NOTE TO USERS

This reproduction is the best copy available.

UMI®

UNIVERSITY OF CINCINNATI

March 14, 19 91

I hereby recommend that the thesis prepared under my supervision by KOULIN HU

entitled Parallel Algorithms and Architectures For Dynamic Programming Methods in Trajectory Optimization

be accepted as fulfilling this part of the requirements for the degree of DOCTOR OF PHILOSOPHY

Approved by:

G. L. Slater G L Slater

G. A. Becus Georges A. Bé

R. J. Kroll R J Kroll

H. W. Carter Harold W. Carter

PARALLEL ALGORITHMS AND ARCHITECTURES FOR DYNAMIC PROGRAMMING METHODS IN ON-LINE TRAJECTORY OPTIMIZATION

A Dissertation submitted to the

Division of Graduate Studies and Research

of the University of Cincinnati

in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in the Department of Aerospace Engineering and Engineering Mechanics

of the College of Engineering

1990

by

Koulin Hu

M.S., Beijing Control Devices Institute, P.R. China, 1986

B.S., Nanjing Aeronautical Institute, P.R. China, 1983

Advisor: Dr. G.L. Slater

Abstract

An approach toward the development of parallel algorithms for the on-line trajectory optimization problem is investigated in this dissertation. The model for the problem is the need to compute optimal flight paths in real time in a Terrain Following/Terrain Avoidance/Threat Avoidance mission for a helicopter. The problem is formulated as an optimal control problem by dynamic programming so as to deal with a complex constraint environment. In order to make it feasible to compute an optimal trajectory, first, the entire computational state space is broken into several reduced band regions. Then several different parallel algorithms based on the corresponding parallel hardware architectures are designed for the models of different complexity. For a simple model, the parallel algorithms and their respective hardware implementations are designed by a direct, heuristic method. For a more complex model, a systematic method is synthesized to design a parallel algorithm using the methodology of the State Increment Dynamic Programming. These algorithms are suitable for the VLSI implementation on parallel architectures.

Acknowledgement

I wish to express my deepest respects and gratitude to my advisor, Dr. Gary L. Slater, for his guidance and support through out my stay as a graduate student at the University of Cincinnati. Without his guidance and support during this research, this dissertation could not be accomplished. I would also like to especially thank Dr. George A. R. Becus, Dr. Robert J. Kroll, and Dr. Harold W. Carter as the members of my dissertation committee for their valuable advice on this dissertation. Finally, I appreciate my wife, Haiyan Han, who patiently endured my absence and allowed me to concentrate on this dissertation, and whose concern and love was the continuous source of inspiration.

This research was supported in part by NASA Ames Research Center under grant NAG2-175.

Table of Contents

List of Figures

1.0 INTRODUCTION

1.1 Introduction

The study of trajectory optimization for aircraft has been an important research topic in recent years. One optimization problem of interest is the on-line determination of trajectories including realistic performance limitations on the vehicle and complex cost constraints due to the operational environment[Cal-2,Men,Sla-3,TAU]. An extreme example of a helicopter in a complex environment is, for example, determination of an optimal flight path for a terrain following/terrain avoidance (TF/TA) type of mission. A more civilian application which encounters similar difficulties is, for example, in the Air traffic Control (ATC) problem.

The importance and the practical value of on-board, real-time algorithms have long been recognized[Den-1,Sla-1,Wen]. "Traditional" shooting or iterative schemes designed for conventional sequential computing algorithms are inherently slow. To decrease computation time enough to allow for on-line computation, parallelization becomes necessary. With the development of modern microcomputers and Very Large Scale Integration (VLSI) technology, it is now feasible to implement on-line calculation of an optimal trajectory, as well as that of guidance and navigation functions.

The optimal TF/TA flight problem can be thought of as a means for processing the global route, accounting for real-time sensor information, lateral and vertical maneuvering degrees of freedom, local topography, and current aircraft state information, to obtain the desired three dimensional flyable trajectories. Many early attempts for TF/TA flight path optimization problems centered on decoupled approaches, where the vertical channel and horizontal channel are treated separately. Typically, in such decoupled approaches, go-over or go-around decisions are made periodically, based on any number of criteria such as flight time, fuel consumption, aircraft performance characteristics, and so on. Very often, such decoupled approaches have proven to be unsatisfactory. This is due to the lack of a unified treatment of vertical and horizontal maneuvers.

Some recent approaches[Den-1,TAU,Wen] to the problem of TF/TA trajectory determination use the concept of separating the trajectory generation and trajectory control functions. The trajectory generation function computes a reference TF/TA trajectory which is optimal with regard to some combined vertical/horizontal performance index. The trajectory control function supplies appropriate guidance commands to the aircraft flight control system to capture and track the desired optimal trajectory.

The main aim of this dissertation research is to examine the application of dynamic programming to the type of complicated trajectory optimization problems, and in particular to examine parallel computing methods for the efficient solution of such

problems. These parallel methods are studied through the application of VLSI technology which allows real-time computation of an optimal trajectory. First, a direct method is applied to a velocity-invariant model for aircraft flight path optimization by the dynamic programming method through a complex constraint region. The parallel algorithms and their respective hardware implementations are presented. Then a more complex model is considered and the corresponding approach to on-line computation of trajectories is studied by the State Increment Dynamic Programming (SIDP) method. A systematic method is synthesized to design the parallel algorithm of the SIDP for the trajectory optimization in a complex constraint environment. Such an algorithm is suitable for the VLSI implementation on a parallel architecture.

1.2 Survey of Trajectory Optimization Problems

The unique and low-speed capabilities of a helicopter have made this vehicle an important mode of transportation for many applications. One of the first attempts at helicopter flight path optimization was done by Schmitz[Sch-1], who investigated the take-off problem for a heavily loaded helicopter using a variational approach and later tested a "suboptimal" implementation of this control policy[Sch-2]. The work of Olsen[Ols] was directed at on-board optimization of climb and cruise trajectories but utilized only a classical quasi-steady performance approach.

Using variational methods, the problems of optimal time and

optimal fuel trajectories for aircraft generally require the solution of nonlinear two-point boundary-value problems. It is generally impossible to compute these problems in real time. To simplify the solution of the problems , many authors[Bar,Bry,Cal-1,Sla-1] have used reduced-order modeling techniques such as energy-state approximations.

An algorithm based on energy-state method was derived by Barman and Erzberger[Bar] for calculating optimum trajectories with a range constraint. The derivation of the algorithm further assumed that each optimum profile consists of at most three segments, namely, climb, cruise, and descent. This assumption allows energy to be an independent variable, therefore eliminating the integration of a separate adjoint differential equation and simplifying the calculus-of-variation problem to one requiring only point-wise extremization of algebraic functions. This algorithm was used to compute optimum fuel and optimum time trajectories for CTOL short-haul aircraft.

Slater and Erzberger[Sla-1] developed the method further and applied it to helicopter flight. They also developed a synthesis procedure to allow on-board generation of "optimal" trajectories for arbitrary weight and wind conditions. In particular, trajectories are determined which minimize a cost function chosen as a weighted sum of time and fuel. The algorithm was applied to obtain optimal climb-cruise-descent trajectories for the S-61N helicopter.

These solutions are restrictive in that either turning or position

dynamics were ignored[Hen,Kel,Par], or assumptions were made concerning the existence of a cruise point as part of the optimal trajectory[Bar,Bry,Hen,Par,Sla-1,Sla-2]. Under varying sets of assumptions, algorithms were developed that could be used for on-line optimal trajectory determination and control.

Calise has shown that singular perturbation methods are useful for extending energy-state modeling approaches to a more general problem formulation[Cal-1,Cal-2,Cal-3]. These methods constitute a reduced-order analysis approach where the system dynamics are separated into slow and fast modes. This permits the solution of high-order problems to be approximated by the solution of a series of low-order problems. Assuming appropriate measurements are available, the developed algorithms are useful for on-board trajectory optimization and control.

But when the performance index becomes more complex, not only including time/fuel optimality but also requiring TF/TA, threat avoidance, and the Nap-of-the-Earth (NOE) environment, then the algorithms based on singular perturbation methods may no longer be effective on-line algorithms. For TF/TA flight the complex performance index includes the cost related to a terrain database stored in computer memory and most probably need to be updated on-line by on-board sensed information. The performance index also includes the cost concerned with threats which may not be known in advance. Therefore the variational Hamiltonian for singular perturbation method involves the above two cost terms to be updated as well as nonlinear

dynamic equations. The necessary conditions of optimal solutions derived from this Hamiltonian may be difficult to be calculated in real time.

Due to the difficulty of the singular perturbation method in dealing with terrain and threat constraints, other method have to be examined and applied. Dynamic programming may be an effective technique for the problem of on-line trajectory optimization with complex constraints because one of the advantages of dynamic programming is to easily deal with complex and irregular constraints that may be updated on-line. However, the inherent disadvantage of dynamic programming, or "curse of dimensionality"[Bel], is encountered. In order to overcome this disadvantage and to meet the real-time requirement, parallel computational algorithms are studied. Parallel algorithms require high quality in computation speed and memory for an on-board computer. Fortunately, recent advances in the design and fabrication of VLSI circuits have shown that it is possible to construct highly parallel cellular computers with hundreds or thousands of processing elements[MIT] in the near future.

It is noted that the number of states required in a model which describes the three-dimensional flight of aircraft increases largely over the two-dimensional flight case and that the computational time of dynamic programming grows exponentially with the number of states. Even if the usable state space is reduced to a specific space band region based on an initial reference trajectory which is generated by the desired waypoints, it is very difficult by the dynamic programming

algorithm to implement the on-line computation of an optimal trajectory. From the standpoint of parallelism, we have to explore an algorithm in parallel so as to make real-time operation practical.

The design of algorithms to achieve real-time TF/TA trajectories is an extremely challenging[Den-2,Hof,Sla-3,Wen]. One algorithm that has been demonstrated is the Feasible Direction Algorithm[Wen]. This algorithm development was worthwhile in quantitatively demonstrating the value of TF/TA, based on perhaps the first mathematically rigorous and potentially feasible approach to TF/TA. Testing of this algorithm in the Flight Dynamics Laboratory has provided insight into its strengths and weaknesses.

An alternate algorithm for TF/TA flight of a helicopter was developed by the TAU corporation[Pek,TAU] and termed "Dynapath" . The algorithm defines the curvature of a trajectory as an only control variable and chooses a mixture of tree searching and dynamic programming. The tree structure is used to handle the aircraft dynamics and dynamic programming reduces the number of possible trajectories that need to be considered. Two methods were used in computing trajectories. The first one was to assume that a three-dimensional trajectory can be decoupled into a ground track and a vertical profile. The ground track is first calculated by control of 5 values of the curvature; the vertical profile is then calculated by control of 3 values of the curvature. they are then combined to form the three-dimensional trajectory. The second one was to directly calculate trajectories in three-dimensional case by a control of 15

values of the curvature. The latter approach was not able to be implemented in real time due to computational constraints. All other parameters of the computed trajectory are calculated from an only control variable or the curvature by a group of algebraic equations. The problem may occur in that the varying velocity and turning rate may not be constrained sufficiently, which means that the maneuvering ability may not be assured along the computed "optimal" trajectory.

Using the Dynapath, Researchers at NASA Ames Research Center conducted a piloted simulation on the Vertical Motion Simulator and developed the guidance algorithm and flight path control system under the condition of constant forward flight[Dor]. The Center further investigated and evaluated the simulation system to develop a low-level, maneuvering penetration guidance law. The results indicated that modification is needed to reflect capabilities of helicopters for pilot acceptance[Swe].

Other authors have used a more conventional variational approach to determine terrain following trajectories. Menon and Kim[Men] presented the method to computer terrain following trajectories. They considered that a constant of motion can be invoked to simplify the solution procedure of a terrain following trajectory. The optimal route to a desired final state constraint depends on the selection of the initial value of the heading angle. The solution to an optimal terrain following trajectory requires to solve an optimal control problem which involves a performance index and three first-order ordinary differential equations. The performance index is linearly

combined by time and terrain mask. The resulting algorithm also requires first and second partial derivatives of the surface describing the terrain, which are calculated by using a cubic spline parameterization of the digital terrain elevation data.

1.3 Organization of Dissertation

This dissertation is organized as follows. Chapter 2 states the problem formulation for the trajectory optimization in a complex constraint environment such as the TF/TA. The energy-state model of a helicopter is synthesized in Section 2.2. Under the consideration of the TF/TA flight, the performance index is defined in Section 2.3 to optimize the TF/TA trajectories.

Chapter 3 give a discussion on VLSI technique available for the design and implementation of parallel algorithms. Some characteristics about systolic arrays in VLSI technology are briefly stated in Section 3.2. The description of the Single-Instruction-Multiple-Dataflow (SIMD) and the Multiple-Instruction-Multiple-Dataflow (MIMD) machines are given in Section 3.3 and the trade-off between systolic arrays and array processors is discussed.

Chapter 4 presents an approach to parallel optimal trajectory computation in the case of using a simple model. The problem formulation based on two-dimensional motion is described in Section 4.2. Two algorithms are designed in Section 4.3 to calculate

trajectories using heuristic parallel architectures. The first systolic algorithm is formulated as a string of equivalent "matrix-by-vector" multiplication operations executed on linear arrays in Section 4.3.1. The second systolic algorithm is formulated as a string of equivalent "matrix-by-matrix" multiplication operations and executed on rectangular arrays in Section 4.3.2. The computational models of the linear arrays and rectangular arrays are also given. The numerical results of the complex 2-D trajectory optimization are presented using the parallel algorithms simulated on a single microcomputer in Section 4.4. The two algorithms and architectures are compared with each other in Section 4.5.

Chapter 5 presents a methodology for the general design of parallel algorithms for trajectory optimization, where we take the serial SIDP algorithm as an example. First, Section 5.2 describes an ad hoc method to reduce the computational state space to a small subspace of the actual state space.In Section 5.3, the parallel computation of the dynamic programming algorithm is reviewed and the SIDP is summarized. In Section 5.4, the method, based on the data dependence structure of an algorithm, is synthesized to design the parallel algorithm for the serial SIDP algorithm. The block structure of the parallel computer system is specified in Section 5.5 to implement the parallel algorithm for on-line trajectory optimization.

Finally, conclusions and recommendations are given in Chapter 6.

2.0 PROBLEM FORMULATION

2.1 Introduction

The amount of difficulty and expense experienced in calculating optimal flight paths depends primarily upon the complexity of the model used to describe the performance of the aircraft and also upon the kind and type of computer available for on-line computation.

The models can range from a simple two-state representation to quite complicated dynamic equations that include a full six-degree-of-freedom representation plus controls of surface equations. For trajectory optimization in the TF/TA environment, the aircraft is only capable of flying at low speed. A two-dimensional model may be used by treating only two kinematic equations for horizontal position and neglecting all the acceleration. A further improvement in accuracy, at the sacrifice of added complexity, is to treat velocity as a state variable with longitudinal acceleration, flight path, angle and heading angle as control variables, where the acceleration normal to the flight path is neglected. Finally a more accurate approximation is to treat flight path angle and heading as state variables with angles of attack and bank as control variables, where the mass of aircraft may be approximated by a function of time.

For trajectory determination problems, but especially for the

on-line case, it is important to use a dynamic model which contains the significant features of a trajectory but is not so complex as to prelude solutions. Indeed it is this dilemma which leads us to consider parallelization as a means of achieving sufficient complexity while still ensure a solution in real-time sense.

In this dissertation, we will consider two models of varying complexity. The important points to be brought out in this comparison is that not only is the computing algorithm different for these cases due to the varying complexity, but even the suggested computer hardware must be different.

2.2 Energy-State Model

For many flight mechanics problems, the energy-state method has been used to formulate the problem[Bry,Sla-1]. The method of energy-state approximation[Bar,Bry] assumes that the total energy can be considered as a state variable in a system and hence is continuous, while the kinetic energy (speed) and potential energy (altitude) components are generally not required to be continuous. The advantage of this model is that the important features of kinetic and potential energy interchange are captured, while only requiring a single state variable. This allows us to have the significant simplification to the full dynamic equations at little penalty. In other application, altitude is restrained as a state variable and hence continuity of altitude and velocity is ensured. This latter approach simplifies only

one equation of motion where the vehicle energy rate may be simpler to express than the acceleration.

Energy-state models for a helicopter are synthesized in this section. We consider the total energy or the sum of kinetic and potential energies:

$$E = mgh + \frac{1}{2} mv^2 \tag{2.1}$$

where m is the helicopter mass which is assumed constant, g is the gravity of the earth, and E is the total energy. Equation (2.1) is generally normalized by dividing by vehicle weight, giving

$$E_n = h + \frac{v^2}{2g} \tag{2.2}$$

The normalized energy rate is expressed as

$$\dot{E}_n = P_E(v,h,W,\pi) - P_D(v,h,W) \tag{2.3}$$

where P_E is the normalized available engine power, π is the engine parameter, and P_D is the dissipated power. Both are functions of velocity v, altitude h, and weight W.

The equation (2.3) is illustrated in Figure 2-1 for given h and W. This function varies with weight as shown in Figure 2-2 for different W at sea level. A detailed computation method for P_D was given by Slater and Stoughton[S1a-4].

The kinematic equations are

$$\dot{x} = v\cos\gamma\cos\psi \tag{2.4}$$

$$\dot{y} = v\cos\gamma\sin\psi \tag{2.5}$$

$$\dot{h} = v\sin\gamma \tag{2.6}$$

where (x,y,h) are the position coordinates, v is the velocity, γ is the flight path angle, and ψ is the heading angle. The equations (2.3)-(2.6) can be thought as a four-state model in which x, y, h and E_n are state variables and P_E, γ, and ψ are control variables. In this formulation process, velocity v becomes an auxiliary variable because v can be calculated from the normalized energy E_n and the altitude h or

$$v = [2g(E_n - h)]^{1/2} \tag{2.7}$$

To further improve the approximation, we consider including the rolling and pitching dynamics if the added complexity is covered by the computation capacity of the on-board computer. The dynamic equations in the heading angle and the flight path angle are

$$\dot{\psi} = \frac{T\cos(\gamma+\alpha)\sin\phi}{mv\cos\gamma} \tag{2.8}$$

$$\dot{\gamma} = \frac{T\cos\alpha\cos\phi}{mv} - \frac{g\cos\gamma}{v} \tag{2.9}$$

where T is the thrust, α is the attack angle, and ϕ is the bank angle as defined in Figure 2-3[sla-4], where T.P.P. stands for tip path plan. A six-state model is represented by the equations (2.3)-(2.9),

where x, y, h, E_n(or v), ψ, and γ are state variables; P_E, ϕ, and α are control variables. In this model, it is assumed that lift balances the component of weight perpendicular to the flight path.

Equation (2.3)-(2.9) are system equations which can be used to generate an optimal trajectory in 3-dimensional real space with variable velocity, turning rate, and flight path angle. Generally in this model, additional dynamical constraints must be considered when computing trajectories. They are expressed as, for example,

$$\begin{aligned} v &\le v_m \\ P_E &\le P_{max} \\ \alpha &\le \alpha_m \\ \phi &\le \phi_m \end{aligned} \qquad (2.10)$$

where v_m is the maximum velocity allowed. The maximum continuous power P_{max} depends on the helicopter engines and flight altitude. is the maximum angle of attack. The maximum bank angle ϕ_m is determined by the maneuver requirement. The larger the value ϕ_m is, the more maneuvers are needed.

2.3 Performance Index

The flight considered in this dissertation is conducted in very close proximity to the ground track in three-dimensional space and in varying course, airspeed, and altitude in order to take the maximum

advantages of cover and concealment offered by terrain, vegetation, and man-made features.

The computation of a trajectory in a complex terrain environment such as discussed here is subject to a number of conflicting factors. We resolve these factors by defining a performance index which is a linear combination of four weighted items, while considering a trajectory that achieves better terrain masking, minimum flight time, and minimum exposure to threats. The approach assumes that the terrain is known exactly. In a realistic implementation, the terrain information would come from the Digital Terrain Elevation Database (DTED) . The data from the DTED are usually coarse and possibly inaccurate. Hence, realistically, the real time information supplied by on-board sensors is blended with the DTED and the combined data are used to determine a terrain model. The performance index is assumed to be

$$\min J = \sum_{k=1}^{N} [\theta_1 D_k + \theta_2 \delta t_k + \theta_3 (h_k - h_c)^2 + \theta_4 P_k] \qquad (2.11)$$

where $\theta_1, \theta_2, \theta_3, \theta_4$ are weighting coefficients that satisfy

$$0 < \theta_1, \theta_2, \theta_3, \theta_4 < 1$$
$$\theta_1 + \theta_2 + \theta_3 + \theta_4 = 1 \qquad (2.12)$$

where D_k is the distance normal to the smoothed terrain. δt_k is one step time. h_c is a desired flight altitude above ground altitude and could be a function of local region for particular application. P_k is

the cost due to the threats, which is associated with helicopter position (x,y,h) and whose distribution can be specified according to the characteristics of threats. For example in Chapter 4, a weighted Gaussian distribution is assumed.

2.4 Summary

The models to describe the aircraft performance for trajectory computation can range from a simple two-state representation to a complex model that includes the full six-degree-of-freedom equations for the aircraft. The model selection is dependent of the accuracy requirement and the applied computer system. The energy-state model for a helicopter is given here and the performance index is defined as a linear combination of four weighted items for trajectory optimization in the TF/TA environment.

Fig. 2–1 Power Distribution

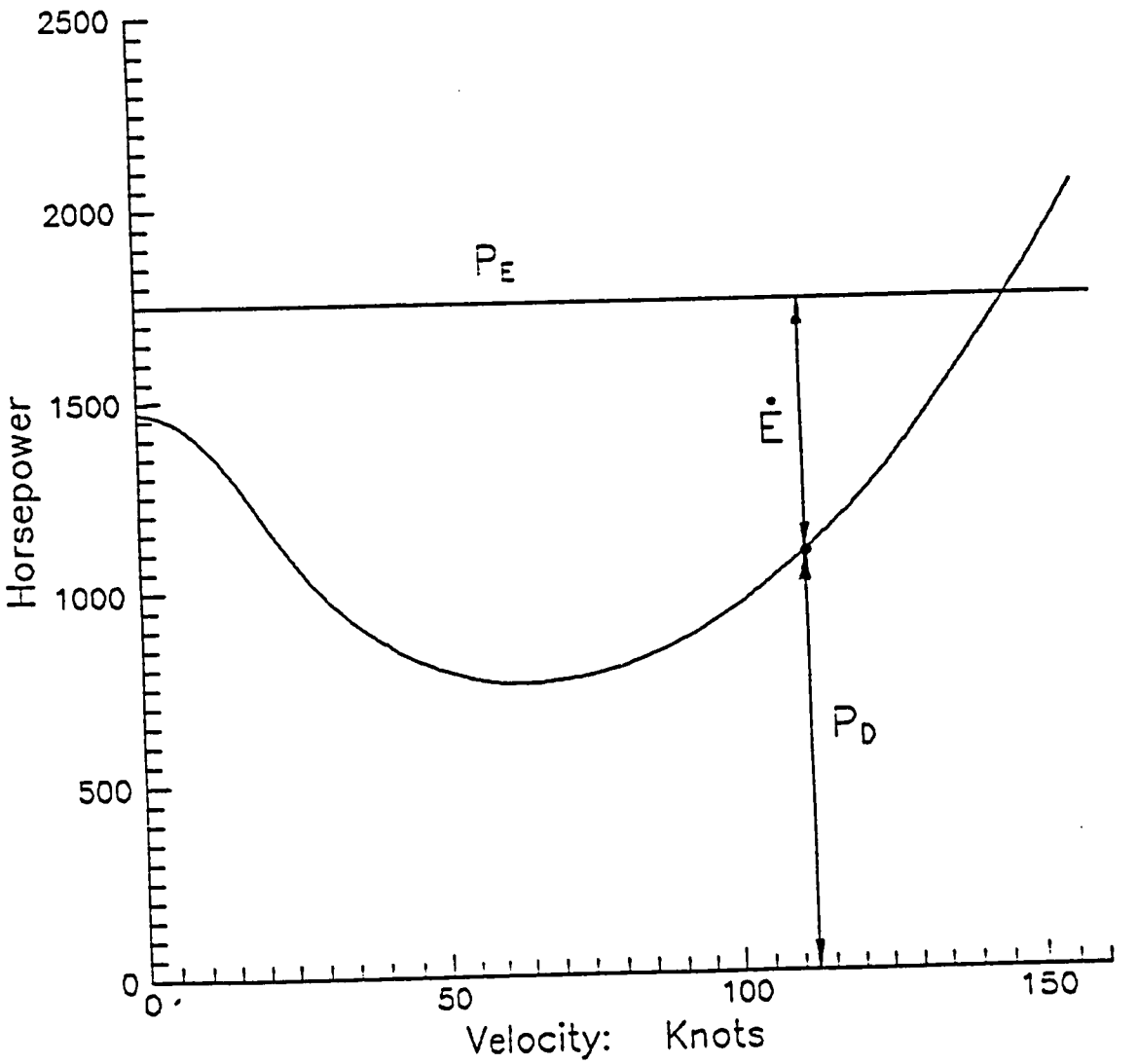
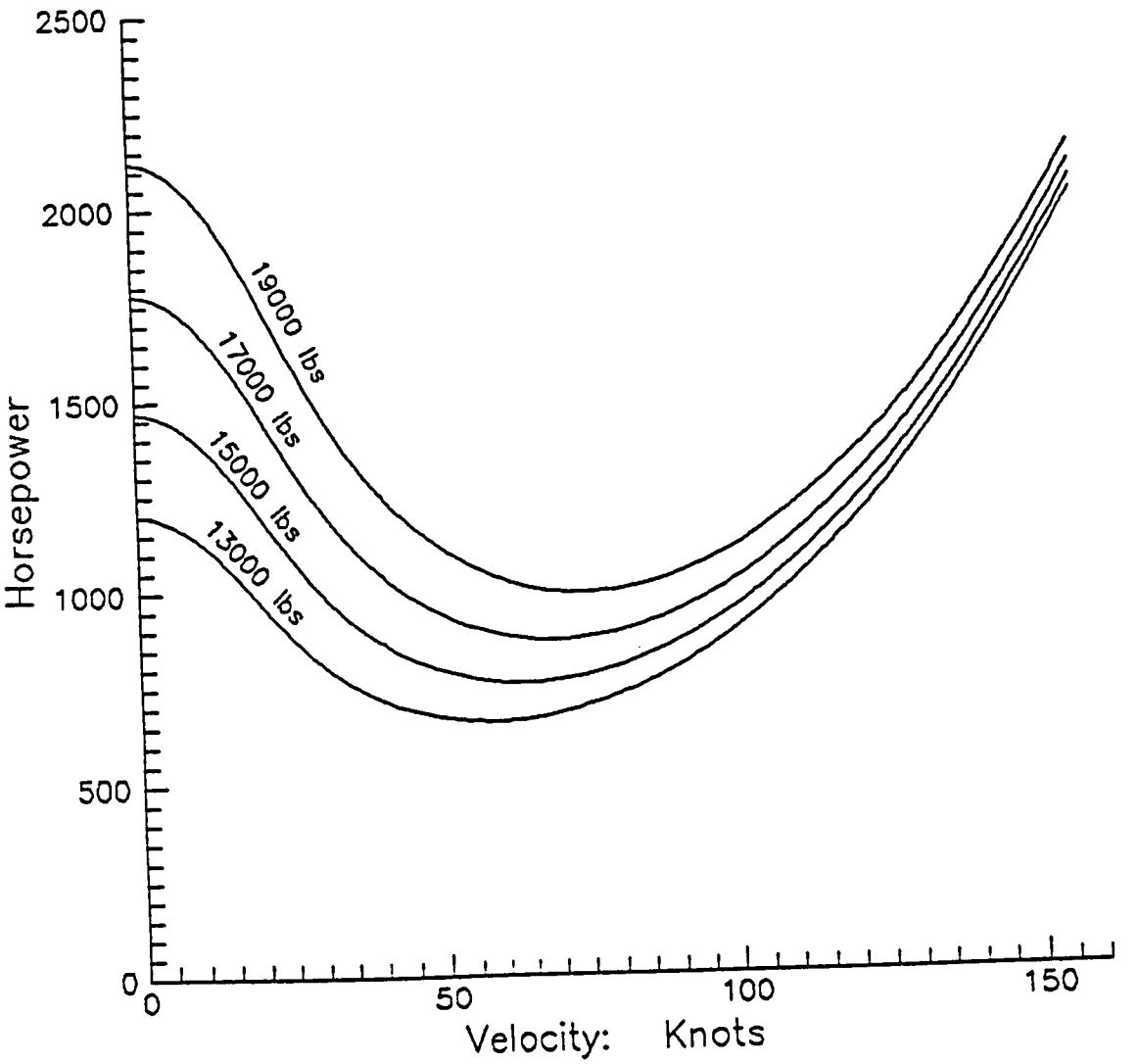


Fig. 2–2 Variation of Power Distribution with Weight



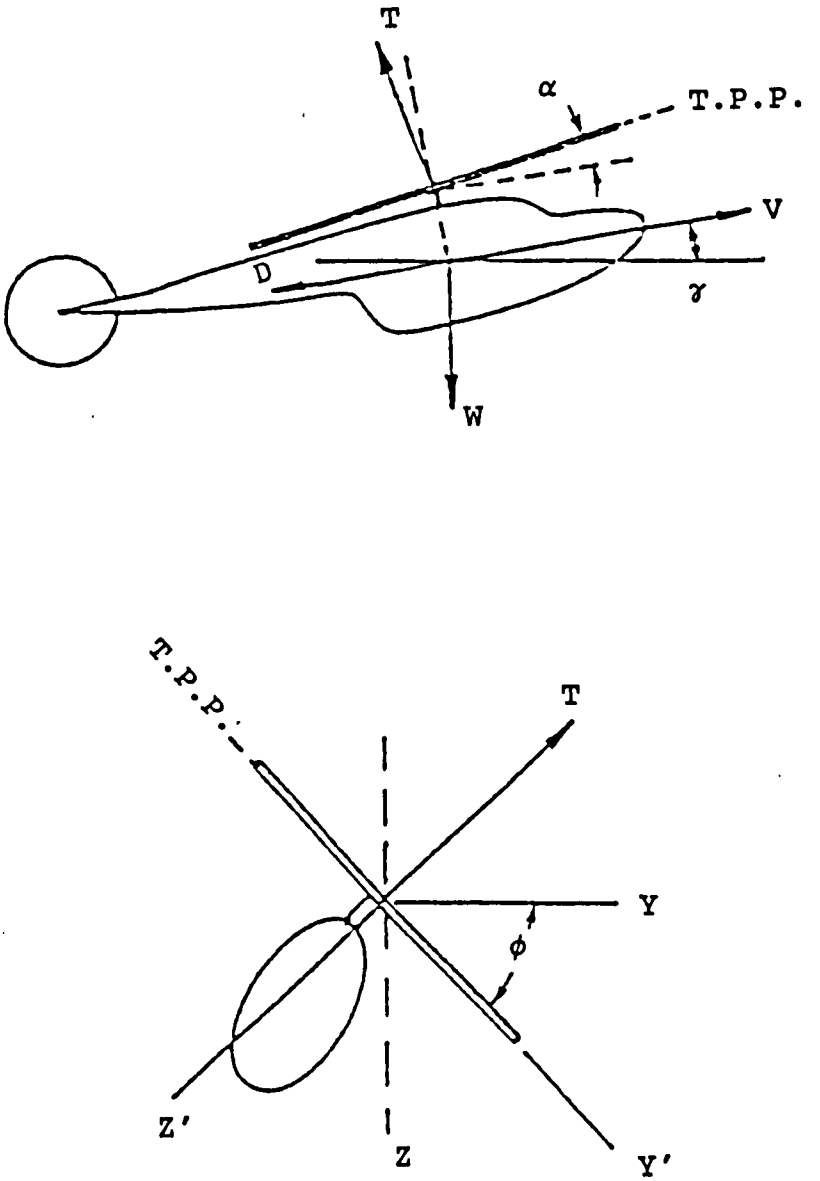


Fig. 2-3 Geometrical Definition of Model Variables

3.0 VLSI TECHNOLOGY FOR IMPLEMENTATION OF PARALLEL ALGORITHM

3.1 Introduction

Parallel algorithms and parallel architectures have been the subjects of intense research efforts in an attempt to achieve greater computational speed and computing capacity. In this dissertation, we examine the application of parallel algorithms to the on-line trajectory optimization problem.

At this time, these parallel methods are a conceptional model whose potential is only now being realized. Parallelism, most observers agree, can revolutionize computing from supercomputers to portable microcomputers. The development of a purpose-special small parallel computer may be even more exciting than the development of a purpose-general supercomputer[Gel].

A traditional parallel processing computer is a shared-memory multiprocessor system. This system comprises a collection of processors connected by a bus to a common pool of memory. The processors do not have their own local memory. In such a system, a processor performs parallel computations in such a way as to dedicate a complete processor to each active subproblem. Each subproblem is free to operate on memory without appreciable interference from the others. When more processors are added and a problem is further

partitioned, higher speeds are generally gained. But a question arises as more processors are added to a bus: When one processor tries to access an address in memory, it must first get permission from the others. Arbitration among the processors leads to contention. Each processor requires a finite amount of time to fetch something from memory, and while this is going on, the other processors must wait if they need data as well. Adding more processors simply makes it worse, and a bottleneck results.

This is the so-called *von Neumann bottleneck.* It is the reason that a multiprocessor system seldom has more than ten processors simultaneously operating in a common pool of memory. The bus bandwidth is restricted by simultaneous requests from the processors. The shared resource leads to a form of inflation, where the cost of performing an operation becomes increasingly expensive and therefore less efficient. This is the key limitation to multiprocessor architectures without local memories: Finite bus bandwidth means that only a fixed number of instructions can be carried out each second. Further, with current technology, growth in the bus bandwidth is relatively slow.

Recent advances in the research and development of Very Large Scale Integration (VLSI) technology have shown that in the near future it will be possible to construct one-chip cellular computers such as a systolic array system which contains hundreds or thousands of cells or processors[MIT]. By using such chips, highly parallel processing will come to an on-board computer for real time tasks. Such a computer will satisfy the requirements of light weight, small size, and powerful

computational capacity.

The VLSI technology provides a natural medium for parallel processing, from switching elements, to logic gates, to functional and control units, and to architectures and algorithms. To utilize its potential fully, parallelism must be applied at increasing high levels. Design methodology, consequently, must go beyond giving low level specifications, and should head towards systematically synthesizing efficient parallel algorithms and architectures. The technology, on the other hand, constrains the way in which parallel computation can be organized. Clearly, to achieve an efficient design, and to take full advantage of what the technology can offer in the present or in the near future, the parallel algorithm and the hardware must be considered simultaneously.

3.2 Systolic Arrays

Since 1978, when H.T. Kung and C.E. Leiserson[Kun-1,Mea] introduced the term "systolic array", much research has been done and much has been written about the design of algorithms and architectures suitable for such structures. Today, the idea of a systolic array is as familiar to many computer scientists and engineers as that of a compiler or a microprocessor.

Systolic architectures[Kun-2] are typically large, regular arrays of non-programmable or programmable processing elements, called PEs or

cells, as shown in Figure 3-1. Such an architecture is called systolic because data is pumped steadily through the array of cells , much like blood in the body. Data items, including inputs and partial and completed results, flow through the structure synchronously in a fixed, regular pattern. All operations involving an item are applied to it as it passes through. This method of computation eliminates the need to retrieve the item from external or global memory every time it is used, a process that typifies von Neumann machines. Consequently, a systolic array can be expanded to provide the increased capacity required by a computation-intensive application without imposing a corresponding increase in external memory bandwidth. This property gives systolic architectures a major advantage over traditional architectures, which are limited by the von Neumann bottleneck.

Besides their low external memory bandwidth requirements, systolic architectures offer other advantages as well. The simplicity of systolic VLSI designs is spatially important for special-purpose applications, where design cost must be amortized over a small production volume. A systolic array is typically made up of a few simple cell types and is therefore cheaper to design than a circuit containing a variety of complex cells. Moreover, the regular pattern of local interconnections between cells simplifies the layout problem.

Unlike architectures that distribute or broadcast data to many points(cells), a systolic architecture can easily be scaled up to handle large problems. Their local interconnection schemes avoid the clock skew that arises when data are broadcast over paths of differing

lengths. Also, the low "fan-in" and "fan-out" degrees of data in a systolic array allows the signal drivers to be independent of the number of cells in the arrays. (The "fan-in" degree of a datum is defined to be the number of data items on which it depends; the "fan-out" degree of a datum is defined to be the number of data items dependent upon it.) Thus, the size of a systolic array can be increased without altering other design parameters.

These properties--low external memory bandwidth, simplicity, regularity, and local communication--make systolic architectures especially well-suited to implementation in VLSI circuits. Systolic design is providing new or improved hardware solutions to many computation-intensive problems.

For the convenience of the parallel algorithm design in Section 5.4.5, the communication mode of a systolic array is described by a pattern matrix. In Figure 3-1-(3), for example, the interconnections between the cells are the six-nearest neighbors, or three-dimensional mesh interconnections. Each programmable cell contains an arithmetic logic unit (ALU), a local memory, several registers, and input/output (I/O) ports. The interconnection networks considered can be represented in a matrix form which is useful for the design of the mapping algorithms. For example, the cell (i,j,k), where (i,j,k) is called the index of the cell, is connected to the cell (i+1,j,k), then the index increases (1,0,0). The (1,0,0) is defined as a communication vector. (The communication vectors defined here refer only to the output ports of one cell.) Two other communication vectors are

obviously (0,1,0), corresponding to cell (i,j,k) connecting to cell (i,j+1,k), and (0,0,1), corresponding to cell (i,j,k) connecting to cell (i,j,k+1). All the communication vectors, including an inner vector (0,0,0) for internal computation, constitute a pattern matrix or the interconnection mode of the systolic array, that is

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This pattern matrix will be used in the design of parallel algorithm for trajectory optimization in Section 5.4.5.

3.3 SIMD and MIMD Computers

Among various parallel processing architectures, there are two primitive types of parallel computer systems besides the systolic array system. They are the Single-Instruction-Multiple-Dataflow (SIMD) type and the Multiple-Instruction-Multiple-Dataflow (MIMD) type[Mik].

A SIMD computer consists of a central processor unit (CPU) and multiple processors. The CPU play a role in system control, which requires broadcast in VLSI circuits. Each processor has a fixed number of registers and local memory, and is capable of performing arithmetic and boolean operations. The communication pattern or interconnections between processors are uniform. All the processors simultaneously execute the same instruction from the CPU or from the program stacks

in parallel on different data items.

A MIMD computer is different from a SIMD machine in that it can execute multiple instructions in parallel, store data in a common memory, and share peripheral equipment. On a MIMD computer, a problem is broken into a number of subproblems. Each subproblem is assigned to a processor with local memory. Each processor executes the necessary series of operations(instructions) on its data items.

Both SIMD and MIMD machines are array processor systems. There are differences between a systolic array system and an array processor system. The former is a pipelined array, while the latter has central control. Generally, an algorithm suitable for implementation on a systolic array has to have a required uniform structure whose typical characteristic is pipelining[Kun-3], while the requirements for an algorithm implemented on an array processor system are more flexible. (The main reason for this is due to the large size of the processing elements' local memory in the array processor system.) However, there are a few similarities between them which make the design method of a parallel algorithm applicable to both. For VLSI devices, interconnections between adjacent processors are required. This requirement is compatible with the architecture of an objective computer system. Also, both systolic arrays and array processors operate with their processing elements working synchronously. There is no bottleneck. The bandwidth rises linearly with the number of cells.

If the limitation on the memory size of cells in a systolic array

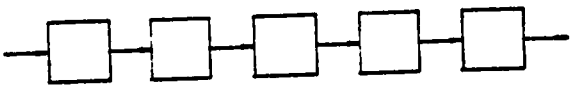
system forces data to be transmitted between the cells frequently, then the communication time increase significantly. In this case, an array processor system may be adopted because its local memory can be fairly large to store intermediate results so that the communication time is reduced. But for the same size of VLSI circuits, the number of the cells in an array processor system is less than that in a systolic array system, and the manufacturing cost of the former is higher than that of the latter.

It will be shown in Chapter 5 that the SIDP for trajectory optimization has significant reduction in fast memory requirement and therefore can be implemented in parallel. Due to this advantage, the size of local memory within each processing element is loosely constrained by the requirement of the fast memory of the original algorithm, and few connections and data transfer between the cells are necessary.

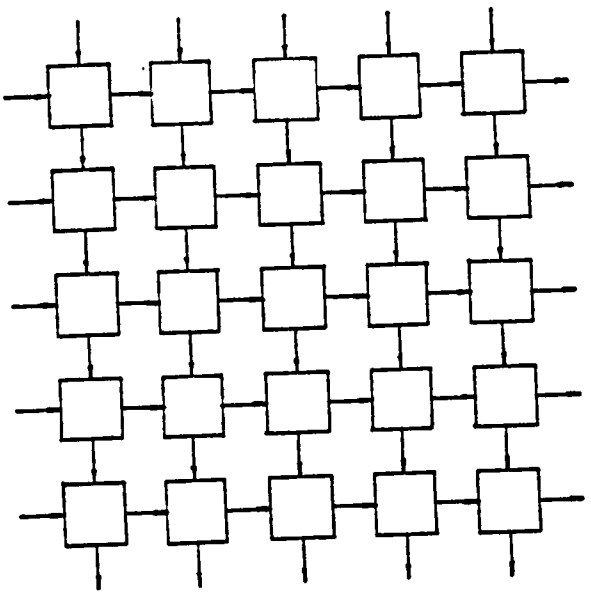
It will also be seen in Chapter 5 that in the parallel algorithm for trajectory optimization, the data dependence matrix D contains many zero elements. This means that data transmitted between the cells are very limited. Therefore, a systolic array system is proposed to be applied for the implementation for the parallel algorithm.

3.4 Summary

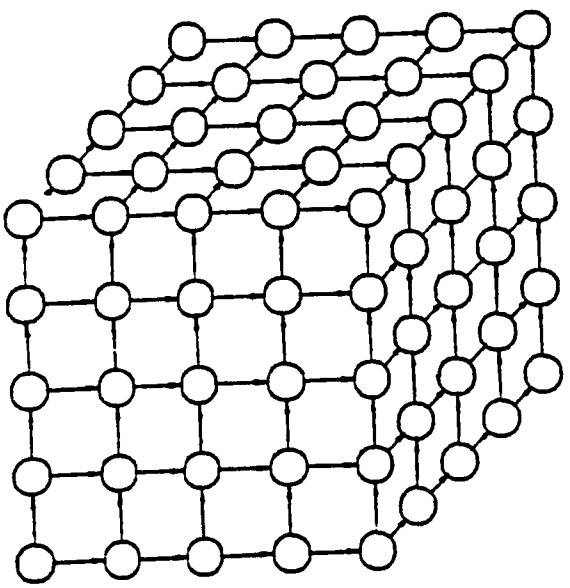
The continuously developing VLSI technology leads to a new design of parallel algorithms based on highly parallel architectures such as systolic arrays and array processors. These parallel architectures overcome the bottleneck problem, which exists in a "traditional" multiprocessor system. Their properties--low external memory bandwidth, simplicity, regularity, and local communication--make them especially well-suited to implementation in VLSI circuits. Such parallel architectures are providing new or improved hardware solutions to many computation-intensive problem, for example, dynamic programming.



(1) Linear Array



(2) Rectangular Array



(3) Cubic Array

Fig. 3-1 Systolic Array

4.0 PARALLEL TRAJECTORY COMPUTATION WITH A SIMPLE MODEL

4.1 Introduction

Parallel systolic algorithms for dynamic programming and their respective hardware implementations are presented for a problem in on-line trajectory optimization in this chapter. The method is applied to a simple model for aircraft flight path optimization through a complex constraint region. In order to overcome one inherent disadvantage of dynamic programming, or "curse of dimensionality"[Bel], and meet the real-time requirement, a parallel computational algorithm is studied. Parallelism is achieved by discretizing the computational state space into a large number of spatial domains. Since optimal segments of the trajectories can be computed in parallel, the locally optimal trajectory is obtained through subsequently combining the segments by a master controller.

One disadvantage of the dynamic programming approach is that the final flight path is only determined at the end of the computation at which time the entire family of optimal trajectories is determined. For flight purposes, a preferred approach is to obtain quickly a feasible flight path (i.e. a possible non-optimal trajectory that meets the boundary conditions of the problem), then use subsequent possible iterative methods to improve the feasible path. At any time, a flyable flight path is available, preventing any flight safety

problem. The approach taken in this chapter is to compute a local optimal trajectory within a band region based on an initial reference trajectory, and then use an iterative dynamic programming approach to obtain a local optimal trajectory in a restricted region of the state space. Once convergence to a local optimum is obtained, additional local optimal trajectories are subsequently determined to find the best global trajectory.

For the purpose of the above computational method, two systolic algorithms and their systems are designed: (1) the first systolic algorithm uses m multi-microprocessors plus two linear arrays, (2) the second systolic algorithm uses m multi-microprocessors plus a number of rectangular arrays.

4.2 Formulation

The model considered has two states and one control for aircraft that flies at low altitude through a region with complex cost penalties associated with vehicle position. For simplicity we consider constant speed motion in a horizontal plane. The single control variable is the vehicle heading. The state space is discretized into a set of Cartesian positions $(x_k, y_{k,l})$. The take-off and landing segments are not considered here.

The performance index is taken as a linear combination of distance traveled and a cost penalty associated with position in the form of

$$\min J = \sum_{k}^{N} [\theta C_k(x_k, y_{k,l}) + (1-\theta) dS_k(x_k, y_{k,l})] \qquad (4.1)$$

where x_k, $y_{k,l}$ are the discrete states, θ is a weighting coefficient, dS_k is the distance from the state at stage (k-1) to the state at stage (k) , C_k is the cost of the constraints associated with each point in the state space. C_k can be arbitrary specified but in this chapter is assumed to be a sum of a number of weighted Gaussian terms in the form of

$$C_k(x_k, y_{k,l}) = \sum_{u} A_u \exp \left\{ - \bar{X}_{k,u}^T R_u^{-1} \bar{X}_{k,u} \right\} \qquad (4.2)$$

where

$$\bar{X}_{k,u} = \begin{bmatrix} x_k - x_u \\ y_{k,l} - y_u \end{bmatrix} \qquad R_u = \begin{bmatrix} \sigma_x^2 & \rho\sigma_x\sigma_y \\ \rho\sigma_x\sigma_y & \sigma_x^2 \end{bmatrix}_u$$

where x_u and y_u are the coordinates of the constraint center, σ_x and σ_y are "variances"(giving the breadth of the region), ρ is the correlation coefficient (giving the skewness of the cost contour), and A_u is a weighting coefficient.

4.3 Parallel Algorithms and Architectures

To utilize the parallel computational approach an initial discretization of the state space is performed to break the whole region into a number of small band regions. The approach taken in this

section is to first align the local coordinate system x-axis with the nominal direction of travel. The spatial x-coordinate is broken into several sub-regions in a band region, so that families of optimal trajectories can be computed independently in each of these sub-regions using a parallel architecture. These segments of trajectories are then combined by a master computer to form an overall optimal flight path. To further speed up the solution process and to reduce memory requirement, the solution region can be reduced by considering a reduced computational region which includes the initial reference trajectory and a number of adjacent states. An iteration procedure is then used to extend this region to ensure optimality. Using this reduced space, the iteration approach for this algorithm is as follows.

First, a feasible initial reference trajectory is quickly determined using large steps and local minimization through the constraint region. For the simple example as shown in Figure 4-6, the total region is 100 by 100 miles. Using a large step of 10 miles, pick up the waypoints in the forward direction, and connect them to obtain a feasible initial reference trajectory. The above operations are completed by the host microcomputer. A band region is formed by extending Y(=10) miles beside the initial reference trajectory. This band region is divided into m sub-regions, each of which is assigned to one microprocessor. In each sub-region we discretize further with a number n_x in the flight direction, or X-axis, and with a number n_y along the Y-axis. The cost matrices of m sub-regions are calculated in parallel in m microprocessors (i.e. μp_1 ,μp_2, ..., μp_m). These

matrices are stored in the "local" memory when the control sub-bus of the microprocessors is effective, and form a larger cost matrix when the host microprocessor takes over the control bus. This is the function of parallelism #1 as shown in Figure 4-1.

Now the resulting trajectory is locally optimal within the reduced band region but may not be a global or even local optimal trajectory within the entire state space. The trajectory is locally optimal if in the constrained sub-region the optimal trajectory does not encounter the sub-region boundary. If the computed trajectory lies on a boundary segment, then the computational region is adjusted so that this becomes the interior region of a subsequent iteration. This iterative adjustment is continued until the trajectory lies within the sub-region boundary, at which point a local optimum is assured. To assure global optimality the entire state space must be covered by the optimality algorithm. This is accomplished by implementing the computation process systematically through different band regions of the entire space until all local minima are achieved. Of course if sufficient parallelism and memory are available to compute on the whole space, then iterative adjustment can be eliminated.

For now two architectures toward the parallel implementation of the dynamic programming algorithms are presented.

4.3.1 First Systolic Algorithm on Linear Arrays

For each of m sub-regions, there are n_x stages with n_y changeable states. For m sub-regions, there are a total of $N(=mn_x)$ stages also with n_y changeable states.

Since the final state is fixed, let the last stage cost vector be

$$\underline{r}^N = \underline{C}^N = \begin{bmatrix} C_1^N \\ C_2^N \\ \cdot \\ \cdot \\ \cdot \\ C_{ny}^N \end{bmatrix} \tag{4.3}$$

where C_i^N is the cost from the final state (destination) to state (i) of stage (N-1). Let the second last cost matrix be

$$C^{N-1} = \begin{bmatrix} C_{1,1}^{N-1}, & C_{1,2}^{N-1}, & \cdots\cdots\cdots\cdots, & C_{1,ny}^{N-1} \\ C_{2,1}^{N-1}, & C_{2,2}^{N-1}, & \cdots\cdots\cdots\cdots, & C_{2,ny}^{N-1} \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ \cdot & & & \cdot \\ C_{ny,1}^{N-1}, & & \cdots\cdots\cdots\cdots\cdots, & C_{ny,ny}^{N-1} \end{bmatrix} \tag{4.4}$$

where $C_{i,j}^{N-1}$ represents the cost from state (i) of stage (N-2) to state (j) of stage (N-1). Now the optimal cost at state (1) of stage (N-1) is given by

$$r_1^{N-1} = \min \{C_{1,1}^{N-1}+C_1^N, C_{1,2}^{N-1}+C_2^N, \cdots, C_{1,ny}^{N-1}+C_{ny}^N\} \tag{4.5}$$

To simplify notation, define the operator "⊙" such that

$$r_1^{N-1} = [C_{1,1}^{N-1}, C_{1,2}^{N-1}, \ldots\ldots, C_{1,ny}^{N-1}] \odot \underline{r}^N \tag{4.6}$$

to mean the minimization. Extending to vector operations we can also write

$$\underline{r}^{N-1} = \begin{bmatrix} r_1^{n-1} \\ r_2^{N-1} \\ \cdot \\ \cdot \\ \cdot \\ r_{ny}^{N-1} \end{bmatrix} = C^{N-1} \odot \underline{C}^N \tag{4.7}$$

Equation (4.7) can clearly be extended to the general stage (k) by

$$\underline{r}^{k-1} = C^{k-1} \odot \underline{r}^k \tag{4.8}$$

To make use of the parallel architecture, these backward dynamic programming operations are performed over half of the region from stage (N) to stage (N/2), where

$$\begin{aligned} \underline{r}^{N/2} &= C^{N/2} \odot \underline{r}^{N/2+1} \\ &= (C^{N/2} \ldots\ldots (C^{N-2} \odot (C^{N-1} \odot \underline{C}^N)) \cdots\cdots) \end{aligned} \tag{4.9}$$

Since the initial state is also fixed, a similar algorithm, starting at the initial stage and moving forward to stage (N/2-1), yields

$$\begin{aligned} \underline{r}^{N/2-1} &= C^{N/2-1} \odot \underline{r}^{N/2-2} \\ &= (C^{N/2-1} \cdots\cdots (C^3 \odot (C^2 \odot \underline{C}^1)) \cdots\cdots) \end{aligned} \tag{4.10}$$

Using the forward and backward vectors, the minimal cost for the entire multistage process is

$$\begin{aligned} J_{min} = \min\{ & r_1^{N/2-1}+r_1^{N/2}, r_1^{N/2-1}+r_2^{N/2}, \cdots, r_1^{N/2-1}+r_{ny}^{N/2}, \\ & r_2^{N/2-1}+r_1^{N/2}, \cdots\cdots, r_{ny}^{N/2-1}+r_{ny}^{N/2} \} \end{aligned} \tag{4.11}$$

At the same time, the local optimal trajectory within the band region can be obtained along with the minimal cost.

From the equation (4.9) and (4.10), it is seen that the solution of the original problem can be thought as two serial "matrix-by-vector" operations. Two linear arrays are designed to compute these operations in parallel at a very high speed. As shown in Figure 4-1, the blocks of parallelism #2 and #3 implement the above operations. The computation of the equation (4.11) can be done in the host microcomputer.

The description about systolic arrays is given in Section 3.2. Here the linear array as shown in Figure 4-2 is designed for the above parallel algorithm. At first, $\underline{C}^N$ is stored in the shift register #1. Each PE inputs r_i^{k+1} and $C_{i,j}^k$ in parallel and performs the operation indicated by equation (4.5). The results of the parameters of the states with minimal cost are first output to the shift register #1.

Then r_i^k is output to register #1, while the parameters are shifted to the shift register #2. The procedure in PE_i is as follow:

```
for k:=N-1 to N/2, step -1, do
    input C^k_{i,1} and r_1^{k+1}
    r_i^k := C^k_{i,1} + r_1^{k+1}
    P_i^k := 1
    for j:= 2 to n_y, step 1, do
        input C^k_{i,j} and r_j^{k+1}
        u := C^k_{i,j} + r_i^{k+1}
        r_i^k := min{r_i^k, u}
        if (r_i^k > u) P_i^k := j
    end
    output P_i^k
    output r_i^k
end
```

4.3.2 Second Systolic Algorithm on Rectangular Arrays

The most efficient representation of a given dynamic programming problem is usually problem dependent. If the number of states in a stage is large and constant, then the divide and conquer[Wah] method may be preferable, because more potential parallelism can be exploited.

The parallel algorithm by the divide and conquer method is derived

directly from equations (4.7) and (4.8). Let i,l,j be the parameters of states respectively at stages (k-1), (k), (k+1), then the minimal cost from state (i) of stage (k-1) to state (j) of stage (k+1) can be represented as

$$D_{i,j} = \min_{l} \{ C^{k}_{i,l} + C^{k+1}_{l,j} \} \tag{4.12}$$

where $i,l,j \in \{1,2, \cdots, n_y\}$. Similar to formula (4.7), formula (4.12) is defined as

$$D = C^{k} \odot C^{k+1} \tag{4.13}$$

where $k \in \{2,3,\cdots\cdots,N-1\}$. Therefore finding a local optimal trajectory is reduced to do a string of equivalent "matrix-by-matrix" operations. The fastest way to multiply N n-by-n matrices is to locate the matrices in the leaves of a complete binary tree of height $\log_2 N$. The N-stage problem can be solved in $O[n(\log_2 N)]$ time units with N/2 matrix-multiplication arrays[Li-1]. In this way, the measures of computational time and local memory in each PE are lower, but many PEs are needed if N is quite large. For the problem studied, $n_y/2$ rectangular arrays are applied to m sub-regions, where the PEs are a little different from those in the linear arrays.

Figure 4-3 shows the block structure of the second systolic algorithm. $R_i (i=1,2,\cdots\cdots,n_y/2)$ represents a rectangular systolic array, which is designed for "matrix-matrix-multiplication" as shown in Figure 4-4. The given rectangular systolic array is a

mesh-connected computer consisting of n_y^2 PEs arranged in a two-dimensional array with interconnections between every pair of horizontally and vertically adjacent processors.

4.4 Simulation

The simulation of the algorithm as stated above has been done on a single microcomputer for a simple model trajectory. The vehicle is assumed to fly at a constant speed and a constant altitude in a 2-dimensional plane. For this problem the desired ending point is 100 miles in the X-direction from an initial state (starting point). The starting and ending y coordinates are assumed to be zero. The space is quantized into 10 sub-regions in the X-direction each of 10 miles,and 20-mile wide segments in the Y-direction. Various discretization levels in the Y-direction are used.

As a simple example, a set of constraints is used as shown in Figure 4-5. Because of the large 'hills' and 'valleys' throughout the state space it is clear that there will be numerous local minimizing curves through the region. (In fact, there will generally be a 'local' minimum through each valley and it is because of this, that the classical variational approach may not be successful in finding the global minimum.) The dynamic programming approach can find the global minimum and all local minima if the entire state space is used in the solution. For on-line computation, however, it may be important to raise the speed required to find a solution, even if that solution may

not be globally optimal. In the approach used here, we choose to first find a set of feasible initial reference trajectories by a very coarse optimization over the state space. We then iterate on these feasible trajectories by using a reduced state space as shown in Figure 4-6 and discretized so as to implement the parallel algorithm. Figure 4-7 shows the four typical initial reference trajectories. A set of iterated trajectories for one initial path is shown in Figure 4-8, and the set of local optimal trajectories corresponding to the starting solutions is shown in Figure 4-9, and in Figure 4-10 if the final state variable y is free. When a possible local optimal path is blocked by a constraint such that an obstacle or a threat, the trajectory go around as shown in Figure 4-11.

The advantage of this approach is that a feasible trajectory can be found more quickly than using the entire state space as the solution region. A disadvantage is that a workable scheme must be implemented to ensure that all local trajectories are searched to find the global minimum. While in a practical situation global optimality may not be required for success, there should be assurance that this global path is ultimately achievable. In all cases iteration could be avoided if the y size of a sub-region was increased. Of course memory and computation time would then become drastically greater.

4.5 Summary

In order to implement on-line trajectory optimization by the dynamic programming method, two systolic algorithms and their corresponding architectures have been designed. The first systolic algorithm is formulated as a string of equivalent "matrix-by-vector" operations. The second systolic algorithm is formulated as a string of equivalent "matrix-by-matrix" operations. The latter has a higher parallelism than the former, which means the execution time in the second architecture is less than that in the first architecture but at the cost of more complex hardware structure.

Both hardware systems of figure 4-1 and figure 4-3 adopt a master-slave organization and consist of a host microprocessor, m slave microprocessors, and several systolic arrays with several levels of buses. The host microprocessor is responsible for high-level task execution and overall timing control. The m microprocessors are used for parallel computation of cost matrices. The systolic arrays are applied as the peripheral devices of the host microprocessor for high-speed computation of the parallel dynamic programming algorithms.

Both the linear systolic arrays and the rectangular systolic arrays applied in the systems have a bounded I/O requirement[Kun-2]. Synchronization between PEs can be achieved by a simple global clock whose rate is independent of the size of the array[Fis]. But the rectangular arrays have many more PEs than the linear arrays. In general, the Very Large Scale Integration (VLSI) implementation of the

rectangular array is more expensive than that of the linear array. Which algorithm is preferable depends on the problem being studied. The first systolic algorithm can be applied with the linear arrays as long as the real-time requirement of the problem is satisfied. If the problem requirements demand an ability to operate at higher speeds, then the second systolic algorithm may be used with the rectangular arrays.

The simulation of the parallel algorithm designed by the direct method has been completed. The numerical results have shown that the algorithm applied to the determination of a feasible trajectory and of the iterative solution of a local optimal trajectory is practical. At this point, the estimate of the timing requirements in a parallel machine has not been made, but it is felt that this approach has effectively made an on-line trajectory optimization algorithm feasible.

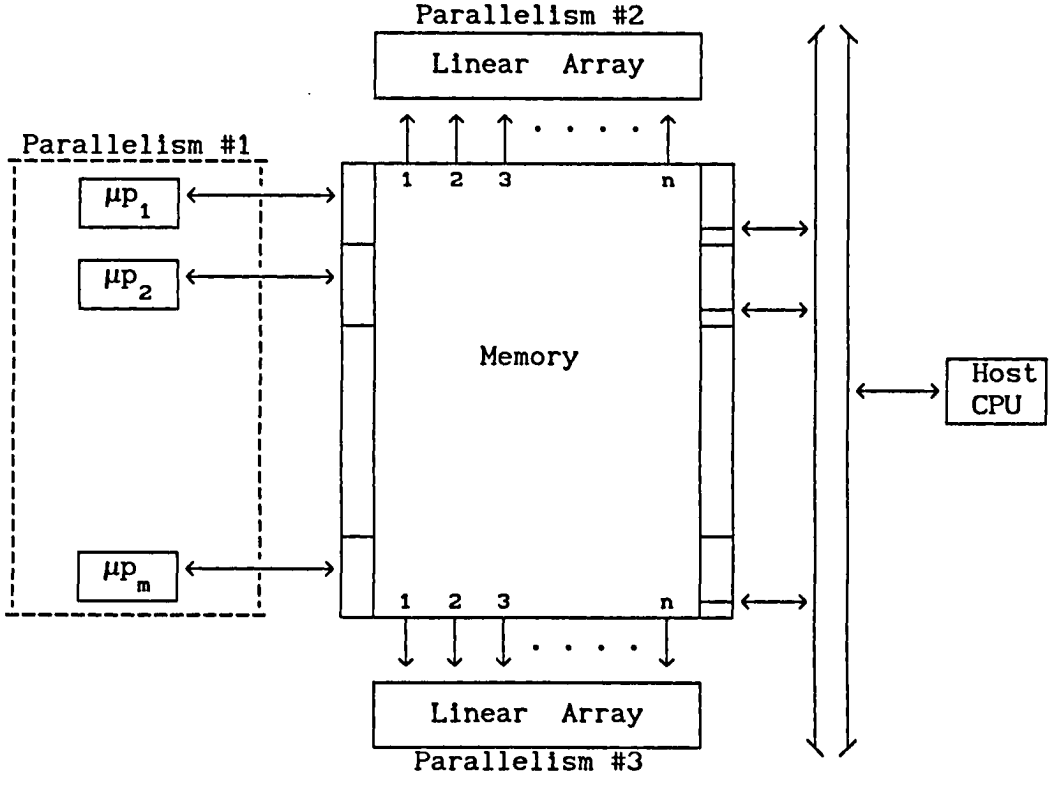


Fig. 4-1 Block Structure of First Algorithm

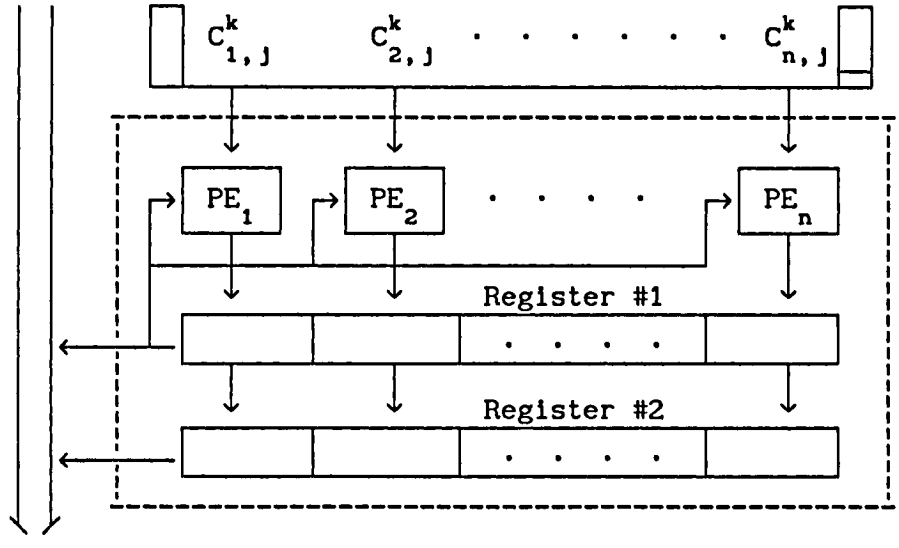


Fig. 4-2 Architecture of Linear Array

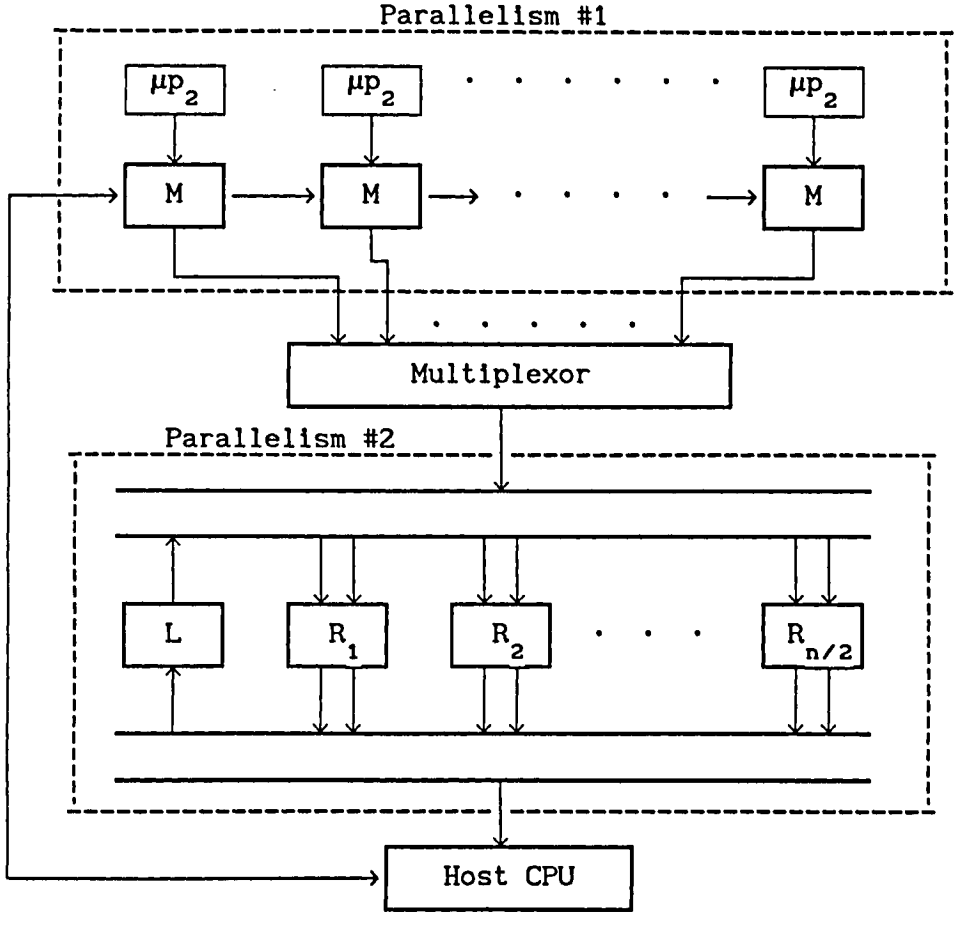


Fig. 4-3 Block Structure of Second Algorithm

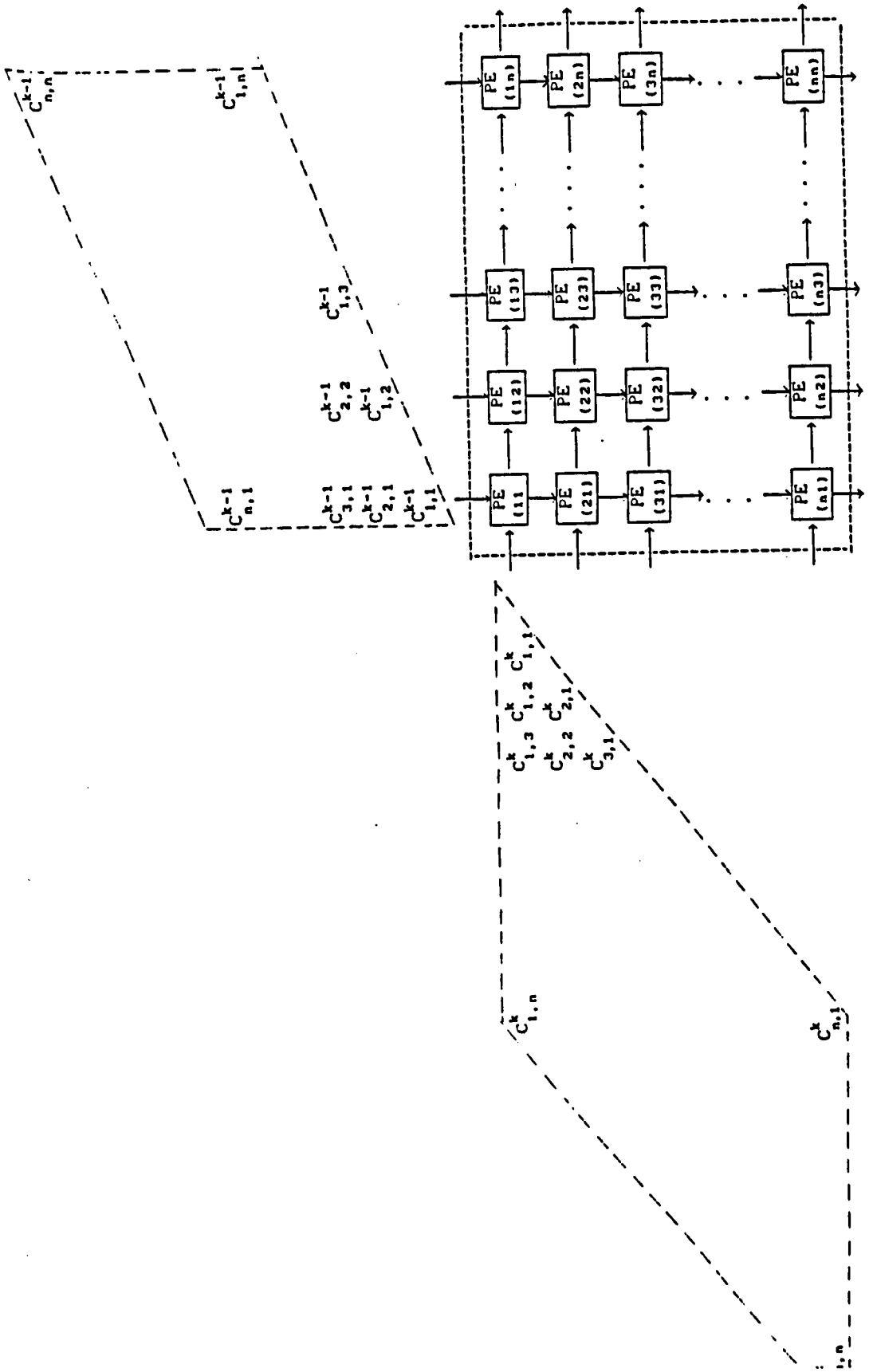
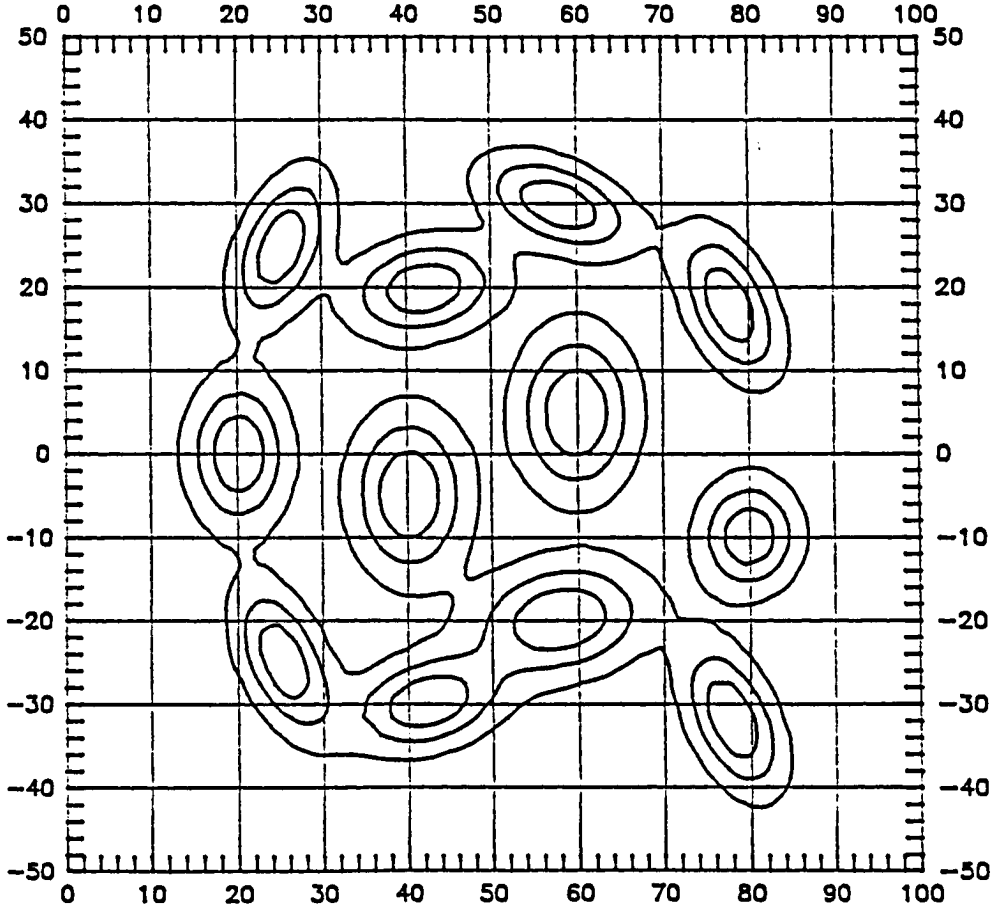


Fig. 4-4 "Matrix-by-Matrix" on Rectangular Array

Fig. 4-5 Constraint Distribution



Flight Direction: mile

Fig. 4-6 Initial Calculation Space

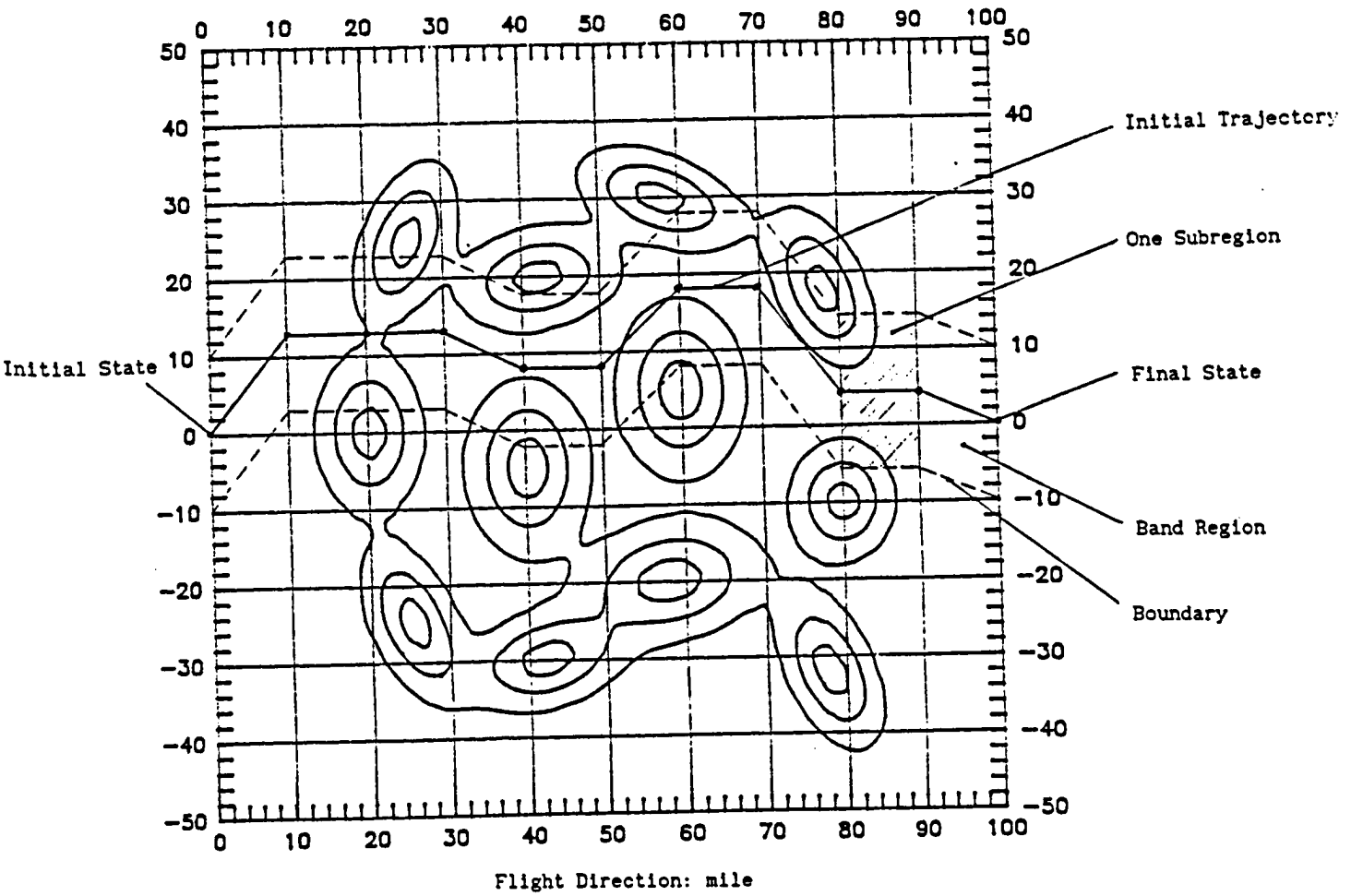


Fig. 4-7 Initial Reference Trajectories

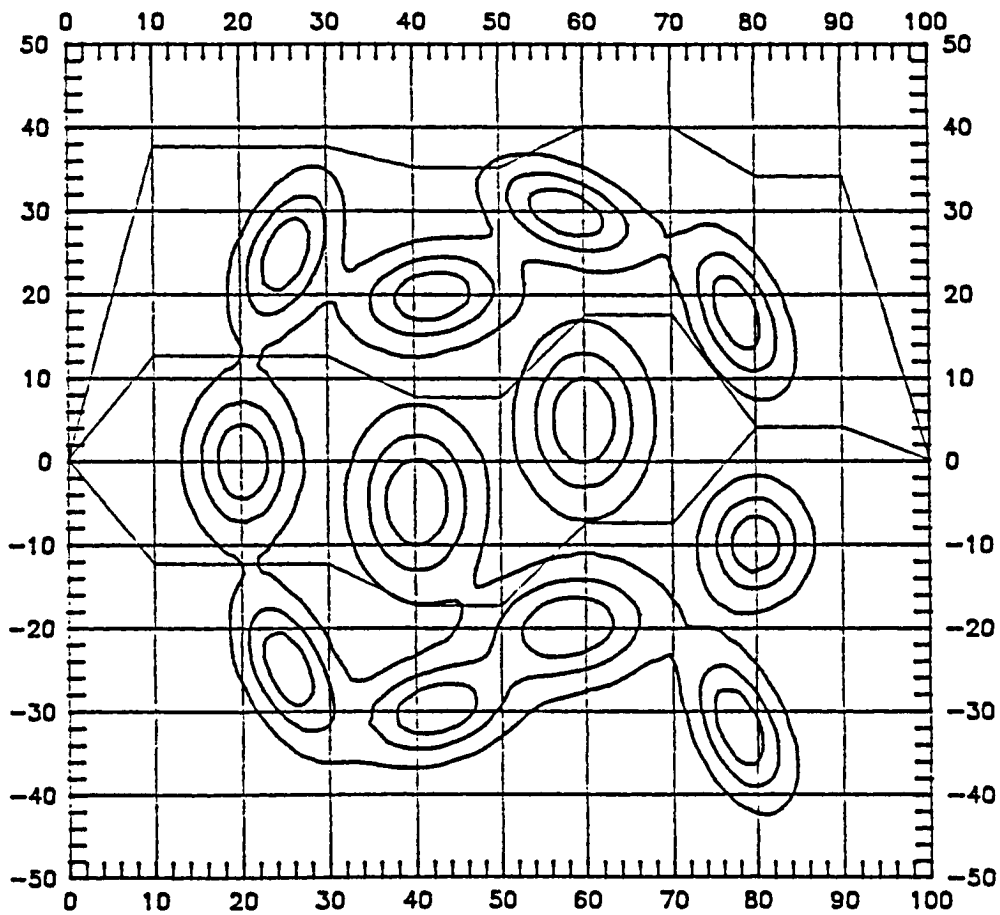


Fig. 4-8 Iterative Local Optimal Trajectories

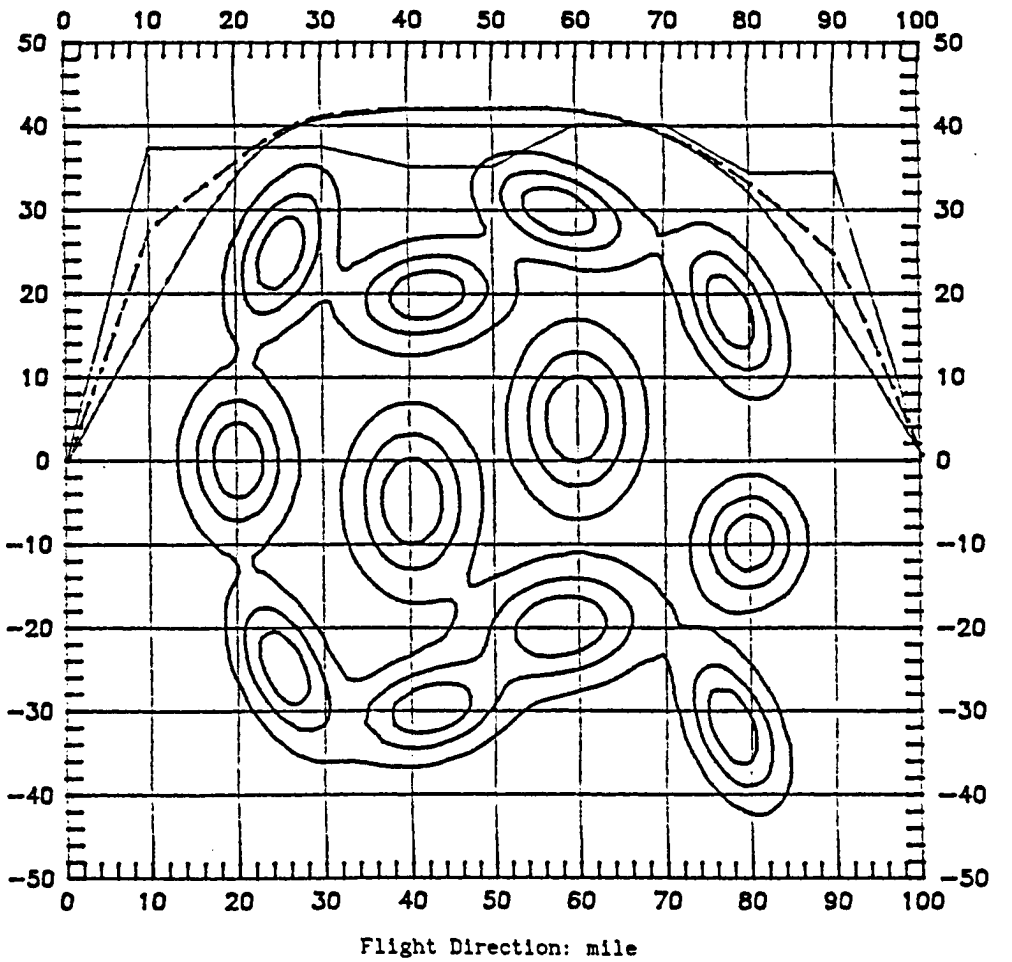


Fig. 4-9 Local Optimal Trajectories

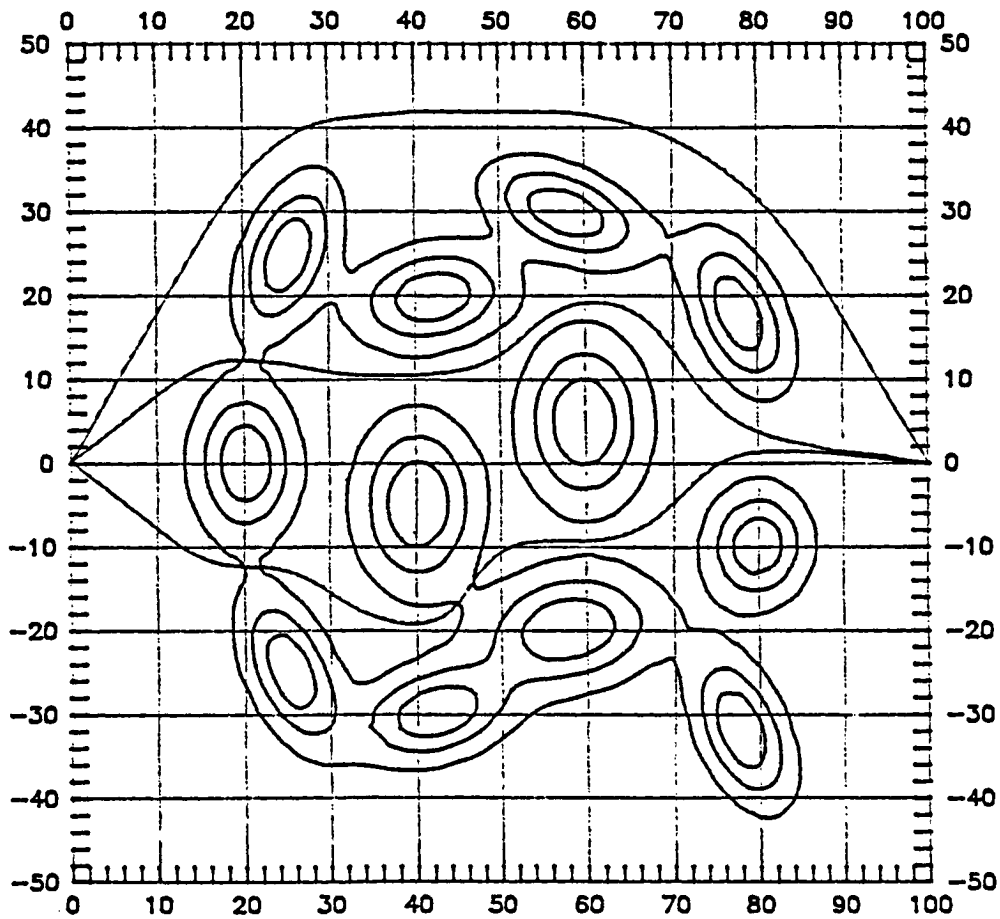


Fig. 4-10 Local Optimal Trajectories with Free Final State Variables

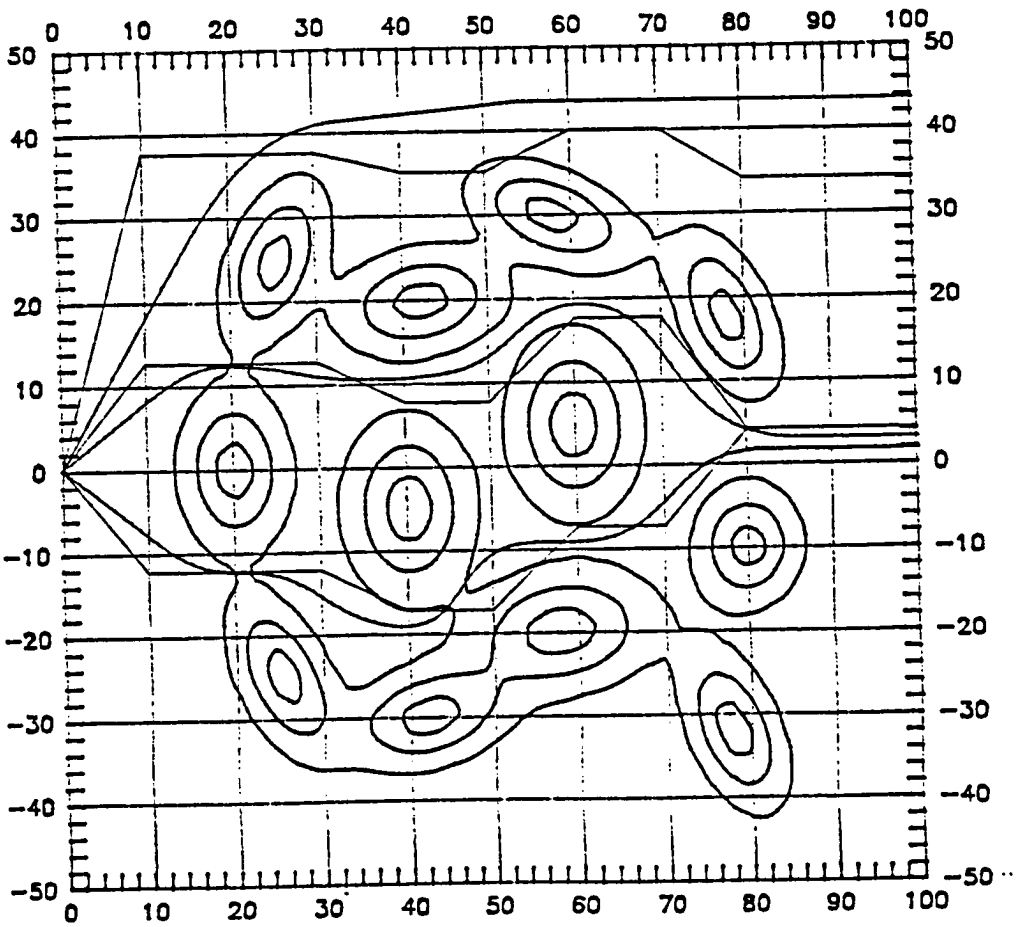
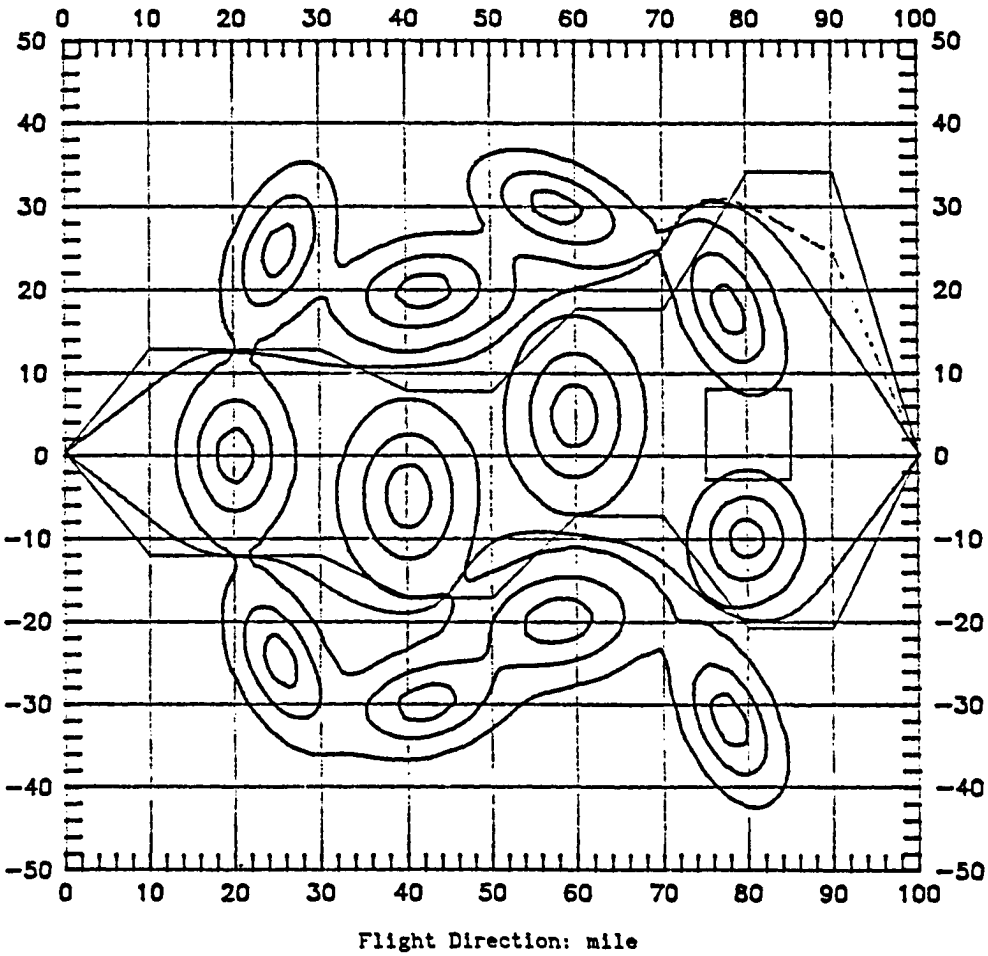


Fig. 4-11 Local Optimal Trajectories with Constraint Avoidance



5.0 PARALLEL TRAJECTORY OPTIMIZATION WITH A COMPLEX MODEL

5.1 Introduction

Using the model with two states and one control in Chapter 4, an effective parallelization technique is to decompose the computational region into m sub-regions. Trajectories can be computed in parallel in m sub-regions and the optimal trajectory can be determined by combining those trajectories within m sub-regions. Due to the low dimensionality of the computational state space, the number of the states on the transferring interface between two sub-regions is relative small since only one state variable discretization required. This means that the number of states on the interface may typically be in the hundreds, and therefore hundreds of trajectories may exist in a sub-region. It is then possible to combine these trajectories to obtain a local optimal trajectory by the algorithms executing on the linear or rectangular arrays. For this size problem, the direct method can be applied which maps the algorithm ("matrix-by-vector" operations or "matrix-by-matrix" operations) onto the systolic arrays (linear arrays or rectangular arrays) because the dimensionality of the cost matrix is low enough so as to fit the size of systolic arrays designable in the Very Large Scale Integration (VLSI) technology. If we consider the more complex model with six states and three controls as given in Section 2.2, the situation is quite different. Suppose that each state be discretized into only ten values, then the number

of the discretized value of the states on the interface to be combined would be 10^5. Accordingly, if the design method of a parallel algorithm similar to that in Chapter 4 is used, the dimensionality of the cost matrix is too high to fit the size of systolic arrays so that further decomposition of the operations become necessary and additional computational time for the decomposition and composition occurs. As a result, the benefit of parallel computation in m sub-regions is severely penalized by the vast amounts of combinational computation of m sub-regions and the additional computational time for the decomposition and composition. Therefore, in this chapter we propose the use of the SIDP algorithm for a more complex model to make use of the advantages of the SIDP, namely, a small fast memory requirement and also a small data dependence domain (see Section 5.4.4). For an application of the SIDP to a full six-dimensional model, however, it is difficult or impossible to find a direct method which will map it onto a parallel architecture such as systolic arrays. A systematic approach is needed to design the parallel algorithm for the serial SIDP algorithm.

The first and probably most challenging problem in the design of a parallel algorithm is in the transformation of a high-level serial algorithm into a parallel algorithm suitable for the VLSI implementation. Such a high-level serial algorithm can be, for example, a system of first-order recurrence equations, a uniform iterative equation, or a nested loop with constant data dependence. Usually, the transformation for these problems is obtained by using techniques similar to those in software compilers: buffering of

variables and addition of new variables. However, it is not well understood how to choose a good transformation particularly for some complex algorithms.

In this chapter, in order to make it feasible to calculate on-line an optimal trajectory by the SIDP for a velocity-variant, three-dimensional model in the TF/TA environment, the entire computational state space in the flight region defined by a mission was broken into several reduced space-band regions. Within a space-band region, the parallel algorithm of the SIDP method is applied to compute a local optimal trajectory using a parallel architecture or a systolic array system.

5.2 INITIAL REFERENCE TRAJECTORIES

An initial reference trajectory is used to form a reduced band area of the state space which is the ultimate computational region for the parallel SIDP scheme. Such an initial reference trajectory may be determined, e.g., in mission planning system. The initial reference trajectory planning can be defined as the process of determining quickly a segment-directional route from a starting location to a destination while accounting for such constraints as mission destination, minimum exposure to threats, and time and terrain restrictions. Unlike searching waypoints forward in the case of Chapter 4 so as to generate a forward initial reference trajectory, a different method for searching waypoints is presented in this chapter

to generate an initial reference trajectory which may involve turn-back segments while considering a more realistic terrain environment. For example, the simulated terrain profile, which we made as shown in Figure 5-1, is used in the analysis of the following section.

Similar to the method in Chapter 4, the entire computational state space in the flight region defined by a mission is broken into a number of small space-band regions to utilize the parallel computational approach. These space-band region are based on the initial reference trajectories. The families of possible local optimal trajectories can be computed independently in each of these space-band regions using a parallel computer system. For a given hardware architecture to speed up the solution process and to reduce memory requirement, the size of the space-band region is reduced and an iteration procedure is used to extend this region to ensure optimality.

5.2.1 Determination of Waypoints

The entire flight region is considered to be a bounded region in the state space containing the starting position and destination of a mission and all "feasible" points. Such a region is represented by S, a set of the state space, where the starting position is located at the position that is close to the left edge of S and the destination is located at the position that is close to the right edge of S as

shown in Figure 5-2. A local coordinate system is aligned whose origin is located around the starting location, the X-axis is along the nominal direction of travel, the Y-axis is orthogonal to the X-axis , and the H-axis is vertical. The distance between the starting position (or destination) and the left (or right) edge depends on terrain profile in the flight region. The altitude range of the points in set S depends on the expected altitude range for the flight paths.

Even with massive parallelism, it is probably not practical at this time to do on-line computation of a global optimal trajectory by the dynamic programming method in the whole flight region S because of the limitations of computer speed and fast memory size. The idea to make real-time computation feasible is first to break S into a number of subsets, and secondly to design a parallel algorithm of dynamic programming based on a parallel architecture such as systolic arrays, and then do computations in each subset in parallel using a parallel computer system.

The reduction in the computational state space will be accomplished by either naturally or artificially imposing waypoint constraints, i.e., points that the optimal trajectory should pass through or near. The automatic determination of such a reference path is necessary to allow autonomous solution of the optimal trajectory problem.

A possible algorithm which will create such a set of waypoints is described as follows. Beginning with the starting point, we take a

waypoint as a center (the first one is the starting point), and then draw a search arc with a search radius r and an angle β as shown in Figure 5-2. (The search radius r and the angle β are based on the general knowledge of the studied problem, for example, the knowledge of DTED.) It can be seen from Figure 5-2 that several valley points along the search arc can be found. These points V_i are examined and their corresponding costs are calculated by an index which is a linear combination of the distance to the destination, altitude, and exposure to threats. These V_i are rearranged in order of their increased costs. The first point V_1 is selected as a candidate for a waypoint W_i at the current search step. The remaining valley points are stored for later use. Now we go to the next search step. Connect W_i to W_{i+1} and take W_{i+1} as a center, and repeat the above operations until we reach a small domain containing the destination point. The curve connecting the starting point, waypoints, and the destination is formed. This curve is named as an initial reference trajectory. If during the initially selecting a candidate a waypoint W_i fails, we go back to the previous search step and select the second point V_2, a new candidate as a waypoint W_i. If an initial reference trajectory is found successfully, we also go back to find all the possible initial reference trajectories. At last, a number of the initial reference trajectories are found as shown in Figure 5-3.

5.2.2 Space-Band Regions--Iteration

For the initial reference trajectory described in Section 5.2.1, a

space-band region can be constructed which contains this path. We expand this initial reference trajectory in the X-Y plan a certain distance D_{xy} normal to the initial reference trajectory and vertically to an altitude H above the corresponding ground height H_g. This space-band region is defined as a subset $\mathbb{X}x\mathbb{Y}x\mathbb{H}$, where $\mathbb{X}x\mathbb{Y}x\mathbb{H}\in\mathbb{S}$. The H_g is thought of as the function of x and y, or $H_g(x,y)$ which is assumed known.

Similar to the situation in Chapter 4, an iteration procedure is applied to obtain a local optimal trajectory. This iteration procedure is restated as follows. Within the reduced space-band region, an optimal trajectory is computed, but it may not be a global or even local optimal trajectory within the entire state space in the specified flight region. The trajectory is locally optimal if in the constrained space-band region the optimal trajectory does not encounter the boundary of the space-band region. If the computed trajectory lies somewhere on the boundary of the space-band region, then the computational space is adjusted so that this becomes the interior region of a subsequent iteration. This iterative adjustment is continued until the trajectory lies within the the boundary of the space-band region, at which point a local optimum is assured. To assure global optimality, the entire state space must be covered by the optimality algorithm. This is accomplished by implementing the computation process systematically through different space-band regions of the entire state space until all local minima are achieved.

5.3 Dynamic Programming Method

Dynamic programming, a powerful optimization methodology, has been widely applied to large classes of optimization problems such as linear and nonlinear programming, pattern recognition, and artificial intelligence[Lar]. In this study, dynamic programming is used to solve the problem of trajectory optimization for aircraft.

Bellman has characterized dynamic programming through the Principle of Optimality, which states that an optimal sequence of decisions has the property that whatever the initial states and decisions are, the remaining decisions must constitute an optimal decision sequence with regarding to the state resulting from the first decisions[Bel]. Subsequently, numerous efforts have been devoted to the rigorous mathematical framework and effective evaluation of the dynamic programming problems[Kar,Iba].

In general, a dynamic programming problem can be formulated by a recursive equation whose left-hand side identifies a function name and whose right-hand side is an expression including the minimum (maximum) operation of some function. Denote optimal cost at the k^{th} stage as

$$I(k) = \min_{u} \sum_{i=k}^{N} L_i[X(i),U(i),i] \qquad (5.1)$$

Then the backward recurrence equation for dynamic programming is

$$I(k) = \min_{u} \{ L_k[X(k),U(k),k] + I(k+1) \} \quad \text{for } k=N-1,\cdots,1 \tag{5.2}$$

Although dynamic programming has been recognized as a powerful approach to solving a wide spectrum of optimization problems, its application is restricted because computational requirements increase dramatically with the size of the state set and control set of a particular problem, which makes dynamic programming unsuitable especially in the real-time case. This is the well-known *curse of dimensionality* described first by Bellman[Bel].

With the advance of the (VLSI) technology, it is feasible to design parallel algorithms of dynamic programming based on parallel architectures of computer hardware, such as systolic arrays[Kun-2], Single Instruction Multiple Dataflow (SIMD), and Multiple Instruction Multiple Dataflow (MIMD) machines[Mik] to overcome *the curse of dimensionality*.

The goal for designing parallel algorithms of dynamic programming is always to reduce the computational time and memory requirement of a serial algorithm by a factor which is a function of the number of available processors.

5.3.1 Review of Parallel Computation of Dynamic Programming

The dynamic programming approach is actually a transformation of

the problem into a more suitable form for computation. The functional equation (5.2) can be interpreted and rewritten into different forms depending on the computational requirement and size of the problem, and the available resources of solving the problem. Many authors, such as Bellman, White, and Larson, viewed dynamic programming as a multistage optimization technique, that is, reducing a single N-dimensional problem to a sequence of N one-dimensional problems[Bel,Bou,Whi]. The decisions that transform an initial state into a final state must be ordered in terms of stages and functional equations which relate to state values in the successive stages. The use of monotone sequential processes has been proven by Karp and Held to correspond naturally to dynamic programming[Kar] and has been further developed by Ibaraki[Iba]. From the viewpoint of multistage decision processes, on the other hand, dynamic programming problems can also be solved through the search of an optimal path on a multistage graph[Lar].

Assuming that T is the computational time of the best serial algorithm of dynamic programming for a given problem, the theoretical computational time of a parallel algorithm for this problem on a parallel machine with P processors is O(T/P), where P is called the optimal speedup ratio. Generally, however, the optimal speedup cannot be reached because of overhead for carrying out the interprocessor communication[Lin]. The most often used criterion for a parallel algorithm is PT^2 or AT^2 in the VLSI terminology, where P is the number of processors, A is the area of the VLSI implementation or memory requirement, and T is the computational time. The efficient design of

a parallel algorithm means that the value of PT^2 or AT^2 required by the parallel algorithm is relatively low.

"Matrix-by-Vector" Method

Based on the fact that a dynamic programming problem can be represented as a multistage graph, therefore, an optimal trajectory (or an optimal decision series) can be obtained by graph search. The reason that the graph search is adopted as a paradigm is to make it convenient to illustrate various possible parallel approaches of dynamic programming, and graphs have certain regular structures so that a regular, limited interconnected serial algorithm or multiprocessor network can be designed to implement parallel algorithms.

For general graph G=(V,E) as shown in Figure 5-4, in which V is the set of vertices and E is the set of edges, let C(i,j) be the minimum cost of edges $e_{i,j}$ from vertex i to vertex j, where $i,j \in V$ and $e_{i,j} \in E$. The cost of a path from source s, without incoming edge, to sink t, without outgoing edge, is the sum of costs on the edges of a path. To find C(i,t), paths through all possible vertices must be compared. Then,

$$C(i,t) = \min_j \{ C(i,j) + C(j,t) \} \qquad \text{for } 0<i<j<N \tag{5.3}$$

Equation (5.3) is termed *backward functional equation*. Similarly starting from the source s

$$C(s,i) = \min_{j} \{ C(s,j) + C(j,i) \} \qquad (5.4)$$

$$\text{for } 0<j<i<N$$

Equation (5.4) is termed a forward functional equation. Note that the minimum cost functions only involve one recursive term in both equations.

Based on the above analysis and formulations by means of graph search, the optimal cost C(s,t) can be obtained through a string of equivalent "matrix-by-vector" operations. For example, the parallel algorithm designed in Chapter 4 , which was applied to on-line trajectory optimization for aircraft, was formulated as two string equivalent "matrix-by-vector" operations executed on the systolic arrays or two linear arrays. The speedup of the algorithm is naturally obvious through the parallel operations on linear arrays with shift registers and global shared memory.

The pipeline architecture and the corresponding parallel algorithm was presented to solve the knapsack problem[Che] (see Appendix A). This problem was mathematically treated as a linear programing problem, and reformulated as a dynamic programming problem similar to equation (5.4). The speedup of the algorithm , which actually involves the "matrix-by-vector" operations, is achieved by linear arrays, queues(a control logic unit), and memory modules. The processor array

is regarded as a pipeline where the parallel algorithm of dynamic programming is implemented through pipelining.

Divide and Conquer Method

A common approach to solving complex problems is to partition the original problem into smallest parts, find solutions to the parts, and then combine the solutions into a solution to the whole. This approach, referred to as a "divide and conquer" strategy applied recursively, is the basis for divide and conquer algorithms. The divide and conquer method is a well-known algorithm that can solve some dynamic programming problems and can be represented as a tree in graph theory. Therefore, it is concluded that a dynamic programming problem can be transformed into the search problem of finding the optimal path on the tree. An important issue in parallel divide and conquer algorithms is to determine the granularity of parallelism, or the minimum size of subproblems, that a processor evaluates in order to achieve optimal performance. If the granularity is large, then the processors can be loosely interconnected; otherwise, tight interconnections among the processors may be necessary as in systolic arrays.

Examining the graph search again in Figure 5-4 and equations (5.2), the optimal path with the minimum cost C(i,j) from vertex i to vertex j can be represented as

$$C(i,j) = \min_k \{ C(i,k) + C(k,j) + D(i,k,j) \}$$
$$\text{for } 0<i<k<j<N,\ j-i\geq 2,\ i<k<j \qquad (5.5)$$
$$C(i,i+1) = Ci \qquad \text{for } i=0,1,\cdots\cdots,N-1$$

where D(i,k,j) is the additional computation cost due to the decomposition of a serial algorithm and Ci is the initial cost at the i^{th} stage in computing. Obviously, equation (5.5) involves two recursive terms. The dynamic programming algorithm in the form of equation (5.5) can be implemented in parallel.

Equation (5.5) can be interpreted as a string of equivalent "matrix-by-matrix" operations[Sla-3]. If the corresponding graph is regular or the states at each stage are equal to each other, the fastest execution of N "matrix-by-matrix" operations is to locate the matrices on the leaves of a complete binary tree with height ($\log_2 N$). The optimal decision of a N-stage graph problem is equivalent to an optimal binary tree search with a minimum cost. The search can be finished in $O(m\cdot\log_2 N)$ time units with N/2 "matrix-multiplication" systolic arrays[Li-1] for the VLSI implementation, where m is the dimension of the cost matrices.

The general method of parallelization of the optimal binary tree search transformed from "matrix-by-matrix" operations, described by Valiant et al[Val], leads directly to the algorithm working on $O(N^9)$ processors in $O(\log_2^2 N)$ time units. Rytter[Ryt] used a pebble game on the subtrees of a complete binary tree so as to decrease the number of processors and simplified the algorithm on the subtrees. It was shown

that the obtained parallel algorithm for a serial dynamic programming problem or an optimal binary tree search problem can be computed in $O(\log_2^2 N)$ time units with $O(N^6/\log_2 N)$ processors on a parallel random-access machine with concurrent reading and exclusive writing (P-RAM-CREW). If a parallel machine with concurrent reading and concurrent writing (P-RAM-CRCW) is available, then the parallel computational time can be reduced from $O(\log_2^2 N)$ to $O(\log_2 N)$. Huang and Liu[Hua] gave a modified algorithm of Rytter's that works in $O(N^{1/2}\log_2 N)$ time units with $O(N^5/\log_2 N)$ processors.

Varman[Var] assumed D(i,k,j) is independent of k, or

$$C(i,j) = \min_k \{ C(i,k) + C(k,j) \} + D(i,j) \qquad (5.6)$$

and presented the parallel algorithm that was designed to be implemented on linear arrays by pipeline processing. The algorithm requires O(N) processors and $O(N^2)$ storage in $O(N^2)$ time units.

The parallel divide and conquer algorithm, involving "matrix-by-matrix" operations which are executed on rectangular arrays, was presented in Chapter 4 to find the optimal trajectory for aircraft. This algorithm needs less execution time than the algorithm characterized by the "matrix-by-vector" operations but requires a more complicated hardware architecture, or rectangular arrays.

Branch and Bound Method

Compared to the Divide and Conquer approach, branch and bound algorithms are alternative effective methods for solving many optimization problems of high complexity such as dynamic programming. The branch and bound approach can be viewed as a technique for the search of solutions in combinational large problems. As these algorithms realize an implicit enumeration of the space of solutions, their computational requirements (time and space) grow exponentially in the problem size. Excess storage and run time requirements can often make the algorithms impractical in conventional serial implementation. Therefore, the idea to parallelize branch and bound algorithms has naturally emerged. In the literature, up to now, different versions of parallel branch and bound algorithms have been proposed, which are implemented on either parallel multiprocessor machines or VLSI systolic arrays. The underlying idea of the branch and bound algorithm is to decompose a given problem into two or more subproblems of smaller size. Then, this partitioning process is repeatedly applied to the generated subproblems until each subproblem is solved or shown not to yield an optimal solution of the original problem. The process of excluding a subproblem from further consideration is based upon the computation of a lower bound on the values of solutions within each subset: subproblems whose bounds exceed the value of some known solution can be discarded[Lai-1].

The "Relaxations and fathoming criteria"[Geo] can be used to identify and to eliminate states whose corresponding decision cannot

lead to an optimal path. This approach has been applied to the traveling salesman problem and the nonlinear knapsack problem[Mor]. Imai, et al[Ima], and El-Dessouki and Huen[Des] investigated parallel branch and bound algorithms based on general-purpose network architecture with limited memory space and slow interprocessor communication. They used depth-first search due to memory limitations. Under the assumptions that only lower bound tests are active, a single shared memory is available, and no approximation is allowed, subproblems are expected to be executed synchronously[Wah].

The speedup of parallel branch and bound algorithms have been studied by Lai and Sahni[Lai-2], Lai and Sprague[Lai-1], and Roucairol[Rou-1]. The applications of parallel branch and bound algorithms to the traveling salesman problem can be found in Reference [Rou-2], to the quadratic assignment problem in Reference [Rou-3], and to the multi knapsack problem in Reference [Pla]. Some experimental results were presented by Rodgers and Pardalos[Rod] for the quadratic 0-1 problem (see Appendix B) on various parallel architectures of computers: the speedup ratio is 3.80 on CRAY-XMP/48 with four processors, 5.22 on IBM 3090 600E with 6 processors, 12.20 on intel iPSC/2 with 16 processors.

AND/OR Graph Method

For searching an optimal path in a multistage graph as shown in Figure 5-4, we suppose that this (N+1)-stage graph with m vertices in

each stage be divided into p subgraphs by $p^q=N$, where q is non-negative integer. Each of subgraphs contains N/p+1 consecutive stages. The optimal path with the minimum cost has to pass through one and only one vertex in stage 0, N/p, $\cdots$, pN/p in the segmented graph. The cost of a path is equal to the sum of costs of the p subgraphs. If all the m^{p+1} subpaths from the m vertices in stage iN/p to the m vertices in stage (i+1)N/p ($0 \le i \le p-1$) have been optimized, there are m^{p+1} possible combinations of subpaths from stage 0 to stage N that must be considered for the optimal path. By using the divide and conquer method, each subgraph with N/p+1 stages is further divided into p smaller subgraphs. This partitioning process continues until each subgraph has one stage. Such a partition process can be conveniently represented as an AND/OR graph, in which an AND-node corresponds to a subproblem sum, and OR-node corresponds to alternative selections or comparisons[Kum]. In this case, we have a regular AND/OR graph of height $m\log_p N$, whose AND-nodes have p branches and whose OR-nodes have m^{p-1}. Figure 5-5 shows an AND/OR graph that represents the reduction of the multistage graph problem with m=2 and p=2 from three stages to one stage, where AND-nodes are represented as circles and indicates summations, and OR-nodes are presented as squares and indicates comparisons. The values in the terminal nodes are edge cost: costs between stage 1 and stage 2 are $a_{i,j}$; costs between stage 2 and stage 3 are $b_{i,j}$, where $i,j \in \{1,2\}$. The four nodes at the top of the AND/OR graph represents the four possible alternate paths in the reduced single stage graph. The optimal path is obtained by a single comparison of these paths.

It is shown that the solution of a dynamic programming problem can be obtained by finding a minimum-cost path in an AND/OR graph. This equivalence allows various graph searching techniques to be used for solving a dynamic programming problem. Martelli and Montanari[Mar] proposed the top-down and bottom-up algorithm to find an optimal path in an AND/OR graph. A similar algorithm for searching AND/OR graphs was discussed by Nilsson[Nil].

By searching an AND/OR graph with height $m\log_p N$ to solve dynamic programming problems, the larger the value(=p) of the partitioned subproblems is, the less the Principle of Optimality is applied. In the extreme case, p=N, the corresponding AND/OR graph search becomes a brute-force search, and the Principle of Optimality is never used. For irregular multistage problems, the number of nodes in the AND/OR graph depends on the ordering of stage reduction. However, it is not difficult to demonstrate that binary partitioning is optimal. Another advantage of the AND/OR graph method is that the mapping of a regular graph onto a systolic array is straight forward[Gul].

5.3.2 State-Increment Dynamic Programming

The motivation to apply the State Increment Dynamic Programming (SIDP) to trajectory computation in the space-band regions is that the SIDP not only greatly saves fast memory but also can be implemented in parallel. The serial algorithm of the SIDP is described in reference [Bou]. Here this algorithm is summarized. Assume that a model for

aircraft trajectory computation is represented as the system of non-linear first-order differential equations or

$$\frac{d\bar{X}(t)}{dt} = \bar{f}[\bar{X}(t),\bar{U}(t),t] \tag{5.7}$$

where $\bar{X}$ = n-dimensional state vector

$\bar{U}$ = m-dimensional control vector

t = time, assumed to be stage variable

$\bar{f}[\cdot]$ = non-linear function vector

The constraints in the problem are expressed as

$$\begin{aligned} &\bar{X}(t) \in \mathbb{X}(t) \\ &\bar{U}(t) \in \mathbb{U}(\bar{X},t) \end{aligned} \tag{5.8}$$

Where $\mathbb{X}$, a set of admissible states, can vary with t, and $\mathbb{U}$, a set of admissible controls, can vary with $\bar{X}$ and t.

The system equations are discretized as

$$\bar{X}(k+1) = \bar{X}(k) + \bar{f}[\bar{X}(k),\bar{U}(k),k]\cdot\Delta t \tag{5.9}$$

In the computational procedure of the SIDP, the stage variable t is also quantized with an increment Δt, and minimum cost and optimal control are computed only at quantified values of $\bar{X}$ and t. However, δt_k, the time increment over which a given piece-wise constant control is applied, is determined as

$$\delta t_k = \min_i \left\{ \frac{\Delta x_i}{f_i[\bar{X}(k),\bar{U}(k)]}, \Delta t \right\} \qquad (5.10)$$

This equation ensures that the change in any state variable x_i over the increment δt_k is at most Δx_i and that δt_k is at most Δt.

The major consequence of this value of δt_k is that $I[\bar{X}(k)+\bar{f}[\cdot]\delta t_k, k+\delta t_k]$ can be determined using only the values of the minimum cost function at neighboring quantized states of several time increments. Interpolation is in (n-1) state variables and time t if $\delta t_k < \Delta t$ and in n state variables if $\delta t_k = \Delta t$.

The basic difference between the SIDP and conventional dynamic programming is time interval over which a given control is applied. With conventional dynamic programming this interval is a fixed value Δt. On the other hand, in the SIDP, this time increment is determined as the interval necessary for at least one of the n state variables to change by one increment δt_k.

This result can be exploited by processing data in units called blocks, which covers a few increments along each state variable axis but several time increments in t. A great savings in high-speed memory can be achieved.

The first step in the reduction of the high-speed memory requirement is the partitioning of the (n+1)-dimensional $\bar{X}$-t space

into rectangular units called blocks. Each block covers b_i increments along the x_i axis. The boundary between adjacent blocks is considered to be included in both blocks. The number b_i is taken as to be a small integer in sequential computation, usually

$$b_i \in [2,5] \tag{5.11}$$

and ΔT is taken as to be considerable larger than the average value of δt_k determined by equation (5.10). But in parallel computation, b_i depends on the method and hardware architecture to be used in parallelization.

The next step is to constrain the next state for any control to be in the same block as the state at which the control is applied. Since by the definition of δt_k, the next state must always lie in a neighborhood of the present state. This constraint is easy to implement in on-line computation. The reduction in the high-speed memory requirement occurs because only several values of the optimal cost function per state in the block are needed in the high-speed memory, rather than one value per state in the entire state space as in conventional dynamic programming.

In order to allow optimal trajectory to pass from block to block, the computation near the boundaries of a block must be slightly modified if its adjacent blocks have been processed. For two adjacent blocks, suppose that one block has been processed and the other has not; a sequence of optimal costs have been computed along the boundary

between these blocks. In the computation of the latter block, these optimal costs are stored in high-speed memory and then are used in evaluating the cost of the control applied at states within one increment of the boundary for which the next state lies on this boundary. It is then true that an optimal trajectory can transit from a state in the interior of the block being computed to the boundary of a previously computed block.

In most problems of physical significance, it is possible to obtain enough feeling about the system so as to determine the direction which optimal trajectories are most likely to take. This direction, for example, was defined as that in Section 5.2. The adjacent blocks in a given time interval are processed in an order such that interblock transition takes place in the preferred direction of motion.

5.4 Synthesis of Parallel Algorithms

It can be seen from the review in Section 5.3.1 that the apparent parallelism and simple data flow characteristics of some important dynamic programming algorithms allowed the design of very efficient VLSI arrays. However, the VLSI implementation of more complex algorithms such as the SIDP requires a precise design methodology. If such methodology is not available, the time and effort dedicated to parallel architecture and algorithm development may dominate the design cycle of new VLSI systems. Two frequently conflicting goals in

the VLSI design are short execution time and simple and regular communication between processors[Li-2]. Consequently, a systematic procedure for mapping algorithms into processor arrays must provide means for analyzing both parallelism and communication requirements and for transforming algorithms in order to improve their space/time characteristics. Previous work related to this problem has been reported in references [Joh,Cap,Kuh]. Johnson and Cohen [Joh] used the expression manipulation and operator calculus to mathematically model and design the VLSI arrays. Cappello and Steiglitz[Cap] discussed the application of geometric transforms to the design of systolic algorithms. Li and Wah[Li-1] showed that many of the VLSI algorithms described in reference [Kun-4] can be automatically generated from loop programs by using adequate index transformations. In this section, we consider the analysis and synthesis of parallel algorithms based on their data dependence.

A systematic method for the design of parallel algorithms is synthesized in this section and applied to the parallelization of the serial SIDP algorithm. This method is based on the data dependence structure of a serial algorithm. In particular, the serial SIDP algorithm is examined and then rewritten in a pipeline form which is suitable for the VLSI implementation. The pipelined algorithm is indexed so that the algorithm in index form is obtained. The data dependence matrix is extracted from the indexed algorithm. Alternatively, such an algorithm, characterized by the data dependence matrix, is interpreted as an acyclic graph with systolic structure. At last, the algorithm is mapped onto a systolic array with specified

structure, where the matrix of space/time transformation of the mapping is achieved through the optimal solution to the linear integer programming problem.

5.4.1 Serial SIDP Algorithm

The forward recursive functional procedure for the serial SIDP algorithm is given in natural language:

ALGORITHM I:

```
for each stage k
   for each state X̄_i(k), i=1, ·····, I
      for each control Ū_j , j=1, ·····, J
          compute  F[X̄_i(k), Ū_j]
          compute  δt
          compute  X̄_i(k+δt)
          compute  L[X̄_i(k+1)]
          compute I[k+1]
          store Ū_j(k)
      end for
   end for
end for
```

These nested loops consume most of the computational time in trajectory solutions. To make on-line computation feasible, parallelizing the loops is necessary. The loops are rewritten as

follows, where main computations are listed.

ALGORITHM II:

for each stage k

.

.

.

for each state $\bar{X}_i(k)$, i=1, ·····, I

for each control $\bar{U}_j$, j=1, ·····, J

S1: $d\bar{X} = \bar{F}[\bar{X}_i(k),\bar{U}_j,k]$

$\delta t = \max\left\{ \frac{\Delta x_p}{d\bar{X}}, \Delta t \,\middle|\, p=1,\cdots,n \right\}$

$\bar{X}_i(k+\delta t) = \bar{X}_i(k) + d\bar{X}\cdot\delta t$

$L[\bar{X}_i(k+1),k+1] = \text{Interp}\{\bar{X}_i(k),\bar{X}_i(k+\delta t),\bar{X}_i(k+1)\}$

$I[\bar{X}_i(k+1),k+1] =$

$\min_u\{I[\bar{X}_i(k+1),k+1], L[\bar{X}_i(k+1),k+1]+I[\bar{X}_i(k),k]\}$

$\bar{U}^*(k) = \text{aug}\ \min_u\{I[\bar{X}_i(k+1,k+1], L[\bar{X}_i(k+1),k+1]+I[\bar{X}_i(k),k]\}$

end for

end for

.

.

.

end for

where the $\bar{F}[\bar{X}_i(k),\bar{U}_j,k]$ is defined in Section 5.3.2. The $L[\bar{X}_i(k+1),k+1]$ is the one step cost at state $\bar{X}_i(k+1)$ at stage k+1, which is calculated by interpolation (Interp{·}).

The complexity of this algorithm is O(I*J*n). For a model we consider in this dissertation, we have $2 \le n \le 6$, and hence $O(I*J*n) \cong O(I*J)$. The number of control variables is always less than

that of state variables, or $J \leq kI$, where $0 \leq k \leq 1$. The complexity is at most $O(I^2)$. If m processors are used in the implementation of the algorithm, we have $O(I^2/m)$. Using a systolic array, we can assume $m \cong I$. Thus, the complexity of the resulting algorithm becomes $O(I)$, which, with appropriate hardware speed, can be executed in real time.

5.4.2 Pipelining

Since it is very expensive to implement broadcasts in the VLSI circuits, the first step is to eliminate all possible data broadcasts which may exist in the original serial algorithm. Typically, data broadcasts are signaled by loop variables with missing indexes (not all loop indexes included). In order to avoid broadcasts and increase pipelining, we first complete all the missing indexes and introduce new artificial variables such that each generated variable has only one destination. The following are two examples which demonstrate how to eliminate data broadcasts and thereby make the algorithm pipelined. In the serial SIDP algorithm, the indices are added in the statement S1 as

Example 5.1: $$d\bar{X}(k,\bar{I},j) = \bar{F}[\bar{X}_1(k,\bar{I},j),\bar{U}(k,\bar{I},j),k]$$

where the vector $\bar{I}$ is defined as $[i_1,i_2,\cdots,i_n]$ for the purpose of concise expression. The i_p is the index of state x_p for $p=1,\cdots,n$.

To further prepare a process for implementation into a parallel algorithm, a statement which references itself is eliminated by

distinguishing between distinct uses of the same symbol. Also each time a variable is reassigned a value, it is renamed, and even if a variable unchanged, it may be necessary to add indices and to distinguish this unchanged variable which is referenced by several statements. For example, the value of the variable $\bar{U}_j$ in the statement S1 remains constant in its inner loop, so an extra statement is augmented such as

Example 5.2: $\bar{U}(k,\bar{I},j) = \bar{U}(k-1,\bar{I},j)$

By means of the renamed variable and the augmented statement, the data broadcast has been eliminated. This process of eliminating all the data broadcast is referred to as pipelining.

Note that the new variables are different from the corresponding old variables only in their indices. In this way, each time that the statements are performed, the indices are different, and hence each index vector related to a variable has a unique value. In other words, the value of a variable is uniquely parameterized at each step of the computation by the index. As a result, the statement is indexed totally.

By repeating this operation for all the statements, all the data broadcasts can be eliminated and each variable depends explicitly on its loop index. The pipelined and indexed SIDP algorithm is obtained as follows

ALGORITHM III:

for each stage k

.
.
.

for each state $\bar{X}_i(k)$, i=1, ·····, I

for each control $\bar{U}_j$, j=1, ·····, J

$\bar{U}(k,\bar{I},j) = \bar{U}(k-1,\bar{I},j)$

S1: $d\bar{X}(k,\bar{I},j) = \bar{F}[\bar{X}(k,\bar{I},j),\bar{U}(k,\bar{I},j),k]$

$$\delta t(k,\bar{I},j) = \max\left\{ \frac{\Delta x_p}{d\bar{X}(k,\bar{I},j)}, \Delta t \,\middle|\, p=1,\cdots,n \right\}$$

$\bar{X}(k+\delta t,\bar{I},j) = \bar{X}(k,\bar{I},j) + d\bar{X}(k,\bar{I},j)\cdot\delta t(k,\bar{I},j)$

$L[\bar{X}(k+1,\bar{I}+\Delta\bar{I},j),k+1] = \text{Interp}\{\bar{X}(k,\bar{I},j),$

$\bar{X}(k+\delta t,\bar{I},j),\bar{X}(k+1,\bar{I}+\Delta\bar{I},j)\}$

$I[\bar{X}(k+1,\bar{I}+\Delta\bar{I},j),k+1] = \min_u\{I[\bar{X}(k,\bar{I}+\Delta\bar{I},j),k+1],$

$L[\bar{X}(k+1,\bar{I}+\Delta\bar{I},j),k+1]+I[\bar{X}(k,\bar{I},j),k]\}$

$\bar{U}^*(k,\bar{I},j) = \text{aug}\min_u\{I[\bar{X}(k,\bar{I}+\Delta\bar{I},j),k+1],$

$L[\bar{X}(k,\bar{I}+\Delta\bar{I},j),k+1]+I[\bar{X}(k,\bar{I},j),k]\}$

end for

end for

.
.
.

end for

where $\Delta\bar{I}=[\Delta i_1,\Delta i_2,\cdots,\Delta i_n]$ and $\Delta i_p \in \{-1,0,+1\}$ for p=1,···,n.

Such processing does not change the original algorithm fundamentally, but only changes the data dependence among the variables. The data dependence in the resulting algorithm is therefore not characteristic of the problem itself but of its parallel

implementation.

5.4.3 Data Dependence Structure of Algorithm

In the pipelined and indexed algorithm, each data must be from somewhere. The term pipelining emphasizes this viewpoint. In fact, all the variables are supplied with their indices so that they become pipelined with respect to their indices.

Based on the fact that all of the indexed variables are parallelized by their loop indices, an appropriate mathematical structure is now associated with the prepared algorithm so that an analysis of data dependence can be performed.

To describe the dependence structure of the algorithm, we define a vector distance which is from the index of computation (or the right side of a statement) to the index point (or the left side of the statement) where the variable is produced. This vector, called the data dependence vector, is null when a variable is an input or is generated by the computation itself. The resulting relation among the data dependence vector, the variable, and indices can be represented by a matrix, called the data dependence matrix, whose columns are the data dependence vector and are labeled by the corresponding variables and index points.

In the following, the energy-state-model with six states is analyzed as an example, where n=6. For the example 5.1 and the example 5.2, the data dependence vectors are, respectively, by the above definition

$$\bar{d}_0 = [0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^T$$
$$\bar{d}_1 = [1\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^T \qquad (5.12)$$

where the null vector $\bar{d}_0$ means the corresponding operation (or the execution of the statement S1) belongs to internal computation and does not enter pipelining. In either $\bar{d}_0$ or $\bar{d}_1$, the first element is corresponding to the distance of index k (stage); the second to seventh elements are corresponding to the distances of indices i_1 (state x_1) to i_6 (state x_6); the last element is corresponding to the distance of index j (control).

For the inner loop in the serial SIDP algorithm, $\Delta\bar{i}$ represents a vector or $[\Delta i_1, \Delta i_2, \Delta i_3, \Delta i_4, \Delta i_5, \Delta i_6]$. By the definition of δt in the SIDP, each element in $\Delta\bar{i}$ must be -1, 0, or +1, whereas it could be less than -1 or greater than +1 in conventional dynamic programming. In other words, the data dependence in the SIDP is weaker than in conventional DP. For the prepared algorithm, the data dependence matrix is extracted from ALGORITHM III:

$$D = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(5.13)

where each column stands for a data dependence vector which is determined by the above definition and the null dependence vectors are not included.

5.4.4 Alternative Representation of Algorithm: Acyclic Graph in Systolic Structure

Assume that the index points are referred to as the nodes of the computation. The currently executed statement requires the results of the previous computation or several nodes at the previous stage. The set of these nodes is defined as the domain of data dependence.

From the data dependence matrix D, it can be seen that one of the advantages of using the SIDP is that the domain of data dependence in the SIDP is much less than that in conventional dynamic programming. Such domain of data dependence plays an important role in the VLSI implementation.

Let $\mathbb{N}$ be the set of the nodes of the computations of the SIDP

algorithm and $\mathbb{D}$ be the set of the nodes of the domain of data dependence. The data dependence vectors are selected to point from a node to the nodes in its domain of data dependence. In other words, the referenced nodes are given as explicit function of the current node. Taking the data dependence vectors to be the set of direct edges $\mathbb{E}$, the pair $(\mathbb{N},\mathbb{E})$ forms an acyclic graph for the reason of the algorithm is already pipelined. On the other hand, such collection of the vectors is independent of the nodes in $\mathbb{N}$ due to the SIDP algorithm with the preferred direction. Therefore, the graph $(\mathbb{N},\mathbb{E})$ is also in systolic structure. This acyclic graph in systolic structure means that each node contains the same computation and the links between them are uniform. In conclusion, the algorithm carried by the graph can perform in parallel. In Section 5.4.5, it will be mapped onto a systolic array which has the same characteristics as the acyclic graph in systolic structure.

5.4.5 Space/Time Transformation

The abstract flow of information in the algorithm is characterized by means of the parallel computational structure or the graph $(\mathbb{N},\mathbb{E})$ which is acyclic and systolic, and developed from the data dependence vectors. Naturally at next step, the realization of this flow of information in a real world or a parallel machine need to be studied.

Lemma: There exists at least one mapping which maps the SIDP

algorithm on to a systolic array.

Proof: (i) The pipelined SIDP algorithm can be represented by an acyclic graph $(\mathbb{N},\mathbb{E})$.

(ii) Every acyclic graph has a physical realization in a systolic structure.

Q.E.D.

Remark: The statement (ii) is easily approved by the so-called *von Neumann implementation* of an algorithm.

To execute the algorithm in parallel on a systolic array gives rise to a mapping which actually is a space/time transformation:

$$(\mathbb{N},\mathbb{E}) \xrightarrow{\Omega} (\mathbb{P},\mathbb{L}) \tag{5.14}$$

where $\mathbb{P}$ is a set of index points each of which stands for a processor, $\mathbb{L}$ is a set of interconnections between the processors, and Ω is a linear space/time transformation which maps the serial algorithm, represented by the graph $(\mathbb{N},\mathbb{E})$, onto a systolic array with a structure $(\mathbb{P},\mathbb{L})$. The Ω is expressed as

$$\Omega = \begin{pmatrix} T \\ S \end{pmatrix} \tag{5.15}$$

The T matrix defines the time transformation whereas the S matrix defines the space transformation to be applied to the data dependence matrix and the index set of an algorithm.

Now let us find Ω. The time, at which a computation indexed by $\bar{j}=(k,\bar{I},j)$ is executed, is determined by $T\bar{j}$, while $S\bar{j}$ specifies which processor is to execute this computation. Of course, T and S must satisfy certain conditions if they are to be considered valid transformations. In fact, the data dependence matrix D imposes the constraint on Ω. Let R be the number of columns in D, then the time transformation must satisfy

$$T\bar{d}_r > 0 \qquad \text{for } r=1,\cdots,R \tag{5.16}$$

where $\bar{d}_r$ is the r^{th} column vector in D. The constraint (5.16) results from the requirement that a variable must be generated before it is used in a computation.

The space transformation S maps the computation indexed by $\bar{j}$ into processor $S\bar{j}$. This assumes a systolic array model consisting of a grid which has the dimensionality of the array. Each point of the grid corresponds to a processor and the coordinates of the point are the index of the processor. Certain restrictions must be placed on possible solutions for S due to the limited regular interconnection arrays available in the VLSI technology. These restrictions can be embodied in the P and K matrices. The P matrix describes the interconnection mode of a specified array which the algorithm is going to map on. The K matrix describes the interconnections used by the transformed algorithm during execution. The relationship between D, S, P, and K obeys

$$SD = PK \tag{5.17}$$

where the entries of K are constrained by

$$\sum_{v=1}^{V} k_{vr} \leq T\bar{d}_r \tag{5.18}$$

The constraint (5.18) requires that time between the generation and use of a variable must be greater than or equal to the number of interconnection primitives needed by the datum to travel from the processor in which it is generated to the processor in which it will be used.

The solution to the equation (5.17) may not be unique or does not exist for each of the possible choice K. If no solution is found, then a new time transformation T must be be selected by the condition (5.16). This means that the time transformation must be traded for the space transformation S. If more than one solution exist, then the one which requires a smaller number of processors should be selected.

Theorem: *Let T be the time transformation and S be the space transformation. Then the mapping* $\Omega = \begin{pmatrix} T \\ S \end{pmatrix}$, *where* Ω: $(\mathbb{N},\mathbb{E}) \xrightarrow{\Omega} (\mathbb{P},\mathbb{L})$, *is optimal in execution time and the number of processors.*

Proof: (1) A mapping must exist by *Lemma.*

(2) Define an optimization problem that minimizes the

norms of the images of all the data dependence vector $\bar{d}_r$ under each solution S:

$$\min_S \left\{ \max_r \left\{ \sum_{p=1}^{P} \sum_{q=1}^{Q} |S_{pq} d_{qr}| \right\} \right\} \qquad (5.19)$$

subject to: $SD = PK$

$$\sum_{v=1}^{V} k_{vr} \leq T\bar{d}_r$$

$$T\bar{d}_r > 0 \qquad \text{for } r=1,\cdots,R$$

Solving this linear integer programming problem (Because the entries of the vectors and the matrices in formula (5.19) are required to be integers.) produces a mapping Ω. It is optimal in time and space because linear integer programming produces an optimal solution subject to the constraints which are defined in time and space.

Q.E.D.

We still use the six-state model as an example. The data dependence matrix D is already obtained. If the matrix P is known, the linear integer programming problem (5.19) can be solved.

Due to inherent physical constraints imposed by driving capability of communication channels, power consumption, heat dissipations, and memory bandwidth in the VLSI circuits, it is reasonable to assume that data can only sent or received simultaneously to or from a limited number of destinations or sources. If the systolic array which the

algorithm maps onto is given in Figure 3-1-(3), then by definition in Section 3.1 the corresponding pattern matrix P is rewritten:

$$P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.20}$$

the optimal solution to the linear integer programming problem (5.19) for the serial SIDP algorithm of on-line trajectory optimization is found as follows:

$$T = [2 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0]$$

$$S = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \tag{5.21}$$

The design of the parallel algorithm for trajectory optimization is completed through the transformation Ω which maps the serial SIDP algorithm onto the cubic mesh-connected systolic array. In summary, a computation with index $\bar{j}=(k,\bar{l},j)$ is executed at time $T\bar{j}$ in processor $S\bar{j}$ using data generated at time $T(\bar{j}-\bar{d}_r)$ in processor $S(\bar{j}-\bar{d}_r)$, where $r=1,\cdots,R$. By using the resultant parallel algorithm structure, the timing and direction of data transmission and the functional description of the computational cells or processors of the array architecture can be systematically driven in computer engineering.

5.5 Parallel Architecture

The evolution in the VLSI technology has a great impact on computer architecture. A hardware model in the form of a simple block diagram for the implementation of the parallel algorithm for on-line optimal trajectory computation is given in this section.

The system, as shown in Figure 5-6, is composed of a central processing unit (CPU), m systolic arrays, buffers, and global memory. The CPU complete initialization, block partition, and necessary global control, or all the computations and operations out of the inner loop in algorithm III in Section 5.4.2. Each systolic array completes the computation of the inner loop in parallel for the states in a block. The m systolic arrays work in parallel. The features about systolic array systems are assumed as follows

(1) The systolic array system consists of a regular cubic connected network of processing cells or processors.

(2) The cells can be pre-programmable and perform different functions.

(3) The interconnections between cells are buses which transfer parallel words.

(4) The operations of the network is synchronous.

The buffers between two systolic arrays are used to transmit data on the boundaries of two blocks. The buffers between the global bus and the systolic arrays are placed to put data in queues, which are input into the systolic arrays, and buffer the resulting data from the

systolic arrays.

When the size of the SIDP problem is greater than the size of a systolic array available, more than one systolic array is needed. The number M of the needed systolic arrays depends on the size of the systolic array to be applied and the size of the SIDP problem. Assume that there are I states at a stage in the SIDP algorithm, then I states are partitioned such that the size of each partitioned part fits the size of the systolic array.

5.6 Summary

In order to make it feasible to on-line calculate an optimal trajectory by the SIDP for a velocity-variant, three-dimensional model in the TF/TA environment, the entire computational state space in the flight region defined by a mission was broken into several reduced space-band regions. In pursuit of this goal, the approach was developed which is to quickly determine a group of waypoints, form an initial reference trajectory by connecting the waypoints, and generate a band region by expanding across the initial reference trajectory in a certain distance.

To further realize the on-line computation of optimal trajectories within the space-band region, a systematic method for the design of the parallel algorithm has been studied. This systematic method, based on a data dependence structure, was synthesized and applied to the

parallelization of the SIDP algorithm. In particular, the serial SIDP algorithm was examined and was rewritten in a pipeline form which is suitable for the VLSI implementation. The pipelined algorithm was indexed so that the algorithm in an index form was obtained. The data dependence matrix was extracted from the indexed algorithm. Alternatively, such an algorithm, characterized by the data dependence matrix, was also interpreted as an acyclic graph with systolic structure. At last, the algorithm was mapped onto a systolic array with specified structure, where the matrix of space/time transformation of the mapping was achieved through the optimal solution to the linear integer programming problem. The detailed design approach was demonstrated by mapping the serial SIDP algorithm onto a cubic mesh-connected systolic array.

The hardware system was described in the form of block structure for the implementation of the parallel algorithms designed by the systematic method. Such a system consists of a host microprocessor or central processing unit (CPU) and several systolic arrays. The host microprocessor is responsible for high-level execution and overall timing control. The systolic arrays, as peripheral devices, execute the operations of the algorithm in parallel. The number of the systolic arrays needed depends on the size of the systolic arrays to be applied and the size of the problem, and is determined such that the size of the partitioned subproblem fits the size of the available systolic arrays.

Unfortunately, the simulation of the parallel algorithm deigned by

the systematic method has not been conducted due to the lack of applicable parallel hardware.

Fig. 5-1 Simulated Terrain Profile

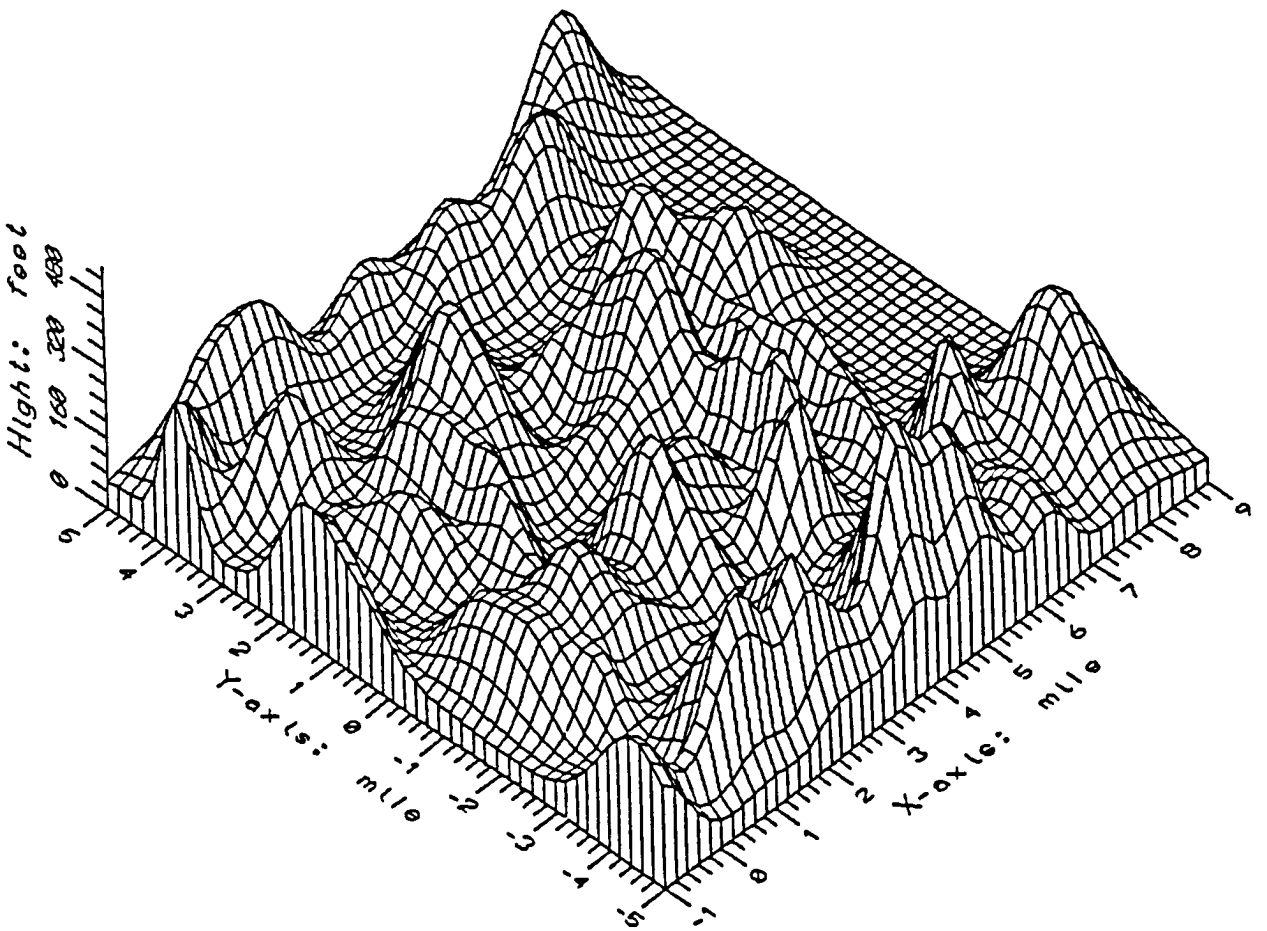


Fig. 5–2 Determination of Waypoints

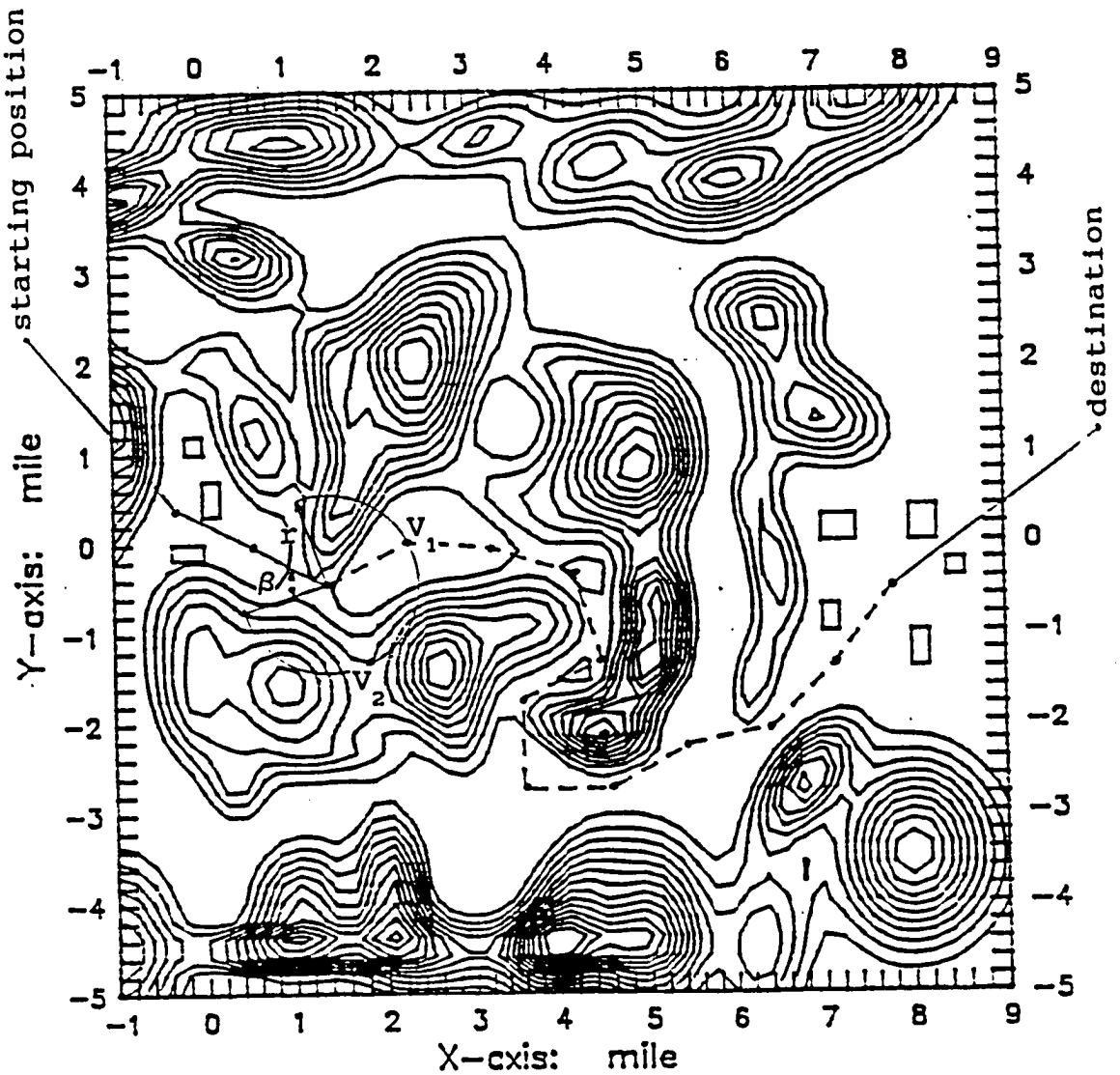
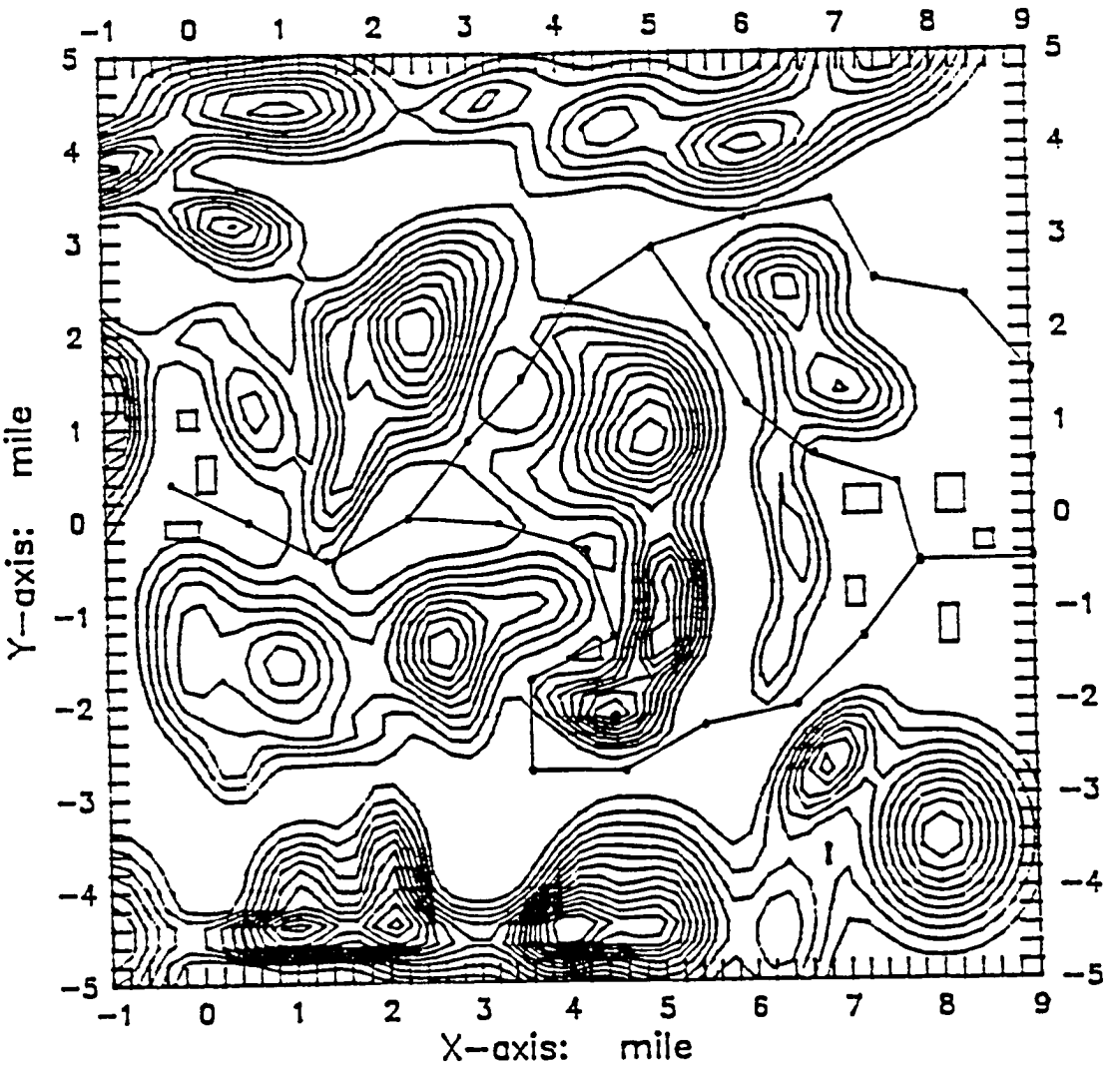


Fig. 5–3 Initial Reference Trajectories



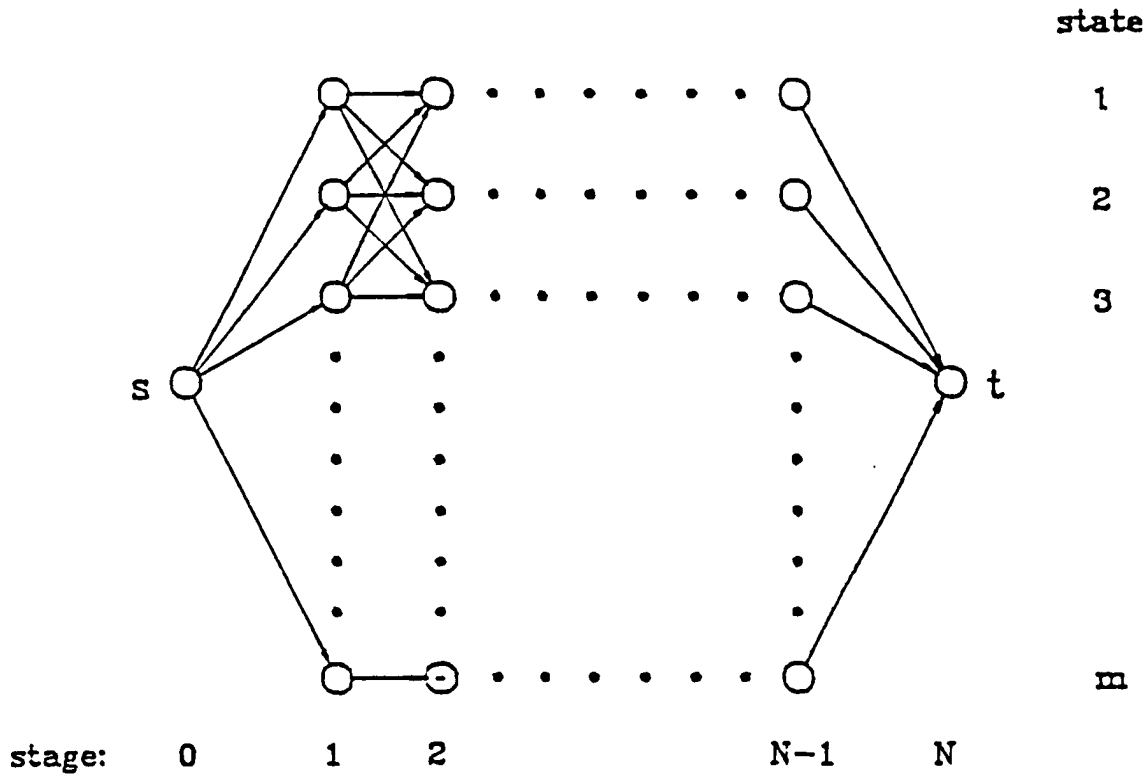
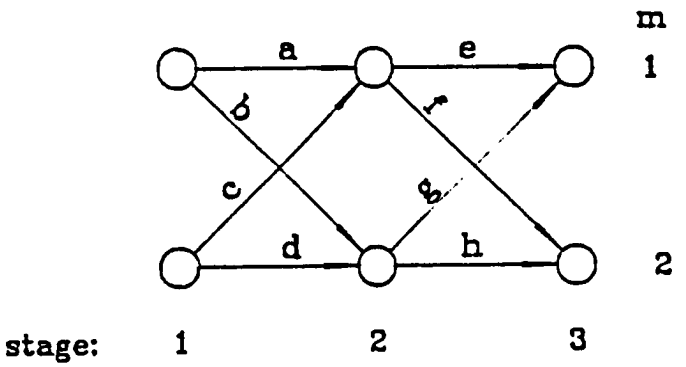
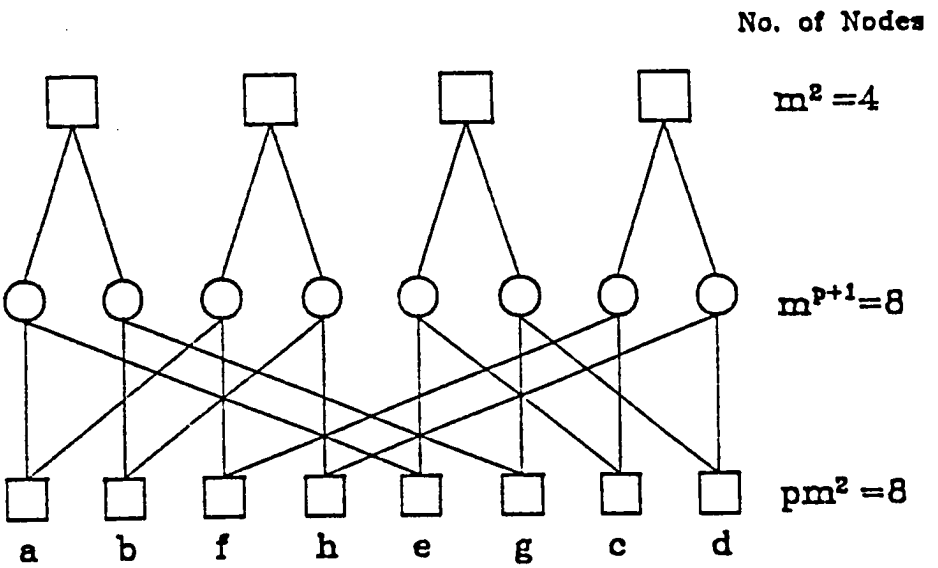


Fig. 5-4 Multistage Graph for Dynamic Programming



(1) Multistage Graph with m=2



(2) AND/OR Graph

Fgi. 5-5 AND/OR Graph Representation of Multistage Graph

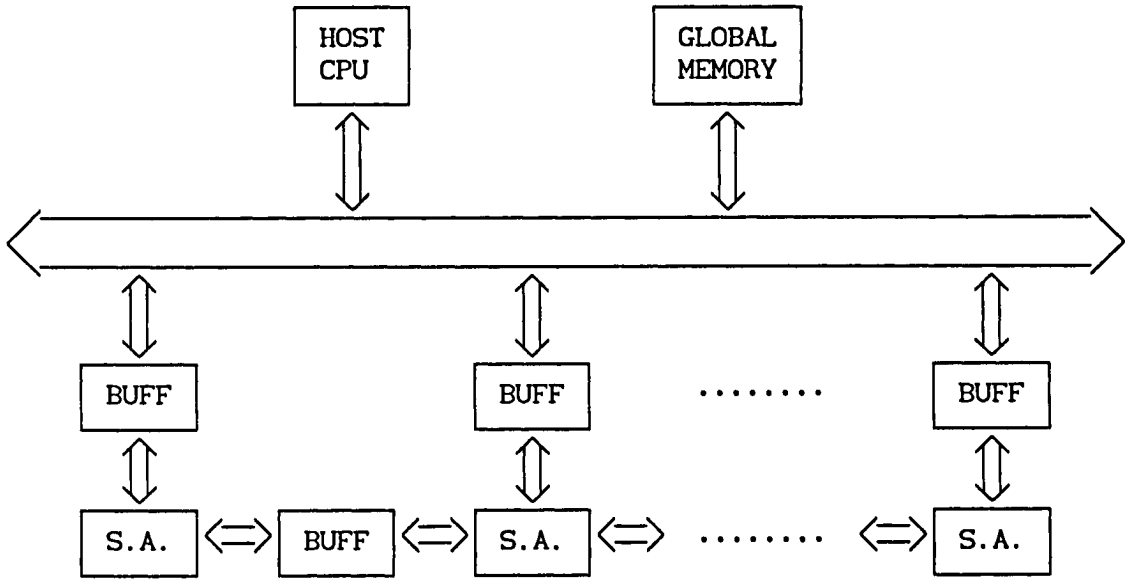


Fig. 5-6 Block Structure for Parallel SIDP Algorithm

6.0 CONCLUSIONS AND RECOMMENDATIONS

Summary and Conclusions

The investigation for the on-line trajectory optimization problem has been conducted in this research work. The problem was formulated as an optimal control problem by dynamic programming to deal with a complex constraint environment. The performance index was defined as a linear combination of four weighted items while considering a trajectory that achieves better terrain masking, minimum flight time, and minimum exposure to threats.

In order to make it feasible to calculate on-line an optimal trajectory by dynamic programming, the entire computational state space in the flight region defined by a mission was broken into several reduced band regions. In pursuit of this goal, the approach was developed which is to quickly determine a group of waypoints, form an initial reference trajectory by connecting the waypoints, and generate a band region by expanding across the initial reference trajectory in a certain distance.

To further realize the on-line computation of optimal trajectories within the band region, the two candidate methods for the design of parallel algorithms have been studied. The first one, a direct method, is applied to a velocity-invariant model with two states (position)

and a control (heading) for aircraft flight path optimization through a complex constraint region. The two parallel systolic algorithms and architectures were designed in this case, The first systolic algorithm is formulated as a string of equivalent "matrix-by-vector" minimum operations executed on linear arrays. The second systolic algorithm is formulated as a string of equivalent "matrix-by-matrix" minimum operations executed on rectangular arrays. Computational models of the linear arrays and rectangular arrays were also conceived. The second algorithm has a higher parallelism than the first one, which means the execution time of the second algorithm is less than that of the first one but at the cost of more complex architecture.

The second method, a systematic method, is applied to the energy-state model of a helicopter for three-dimensional and velocity-variant trajectory computation by the SIDP in a complex constraint environment such as the TF/TA environment. This systematic method, based on a data dependence structure, was synthesized and applied to the parallelization of the SIDP algorithm. In particular, the serial SIDP algorithm was examined and was rewritten in a pipeline form suitable for the VLSI implementation. The pipelined algorithm was indexed so that the algorithm in an index form was obtained. The data dependence matrix was extracted from the indexed algorithm. Alternatively, such an algorithm, characterized by the data dependence matrix, was also interpreted as an acyclic graph with systolic structure. At last, the algorithm was mapped onto a systolic array with specified structure, where the matrix of space/time transformation of the mapping was achieved through the optimal

solution to the linear integer programming problem. The detailed design approach was demonstrated by mapping the serial SIDP algorithm onto a cubic mesh-connected systolic array.

The hardware systems were described in the form of a block structure for the implementation of the parallel algorithms designed by both the direct method and the systematic method. Such a system consists of a host microprocessor or central processing unit (CPU) and several systolic arrays. The host microprocessor is responsible for high-level execution and overall timing control. The systolic arrays, as peripheral devices, execute the operations of the algorithm in parallel.

The simulation of the parallel algorithm designed by the direct method has been completed. The numerical results have shown that the algorithm applied to the determination of a feasible trajectory and of the iterative solution of a local optimal trajectory is practical. At this point, the estimate of the timing requirements in a parallel machine has not been made, but it is felt that this approach has effectively made an on-line trajectory optimization algorithm feasible.

Unfortunately, the simulation of the parallel algorithm deigned by the systematic method has not been conducted due to the lack of applicable parallel hardware.

As stated above, the main contribution of this research work has

been to synthesize a methodology, which includes the approach to the determination of initial reference trajectories and the design of parallel algorithms by the direct method or the systematic method. The resulting parallel algorithms by this methodology make it feasible to compute on-line an optimal trajectory by dynamic programming in a complex constraint environment in real time as soon as systolic arrays are available. In fact, the advances in the VLSI technology have shown that such systolic arrays will be available in the near future.

This methodology is not restricted to the problem associated with the model of a helicopter and the TF/TA environment. Actually, it can be applied to other trajectory optimization problems in different constraint environment such that the Air Traffic Control problem could be a perspective application.

It has been shown from the synthesis of the systematic method that the SIDP algorithm appeared only as an example of the design by the method, which means that no matter what a problem is, the application of the method only depends on the form of the algorithm of that problem. This systematic method has many applications to the effective design of the parallel algorithms of problems which have such forms as recurrence equations, uniform iterative equations, and nested loops with constant data dependence.

Recommendations for Future Work

The parallel algorithms designed in this dissertation are based on a special parallel computer system with systolic arrays. The different computer systems for the different parallel algorithms have been described in the form of simple block structures. However, a more detail and precise parallel computer model for the implementation of the designed parallel algorithms is needed to be built in future work in order to analyze the complete implementation of the parallel algorithm. (This needs a sound background in computer engineering.) With the designed parallel algorithms and the detail and precise computer model, trade-off between processor number, memory size, computational time, and communication time can be made carefully. Then, the estimate of timing requirements can be obtained by further simulation.

Simulation results for using a simple model have been obtained by using only a single micro-computer in the case of using a simple model and no actual parallel hardware has been utilized. Future efforts must be directed to quantify performance advantages expected in the implementation of the parallel algorithms, in a realistic parallel hardware environment or a computer system with parallel architecture.

Bibliography

[Alo] A.V. Alo et al., *Design and Analysis of Algorithms*, Addison-Wesley, 1976.

[Bar] J.F. Barman and H. Erzberger, "Fixed-Range Optimum Trajectories for Short-Haul Aircraft", *Journal of Aircraft*, Vol. 13, No.10, Oct. 1976.

[Bel] R. Bellman and S. Dreyfus, *Applied Dynamic Programing*, Prinston University Press, N.J., 1962.

[Bou] R. Boudarel, J. Delmas, and P. Guichet, *Dynamic Programming and its Application to Optimal Control*, Academic Press, 1971.

[Bry] A.E. Bryson, M.N. Desai, and W.L. Hoffman, "The Energy State Approximation in Performance Optimization of Supersonic Aircraft", *Journal of Aircraft*, Vol. 6, Nov.-Dec. 1969.

[Cal-1] A.J. Calise, "Extended Energy Management for Variational Problems in Aircraft Flight", *AIAA Journal*, Vol. 15, March 1977.

[Cal-2] A.J. Calise and D.D. Moerder, "Singular Perturbation Techniques for Real Time Aircraft Trajectory Optimization and Control", NASA Report, N82-31330.

[Cal-3] A.J. Calise, G.A. Flandro, and J.E. Corban, "Trajectory Optimization and Guidance Law development for National Aerospace Plan Applications", NASA Report, N89-12538.

[Cap] P.R. Cappello and L. Steiglitz, "Unifying VLSI Array Design with Geometric Transformation", *Proc. of 1983 Conf. on Parallel Processing*, pp. 448-457.

[Che] G. Chen et al., "Pipeline Architectures for Dynamic Programming Algorithms", *Parallel Computing*, Vol. 13, 1990, pp. 111-117.

[Cur] H.C. Jr. Curtiss, "Studies of Rotorcraft Agility and Maneuverability", *Tenth European Rotorcraft Forum*, August 1984.

[Den-1] R.V. Denton, R.P. Denaro, and J.E. Jones, "Use of the DMA Digital Terrain Elevation Data Base for Flight Trajectory

Generation, Terrain Following/Terrain Avoidance, and Weapon Delivery", AGARD-LS-122, March 1983.

[Den-2] R.V. Denton, J. Jones, and P.L. Froeberg, "A New Technique Terrain Following/Terrain Avoidance Guidance Generation", AGARD-CP-387.

[Des] O.I. El-Dessouki and W.H. Huen, "Distributed Enumeration on Network Computers", *IEEE Trans. on Computers*, Vol. C-29, No. 9, pp. 818-825.

[Dor] D.R. Dorr, "Rotary Wing Aircraft Terrain Following/Terrain Avoidance System Development", NASA-TM-88322, 1986.

[Fis] A. L. Fisher and H. T. Kung, "Synchronizing Large VLSI Processor Arrays", *ACM symposium on Computers Architecture*, June, 1983.

[Gel] D. Gelernter, "Parallel Processing: Getting the Job Done", *BYTE*, Nov. 1988.

[Geo] A.M. Geoffrion and R.E. Marsten, "Integer Programming Algorithms: A Framework and State-of-the-Art Survey", *Management Science*, Vol. 18, 1972, pp. 465-491.

[Gui] L. Guibas, "Direct VLSI Implementation for combinational Algorithms", *Proc. of Caltech Conf. on VLSI: Architectures, Design, and Fabrication*, 1979, pp 509-525.

[Hen] J.K. Hendrick and A.E. Bryson, "The Three-Dimensional Minimum Time Turns for a Supersonic Aircraft", *Journal of Aircraft*, Vol. 9, Feb. 1972.

[Hof] J.D. Hoffman, "Terrain Following/Terrain Avoidance for Helicopter Applications", *AHS Proceedings National Specialists' Meeting in Rotorcraft Flight Controls and Avionics*, Oct. 1987.

[Hoh] R.H. Hoh, et al., "Background Information and User's Guide for Proposed Handling Qualities Requirements for Military Aircraft". NAS2-11304, 1985.

[Hua] H. Huang and H. Liu, "A Sublinear Parallel Algorithm for Some Dynamic Programming Problems", *Proc. of 1990 Int'l Conf. on Parallel Processing*, Vol. 3, pp 261-264.

[Iba] T. Ibaraki, "Solvable Classes of Discrete Dynamic Programming", *J. of Mathematical Analysis and Application*,

Vol. 43, 1973, pp. 642-693.

[Ima] M. Imai et al., "A Parallelized Branch and Bound Algorithm: Implementation and Efficiency", *System Computer Controls*, Vol. 10, No. 3, 1979, pp 62-70.

[Joh] L. Johnson and D. Cohen, "A Mathematical Approach to Modeling the Flow of Data and Control in Computational Networks", *CMU Conf. on VLSI Systems and Comp.*, Oct. 1981, pp. 226-234.

[Kar] R. Karp and M. Held, "Finite State Process and Dynamic Programming", *SIAM J. on Appl. Math.*, Vol. 15, 1967, pp. 693-718.

[Kel] H.J. Kelley and L. Lefton, "Supersonic Aircraft Energy Turns", *5th IFAC Conference*, Paris, France, June 1972.

[Kuh] R.H. Kuhn, "Optimization and Interconnection Complexity for Parallel Processors", Ph.D. Thesis, Dept. of Computer Science, University of Illinois, Feb. 1980.

[Kum] V. Kumar, "A General Bottom-up Procedure for Searching in AND/OR Graphs", *Proc. AAAI*, 1984, pp. 182-187.

[Kun-1] H.T. Kung and C.E. Leiserson, "Systolic Arrays for VLSI", *Sparse Matrix Proc. 1987*, Academic Press, Orlando, Fla., pp. 256-282.

[Kun-2] H.T. Kung, "Why Systolic Architectures?", *IEEE Trans. on Computers*, Jan. 1982, pp 37-46.

[Kun-3] H.T. Kung, " The Structures of Parallel Algorithms", *Advances in Computers*, Vol. 19, Academic Press, New York, 1980, pp.65-112.

[Kun-4] H.T. Kung and C. Leiserson, "Algorithms for VLSI Processor Arrays", *Introduction to VLSI Systems*, Addison-Wesley, Reading, Mass., 1980.

[Lai-1] T. Lai and A. Sprague, "Performance of Branch and Bound Algorithms", IEEE Trans. on Computers, Vol. C-34, No. 10, 1985, pp. 194-201.

[Lai-2] H. Lai and S. Sahni, "Anomalies on Branch and Bound", *Com. of ACM*, Vol. 27, No. 6, 1984, pp. 594-602.

[Lar] R.E. Larson and J.L. Casti, *Principles of Dynamic Programming*, Marcel Dekker, Inc., New York and Basel, 1978.

[Leg] P.J. Legge, P.W. Fortescue, and P. T. aylor, "Preliminary

Investigation into the Addition of Auxiliary Longitudinal Thrust on Helicopter Agility". HMSO, 1981.

[Li-1] G.-L. Li and B. W. Wah, "The Design of Optimal Systolic Algorithms", *IEEE Trans. on Computers*, Vol. C-34, No. 1, Jan., 1985.

[Li-2] G-L. Li and B.W. Wah, "The Design of Optimal Systolic Arrays", *IEEE Trans. on Computers*, Vol. C-34, No. 1, Jan. 1985.

[Lin] B. Lint et al., "Communication Issues in Design and analysis of Parallel Algorithm", *IEEE Trans. on Software Engr.*, Vol. 7, No. 2, March 1981.

[Mar] A. Martelli and U. Montanari, "Additive AND/OR Graphs", *IJCAI*, 1973, pp. 1-11.

[Mea] C. A. Mead and L. A. Conway, *Introduction to VLSI Systems*, Addison-Wesley, Reading, Massachusetts, 1980.

[Men] P.K.A. Menon, E. Kim, V.L. Cheng, "Optimal Trajectory Synthesis for Terrain Following Flight", *J. of Guidance, Control, and Dynamics*, 1990.

[Mik] J. Miklosko, al et, *fast Algorithms and their Implementation on Specialized Parallel Computers*, North-Holland, 1989.

[MIT] "Advanced Research in VLSI: Proc. of the Fifth MIT Conference", March 1988.

[Mor] T.L. Morin and R.E. Marsten, "Branch and Bound Strategies for Dynamic Programming", *Operation Research*, Vol. 24, No. 4, 1976.

[Nil] N.J. Nilsson, *Principle of Artificial Intelligence*, Tioga, Palo Alto, CA, 1980.

[Nor] R. Nordmeyer, "Enhanced Terrain Masked Penetration Final Technical Report", AFWAL-TR-86-1079, Sept. 1986.

[Ols] J. Olsen, "Helicopter Mission Optimization Study", NASA-CR-50998, June 1978.

[Par] M.C. Parsons, A.E. Bryson and W.E. Hoffman, "Long-Range Energy-State Maneuvers for Minimum Time to Specified Terminal Conditions", AIAA Paper No. 73-229, 11th Aerospace Science Meeting, Jan. 1973.

[Pek-1] N.J. Pekelsma and R.V. Denton, "Pilot Oriented Aids for Helicopter Automatic Nap-of-the-Earth Flight", *AHS Proceedings*

National Specialists' Meeting in Rotorcraft Flight Controls and Avionics, Oct. 1987.

[Pek-2] N.J. Pekelsma, "Optimal Guidance with Obstacle Avoidance for Nap-of-the-Earth flight", NASA-CR-177515, TAU Corporation, Dec. 1988.

[Pla] G. Plateau et al., "Algorithm PR2 88 for the Parallel Resolution of the 0-1 Multiknapsack Problem", *PR INRIA*, 1988.

[Rod] G. Rodgers and P. Pardalos, "Parallel Branch and Bound Algorithms for Quadratic 0-1 Programming", RR CS 88-17, Dept. of Computer Science, Pennsylvania State University, 1988.

[Rou-1] C. Roucairol, "Parallel Computing in Combinational Optimization", *Proc. of Numerical Methods for Parallel Vector Computers*, 1989.

[Rou-2] C. Roucairol, "Experiments with Parallel Algorithms for the Asymmetric Salesman Problem", *EURO VIII*, Lisboa, Portugal, 1986.

[Rou-3] C. Roucairol, "A Parallel Branch and Bound Algorithm for the Quadratic Assignment Problems", *Disc. Appl. Math.*, Vol. 18, 1984, pp. 211-225.

[Ryt] W. Rytter, "Note on Efficient Parallel Computations for Some Dynamic Programming Problems", *Theoretical Computer Science*, Vol. 59, 1988, pp. 297-307.

[Sch-1] F.H. Schmitz, "Optimal Take-Off Trajectories of a Heavily Loaded helicopter", *Journal of Aircraft*, Vol. 8, Sept. 1971.

[Sch-2] F.H. Schmitz and C.R. Vause, "A Simple, Near-Optimal Take-Off Control Policy for a Heavily Loaded Helicopter Operating from a Restricted Area", AIAA Paper No. 74-812, Aug. 1974.

[Sla-1] G. L. Slater and H. Erzberger, "Optimal Short-Range Trajectories for Helicopters", *J. of Guidance, Control, and Dynamics*, Vol. 7, No. 4, July-August 1984.

[Sla-2] G. L. Slater , "Determination of Maneuvering Flight Paths for Rotary Wing Aircraft", NAG2-175, Final Report, Feb., 1986.

[Sla-3] G.L. Slater and K. Hu, "Parallel Dynamic Programing for On-line Flight Path Optimization", AIAA Paper No. 89-3615, Proc. of AIAA Guidance and Control Meeting, Boston, MA, August 1989.

[Sla-4] G.L. Slater and M. Stoughton, "On-line Determination of Optimal Paths for Helicopters", *40th American Helicopter Society (AHS) Annual Forum*, D.C., 1984.

[Sla-5] G. L. Slater, "Guidance on Maneuvering Flight Paths for Rotary Wing Aircraft", AIAA Paper No. 87P-2406, Guidance and Control Meeting, Monterey, CA, August, 1987.

[Swe] H.N. Swenson and G.H. Hardy, "Simulation Evaluation of Helicopter Terrain Following/Terrain Avoidance Concepts", AIAA-88-3924-CP, Oct. 1988.

[TAU] "Terrain Following/Terrain Avoidance Algorithm Study", TAU Corporation, May 1985.

[Val] L. Valiant et al., "Fast Parallel Computation of Polynomials Using few Processors", *SIAM J. Comput.*, Vol. 12, No. 2, 1983, pp. 641-644.

[Var] P.J. Varman and I.V. Ramakrishnan, "Dynamic Programming and Transitive Closure on Linear Pipelines", *Proc. of 1984 Int'l Conf. on Parallel Processing*, pp. 359-364.

[Wah] B. W. Wah, G.-J. Li and C. F. Yu, "Multiprocessing of Combinatorial Search Problems", *IEEE Computer Magazine*, June, 1985.

[Wen] M.J. Wendl, J.E. Wall, and G.D. Young, "Advanced Automatic Terrain Following/Terrain Avoidance Control Concept Study", *1982 NAECON Conference Proceedings*.

[Whi] D. White, *Dynamic Programming*, Oliver & Boyd, Edinburgh, 1969.

[Wit] D.G. Mitchell, R.H. Hoh, A. Jr. Atencio, "Classification of Response-Types for Single-Pilot NOE Helicopter Combat Tasks", *43th AHS Annual Forum*, May 1987.

Appendix A The Knapsack Problem

The knapsack problem can be stated as follows. A knapsack and n types of objects are given. Each object of type i has a weight w_i and the knapsack has a capacity M. If an object of type i is placed into the knapsack, and then a profit of p_i is earned. The values of w_i, p_i, and M are all positive integers. The objective is to obtain a filling of the knapsack that maximizes the total profit earned. Mathematically, the knapsack problem can be formulated as

$$\text{maximize}_{x_i} \quad \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i \le M$$

$$x_i \ge 0 \text{ and integer}, i=1,\cdots,n$$

Let f(k,g) be the maximal value of the objective function using only the first k ($1 \le k \le n$) items with capacity limitation g ($0 \le g \le M$). This is

$$f(k,g) = \max_{x_i} \left\{ \sum_{i=1}^{k} p_i x_i \,\middle|\, \sum_{i=1}^{k} w_i x_i \le g \right\}$$

Reformulate it in recursive equation

$$f(k,g) = \max \{ f(k-1,g), p_k x_k + f(k,g-w_k x_k) \}$$

the above equation can be solved for an optimal value f(n,M) using the initial conditions f(0,g)=0, f(k,0)=0, and $f(k,y)=-\infty$ for $1 \le k \le n$, $0 \le g \le M$,

and $y<0$.

If additional conditions $x_i=0,1$ for $i=1,\cdots,n$ are assumed, the knapsack problem is called the 0-1 knapsack problem.

Appendix B The Quadratic Assignment Problem

The quadratic assignment problem can be thought of as a location problem. There is a set of n location in which m plants must be constructed, where m<n. The unit cost of shipping from location i to location j is c_{ij}. Also, the volume required to be shipped from plant k to plant l is d_{kl}. Let

$$x_{ik} = \begin{cases} 1 & \text{if plant k is assigned to location i} \\ 0 & \text{otherwise} \end{cases} \qquad \text{(B1)}$$

Then

$$\sum_{i=1}^{n} x_{ik} = 1 \qquad \text{for } k = 1,\cdots,m \qquad \text{(B2)}$$

since each plant must have exactly one location. Also,

$$\sum_{k=1}^{m} x_{ik} \leq 1 \qquad \text{for } i = 1,\cdots,n \qquad \text{(B3)}$$

since each location can have at most one plant. If plant k is assigned to location i and plant l to location j, the shipping cost is $c_{ij}d_{kl}+c_{ji}d_{lk}$. Therefore, the total shipping cost is

$$\sum_{\substack{i=1 \\ j=i}}^{m} \sum_{\substack{k=1 \\ l=1}}^{m} c_{ij}d_{kl}x_{ik}x_{jl} \qquad \text{for } i\neq j \text{ and } k\neq l \qquad \text{(B4)}$$

Thus the quadratic assignment problem is defined as to minimize (B4) subject to the constraints (B1)-(B3).