MARSHALL GRANT
IN-60-CR
131805

COMPUTER ARCHITECTURE FOR EFFICIENT ALGORITHMIC EXECUTIONS IN REAL-TIME SYSTEMS: NEW TECHNOLOGY FOR AVIONICS SYSTEMS AND ADVANCED SPACE VEHICLES

82 P.

Prepared by

Chester C. Carroll
Cudworth Professor of Computer Architecture
Department of Electrical Engineering

and

John N. Youngblood
Professor
Department of Mechanical Engineering

and

Aindam Saha
Graduate Research Assistant

The College of Engineering
The University of Alabama
Tuscaloosa, Alabama

Prepared for

The National Aeronautics and Space Administration

Grant Number NAG8-093

Computer Architecture Laboratory
The Bureau of Engineering Research
The University of Alabama
December 1987

BER Report No. 410-17

The University of Alabama
College of Engineering
Bureau of Engineering Research
P.O. Box 1968
University, Alabama 35486
Telephone: (205) 348-1591

THE UNIVERSITY OF ALABAMA
COLLEGE OF ENGINEERING

The College of Engineering at The University of Alabama has an undergraduate enrollment of more than 2,300 students and a graduate enrollment exceeding 180. There are approximately 100 faculty members, a significant number of whom conduct research in addition to teaching.

Research is an integral part of the educational program, and research interests of the faculty parallel academic specialities. A wide variety of projects are included in the overall research effort of the College, and these projects form a solid base for the graduate program which offers fourteen different master's and five different doctor of philosophy degrees.

Other organizations on the University campus that contribute to particular research needs of the College of Engineering are the Charles L. Seebeck Computer Center, Geological Survey of Alabama, Marine Environmental Sciences Consortium, Mineral Resources Institute—State Mine Experiment Station, Mineral Resources Research Institute, Natural Resources Center, School of Mines and Energy Development, Tuscaloosa Metallurgy Research Center of the U.S. Bureau of Mines, and the Research Grants Committee.

This University community provides opportunities for interdisciplinary work in pursuit of the basic goals of teaching, research, and public service.

BUREAU OF ENGINEERING RESEARCH

The Bureau of Engineering Research (BER) is an integral part of the College of Engineering of The University of Alabama. The primary functions of the BER include: 1) identifying sources of funds and other outside support bases to encourage and promote the research and educational activities within the College of Engineering; 2) organizing and promoting the research interests and accomplishments of the engineering faculty and students; 3) assisting in the preparation, coordination, and execution of proposals, including research, equipment, and instructional proposals; 4) providing engineering faculty, students, and staff with services such as graphics and audiovisual support and typing and editing of proposals and scholarly works; 5) promoting faculty and staff development through travel and seed project support, incentive stipends, and publicity related to engineering faculty, students, and programs; 6) developing innovative methods by which the College of Engineering can increase its effectiveness in providing high quality educational opportunities for those with whom it has contact; and 7) providing a source of timely and accurate data that reflect the variety and depth of contributions made by the faculty, students, and staff of the College of Engineering to the overall success of the University in meeting its mission.

Through these activities, the BER serves as a unit dedicated to assisting the College of Engineering faculty by providing significant and quality service activities.

Grant Number NAG8-093

COMPUTER ARCHITECTURE FOR EFFICIENT ALGORITHMIC EXECUTIONS IN REAL-TIME SYSTEMS: NEW TECHNOLOGY FOR AVIONICS SYSTEMS AND ADVANCED SPACE VEHICLES

by

Chester C. Carroll
Cudworth Professor of Computer Architecture

John N. Youngblood
Professor of Mechanical Engineering

and

Aindam Saha
Graduate Research Assistant

Prepared for

The National Aeronautics and Space Administration

Computer Architecture Laboratory
Bureau of Engineering Research
The University of Alabama
December 1987

BER Report No. 410-17

TABLE OF CONTENTS

LIST OF FIGURES

Figure

Chapter 1
INTRODUCTION

Improvements and advances in the development of computer architecture, both hardware and software structures, now provide innovative technology for the recasting of traditional sequential solutions into high-performance, low-cost, parallel systems to increase system performance. New processes and methodologies influence the implementation of real-time systems like guidance, control, avionics, robotics, and so on. The increasing demand of faster, real-time computation speed can be met with parallel path approaches at the algorithmic, as well as, hardware levels.

This report is the result of research conducted in development of specialized computer architecture for the algorithmic execution of a avionics system, guidance and control problem in real time. The objective of this research is to enhance vehicle guidance resulting from optimal guidance strategies by incorporating high-speed parallel processing. This report presents a comprehensive treatment of both the hardware and software structures of a customized computer which performs real-time computation of guidance commands with updated estimates of target motion and time-to-go.

Optimal control strategies are available for use in many guidance problems. In this research, the performance index for optimal guidance is chosen as a quadratic function of terminal miss and control action costs. A set of coupled, first order differential equations are solved to compute the optimal commanded acceleration of the advanced space vehicle.

To exploit a high degree of concurrency, the sequential algorithm is restructured. A parallel, multi-step, predictor corrector numerical integration technique is employed to solve the set of differential equations. A subsequent effort is devoted to segmenting or decomposing the algorithm into parallel and concurrent processes. Evaluation of derivatives and the integration of state variables are partitioned as distinct tasks. A task graph is constructed by considering the sequence in which the tasks are to be executed, satisfying all the precedence constraints.

An important aspect of this research is the development of an optimum, real-time allocation algorithm which maps the algorithmic tasks onto the processing elements. This allocation is based on the critical path analysis, a widely used technique in graph theory and operations research. For the particular task graph considered, an optimal allocation has been obtained with 29 processing elements. This enables the execution of the graph in the minimum possible time, as dictated by the precedence constraints of the graph and availability of resources.

The final stage is the design and development of the hardware structures suitable for the efficient execution of the allocated task graph. The system is data-driven, i.e., when the necessary operands for a task arrive at a particular processing element, the task is immediately executed. The basic system architecture consists of two star-shaped clusters , each consisting of 64 processing elements. A high-speed, buffered, crossbar, delta network allows parallel communication between pairs of processing elements within a cluster.

The processing element is designed for rapid execution of the allocated tasks. It contains local storage for both instructions and operands and extensive fault tolerance capabilities. Fault tolerance is a key feature of the overall architecture.

The remaining chapters of this report consider the various aspects of the research in complete detail. In the second chapter, the guidance problem is completely examined and mathematically defined. Chapter 3 deals with the restructuring of the sequential algorithm. Parallel numerical integration techniques, task definitions, and allocation algorithms are discussed in the third chapter. In Chapter 4 the parallel implementation is analytically verified and the experimental results are presented. Chapter 5 discusses the design of the data-driven computer architecture, customized for the execution of the particular algorithm. Some conclusions and recommendations are made in the last chapter.

Chapter 2
THE GUIDANCE AND CONTROL PROBLEM

2.1 BACKGROUND AND OBJECTIVE

The objective of this research is to investigate the enhancement of vehicle guidance resulting from the incorporation of optimal guidance strategies made possible by high speed parallel processing in guidance computation. The critical objective of this study is the determination of realistic cycle periods for repetitive, real-time computation of guidance commands with updated estimates of target motion and time-to-go.

For some time maneuvering vehicles have employed some form of proportional guidance, which is optimal, or near optimal in many engagements. In other conditions, its performance may be acceptable but less than perfect. Due mainly to recent advancements in microprocessor technology, more sophisticated techniques of advanced estimation and control theory may be implemented in a relatively small and inexpensive avionics package.

A number of studies have been directed toward the application of optimal control theory and estimation to a related guidance area [1-10]. Simulation results have indicated improved performance subject to the suitability of the performance criteria, the critical estimates of time-to-go and target acceleration, and the sensitivity of the guidance law to the assumed missile dynamics.

In at least one case, optimal guidance strategies were used in a highly accurate, nonlinear, six degree-of-freedom simulation of a tactical missile [5-8]. Guidance gains were computed as a function of time-to-go and constant target acceleration for a second-order, linear model system and used in the nonlinear simulation with various estimate procedures for time-to-go and target acceleration. The results were promising, but it was difficult to accurately assess the contribution of model error, estimation error, or nonjudicious performance indices to the miss distance.

With the prospect of extremely rapid computation of optimal guidance algorithms with specially designed computer architectures and parallel processing, as well as improved estimates of target acceleration, there is the prospect of solving the equations for guidance gains repetitively during the course of the intercept using more accurate dynamic models with adjustable parameter values and fresh estimates of target motion.

2.2 DEVELOPMENT

This section examines minimum control cost, minimum terminal miss guidance for the intercept of a moving target by a vehicle with inherent airframe and control system dynamic properties. Particular attention is given to the idealized problem of zero terminal miss, wherein the control gains are given in terms of the state transition of the uncoupled airframe dynamics. This approach separates the kinematic portion of the intercept dynamics, which is common for all intercept problems, from the kinetic portion of the vehicle dynamics. The particular problem structure makes the results applicable to a variety of engagement problems in which the vehicle airframe can be represented by a linear model.

The resulting control law has a term related to the intercept kinematics, which is recognizable as the familiar generalized proportional navigation term. A second term in the control law is a linear function of the vehicle airframe state and represents the guidance compensation due to finite airframe dynamics. The final term in the guidance law is related to target motion, providing an effective control in cases where target motion can be measured or predicted accurately.

The formulation structures the guidance problem for separation of the intercept kinematics from the dynamics of the vehicle. The motion of the target is accepted as an uncontrollable input to the problem; however, the kinetic state equation can be augmented with a target model if available.

2.2.1 Kinematics

Let the vector position and velocity of the target (T) and controlled vehicle (I) be represented in an inertial frame by y and v, as illustrated in Figure 1.

$$\begin{aligned} \dot{y}_I &= v_I & \qquad \dot{y}_T &= v_T \\ \dot{v}_I &= a_I & \qquad \dot{v}_T &= a_T \end{aligned} \tag{1}$$

Defining the relative position and velocity of the target with respect to the missile yields

$$\begin{aligned} \dot{y} &= v \\ \dot{v} &= a_T - a_I \end{aligned} \tag{2}$$

Letting the vector x represent the kinematic state of the intercept,

$$x = \begin{bmatrix} y \\ v \end{bmatrix} \tag{3}$$

then

$$\dot{x} = Ax + B(a_I - a_T)$$

where

$$A = \begin{bmatrix} 0 & I \\ 0 & 0 \end{bmatrix} \qquad B = - \begin{bmatrix} 0 \\ I \end{bmatrix}$$

In (3) the identity and null submatrices reflect the dimensionality of the problem.

2.2.2 Kinetics

The airframe/control response state is designated as z and satisfies the linear equation

$$\dot{z} = Dx + Ez + Fu \tag{4}$$

subject to the airframe control u (thrust, control surface deflection, etc.). The dimensions and components of the coefficient matrices in (4) are vehicle-dependent and provide the generality in the problem.

The intercept kinematics are coupled to the airframe dynamics by

$$a_I = Gz + Hu \tag{5}$$

Thus any linearized airframe describable by (4) and (5) is subject to the analysis.

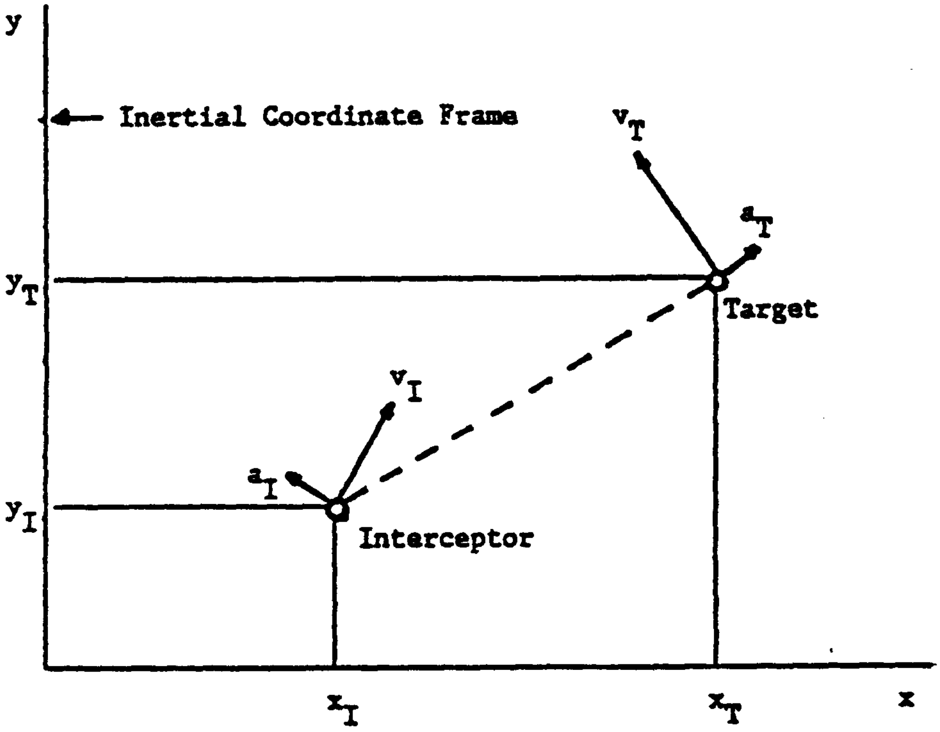

Fig 2.1 Inertial Frame of an interceptor and a target

2.2.3 Optimal Guidance

The conventional guidance performance index for targeting vehicles is of the form

$$J = 0.5x(t_f)^T S_f x(t_f) + \int_{t_o}^{t_f} (0.5u^T Ru + \Gamma)dt, \tag{6}$$

where S_f, R, and Γ weigh the costs associated with terminal miss, control cost and time, respectively. In those cases where the terminal miss is the significant cost, it is logical to constrain the final position $y(t_f)$ to zero and to develop the corresponding control law under this condition. Equation (6) may be replaced by

$$\int_{t_o}^{t_f} (0.5u^T Ru + \Gamma)dt \tag{7}$$

and

$$Tx(t_f) = 0$$

where

$$T = \begin{bmatrix} I & 0 \end{bmatrix}$$

The equations (3-7) are collected below.

Minimum Miss

$$\dot{x} = Ax + BGz + BHu - Ba_T$$

$$\dot{z} = Dx + Ez + Fu$$

$$J = 0.5x(t_f)^T S_f x(t_f) + \int_{t_o}^{t_f} (0.5u^T \ Ru + \Gamma)dt \tag{8}$$

Zero Miss

$$\dot{x} = Ax + BGz + BHu - Ba_T$$

$$\dot{z} = Dx + Ez + Fu$$

$$Tx(t_f) = 0$$

$$J = \int_{t_o}^{t_f} (0.5u^TRu + \Gamma)dt \tag{9}$$

2.3 MINIMUM TERMINAL MISS

If the performance index in (8) is augmented in the usual manner, the Hamiltonian for fixed terminal time is

$$H = 0.5u^T Ru + \lambda^T[Ax+BGz+BHu-Ba_T] + \mu^T[Dx+Ez+Fu] \tag{10}$$

The resulting boundary value problem is

$$\begin{aligned} \dot{x} &= Ax + BGz + BHu - Ba_T & x(t_o) &= x_o \\ \dot{z} &= Dx + Ez + Fu & z(t_o) &= z_o \\ \dot{\lambda} &= -A^T\lambda - D^T\mu & \lambda(t_f) &= S_fx(t_f) \\ \dot{\mu} &= -G^TB^T\lambda - E^T\mu & \mu(t_f) &= 0 \\ u &= -R^{-1}(H^TB^T\lambda + F^T\mu) \end{aligned} \tag{11}$$

The computation of the control gains is achieved via the inverse formulation

$$x = Q_1\lambda + Q_2\mu + Q_3$$

$$z = Q_4\lambda + Q_5\mu + Q_6$$

leading to the equations

$$\begin{array}{ll}
\dot{Q}_1 = AQ_1 + Q_1A^T + Q_2G^TB^T + BGQ_2^T - BHR^{-1}H^TB^T & Q_1(t_f) = S_f^{-1} \\
\dot{Q}_2 = AQ_2 + Q_2E^T + Q_1D^T + BGQ_5 - BHR^{-1}F^T & Q_2(t_f) = 0 \\
\dot{Q}_3 = AQ_3 + BGQ_6 - Ba_T & Q_3(t_f) = 0 \\
\dot{Q}_4 = Q_2^T & Q_4(t_f) = 0 \\
\dot{Q}_5 = EQ_5 + Q_5E^T + DQ_2 + Q_2^TD^T - FR^{-1}F^T & Q_5(t_f) = S_f^{*-1} \\
\dot{Q}_6 = EQ_6 + DQ_3 & Q_6(T_f) = 0
\end{array} \qquad (12)$$

The nonsingular diagonal matrix S_f^* is used for the computation of the inverse problem and its elements are set to zero in the solution for the gains.

The resulting control vector u is

$$u = K_1x + K_2z + K_3$$

where

$$\begin{aligned}
K_1 &= -R^{-1}(H^TB^TP_1 + F^TP_2^T) \\
K_2 &= -R^{-1}(H^TB^TP_2 + F^TP_5) \\
K_3 &= -R^{-1}(H^TB^TP_3 + F^TP_6)
\end{aligned} \qquad (13)$$

and

$$\begin{aligned}
P_1 &= [Q_1 - Q_2Q_5^{-1}Q_2^T]^{-1} \\
P_2 &= -[Q_1 - Q_2Q_5^{-1}Q_2^T]^{-1} Q_2Q_5^{-1} \\
P_5 &= [Q_5 - Q_2^TQ_1^{-1}Q_2]^{-1} \\
P_3 &= -P_1Q_3 - P_2Q_6 \\
P_6 &= -P_2^TQ_3 - P_5Q_6
\end{aligned} \qquad (14)$$

The particular case of interest is that in which the kinetics of the vehicle are independent of the intercept position and velocity, i.e.,

$$D = 0$$

In this case, the equations (11) are integrable and yield

$$
\begin{aligned}
Q_5 &= \int_t^{t_f} \Phi_E(t-\tau)FR^{-1}\, F^T\Phi_E{}^T(t-\tau)\; d\tau + \Phi_E(t-t_f)S_f{}^{*-1}\Phi_E{}^T(t-t_f) \\
Q_2 &= \int_t^{t_f} \Phi_A(t-\tau)\; [HR^{-1}F^T - GQ_5(\tau)]\; \Phi_E{}^T(t-\tau)\; d\tau \qquad (15) \\
Q_1 &= \int_t^{t_f} \Phi_A(t-\tau)[BHR^{-1}H^TB^T - BGQ_2{}^T(\tau) - Q_2G^TB^T]\Phi_A{}^T(t-\tau)d\tau \\
&\qquad + \Phi_A(t-t_f)S_f{}^{-1}\Phi_A{}^T(t-t_f) \\
Q_6 &= 0 \\
Q_3 &= -\int_t^{t_f} \Phi_A(t-\tau)\; Ba_T(\tau)d\tau
\end{aligned}
$$

where

$$\Phi_A = \begin{bmatrix} I & tI \\ 0 & I \end{bmatrix}$$

and

$$\dot{\Phi}_E = E\Phi_E \quad , \quad \Phi_E(0) = I$$

The elements of $S_f{}^*$ are set to zero in the resulting expressions for the elements of Q.

The minimum terminal miss control law is determined by (13) -(15). The case where the terminal miss weighting is large may be solved by constraining the terminal position, as is done in the next section.

2.4 ZERO TERMINAL MISS

The zero terminal miss and minimum control cost formulated in an earlier section and outlined in (9) is solved here. This problem is also treated in Reference 4. The augmented performance index for the fixed-time problem is

$$J = \nu^T T x(t_f) + \int_t^{t_f} [0.5u^T Ru + \lambda^T(Ax+BGz+BHu-Ba_T) + \mu^T(Dx+Ez+Fu)]dt$$

yields the boundary value problem described by

$$\begin{aligned}
\dot{x} &= Ax + BGz + BHu - Ba_T & x(t_o) &= x_o \\
\dot{z} &= Dx + Ez + Fu & z(t_o) &= z_o \\
\dot{\lambda} &= -A^T\lambda - D^T\mu & \lambda(t_f) &= T^T\nu \qquad (16) \\
\dot{\mu} &= -G^T B^T\lambda - E^T\mu & \mu(t_f) &= 0 \\
u &= -R^{-1}(H^T B^T\lambda + F^T\mu)
\end{aligned}$$

Selecting

$$\begin{aligned} \lambda &= S_1 x + S_2 z + S_3 \nu + S_4 \\ \mu &= S_5 x + S_6 z + S_7 \nu + S_8 \end{aligned}$$

(16) becomes

$$\begin{aligned} &\dot{S}_1 + A^T S_1 + S_1 A_2 + D^T S_5 + S_2 D_2 = 0 &\qquad S_1(t_f) = 0 \\ &\dot{S}_2 + A^T S_2 + S_2 E_2 + D^T S_6 + S_1 B_2 = 0 &\qquad S_2(T_F) = 0 \\ &\dot{S}_3 + A^T S_3 + D^T S_7 + S_1 C_2 + S_2 F_2 = 0 &\qquad S_3(t_f) = T^T \\ &\dot{S}_4 + A^T S_4 + S_2 H_2 + S_1 G_2 + D^T S_8 = 0 &\qquad S_4(t_f) = 0 \\ &\dot{S}_5 + E^T S_5 + S_5 A_2 + G^T B^T S_1 + S_6 D_2 = 0 &\qquad S_5(t_f) = 0 \\ &\dot{S}_6 + E^T S_6 + S_6 E_2 + G^T B^T S_2 + S_5 B_2 = 0 &\qquad S_6(t_f) = 0 \\ &\dot{S}_7 + E^T S_7 + G^T B^T S_3 + S_5 C_2 + S_6 F_2 = 0 &\qquad S_7(t_f) = 0 \\ &\dot{S}_8 + E^T S_8 + G^T B^T S_4 + S_5 G_2 + S_6 H_2 = 0 &\qquad S_8(t_f) = 0 \end{aligned} \qquad (17)$$

where

$$\begin{aligned} A_2 &= A - BHR^{-1}(H^T B^T S_1 + F^T S_5) \\ B_2 &= BG - BHR^{-1}(H^T B^T S_2 + F^T S_6) \\ C_2 &= -BHR^{-1}(H^T B^T S_3 + F^T S_7) \\ D_2 &= D - FR^{-1}(H^T B^T S_1 + F^T S_5) \\ E_2 &= E - FR^{-1}(H^T B^T S_2 + F^T S_6) \\ F_2 &= -FR^{-1}(H^T B^T S_3 + F^T S_7) \\ G_2 &= -BHR^{-1}(H^T B^T S_4 + F^T S_8) - Ba_T \\ H_2 &= -FR^{-1}(H^T B^T S_4 + F^T S_8) \end{aligned} \qquad (18)$$

Inspection of (17) and (18) shows that all unknown matrices except S_3 and S_7 are null. Leaving

$$\begin{aligned} &\dot{S}_3 + A^T S_3 + D^T S_7 = 0 &\qquad S_3(t_f) = T^T \\ &\dot{S}_7 + E^T S_7 + G^T B^T S_3 = 0 &\qquad S_7(t_f) = 0 \\ &u = -R^{-1}(H^T B^T S_3 + F^T S_7)\nu & \end{aligned} \qquad (19)$$

The invariance of the terminal manifold

$$\Psi(t) = S_9x + S_{10}z + S_{11}\nu + S_{12}$$

implies

$$\begin{array}{ll} \dot{S}_9 + S_9A + S_{10}D = 0 & S_9(t_f) = T \\ \dot{S}_{10} + S_{10}E + S_9BG = 0 & S_{10}(t_f) = 0 \\ \dot{S}_{11} -(S_9BH + S_{10}F)R^{-1}(H^TB^TS_3 + F^TS_7) = 0 & S_{11}(f_f) = 0 \\ \dot{S}_{12} - S_9Ba_T = 0 & S_{12}(t_f) = 0 \end{array} \qquad (20)$$

where

$$\nu = -S_{11}{}^{-1}[S_9x + S_{10}z + S_{12}]$$

Therefore the control law for zero miss is

$$u = K_1x + K_2z + K_3$$

where the navigation, vehicle airframe, and target position components of the position and velocity control are evident.

$$K_1 = R^{-1}(H^TB^TS_3 + F^TS_7)S_{11}{}^{-1}S_9$$

$$K_2 = R^{-1}(H^TB^TS_3 + F^TS_7)S_{11}{}^{-1}S_{10}$$

$$K_3 = R^{-1}(H^TB^TS_3 + F^TS_7)S_{11}{}^{-1}S_{12}$$

For the uncoupled case where D is null, the equations (19) and (20) are integrated for computation of the optimal gains. Integration yields

$$S_3 = S_9{}^T = \begin{bmatrix} I \\ \\ (t_f - t)^I \end{bmatrix}$$

$$S_{10} = S_7{}^T = \int_t^{t_f} S_9(\tau)BG\Phi_E{}^*(t-\tau)\ d\tau$$

$$S_{11}(t) = - \int_{t}^{t_f} [S_{10}(\tau)F+S_9(\tau)BH]R^{-1}[S_{10}(\tau)BH]^T \, d\tau \qquad (21)$$

$$S_{12}(t) = - \int_{t}^{t_f} S_9(\tau)Ba_T(\tau) \, d\tau$$

where

$$\dot{\Phi}_E^* = -\Phi_E^*E \qquad \Phi^*(0) = I$$

The target motion term of the control for the uncoupled case can be written

$$K3 = K_o \int_{t}^{t_f} (t_f - \tau) \, a_T(\tau) \, d\tau$$

where

$$K_o = R^{-1}(H^TB^TS_3 + F^TS_7)S_{11}^{-1}$$

Also

$$K_3 = K_o \, (t_f - t) \, [V_A(t) - V_T(t)]$$

where $V_T(t)$ represents present target velocity and $V_A(t)$ is the average relative velocity on the terminal interval (t, t_f). The need for effective estimation of target motion is recognized.

2.5 IMPLEMENTATION

To develop an implementation of a real-time optimal guidance and control processor which is usable in an adaptive mode by continuously updating the coefficients, a second order, variable parameter vehicle model was chosen. This model represents only a single dimension of the problem, but is highly representative of an actual system. Furthermore, the control computation for the other plane is inherently a totally parallel process. A minimum terminal-miss, minimum control-action optimal strategy was chosen.

To implement the algorithm in a realistic planer problem the following scalar dynamic equations are chosen:

$$\dot{y} = v$$

$$\dot{v} = a_T - a_I$$

with the airframe model responding in accordance with

$$\ddot{a}_I + 2\zeta\omega_n\dot{a}_I + \omega_n^2 a_I = \omega_n^2 a_c$$

where a_c is the commanded acceleration specified by optimal guidance.

The performance index for optimization is chosen as

$$J = 0.5sy^2(t_f) + 0.5r\int_{t_o}^{t_f} a_c^2(t)\,dt$$

where s and r are scalar performance parameters weighing terminal miss and control action costs.

System parameter matrices corresponding to equation (8) are

$$A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$D = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \quad E = \begin{bmatrix} 0 & 1 \\ -\omega_n^2 & -2\zeta\omega_n^2 \end{bmatrix} \quad F = \begin{bmatrix} 0 \\ \omega_n^2 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 \end{bmatrix} \qquad H = 0$$

$$S_f = \begin{bmatrix} s & 0 \\ 0 & 0 \end{bmatrix} \qquad R = r$$

The optimal commanded acceleration is then given by:

$$a_c = - \omega_n^2/r[\ S_4y + S_7v + S_9a_I + S_{10}\dot{a}_I + S_{14}a_T\ ]$$

where the optimal gain coefficients satisfy

$$\dot{S}_1 - \alpha S_4^2 = 0 \qquad S_1(t_f) = s$$

$$\dot{S}_2 + S_1 - \alpha S_4S_7 = 0 \qquad S_2(t_f) = 0$$

$$\dot{S}_3 - S_2 - \beta S_4 - \alpha S_4S_9 = 0 \qquad S_3(t_f) = 0$$

$$\dot{S}_4 + S_3 - \gamma S_4 - \alpha S_4S_{10} = 0 \qquad S_4(t_f) = 0$$

$$\dot{S}_5 + 2S_2 - \alpha S_7^2 = 0 \qquad S_5(t_f) = 0$$

$$\dot{S}_6 + S_3 - S5 - \beta S_7 - \alpha S_7S_9 = 0 \qquad S_6(t_f) = 0$$

$$\dot{S}_7 + S_4 + S_6 - \gamma S_9 - \alpha S_7S_{10} = 0 \qquad S_7(t_f) = 0$$

$$\dot{S}_8 - 2S_6 - 2\beta S_9 - \alpha S_9^2 = 0 \qquad S_8(t_f) = 0$$

$$\dot{S}_9 + S8 - S_7 - \gamma S_9 - \beta S_{10} - \alpha S_9S_{10} = 0 \qquad S_9(t_f) = 0$$

$$\dot{S}_{10} + 2S_9 - 2\gamma S_{10} - \alpha S_{10}^2 = 0 \qquad S_{10}(t_f) = 0$$

$$\dot{S}_{11} - \alpha S_4S_{14} + S_2a_T = 0 \qquad S_{11}(t_f) = 0$$

$$\dot{S}_{12} + S_{11} - \alpha S_7S_{14} + S_2a_T = 0 \qquad S_{12}(t_f) = 0$$

$$\dot{S}_{13} - S_{12} - \beta S_{14} - \alpha S_9S_{14} + S_6a_T = 0 \qquad S_{13}(t_f) = 0$$

$$\dot{S}_{14} + S_{13} - \gamma S_{14} - \alpha S_{10}S_{14} + S_7a_T = 0 \qquad S_{14}(t_f) = 0$$

and

$$\alpha = \omega_n^4/r$$

$$\beta = \omega_n^2$$

$$\gamma = 2\zeta\omega_n$$

These equations, processed from t_f to 0, are the objective guidance equations for implementation of the real-time processor developed in the subsequent chapters.

Chapter 3
PARALLEL IMPLEMENTATION

In this chapter the restructuring of the sequential algorithm is discussed so that a real-time, efficient, and parallel implementation can be developed. The performance of the modified algorithm and the experimental results are discussed in Chapter 4. The steps of the restructured algorithm are outlined in Section 3.1. Parallel numerical integration techniques are reviewed in Section 3.2. Section 3.3 presents the concepts of a task graph. Section 3.4 deals with the allocation problem in general, and two allocation schemes in specific.

3.1 RESTRUCTURING

To calculate the commanded acceleration of the vehicle in one direction 14 ordinary differential equations must be solved in real-time. Speed and accuracy are the salient requirements. The computer architecture suitable for this parallel algorithm is of no less importance and this is the focus of Chapter 5. Here only the algorithmic implications are considered.

The key to parallel implementation is identifying as many operations as possible to be executed in parallel and removing dependencies wherever feasible. The overall problem is segmented into several tasks which are simultaneously executable. These asynchronously cooperating tasks will be executed on different processors if available.

The basic steps of the restructured algorithm are as follows :

1.) Remove the sequential bottlenecks of numerical integration of ordinary differential equations. In the past three decades several authors have come up with efficient and parallel numerical integration techniques.

2.) Segment the evaluation of derivatives and the integration of state variables into several distinct tasks.

3.) Construct a maximally parallel task system considering all the precedence constraints.

4.) Develop an efficient allocation algorithm to schedule the tasks to different processing elements in a multiprocessor environment.

3.2 PARALLEL NUMERICAL INTEGRATION TECHNIQUES

There has been some effort over the years to speed up the numerical integration of an ordinary differential equation (ODE). The old sequential techniques have been modified so that they are suitable in an environment with a plurality of processors. In this section some of these parallel numerical integration techniques are reviewed.

The parallel methods compute the solution of a set of n O.D.E.'s developed by

$$\dot{y}(t) = f(t,y(t)), \quad y(t_o) = y_o.$$

Most methods generate y_n, an approximation to $y(t_n)$ on a mesh $a = t_0<t_1<t_2<...<t_n = b$. These are called step-by-step difference methods. An r-step difference method is one which computes y_{n+1} using r earlier values y_n, $y_{n+1}, \ldots, y_{n-r+1}$. This numerical integration by finite differences is a sequential calculation. Lately, the question of using these formulas simultaneously on a set of arithmetic processors to increase the speed has been addressed by many authors.

3.2.1 Runge-Kutta (RK) Methods

The general form of an r-step RK method, the integration step leading from Y_n to Y_{n+1} consists of computing

$$K_1 = h_n f(t_n, Y_n)$$

$$K_i = h_n f(t_n + a_i h_n,\ Y_n + \sum_{j=1}^{i-1} b_{ij} K_j)$$

$$Y_{n+1} = Y_n + \sum_{i=1}^{r} R_i K_i$$

with appropriate values of a's, b's, and R's. A classical 4-step RK method is

$$K_1 = hf(t_n, Y_n)$$

$$K_2 = hf(t_n + h/2\ ,\ Y_n + (1/2)K_1)$$

$$K_3 = hf(t_n + h/2,\ Y_n + (1/2)K_2)$$

$$K_4 = hf(t_n + h,\ Y_n + K_3)$$

$$Y_{n+1} = Y_n + (1/6)\ (K_1 + 2K_2 + 2K_3 + K_4)$$

Miranker and Liniger [11] developed parallel Runga-Kutta formulas. In the parallel computation of a third-order approximation $y_1{}^3$, first-and second-order approximations, $y_0{}^1$ and $y_0{}^2$, respectively in addition to $y_0{}^3$ must be computed. As a consequence, the third-order parallel method gives a second-order parallel scheme as a by-product. The formulas of the parallel schemes have the structures:

first order:
RK1

$$K_1 = hf(t_n,\ Y,{}^1)$$

$$Y_n{}^1{}_{+1} = Y_n{}^1 + K_1$$

second order: RK2

$$K_1{}^2 - K_1 - h\ f(t_n,\ Y_n{}^1)$$

$$K_2 - h\ f(t_n + ah,\ Y_n{}^1 + bK_1{}^2)$$

$$Y_n{}^2{}_{+1} - R_1{}^2\ K_1{}^2 + R_2{}^2\ K_2$$

third order: RK3

$$K_1{}^3 - K_1$$

$$K_2{}^3 - K_2$$

$$K_3 - h\ f(t_n + ah\ ,\ Y_n{}^2 + bK_1{}^3 + cK_2{}^3)$$

$$Y_n{}^3{}_{+1} - R_1{}^3\ K_1{}^3 + R_2{}^3\ K_2{}^3 + R_3{}^3\ K_3$$

The parallel character of the above formulas insures that RKi is independent of RKj if and only if $i<j$, $i,j-1,2,3$. This means that if RK1 runs one step ahead of RK2, and RK2 runs one step ahead of RK3, then they can be executed simultaneously. Using Kopal's [12] values of the R's, the parallel third order RK formula is given by:

$$K^1{}_{n+2} - hf(t_{n+2},\ Y^1{}_{n+2})$$

$$Y^1{}_{n+3} - Y^1{}_{n+2} + K^1{}_{n+2}$$

$$K^2{}_{n+1} - hf(t_{n+1} + ah,\ Y^1{}_{n+1} + aK^1{}_{n+1})$$

$$Y^2{}_{n+2} - Y^2{}_{n+1} + (1-(1/2a))K^1{}_{n+1} + (1/2a)K^2{}_{n+1}$$

$$K^3{}_n - hf(t_n + a_1h,\ Y^2{}_n + (a_1-(1/6a))K^1{}_n + (1/6a)K^2{}_n)$$

$$Y^3{}_{n+1} - Y^3{}_n + ((2a_1-1)/2a)(K^1{}_n - K^2{}_n) + K^3{}_n.$$

where $a - 2(1-3a_1a_1)/(3(1-2a_1))$.

The above 3rd order RK formulas require 3 processors to compute the three functions in parallel. The main drawback of this scheme is that it is weakly stable and leads to an error that grows linearly with n.

3.2.2 Interpolation Method

Nievergelt [13] proposed a parallel form of a serial integration method in which the algorithm is divided into several subtasks which are computed independently. The idea is to divide the integration interval [a,b] into N equal subintervals $[t_{i-1}, t_i]$, t_0=a, t_N=b, i=1,2,3,...,N, to make a rough prediction yi^0 of the solution $y(t_i)$, to select a certain number M_i of values y^{ij}, j=1,...,M_i in the vicinity of yi^0 and finally to integrate the system with an accurate integration method M.

$$\dot{y}=f(t,y),\ y(t_0) = y_0, \quad t \in [t_0,t_1]$$
$$\dot{y}=f(t,y),\ y(t_i) = y_{ij}, \quad t \in [t_i,t_{i+1}],\ j=1...,M_i,\ i=1,...,N-1.$$

The integration interval [a,b] is covered with solution segments of equal length, (b-a)/N. The connection between these branches is made by interpolating the previous solution segment over the next interval to the right. The time of this computation can be represented by

T_{p1} = time for serial integration/N + time to predict yi^0 + interpolation time + bookkeeping time.

Interpolation can be done in parallel. If it is assumed that the time consuming part is the evaluation of f(t,y) and the other contributions to the total computation time are negligible, the speed up is 1/N. But to compare this method with serial integration from a to b using method M, the error introduced by method M is significant. This error depends on the problem, not on the method. For linear problems the error is bounded, but for nonlinear problems it may not be. Thus, the usefulness of this method is restricted to a specific class of problems, and depends on the choice of parameters like yi^0, M_i and method M.

3.2.3 Predictor Corrector (PC) Methods

One step methods do not make full use of the available information. It seems plausible that more accuracy can be obtained if the value of y_{n+1} is made to depend not only on y_n but also, on $y_{n-1}, \ldots,$ and f_{n-1}, $f_{n-2}, \ldots$. For this reason multi-step methods have become very popular. For high accuracy they usually require less computation than one-step methods.

A standard fourth-order serial predictor corrector given by Adams-Moulton is:

$$Y^{p}{}_{i+1} = Y^{c}{}_{i} + (h/24)(55\ f^{c}{}_{i} - 59f^{c}{}_{i-1} + 37f^{c}{}_{i-2} - 9f^{c}{}_{i-3})$$

$$Y^{c}{}_{i+1} = Y^{c}{}_{i} + (h/24)(9f^{p}{}_{i+1} + 19f^{c}{}_{i-1} - 5f^{c}{}_{i-2} + f^{c}{}_{i-3})$$

The following computation scheme, PECE, of the PC step to calculate y_{i+1} is:

1. Use the predictor equation to calculate and initial approximation to y_{i+1}.
2. Evaluate the derivative function $f^{p}{}_{i+1}$.
3. Use the corrector equation to calculate a better approximation to y_{i+1}.
4. Evaluate the derivative function $f^{c}{}_{i+1}$.
5. Check the termination rule. If it is not time to stop, increment i, set $y_{i+1} = y^{c}{}_{i+1}$ and return to step 1.

Let

T_f = total time taken by function evaluation done for one step

T_{PCE} = time taken to compute predictor (corrector) value for a single equation

then time taken by one step in serial predictor corrector is

$$T = 2(nT_{PCE} + T_f)$$

The serial method is schematized in Fig. 3.1

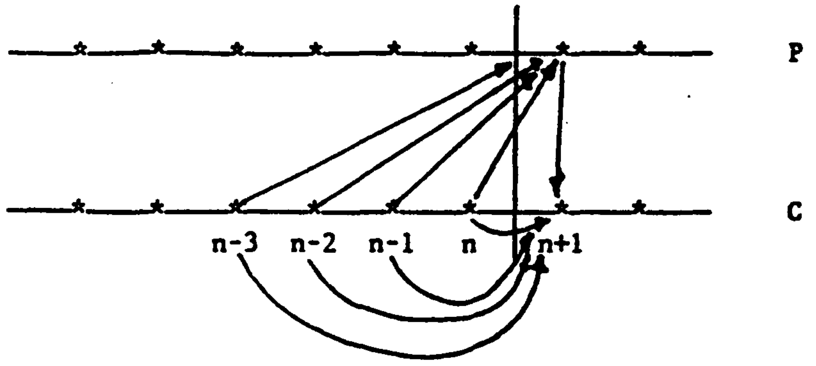

Fig. 3.1 Serial Predictor Corrector method

The upper line represents the progress of the computation at the mesh points for $y_n{}^P$ and $f_n{}^P$, and the lower for $y_n{}^C$ and $f_n{}^C$. The broken vertical line is the computation front. The calculations ahead of the front depend on information on both sides. This a characteristic of sequential calculation.

Miranker and Liniger developed formulas for the PC method in which the corrector does not depend serially upon the predictor, so that the predictor and corrector calculations can be performed simultaneously. The parallel predictor corrector (PPC) also operates in a PECE mode, and the calculation advances s steps at a time. There are 2s processors and each processor performs either a predictor or a corrector calculation.

A fourth order PPC is given by:

$$y_{i+1}{}^P = y_{i-1}{}^c + (h/3)(8f_i{}^P - 5f_{i-1}{}^c + 4f_{i-2}{}^c - f_{i-3}{}^c)$$

$$y_i{}^c = y_{i-1}{}^c + (h/24)(9f_i{}^P + 18f_{i-1}{}^c - 5f_{i-2}{}^c + f_{i-3}{}^c)$$

The method is schematized in Fig. 3.2.

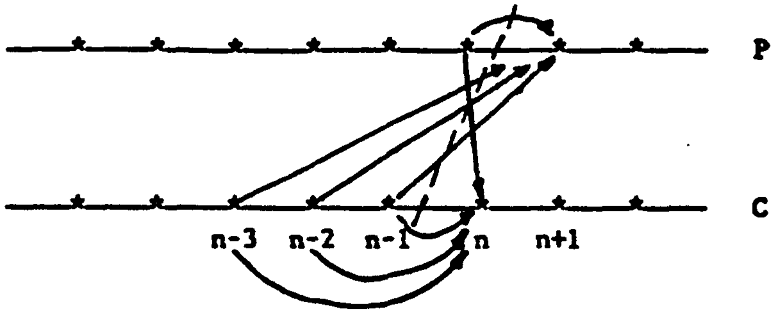

Fig. 3.2 Parallel predictor Corrector method

The computations at points ahead of the front depend only on information behind the front, a characteristic of parallel computation. The sequence of computation is divided and each of its two parts

$$y_{n+1}^{P}, f_{n-1}^{c}, \ldots\ldots\ldots \qquad \text{and} \quad y_n^{c}, f_n^{c} \ \ldots\ldots\ldots$$

may be simultaneously executed on separate processors.

As shown in [14], the parallel time for a single step of the fourth order PPC method is given by:

$$T = nT_{PCE} + t_f + 3nT_{DC} + 2T_s$$

where

T_{PCE} = T_f as defined before and
T_{DC} = time taken for data communication
T_s = time taken for synchronization

For 4 processors (s=2) the parallel PPC formulas are:

$$Y^{P}_{2n+2} = Y^{c}_{2n-2} + 4hf^{P}_{2n}$$

$$Y^{P}_{2n+1} = Y^{c}_{2n-2} + (3h/2)(f^{P}_{2n} + f^{P}_{2n-1})$$

$$Y^{c}_{2n} = Y^{c}_{2n-2} - (h/2)(3f^{P}_{2n} + 9f^{P}_{2n-1})$$

$$Y^{c}_{2n-1} = Y^{c}_{2n-3} + 2hf^{c}_{2n-2}.$$

Generally higher accuracy and fewer function evaluations of PC methods,as compared to RK methods, are obtained at the cost of increased complexity and possible numerical instability. The parallel RK methods do not inherit the stability of their serial counterparts. On the other hand, PPC methods are as stable as their serial formulas. This is proven by Katz, et al., [15].

3.3 GENERATION OF THE TASK GRAPH

A task is defined as a unit of computational activity specified in terms of the input variables that it requires, the output variables that it generates, and its execution time. The specific transformation that it imposes on the inputs to produce the output is not a part of the specification of the task. Thus, the task may be considered uninterpreted.

Let $J = (T_1, T_2, \ldots, T_n)$ be a set of tasks and <. an irreflexive partial order or procedure relation defined on J. Then $C = (J, <.)$ is called a task system. The precedence relation means that, if $T <. T'$, then T must complete execution before T' is started.

From this definition a graphical representation, called a task graph, is obtained for a task system. This consists of a directed graph whose vertices, or nodes, are the tasks J and which has an edge from T to T', if $T <. T'$ and there is no T'' such that $T <. T'' <. T'$. Thus, the set of edges in the task graph represents the smallest relation whose transitive closure is <. .

With each task T two events are associated, initiation and termination. An execution sequence of an n-task system $C = (J, <.)$ is any string $\delta = \alpha_1, \alpha_2, \ldots, \alpha_{2n}$ of task events satisfying the precedence relation (i.e., if $T<.T^{\prime}$, the termination event of T must occur prior to the initiation event of $T^{\prime}$) and consisting of exactly one initiation event and one termination event for each task. A task system that represents a sequential program has only one execution sequence; however, for other task systems there may be several.

To discuss determinant task systems, let the physical system on which task systems execute be represented by an ordered set of memory cells $M = (M_1, M_2, \ldots, M_m)$. With each task in a system C two, possibly overlapping, ordered subsets of M are associated,the domain D_T and the range R_T. When T is initiated, it reads the values stored in its domain cells, and when it terminates, it writes values in its range cells. Given an execution system w for a task system, the value sequence $V(M_i,\delta)$ is defined as the sequence of values written by terminating tasks in δ for which $M_i \in R_T$.

The intuitive concept of determinant task systems is more rigorously defined as follows :

A task system C is determinant, if for any given initial state P_0, $V(M_i,\delta)= V(M_i,\delta^{\prime})$, $i \in [1,m]$ for all execution sequences δ and $\delta^{\prime}$.

From this definition, it is clear that a task system representing a sequential program is determinant since there is only one execution sequence. Two task systems both consisting of the same tasks are said to be equivalent if they are determinant and produce the same value sequences for the same initial state. The goal is to convert the determinant task system represented by the sequential algorithm into an equivalent determinant task system which has more parallelism.

Given a task system C, tasks T and T' are noninterfering if $T<.T'$ or $T^1<\cdot T$. Task systems consisting of mutually noninterfering tasks are determinant [16]. With the above background in mind the task graph is generated. The exact details of this are given in Chapter 4.

3.4 ALLOCATION

Given a determinant task system in the form of a task graph and the execution time of each task, the next step is the assignment of the tasks to p processors. This is termed the allocation phase which is a part of the preprocessing stage.

The following parameters are available for allocation :

1) a set of tasks $J = (T_1, T_2, \ldots, T_n)$,

2) an irreflexive partial order <. on J,

3) a weighting function W from S to be positive integers, representing the execution time of each of the tasks, and

4) the number of processors p.

As many as p tasks can be executed in parallel at any time. If task T is first executed at time t using processor K, then it is executed only at times t, t+1,...,t+W(T)-1 using processor K each time. This is an example of non-preemptive allocation, where once a task is assigned to a processor it must be completed before any other task is assigned to the same processor. An additional requirement is that any task T', such that $T'<.T$, complete execution at time t' where $t' \leq t$.

A schedule is an assignment of tasks to processors that satisfies the above conditions and has length tmax, where tmax is the maximum time at which the termination events occur for all tasks. The allocation problem is the determination of an assignment that minimizes tmax with a minimum number of parallel processors. This type of problem has been studied extensively by a number of pioneering researchers [17]. It has been shown to be NP-complete [18] and can be considered intractable. When the number of processors, the task processing times and the precedence constraints are all arbitrary, the complexity of such an allocation problem becomes NP-hard in the strong sense. Hence, unless P equals NP, it is impossible to construct either a pseudopolynomial time allocation algorithm or a fully polynomial time approximation scheme [19].

In order to circumvent these difficulties, heuristic algorithms have been considered to be the most powerful tools. Indeed, the critical path (CP) method [22] and HLFET (highest levels first with estimated times) [20], which essentially is a sort of list scheduling method, are proposed.

Two different allocation schemes are discussed in this chapter. The first, proposed by Kasahara and Narita [21], is known as the CP/MISF (critical path/most immediate successors first) method, is an improved version of the CP-method. The second, a newly proposed algorithm based on the application of the branch and bound technique, is termed BBAS (branch and bound allocation scheme).

3.4.1 The CP/MISF Method

A critical path is defined as the path from the exit node to the entry node having the longest path length In mathematical terms

$$tcr = \max_{k} \sum_{i \in \theta_k} t_i$$

where θ_k represents the kth path from the exit node to the entry node.

tcr is equal to the minimum possible execution time for a plurality of parallel processors to process the tasks involved in a given task graph. The level l_i, defined for each task, serves as the basis for constructing the priority list. The level l_i of task i is defined to be the longest path length from the exit node to task i, or

$$l_i = \max_{k} \sum_{j \in \Pi_k} t_j$$

where Π_k stands for the kth path from the exit node to N_i.

The CP method is essentially the generalization of Hu's algorithm [22]. Since the priority order cannot be uniquely determined when there exists more than one task having the same level, the worst schedule may result depending upon the task chosen. In this method when two tasks have the same level, the task having the largest number of immediately successive tasks is assigned the higher priority.

The CP/MISF method consists of the following steps:

Step 1: Determine the level l_i for each task.

Step 2: Construct a priority list in the descending order of l_i and the number of immediately successive tasks.

Step 3: Renumber the tasks from 1 to n in the descending order of priority.

Step 4: Execute list scheduling on the basis of the priority list.

The problem of determining the level of each task involves the calculation of the longest path from the exit node to each node. In the case of a single exit node, a dummy exit node is added. Since all arcs are directed from the entry node toward the exit node, the longest path is measured in the direction opposite to the orientation of each arc. This problem can be solved in $O(n^2)$ by solving the Bellman's equations [23].

Let

$$a_{ij} = 0 \quad \text{if link } (j,i) \text{ exists}$$
$$= -\infty \quad \text{otherwise}$$

t_i = time for executing task i

l_i = level of task i

Then for n tasks,

$$l_n = t_n$$

$$l_j = \max_{k=j} \{l_k + a_{kj} + t_j\} \quad j = n-1,\ n-2, \ldots, 2, 1.$$

For each node j, j not equal to n, there must be some arc (k,j) in a longest path from n to j. Whatever the identity of k, it is certain that $l_j = l_k + a_{kj} + t_j$. This follows from the fact that the part of the path which extends to node k must be the longest path from n to k, if this were not so, the overall path to j would not be as long as possible. (This is the "Principle of Optimality"). But there are only finite choices of k, i.e, k =n, n-1,...,j+1, j-1,...2,1. Clearly k must be a node for which l_j is as large as possible. In fact the effect of the most immediate successive tasks can be incorporated in the same step by modifying l_j

$$l_j = l_j + imsucc_j/n$$

where $imsucc_j$ is the number of immediate successive tasks for task j and n is the total number of tasks.

Actual experimental results with respect to the specific problem are discussed later. It has been shown [21] that optimal solutions were obtained for about 67 percent of some 200 cases tested by the CP/MISF method. Approximate solutions with error less than 5 percent were obtained for 87 percent of the cases and those less than 10 percent for 98.5 percent of the cases.

3.4.2 The BBAS

Branch and bound implicit enumeration algorithms have emerged as the principal method for finding optimum solutions to discrete optimization problems. Kohler's [24] general representation can be used for the classification of the branch and bound technique. In the expression BB(B_p,S,E,L,U,RB) each parameter has the following significance:

Bp	:	branching rule
S	:	selection rule of next branching node
E	:	elimination rule
L	:	lower bound function
U	:	upper bound cost
RB	:	resource bounds

The proposed algorithm works by partitioning the set of schedules into smaller and smaller subsets, finding lower bounds on total execution times of each of the subsets, and using these bounds to guide further partitioning until a single schedule is obtained whose total execution time is less than or equal to the lower bounds of all the other subsets.

Branching Rule, B_p :

An allocation instant is defined as the time when one or more processors have just finished execution of the allocated task(s) and succeeding task(s) becomes executable. Since the task times are different, there is the possibility that the optimal schedule may not be obtained by simple list scheduling methods. At each stage of branching procedure, nodes should be generated to include the cases where a processor or processors become idle. To this end fictitious tasks which correspond to idle processors are introduced, as done in [21]. These idle tasks, together with ready tasks are allocated.

$$\begin{aligned} N_{idle} &= M_{av}\text{-}1 \quad \text{for } M_{av} = M \\ &= M_{av} \qquad \text{for } 1<M_{av}<M. \end{aligned}$$

where N_{idle} is the number of idle tasks, N_{ready} is the number of ready tasks, M_{av} is the number of available processors, and M is the total number of processors.

Then the number of nodes generated from each branching node is given by

$$N_{branch} = N_{alloc}\; C_{Mav}$$

where C is the number of combinations and

$$N_{alloc} = N_{ready} + N_{idle}.$$

The set of allocatable tasks is represented by A.

Selection Rule, S :

The selection rule is used to choose the next branching node from the set of currently active nodes. The rule used in the algorithm is least lower bound or LLB. The next branching node is chosen by calculating and comparing the lower bounds for all the active nodes at each branching instant.

Lower bound, L :

The lower bound, in our case, is simply the total execution time for the partial schedule represented by each node.

Upper bound cost, U :

When the solution of the original problem is known a priori, its value can be used as U. Otherwise, set U equal to ∞. The value U is updated whenever a smaller solution U' is obtained. The smaller the value of U at an early stage of the search process, the shorter is the search time, and a reasonable value of U is evaluated with the help of a heuristic algorithm.

Elimination Rule, E :

To eliminate some of the active nodes the following rule is employed. Whenever the lower bound $LB(n_i)/U^{/}$, the node n_i is eliminated.

Resource Bound, RB :

This is the allowable computing time limit and storage capacity limit.

At first glance the simplistic BBAS seems to have enormous time and space complexities, but the greatest advantage of this scheme is its inherent parallelism. The potential parallel paths in the control flow of this algorithm may be explicated and computed by multiple processes. In other words the loop is unfolded to let the multiple processes work on different iterations of the unfolded loop. This algorithm is in its rudimentary stage and will be further investigated later. Presently the authors are working on a possible parallel implementation of this new allocation scheme.

Chapter 4
VERIFICATION & EXPERIMENTAL RESULTS

The restructured parallel algorithm was verified to check its validity with respect to the actual problem in hand. The experiments were mainly performed to show the improved performance of the new parallel approach compared to the conventional sequential one. Results are shown corresponding to one iteration of the integration of all the 14 differential equations, listed in Chapter 2. Section 4.1 deals with the construction of the task graph. The implementation of the allocation algorithm is discussed in Section 4.2. The experimental results are analyzed in Section 4.3.

4.1 CONSTRUCTION OF TASK GRAPH

The fourth order, 2-processor parallel predictor corrector method, outlined in Chapter 3, is chosen for solving the differential equations. The basis of constructing the task graph lies in the definition of tasks and their appropriate precedence constraints. Basically there awe two types of operations, updating the dependent variables and calculating the functions. Each update of a dependent variable is defined to be a task. Hence, for the fourteen differential equations involving P_1, P_2,...P_{14} there are twenty-eight different tasks as follows, $(Pi)^P$ and $(Pi)^c$ for $i = 1...14$. Note that for a particular iteration level j, the corresponding tasks are $(Pi)_{j+1}{}^P$ and $(Pi)_j{}^c$ for $i = 1...14$.

Due to the highly coupled nature of the differential equations, there are dependencies between the various functions. It is noted that decoupling methods can improve the situation. The function values are evaluated from the updated dependent variables with each function evaluation task fragmented into smaller subtasks. These smaller tasks are defined in a manner such that each of them have some uniform execution time. In this way the task graph becomes more or less balanced and parallelism is optimally exploited.

Some sub-expressions, which are used a number of times, are identified. Each of these sub-expressions are defined as a separate task to prevent repetitive calculations. There are some other tasks required to calculate some constants.

All the tasks are listed in APPENDIX A. Each task is associated with task number, task time, predecessor tasks and successor tasks. Initially the tasks are numbered randomly. Later the tasks are renumbered, by the allocation algorithm, according to their respective priorities. Task times are calculated with the assumption that multiplication and addition take 30 and 20 time units respectively. The time units are not explicitly stated because they are dependent on the hardware used, and hardware is the subject of later research. The predecessor and successor tasks for any task are defined in terms of the inputs consumed by that task and the tasks receiving its output.

4.2 ALLOCATION PROCESS

This is perhaps the most important phase in the parallel method. The CP/MISF algorithm, outlined in Chapter 3, was fully implemented in PASCAL. The program is listed in APPENDIX B. The allocation process translates the task graph into an execution schedule. The execution schedule is the sequence at which the tasks are executed by the various processors.

The allocation program takes the task graph as its input. The input is provided in the form of task number, task time and links of the task graph. The program is modularized into various procedures taking advantage of the structured nature of PASCAL. The 'initialize' procedure reads the input data from an external file and generates an adjacency matrix A such that

$$A[i,j] = 1 \text{ if there is a link from task } i \text{ to task } j$$
$$= 0 \text{ otherwise.}$$

The 'level' procedure calculates the level of each task in the manner described in section 3.4.1. A matrix B is constructed such that

$$B[i,j] = 0 \text{ if there is a link from task } j \text{ to task } i$$
$$= -\infty \quad \text{otherwise.}$$

The 'renumber' procedure generates the new numbers of all the tasks in the descending order of the priorities. The adjacency matrix is correspondingly modified.

Then the 'main' program does the actual allocation job. Before allocating a task to a processor, it checks whether the predecessors have finished execution and whether the processor is free. Note that processors are chosen in the ascending order of the processor array, because uniform inter-processor communication time is assumed. In the case of nonuniform inter-processor communication, this part of the program can be modified so that a processor is chosen to minimize the communication time.

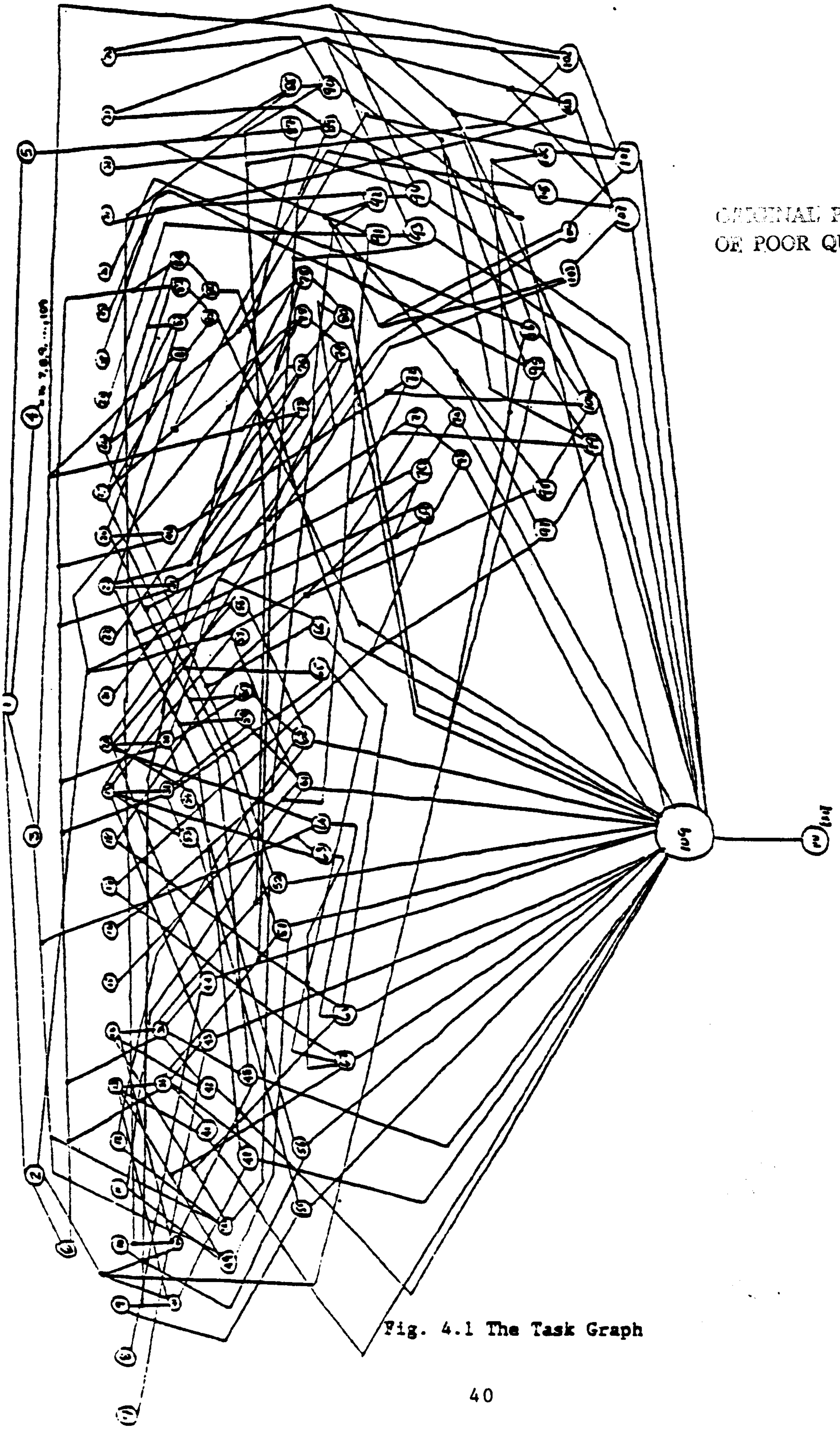

Fig. 4.1 The Task Graph

4.3 DISCUSSION OF THE RESULTS

The following three parameters are computed:

1) Total execution time.

2) Algorithm execution factor (AEF) defined as the ratio of the serial and parallel times.

3) Hardware utilization factor (HUF) defined as the ratio of the AEF and the number of processors.

It is found that the critical path of the task graph takes 300 time units, meaning that with the given task graph the minimum total execution time is 300 time units. As shown in Fig. 4.2 in the serial execution time is 6500 units. The total execution time progressively decreases as the number of processors increases. The critical time is obtained with 29 processors, and an optimum schedule is achieved. Beyond this point the increase in the number of processors has no effect on the execution time.

Fig. 4.3 shows the variation of the AEF with the number of processors. Note that the maximum AEF cannot exceed the total number of processors. The results show that AEF almost takes its maximum value in each case and is 21.67 when n = 29.

Fig. 4.4 is a plot of the HUF and the number of processors. A HUF of 100% means that the processors are fully utilized. It is observed that the HUF decreases with an increase in the number of processors. With 29 processors a hardware utilization factor of over 72% is achieved.

It is concluded that for the given task graph the optimum schedule is achieved with 29 processors. At this point the speedup compared to sequential execution reaches its maximum possible value of 21.67 and the hardware utilization is as high as 72.10%. These results show the validity of the parallel approach and also justifies the use of such an approach. The restructured method is much superior to the sequential algorithm and promises a substantial improvement in system performance.

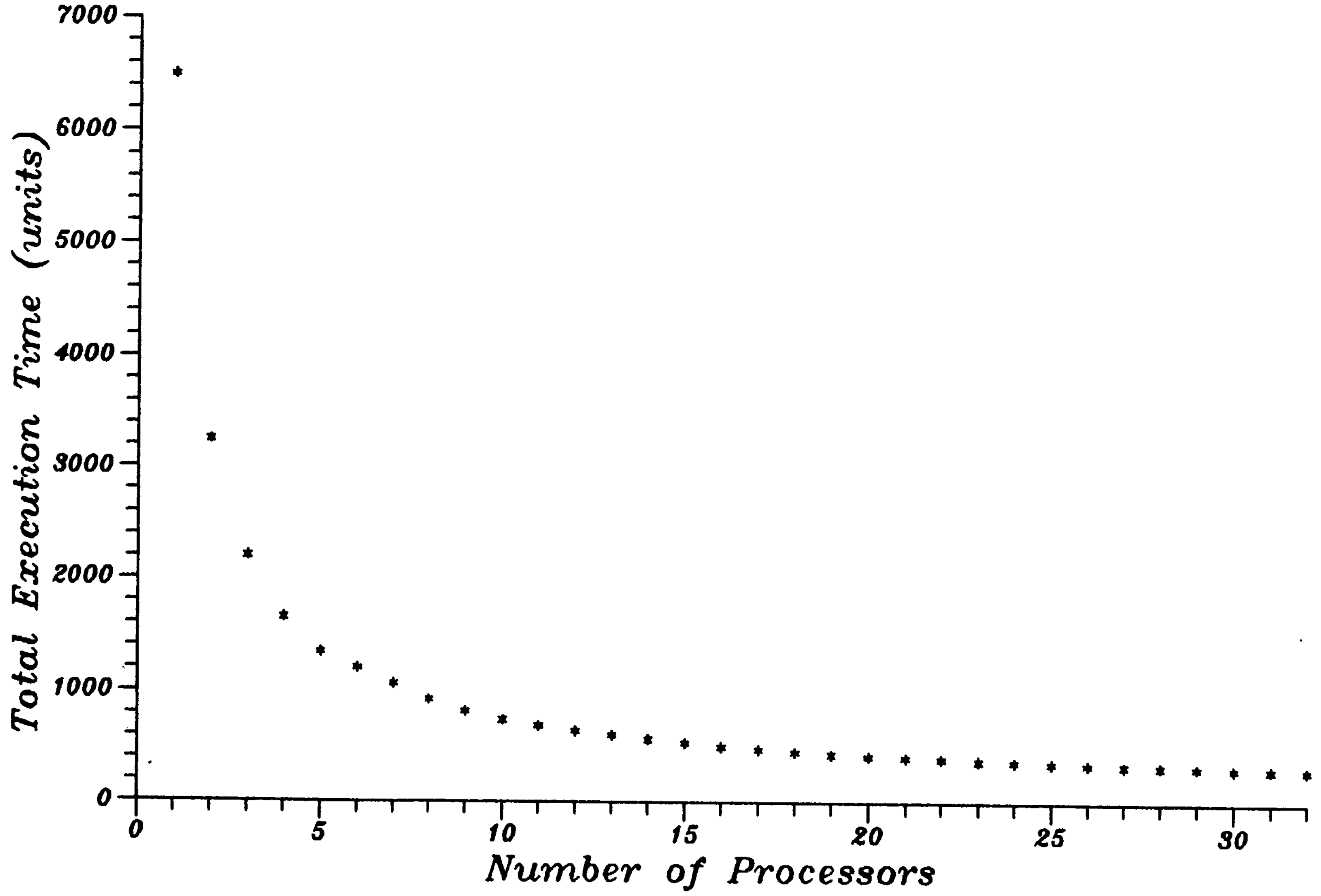

Figure 4.2 Total Execution Time vs. Number of Processors

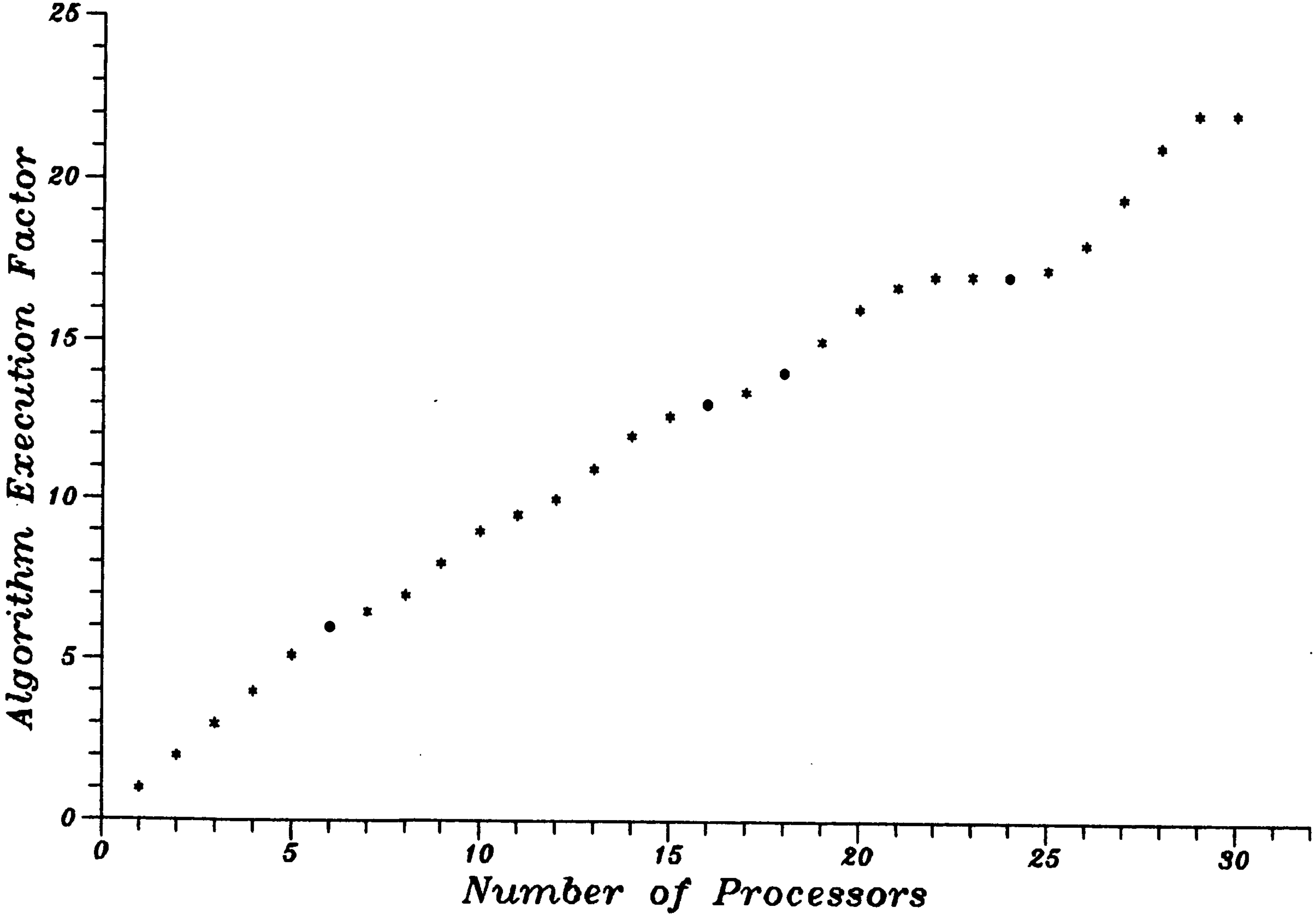

Figure 4.3 Algorithm Execution Factor vs. Number of Processors

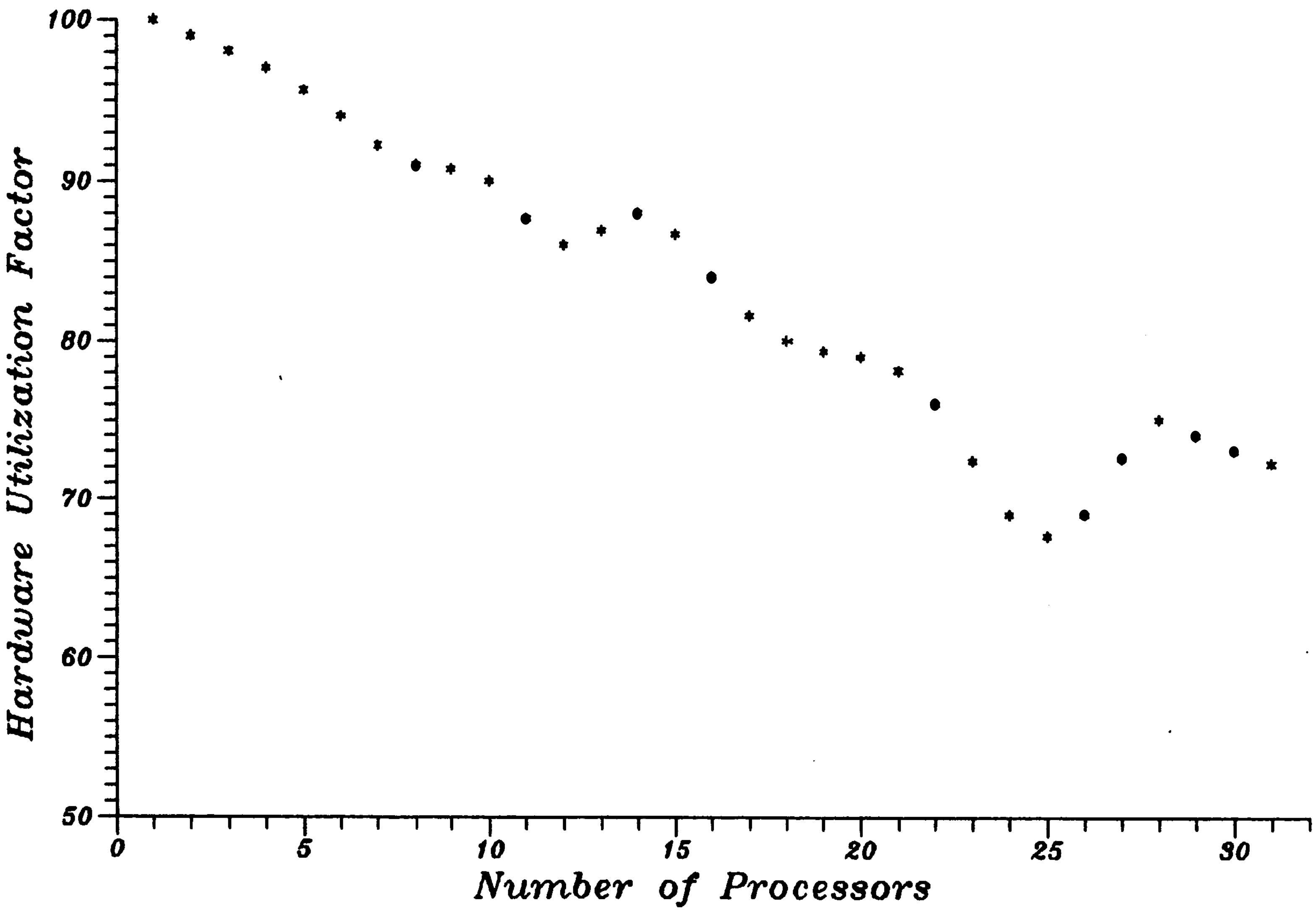

Figure 4.4 Hardware Utilization Factor vs. Number of Processors

Chapter 5
HARDWARE AND SOFTWARE ASPECTS

The design of a suitable multiprocessor computer, which optimally executes the various independent tasks, is discussed in this chapter. The parallelism analysis of the restructured algorithm assumes a multiprocessor environment with uniform interprocessor communication times and no hardware conflicts. As shown, the algorithm is optimally executed with 29 processors, providing the commanded acceleration in one direction. This chapter outlines the proposed computer architecture which is customized for the specific application.

5.1 PITFALLS OF VON NEUMANN MULTIPROCESSING

Most existing multiprocessors are variations of the von Neumann model of computation and have so far failed to yield any substantial benefits over single processor systems. As discussed by Arvind and Iannucci [25], there are several problems confronting the von Neumann style of multiprocessing.

The first problem is that of memory latency, the time between issuing a memory request and getting a response. If the computer contains a significant number of processors, and each is fast enough that its cycle time is limited by the speed of light, then the physical size of the whole computer will make most of the memory a significant distance away from any one processor. That is, several instruction times are needed to access most of the memory, if it is to be shared. Competition by several processors for the same memory at the same time makes the problem more severe. In trying to solve this problem, many architects use only messages, prohibiting shared memory entirely and surrendering flexibility and responsiveness. Others allow shared memory, but make local copies of the data in caches. This solution exchanges the latency problem for the cache coherence problem, i.e., how to maintain consistency when one or more of the copies is written.

A second problem is that of effective synchronization. Parallel processes must be able to wait for others without having to execute many extra instructions or waste time in other ways, and without significantly affecting the other processes that are running in parallel and not waiting. The use of traditional methods like interrupts, limits the synchronization rates to once every few hundred instructions. Primitives like test-and-set which wait busily and thereby can avoid exchanging processes are better, but these approaches usually waste instructions to accomplish waiting.

A third problem is the avoidance of bottlenecks which inhibit the amount of parallelism that can be attained, thereby limiting the number of processors that is practical. Changing an architecture, especially the instruction set, to correct bottlenecks in parallelism is inefficient because it destroys software compatibility.

5.2 DATA-DRIVEN PRINCIPLES

The solution of the control problem necessitates an efficient and fast way of handling the movement of large amounts of data among various processors. This makes the data-driven mode of computation an ideal candidate. Moreover, the problems associated with the von Neumann style of multiprocessing are avoided at the very basic level in the data-driven computer.

Instruction execution in a conventional von Neumann computer is under the control of the program counter. Whereas, the data-driven model of computation is based on the following two principles:

1) Any computation can proceed as soon as its operands become available. Potentially all operations that are thus enabled can execute simultaneously.

2) All operations are free of side-effects, so that two enabled operations can execute in either order, or concurrently, without error.

If a program has a sufficient amount of parallelism, then a data-driven processor can be kept fully utilized. In the previous chapters it is shown that there is an enormous amount of parallelism inherent in the avionics application. As discussed later, an execution unit in the proposed data-driven processing element receives enabled instructions only, and waiting for operands is done in a separate section. A data-driven processor, unlike a processor with a program counter, executes a stream of enabled instructions in a highly pipelined manner and allows greater freedom in the order of execution of the enabled instructions.

Data-driven architectures are usually classified as either static or dynamic. In a static architecture the nodes of a program graph are loaded into memory before computation begins, and, at most, one instance of a node at a time is enabled for firing. A dynamic architecture facilitates the simultaneous firing of several instances of a node, and these can be created at runtime. The architecture proposed in this chapter is of the latter type. The parallelism analyses, in the previous chapters, are based on a single iteration of the integration of the fourteen differential equations, but there are obvious concurrencies between the various iterations. This architecture has the provision of unfolding the integration loop at runtime by creating multiple instances of the loop body and then executing these instances concurrently.

5.3 DIFFERENT SOFTWARE AND HARDWARE STRUCTURES

In this section the salient software and hardware aspects of the proposed data-driven computer are outlined. The previous chapters emphasized the software aspects which are accomplished in the preprocessing stage. The actual machine to execute the tasks constitutes the hardware aspect.

5.3.1 Language Considerations

The entire process of defining the algorithm, removing dependencies, constructing a task graph, and finally allocating the tasks to the various processors must be completed before the actual execution starts. This process is deliberately kept language-independent to gain flexibility. Since the architecture is data-driven, it is more advantageous to use a functional language than a conventional imperative language as the high-level language to represent the problem. There is a difference between the high-level language required to represent the problem and the base language which is efficiently implemented by the architecture. The high-level language should satisfy the following properties:

1) Freedom from side-effects : This is necessary to ensure that data dependencies are the same as the sequencing constraints. Global variables are not allowed and procedures cannot modify variables in the calling program. Updating a variable results in the creation of new variables.

2) Locality of effect : To avoid memory overflow variables should have a definite region of operation or scope. This also avoids the apparent dependencies that result from duplication of labels.

3) Equivalence of instruction scheduling constraints with data dependencies : This means that all the information needed to execute a program is contained in the task graph.

4) Single assignment : This means that each variable may appear on the left side of only one assignment statement in the part of the program in which it is active.

There are three main categories of programming languages, functional, actor, and logic, that are suitable for data-driven computations. Functional languages can either be single-assignment, like ID, VAL, VALID, and LUCID, or applicative, like pure LISP, SASL, and FP. Actor languages are programming systems composed of objects that interact only by sending and receiving messages. SMALLTALK is an actor language. Logic languages are based on symbolic logic and PROLOG is an example. Any one of these languages can be chosen as a high-level language for this architecture.

The base language of this computer is the graphical representation termed the task graph, discussed in Chapter 3. The machine efficiently executes the tasks shown in the task graph, satisfying the precedence constraints of the graph. A task can be executed as soon as all its inputs are available.

5.3.2 Tagged Tokens

In a manner similar to Arvind [26] and the Manchester Dataflow machine [27], information is carried by tokens that flow along the arcs of the task graph. A task is enabled when, and only when, all of its input tokens are present. An enabled task fires by absorbing its input tokens and producing output tokens that carry the result as their value. The order of execution is unimportant since there are no races.

5.3.3 The Overall Architecture

The block diagram of the overall architecture is shown in Fig. 5.1. There are three basic stages, the preprocessing stage, the execution stage, and the output stage. In the preprocessing stage a conventional host computer gathers the various input data and coordinates the overall activities. An important part of this stage is the allocation unit which allocates the tasks to the various processors in an optimal fashion. Following the allocation phase, the data and instruction tokens are downloaded onto the individual memories of the processing elements (PEs).

In the heart of the architecture lies the data-driven execution stage or the PE array. There are two clusters of 64 PEs each, which are connected in a star configuration. As noted in Chapter 4, the optimum execution of the algorithm requires 29 PEs. 60 'workhorse' PEs are used for computation purposes only, and the remaining 4 PEs in each cluster are dedicated for various purposes. One dedicated PE is the communication link between the preprocessing stage and the cluster. The second dedicated PE is reserved for diagnostic purposes. This diagnostic processor helps in recovery from faults and in reassignment of PEs. The third is reserved for inter-cluster communication, and the remaining one serves as a link between the cluster and the output stage. At any given time, only 29 workhorse PEs function within a cluster. The others remain in standby and are used when a PE must be aborted after a fault is detected.

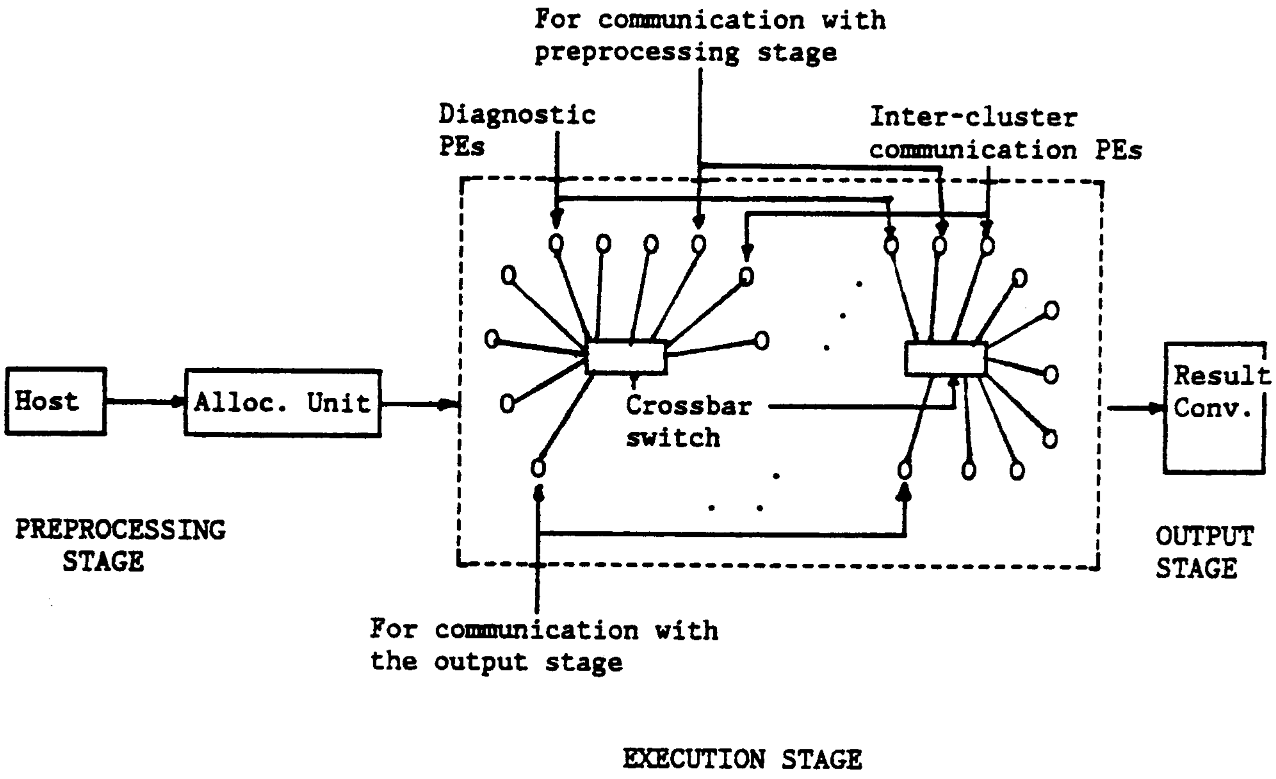

Fig. 5.1 Block Diagram of the Overall Architecture

Two clusters are required because an identical number of calculations must be performed to compute the acceleration of the vehicle in the transverse direction. The efficiency of the architecture results from the fast movement of data through the execution stage, which is devoid of the conventional von Neumann bottlenecks.

The output stage receives the final result tokens from the execution stage via the dedicated PEs in each cluster. These tokens are converted into a form that serves as an input to the subsequent motion control actuators.

5.3.4 Network Topology of the Execution Stage

Among the possible configurations are the ring, tree, completely connected, and the star topologies. In the ring network N processors are connected on a circular bus. Only 1/N of the bus bandwidth is available to each processor, and failure of a single node or path within the ring may halt communication in the entire ring. To alleviate this problem, designers have constructed partially and completely connected rings at the expense of increased network complexity and cost. Also, the number of nodes in a ring is limited because message delays increase linearly with the number of nodes, making a ring inefficient for heavy traffic.

A tree network uses the minimal number of connections per processor. Communication between remote leaves faces a bottleneck towards the top of the tree, and the data paths become longer as the number of nodes increases. Hence a tree is also unsuitable for heavy communication. The completely connected network requires N^2 connection links for N processors, which is prohibitively expensive.

The star network has both logical and hardware simplicities. If the tasks are uniformly distributed, most messages traverse only two communication paths. A major vulnerability of this topology is the active hub which, not only introduces queuing delay, but also disables the entire network, once it fails. For this reason a passive hub, which is nothing but a physical connection of the various paths, is used.

The result token from every PE is broadcast to all PEs in the cluster. The result token contains a field denoting the destination address, and all PEs decode this field to find a match.

5.3.5 Crossbar Switching

An alternative to broadcast communication is the use of crossbar switch networks. Communication is inherently sequential if the message is broadcast to all the PEs. A high-speed crossbar switch can be used at the hub of the star for routing the information to the appropriate processor. The crossbar switch gives the cluster the capability of parallel communication between pairs of processors. Extremely fast switches are available which make the switching time negligible compared to token formatting and communication times.

An n by m crossbar switch is a device with n inlets, m outlets, and an array of n*m contacts, sometimes called crosspoints, for connecting each inlet to each outlet. A crossbar network is an interconnection of crossbar switches in accordance with certain rules. The switches must be partitioned into a number of classes called stages in such a way that all switches in a given stage have the same number of inlets and outlets. The inlets of the switches in the first stage are the inputs of the network. The outlets of the switches in each stage except the last are connected in an one-to-one fashion to the inlets of the switches in the following stage by links. Finally the outlets of the switches in the last stage are the outputs of the network.

Delta networks are multiple-stage networks with each stage consisting of several crossbar switches, as shown in Fig. 5.2. The switches in a buffered delta network have buffers to temporarily store tokens which cannot be forwarded in the current cycle. An N by N buffered delta network can be constructed from B by B crossbar switches, each capable of forwarding a token that arrives at any of its B inputs to any of its B outputs (see Fig. 5.2, where N=8 and B=2). The network has n/b stages (numbered 1,2,...,n/b, where $n=\log_2 N$ and $b=\log_2 B$), and each stage has $2^{(n-b)}$ crossbar switches.

For each of its output ports, a switch selects one token from the set of tokens contending for that port and offers it to the next stage connected to that port. The output port through which a token leaves the switch is determined by the switch from a destination address included in the token. Generally, the output ports requested by the tokens at the heads of the switch buffers are considered. For each of these output ports, one of the requesting tokens is selected equiprobably and offered to a switch in the next connected stage. The switches with input tokens forward them to the intended buffers, if these buffers have a vacancy at the beginning of the clock cycle. For each accepted token an acknowledge signal is sent to the switch from which the token came.

Three major factors v'hich influence the performance of a buffered delta network are the size of the switches, the size of the buffers used in each switch, and their position with respect to the switch. It is shown in [28] that for small buffer sizes, delta networks constructed with 4 by 4 switches provide slightly better throughput and substantially lower delay than 2 by 2 switches. However, for large buffer sizes the delta networks constructed from 2 by 2 switches provide better throughput at the expense of larger delays.

Tokens are blocked when there is either more than one token in the switch (switch blocking) or insufficient buffer capacity (buffer blocking). In networks with large buffer sizes, buffer blocking is minimized and the degradation of throughput is primarily due to switch blocking. Since switch blocking increases with the size of the crossbar switch, it is advantageous to use 2 by 2 switches instead of 4 by 4 ones. The buffers can be provided at the input links of each switch, as shown in Fig. 5.3a, or they can be inside the switch as shown in Fig. 5.3b. Kumar and Jump [28] have deduced that with large buffer sizes it is advantageous to use buffers inside the switches, in terms of both throughput and delay.

In a crossbar switch several tokens may simultaneously request the same output port, so various priority schemes can be used for selection. The simplest method is to select one of the tokens randomly. Another is the rotating priority scheme in which all buffers in the switch are assigned a permanent cyclic order. In each clock cycle all the buffers are considered in this order, starting from a designated high-priority buffer. The buffer adjacent to the high-priority one in the current cycle becomes the high-priority buffer in the next cycle. In another scheme, a token in any buffer is considered to have a priority equal to

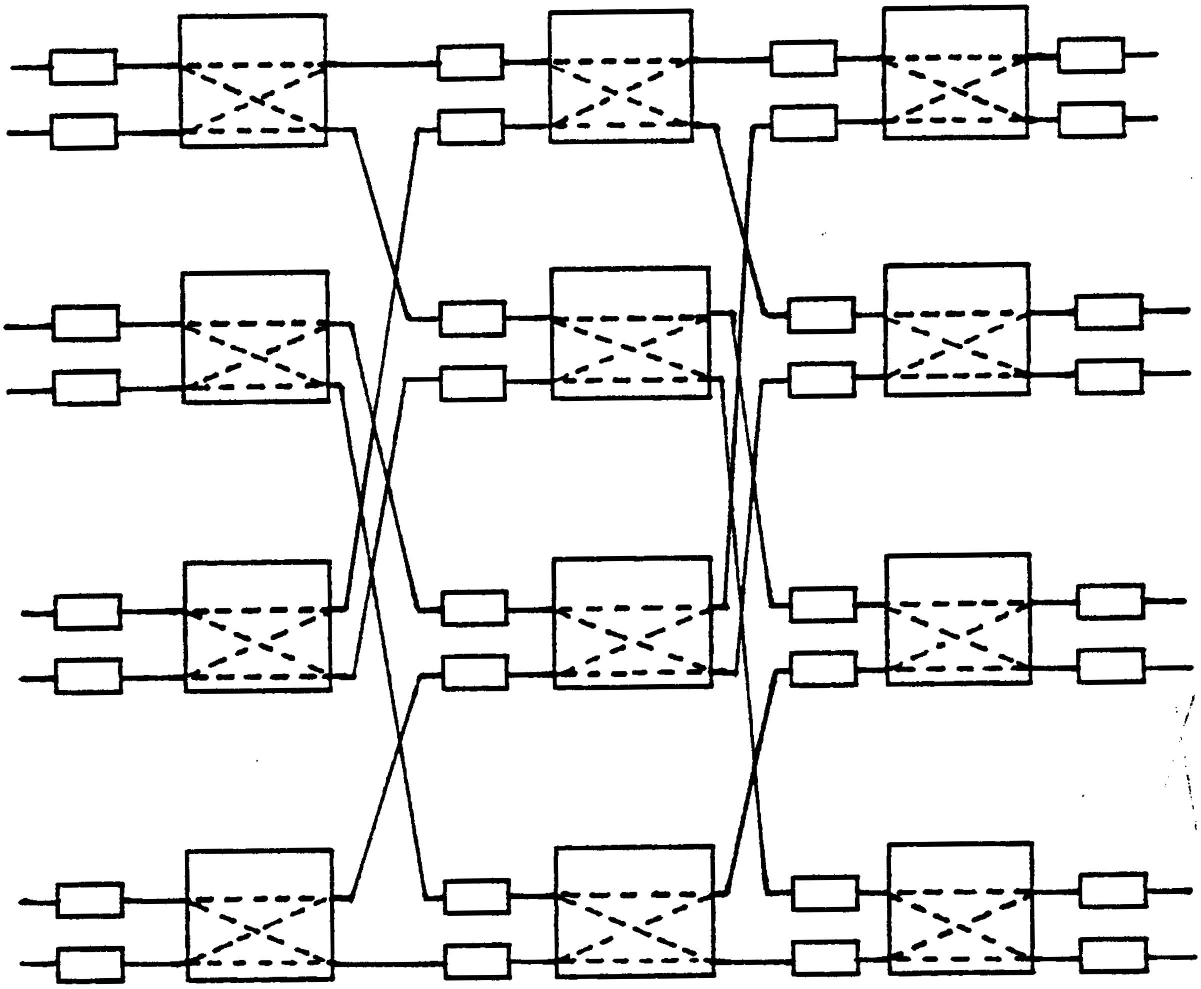

Fig. 5.2 A Buffered Delta Network

the number of tokens in that buffer. For tokens with the same priority measure, one is selected equiprobably. From the results shown in [28], the performance of the random selection scheme is found to be similar to the other two schemes, and is the easiest one to implement.

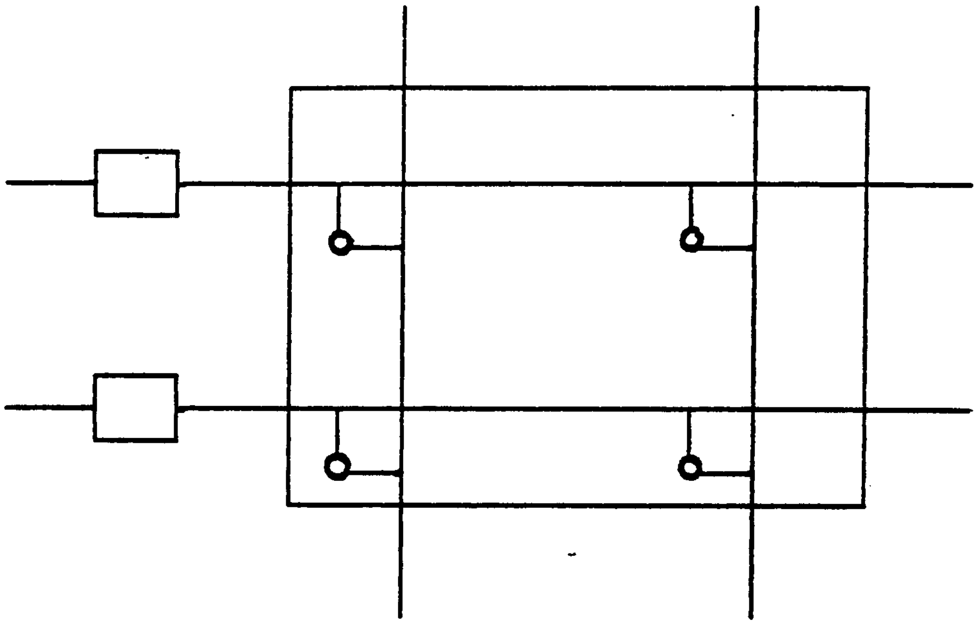

Fig 5.3a Crossbar Switch with buffers at the input links

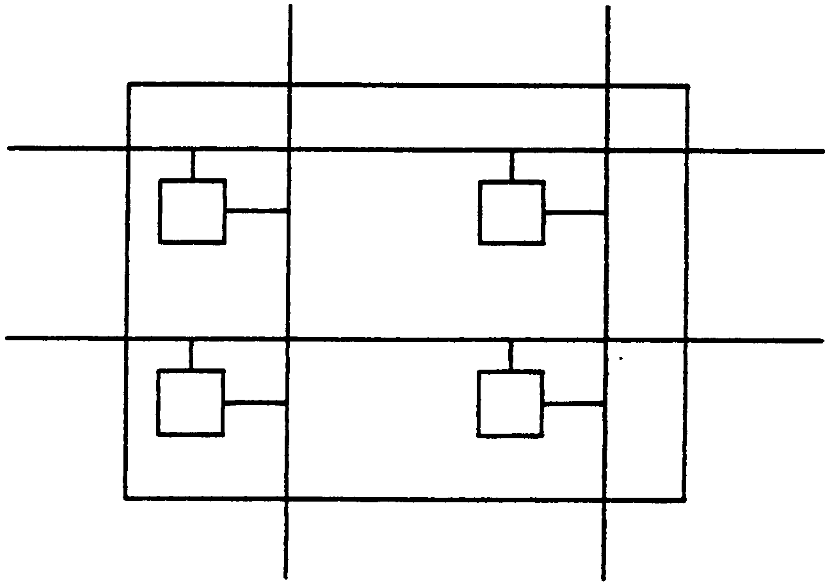

Fig 5.3b Crossbar Switch with the buffers inside the switch

5.3.6 Token Format

As mentioned earlier, information is communicated between PEs via tokens or packets. There are two types of tokens, the instruction token containing the required information for a node execution and the data token carring data required to enable the node. A 64-bit token size accomodates a 32-bit data or operation field and 32 bits for the control directives. The control portion of the token includes the following subfields:

i) Check field : This determines whether it is an instruction token or a data token.

ii) Module field : This identifies the block of code (procedure or loop) to which the token belongs.

iii) Instruction number field : This identifies the instruction number within a specific block.

iv) Processor field : This denotes the processor responsible for executing the code.

v) Error check field : This contains information for checking the validity of the token. Error detection and correction (EDAC) codes like cyclic redundancy check (CRC) and Hamming codes can be used.

vi) Data counter field : This is a part of the data token only. It indicates the number of operands required to enable a node.

The remaining 32 bits of the token contains the data value or the instruction code, as the case may be. From the software point of view, a longer token is better since it can carry more information. On the other hand, a shorter token is better from the hardware point of view,because it reduces the amount of hardware and network connections. Hence, the size of the token should be chosen in an optimal manner.

5.3.7 Design of the Processing Element

The processing element (PE) is designed to meet some specific requirements. Every token has a tag or control field, and the PE has the ability to decode the tag and route the token to the separate components of the PE. The PE provides local storage for instruction and data tokens. A Contents Associative Memory (CAM) is simulated using the hardware hashing technique. A hash table is accessed by the hash key generated from the tag at a very high search speed. The PE provides circuitry for EDAC decoding and encoding, and has the ability to detect, isolate, and rectify faults. Most of the units are self-checking.

A block diagram of the proposed PE design is shown in Fig.5.4. It consists of an input queue, an EDAC decode unit, a wait-store-match unit, an execution unit, an output unit, and the overflow and intermediate buffers.

The input queue is a FIFO buffer, receiving tokens from other PEs and sending them to the EDAC decode unit. It works as a rate balancing mechanism, attempting to even the rate of token production and consumption. Therefore, it allows the wait-match-store unit and the execution unit to work concurrently.

The EDAC decode unit checks the error code of the token. If a correctable fault is detected, it passes the rectified token to the wait-match-store unit. If the fault is not rectified, it informs the diagnostic unit.

The wait-match-store unit consists of a code memory to store the instructions and an operand memory for data. After a token is received, the hashing mechanism generates a hash key to address the hash table, called the Operand Block Table (OBT). Each entry of the operand memory consists of 34 bits, which includes a 2-bit Operand Enable Flag (OEF) and a 32-bit data value. The OEF shows the existence of the operand. It is set to 00 if no token has arrived. When a data token arrives, the OEF is set to 10 or 01, depending on whether it is the left or right operand (denoted by the data counter field), and the data is stored in the data area.

After a token is received, the unit starts accessing both the operand and code memories simultaneously. If the instruction is monadic, the operand memory is not searched, and the executable packet is immediately generated. For a dyadic instruction the operand memory is searched, and if the matched token is found, the executable packet is generated, and the OEF is set to 00. The executable packet is passed onto the execution unit.

The execution unit performs all arithmetic and logic instructions. Some commonly used instructions can be hardwired to enhance the speed of execution. The result tokens are forwarded to the output unit.

The output unit generates the tag field of the result token. The token is properly formatted and the EDAC code is embedded in it. Since it is possible to encounter delay while transmitting a token through the communication network, a buffer is also provided in the output unit.

The overflow memory is provided to augment the operand memory. An overflow occurs when all locations in the operand memory are occupied. Then the unmatched incoming token is stored in the overflow buffer, and indicator flags are set up to notify subsequent tokens. The intermediate buffer stores the matched tokens of an enabled instruction, so that the tokens can be retrieved when a fault is detected after execution.

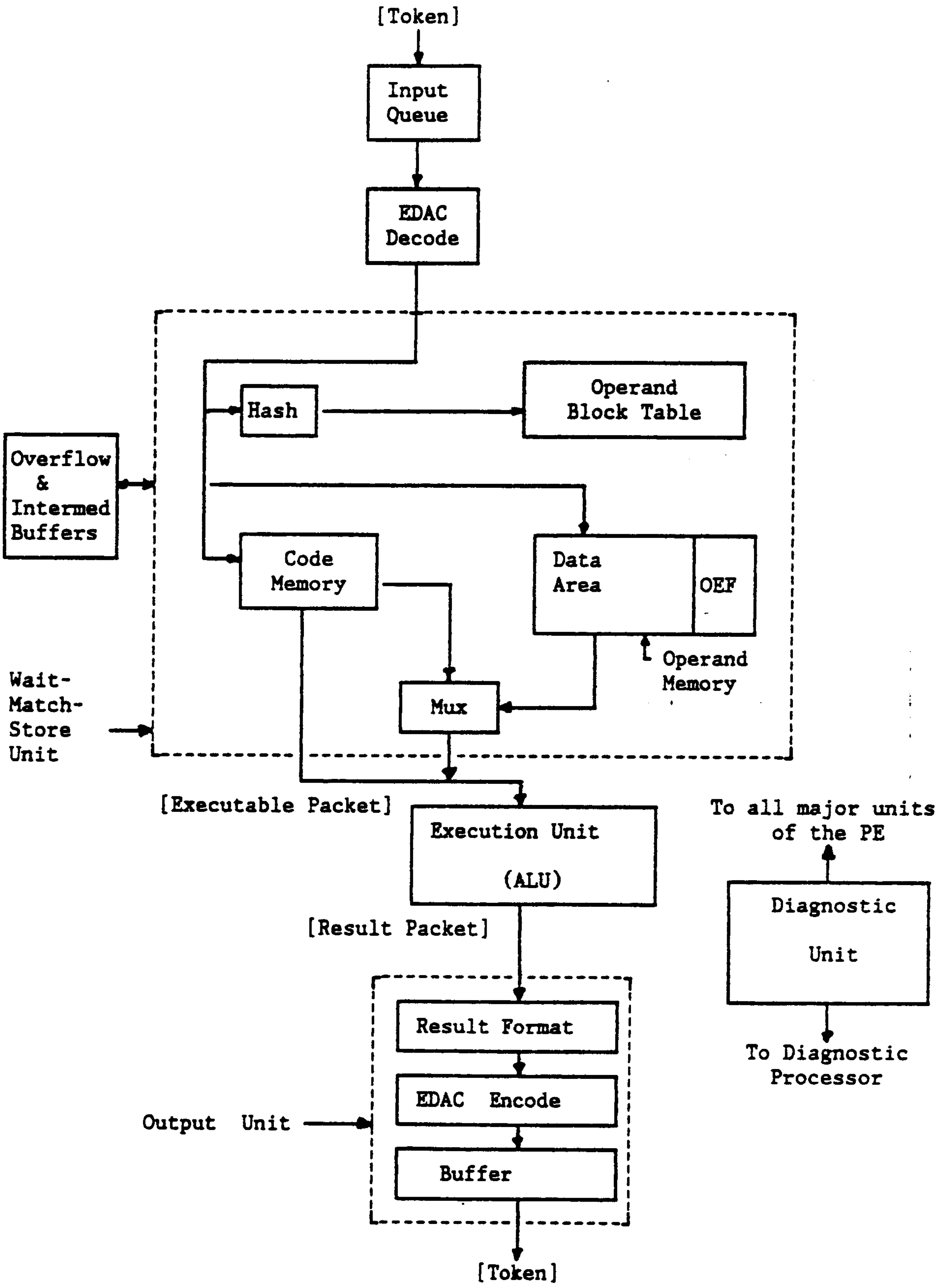

Fig. 5.4 Block Diagram of the proposed PE

The diagnostic unit periodically checks the major components of the PE. These tests can be executed either in parallel with the normal execution or when the tested unit is inactive to avoid any degradation of system performance. Once a fault is detected, the diagnostic unit aborts the PE and informs the diagnostic processor, which in turn initializes the reassignment process. The diagnostic unit can prevent the output unit from transmitting a faulty token, and thus can localize the fault.

5.3.8 Fault Tolerance

Fault tolerance is an important requirement of any multiprocessor architecture. In the proposed architecture both hardware and software fault tolerances are used to improve system reliability. The workhorse PEs within a cluster are duplicated to provide hardware redundancy, and are built with extremely reliable components. The communication network is very rugged and reliable. Fine-grained fault tolerance is provided within every PE with the help of self-checking circuitry and diagnostic units.

Software fault tolerance is provided by watchdog timers and EDAC codes. A watchdog timer is a simple and inexpensive way of monitoring proper process functions. A timer is maintained as a process separate from the one it checks, and is set as soon as the process starts. The process resets the timer after completing successful completion. If the timer is not reset, then a process failure is assumed.

Transmission faults are easily detected and corrected using various codes. A simple example is the single error correction and double error detection Hamming code. For a 8-bit data word the encoder generates a 12-bit word like

$$H_1\ H_2\ D_3\ H_4\ D_5\ D_6\ D_7\ H_8\ D_9\ D_{10}\ D_{11}\ D_{12}$$

where H's are the check bits and D's are the data bits.

The following are called the syndrome equations :

$$S_1 = H_1 + D_3 + D_5 + D_7 + D_9 + D_{11}$$
$$S_2 = H_2 + D_3 + D_6 + D_7 + D_{10}$$
$$S_4 = H_4 + D_5 + D_6 + D_7 + D_{12}$$
$$S_8 = H_8 + D_9 + D_{10} + D_{11} + D_{12}$$

where '+' denotes an exclusive OR operation. While generating the code, the check bits are produced by setting the syndrome bits to zero. During the checking process, the syndrome bits are checked. If any of the resulting syndrome bits is nonzero, then a detectable error has occurred. An error can be corrected provided only bit is erroneous. The binary number $S_8S_4S_2S_1$ gives the position of the erroneous bit. For example, if D_{12} is changed, then S_8 and S_4 are nonzero and $S_8S_4S_2S_1 =$ 1100 = 12 in decimal. This code can be extended to accommodate longer data words.

5.3.9 Stacked Hybrid WSI Technology

The entire architecture must be housed within a small package. Hence the dimensions, weight, and cost of the hardware are important considerations. Hybrid Wafer Scale Integration (WSI) is a possible solution. This technology involves scribing the wafer after fabrication. The actual PEs are then separated and remounted in preassigned positions onto a substrate of polyimide. The inter-processor links are fabricated by ion-implantation techniques. Hybrid WSI partially eliminates the two major problems of traditional WSI, viz., yield and power dissipation. Since each PE is scribed and then separated, partial testing may be performed prior to remounting. Power is more easily dissipated through the polyimide substrate.

Multi-stack wafers can be used to house the two clusters and other units. The 3D Computer Studies Department at the Hughes Research Laboratories, Malibu, California has built an image processing cellular array of stacked CMOS wafers with feedthroughs and interconnects. A significant advantage of such a scheme is the upgradeability of the architecture as additional features are accommodated by introducing more wafer stacks.

Chapter 6
CONCLUSIONS AND RECOMMENDATIONS

The previous chapters have discussed the need, development, utilization, and validity of this research. Experimental results have been discussed in Chapter 4. The purpose of this chapter is to express some general conclusions and recommendations.

6.1 CONCLUSIONS

This research has proved that acceptable results can be obtained by using parallel processing in real-time systems. It has shown that enhancement of avionics design and vehicle control is possible by computing the guidance commands in real-time, exploiting the parallelism inherent in the problem.

There are various ways of applying parallel processing techniques to meet the need of rapid and real-time computation. It is concluded that one of these approaches, outlined in this report, comprises the following two major phases :

1.) The sequential algorithm is suitably restructured by removing dependencies, identifying concurrent tasks, exploiting optimum parallelism, and optimally allocating the tasks to available resources.

2.) Appropriate hardware structures are designed to implement the parallel or modified algorithm.

Together, the above two phases constitute an innovative, customized computer architecture for the algorithmic execution of a real-time system. The data-driven mode of computation is ideally suited for the real-time solution of control processing, avoiding the bottlenecks of von Neumann multiprocessing.

This research has also demonstrated the significance of the allocation process in a parallel processing application. Optimal allocation can be achieved with the help of heuristic algorithms.

6.2 RECOMMENDATIONS

The scope of this research is not limited to the specific field of guidance and control. The authors recommend the use of the techniques, outlined in this report, to similar problem areas in other real-time systems.

The significance of the allocation process has been demonstrated by this research. The authors suggest the design of allocation algorithms which are themselves suitable for parallel implementation, an example of which is the branch and bound technique, discussed in section 3.4.2.

The effect of using decoupling techniques to reduce dependencies between differential equations should be investigated. The performance can be further improved by defining tasks with optimum granularity. This can be achieved by striking a suitable balance between the execution and communication times. The authors also recommend further research in the design of hardware structures which are capable of executing specific algorithms and graphs.

REFERENCES

1. Vitalji Garber, Walter S. Flory,III, "Optimum Intercept Laws", Technical Report No. RD-TR-67-10, Advanced Systems Laboratory, U.S.

   Army Missile Command, Redstone Arsenal, Alabama, December 1967.

2. G.Willems, "Optimal Controllers for Homing Missiles", Report No. RE-TR-68-15, U.S. Army Missile Command, Redstone Arsenal, Alabama, September 1968.

3. John J.Deyst Jr. & Charles F.Price, "Optimal Stochastic Guidance Laws for Tactical Missiles", J. Spacecraft, vol.10, no.5, May 1973.

4. R.B.Asher & J.P.Matuszewski, "Optimal Guidance with Maneuvering Targets", J. Spacecraft, vol.11, no.3, pp 204-206, March 1974.

5. J.N.Youngblood & M.J.Clauda, "Optimal Guidance for Missiles with Airframe Dynamics Against Maneuvering Targets", Technical Report AFATL-TR-77-66, Air Force Armament Laboratory, Eglin AFB, Florida, May 1977.

6. J.N.Youngblood, "Optimal Linear Guidance of Air-to-Air Missiles", Technical Report AFATL-TR-78-12, Air Force Armament Laboratory, Eglin AFB, Florida, February 1978.

7. J.N.Youngblood, "Advanced Linear Guidance Laws for Air-to-Air Missiles", Bureau of Engineering Research, Report No. 253-177, University of Alabama, January 1980.

8. J.N.Youngblood, "Optimal Linear Guidance of Air-to-Air Missiles", Bureau of Engineering Research, Report No. 222-09, University of Alabama, July 1978.

9. L.Stockum & F.C.Weimer, "Optimal and Suboptimal Guidance for a Short Range Homing Missile", IEEE Trans. Aerospace and Electronic Systems, vol.AES-12, pp 353-361, May 1976.

10. B. Sridhar & N.K.Gupta, "Accurate Real Time SRAMM Guidance Using Singular Perturbation Optimal Control", National Aerospace and Electronic Conferenece, Dayton, Ohio, May 1979.

11. W.L.Miranker & W,Liniger, "Parallel Methods for the Numerical Integration of Ordinary Differential Equations",Math.Comput.,vol.21, pp 303-320,1967.

12. Z.Kopal,Numerical Analysis with emphasis on the application of Numerical Techniques to Problems of Infinitesimal Calculus in Single Variable , Wiley N.Y., Chapman Hall, London, 1955.

13. J.Nievergelt, "Parallel Methods for Integrating Ordinary Differential Equations",CACM vol. 7, no. 12, Dec 1964, pp 731-733.

14. R.Lord & S.Kumar, "Parallel Solution of Flight Simulation Equations", Proc. of the 1980 Summer Computer Simulation Conference, Seattle WA, August 1980, AFIPS Press, pp 217-233.

15. N.Katz,M.A.Franklin,& A.Sen, "Optimally Stable Parallel Predictors for Adams-Moulton Correctors", Comp. and Math. with Applications, vol.3, 1977, pp 217-233.

16. E.G.Coffman & P.J.Denning, Operating Systems Theory, Prentice Hall, Englewood Cliffs, N.J., 1974.

17. E.G.Coffman, Computer and Job-Shop Scheduling Theory, New York, Wiley, 1976.

18. J.D.Ullman, "Polynomial NP-Complete Scheduling Problems", Operating Systems Review, vol.7, no.4, 1973, pp 96-101.

19. M.R.Garey & D.S.Johnson, Computers and Intractability : A Guide to the Theory of NP-Completeness, San Francisco, CA., Freeman, 1979.

20. T.L.Adam, K.P.Chandy, & J.R.Dickson, "A Comparison of list schedules for Parallel Processing Systems", Comm. ACM, vol.17, Dec 1974, pp 685-690.

21. H.Kasahara & S.Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing",IEEE Trans. on Computers, vol.C-33, no.11, Nov. 1984, pp 1023-1029.

22. T.C.Hu, "Parallel Sequencing and Assembly line Problem", Operations Research, vol.9, Nov. 1961, pp 841-848.

23. E.L.Lawler, Combinatorial Optimization : Networks and Matroids, New York, Holt, Rinehart, and Winston, 1976.

24. W.H.Kohler, "Characterization and Theoretical Comparisons of Branch-and-Bound Algorithms for Permutation problems", J. of ACM, vol.21, Jan. 1974, pp 140-156.

25. Arvind & R.A. Iannucci, "A Critique of Multiprocessing von Neumann Style", Proceedings of the 10th Symposium on Computer Architecture, 1983, pp 426-436.

26. Arvind, "The Tagged Token Dataflow Architecture", Draft, Laboratory for Computer Science, MIT, November 1982.

27. I. Watson & J. Gurd, "A Pro'otype Dataflow Computer with Token Labeling", AFIPS Conference Proceedings, vol. 48, June 1979, pp 623-628.

28. M. Kumar & J.R. Jump, "Performance Enhancement in Buffered Delta Networks Using Crossbar Switches and Multiple Links", Journal of Parallel and Distributed Computing 1, !984, pp 81-103.

Appendix A
TASK LIST

DESCRIPTION	NO.	TIME	PREDECESSORS	SUCCESSORS
Entry	1	0	None	2,3,4,5
$a=-w_n*w_n$	2	30	1	6,45,46,57,58,69,70 77,78,95,96,105,106
$b=-2*w*w_n$	3	50	1	49,50,63,64,75,76,81, 82,101,102
INC i	4	20	1	7,8,...108,109
A_T	5	20	1	87,88,91,92,97,98, 103,104
Q=a*a/r	6	60	2	35,36,37,38,39,40,83,84
$P1^p$	7	110	4	43
$P1^c$	8	110	4	44
$P2^p$	9	110	4	45,55,87
$P2^c$	10	110	4	46,56,88
$P3^p$	11	110	4	49,61
$P3^c$	12	110	4	50,62
$P4^p$	13	110	4	35,41,45,49,67
$P4^c$	14	110	4	36,42,46,50,68
$P5^p$	15	110	4	61,91
$P5^c$	16	110	4	62,92
$P6^p$	17	110	4	67,69,97
$P6^c$	18	110	4	68,70,98
$P7^p$	19	110	4	37,43,53,57,63,77,103
$P7^c$	20	110	4	38,44,54,58,64,78,104
$P8^p$	21	110	4	75
$P8^c$	22	110	4	76
$P9^p$	23	110	4	39,47,59,69,71,75,79,81
$P9^c$	24	110	4	40,48,60,70,72,76,80,82
$P10^p$	25	110	4	51,65,77,81,83,105
$P10^c$	26	110	4	52,66,78,82,84,106
$P11^p$	27	110	4	91
$P11^c$	28	110	4	92
$P12^p$	29	110	4	95
$P12^c$	30	110	4	96
$P13^p$	31	110	4	101
$P13^c$	32	110	4	102
$P14^p$	33	110	4	89,93,95,99,101,107
$P14^c$	34	110	4	90,94,96,100,102,108
$TEMP1^p=Q*P4^p$	35	30	6,13	41,43,47,51,89,93
$TEMP1^c$	36	30	6,14	42,44,48,52,90,94
$TEMP2^p=Q*P7^p$	37	30	6,19	53,59,65
$TEMP2^c$	38	30	6,20	54,60,66
$TEMP3^p=Q*P9^p$	39	30	6,23	71,79,99
$TEMP3^c$	40	30	6,24	72,80,100
$f1^p=-P4^p*TEMP1^p$	41	30	13,35	109
$f1^c$	42	30	14,36	119
$f2^p=P1^p-P7^p*TEMP1$	43	50	7,19,35	109
$f2^c$	44	50	8,20,36	109
$TEMP4^p=a*P4^p+p^{2p}$	45	50	2,9,13	47

TEMP4^c	46	50	2,10,14	48
f3^P=TEMP1^P*P9^P + TEMP4^P	47	50	23,35,45	109
f3^c	48	50	24,36,46	109
TEMP5^P=P3^P+P4^P*b	49	50	3,11,13	51
TEMP5^c	50	50	3,12,14	52
f4^P=TEMP1^P*P10^P + TEMP5^P	51	50	25,35,49	109
f4^c	52	50	26,36,50	109
TEMP6^P=TEMP2^P*P7^P	53	30	19,37	55
TEMP6^c	54	30	20,38	56
f5^P=2*P2^P-TEMP6^P	55	40	9,53	109
f5^c	56	40	10,54	109
TEMP7^P=a*P7^P	57	30	2,19	61
TEMP7^c	58	30	2,20	62
TEMP8^P=TEMP2^P*P9^P	59	30	23,37	61
TEMP8^c	60	30	24,38	62
f6^P=P3^P+P5^P+TEMP7^P -TEMP8^P	61	60	11,15,57,59	109
f6^c	62	60	12,16,58,60	109
TEMP9^P=b*P7^P	63	30	3,19	67
TEMP9^c	64	30	3,20	68
TEMP10^P=TEMP2^P*P10^P	65	30	25,37	67
TEMP10^c	66	30	26,38	68
f7^P=P6^P+P4^P+TEMP9^P -TEMP10^P	67	60	13,17,63,65	109
f7^c	68	60	14,18,64,66	109
TEMP11^P=P6^P+a*P9^P	69	50	2,17,23	73
TEMP11^c	70	50	2,18,24	74
TEMP12^P=TEMP3^P*P9^P	71	30	23,39	73
TEMP12^c	72	30	24,40	74
f8^P=2*TEMP11^P -TEMP12^P	73	40	69,71	109
f8^c	74	40	70,72	109
TEMP13^P=P8^P+b*P9^P	75	50	3,21,23	79
TEMP13^c	76	50	3,22,24	80
TEMP14^P=P7^P+a*P10^P	77	50	2,19,25	79
TEMP14^c	78	50	2,20,26	80
f9^P=TEMP13^P+TEMP14^P -TEMP3^P*P9^P	79	70	23,39,75,77	109
f9^c	80	70	24,40,76,78	109
TEMP15^P=P9^P+b*P10^P	81	50	3,23,25	85
TEMP15^c	82	50	3,24,26	86
TEMP16^P=Q*P10^P	83	30	6,25	85
TEMP16^c	84	30	6,26	86
f10^P=2*TEMP15^P -TEMP16^P	85	40	81,83	109
f10^c	86	40	82,84	109
TEMP17^P=P2*A^T	87	30	5,9	89
TEMP17^c	88	30	5,10	90
f11^P=TEMP17^P- TEMP1^P*P4^P	89	50	33,35,87	109
f11^c	90	50	34,36,88	109
TEMP18^P=P11^P-P5^P*A^T	91	50	5,15,27	93

$TEMP18^{c}$	92	50	5,16,28	94
$f12^{P}=TEMP18^{P}-TEMP1^{P}*P14^{P}$	93	50	33,35,91	109
$f12^{c}$	94	50	34,36,92	109
$TEMP19^{P}=P12^{P}+a*P14^{P}$	95	50	2,29,33	99
$TEMP19^{c}$	96	50	2,30,34	100
$TEMP20^{P}=P6^{P}*A^{T}$	97	30	5,17	99
$TEMP20^{c}$	98	30	5,18	100
$f13^{P}=TEMP19^{P}-TEMP20^{P}-TEMP3^{P}*P4^{P}$	99	70	33,39,95,97	109
$f13^{c}$	100	70	34,40,96,98	109
$TEMP21^{P}=P13^{P}+b*P14^{P}$	101	50	3,31,33	107
$TEMP21^{c}$	102	50	3,32,34	108
$TEMP22^{P}=P7^{P}*A^{T}$	103	30	5,19	107
$TEMP22^{c}$	104	30	5,20	108
$TEMP23^{P}=Q*P10^{P}$	105	30	2,25	107
$TEMP23^{c}$	106	30	2,26	108
$f14^{P}=-TEMP23^{P}*P14^{P}+TEMP21^{P}-TEMP22^{P}$	107	70	33,101,103,105	109
$f14^{c}$	108	70	34,102,104,106	109
CMP	109	50	41...44,47,48,51, 52,55,56,61,62,67, 68,73,74,79,80,85, 86,89,90,93,94,99, 100,107,108	110
EXIT	110	0	109	NONE

Appendix B
TASK GRAPH PROGRAM

```
{ Author : Arindam Saha        All Rights Reserved by author              }
{ Program explained in Section 3.2 also.                                  }
{ Extensive documentation is provided with the program.                   }
{ This program maps any given task graph onto a group of processors.      }
{ It requires the vertices and the edges of the task graph as its         }
{ input and provides the schedule ,i.e., which task is to be executed     }
{ by which processor and at what time. The number of processors is a      }
{ variable. The program implements the CP/MISF (explained in chapter      }
{ three) algorithm.                                                       }

 program cpmisf(input,output);

type
 proc = record
       busy : boolean;     { Each processor is either busy or free }
     end;
        { This is the task definition }
 tas = record
       enabled : boolean;      { Task is enabled when all its predecessors    }
                               {  have been executed                          }
       assigned : boolean;     { Task is assigned to an available processor   }
       executed : boolean;     { Task has finished execution or not           }
       time : integer;         { Execution time of the task                   }
       resource : integer;     { The processor number to which it is assigned }
       starttime : integer;    { Time instant at which it starts execution    }
     end;
 matrix=array[1..110,1..110] of boolean;

var
 a,newa : matrix;  { a : adjacency matrix   newa : modified a after renumber }
 pr,prl : array[1..110] of real;  { priority lists }
 time,imsucc,newtime : array [1..110] of integer;
 i,j,k,l,v,t,p,serialtime : integer;
 filvar,filvarl : text;              { input and output data files }
 task : array[1..110] of tas;
 processor :array[1..35] of proc;
 over : boolean;
 speedup,eff : real;                 { performance indices }

{ This procedure reads the input data and initializes all the variables      }

procedure initialize;
var
 x,y,e,vl,v2 : integer;

begin
  readln(filvarl,v,e);
  for x:=1 to v do
   for y:=1 to v do a[x,y]:=false;
  for j:=1 to e do begin
    readln(filvar1,v1,v2);
    a[v1,v2]:=true;
    end;{for}

   for i:=7 to 109 do a[4,i]:=true;  { This is for the particular graph.      }
   for j:=1 to v do begin
    readln(filvar1,time[j],imsucc[j]);
   end;{for}
end;{initialize}

{ This procedure calculates the level (defined in chapter 3) of each task.  }

procedure level;
var
 b : array[1..110,1..110] of integer;
 temp,temp4 : real;
begin
  for i:=1 to v do
   for j:=1 to v do
    if a[i,j] then b[j,i]:=0 else b[j,i]:=-maxint;

  pr[v]:=time[v]; prl[v]:=time[v];
  for i:=1 to (v-1) do begin
    temp:=0.; k:=v-i;temp4:=0.;
    for j:=k to (v-1) do begin
      prl[k]:=prl[j+1] + b[j+1,k] + time[k]; { prl is used to calculate the }
                                             { critical time of the graph.  }
      pr[k]:=pr[j+1] + b[j+1,k] + time[k] + imsucc[k]/v;  { The last factor }
      { considers the effect due to the number of successors               }
      if pr[k]>temp then temp:=pr[k];
      if prl[k]>temp4 then temp4:=prl[k];
      end;{for j}
     pr[k]:=temp;prl[k]:=temp4;
  end;{for i}
end;{level}

{ This procedure renumbers the tasks according to their priorities, with    }
{ task one having the highest priority.                                     }

procedure renumber;
var
 max : real;
 newno : array[1..110] of integer;

begin
  for k:=1 to v do begin
    max:=pr[1]; j:=1;

    for i:=2 to v do begin
      if pr[i]>max then begin
        max:=pr[i];
        j:=i;
        end;{if}
      end;{for i}

     newtime[k]:=time[j];pr[j]:=-1.;
     newno[j]:=k;
   end;{for k}
{   for i:=1 to v do writeln(filvar,'New number[',i,'] = ',newno[i]);}
{ Modifying the adjacency matrix according to the new numbers        }
   for i:=1 to v do
    for j:=1 to v do newa[i,j]:=false;
   for i:=1 to v do
    for j:=1 to v do if a[i,j] then newa[newno[i],newno[j]]:=true;
end;{renumber}

{ This procedure updates the adjacency matrix when task z has finished }
{ execution.                                                            }

procedure update(z : integer);

var
m,n : integer;

begin
 for m:=1 to v do
  if newa[z,m] then begin
   newa[z,m]:=false;
   task[m].enabled:=true;
   for n:=1 to v do if newa[n,m] then task[m].enabled:=false;
  end; {then}
end; {update}

{main program}

begin
 assign(filvar1,'a:input1.dat');
 assign(filvar,'a:output30.dat');
 reset(filvar1);
 rewrite(filvar);
  writeln('Enter the order of the graph and the no. of processors');
  readln(v,p);
  initialize;
  level;
  renumber;

 for i:=1 to p do processor[i].busy:=false;{ all processors are initialized }
{ Tasks are initialized  }
 for i:=1 to v do begin
    task[i].assigned:=false;
    task[i].enabled:=false;
    task[i].executed:=false;
    task[i].time:=newtime[i];
   end; {for i}

  task[1].executed:=true ; task[v].executed:=true ;
  update(1); { Task one is the entry node }
```

```
   t:=0;
{ Initial tasks are asigned }
   for j:=1 to p do
    for i:=1 to v do if (task[i].enabled and not(task[i].assigned) and
   not(processor[j].busy)) then begin
      task[i].resource:=j;
      task[i].starttime:=t;
      task[i].assigned:=true;
      processor[j].busy:=true;
      end; {then}

   t:=1;

repeat

  for i:=1 to v do if (task[i].assigned and not(task[i].executed)) then begin
    task[i].time:= task[i].time - 1;  { finished one unit of time }
    if task[i].time=0 then begin
      task[i].executed:=true;  { task i has finished execution }
      update(i);
      processor[task[i].resource].busy:=false; { processor becomes free }
     end; {time:=0}
    end; {for i}

{ If any processor is free then we check all the enabled but not assigned }
{ tasks which can now be assigned.                                         }
for j:=1 to p do if not(processor[j].busy) then
 for i:=1 to v do if (task[i].enabled and not(task[i].assigned) and
 not(processor[j].busy)) then begin
  task[i].resource:=j;
  task[i].assigned:=true;
  task[i].starttime:=t;
  processor[j].busy:=true;
 end; {for i then}

over:=true;
for i:=1 to v do if not(task[i].executed) then over:=false;
t:=t + 1;
until (over) ; { until all tasks have been executed }

{ Outputting data }
writeln(filvar,'Critical time for this graph is = ',prl[1]);
writeln(filvar,'Total time taken = ',(t-1),' units');
writeln(filvar,'Task    Starttime     Resource');
for i:=2 to v-1 do begin
 write(filvar,' ',i,'          ',task[i].starttime,'          ',task[i].resourc
e);
 writeln(filvar);
end; {for}

{ Calculating the performance indices }
serialtime:=0;
for i:=1 to v do serialtime:=serialtime + time[i];
speedup:=serialtime/(t-1);
eff:=speedup/p;
```

```
writeln(filvar,'With ',p,' processors Speedup=',speedup,';  Efficiency=',eff);
close(filvar);
close(filvar1);
end.
```