrent -Write)

  Most powerful, fastest.

7.2.4. Dynamic Interconnection Networks

Crossbar Switching Networks

It is a non-blocking kind of architecture. There are p processors connected to *b* memory banks (b>=p). As *p* is increased complexity of network grows $\Omega(p^2)$.

Figure 6.2. shows us that this architecture is not scalable in terms of cost, but since all processors can have direct communication with any memory banks it is scalable in terms of performance.

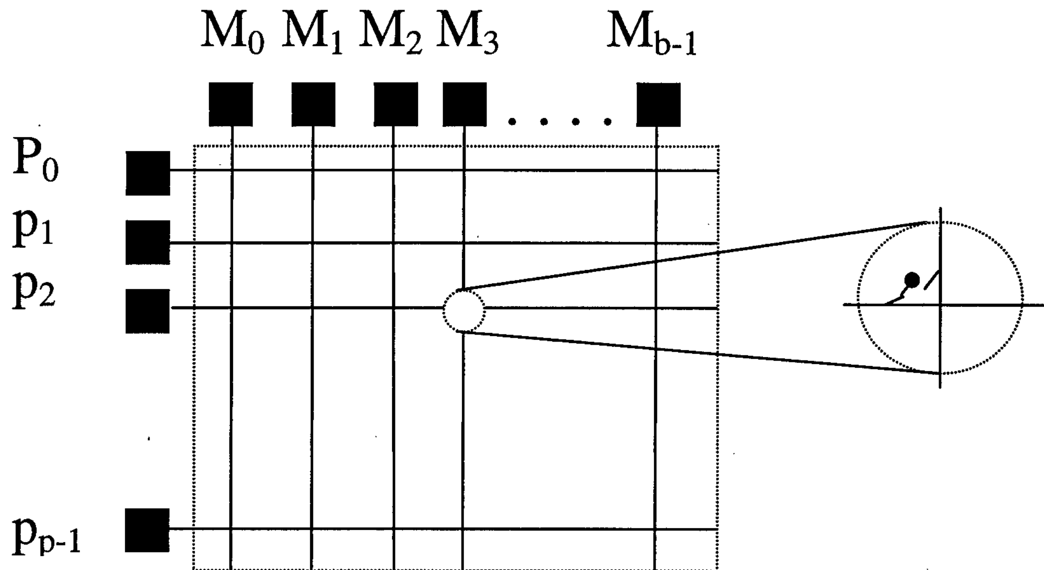


Figure 7.2. A representation of a crossbar switching network.

Bus based Networks

This architecture is very simple to build, and supports uniform access to shared memory. Bus contention is a big problem of this architecture. To alleviate bus bottlenecks, caching techniques are used as seen in Figure 6.3. Bus based networks are scalable in terms of cost, but not scalable in terms of performance.

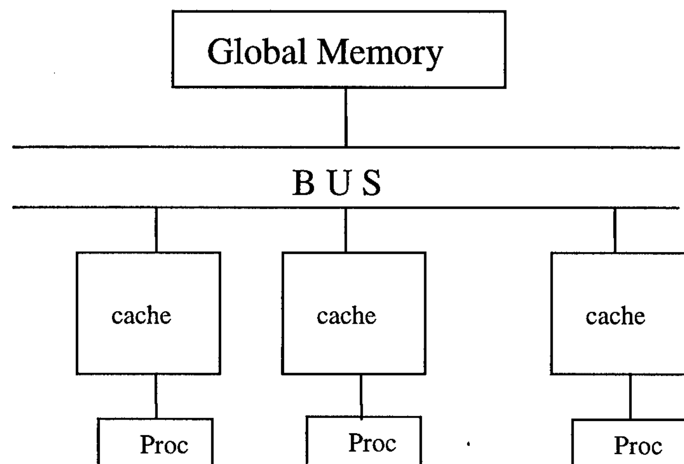


Figure 7.3. The representation of caching on a multiprocessor platform.

Multistage Interconnection Networks

This system is more performance scalable than bus, more cost scalable than X-bar architectures. Omega network is an example of Multistage networks with Log(p) stages. It is also a blocking type of system. Its overall cost is Θ(plogp). Below is a diagram of such a network in Figure 6.4.

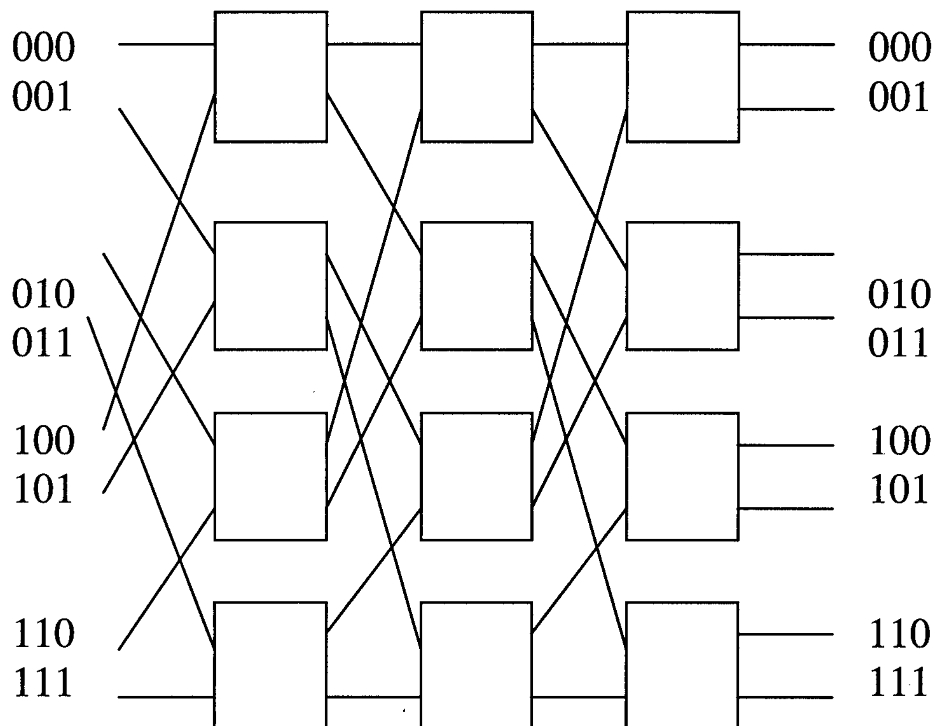


Figure 7.4. A representation of a multistage interconnection network.

7.2.5. Static Interconnection Networks

These are usually used by Message Passing architectures. Figure 6.5 shows different kinds of static interconnection networks.

Completely Connected Network

It resembles crossbar architecture. It is faster but the cost to get that performance is very high, since it requires a direct connection between all the nodes. It has a non-

blocking characteristic. It can support communication over multiple channels, which increases performance

Star-Connected Network

Central processor is the bottleneck, because it routes all the communication between other processors.

Linear Array and Ring

Linear array is an array of processors connected to each other without its end point processors directly connected. Ring is a linear array with a wraparound.

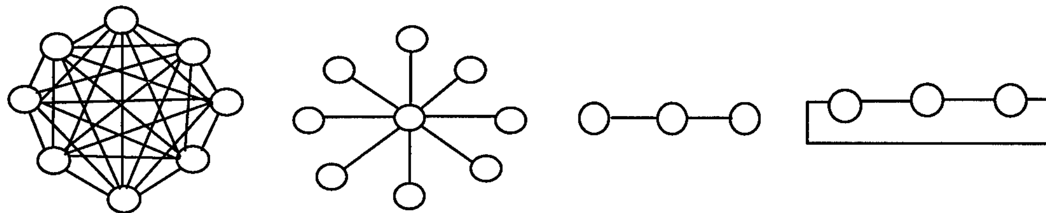


Figure 7.5. Static Interconnection Networks.

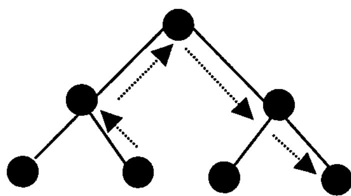
Mesh Network (n-dimension)

It is a network with different dimensions. Two-dimensional mesh is an extension of linear array. If both dimensions are the same then it is called a square mesh. Wraparound mesh architecture is also known as torus.

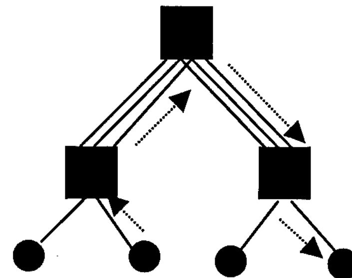
Figure 7.6. A visual description of 2-D and 3-D Mesh architectures.

Tree Network

There is only one path between any pair of processors. In a static tree each node is a processor, whereas in a dynamic tree all intermediate nodes are switching elements, leaves are processors. The reason it is called dynamic is routing is determined dynamically. There is a communication bottleneck at higher levels. This problem can be alleviated by increasing the number of links between nodes of higher levels, leading to a Fat Tree.



a) A static tree



b) A dynamic tree

Figure 7.7. A static and a dynamic tree diagrams.
Dashed lines show the route that a message has to follow from one processor to the other.

Hypercube Network

Hypercube is basically a multidimensional mesh of with exactly two processors in each dimension. Nodes of a hypercube are connected if their labels differ exactly one bit position. In a d-dimensional hypercube, each node is connected to d others. A d-dimensional hypercube can be partitioned into two (d-1) dimension hypercubes. Processor labels have only d bits. Total number of differing bit positions is called Hamming Distance. Shortest path is found by xor-ing the labels. 1s in the result points to the dimensions to route the message. These properties of hypercube make it easy for the system developer to apply appropriate routing techniques. Since it has a very complex nature hypercube networks are difficult and expensive to build.

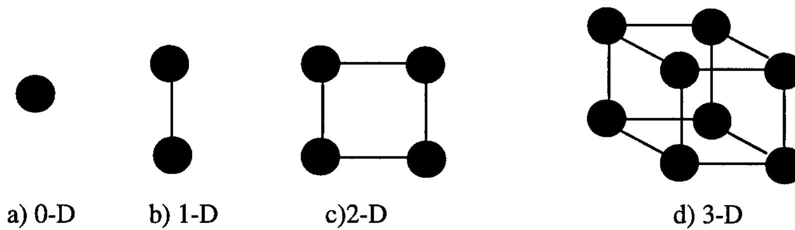


Figure 7.8. Hypercubes with different dimensions.

7.2.6. Evaluating Static Interconnections Networks

Diameter - Maximum distance between any two processors. (distance is the shortest path between two nodes.). Networks with small diameters are desired.

Connectivity - Multiplicity of paths between any two processors. (the minimum number of arcs or nodes that must be removed to break it into two disconnected networks). High connectivity is desired.

Bisection Width - Minimum number of communication links that have to be removed to partition the network into two equal halves.

Cost - Generally defined as the number of communication links required by the network. Completely connected and hypercube networks are the most expensive ones.

7.2.6.1. The Effect of Granularity and Data Mapping on Performance

Since using as many processors as the number of inputs is not practical at all, we need to increase the granularity of computation on the processors. Using fewer than the maximum number of processors to execute a parallel algorithm is called scaling down a parallel system in terms of the number of processors. Conceiving the finest-grain algorithm is usually easy, therefore many developers may choose this approach as their first step when they design a new parallel system. However, we should spent more time on how to map data onto the processors and their implementation on different number of

processors. The most fine-grain algorithm is not always the optimal algorithm for solving a parallel design issue. We need to think about the time it takes for the processors to communicate among each other to produce a result. Then we might make some choices depending on the nature of the problem, type of parallel algorithm, the architecture of the processors and how they communicate. It is good to note here that cut-through routing has much faster transfer rate of large messages than store-and-forward routing.

7.2.6.2. The Isoeffeciency Metric of Scalability

To have a scalable system we need to increase both the number of processors and the size of the problem. It is good to be able to determine the rate at which the problem size must increase with respect to the number of processors to keep the efficiency fixed. This rate is the key element in determining the degree of scalability of parallel systems.

Problem size can be defined as the number of basic computation steps in the best sequential algorithm to solve the problem on a single processor. When we increase the problem size a certain times, it should mean that we need to perform that certain times as much computation. Parallel systems do not achieve a speedup of p or an efficiency of one. All causes of nonoptimal efficiency of a parallel system are referred to as the overhead due to parallel processing. It is the total time spent by all the processors in addition to that required by the fastest known sequential algorithm for solving the same problem on a single processor. The relation between cost (pT_p), problem size (W), and the overhead is given by :

$$T_0 = pT_p - W$$

In poorly scalable parallel systems if we increase the number of processors by a small number we need to increase the size of the problem enormously. On the other hand, if W

grows linearly with respect to p, then the parallel system is highly scalable, and we can get speedups proportional to the number of processors. The isoefficiency function of the parallel system is the one that dictates the growth rate of W required to keep the efficiency fixed as p increases. It gives us an idea about the ease with which a parallel system can maintain a constant efficiency and hence achieve speedups increasing in proportion to the number of processors. A small isoefficiency means that small increments in the problem size are sufficient for the efficient utilization of an increasing number of processors, indicating that the parallel system is highly scalable. A large isoefficiency function indicates that the parallel system is poorly scalable. The isoeffieciency function does not exist for unscalable parallel systems, because in such systems the efficiency cannot be kept at any constant value as p increases, no matter how fast the problem size is increased.

We can say that a parallel system is cost-optimal if and only if $pT_p = \Theta(W)$. In other words it is cost-optimal if and only if its overhead function does not asymptotically exceed the problem size. By checking this function we can decide if the parallel system in question is scalable or not.

7.2.6.3. The Degree of Concurrency and Isoefficiency Function

Even for an ideal system with no communication or other overhead, the efficiency will drop because processors added beyond p=W will be idle. Thus, asymptotically, the problem size must increase at least as fast as $\theta(p)$ to maintain fixed efficiency; hence, $\Omega(p)$ is the asymptotic lower bound on the isoefficiency function. It follows that the isoefficiency function of an ideally scalable system is $\theta(p)$. This lower bound is imposed on the isoefficiency function by the number of operations that can be performed

concurrently. The maximum number of tasks that can be executed simultaneously at any time in a parallel algorithm is called its degree of concurrency. It is independent of the parallel architecture. If C(W) is the degree of concurrency of a parallel algorithm, then for a problem size of W, no more than C(W) processors can be employed effectively.

The isoefficiency function due to concurrency is said to be optimal, i.e., $\theta(p)$, if the degree of concurrency is $\theta(W)$. If the degree of concurrency is less than $\theta(W)$, then the isoefficiency function due to concurrency is worse (i.e., greater) than $\theta(p)$.

7.2.6.4. Physical Machine Models

- Single-instruction multiple-data (SIMD) machines
- Parallel vector processors (PVP)
- Symmetric multiprocessors (SMP)
- Massively Parallel processors (MPP)
- Cluster of workstations (COW)
- Distributed Shared Memory (DSM) multiprocessors

PVP: It uses High-bandwidth crossbar switch network to connect processors to a number of shared memory modules. Memory modules can provide a very high speed data access. Instead of cache memory they generally use vector registers and an instruction buffer.

SMP: Processors are connected to a shared memory through a high-speed snoopy bus. (sometimes a crossbar is also used in addition). Unlike PVPs they have cache memory. They are heavily used in commercial applications. It is a symmetric system, i.e., each processor has equal access to the shared memory.

MPP: It uses commodity processors, and physically distributed memory over processing nodes. It also has an interconnect with high communication bandwidth and low latency. It can be scaled up to thousands of processors. It is an asynchronous MIMD machine, but processors are synchronized by blocking message-passing operations. The nodes are connected by a high-speed network, and said to be tightly coupled.

DSM: Cache directory is used to support distributed coherent caches. Memory is physically distributed among different nodes. That separates DSMs from SMPs.

COW: It has a lower cost than the MPPs. Each node is a complete workstation. A node may be a PC or an SMP. Nodes are connected by a low-cost commodity network. It is loosely coupled with I/O bus in contrast with MPPs. There is always a local disk, whereas MPPs don't have to have one. Each node has a complete operating system internally. MPPs might have only a microkernel. Clusters of computers are now filling the processing niche once occupied by more powerful stand-alone machines.

7.2.6.5. Coordination of Parallel and Distributed Applications

Programming of parallel systems is often considerably more difficult than (what intuitively seems to be) necessary [17]. It is widely known that major hindrance against widespread use of massive parallelism is the lack of a coherent model of how concurrent systems must be organized and programmed. Some views of concurrency on programming languages that are based on extensions of the sequential programming are not suitable to meet this objective. Coordination models and languages have the objective of finding solutions to the problem of managing the interactions among the active entities in concurrent programs. They can also be thought of as linguistic counterpart of software

libraries and platforms, like MPI, or PVM, which offer extralinguistic support for parallel programming.

7.2.6.6. Comparison of two prominent message passing languages, MPI & PVM

There are different parallel message passing languages in the arena, but MPI (Message Passing Interface) and PVM (Parallel Virtual Machine) are two of the most used ones. MPI is also used here in AFIT for parallel computing studies. There are two versions of MPI to work on Sun Solaris UNIX system and on AFIT Bimodal Cluster (ABC). PVM is supported by IBM, and there are codes written for different applications. Its functionality and performance is poor when compared with that of MPI. MPI is also more portable. That it is not supported by IBM is a disadvantage for MPI. MPI has a library for multidisciplinary applications. MPI's advantage is its ease of use against other languages, but PVM has its own control structures relieving the programmer from keeping track of the messages. Overall performance and functionality favor more toward using MPI, but all languages have pros and cons, and they have to be well evaluated.

7.2.6.7. Dominant Programming Models In Parallel Computing

In order for the applications execute in parallel the system must provide support for prevalent parallel programming and styles, and provide a comprehensive set of tools and environments (for languages like Fortran and C) for the development of new parallel applications, the porting of existing parallel applications, and the conversion of existing serial applications. There are essentially three parallel programming models that are being used in large scaleable systems, the message-passing programming model, the shared-memory programming model, and the data parallel programming model [35].

With the explicit message-passing model, processes in a parallel application gave their own private address spaces and share data via explicit messages. Such programs execute in a loosely synchronous style with computation phases alternating with communication phases.

With the shared-memory model, processes in a parallel application share a common address space, and data are shared by a process directly referencing that address space. A programmer must identify when and what data are being shared and must properly synchronize the processes using special synchronization constructs. This model is often associated with dynamic control parallelism, where logically independent threads of execution are spawned at the level of functional tasks or at the level of loop iterations.

The data parallel model is supported by a data parallel language such as High Performance Fortran. A High Performance Fortran preprocessor or compiler then translates the source code into an equivalent SPMD program with message passing calls, (if the target is a system with an underlying message passing architecture) or with proper synchronizations (when the target system has a shared-memory architecture. Support of these programming models with the efficient libraries and language support and an easy-to-use program development and execution environment are critical for system development.

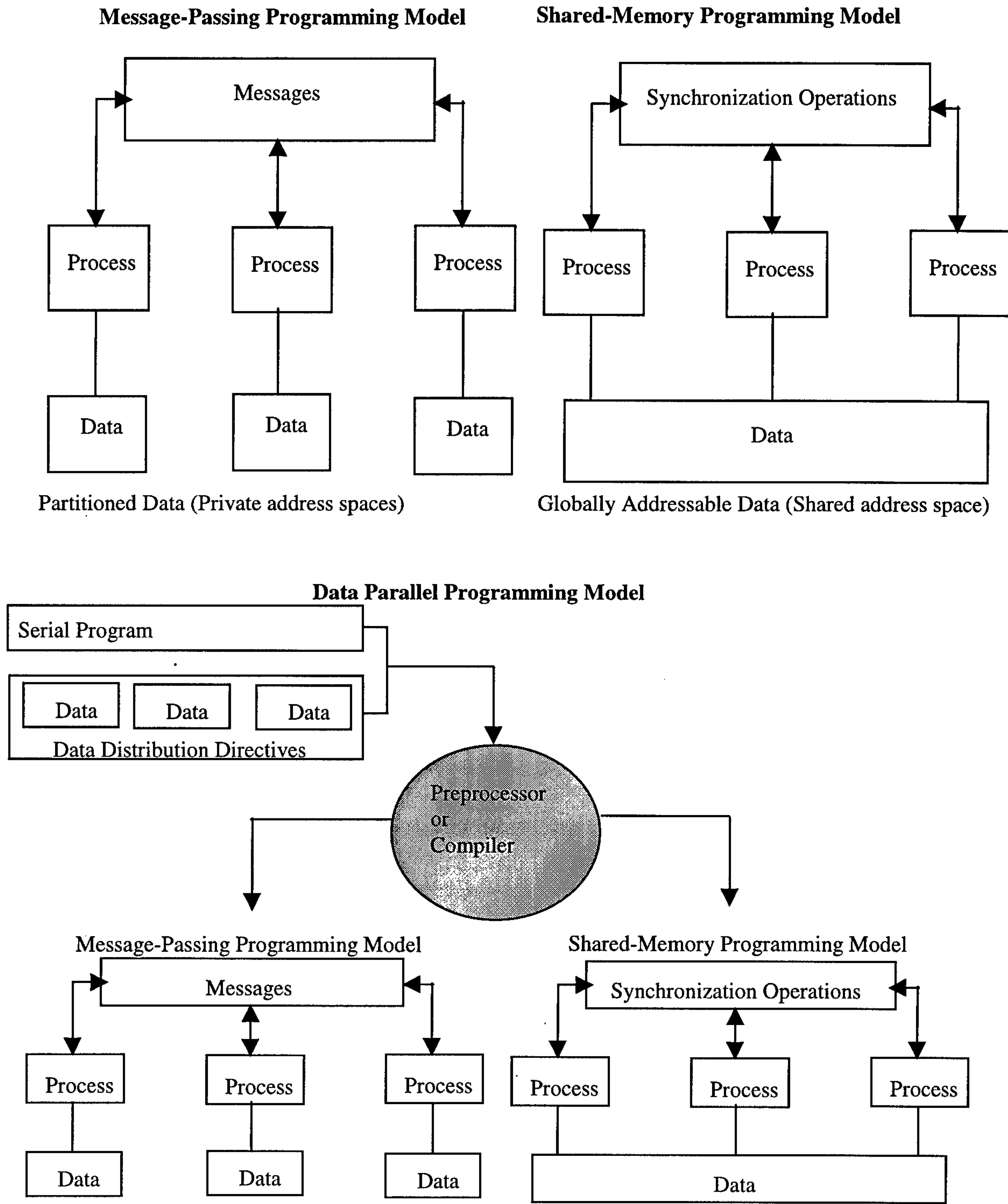


Figure 7.9. Programming Models commonly used in Parallel Computing.

8. Appendix B - Search Techniques

8.1. Introduction

This section is an overview of the search techniques and their areas of applicability. The information presented here is a good foundation towards understanding the approaches taken to solve MRP.

8.2. Search Techniques

Below is some features of different search techniques used for different problems in optimization and search problems.

8.2.1. Selecting a representation for the Search

The selection of a representation to fit a given problem is sometimes dictated by the problem specifications and sometimes is a matter of choice. But, as a general rule, problem-reduction representations are more suited to problems in which the final solution is conveniently represented as a tree (or graph) structure. State-space representations are more suited to problems in which the final solution can be specified in the form of a path or as a single node.

AND/OR graphs are widely used for the representation of strategy seeking problems. AND links are used to represent the changes in the problem situation caused by external, uncontrolled conditions, and OR links are used to alternative ways of reacting to such changes. In games we can think of these uncontrolled conditions as the moves made by the player.

In general if the solution is a prescription for responding to observations, then that problem can be thought as a strategy-seeking problem, and AND/OR graph is the most suitable representation for these problems. If, on the other hand, the solution can be

expressed as an unconditional sequence of actions or as a single object with a given set of characteristics, then we have a path-finding or constraint-satisfaction problem and a state-space representation would be the most appropriate. AND/OR graphs can also be used to represent problems whose solution is a partially ordered sequence of actions. This property occurs when each available operator replaces a single component of the database by a set of new components. The tasks of logical reasoning and theorem proving also give rise to AND/OR structures.

In certain problems both state-space and problem reduction formulations appear equally appropriate and the choice of representation requires additional insight. We have to analyze the problem in detail so that we can choose the most appropriate representation. The reasoning behind problem-reduction is called means-end analysis. The basic difference between this method and the state-space approach is its purposeful behavior: operators are invoked by virtue of their potential in fulfilling a desirable subgoal, and new subgoals are created in order to enable the activation of a desirable operator.

8.2.2. Split and prune method

Among the most desired of an object's code features are its definiteness and whether it takes into account a problem's structure. This means that an object's code should depict an object's individual characteristics and a subset of potential solutions related to that object. This representation lets us to effectively transform sets by splitting. Splitting is founded on partitioning a problem into subproblems, pruning some objects, and the investigation of only the most promising ones. Successive splitting operations may lead to a problem, which is easily solved.

This technique is called split-and-prune method. In artificial intelligence, owing to the large sizes of the problems, analogous techniques are used. These are called as generate-and-test methods. Instead of pruning or rejecting objects of some set, new objects are generated, and only some of them are then used in later investigations.

8.2.3. Exhaustive Graph Searching

The basic algorithm used when solving various kinds of graph problems is an exhaustive graph search starting from some initial node p. The goal of the algorithm is to examine all graph nodes. We call the method as exhaustive, because in successive steps it explores edges leading out of nodes of the set being searched, puts the endpoints of these into the set, and marks the edges as inspected. The algorithm terminates when every edge leading out of every node has been marked as inspected.

8.2.4. State-space graph searching

In most practical problems, and especially those related to artificial intelligence, the size of sets examined are so large that it is impossible to store all of its elements. For this reason generation instead of splitting is used. The techniques are still exhaustive but, they do not require as much storage.

There are two kinds of strategies: blind – not taking advantage of information; and directed – using information. In the former, the order of searching, i.e. of executing the above operations, depends solely on the nodes already examined. In the latter, information on nodes, which have not been examined, may affect the order of searching, i.e. the order of selection and generation operations.

An example of state-space graph is shown in Figure 7.1.

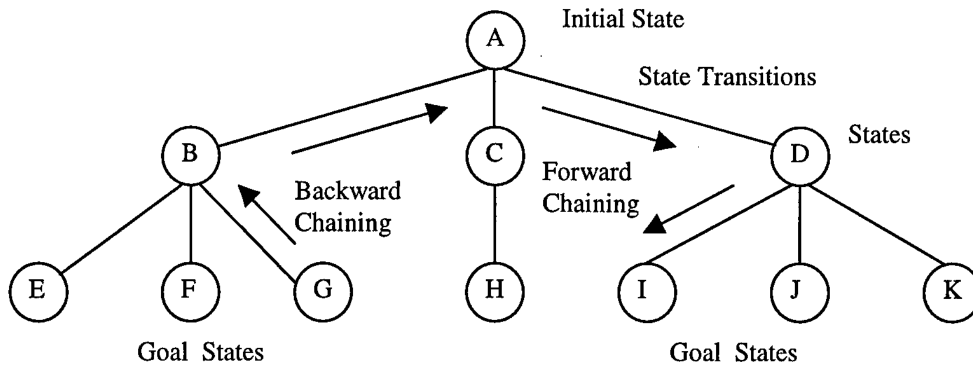


Figure 8.1. An example of state-space graph.

8.2.5. Problem reduction representation and AND/OR Graph Searching

Not every problem can be solved by finding a path in a state-space graph. There are problems in which we look for strategies satisfying desired conditions. Strategy is not just a sequence of operations, but a prescription to execute an operation in response to an external event when we are thinking about solving problems we can regard external events as the outcome of tests or procedures. For various possible external events various actions are undertaken, so a solution is not a path (as it was in the state-space graphs case), but a tree.

The unique feature of this class of problems is that each residual problem posed by some subset of candidates can be viewed as a conjunction of several subproblems that may be solved independently of each other, may require specialized treatments, and should, therefore, be represented as separate entities. So, each node stands for one subproblem, and the curved arcs indicate which sets of subproblems must be solved conjunctively to make up a complete solution [36].

Assigning nodes to subproblems being solved, we are left with a directed graph with two kinds of edges. Edges marking a passage to subproblems alternative with respect to

other subproblems derived from the same problem (node) are called the OR edges. Edges representing a passage to components of a product of subproblems are called AND edges. Without the loss of generality, we can assume the uniformity of a directed graph, i.e. that only one kind of edge can lead out of any graph node. If in a graph there were a node with both OR and AND edges leading out of it, we could split subproblems related to this node into alternative and product ones and introduce an additional node thus, achieving graph uniformity. In a uniform directed graph, nodes with alternative edges are called OR nodes and nodes with product edges are called the AND nodes.

AND/OR graphs are used for problem representation. Each graph node represents a problem to be solved or a potential goal (problem with known solution). The initial node represents the original problem [37]. State-space graphs that consist solely of OR edges are a special case of AND/OR graphs.

Finding a solution in an AND/OR graph results in finding a solution tree (an AND/OR subgraph) which satisfies the following:

- It contains the initial node.
- All terminal nodes represent trivial problems with known solutions.
- If it contains any AND edge leading out of node *w,* then it contains all other AND edges leading out of *w*.

A node is labelled as "solved" if:

- It satisfies the goal criterion (representing a trivial problem).
- It is a non-terminal OR node, and at least one of its successors is labeled as "solved".
- It is a non-terminal AND node and all its successors are labeled as "solved".

Search terminates if the initial node (representing the original problem) can be labeled. An example of an AND/OR graph is shown in Figure 7.2.

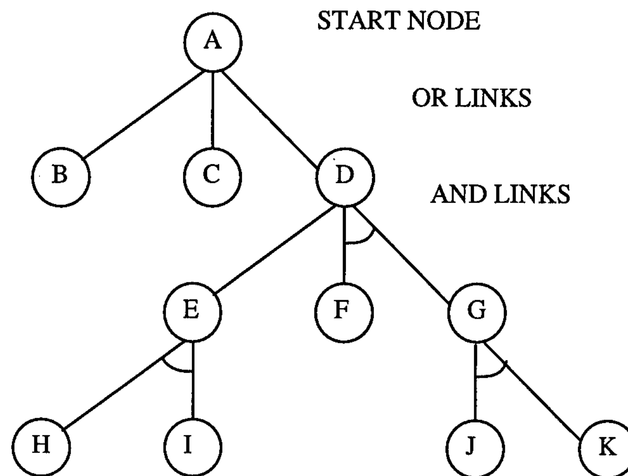


Figure 8.2. An example of AND/OR Graph with arcs between branches representing AND.

8.2.6. Stochastic methods and probabilistic algorithms

As their names imply, these techniques incorporate some means of probabilistic methods in their search process. Below are brief descriptions about some of the most widely-used probabilistic algorithms.

8.2.6.1. Random search

Random search is just random guessing of solutions. Although it is unlikely to produce an optimal solution, its execution time is too small and it can be used to provide known bounds to other approaches. It is also a good technique to be used when the cost of finding a solution is a constraint.

8.2.6.2. Potential Fields

Potential fields technique uses the principles of electromagnetism. The goal point is given a strong positive potential, and the starting point (source) is given a potential less

than (i.e. more negative than) the goal. The moving point is assigned the same potential as the source. Obstacles are assigned different potentials with values less than the source. The moving point is then repelled by the source and obstacles, and attracted towards the goal. The primary disadvantage with this method is that it tends to get trapped in local field minima and never reach the goal. Also it does not guarantee an optimal solution. Some path-planning algorithms use this technique.

8.2.6.3. Simulated Annealing

SA algorithms use the concepts of annealing in thermodynamics. Annealing is the heating and slow cooling of an object. The algorithm assigns the some of the points as hot. Then over successive iterations the points are cooled according to a probabilistic cooling scheme. Unwanted points are avoided by assigning them higher energy states. When the algorithm finds a good area to search it employs a local search such as greedy algorithm. Over time the algorithm will settle on a minimum temperature which should give an optimal solution. Cooling techniques are important for achieving optimality, and the algorithm has a tendency to get trapped in local minima.

8.2.6.4. Genetic Algorithms

Genetic algorithms (GA) are probabilistic algorithms based on theories of evolution and natural selection. They generate a pool of random solutions. The number of these solutions is much smaller than the search space. Through the application of genetic operators to an initial population of string of structures, a new generation called offspring is generated. The strings with the highest fitness values will be kept for breeding the next generation, whereas the ones with lower fitness are eliminated. GAs have at least three operators, reproduction, crossover, and mutation. Reproduction employs a selection

strategy to determine which strings will survive or be copied over into the next generation. Crossover operator creates new strings using pieces of the strings from the previous generation. A typical crossover operator mates a pair of strings by randomly selecting a crossover point along the length of the string and swapping the string sequence from the crossover point to the end of the string. The mutation operator randomly selects a string within the population and alters part of the string. The objective behind GAs is that two good solutions combine their good characteristics to form an even better solution. By applying the genetic algorithm operators to an initial population and simulating the process of natural selection over many generations, a final population of the fittest strings is formed. With proper formulation of the problem, the longer the algorithm runs, the greater the probability of finding an optimal solution. A very good advantage of GAs over other known algorithms is that they are easily parallelized. Unfortunately, GAs do not necessarily find the "best" solution if one exists. The search sometimes converge to a poor solution even when a better solution is close by.

8.2.6.5. Monte Carlo and Las Vegas

Monte Carlo algorithms are those that are always fast and probably correct, whereas Las Vegas algorithms are probably fast and always correct. A Monte Carlo algorithm occasionally makes an error, but it finds a correct solution with high probability whatever the instance considered. Since the number of the errors and their occurrence frequency is not known it cannot be employed for MRP problem. A Las Vegas algorithm, on the other hand, always produces a correct answer given enough time. However, due to the randomness of the selection decisions, the algorithm may require vastly different

execution times for the same problem instance. This algorithm is also not appropriate for MRP problem.

8.2.7. Deterministic Search Methods

8.2.7.1. Uninformed Search

The simplest of deterministic search algorithm is uninformed (blind, or undirected) search. A search is said to be uninformed if the location of the goal does not affect the number of input points explored, except for the termination conditions [36]. They do not use any heuristics to guide the search process.

8.2.7.2. Depth first search

If a search space is organized as a stack, the search strategy is called a depth first strategy and is LIFO (last in first out) in nature. The name emphasizes the order in which nodes are examined. A single path is investigated as long as its last element is not found to be a goal or a terminal node. Depth first strategy is ineffective if used for large graphs and especially for graphs with an infinite depth even if the solution path is at a finite depth. That is why most of the depth first techniques employ a depth control rule, by setting a depth-bound. When the depth-bound is achieved or a node satisfies some goal property, the search backtracks. If we notice that at any given time the CLOSED list forms a single path from the start node to the currently expanded node. This feature reflects the storage economy of depth first search. The maximum storage required cannot exceed the product of the depth-bound and the branching degree.

In depth first search, as well as in the popular variation called backtracking, priority is given to nodes at deeper levels of the search graph. The finest computational step in depth first search is node expansion, i.e. each node chosen for exploration gets all its

successors generated before another node is explored. After each node is expanded, one of its newly generated children is again selected for expansion and this forward exploration is pursued until, for some reason, progress is blocked.

8.2.7.3. Backtracking

Backtracking is a version of depth first search that applies the last in first out policy to node generation instead of node expansion. When a node is first selected for exploration, only one of its successors is generated and this newly generated node, unless it is found to be terminal or dead end, is again submitted for exploration. If, however, the generated node meets some stopping criterion, the program backtracks to the closest unexpanded ancestor, that is, an ancestor still having ungenerated successors.

The main advantage of backtracking in comparison with the depth first search is storage conservation. Instead of storing all successors of a node as in the depth first strategy, only one node is being stored in backtracking strategy. This strategy guarantees that after its termination (i.e. after a goal node is found or all graph nodes are examined), all the generated nodes are expanded. So, all the nodes placed in the main storage actively take part in the operation of the algorithm. A similar feature is not guaranteed by the depth first strategy. There is a trade off between space saving and algorithm complexity in the backtracking strategy. Backtracking is particularly useful in optimizing problems. In a graph containing many goal nodes it is possible to interrupt the examination of the current path if its properties are worse than that of the currently optimal one.

Depth-first search:

1. Put the start node on OPEN.
2. If OPEN is empty, exit with failure; otherwise continue.
3. Remove the topmost node from OPEN and put it on CLOSED. Call this node *n*.
4. If the depth of n is equal to the depth bound, clean up CLOSED and go to step 2; otherwise continue
5. Expand *n*, generating all of its successors. Put these successors (in no particular order) on top of OPEN and provide for each a pointer back to *n*.
6. If any of these successors is a goal node, exit with the solution obtained by tracing back thorough its pointers; otherwise continue.
7. If any of these successors is a dead end, remove it from OPEN and clean up CLOSED.
8. Go to step 2.

   The operation "clean up CLOSED" in steps 4 and 7 is optional and used only for memory conservation purposes. It is basically purging from CLOSED all those ancestors of the nodes passing the tests in step 4 and 7 that do not have sons in OPEN.

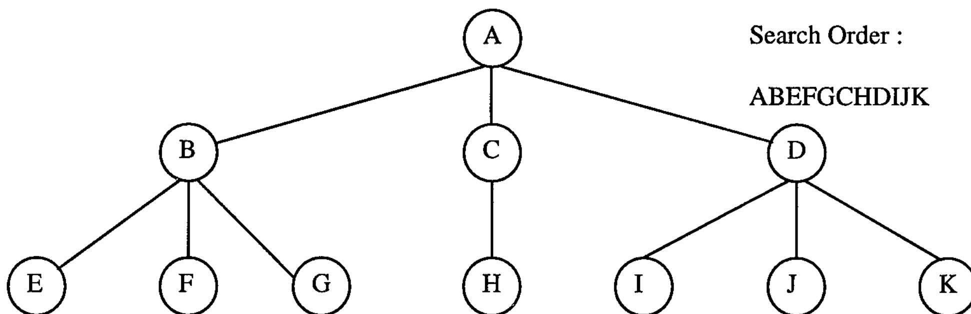


Figure 8.3. Depth-first search with an example.

8.2.7.4. Breadth First Search

BFS, as opposed to DFS, assign a higher priority to nodes at the shallower levels of the searched graph, progressively exploring sections of that graph in layers of equal depth. Thus instead of a LIFO policy, bfs is implemented by a first in first out (FIFO) policy, giving first priority to the nodes residing in OPEN for the longest time.

Unlike dfs, bfs of locally finite graphs is guaranteed to terminate with a solution if a solution exists. Moreover, it is also guaranteed to find the shallowest possible solution. Instead of a single traversal path, bfs must retain in storage the entire portion of the graph that it explores. Only by retaining a full copy of the search graph can it shift attention away from newly generated nodes and come back to expand nodes suspended many steps earlier. The need for this abrupt shift of attention, combined with the large storage requirements, is the main reason that bfs is rarely used for solving large problems and for searching in large graphs. In particular, it cannot be used for high complexity problems in the area of artificial intelligence [16].

8.2.7.5. The Uniform-cost procedure

To be able to utilize from bfs in optimization problems, an important variation, called the uniform-cost or cheapest-first strategy, is often employed. Instead of progressing in layers of equal depth, we unravel the search space in layers of equal cost. If our task is to find the cheapest path from the start node to a goal and if path costs are nondecreasing with length, we can make sure that no node of cost greater than C is ever expanded while other candidates, promising costs lower than c, are waiting their turn. This guarantees that the first node chosen for expansion is also the cheapest solution.

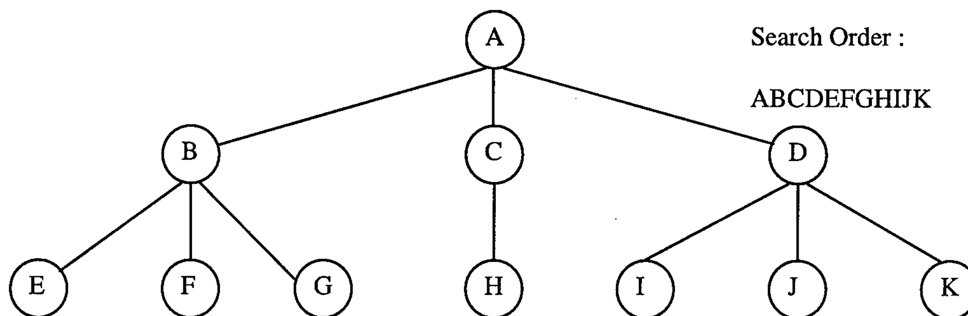


Figure 8.4. An example of Breadth-First Search.

8.2.7.6. Dynamic programming

This is a recursive formulation of breadth first search, which, for regular structures, may yield analytical expressions for the optimal cost or optimal policy. The problem is subdivided into smaller and smaller subproblems which are solved one-by-one and assembled until the complete problem is solved. A list is kept of the generated subproblems so as to make sure that the same problem is never solved twice. The optimal solution is arrived at by employing the principle of optimality, which requires that for any intermediate state of the problem and the respective intermediate solution for this state, the solutions of the subsequent suproblems must constitute an optimal solution sequence with regard to the problem state resulting from the stated intermediate solution.

Dynamic programming has been used to solve shortest-path problems, and has been applied to solve the MRP problem. If we compare the performance of A* and dynamic programming search techniques we see that the result is greatly dependent on the objective function used for each one. Dynamic programming evaluates paths to all points in a search graph, and its efficiency relies on the optimal substructure of the shortest path problem, and the fact that it does not repeat calculations. The A* algorithm will explore fewer paths if its heuristic objective function is monotonic. Dynamic programming with a good objective function may outperform the A* algorithm with a poor heuristic function.

8.2.7.7. Best-First Strategy

This is an informed search, and unlike depth-first or breadth-first searches, they use some knowledge about the problem space to try to reduce the number of nodes examined in the search process. The most important characteristic of this search is that the nodes are ordered by some criteria and the first in the ordering is selected for evaluation [37].

This strategy enables us to employ heuristics in the search process. These heuristics are used while deciding about the next node to be examined. The best node among all the others is then selected after applying the heuristics. The criteria used by the heuristics are as follows:

- Convergence, i.e. goal node reachability.
- The lowest cost of the path from the initial node, through *w,* to the goal node.
- Lowest computational complexity of the search process.

Those criteria represent a compromise between two requirements: The need to make such criteria simple and, at the same time, the desire to see them discriminate correctly between good and bad selections. Most complex problems require the evaluation of an immense number of possibilities to determine an exact solution. To be able to get to that precise solution requires an evaluation of almost a lifetime. Heuristics become a good help by indicating good ways to reduce the number of evaluations and thus resulting in solutions within reasonable time limits.

Heuristics information can be gathered in various ways. One way is to assess the difficulty of solving the subproblem represented by the node. Another way is to estimate the quality of the set of candidate solutions encoded by the node, that is, those containing the path found leading to that node. A third alternative is to consider the amount of information that we anticipate gaining by expanding a given node and the importance of this information in guiding the overall search strategy. In all these cases, the promise of a node is estimated numerically by a heuristic evaluation function f(n) which, in general, may depend on the description of n, the description of the goal, the information gathered

by the search up to that point, and most important, on any extra knowledge about the problem domain.

We can assume that a problem being solved can be described in terms of objects (states) and operators. Operators applied to objects generate (if possible) new objects. Operators are represented by object generation rules. While selecting the most promising operators, heuristic search uses various methods whose purpose is to limit the set of objects examined. Even though they do not guarantee to find a solution, in principle, heuristics help determine the best solution obtainable within a set time. To achieve effectiveness, heuristics should have the following characteristics:

- Uncertainty of the result
- Incompleteness of available knowledge.
- Improvement of the solution found.

Some other algorithms, such as hill-climbing, also use heuristic information, but what makes best-first search different form all the others is the commitment to select the best from among all the nodes encountered so far, no matter where it is in the partially developed tree.

8.2.8. Analysis of Heuristic Search Strategies

One of the most desirable features of search algorithms is its convergence. A search strategy is convergent if it guarantees finding a path or a solution graph, if they exist, or information that there is no solution (in the case of finite search spaces). In the case of infinite search spaces, a convergent strategy does not have to give information about the lack of solution.

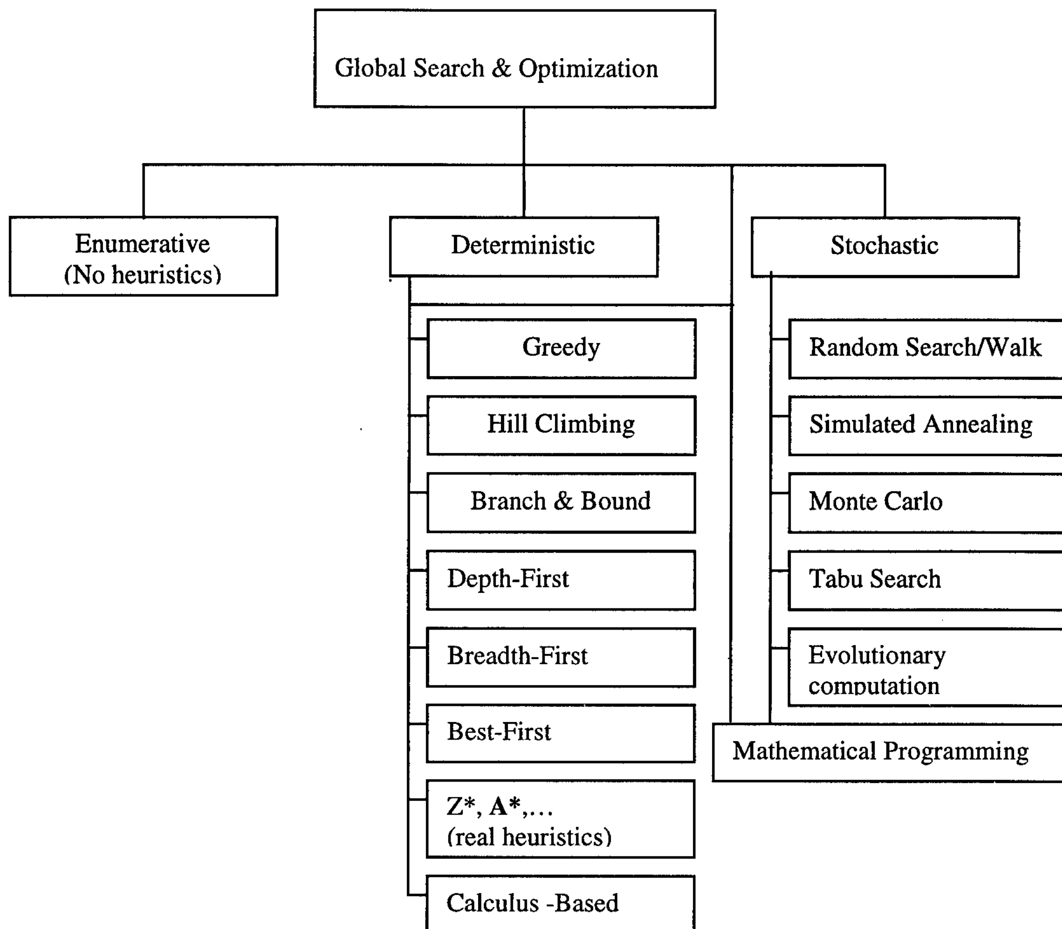


Figure 8.5. Global search & optimization algorithms.

Strategies that guarantee finding the optimal solution (if a solution exist at all) are called admissible. When comparing various strategies, a notion of strategy dominance is used. Strategy S1 dominates over strategy S2 if every node examined by S2 is also examined by S1. A strategy is called optimal in a class if it dominates all other strategies of that class.

The main goal of a heuristic is to improve the quality of a solution, and to reduce the complexity of a search for a solution by limiting the number of alternative search directions examined.

*8.2.8.1. The importance of Heuristic Functions Used in A**

The quality of heuristics used in search strategies directly influences the complexity of a solution (both the finding of a solution and its execution). If a heuristic estimate h coincides with h*, then A* examines only one, optimal path. If, on the other hand, h is constant for every node, then A* reduces to the uniform cost algorithm. In that case, to find a solution it is necessary to examine all those nodes for which paths leading to them have smaller than the optimal cost (if the cost of the optimal path is known).

The information about the relations between h, the heuristic function used and the optimal function h* is usually hard to obtain. A heuristic function h_1 uses more information than a heuristic function h_2, if both of them are reducible and for every non-goal node n

$$h_2(n) > h_1(n)$$

is satisfied. If A^*_1 uses more information than A^*_2, then A^*_1 dominates A^*_2. So, the use of a heuristic using more information guarantees greater effectiveness of A*.

*8.2.8.2. Properties of f**

The node selection function f used by A* consists of adding two parts, g(n) and h(n), where:

> g(n) = the cost of $PP_{s\text{-}n}$, the current path from s to n, with g(s) = 0.
>
> h(n) = an estimate of h*(n), such that h(goal node) = 0.
>
> g*(n) = the cheapest cost of paths going from s to some node n.
>
> h*(n) = the cheapest cost of paths going from n to some goal node.
>
> C*(n) = the cheapest cost of paths going from s to some goal node.

When g and h coincide with their optimal values, the resulting f assumes a special meaning.

$$f^*(n) = g^*(n) + h^*(n)$$

We can say that a path is optimal if and only if every segment of it is optimal, and is a unique feature of the recursive and order-preserving properties of the additive cost measure.

The trouble, of course, is that we normally do not possess the extra information or insight required to compute f*, especially the h* part. If we try to compute it by searching the graph itself (as is actually done in dynamic programming), we defeat the main purpose, which is to avoid excessive search, because many off-track nodes need to be explored in the computation of h*.

There still remains the possibility of approximating f* by an easily computable function f, and this is exactly what A* attempts to achieve via the combination

$$f = g + h.$$

A* always terminates on finite graphs. The reason is that the number of acyclic paths in each such graph is finite and with every node expansion A* adds new links to its traversal tree. Each newly added link represents a new acyclic path and so the reservoir of paths must eventually be exhausted. Note that A* only reopens a node on CLOSED when it finds a strictly cheaper path to it.

A* is also complete on finite graphs. A failure is returned only when OPEN is found empty and, if a solution path is discovered. For if it did there would be at least a node on

8.2.8.3. A Search Algorithm*

A* is a specialization of Z* where the target of pursuit is the path of minimum sum-cost. A* is a best-first, branch-and-bound search process. It has some basic characteristics as described below.

Completeness : an algorithm is said to be complete if it terminates with a solution when one exists.

Admissibility : an algorithm is admissible if it is guaranteed to return an optimal solution whenever a solution exists.

Dominance : an algorithm A_1 is said to dominate A_2 if every node expanded by A_1 is also expanded by A_2. Similarly, A_1 strictly dominates A_2 if A_1 dominates A_2 and A_2 does not dominate A_1. It is also appropriate to use the term more efficient than interchangeably with dominates.

Optimality : an algorithm is said to be optimal over a class of algorithms if it dominates all members of that class.

8.2.8.4. A's convergence*

For a finite network with a non-negative cost function, if A* terminates after finding a solution, or if there is no solution, then it is convergent. We know that the number of acyclic paths is finite. An already examined node w can only be included in the search again, if a new path to w was found with a smaller cost than the previous one. The cost function is non-negative, so an edge can be examined only once. Thus, A* is convergent.

If some additional conditions are satisfied it can be shown that A* is convergent for infinite (locally finite) graphs. For an infinite network (a locally finite graph) with a

positive and bounded cost function (the lower limit of the cost of an edge is greater than zero), the A* strategy is convergent.

8.2.8.5. A's Admissibility-A guarantee for an optimal solution*

Whenever h(n) is an optimistic estimate of h*(n), where h*(n) is the minimal cost of a path leading from n to a goal node, A* will return an optimal solution. A heuristic function h is said to be admissible if

$$h(n) \leq h^*(n) \quad \forall n$$

Suppose that A* terminates with a goal node t for which f(t)=g(t)>C*. A* inspects nodes for compliance with the termination criterion only after it selects them for expansion. So, when t was chosen for expansion, it satisfied:

$$f(t) \leq f(n) \quad \forall n \in \text{OPEN}$$

This means that, just prior to termination, all nodes on OPEN satisfied f(n)>C*. Therefore the terminating t must have g(t)=C*, which means that A* returns an optimal path.

The most efficient algorithm would be one with h(n) =h*(n). since most of the problems cannot be formulated such that h(n) matches h*(n), monotonicity must be applied to obtain efficiency. It is not required for A* algorithm to find the optimal solution, but it is important for reducing the number of node expansions performed.

A heuristic function is said to be monotonic if it satisfies:

$h(n) \leq c(n,n')+h(n')$, for every n, n' where n' is a successor of n.

8.2.8.6. A's Complexity*

Computational complexity is that feature of search algorithms whose general investigation is particularly difficult. Computational complexity of a strategy can be expressed as a function N- the number of possible nodes in a graph that can be extended. The estimate of the upper limit for the number of nodes examined for any graph is $O(L^N)$, where L is a constant.

For a monotonic heuristic function, it is shown that A* finds the optimal path to all nodes extended, and that its complexity can be limited to O(N). However, for a majority of practical problems this number is too large. So, this strategy's complexity is expressed in terms of M, a number of nodes in the optimal path.

There are investigations of the A* strategy for an almost exact heuristic function. A heuristic function is almost exact if a relation induced by it orders the set of all nodes in exactly the same way as h* does. For such a heuristic function the complexity of A* is O(M).

The complexity of A* depends heavily on the assumptions made. If we employ different assumptions, we can come up with different complexities.

9. Appendix C - Porting from NX to MPI

9.1. Introduction

MPI has a library for multidisciplinary applications. MPI's advantage is its ease of use against other languages, but PVM, for example, has its own control structures relieving the programmer from keeping track of the messages. Overall performance and functionality favor more toward using MPI, but all languages have pros and cons, and they have to be well evaluated. There are some guidelines to be used when making the choice.

It is appropriate to use MPI when:

- A portable parallel program is needed
- Writing a parallel library
- Having irregular or dynamic data relationships that do not fit a data parallel model

It is not appropriate to use MPI when:

- HPF or a parallel Fortran 90 can do the job
- Parallelism is not needed at all

One of the objectives of this research effort is to port NX code into MPI for interoperability reasons. Also, the data structure is irregular in nature and restructuring is required in order to communicate this data. As can be seen from the guidelines as well, it is appropriate to use MPI for this purpose. In the following section details of porting the code into MPI are explained.

9.2. Porting of Initialization and Simple Send & Receive Instructions

The initial step in this research effort was to rehost the NX implementation of MRP to an MPI implementation using Mpich implementation of MPI developed at Argonne National Lab and Mississippi State University. The purpose of this appendix is to discuss how the NX code is rehosted, and how some features of MPI are used within the C MPI

program structure. Only the structures and features of MPI that are used are explained here. For detailed information on other areas about MPI, one should refer to the references [22, 23, 24].

One has to include the MPI library in the program compilation so that the execution recognizes the MPI functions. The program recognizes the functions between MPI_Init, and MPI_Finalize. The program structure is as follows.

```
#include <mpi.h>
...
main (int argc, char* argv[]) {
        /* No MPI functions are called before this */
        MPI_Init(&argc, &argv);      /* Initialize MPI */
        MPI_Comm_size( MPI_COMM_WORLD, &NumNodes);
        /* Finds out how many processors there are. */
        MPI_Comm_rank( MPI_COMM_WORLD, &Iam);
        /* Finds out which processor I am. */
        ...
        MPI_Finalize();        /* Finalize MPI */
        /* No MPI functions are called after this */
} /* main */
```

The most basic functions of MPI are the "send" and "receive" commands. They have the formats shown below.

```
Int MPI_Send(void*              buffer          /* in */,
             Int                count           /* in */,
             MPI_Datatype       datatype                /* in */,
             Int                destination     /* in */,
             Int                tag             /* in */,
             MPI_Comm           communicator            /* in */)

Int MPI_Recv(void*              buffer          /* out  */,
             Int                count           /* in   */,
             MPI_Datatype       datatype                /* in   */,
             Int                source          /* in   */,
             Int                tag             /* in   */,
             MPI_Comm           communicator            /* in   */,
             MPI_Status*        status          /* out  */)
```

There are four different types of communication modes: standard, synchronous, buffered, and ready. The communication can be a blocking, or a non-blocking type. Different formats may be used for MPI functions.

MPI_Send is a blocking type of function, i.e. the program does not progress until the send operation finishes its execution. A different format for the non-blocking function for send is used in the program. For this type of functions the program may execute other commands after the MPI_Isend command until MPI_Wait. The processor has to wait idle at that point until the MPI_Isend function completes. 'I' stands for "immediate" and these commands return immediately from the call. Both types of send functions are used in the code. For both MPI_Isend, and MPI_Irecv, the only difference in the message structure is the addition of

```
MPI_Request*        request        /* out */
```

as the last parameter.

```
Int MPI_Wait(
        MPI_Request*              request/* in/out       */,
        MPI_Status*               status   /* out        */ )
```

The NX commands "csend" and "crecv" that are used in the older version of MRP [6], and their corresponding MPI functions are as follows:

Table 9.1. Implementation of NX "csend" and "crecv" commands in MPI.

NX	**csend(EXPAND_NODE, &ENode, sizeof(ENode), To, NODE_PID);**
MPI	MPI_Send(&ENode,1,Pathtype,To,EXPAND_NODE,MPI_COMM_WORLD);
	Or MPI_Isend and MPI_Wait couple may be used.
MPI	MPI_Isend(&ENode,1,Pathtype,To,EXPAND_NODE,MPI_COMM_WORLD,&request); ... MPI_Wait(&request, &dummy);

NX	**crecv (EXPAND_NODE, &ENode, sizeof(ENode));**
MPI	MPI_Recv(&ENode,1,Pathtype,From,EXPAND_NODE,MPI_COMM_WORLD,&dummy);

Table 9.2. shows possible data types that are used in the MPI language, and their C language correspondents.

Table 9.2. MPI Datatypes and their C language values.

MPI_Datatype	C Datatype
MPI_CHAR	Signed char
MPI_SHORT	Signed short int
MPI_INT	Signed int
MPI_LONG	Signed long int
MPI_UNSIGNED_CHAR	Unsigned char
MPI_UNSIGNED_SHORT	Unsigned short int
MPI_UNSIGNED	Unsigned int
MPI_UNSIGNED_LONG	Unsigned long int
MPI_FLOAT	Float
MPI_DOUBLE	Double
MPI_LONG_DOUBLE	Long double
MPI_BYTE	
MPI_PACKED	

As can be seen from Table 9.2, the last two entries do not correspond to a C data type. MPI_Byte is a byte (8 binary digits), and different from a character. A character maybe interpreted differently in different machines, but a byte is the same on all machines. This is a useful feature, because even if the type does not match to any of the above, it can still be posted as MPI_Byte. Sending one MPI_Char type variable, such as mydata, is the same as sending sizeof(mydata) MPI_Byte type.

In the previous MPI_Send and MPI_Recv examples, a different data type, Pathtype, is used. This is a derived data type. Users can create data types according to their needs. One instance of creating such a data type is shown in Table 9.3. This is the data type that

is sent and received. It has a record structure, consisting of different fields of different data types.

Table 9.3. "Path" data type as implemented in the code.

Path Information Record
typedef struct {
int number; /* Number of entries in the route */
US x [MAX_PATH_LENGTH+1]; /* Vector of x locations */
US y [MAX_PATH_LENGTH+1]; /* Vector of y locations */
US z [MAX_PATH_LENGTH+1]; /* Vector of z locations */
Int VectorX, /* Direction vector in x direction */
Int VectorY, /* Direction vector in y direction */
Int VectorZ, /* Direction vector in z direction */
Float distance, /* Cumulative distance of the route */
Float radar, /* Cumulative radar detection cost */
Float g, /* Cost of the given route */
Float cost, /* Calculated cost (f') of route */
Int link, /* Forward Links for the OPEN list */
} PATH;

A procedure to create a new data type for this kind of data exists. Thus, as a result of the following sequence of instructions a new data type called Pathtype is formed.

```
/* set up 5 blocks for 5 different types of data. */
        int             Eblockcounts[5] = {1, 3*MAX_PATH_LENGTH+3, 3, 4, 1};
        MPI_Datatype        Etypes[5];
        MPI_Aint            Edispls[5];
        MPI_Datatype        Pathtype;

/* initialize types and displacements with addresses of items */
        MPI_Address( &ENode.number, &Edispls[0] );
        MPI_Address( &ENode.x, &Edispls[1] );
        MPI_Address( &ENode.VectorX, &Edispls[2] );
        MPI_Address( &ENode.distance, &Edispls[3] );
        MPI_Address( &ENode.link, &Edispls[4] );
        Etypes[0]=MPI_INT;
        Etypes[1]=MPI_UNSIGNED_SHORT;
        Etypes[2]=MPI_INT;
        Etypes[3]=MPI_FLOAT;
        Etypes[4]=MPI_INT;
        for(i=4;i>=0;i--) Edispls[i]-=Edispls[0];
        MPI_Type_struct(5, Eblockcounts, Edispls, Etypes, &Pathtype );
        MPI_Type_commit( &Pathtype );
```

Now, it is possible to send and receive messages containing this new type of data, as shown in the send and receive examples. This feature of creating derived data types is a very effective way of putting related data into one place and send them all at once, instead of sending them one by one. In the NX implementation, the programmer can not create a new data type. Of course, it is possible to send it without creating one. The only thing necessary in NX implementation is to give the name of the variable, and its size. But in the MPI implementation the programmer has to give a data type to each message being sent.

9.3. Translation of Iprobe

Another MPI command is MPI_Iprobe, which is used to see if there is a message waiting to be received. Again the NX implementation gives more flexibility to the programmer on the use of this function. The only information needed to check is the message tag, whereas MPI requires some other fields to identify the messages. The iprobe function in NX returns true or false, whereas the MPI_Iprobe returns that value in one of its parameters as a flag.

```
Int MPI_Iprobe(
        Int             source /* in   */,
        Int             tag    /* in   */,
        MPI_Comm        comm.  /* in   */,
        Int*            flag   /* out  */,
        MPI_Status      status /* out  */)
```

Table 9.4. Translation of NX "iprobe" command into MPI.

NX	**if (iprobe(EXPAND_NODE)) { ... }**
MPI	MPI_Iprobe(0, EXPAND_NODE, workersall, &flag, &dummy); if (flag) { … }

With either MPI_Recv function or MPI_Iprobe function it is possible to use wildcards for the tag and source fields. That way it is possible to check for or receive any kind of messages from any other processor in the specified communication group.

```
Source = MPI_ANY_SOURCE;
Tag    = MPI_ANY_TAG;
```

In this MRP implementation, MPI_ANY_SOURCE is heavily used, since each processors is broadcasting information, it is not appropriate to specify the source, since the source may be any other processor. On the other hand, the messages need to be tagged appropriately in order to distinguish messages of different types. Some of the message tags used in the code are CLOSED_NODE, FIND_MIN, EXPAND_NODE, WORK_REQUEST, and FINISHED.

9.4. Translation of csend as a Broadcast Message

While rehosting the code one of the most difficult tasks was to implement broadcast messages. In NX, this is done buy just adding a "-1" in the destination section of the csend message. MPI has a new function for this purpose, MPI_Bcast.

Its syntax is :

```
Int MPI_Bcast(
        Void*           message     /* in/out    */,
        Int             count       /* in        */,
        MPI_Datatype    datatype    /* in        */,
        Int             root        /* in        */,
        MPI_Comm        comm.       /* in        */)
```

This MPI form of broadcast does not have a tag field associated with it. In the MRP code, the messages are distinguished from each other by their tag fields. MPI_Iprobe also checks the tag field to see if there is a message waiting to be received. Then the only way to implement these broadcast messages without changing the whole structure of the code

is to use a send operation as many as the number of processors as seen in Table 9.5. This makes the code more complex, leading to a larger number of instructions for a simple broadcast message.

Table 9.5. Implementation of NX "csend" for Broadcasting a message in MPI.

NX	**csend(FIND_MIN,&TheBest,sizeof(TheBest),-1,NODE_PID);**
	MPI_Bcast (&TheBest, 1, MPI_FLOAT, Iam, MPI_COMM_WORLD); This is not used, instead the structure below is used.
MPI	for (v=1;v<NumNodes;v++) MPI_Send(&TheBest, 1, MPI_FLOAT v, FIND_MIN, MPI_COMM_WORLD);

9.5. Translation of Reductions

Another group of functions used in the code is reductions. Reductions are functions that require all the processors in the group to participate to calculate a certain value. Every processor sends its own value, and they receive the values from other processors. According to the type of reduction, minimum, maximum, average, etc. NX has different functions for all these reduction types, such as gslow, gihigh, etc. MPI has one function name for reductions, where only the parameters change depending on the reduction type. MPI_Reduce makes the result available for only the calling processor, whereas MPI_Allreduce makes it available for all the participating processors. For this type of instructions the code requires synchronization. Even if the result is not available, the code may progress until this synchronization, then it has to wait for the result of the reduction. This is done with MPI_Barrier function. It causes each process in comm to block until every process in comm has called it. These instructions are shown in Table 9.6.

```
Int MPI_Allreduce(
        Void*              operand     /* in   */,
        Void*              result      /* out  */,
        Int                count       /* in   */,
        MPI_Datatype datatype          /* in   */,
        MPI_Op             operator    /* in   */,
        MPI_Comm           comm.       /* in   */)
```

```
Int MPI_Barrier(
            MPI_Comm    comm. /* in */
```

Table 9.6. MPI counterparts of the NX commands "gslow", "gihigh", and "gsync".

NX	**gslow(&TheBest,1, &Temp);**
MPI	MPI_Allreduce(&Temp,&TheBest,1,MPI_FLOAT,MPI_MIN,MPI_COMM_WORLD);
NX	**gihigh(&Qmax,1,&Qtemp);**
MPI	MPI_Allreduce(&Qtemp,&Qmax,1,MPI_INT,MPI_MAX,MPI_COMM_WORLD);
NX	**gsync();**
MPI	MPI_Barrier(MPI_COMM_WORLD);

9.6. Translation of Time-Recording Instructions

To get the execution time of the code, MPI_Wtime is used. The structure and code sequence for this is :

```
Double MPI_Wtime(void);

StartTime     = MPI_Wtime();             /* Right after MPI_Init           */
EndTime       = MPI_Wtime();             /* Right before MPI_Finalize      */
MyTotalTime = EndTime - StartTime;       /* Anywhere after both available  */
```

The time is recorded as ticks, and then it is converted into seconds. This time is referred to as wall clock time. It includes everything from the StartTime until Endtime, such as CPU time, idling time, I/O, disk access time, etc.

9.7. Creating Separate Communication Groups

Creating new communication groups within the global MPI_COMM_WORLD is an important process in the MRP, for example, the multiple target case is handled with communication groups in the code. The variable definition to create MPI groups is as follows:

```
int     Iam, NumNodes, workerid, workersize, workerid2, workersize2,
        workersallsize, workersallid;
MPI_Comm world, workers, workers2, workersall;
MPI_Group world_group, worker_group, worker_group2;
int ranks[50];
int CONTROLLER;
int NUMBER=2;
```

When the variables are defined the rest of the work to create groups is done after MPI initializes. The code for that purpose is:

```
world=MPI_COMM_WORLD;
MPI_Comm_group(world, &world_group);

/* This for loop is used to exclude first half of the processors */

for (x=0;x<(NumNodes/NUMBER);x++)
ranks[x]=x;
MPI_Group_excl(world_group, (NumNodes/NUMBER), ranks, &worker_group);

/* This for loop is used to exclude second half of the processors */

for (x=NumNodes/NUMBER;x<NumNodes;x++)
ranks[x-(NumNodes/NUMBER)]=x;
MPI_Group_excl(world_group,(NumNodes-NumNodes/NUMBER),ranks,
            &worker_group2);

/*Workers is created and consists of the last half of the processors.*/

MPI_Comm_create(world, worker_group, &workers);

/*Workers2 is created and consists of the first half. */
MPI_Comm_create(world, worker_group2, &workers2);

/* Groups that are no longer needed are freed.*/
MPI_Group_free(&worker_group2);
MPI_Group_free(&worker_group);
MPI_Group_free(&world_group);
```

9.8. Manipulation of Communication Groups in the MRP

Processors initialize their variables according to their group defaults. After that, the same variables are called, but they have different values for different group members. That way it is possible to write the same code and have the groups execute the same instructions with their group defaults.

```
if(Iam>=(NumNodes/NUMBER)) /* The last half of the processors.*/
{
MPI_Comm_rank( workers, &workerid);
MPI_Comm_size( workers, &workersize);
CONTROLLER=(NumNodes/NUMBER);
workersall=workers;
workersallsize=workersize;
workersallid=workerid;
}
else /* the first half of the processors.*/
{
MPI_Comm_rank( workers2, &workerid2);
MPI_Comm_size( workers2, &workersize2);
CONTROLLER=0;
workersall=workers2;
workersallsize=workersize2;
workersallid=workerid2;
}
```

After the job of groups are done, and before MPI finalizes, the groups just created are also freed with the instruction

```
MPI_Comm_free(&workersall); /* Frees both groups. */
```

9.9. Summary

In Appendix C, some information is given about converting NX to MPI. Since the original code was written using NX instructions, which had a different format, some restructuring is required in order to use MPI instructions instead. Since MPI can be implemented on different platforms, rehosting the existing code using MPI provides interoperability for the implementation. A program can be written with only a few basic MPI commands, and it is enough. But, in order to establish a more complex program communication constructs, MPI offers a wide variety of instructions as well. As problems have been encountered during the porting part of this research effort, more complicated features of MPI are used. Appendix C explains the difficulties and the solutions for these mappings as well.

10. Appendix D – Matlab Code For Visualization

Below is the Matlab implementation used for visualizing the results. The explanations for instructions are given within the program.

```
clear all;
close all;

%Show the radars as spheres?
fShowRadars = 1;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This block creates the colormap%%%%%%%%%%%%%%%%%%%%

brown = [128 89 19]/255;
lightbrown = ([128 89 19]/255)*1.7;
earthColors = repmat(0,[64,3]);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Extra colormap option
for j = 1:64,
      earthColors(j,:) = (64-j)/64 * brown + j/64*lightbrown;
end;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This block reads the terrain data%%%%%%%%%%%%%%%%%%
filename = 'Terrain.s1';
permissions = 'rt';

fid = fopen(filename,permissions);

format = '%i';

m = fscanf(fid,format,[1,1]);
n = fscanf(fid,format,[1,1]);
kMaxElevationSteps = fscanf(fid,format,[1,1]);
kScaleFactor = fscanf(fid,format,[1,1]);
terrainData = fscanf(fid,format,[m,n])';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This block reads the route1 data%%%%%%%%%%%%%%%%%%%%
filename = 'Route.s5';
permissions = 'rt';

fid = fopen(filename,permissions);

format = '%i';

kRouteLength = fscanf(fid,format,[1,1]);
routeData = fscanf(fid,format,[4,kRouteLength])';
path = routeData(:,1:3);
```

```
path(:,3) = path(:,3)*kScaleFactor;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This block reads the route2 data%%%%%%%%%%%%%%%%%%%%
filename = 'Route.s3';
permissions = 'rt';

fid = fopen(filename,permissions);

format = '%i';

kRouteLength2 = fscanf(fid,format,[1,1]);
routeData2 = fscanf(fid,format,[4,kRouteLength2])';

path2 = routeData2(:,1:3);
path2(:,3) = path2(:,3)*kScaleFactor;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This block reads the radar data%%%%%%%%%%%%%%%%%%%%
filename = 'Myradars.3';
permissions = 'rt';

fid = fopen(filename,permissions);

format = '%i';

kNumRadars = fscanf(fid,format,[1,1]);
kRadarPower = fscanf(fid,format,[1,1]);
kTxGain = fscanf(fid,format,[1,1]);
kRxGain = fscanf(fid,format,[1,1]);
kWaveLength = fscanf(fid,'%f',[1,1]);
radarData = fscanf(fid,format,[4,kNumRadars])';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This block draws the terrain%%%%%%%%%%%%%%%%%%%%%%%
figure(1);
surf(terrainData);
meshc(terrainData);
surfl(terrainData);
%axis equal;
%axis vis3d;
%contour(terrainData);
hold on;
title('Mission Route Visualization');
xlabel('X-Coordinate');
ylabel('Y-Coordinate');
zlabel('Elevation/Altitude');
shading interp;
%shading flat;
colormap(earthColors);
colormap(gray);
colormap(jet);
colormap(copper);
rotate3d;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This block draws radar stations%%%%%%%%%%%%%%%%%%%%

for j = 1:kNumRadars,
      [X,Y,Z] = sphere(5);%sphere(radarData(j,4));
   X = X + radarData(j,1);
   Y = Y + radarData(j,2);
   Z = Z + radarData(j,3)*250;
   surf(X,Y,Z);
   shading interp;
end;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%This line plots the aircraft route
%plot3(path(:,1),path(:,2),path(:,3),'black');
%This block draws threat dots as well %%%%%%%%%%%%%%%%%%%%%%%

for j = 1:kRouteLength,
   plot3(path(j,1),path(j,2),path(j,3),'black');
   if routeData(j,4),
      plot3(path(j,1),path(j,2),path(j,3),'r.','MarkerSize',20);
  end;
end;

for j = 1:kRouteLength2,
   plot3(path2(j,1),path2(j,2),path2(j,3),'white');
   if routeData2(j,4),
      plot3(path2(j,1),path2(j,2),path2(j,3),'r.','MarkerSize',20);
   end;
end;
%The End
```

Bibliography

[1] Bahnij, Maj Robert B. A Fighter Pilot's Intelligent Aide for Tactical Mission Planning. MS thesis, AFIT/GCS/ENG/85D-1, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1985

[2] Bradshaw, 2Lt Jeffrey S. A Pilot's Planning Aid for Route Selection and Threat Analysis in a Tactical Environment. MS thesis, AFIT/GCS/ENG/86D-11, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1986.

[3] Spear, Capt Jon L. Improvements to the AFIT Tactical Mission Planner. MS thesis, AFIT/GCS/ENG/88D-20, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1988

[4] Grimm, James J. Solution to a Multicriteria Aircraft Arouting Problem Utilizing Parallel Search Techniques. MS thesis, AFIT/GCE/ENG/92D-04, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1992

[5] Droddy, Vincent A. Multicriteria Mission Route Planning Using Parallelized A* Search. MS thesis, AFIT/GCE/ENG/93D-04, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1993.

[6] Gudaitis, Michael S. Multicriteria Mission Route Planning Using a Parallel A* Search. MS thesis, AFIT/GCS/ENG/94D-05, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1994.

[7] Skolnik, Merrill I., editor. Radar Handbook.McGraw Hill, 1990.

[8] Chandy, K. Mani and Jayadev Misra. Parallel Program Design: A Foundation. Addison Wesley Reading, MA, 1988.

[9] Glassner, Andrew S. Graphics Gems. Academic Press, 19990.

[10] Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis, "Introduction to Parallel Computing Design and Analysis of Parallel Algorithms", The Benjamin/Cummings Publishing Company, Inc. 1994.

[11] Kai Hwang, Zhiwei Xu, "Scalable Parallel Computing, Technology, Architecture, Programming", McGraw-Hill, 1998. [50] Determination of Algorithm Parallelism in NP-Complete Problems For Distributed Architectures, R. Andrew Beard, Gary B. Lamont, Department of Electrical and Computer Engineering School of Engineering Air Force Institute of Technology WPAFB, OH.

[12] Golden, Bruce L., et al. "The Orienteering Problem," Naval Research Logistics, 34:307-318 (1987).

[13] Pressman, Roger S., Software Engineering, A Practitioner's Approach, 4th edition, Mc Graw-Hill, 1997.

[14] Cormen, Thomas H., Charles, Leiserson E., Rivest, Ronald L., Introduction to Algorithms, Mc Graw-Hill, 1997.

[15] Shantanu Dutt, Nihar R. Mahapatra, Parallel A* Algorithms and their Performance on Hypercube Multiprocessors, University of Minnesota, Minneapolis, MN.

[16] Bolc, Leonard, Cytowski, Jerzy, Search Methods for Artificial Intelligence. Academic Press Ltd., 1992.

[17] Farhad Arbab, "Coordination Programming for Parallel and Distributed Applications", PDPTA '99 International Conference.

[18] Carroll, K. P., et. al. Autonomous Underwater Vehicle (AUV) Path Planning: An A* Approach to Path Planning with Consideration of Variable Vehicle Speed and Multiple, Overlapping, Time-Dependent Exclusion Zones. IEEE Symposium on AUV Technology. 79-84. 1992.

[19] Schulmeyer, G. Gordon and James, I. MacManus, editors. Total Quality Management for Software. Van Nostrand Rienhold, 1992.

[20] Pal, Prabir K. and K. Jayarajan. Fast Path Planning for Robot Manipulators Using Spatial Relations in the Configuration Space. IEEE International Conference on Robotics and Automation2. 668-673. 1993.

[21] Olsan, James, B. Genetic Algorithms Applied to a Mission Routing Problem. MS thesis, AFIT/GCE/ENG/93-12, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, 1993.

[22] Pacheco, Peter S., Parallel Programming with MPI, Morgan Kaufmann Publishers, Inc., California, 1997.

[23] Gropp, William et. al., Using MPI, Portable Parallel Programming with the

[24] Snir, Marc et. al., MPI The Complete Reference The MIT Press, England, 1996.

[25] Dutt, Shantanu and Nihar R. Mahapatra. Parallel A* Algorithms and their Performance on Hypercube Multiprocessors. Seventh International Parallel Processing Symposium. April 1993.

[26] Teng, Y. Ansel, et. al. Stealth Terrain Navigation, IEEE Transactions on Systems, Man, and Cybernetics, January 1993.

[27] Stiles, P. N. and I.S. Glickstein. Highly Parallelizable Route Planner Based On Cellular Automata Algorithms, IBM Journal of Research and Development, March 1994.

[28] Hart, P.E., et. al. A Formal Basis for the Heuristic Determination of minimum Cost Paths., IEEE Trans. Systems Science and Cybernetics. 1968.

[29] Flood, Merrill M. "The Travelling Salesman Problem," Journal of the Operational Research Society of America, 4:61-75 (1956).

[30] Laporte, Gilbert and Silvano Martello. "The Selective Travelling Salesman Poblem," Discrete Applied Mathematics, 26:193-207 (March 1990).

[31] Golden, Bruce L., et al. "A Multifaceted Heuristic for the Orienteering Problem," Naval Research Logistics, 35;359-366 (1988).

[32] Bramanti-Gregor, Anna. Strengthening Heuristic Knowledge in A* Search. PhD dissertation Wright-State University, 1993.

[33] Pellazar, Miles B. "Multi-Vehicle Route Planning with Constraints Using Genetic Algorithms," National Aerospace and Electronics Conference. May 1994.

[34] Brassard, Gilles and Paul Bratley. Algorithmics: Theory and Practice. Prentice-Hall, Inc., 1988.

[35] T. Agerwala, J. L. Martin, J. H. Mirza, D. C. Sadler, D.M. Dias, M. Snir, "Programming models in technical computing ", IBM Systems Journal, vol. 38, 1999.

[36] Pearl, Judea, Heuristics, Intelligent Search Strategies for Computer Problem solving.Addison-Wesley Publishing Company 1984.

[37] Shakley, Donald J. Parallel Artificial Intelligence Search Techniques For Real Time Applications. MS thesis, AFIT/GCS/ENG/87D-24, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB OH, December 1987.

[38] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., Introduction to Algorithms, McGraw-Hill, 1997.

[39] Gregory M. Nielson, et al., Scientific Visualization, Overviews, Methodologies, and Techniques. IEEE Computer Society, Los Alamos, California, 1997.

Vita

Ergin Sezer was born in Kastamonu, Turkey on April 8, 1972. He graduated from Kuleli Military High School in Istanbul, Turkey in 1990. He attended the Turkish Air Force Academy and studied Computer Engineering. He graduated as a 2nd lieutenant on the 30th day of August 1994. His first assignment was Euro-NATO Joint Jet Pilot Training on Sheppard Air Force Base, Wichita Falls, TX. He graduated as a pilot on the 5th day of April 1996 as a 96-04 graduate. He finished F-5 and F-16 courses successfully and assigned to 182nd "Atmaca Filo" Squadron on 8th Jet Base Diyarbakir, Turkey. After spending one year in his squadron, in August 1998, he was assigned to the School of Engineering and Management, Air Force Institute of Technology, Wright-Patterson Air Force Base, Dayton, OH to complete a Master of Science degree in Electrical and Computer Engineering. He studied Computer Communication Networks & Parallel Processing and High Speed Computing. His next assignment is at 8nci Ana Jet Us 182nci Filo Diyarbakir-Turkey.

Permanent Address: Alparslan Türkeş Bulvarı
Sezer Apt. B Blok Kat 3 D:5
37200 Kastamonu-Turkey

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*
10 March 2000

2. REPORT TYPE
Master's Thesis

3. DATES COVERED *(From - To)*
APR 1999 - MAR 2000

4. TITLE AND SUBTITLE
MISSION ROUTE PLANNING WITH MULTIPLE AIRCRAFT & TARGETS USING PARALLEL A* ALGORITHM

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Sezer, Ergin, 1st Lt, TUAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 P Street, Building 640
WPAFB, OH 45433-7303
DSN: 785-2811 x 4364

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCE/ENG/00M-04

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFRL/SNAT
Attn: Malhotra, P. Raj
U.S. Air Force Research Laboratory, Sensors Directorate
2241 Avionics Circle
WPAFB, OH 45433
Ph: 937/255-1115 x 4291

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES
Professor Gary B. Lamont, CIV AFIT/ENG, Ph: 937 255-3450 x 4718

14. ABSTRACT
The general Mission Route Planning (MRP) Problem is the process of selecting an aircraft flight path in order to fly from a starting point through defended terrain to target(s), and return to a safe destination. MRP is a three-dimensional, multi-criteria path search. Planning of aircraft routes involves an elaborate search through numerous possibilities, which can severely task the resources of the system being used to compute the routes. Operational systems can take up to a day to arrive at a solution due to the combinatoric nature of the problem, which is not acceptable, because time is critical in aviation. Also, the information that the software is using to solve the MRP may become invalid during the computation. An effective and efficient way of solving the MRP with multiple aircraft and multiple targets is desired using parallel computing techniques. Processors find the optimal solution by exploring in parallel the MRP search space. With this distributed decomposition the time required for an optimal solution is reduced as compared to a sequential version. We have designed an effective and scalable MRP solution using a parallelized version of the A* search algorithm. Efficient implementation and extensive testing was done using MPI on clusters of workstations and PCs.

15. SUBJECT TERMS
Parallel A* algorithm, Multicriteria Mission Route Planning, Message Passing Interface (MPI), Heuristic Search, Parallel Processing, Optimization, NP-Complete Problems.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Lamont, B. Gary, Professor, CIV AFIT/ENG
Unclassified	Unclassified	Unclassified	UL	168	19b. TELEPHONE NUMBER *(Include area code)* 937 255-3450 x 4718