Grant Number NAG8-093

AN INTELLIGENT PROCESSING ENVIRONMENT FOR REAL-TIME SIMULATION

by

Chester C. Carroll
Cudworth Professor of Computer Architecture
Department of Electrical Engineering
College of Engineering
The University of Alabama
Tuscaloosa, Alabama

and

Buren Earl Wells, Jr.
Graduate Research Assistant

Prepared for

National Aeronautics and Space Administration

Bureau of Engineering Research
The University of Alabama
May 1988

BER Report No. 426-17

The University of Alabama
College of Engineering
Bureau of Engineering Research
P.O. Box 1968
Tuscaloosa, Alabama 35487-1968
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 2000 students and a graduate enrollment exceeding 200. 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

AN INTELLIGENT PROCESSING ENVIRONMENT FOR REAL-TIME SIMULATION

by

Chester C. Carroll
Cudworth Professor of Computer Architecture

and

Buren Earl Wells, Jr.
Graduate Research Assistant

Prepared for

The National Aeronautics and Space Administration

Bureau of Engineering Research
The University of Alabama
May 1988

BER Report No. 426-17

ABSTRACT

This report is concerned with the development of a highly efficient and thus truly intelligent processing environment for real-time general purpose simulation of continuous systems. Such an environment can be created by mapping the simulation process directly onto The University of Alabama's OPERA architecture. To facilitate this effort this report explores the field of continuous simulation, highlighting areas in which efficiency can be improved. Areas in which parallel processing can be applied are also identified, and several general OPERA type hardware configurations that support improved simulation are investigated. The report then introduces three direct execution parallel processing environments each of which greatly improves efficiency by exploiting distinct areas of the simulation process. These suggested environments are candidate architectures around which a highly intelligent real-time simulation configuration can be developed.

LIST OF ABBREVIATIONS

ACSL	Advanced Continuous Simulation Language
B42	Fourth-Order Parallel Two Processor Block Predictor-Corrector
CISC	Complex Instruction Set Computer
ER	Euler
h	Integration Step Size or Calculation Interval
I/O	Input/Output
OPERA	Optimally Parallel Environment for Real-Time Applications
P2	Second-Order Predictor-Corrector
P4	Fourth-Order Predictor-Corrector
P22	Second-Order Parallel Two Processor Predictor-Corrector
P24	Second-Order Parallel Four Processor Predictor-Corrector
P42	Fourth-Order Parallel Two Processor Predictor-Corrector
RISC	Reduced Instruction Set Computer
R2	Second-Order Serial Runge-Kutta
R4	Fourth-Order Serial Runge-Kutta
TP	Variable-Step Trapezoidal

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Computer simulation is a major application area of digital computers, through which the behavior of physical systems can be better understood by observing how their computer models respond as the external conditions are varied. In this way physical systems can be carefully evaluated before they are actually constructed. For certain controller type applications the computer system performing the simulation is interfaced directly to the outside world through a group of sensors and actuators, with the computer acting as a substitute for the actual physical system it is simulating. In these cases, the computer simulation must be able to respond within the real-time environment of the actual physical system.

The steps needed to perform computer simulation of physical systems are shown in Figure 1.1. The first step in the simulation process is to fully understand the physical system so that it can be expressed mathematically using numerical and logical relationships. This step is usually not trivial and must be done accurately, or the simulation will be of little practical importance. The next step is to model the system by coding the mixture of numerical and logical relationships within the computer. This can be done through the use of a general purpose computer language or by using a special purpose simulation language

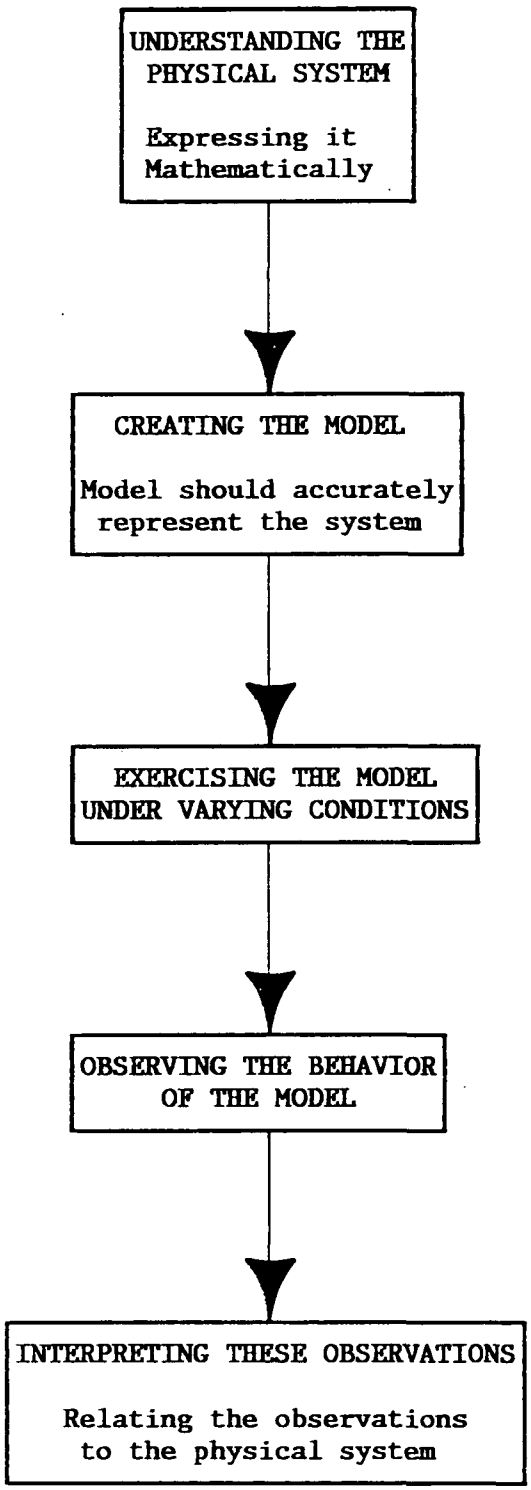


Figure 1.1 Steps in Computer Simulation

specifically designed for computer simulation applications. The next step in the simulation process is to exercise this model by applying external conditions to it representing conditions that might be applied to the actual physical system. In a controller type implementation these external conditions would originate in real-time from physical systems present in the outside world. As the model is exercised its behavior should be observed. In a controller type environment these observations result in real-time control data being transferred to the outside world to control some physical system. In a nonreal-time environment, time may be taken to interpret these results through the creation of charts and graphs. In such a way the results of the simulation can be related to the actual physical system.

Computer simulation can be divided into three different areas, continuous system simulation, discrete event simulation, and combined simulation. Continuous systems are time varying in nature and are usually modeled within the computer using sets of differential equations and/or transfer functions. In continuous simulation the dependent variables change in a continuous manner as functions of the independent variables. Discrete event simulation is concerned with the occurrence of distinct events in time and space. Examples of such simulations are stochastic processes and queuing problems. In discrete simulation the dependent variables change in a discontinuous manner as functions of the independent variables. Combined simulation allows for portions of the problem to be modeled continuously and other portions to be modeled discretely. Such simulations are in general difficult to implement.

Most physical systems can be adequately represented using continuous simulation. Thus within this report only continuous system simulation will be considered.

Today's computers can be used to simulate most physical systems accurately, allowing for much flexibility. Unfortunately, such simulations often require an excessive amount of time to execute, usually running much slower than real-time. They also tend to be very sensitive to the complexity of the physical system that is being simulated, the more complex the system the slower its simulation. This report researches the creation of intelligent processing environments leading to more efficient and therefore faster general purpose continuous simulation.

The remainder of this report is organized in the following manner. Chapter 2 describes the common features and general structure associated with most conventional general purpose continuous simulation languages. Continuous simulation languages are surveyed and described, and several popular integration algorithms are introduced. Chapter 3 is concerned with how to best apply parallel techniques to simulations written in continuous simulation languages. Three major methods in which parallel processing can be applied to continuous simulation are discussed. These include parallel derivative function evaluations, the use of parallel integration algorithms, and combining parallel derivative evaluations with parallel integration methods. Also, within this chapter several integration algorithms, parallel and serial, are compared with one another through a number of benchmark examples. Chapter 4 suggests several general hardware configurations that can be used to improve

efficiency during continuous simulation. These configurations support the direct compilation of a continuous simulation language to object code, the direct translation of the continuous simulation language to a general purpose high-level language, or the direct execution of the continuous simulation language itself. Chapter 5 discusses the design of three types of parallel continuous simulation language direct execution environments based around many of the concepts employed within The University of Alabama's OPERA computer architecture. All three of these environments represent an intelligent processing environment suitable for real-time simulation.

CHAPTER 2

CONTINUOUS SIMULATION LANGUAGES

Continuous simulation languages have been developed to support the special needs associated with the simulation process. Such languages allow attention to be focused upon the actual physical system that is being simulated, removing many of the details that have to be understood when programming is done using a more general purpose type language. They also provide a possible mechanism for improved simulation. If the hardware environment is specifically tailored around the execution of a continuous simulation language then improved simulation speed is possible. Before this can be accomplished the general execution of continuous simulation languages and principles of continuous simulations must be thoroughly understood.

2.1 Common Features

Unlike general purpose languages such as FORTRAN, continuous simulation languages were developed specifically for the purpose of allowing the simulation of complex continuous systems. Such languages have been designed to allow easy modeling of complex physical systems and provide the control mechanism needed for the simulation of such

systems. Most continuous systems can be described by time dependent nonlinear or linear differential equations and/or transfer functions. Continuous simulation languages facilitate the entry and execution of such relationships through the unique features they contain. Examples of continuous simulation languages that have been introduced recently include CSMP [1], developed by IBM; CSSL-IV [2-3], developed by Simulation Services Inc; EASY5 [4], developed by Boeing Computer Services; DARE [5], developed at the University of Arizona; COMET [6], developed at The University of Alabama; and ACSL [7-8], developed by Mitchell and Gauthier Associates. Although each of these languages differs in some way from the others, there are several important features common to each of them.

In 1967 the Society for Computer Simulations (SCS) developed a set of guidelines by which future continuous simulation languages were to be based [9]. Most of the modern continuous simulation languages adhere quite closely to these guidelines, including most of them as features of the language. The guidelines require that the structure of the language be clear, allowing for a block-oriented representation of the physical system. Furthermore, the continuous simulation language should be easy to use, having a set of operators capable of easily handling most problems that are modeled with differential equations. Since integration is a major part of the continuous simulation process, the guidelines require that the language contain a number of built-in integration routines in addition to allowing the inclusion of integration routines written by the user. The guidelines also specify that the language be expandable through user-created subroutines written in a

separate general purpose high-level language. Another requirement is that the language contain a set of prepackaged input and output routines while also allowing for the users to implement custom input and output routines of their own. The guidelines also suggest that the simulation language be able to automatically sort modeling statements to assure their proper execution order. Furthermore, the user should be able to modify all parameters between simulation runs, but not necessarily during the simulation. As with most computer languages, these continuous simulation languages are expected to have some sort of diagnostic capability to help define and locate user errors.

Many continuous simulation languages have an added number of features not mentioned in the guidelines. Some languages have a more interactive set of run-time commands that allow the user to interact in a limited way with parameters as the simulation progresses. Some allow the user to create macros within the simulation language, allowing the language to be expanded internally without the need for creating routines in a separate high-level language. Others have built-in statistical routines that can be used in comparing accuracy of simulations run under differing conditions.

2.2 General Structure

The general structure of a continuous simulation language is different from that of a typical high-level language. First, most continuous simulation languages have two basic types of commands,

modeling statements and run-time commands. The modeling statements are used to describe the physical system, and the run-time commands exercise the model allowing various parameters to be altered. The modeling statements are usually contained within a separate modeling definition file, and the run-time commands are executed in an interactive manner. The structure of the continuous simulation language can be understood by carefully examining the execution of each section within the model definition file as the simulation progresses.

Figure 2.1 shows the general structure of a Model Definition File required by the Advanced Continuous Simulation Language (ACSL) [8]. This file is similar in structure to model definition files required by a number of other continuous simulation languages. The file has three main sections, INITIAL, DYNAMIC, and TERMINAL. The INITIAL and TERMINAL sections both execute in a standard sequential manner. Program control is passed to the INITIAL section at the beginning of each simulation run, and control is passed to the TERMINAL section at the end of the run. The major portion of the simulation is spent processing the DYNAMIC section which is not executed sequentially. Instead the DYNAMIC section is processed at regular intervals throughout the simulation. Statements that reside within the DYNAMIC section but not within a subsection are executed once every communication interval. The communication interval is the amount of time that transpires between the transferring of data with the outside world. Usually the I/O statements reside within this section.

The DYNAMIC section also contains two subsections, the DERIVATIVE and the DISCRETE subsections, which will now be discussed. By far the

```
PROGRAM

    INITIAL

            Statements performed before the run begins.

    END

    DYNAMIC

        DERIVATIVE

                Statements executed at every calculation
                interval.  These statements are needed to
                perform each integration step.

        END

        DISCRETE

                Statements executed after a each specified
                time interval.  This subsection is designed
                to provide a method for communication to
                occur between the outside world and the
                simulation at fixed intervals.

        END

                Statements executed at every communication
                interval.  This usually includes the I/O
                statements.

    END

    TERMINAL

                Statements executed at the end of the
                simulation.  Section is entered when
                the termination condition becomes true.

    END

END
```

Figure 2.1 Structure of the ACSL Model Definition File

most important subsection is the DERIVATIVE subsection which contains a set of algebraic and differential relationships that make up the model. The differential relationships are entered by the user as a set of first-order differential equations defined through the use of the integration operator. These relationships are automatically sorted to insure the proper order of execution. The order that the statements are entered into the section have no bearing on the solution (this is not the case in the INITIAL and TERMINAL sections). Code within the DERIVATIVE Section is executed at least once during each calculation interval; the actual number of times being dependent on the integration algorithm chosen. The calculation interval is a variable that is specified by the user and indicates the step size of the integration process. It must be set equal to or smaller than the communication interval. As will be discussed later, some integration algorithms cause the step size to vary throughout the simulation. In such cases the limits over which the step size is allowed to vary are entered by the user.

The DISCRETE section allows communication to occur with the outside world at fixed intervals of time independent of the communication interval. This section is provided to improve the flexibility at which the simulation can be made to operate. Equivalent subsections are not present in many simulation languages.

Knowledge of the structure of simulation languages allows for the development of better hardware configurations that support the execution of such languages. For example, since the DERIVATIVE subsection is generally executed most often during the simulation, it is a reasonable

assumption that the overall execution speed of the simulation can be increased by placing this subsection in the fastest memory possible. Thus in a system with a standard memory hierarchy the DERIVATIVE section might reside in fast external cache type memory or, if cost permits, in the very fast on-chip memory of the processor. Chapter 4 describes general configurations that improve the execution of continuous simulations.

2.3 Popular Integration Methods

Continuous simulation languages allow simulation to occur through the repeated execution of the derivative type section (sometimes called the derivative function evaluation) under the direction of an integration formula. In this way continuous simulation languages are based around numerical integration routines that solve sets of first-order ordinary differential equations. Normally the simulation language provides the user with a choice of integration algorithms that can be used during the simulation process. This flexibility is provided because there is currently no one integration algorithm that works best for all types of applications.

Most simulation languages have a derivative type section composed of algebraic and differential equations that are executed a number of times during each integration step. The number of times this section is executed depends on the integration algorithm. In most cases the separate algebraic and differential relationships contained in this

section can be combined and simplified in such a way that the section can be considered to be made up only of a set of first-order differential equations. Thus the section can be described in the "state variable" form

$$\begin{aligned} \dot{Y}_1 &= F_1(Y_1,Y_2,\ldots,Y_m,t), \\ \dot{Y}_2 &= F_2(Y_1,Y_2,\ldots,Y_m,t), \\ &\ldots\ldots\ldots\ldots\ldots\ldots, \\ \dot{Y}_m &= F_m(Y_1,Y_2,\ldots,Y_m,t), \end{aligned} \tag{2.1}$$

where $Y_1,Y_2,\ldots,Y_m$ represent the state variables of the system,
$\dot{Y}_1,\dot{Y}_2,\ldots,\dot{Y}_m$ represent the rates of change with respect to time (the derivatives) of the state variables, and
t represents the independent variable, time.

Throughout the remainder of this report this "state variable" form is described using the vector notation

$$\dot{Y}_i = F(Y_i,t) = F_i, \tag{2.2}$$

where the vector $F(Y_i,t)$ or F_i represents the derivative function evaluation for the set of differential equations at integration step i. This notation simplifies the integration equations and makes it easier to describe each method of integration.

With this in mind several of the more common and popular integration methods will now be discussed.

2.3.1 The Euler Method

The Euler Integration Algorithm is by far the simplest integration method discussed within this report. It requires that the derivative

type section of the continuous simulation be executed but once during each integration step. It can be represented by

$$Y_{i+1} = Y_i + hF_i, \tag{2.3}$$

where i is the current integration step number,

h is the integration step size,

Y_i is the current state variable vector, and

Y_{i+1} is the state variable vector for the next integration step.

The amount of local error introduced with this method is large, on the order of h^2. Thus a relatively small step size is usually required to achieve a reasonable amount of accuracy. Therefore, this method is rarely used to perform continuous simulations.

2.3.2 Multistep Predictor-Corrector Methods

Multistep integration methods arrive at new solution points by considering the solutions found at other points in time. The predictor-corrector methods discussed in this section are made up of two separate multistep equations. One of these equations uses solution values found for past integration steps to arrive at a predicted solution for the next integration step. The other improves the accuracy of this solution by combining the predicted solution found in the previous equation and a certain number of solutions taken at past integration steps into a multistep corrector formula. The number of solution points considered within each of the equations represents the order of that equation. Most predictor-corrector integration schemes

contain predictor and corrector equations which are of the same order. In general, the larger the order of a predictor-corrector scheme the more computer memory will be required to store past solution points.

The Adams-Moulton predictor-corrector method makes use of derivative function evaluations, F(Y,t), that have occurred at past integration steps to arrive at new values. During each integration step only two new function evaluations must be performed regardless of the order of the algorithm. The equations for the second-order and fourth-order Adams-Moulton predictor-corrector methods are

second-order predictor-corrector equations

$$\begin{aligned} Y^p_{i+1} &= Y^c_i + h/2(3F^c_i - F^c_{i-1}), \\ Y^c_{i+1} &= Y^c_i + h/2(F^p_{i+1} + F^c_i), \end{aligned} \tag{2.4}$$

fourth-order predictor-corrector equations

$$\begin{aligned} Y^p_{i+1} &= Y^c_i + h/24(55F^c_i - 59F^c_{i-1} + 37F^c_{i-2} - 9F^c_{i-3}), \\ Y^c_{i+1} &= Y^c_i + h/24(9F^p_{i+1} + 19F^c_i - 5F^c_{i-1} + F^c_{i-2}). \end{aligned} \tag{2.5}$$

Since the predictor-corrector equations depend on a number of past values, the method is not self starting. Thus the first few data points must be calculated by some other method before the predictor-corrector routines can be implemented.

The corrector equation in a predictor-corrector pair can be used to estimate the relative accuracy of the solution for each integration step. In this way the adequacy of the step size can be determined during the simulation. If the step size is not proper it can be adjusted accordingly. Such variable-step approaches add to the complexity of the integration software and often do not perform well due to the added overhead associated with constantly changing the step size.

Another type of variable-step approach, this one based upon the trapezoidal formula, is presented in reference [10]. This approach uses a simple Euler type predictor formula to obtain an initial predictor value. This formula is

$$Y^{p}_{i+1} = Y^{c}_{i} + hF^{c}_{i}.$$

The predicted value, Y^{p}_{i+1} is then initially placed into the recursive corrector formula

$$Y^{c}_{i+1} = Y^{c}_{i} + h/2(F^{c}_{i} + F^{c}_{i+1}), \qquad (2.6)$$

for the Y^{c}_{i+1} variable. This equation is then repeatedly executed in a recursive manner with each value of Y^{c}_{i+1} being compared with the previous Y^{c}_{i+1} value. When these values differ percentage wise from each other by less than some chosen amount, then the value of Y^{c}_{i+1} is accepted and a new integration step is begun by reapplying the predictor formula. The accuracy of this integration method is dependent on the accuracy of the corrector formula and the integration step size.

2.3.3 Runge-Kutta Methods

Runge-Kutta methods comprise a popular set of single-step integration methods commonly used for continuous simulation. Because these are single-step methods, all calculations are contained within each integration step, and thus each method is self starting. Each Runge-Kutta algorithm causes a derivative function evaluation, F(Y,t), to be performed a number of times during each integration step. The number of function evaluations that occur during each step is equal to

the order of the algorithm. The equations for the second-order and fourth-order Runge-Kutta algorithm are

second-order Runge-Kutta equations

$$Y_{i+1} = Y_i + 1/2(k_1 + k_2), \tag{2.7}$$

where $k_1 = hF(Y_i,t)$,

$k_2 = hF(Y_i+k_1,t+h)$,

fourth-order Runge-Kutta equations

$$Y_{i+1} = Y_i + 1/6(k_1 + 2k_2 + 2k_3 + k_4), \tag{2.8}$$

where $k1 = hF(Y_i,t)$,

$k_2 = hF(Y_i+k_1/2,t+h/2)$,

$k_3 = hF(Y_i+k_2/2,t+h/2)$,

$k_4 = hF(Y_i+k_3,t+h)$.

The local error associated with these Runge-Kutta methods are relatively small, on the order of h^3 and h^5 for the second-order and fourth-order routines, respectively, meaning a relatively large step size can be used during the simulation. The improvement in simulation speed associated with this large step size is at least partly offset by the fact that in the higher order methods several function evaluations must be performed for each integration step. In most cases these function evaluations represent the most time consuming portion of the simulation.

With the standard Runge-Kutta equations discussed above, the local error is difficult to estimate, and it is hard to develop a variable-step approach. However, there are different variations of Runge-Kutta methods that allow for automatic step size control, such as the Merson, Verner, and Fehlberg methods [10].

2.3.4 Numeric Stability

An integration method is unstable if the accuracy of its results begins to decrease drastically as the computation proceeds. Stability depends on the properties of the algorithm and the properties of the system being simulated. In addition, most integration routines become unstable if the integration step is made too long relative to the fastest time constants of the system. The single-step Euler method (Equation 2.3) and Runge-Kutta methods (Equations 2.7 & 2.8) are very stable if the step size is made sufficiently small. The multistep predictor-corrector (Equations 2.4 - 2.6) routines are somewhat less stable. There are currently several integration routines that have been developed for improved stability at the expense of large step size accuracy [11]. Such algorithms can be used to simulate systems that exhibit qualities associated with instability, such as widely varying time constants.

CHAPTER 3

PARALLEL TECHNIQUES FOR IMPROVED SIMULATION

A logical approach to improving the execution speed of continuous simulation is to create a processing environment in which several processors work together in parallel, each on separate portions of the problem, to obtain the solution. Such an environment has the potential of simulating continuous systems several orders of magnitude faster than is possible with a totally sequential environment. In addition, performance in a parallel environment tends to be more independent of the complexity of the system that is being simulated. This is because conventional sequential environments are naturally very sensitive to the complexity of the system. As the system becomes more complex the sequential simulation time will continue to increase. Another factor favoring parallel processing is that the current state of technology has progressed to the point where implementing highly parallel computer systems is now economically feasible.

As discussed in Chapter 2 the execution of continuous simulation languages on a digital computer results from two basic processes being performed, the derivative function evaluation and the integration formula execution. Both of these lend themselves to parallel processing techniques which will result in greatly improved execution time.

3.1 Parallel Derivative Function Evaluations

During each step (calculation interval) of a continuous simulation run there is at least one derivative function evaluation that is performed. The actual number of function evaluations depends on the particular integration algorithm that is used, with some single-step algorithms requiring four or more derivative function evaluations for each calculation interval. For most applications, the function evaluation process is the most time consuming portion of the simulation. Therefore a large improvement in overall system performance can be expected if parallel processing is effectively used to improve the execution time of this portion of the simulation.

3.1.1 Partitioning by Differential Equations

Before parallel processing can be used, the function evaluation section must be broken up into a number of concurrent tasks with each task being assigned to separate processors of the system. One way this can be done, without effecting simulation accuracy, is to assign each processor a set of one or more first-order differential equations to evaluate [12]. Any number between one and N parallel processors can be used to speed up the simulation; where N is the number of first-order differential equations that make up the system. With this assignment scheme the maximum amount of parallelism possible occurs when each differential equation in the system is individually assigned to a

separate processor. As will be noted later, this maximum amount of parallelism does not always lead to the most efficient simulation.

Some continuous systems contain a set of differential equations that are a good bit more complex than the other differential equations in the system. This is especially true when a system contains a few nonlinear differential equations that are very complex or require the use of a number of system library routines (such as logarithmic or trigonometric function evaluations). In this case it is beneficial to implement another level of parallelism to further subdivide the processing of each differential equation. The University of Alabama's OPERA architecture provides a hardware environment that fully supports such two-level parallelism. More will be discussed on this topic in Chapter 5.

In the parallel processing environment just described, every time a function evaluation is performed the processors must exchange a number of state variables. This means that a certain amount of time will be lost, due to the communication delay associated with the network that interconnects the processors. This delay is highly dependent on the type and speed of the interprocessor communication network used and can include such factors as propagation delay, delay due to contention, and delay due to switching time of dynamic elements. The number of state variables that must be shared between processors depends on how closely coupled the differential equations are and how the differential equations are partitioned among the processing elements. For a given continuous system this number can be anywhere from 0 to N; where N is the number of differential equations in the system. It is desirable

to provide a balance between the amount of processing that occurs within each processing element of the system and the amount of interprocessor communication. Such a processing environment makes optimal use of parallelism and results from an intelligent allocation of differential equations to processing elements.

In cases where the number of first-order differential equations exceeds the number of processors to be used, the intelligent allocation of differential equations to processing elements is a very complex task. This allocation process can theoretically be accomplished dynamically during the simulation, or can occur statically in a preprocessing environment. Dynamic allocation has the advantage that the allocation scheme can be adjusted to reflect changing conditions imposed on the simulation by the outside world. The complexity of this allocation process, however, tends to favor static allocation, since the proper allocation of differential equations to processors can be very time consuming. There are several factors inherent in the continuous system to be simulated that are important for proper allocation. The processing time of each differential equation, the number of state variables that must be transferred between processors, and the dynamic requirements of the continuous system, must all be considered before the differential equations can be properly partitioned and assigned.

Most continuous systems are modeled using a set of differential equations that vary from each other in complexity. Since the processing time required is generally proportional to complexity, it is reasonable to assume that these differential equations will have varying execution times. If communication delay and the dynamic requirements of the system are ignored, then the optimal allocation is the one that best

balances the amount of execution time required at each processor. Even under these ideal conditions it is very difficult to determine if a particular allocation is optimal, because the list of possible assignments is often very large. Also, even an optimal assignment will most often result in a speedup of less than would be predicted by dividing the sequential processing time by the number of processors in the system. This is because it may not be possible to distribute the processing load evenly among all of the available processors.

Unfortunately, with the speed of today's interconnection networks, the interprocessor communication delay cannot be ignored. Instead it must be considered carefully to arrive at an optimal allocation. Such an allocation results in the fastest run time by partitioning the differential equations in such a way that there is an appropriate balance between the amount of processing performed by the processors and the amount of communication performed between the processors. Differential equations that describe many physical continuous systems can be easily partitioned into separate blocks that require relatively little intercommunication between each block [13]. Often such partitioning results in decreased execution time, even though the amount of processing that occurs at each processor is not balanced.

There exists a general class of physical systems called delay-line models that are especially easy to partition in a way that allows a balanced amount of processing and only a small amount of interprocessor communication. In such systems, only a few state variables need to be passed between adjacent processors. This means a relatively simple interconnection network such as a nearest neighbor mesh or systolic

array [14-15] can be implemented. These delay-line systems include such real-world phenomenons as transmission line analysis, blood circulation, fluid transport phenomena, pollution diffusion in river systems, irrigation systems, and sewers [16]. Of course, a general purpose continuous simulation language system should not be restricted to just these applications. Therefore a more general type of interconnection scheme seems more appropriate.

Some physical systems are modeled using a set of differential equations in which certain sections change with time much more rapidly than others. In other words, the dynamics of the system are such that several sections have greatly differing time constants. Partitioning in such "stiff" systems can be performed in such a way that the fast sections are processed by separate processors. On processors that are processing these fast sections a different integration formula can be used [13]. Since this integration formula is better suited to process the fast sections than the general purpose one used by the other processors, the calculation interval can remain relatively large. This effectively decreases the amount of processing time required for accurate simulation.

Unfortunately, partitioning the system of differential equations on the basis of system dynamics alone cannot be expected to result in balanced processing or small interprocessor communication time [16]. There is usually a point reached where the benefits associated with partitioning by system dynamics are overshadowed by that obtained by balancing processing load with interprocessor communication.

Currently there is much research worldwide into the development of allocation algorithms that provide optimal or near optimal process

allocation [17-21]. Some of these algorithms work well in cases where the number of processors is relatively small, but not so well in larger cases. Other algorithms are centered around interconnection structures that are not well suited for general purpose continuous system simulation. Research in this area is just in its beginning stages, and a major breakthrough might occur at anytime. Until then, allocation of differential equations to processing elements will have to be performed using algorithms that have a limited scope. Perhaps the user can be allowed to choose from several allocation algorithms that are made available as part of the continuous simulation language. The user can then select an allocation algorithm in much the same manner as the integration algorithm is currently selected. This would give the user the flexibility to try several allocation schemes before choosing the one that is more suitable to the particular application.

3.1.2 Guided Missile Example

To illustrate the improvement possible by applying parallel processing to derivative function evaluations, consider the Optimal Control of Guided Missile Example. This example is a variation of a well known control type problem which has been presented in a number of publications [22-25]. The Advanced Continuous Simulation Language (ACSL) source code representation of this example is shown in Figure 3.1. The example is modeled with fourteen first-order ordinary differential equations which are tightly coupled. These equations, along with the initial conditions, are shown in Figure 3.2.

```
PROGRAM GUIDED MISSILE

      INITIAL
            CONSTANT TEND=1
            CONSTANT A=-9,B=17,G=0.5,QT=3
            CONSTANT SS1=14.5

            MINTERVAL=1.0E-7
            MAXTERVAL=1.0
      END

      DYNAMIC
            DERIVATIVE
                  P1DOT=-G*P4*P4
                  P2DOT=P1-G*P4*P7
                  P3DOT=P2+A*P4-G*P4*P9
                  P4DOT=P3+B*P4-G*P4*P10
                  P5DOT=2*P2-G*P7*P7
                  P6DOT=P3+P5+A*P7-G*P7*P9
                  P7DOT=P6+B*P7+P4-G*P7*P10
                  P8DOT=2*P6+2*A*P9-G*P9*P9
                  P9DOT=P8+B*P9+P7+A*P10-G*P9*P10
                  P10DOT=2*P9+2*B*P10-G*P10*P10
                  P11DOT=-P2*QT-G*P4*P14
                  P12DOT=P11-P5*QT-G*P7*P14
                  P13DOT=P12+A*P14-P6*QT-G*P9*P14
                  P14DOT=P13+B*P14-P7*QT-G*P10*P14

                  P1=INTEG(P1DOT,SS1)
                  P2=INTEG(P2DOT,0)
                  P3=INTEG(P3DOT,0)
                  P4=INTEG(P4DOT,0)
                  P5=INTEG(P5DOT,0)
                  P6=INTEG(P6DOT,0)
                  P7=INTEG(P7DOT,0)
                  P8=INTEG(P8DOT,0)
                  P9=INTEG(P9DOT,0)
                  P10=INTEG(P10DOT,0)
                  P11=INTEG(P11DOT,0)
                  P12=INTEG(P12DOT,0)
                  P13=INTEG(P13DOT,0)
                  P14=INTEG(P14DOT,0)

                  TERMT(T .GT. TEND)
            END
      END
END
```

Figure 3.1 ACSL Source Code for Guided Missile Example

Constants:

A=-9,B=17,G=0.5,QT=3

Set of Differential Equations:

$$\dot{P1} = -G(P4)(P4)$$

$$\dot{P2} = P1 - G(P4)(P7)$$

$$\dot{P3} = P2 + A(P4) - G(P4)(P9)$$

$$\dot{P4} = P3 + B(P4) - G(P4)(P10)$$

$$\dot{P5} = 2(P2) - G(P7)(P7)$$

$$\dot{P6} = P3 + P5 + A(P7) - G(P7)(P9)$$

$$\dot{P7} = P6 + B(P7) + P4 - G(P7)(P10)$$

$$\dot{P8} = 2(P6) + 2(A)(P9) - G(P9)(P9)$$

$$\dot{P9} = P8 + B(P9) + P7 + A(P10) - G(P9)(P10)$$

$$\dot{P10} = 2(P9) + 2(B)(P10) - G(P10)(P10)$$

$$\dot{P11} = -P2(QT) - G(P4)(P14)$$

$$\dot{P12} = P11-P5(QT)-G(P7)(P14)$$

$$\dot{P13} = P12 + A(P14) - P6(QT) - G(P9)(P14)$$

$$\dot{P14} = P13 + B(P14) - P7(QT) - G(P10)(P14)$$

Initial Conditions:

P1=14.5,
P2=P3=P4=P5=P6=P7=P8=P9=P10=P11=P12=P13=P14=0

Figure 3.2 Set of Differential Equations for Guided Missile Example

Before a comparison can be made between the serial and parallel execution of the derivative function evaluations, certain performance measures must be defined. These measures include the speed-up ratio, utilization, cost, and effectiveness, all of which are described in reference [26].

The speed-up ratio, Sp, for a parallel implementation in comparison with serial implementation is defined by

$$Sp = Ts/Tp, \tag{3.1}$$

where Ts is the serial execution time, and
Tp is the parallel execution time.

It is always desirable to have a speed-up ratio much greater than one. The ideal speed-up ratio is equal to the number of processors in the system.

The efficiency or utilization, Ep, of a parallel implementation is

$$Ep = Sp/p, \tag{3.2}$$

where Sp is the speed-up ratio, and
p is the number of processors.

This is the proportion of time that the least productive processor in the system is busy carrying out useful calculations.

The relative cost, Cp, of the parallel implementation is defined by

$$Cp = p*Tp, \tag{3.3}$$

where p is the number of processors, and
Tp is the parallel execution time.

This relative cost should be compared with the serial execution time. Ideally, the cost should remain constant regardless of the number of processors that are in the system. In most practical situations, however, the relative cost of a parallel implementation will continue to rise as the number of processors increases.

The relative effectiveness, REp, of the parallel implementation is defined by

$$REp = (Sp)*(Ep), \tag{3.4}$$

where Sp is the speed-up ratio, and
Ep is the efficiency or utilization.

If the speed-up ratio and efficiency are of equal importance, then the relative effectiveness is a good figure of merit to use to determine the optimal number of processors needed for the parallel implementation.

For cases where the Euler integration algorithm is selected to perform continuous simulation, the total execution time can easily be described by a set of algebraic equations. (The Euler integration algorithm is considered here due to its simplicity; similar equations can be derived to determine the execution time for other integration methods.) These equations take into account the communication delay of the interprocessor communication network, and are valid provided the following assumptions concerning the parallel processing environment are true:

(1) The communication network allows for general broadcast type communication to occur between processors.

(2) Only one broadcast type message can occur at a given time throughout the network.

(3) No interprocessor communication can occur when processing is occurring within any of the processing elements.

Using these assumptions the processing time, TPi, required for each processor i is given by

$$TPi = (TD1+...+TDj) + j*2 + 1, \tag{3.5}$$

where j is the number of differential equations allocated to processor i, and

TD1+...+TDj is the total processing time for the differential equation(s) allocated to processor i.

Therefore the total parallel execution time, Tp, is given by

$$Tp = \max(TP1,\ldots,TPi,\ldots,TPn) + Sv*cd, \qquad (3.6)$$

where TP1,...,TPi,...,TPn represent the set of processing times for each processor i of the system,

Sv is the number of state variables that must be passed between processing elements, and

cd is the communication delay associated with the network.

These performance measures can now be used to better understand the merits of parallel derivative function evaluations during the simulation of the Guided Missile Example.

In investigating this example, the number of floating point operations (additions, subtractions, multiplications, and divisions) occurring in each of the fourteen differential equations of Figure 3.2 were individually totaled. For simplicity each floating point operation was given an equal weight. These values were then used as rough estimates of the relative time required to execute each differential equation.

Before this example was simulated, the allocation of the differential equations to processing elements was performed using a very primitive software routine that is contained in Appendix A. As the number of processors in the system was varied from two to twelve, the routine produced an allocation based on balanced processing time. In each case, the allocation produced was the one with the best run time of the approximately 60,000 random allocations tried. In the cases of systems with thirteen and fourteen processors, an optimal allocation was manually performed. A list of these allocations also appears in Appendix A. For each case, the communication delay, cd, of the interprocessor communication network was allowed to take on four distinct values. The communication delay was measured in relative

terms, as a multiple of floating point operations. Substituting these selected values of communication delay into the equations discussed previously, four tables of performance measurements were created (see Appendix A).

From the data in these tables, the performance measurements are charted on separate graphs as the number of processors is allowed to vary from one to fourteen. Within each graph, separate performance curves are plotted for interprocessor communication delays of 0.0, 0.5, 1.0, and 3.0 floating point operations, respectively. Each of these graphs will now be discussed.

Figure 3.3 shows the speed-up ratio versus the number of processors for the Guided Missile Example. Notice for all curves on the graph, that as the number of processors increases, the speed-up ratio also tends to increase but at a diminishing rate. In the case where communication delay is ignored (cd=0.0), the maximum speed-up ratio of 9.73 occurs in systems that have thirteen or fourteen processors. Since it is unlikely that the allocations made on this graph are all optimum, there is a chance that the maximum speed-up ratio of 9.73 could also occur in systems that use ten, eleven or twelve processors if a better allocation can be found. (It would take at least ten processors to obtain a speed-up ratio of 9.73, since the maximum speed-up ratio possible for a system is equal to the number of processors in that system.) In cases where the communication delay does not equal zero, the graph shows that the speed-up ratio is reduced by a considerable amount. In the case where the communication delay is the largest (cd=3.0), the speed-up ratio peaks at about 2.0. If the communication delay were increased much beyond this point, the speed-up ratio would be less than 1.0 and approach zero as the number of processors increases.

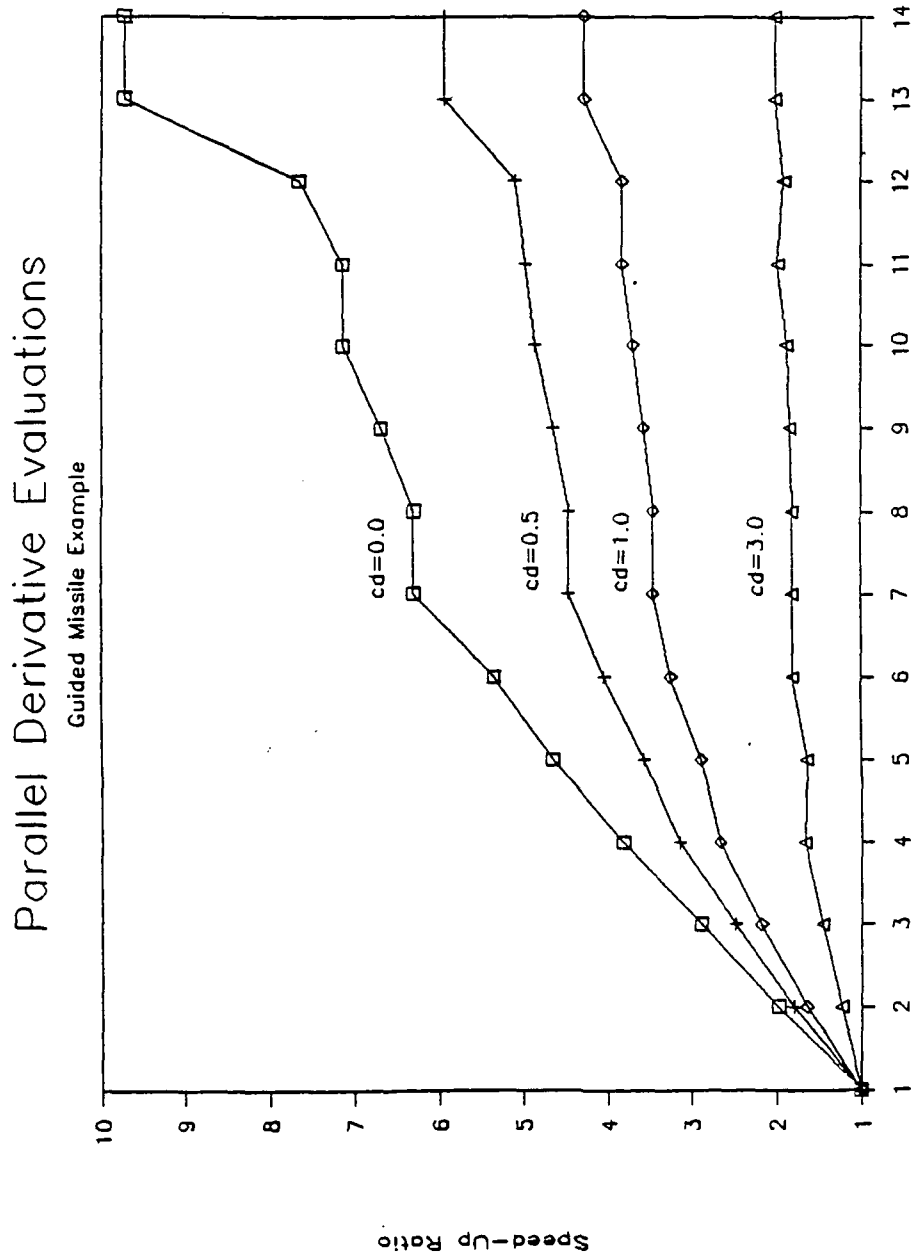


Figure 3.3 Speed-Up Ratio versus Number of Processors

This tends to illustrate the importance of having the fastest possible interprocessor communication network, and the need to consider interprocessor communication time during the allocation process. An allocation that considers interprocessor communication time tends to minimize the use of the communication network and is therefore less sensitive to the speed of such a network.

Figure 3.4 shows the relative "cost" of each parallel implementation as the number of processors is varied. With a few exceptions this "cost" tends to increase on the given system as the number of processors increases. The exceptions to this rule are caused by a more efficient allocation of the differential equations to the processing elements. The graph shows that relative "cost" is greatly affected by the amount of communication delay present in the network. As the communication delay is increased, this "cost" can skyrocket.

Figure 3.5 illustrates the utilization or efficiency that occurs under the current allocation, as the number of processors is varied. This utilization can be thought of as the proportion of time that the least productive processor in the system is busy carrying out useful calculations. As might be expected, the graph shows that for each case, the utilization tends to decrease as the number of processors is increased. There are a few exceptions to this, caused again by the more efficient allocation of the differential equations to the processors. The effect of increasing communication delay in the network is also shown in the graph. As the communication delay increases, there is a sharp decrease in utilization present in the system.

In Figure 3.6 the effectiveness is plotted with respect to the number of processors used. This performance measure can be used to find

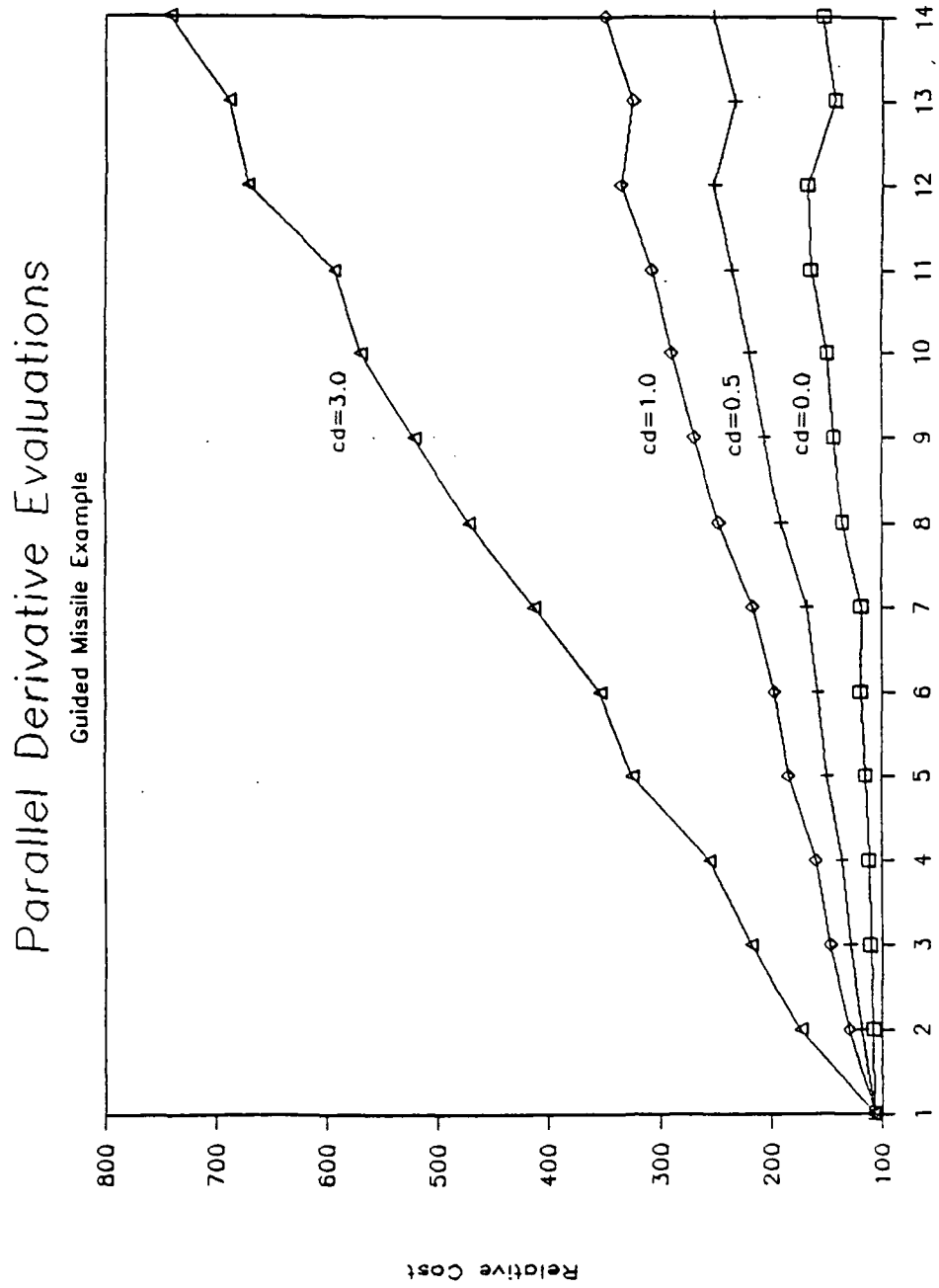


Figure 3.4 Relative Cost versus Number of Processors

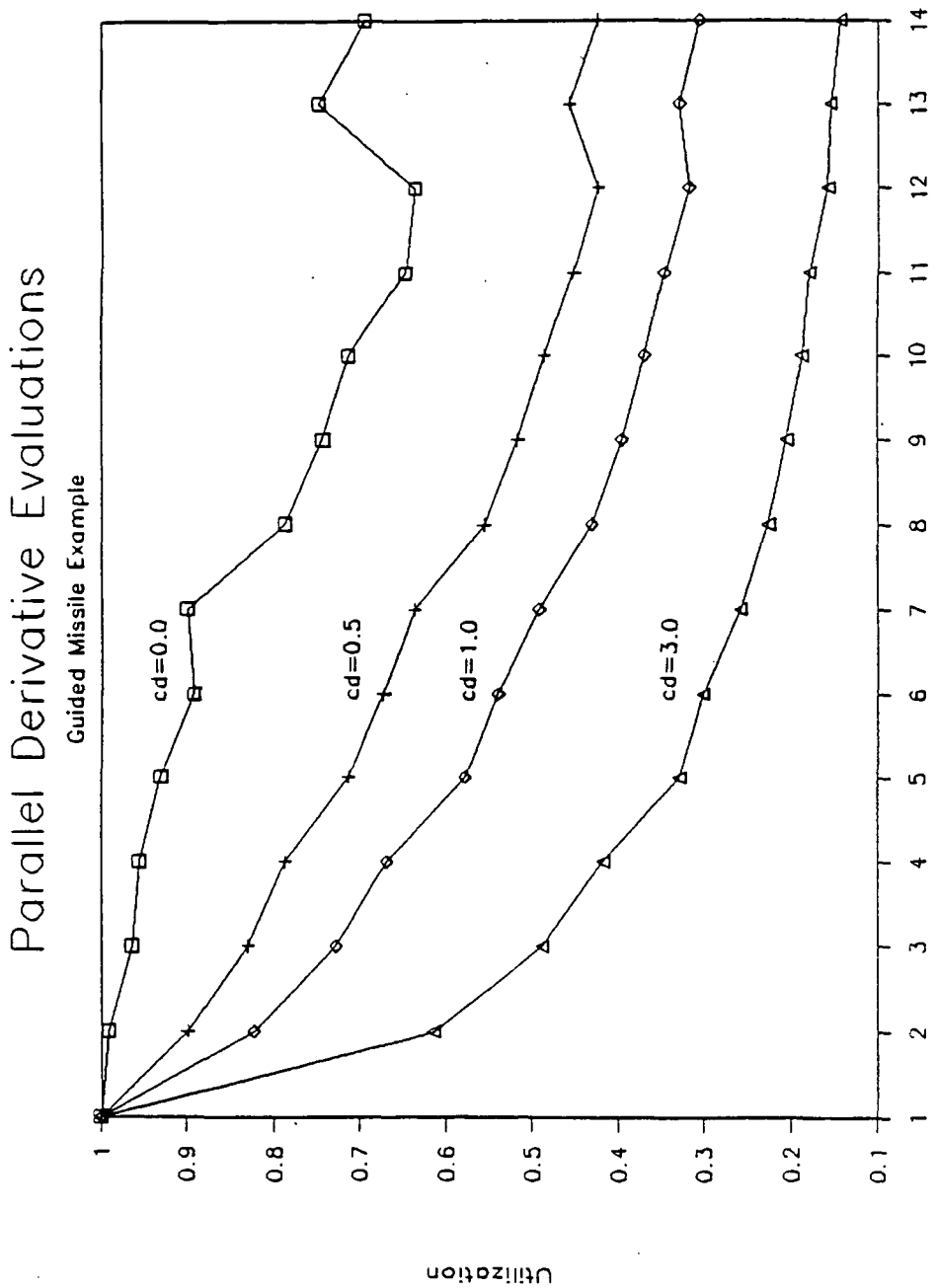


Figure 3.5 Utilization versus Number of Processors

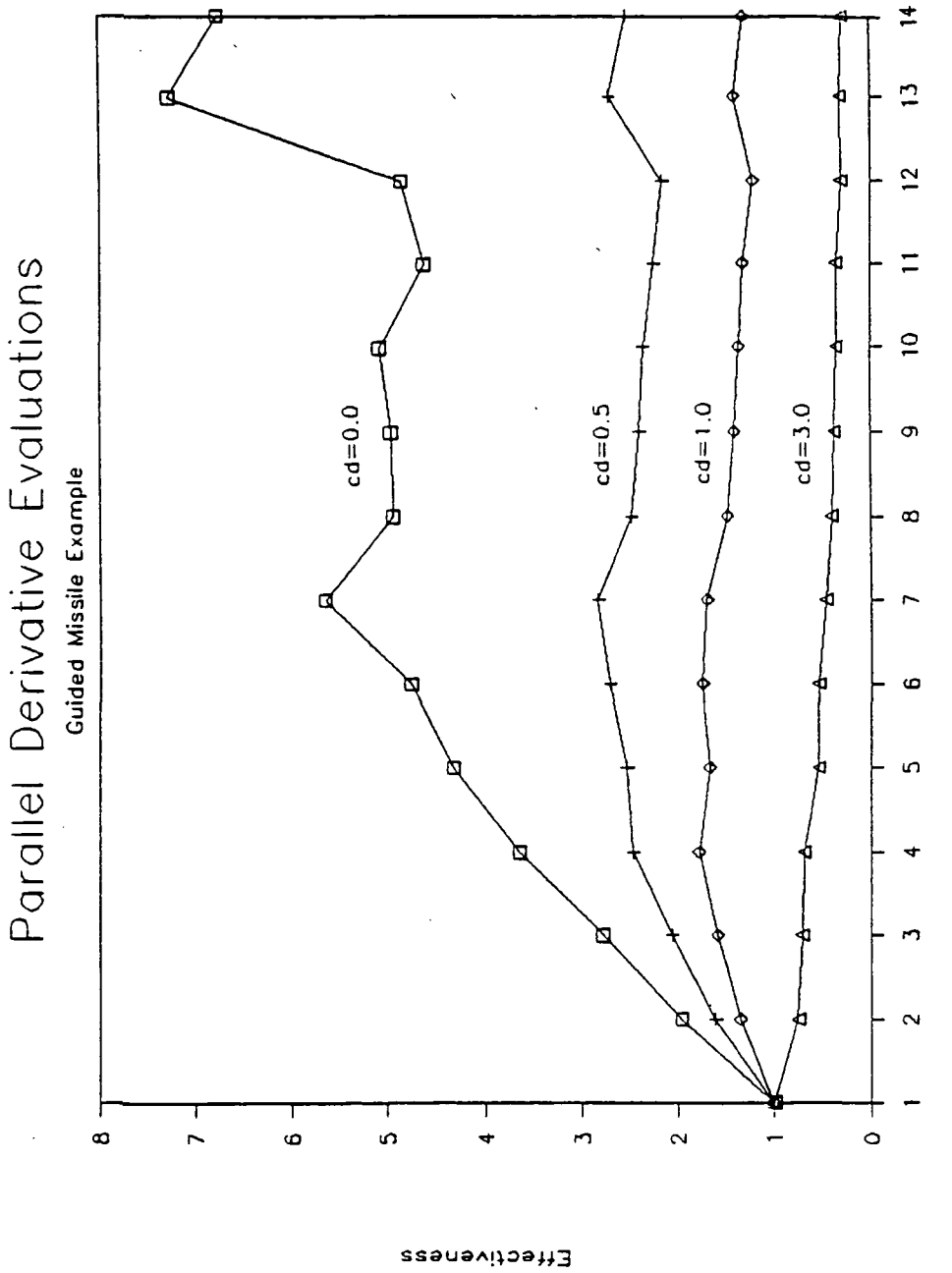


Figure 3.6 Effectiveness versus Number of Processors

the optimal number of processors if both the speed-up ratio and the utilization are considered to be of equal importance. In the case where the communication delay is zero (cd=0.0), the maximum effectiveness occurs when thirteen processors are used. Notice that when the communication delay is not zero, the point at which the maximum effectiveness occurs changes. In the case where the communication delay equals 0.5, the maximum effectiveness occurs when seven processors are used. When the communication delay is 1.0, the maximum effectiveness occurs with four processors, and when the communication delay is 3.0, the maximum effectiveness occurs with one processor.

3.1.3 Partitioning by Low-Level Tasks

Another approach to the parallel execution of derivative function evaluations is to partition the function, not at the differential equation boundary, but rather into lower-level concurrent tasks. These low-level tasks can then be assigned to separate processors for parallel execution. With this allocation scheme it is possible to implement a higher degree of parallelism than can often be done in cases where the smallest unit of allocation is the differential equation. The amount of processing performed by each processor in the system is also more evenly distributed.

Much research has been undertaken into the general area of decomposing equations into a set of concurrent tasks [27-29]. A major problem stems from the fact that most of the methods that have been

developed produce such finely grained tasks, that if each task is executed on a separate processor the communication delay negates any possible performance advantage. Allocating several of these finely grained tasks to each processor in the system is one approach to reduce this communication delay. Unfortunately, this is not always effective, and is most often a very complex process. Another problem is that few of these decomposition strategies have been converted into software, which tends to illustrate their relative complexity.

3.2 Parallel Integration Algorithms

The major problem with the parallel execution of derivative function evaluations is the difficulty associated with properly allocating and partitioning the problem in an optimal manner. Because the efficiency of execution is highly dependent upon how the partitioning is performed, it is hard to predict the amount of performance gain that can be expected without carefully analyzing each application.

A more general approach would be to improve performance by making use of parallel integration algorithms. With this approach, parallelism results from the nature of the algorithm itself, not by parallel operations being performed within the set of differential equations. Such an approach allows for a more predictable and application independent implementation, which is much easier to realize. It does not, however, allow for the ability to take advantage of the decreased interprocessor communication that occurs in lightly coupled systems.

3.2.1 Parallel Predictor-Corrector Algorithm

A parallel variation of the classical Adam-Moulton predictor-corrector algorithm has been presented by Miranker and Liniger [30]. This algorithm allows for processing to be divided among an even number of processing elements, resulting in half the processing elements processing predictor equations and the other half processing corrector formulas. Each processing element processes a different integration step, with the predictor processors always processing integration steps that are ahead of the corrector processors. During every integration step, each processor will perform one derivative function evaluation. The set of differential equations are assumed to reside within the local memory of each of the processors in the system, otherwise contention will result.

As with the serial Adam-Moulton method, this parallel algorithm is not self starting. Before the simulation begins, the first few integration steps have to be performed using another integration algorithm. This algorithm can be expanded to incorporate any even number of processing elements, but only the two and four processor cases will be presented here.

The equations for the second-order and fourth-order two processor predictor-corrector algorithms are

second-order two processor predictor-corrector equations

$$\begin{aligned} Y^{p}_{i+1} &= Y^{c}_{i-1} + 2hF^{p}_{i}, \\ Y^{c}_{i} &= Y^{c}_{i-1} + h/2(F^{p}_{i} + F^{c}_{i-1}), \end{aligned} \tag{3.7}$$

fourth-order two processor predictor-corrector equations

$$\begin{aligned} Y^{p}_{i+1} &= Y^{c}_{i-1} + h/3(8F^{p}_{i} - 5F^{c}_{i-1} + 4F^{c}_{i-2} - F^{c}_{i-3}), \\ Y^{c}_{i} &= Y^{c}_{i-1} + h/24(9F^{p}_{i} + 19F^{c}_{i-1} - 5F^{c}_{i-2} - F^{c}_{i-3}). \end{aligned} \tag{3.8}$$

A simple flow diagram of the second-order and the fourth-order two processor cases are shown in Figures 3.7a and 3.7b. Notice in both cases, one processor executes the predictor equations, and the other executes the corrector equations. As would be expected the predictor processor is one integration step ahead of the corrector processor. At the end of each integration step, data is transferred between the processors. This data includes the predicted derivative values (F^{p}), the corrected state variables (Y^{c}) and, in the fourth-order case, the corrected derivative values (F^{c}); all of which are vectors of size equal to the number of differential equations in the system. At the end of every step, each processor's internal time variable is incremented by one calculation interval, and the next processing step is begun.

The equations for the second-order four processor case are

predictor equations

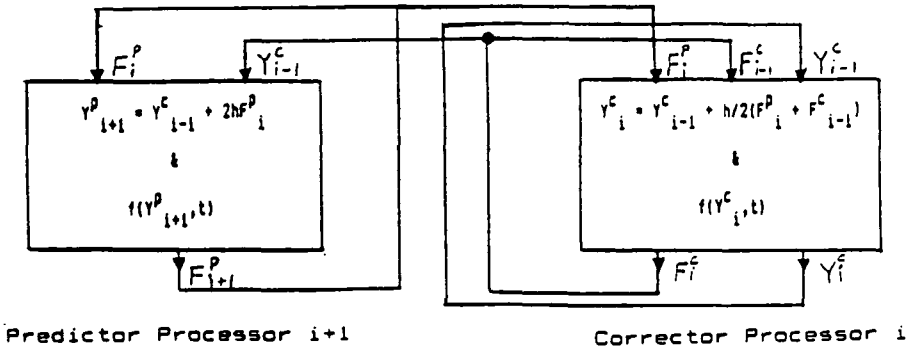
$$\begin{aligned} Y^{p}_{2i+2} &= Y^{c}_{2i-2} + 4hF^{p}_{2i}, \\ Y^{p}_{2i+1} &= Y^{c}_{2i-2} + 3h/2(F^{p}_{2i} + F^{p}_{2i-1}), \end{aligned}$$

corrector equations

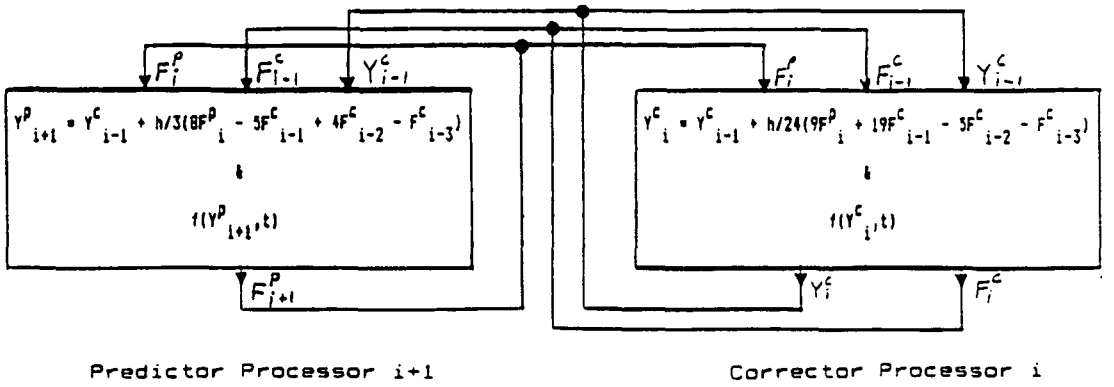
$$\begin{aligned} Y^{c}_{2i} &= Y^{c}_{2i-3} - h/2(3F^{p}_{2i} - 9F^{p}_{2i-1}), \\ Y^{c}_{2i-1} &= Y^{c}_{2i-3} + 2hF^{c}_{2i-2}. \end{aligned} \tag{3.9}$$

The flow diagram for this four processor case is shown in Figure 3.7c. Notice that two processors execute predictor equations, and the other two execute the corrector routines. One of the predictor

(a) 2nd-Order Parallel Predictor-Corrector Method
(Two Processor Case)



(b) 4th-Order Parallel Predictor-Corrector Method
(Two Processor Case)



(c) 2nd-Order Parallel Predictor-Corrector Method
(Four Processor Case)

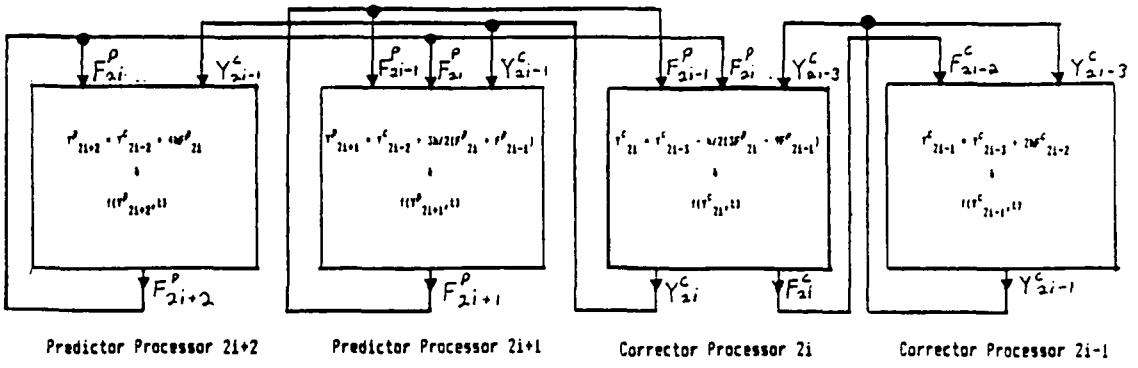


Figure 3.7 Flow Diagram for Parallel Predictor–Corrector Method

processors executes one integration step ahead of the other predictor processor and, in a similar way, one of the corrector processors executes one integration step ahead of the other corrector processor. Both predictor processors execute ahead of the two corrector processors. At the end of each processing step, data is transferred between the processors. This data includes the predicted derivative values (F^P's), the corrected state variables (Y^C's), and the corrected derivative values (F^C's); again all of which are vectors whose size is equal to the number of differential equations in the system. At the end of every processing step, each processor's internal time variable is incremented by two calculation intervals, and the next processing step is begun.

The speed-up ratio associated with these methods can be easily calculated. In the two processor cases (Equations 3.7 & 3.8) there is a theoretical speed-up ratio of two over the sequential predictor-corrector methods (Equations 2.4 & 2.5), since the predictor and corrector equations are now being processed concurrently. In the four processor case (Equation 3.9), there is a theoretical speed-up ratio of four, since in addition to spreading out the predictor and corrector calculations between separate processors, state variable calculations are being made for two integration steps at a time. This of course assumes no interprocessor communication delay. For the two processor cases, there are two or three vectors each of size N; where N is the number of differential equations in the system that must be transferred between processors. In the four processor case there are five such vectors. This means that there is the potential for a large amount of communication delay, if the number of differential equations

is large. This can be largely overcome by carefully designing the system architecture around the integration algorithm.

Unlike the case where parallelism occurs through parallel execution of the derivative functions, using a parallel integration algorithm can have an effect on simulation accuracy and stability. According to Miranker and Liniger [30] the error characteristics of the parallel predictor-corrector algorithm are comparable to that of the serial method. To confirm this conclusion both the serial and parallel algorithms were applied to a number of benchmark examples (see Appendixes B and C). The two processor predictor-corrector methods produce results that have approximately the same accuracy as the same order serial method. With the four processor predictor-corrector method the accuracy of the results decreases somewhat for each benchmark. This causes concern about the practicality of expanding the number of processors with this algorithm indefinitely. Furthermore, the stability observed in these benchmarks tends to decrease as the number of processors in the system is increased. The serial algorithm provides the most stability, and the four processor parallel algorithm is least stable. Whether this is a serious problem depends on such factors as the stability of the system being simulated and the required accuracy.

3.2.2 Parallel Block Predictor-Corrector Algorithm

Another parallel predictor-corrector algorithm is the block implicit algorithm presented by Shampine and Watts [31-32]. This

algorithm requires that time be divided into series of blocks, where each block is in turn subdivided into a number of steps of time. The predictor equations for each step within a block can then be concurrently processed, since they only require values from previously executed blocks. Derivative function evaluations at each step can also be executed concurrently. At this point, data must be transferred between the predictor portion to the corrector portion of the block. The corrector equations for each step and the derivative function evaluations can then be concurrently executed. With this algorithm, data is transferred between tasks twice during each block, once in the middle of the block when data is shared between different predictor and corrector steps, and once at the end of the block.

As with the other predictor-corrector algorithms the block method is not self starting. Before the simulation begins, the first few integration steps have to be performed using some other integration algorithm.

The equations for the fourth-order two-step parallel block predictor-corrector algorithm are

predictor equations

$$Y^p_{i+1} = 1/3(Y^c_{i-2} + Y^c_{i-1} + Y^c_i) + h/6(3F^c_{i-2} - F^c_{1-1} + 13F^c_i),$$

$$Y^p_{i+2} = 1/3(Y^c_{i-2} + Y^c_{i-1} + Y^c_i) + h/12(29F^c_{i-2} - 72F^c_{1-1} + 79F^c_i), \qquad (3.10)$$

corrector equations

$$Y^c_{i+1} = Y^c_i + h/12(5F^c_i + 8F^p_{i+1} - F^p_{i+2}),$$

$$Y^c_{i+2} = Y^c_i + h/3(F^c_i + 4F^p_{i+1} + F^p_{i+2}).$$

Figure 3.8 represents a flow diagram for this case. Notice that there are two time steps, i+1 and i+2, that are being processed during each simulation block. Both time steps are first processed by separate predictor equations and then by separate corrector equations. Derivative function evaluations occur directly after processing by each equation. Since the two predictor and two corrector equations are completely independent from each other, separate processors can be assigned to process the two time steps. As Figure 3.8 shows, between the predictor and the corrector sections, predicted data is transferred between processors. This data is in the form of the F^{p}_{i+1} and the F^{p}_{i+2} vectors which are each of size N; where N is the number of differential equations in the system. After the corrector sections have completed their execution, the corrector data must be transferred between processors, this time to start a new simulation block. The corrector data that is transferred during this time are the Y^{c}_{i+1}, F^{c}_{i+1}, Y^{c}_{i+2}, and the F^{c}_{i+2} vectors, each of which are also of size N; where N is again equal to the number of differential equations in the system. At the end of each block, both processor's internal time variables are incremented by two calculation intervals, and the execution of the next block is begun.

With this two-step parallel algorithm there is a theoretical speed-up ratio of two over the same order sequential predictor-corrector method (Equation 2.5). This is because processing is performed for two integration steps at a time. This of course is assuming the ideal case of no interprocessor communication delay. During each block there are a total of six vectors that must be transferred between the processors.

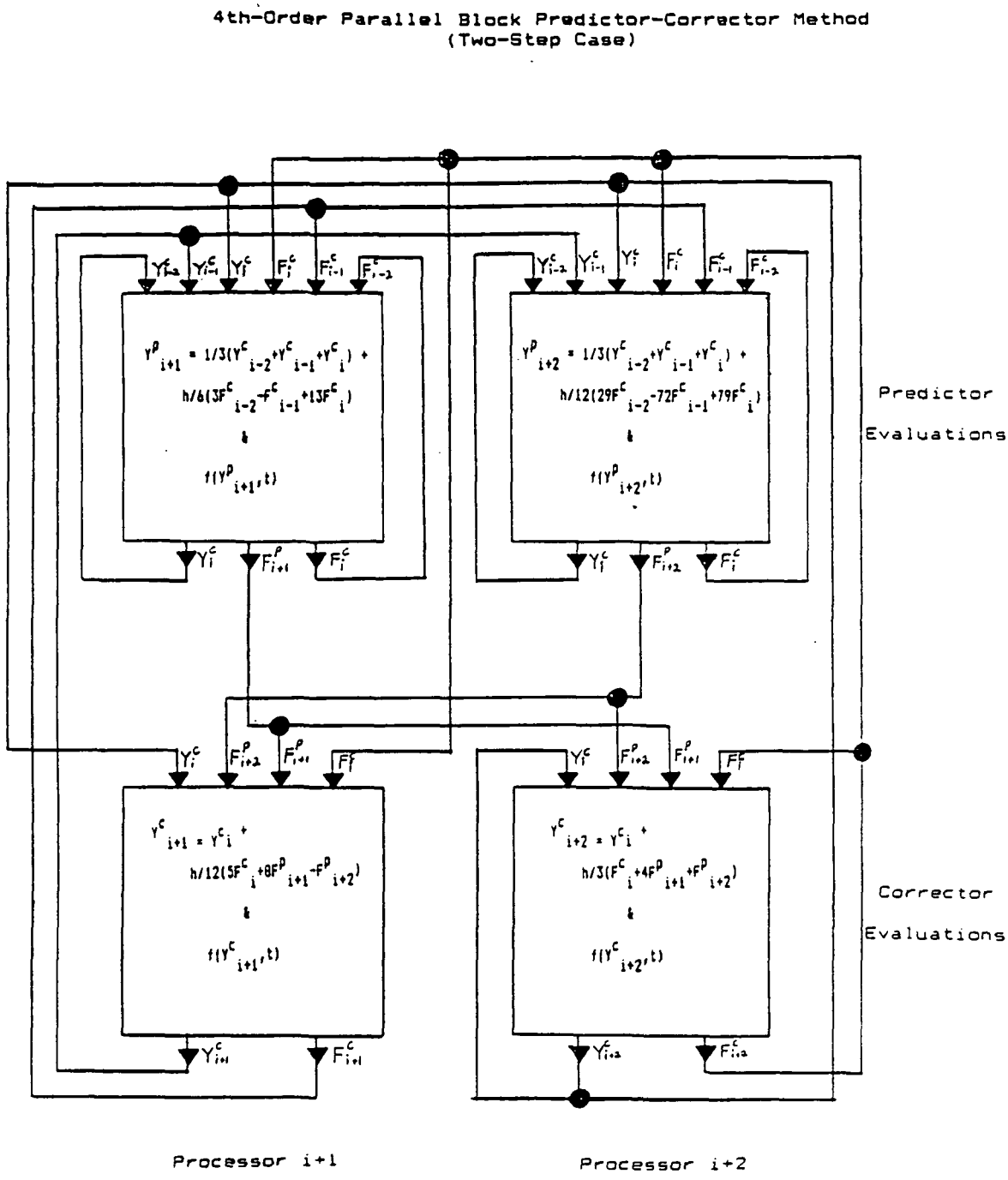


Figure 3.8 Flow Diagram for Parallel Block Predictor–Corrector Method

This compares to the three vectors that must be transferred when using the parallel two processor predictor-corrector algorithm presented in Equation 3.8. The actual amount of interprocessor communication that occurs in both instances is actually the same, since the two-step block method produces values for two time intervals, whereas the other method processes data for only one. As has been discussed previously, the adverse effect of interprocessor communication delay can be negated somewhat by carefully mapping the parallel computer architecture directly to the integration algorithm.

The accuracy of the block method compares closely with that of the serial predictor-corrector algorithm for each of the four benchmark examples contained in Appendix C. The stability was found to be less than the serial method, but greater than the four processor predictor-corrector algorithm described by Equation 3.9. The stability of this two-step block method generally compares favorably with that of the two processor predictor-corrector method of Equation 3.8. In exploring the benchmark examples, the stability of the block method was found to be equal to, or only slightly less than, that of the two processor predictor-corrector method.

3.2.3 Parallel Taylor Series Algorithm

Another parallel integration approach is introduced through the EMPRESS project of ETH in Zurich, Switzerland [33-34]. This project centers around the development of a parallel simulation environment using the the Taylor Series Expansion as the means of integrating

differential equations. This algorithm uses symbolic analytic recursion techniques to calculate the higher derivatives that are necessary for acceptable accuracy. The algorithm has been proven very well suited to parallel processing despite its heavy use of recursion, and the resulting method seems to work well for a number of engineering type applications.

In the EMPRESS project, the algorithm is processed in an environment where the processors execute under either a master or slave status. Synchronization is maintained by a centralized device called the job control unit that is also responsible for assigning the processors their master or slave status.

This algorithm is difficult to explain and depends heavily on the type of implementation. Therefore it will not be discussed further.

3.2.4 Parallel Runge-Kutta Algorithm

A parallel version of the Runge-Kutta algorithm is also presented by Miranker and Liniger [30]. This algorithm is inherently unstable, leading to greater error as the integration step size is made smaller. For this reason, the algorithm is not considered further.

3.2.5 Integration Algorithm Comparison

To gain a practical perspective into the continuous simulation process, six serial and four parallel integration algorithms were

applied to four benchmark examples (see Appendixes B and C). The integration algorithms were implemented in the form of computer programs written in the C language. Each algorithm was then compiled into a separate file. In a similar manner, the four benchmark examples were also coded in the C programming language and compiled into separate files. Then each one of the algorithm files was linked together with each of the benchmark files, allowing the ten integration routines to be applied to all four benchmarks. The programs were written in such a way that the total number of derivative function calls, the total number of floating point operations, and the maximum local error that occurred during the simulation run, were all calculated and reported. In the cases of the parallel algorithms, the effective number of function calls and floating point operations were reported, taking into account the parallelism of the algorithm. Through these measures the various algorithms were compared.

The six serial integration algorithms that were implemented were the Euler (Equation 2.3), the variable-step trapezoidal (Equation 2.6), the second-order Runge-Kutta (Equation 2.7), the fourth-order Runge-Kutta (Equation 2.8), the second-order Adam-Moulton predictor-corrector (Equation 2.4), and the fourth-order Adam-Moulton predictor-corrector (Equation 2.5) algorithms. The parallel algorithms were the second-order parallel predictor-corrector two processor case (Equation 3.7), the fourth-order parallel predictor-corrector two processor case (Equation 3.8), the second-order parallel predictor-corrector four processor case (Equation 3.9), and the fourth-order two-step parallel block predictor-corrector (Equation 3.10) methods. The continuous systems that served as benchmarks were the Spring Dashpot System [35], Orbiting Maneuvering Vehicle [36], Pilot

Ejection [8], and the Optimal Control of Guided Missile [21-25] Examples. Each of these benchmarks appears in Appendix C.

In applications where the number of differential equations is large or exceedingly complex, the dominant amount of time during the simulation will be spent performing derivative function evaluations. In these cases a good indication of execution time is the total number of equivalent function evaluations that are to be performed. If accuracy is not considered, then for a given calculation interval the integration algorithms will have the following relative execution times. The four processor predictor-corrector algorithm will have the fastest execution. This is followed by the serial Euler method along with the block and the two processor parallel predictor-corrector routines, each of which have the same execution time. The serial predictor-corrector algorithm and the second-order Runge-Kutta algorithm then follow, each with similar execution times. The slowest case is the fourth-order Runge-Kutta with an execution time of eight times slower than the fastest method--the four processor predictor-corrector method.

Of course the speed of the simulation is of little consequence if its results are not accurate. In fact, the accuracy of the simulation is of vital importance in determining the relative performance of each integration algorithm. An integration routine that executes faster with a given integration step size (calculation interval), but is less accurate than another routine, often requires a longer execution time to achieve results of the same accuracy. To increase the accuracy of such a routine a smaller integration step size must be used. Thus a larger number of integration steps will be executed during the simulation, which in turn leads to an increased execution time. The Euler integration method is an example of an algorithm that executes very fast

for a given integration step size, but in general is not very accurate. Because of its error characteristics this algorithm results in very slow simulation times to obtain reasonable accuracy and is rarely used to simulate most continuous systems.

Figure 3.9 shows the effective number of derivative function calls versus the maximum local error for the ten integration methods that were applied to the Spring Dashpot Benchmark. As would be expected for each integration algorithm, when the amount of error in the simulation is allowed to increase, the number of derivative function calls required decreases. If the derivative function evaluations are the dominant portion of the simulation, then the effective number of derivative function calls will be roughly proportional to the execution time for the simulation. Thus the graph can be interpreted to show, that as the solution accuracy decreases, the execution time for each method also decreases.

A comparison of the different integration algorithms in Figure 3.9 shows that the fourth-order algorithms tend to execute faster for this application than the second-order ones. The faster execution times associated with the second-order algorithms for a given integration step size are more than offset by the amount of error that is present in the results. The routine with the fastest execution time is the fourth-order two processor parallel predictor-corrector algorithm followed closely by the fourth-order parallel two-step block predictor-corrector algorithm. The fourth-order serial predictor-corrector algorithm and the fourth-order Runge-Kutta algorithms are next. When the maximum error is less than 10^{-3} the fifth and six fastest algorithms are the second-order four processor parallel predictor-corrector algorithm and the second-order two processor

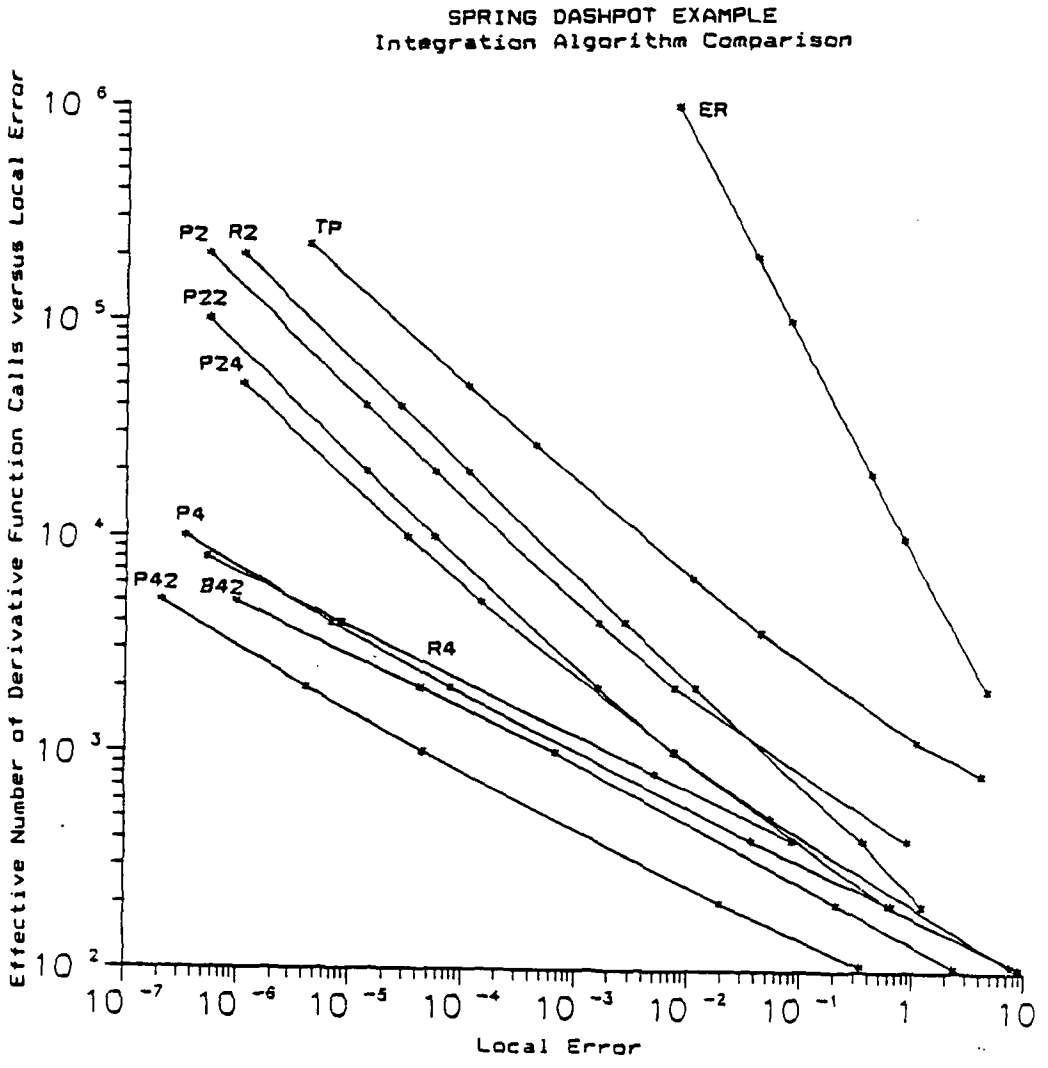


Figure 3.9 Effective Number of Derivative Calls versus Local Error

parallel predictor-corrector algorithm, respectively. For errors greater than 10^{-3}, the two algorithms execute at about the same speed. Thus even the speedup gained through the use of four processors does not always make up for the inaccuracy of the algorithm. The four slowest integration algorithms for this example are all low-order serial algorithms. The seventh fastest algorithm is the second-order predictor-corrector, followed closely by the second-order Runge-Kutta algorithm. The variable-step trapezoidal algorithm is next, with the Euler algorithm being by far the slowest algorithm.

The Spring Dashpot System is modeled using only two rather simple differential equations. Therefore, the derivative function calls may not represent the dominant portion of the simulation. It is probably more accurate to estimate the relative execution time by the equivalent number of floating point operations that are performed during the simulation. This was done for each of the ten integration methods, with each type of floating point operation being given the same weight. Figure 3.10 shows a graph of the effective number of floating point operations versus the error for each of the ten integration algorithms in the Spring Dashpot System. A primary difference between this graph and that of Figure 3.9 is that the serial fourth-order Runge-Kutta algorithm is now the second fastest algorithm, replacing the parallel fourth-order two-step block predictor-corrector algorithm. With this particular system, implementation of a serial integration algorithm is almost as efficient as implementing the fastest parallel algorithm. Another important difference between this graph and that of Figure 3.10 is the four processor second-order parallel predictor-corrector algorithm is actually slower than the same order two processor

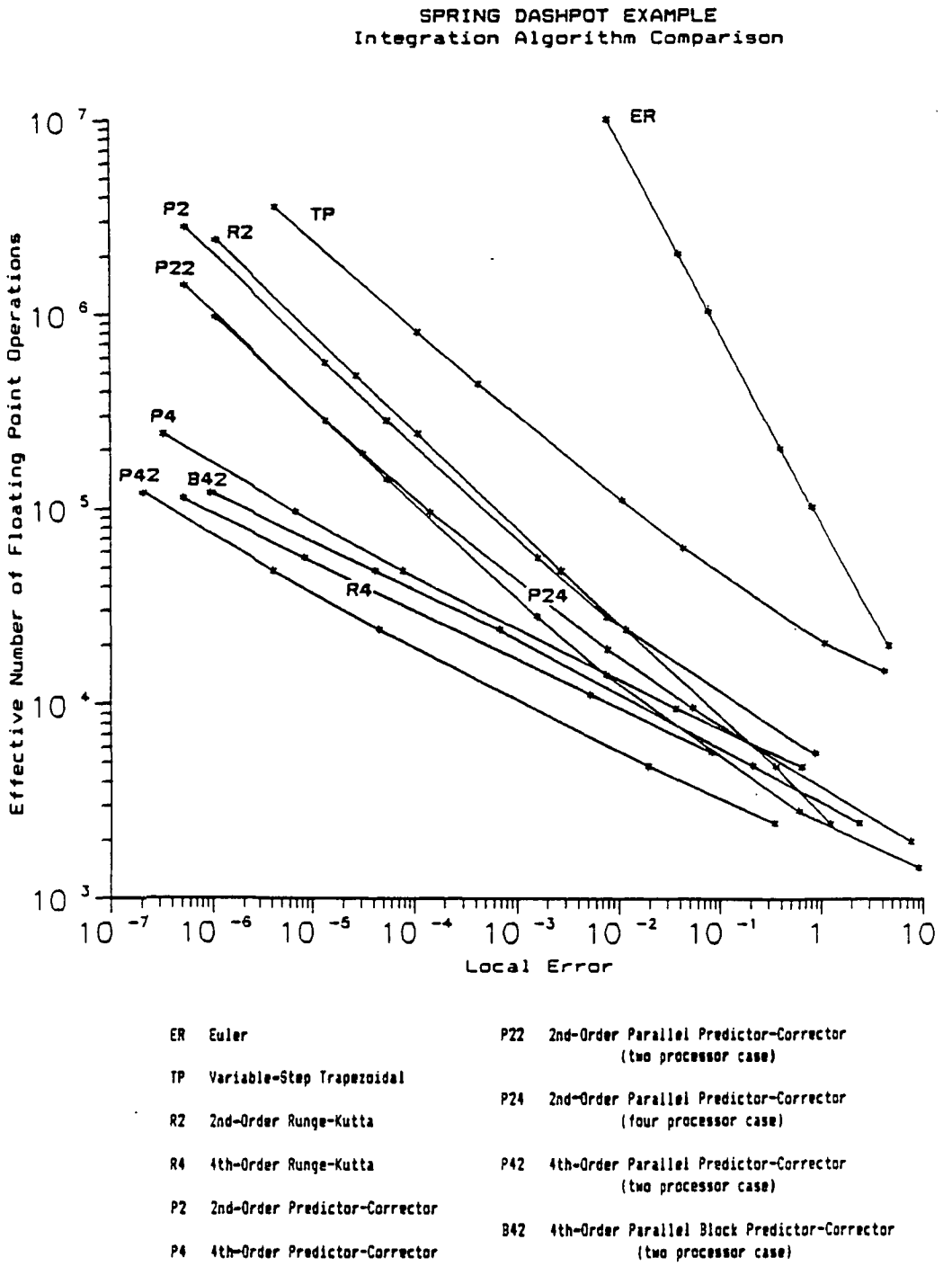


Figure 3.10 Effective Number of Floating Point Operations versus Local Error

algorithm. This is due to the added complexity of the four processor algorithm combined with its slight decrease in accuracy.

Several conclusions can now be made. First, when the amount of error is considered the integration algorithm that provides the most parallelism is not necessarily the fastest algorithm. There appears to be a point with most parallel algorithms where adding more processors to the system causes a decrease in performance due to the increased amount of error. Second, if the relative size and complexity of the system to be simulated is small then a sequential integration algorithm may be more efficient than a parallel algorithm. In a general purpose system, serial algorithms should be made available for use, in addition to the parallel ones. Finally, the optimal integration algorithm is highly dependent on the particular system that is being simulated. No algorithm, serial or parallel, works the best in all situations.

3.3 Combined Approach

The parallel execution of derivative function evaluations and the use of parallel algorithms can be combined through a computer architecture that allows for two or more levels of parallelism. This permits parallelism to be more easily spread out among the processing elements. In such an architecture, the parallel algorithm is executed at the upper level of the hierarchy, and the lower levels are devoted to the parallel execution of derivative function evaluations. Each level of the architecture is linked by an interconnection network that is specifically designed to handle the type and amount of interprocessor communication for that level.

Unfortunately, this combined approach inherits the problems associated with the two individual approaches. For example, there is an allocation problem caused by the parallel execution of the derivative function evaluations, and the parallel integration routine must be chosen carefully or excessive error will result. Still this approach provides much of the criteria necessary for the development of an improved processing environment for continuous simulation.

CHAPTER 4

GENERAL CONFIGURATIONS THAT SUPPORT IMPROVED SIMULATION

4.1 Current Configurations

Today most continuous simulations are performed using computer systems that have very traditional architectures. These systems are usually configured to execute program code sequentially, or in a vector processing mode using one or two processing elements. It is common for such systems to contain a large amount of global memory and only a small amount of on-chip cache memory in which both instructions and data are stored.

Furthermore, most computer systems that are used for continuous simulation have been designed to execute computationally intensive software written in conventional high-level languages. Software written in a continuous simulation language, such as ACSL, is often translated into a general purpose high-level language, such as FORTRAN, before it is compiled into machine code. This means that program code for a continuous simulation is present on the system in three forms, Continuous Simulation Language source code, High-Level Language source code, and the executable object code.

The two step process of translation from continuous simulation languages into other high-level languages before compiling to machine

code is usually a source of inefficiency. This inefficiency directly affects the execution time of the resulting simulation. Overall performance can be improved by eliminating one or both of the translation/compilation steps. There are a number of possible system configurations that allow more efficient execution of continuous simulations. These include configurations that support the direct compilation of continuous simulation languages to object code, direct translation of continuous simulation languages to an intermediate high-level language which is directly executed, and direct execution of the continuous simulation language. Each has its advantages and disadvantages as will be discussed in the following sections.

4.2 Direct Compilation

Direct compilation of the continuous simulation language into object code is one method by which overall performance can be improved. This method requires a separate compiler be written that converts the continuous simulation language directly into the executable object code to be run on the system. The advantage of such an approach is that a specially designed compiler can be optimized to produce much more efficient code than the general purpose compiler. Also, only two forms of code would need to be present on the system at a given time, the continuous simulation source code and the executable object code. The primary disadvantage of this scheme is the loss of flexibility resulting from the absence of a general purpose high-level language. It would now be impossible to extend the capabilities of the continuous simulation

language simply by writing subroutines in a general purpose high-level language. Another disadvantage of this scheme is that creating a compiler is usually a much harder task than creating a translator.

Compiler complexity depends to a large extent on how well the instruction set of the processing elements of a computer system relates to the constructs of the continuous simulation language. If there is a close match, then the creation of a compiler is simplified. If not, then compiler creation is more complex. Current computer systems are often designed around processing elements that have relatively complex instruction sets (CISC), but there is also a trend to build computer systems using reduced instruction set processors (RISC) [37]. Current research has provided evidence that for some applications a computer system built using very efficient reduced instruction set processors can out perform a comparable system that uses complex instruction set processors [38]. This is supported by the fact that often application programs in a complex instruction set system make primary use of only a small percentage of the instructions available. The inclusion of the rarely used instructions in the instruction set of a complex instruction set processor is a source of inefficiency which tends to diminish the overall execution speed of all instructions executed. Compilers for complex instruction set machines are moderately complex whereas compilers for reduced instruction set machines can be very complex.

4.3 Direct Translation

Another implementation that improves efficiency is the creation of a hardware environment that allows the continuous simulation language to be translated directly into a general purpose high-level language that is the base language of the system. This means that the continuous simulation language is first translated into the general purpose high-level language, which is then directly executed by the system. In such a system the hardware would be designed to interpret and execute each instruction of the high-level language. There is be no need for a compiler because the particular high-level language acts as the lowest-level language of the system. A primary advantage of this approach is that by eliminating the compiler, efficiency is improved without the loss of flexibility associated with having a high-level language present. Also, the simulation process can be made more interactive than on conventional systems. This is because on current systems, after a change has been made to a simulation run, the high-level language must be recompiled, and the act of compiling the high-level language to source code is often a very time intensive process. Another advantage to the intermediate execution approach is that only two forms of the continuous simulation will ever need to be present on the system at the same time, the continuous simulation language source code, and the high-level language code.

The primary disadvantage to the direct translation approach is the large amount of hardware required to effectively interpret and execute a general purpose high-level language. Unfortunately, much of this hardware is rarely used by the simulator. This is because a general

purpose high-level language often contains many more instructions and features than are actually needed for continuous system simulation. Even though there has been much worldwide research in developing direct execution processing elements for a number of high-level languages there are only a few that are currently available on the commercial market. Most of these processing elements process various versions of the LISP language [39]. For continuous system simulation there is some question as to whether the LISP language would make an effective intermediate language. This is because LISP is much more suited to perform symbolic operations than numerical computations, and continuous simulation is a very numerically intensive process.

4.4 Direct Execution

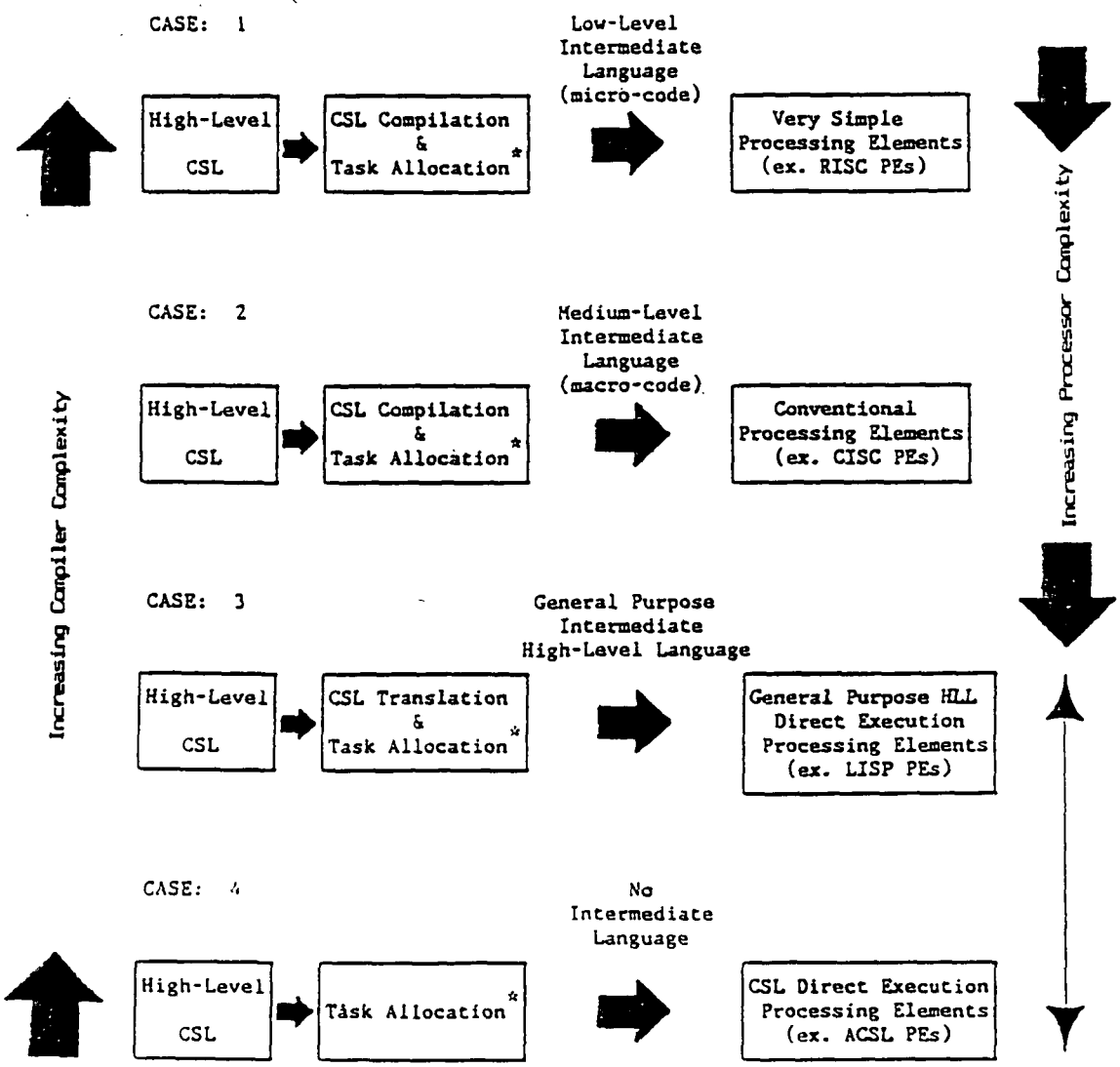
The final implementation to consider is one that improves efficiency by creating a hardware environment that directly executes the continuous simulation language. This means that the continuous simulation language is interpreted and executed directly by the hardware of the system. There are several advantages to this approach. First, execution can be made to be very efficient, since the system architecture is designed to directly match all the constructs of the continuous simulation language. Also, no compiler and no translator is present on the system, so simulation runs can be made very interactive. A further advantage is that continuous simulation code exists in only one form, the continuous simulation source language.

A primary disadvantage to this approach is that there are currently few, if any, processing elements commercially available for the direct execution of continuous simulation languages. Before a system can be constructed, the direct execution processor must be designed and implemented. The design of the continuous simulation language direct execution processor will undoubtedly be very complex (but probably not as complex as the design for a general purpose high-level language direct execution processor). Another possible disadvantage to this approach stems from the general lack of flexibility that is present within the system. Not only will it be impossible to make use of an additional high-level language to extend the capabilities of the continuous simulation language, it is very hard to use the system for any other application.

4.5 Parallel Configurations

All three configurations for improved execution can be adapted to incorporate any number of processing elements. In cases where the number of processors is greater than one, the job of partitioning the problem into a number of concurrent tasks and assigning these tasks to the processors is an added concern. Figure 4.1 illustrates how these environments can be adapted to support the parallel execution of high-level continuous simulation languages.

Computer Configurations that Support the Parallel Execution of
High-Level Continuous Simulation Languages (CSLs)



*Task Allocation complexity depends on a number of factors including process granularity and how well endowed the CSL is with parallel constructs.

Figure 4.1 Parallel Configurations for Improved Execution

Case 1 and Case 2 of Figure 4.1 illustrate the direct compilation approach within a parallel environment. In both cases the continuous simulation language is processed by a compiler and a task allocator. In Case 1 the compiler produces code that is very low level, containing only a small number of simple instructions. This code retains few of the constructs contained in the original continuous simulation language. The code must be partitioned and assigned among a large number of finely grained processing elements. These RISC type processing elements are specifically designed to efficiently execute the low-level code. In general, the compiler for this case must be very complex--producing a large amount of object code. The hardware within the processing elements is relatively simple, but the network interconnecting the processors may be very complex. The problem of task allocation is also large, since there is generally such a large number of finely grained tasks to be allocated.

In Case 2 of the figure, the compiler produces a medium-level code to be executed by a system composed of several more conventional type processing elements. This medium-level code is closer in structure to the actual continuous simulation language, somewhat simplifying compiler construction. There is a trade-off between compiler complexity and processing element complexity, with the processing element in this case being more complex than before.

Case 3 of Figure 4.1 represents the case where the simulation is performed on a system made up of processing elements that directly execute one particular general purpose high-level language. The continuous simulation language must be appropriately translated into

this high-level language in such a way that parallel processing can occur. The general translation processes is usually simple because of the similarity of constructs in both languages, but the allocation process is highly dependent on how well the high-level language can be made to utilize parallel processing techniques. The hardware complexity of the individual processing elements is very large compared to the other cases discussed, and there are a few such processing elements commercially available that will function in a parallel environment [39-40].

Case 4 of Figure 4.1 illustrates how continuous simulation can be performed on a system that uses a set of direct execution continuous simulation language processors. Each of these processors must be designed to directly execute the continuous simulation language and to work with other direct execution processors in a parallel environment. In such an environment one continuous simulation processor acts as host and allocator, providing the other processors in the system with subsections of the source code to concurrently execute. Thus task allocation will occur in a way that is transparent to the user, allowing each processing element to execute portions of the problem in a cooperative manner. The hardware complexity of such processing elements is complex, but probably not as complex as that of a general purpose high-level language processing element.

CHAPTER 5

INTELLIGENT PROCESSING ENVIRONMENTS

Intelligent real-time processing environments can now be developed using many of the concepts discussed previously. A parallel direct execution continuous simulation language type configuration provides an excellent base for such an environment. The direct execution environment allows for single storage of simulation source code, transparent task allocation, and interactive simulation capability. Such an environment can be created by adapting the architecture developed in The University of Alabama's OPERA project to continuous simulation.

The University of Alabama's OPERA architecture (Optimally Parallel Environment for Real-Time Applications) is a very flexible parallel computer architecture that can be adjusted to fit many real-time applications. Continuous simulation is one such application. The architecture is message based, using packet switching techniques as the means for interprocessor communication. Each processing element contains its own local memory, and the number of processing elements in a system can be expanded indefinitely. Several groups of processing elements are connected by high-speed interconnection networks to form clusters. The clusters are then connected by an intracluster interconnection network to form the architecture for the system. At least one processing element within each cluster is dedicated to

intracluster communication. In this way, the architecture is built to support two levels of parallelism.

The mix of processing elements within each cluster can be either homogenous or heterogeneous. In a homogenous cluster, all the processing elements, except possibly the I/O processing element, are identical. In a heterogeneous cluster, the processing elements differ in type and possibly in speed of execution. It is also possible for a system to contain different types of clusters dedicated to specific tasks such as system I/O, computation, etc. The optimal choice is application dependent, depending on the algorithm that is being implemented.

The following sections provide some insight into the general hardware structures needed to implement an intelligent processing environment using an OPERA type architecture. In Section 5.1, a parallel direct execution environment designed to facilitate the parallel execution of derivative function evaluations is described. A direct execution environment suited for simulations that utilize a parallel integration algorithm is described in Section 5.2. Section 5.3 discusses combined direct execution environments, where parallelism is allowed to exist both through the parallel integration algorithm and within the derivative evaluations.

5.1 An Environment for Parallel Derivative Evaluations

One of the parallel methods discussed in Chapter 3 allows parallelism to be introduced by partitioning the set of differential

equations into a number of subsets, each of which are then processed concurrently. A direct execution environment designed to implement such a method is shown in Figure 5.1. This figure shows a system with several clusters connected by a central interconnection network. Each cluster is connected to this network through one of its processing elements, designed specifically to perform the I/O for the cluster. There is one cluster that is designated as the Host Cluster and any number of general purpose clusters contained within the system. Each cluster is capable of directly executing each construct of the continuous simulation language. The clusters themselves are made up of an arbitrary number of processing elements; the actual number being dependent on the desired size of the particular implementation. The individual processors within a cluster are connected using a very fast packet switching type network such as a fiber optic star, or a highly parallel crossbar configuration. The network that runs between the clusters is also a fast packet switching type network that has broadcast capability. This allows data generated at one cluster to be simultaneously transmitted to all the clusters in the system.

The Host Cluster is responsible for setting up the preprocessing environment, for allocating tasks to the general purpose clusters, and for executing the serial portions of the simulation. The amount of parallel processing that is performed within the Host Cluster is limited, so it does not need to contain a large number of processing elements. In fact, it may be possible in some cases to adapt a conventional computer system to act as the Host Cluster.

For each system to be simulated, the entire continuous simulation language source code program is stored within the memory of the Host

Direct Execution Environment for the Parallel Execution
of Derivative Function Evaluations

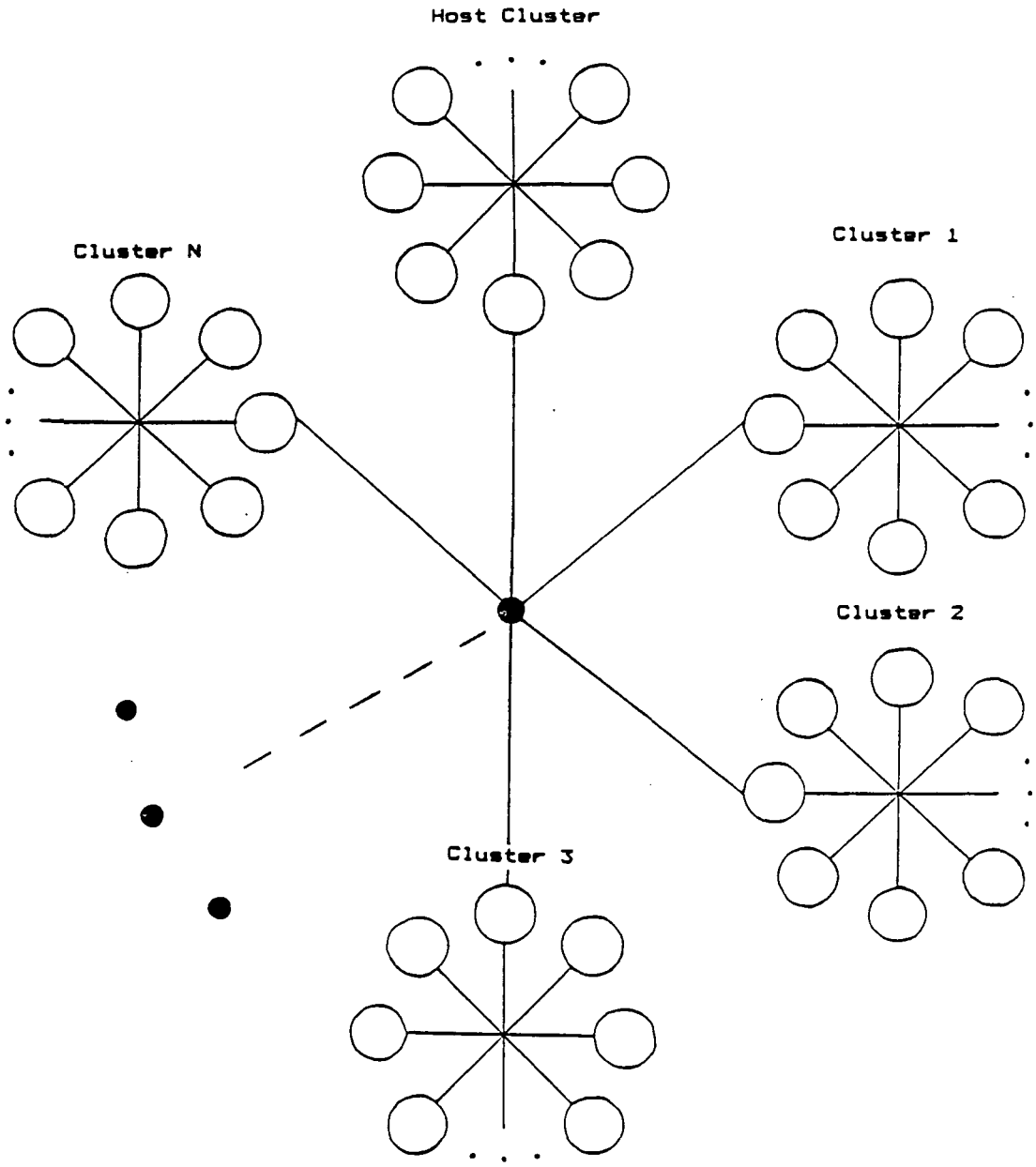


Figure 5.1 Parallel Derivative Function Environment

Cluster. Portions of this code are then allocated by the host to the general purpose clusters of the system for parallel execution. This allocation occurs statically, before run time, in accordance with the integration algorithm and allocation scheme chosen. Allocation always occurs in such a way that each cluster receives at least one state variable (differential equation) to evaluate. In cases where different sections of the derivative evaluations have widely different time constants, a different integration formula can be manually chosen to process these different sections.

The serial portions of the simulation occur at the beginning and end of the simulation run. If the Host Cluster contains several processors, then these sections of the simulation can also be executed in parallel. Since the relative size of these sections is usually small compared to the dynamic portion of the simulation, such parallel execution is rarely warranted.

Before control is transferred to the general purpose clusters, all state variables are assigned their initial values by the Host Cluster. If an integration algorithm is chosen that is not self starting, the Host Cluster is also responsible for calculating a number of integration points before control is transferred to the other clusters for the dynamic portion of the simulation. During the dynamic portion the Host Cluster acts as the I/O point, allowing real-time communication to occur with the outside world.

The general purpose clusters are each assigned a set of differential equations to process in accordance with the integration algorithm chosen. Communication usually occurs between clusters after

each set of derivative function evaluations. Depending on the coupling present between the differential equations and the allocation scheme employed, the number of state variables that must be transferred during this time can be anywhere from zero to the total number of differential equations in the simulation. Thus the speedup for a given system is very application dependent and hard to predict. The number of times data is transferred through the network during each calculation interval depends directly on the number of times derivative function evaluations occur. This in turn is dependent on the particular integration algorithm that is chosen. In tightly coupled systems, integration algorithms that result in fewer function calls may be preferred over others, due to their decreased use of the interconnection network.

Communication between clusters must occur on a regular basis, and each cluster must be synchronized to process the same integration step. This synchronization can be accomplished by a regular polling scheme in which each cluster is given a time slot to broadcast its data to the other clusters. When all the clusters have been polled, concurrent execution can resume within each cluster. As long as the ratio of communication time to execution time remains small, improved execution will result.

The processing of complex differential equations usually results in relatively long execution times. In systems that contain only a few complex equations, an imbalance of computational effort among the processing clusters will result in unacceptably slow simulations. It is for this reason that each cluster has been designed to contain a number of processing elements, thereby providing another level of parallelism.

This level of parallelism can be used to better balance the computational load.

One way to utilize this second level of parallelism is to partition the time consuming built-in functions of the simulation language (such as trigonometric, logarithmic, etc.) among the processing elements of each cluster. These functions are usually the most time consuming portions of the simulation. This partitioning is predetermined in an optimum manner at the time the cluster is constructed, and thus is built into the hardware of the system.

This processing environment has been designed to facilitate the use of conventional sequential integration algorithms during continuous simulation. It is not well suited to perform continuous simulations under the direction of parallel integration algorithms, because the connection network between clusters fails to take advantage of the parallel nature of the algorithms. An alternative environment that can make use of parallel algorithms is discussed in the next section.

5.2 An Environment for Parallel Integration Algorithms

Parallel integration algorithms result in parallelism being applied across the algorithm rather than across the set of differential equations. Direct execution environments can be created specifically to execute simulations governed by parallel integration algorithms. The configurations of such processing environments must reflect the parallel structure associated with the chosen integration algorithm. Such an

approach has the advantage of greatly simplifying the allocation processes, because the integration algorithm itself partitions the problem into groups of concurrent tasks. The amount of coupling present among the differential equations is also not a factor when a parallel integration method is used.

Figure 5.2 shows an OPERA type processing cluster designed specifically to implement the second-order four processor parallel predictor-corrector integration algorithm described by Equation 3.9. Only one cluster is needed to support the simulation process, since parallel integration algorithms require only one level of parallelism. The cluster is made up of one Host/Arbitator type processing element and four general purpose processing elements. Two general purpose processing elements are assigned the job of performing predictor type evaluations, and the other two processing elements are assigned the job of performing corrector type evaluations. Each of the predictor and corrector processing elements receives an identical copy of the derivative function at the beginning of the simulation run. During the simulation, all four processing elements concurrently evaluate the derivative function in the manner described by its unique integration formula. At the end of each integration step data is transferred between the processing elements.

Data is transferred through a number of high-speed data links as shown in Figure 5.2. These data links are arranged in such a way as to reflect the structure of the integration algorithm. Through each data link, a vector of size N is passed between processors; where N again is the number of differential equations (state variables) in the system

Direct Execution Environment for the Processing of the
2nd-Order Parallel Predictor-Corrector Algorithm
(Four Processor Case)

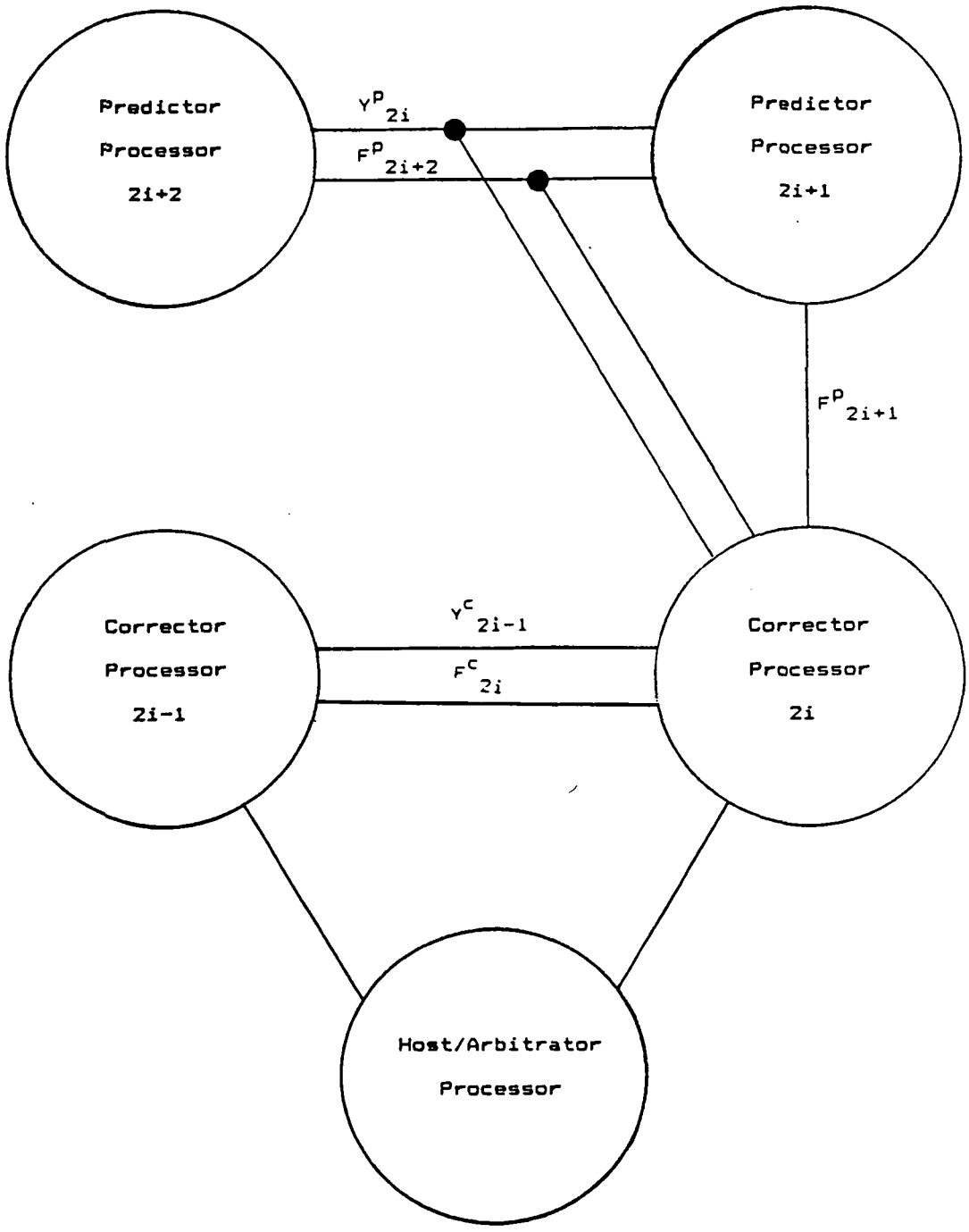


Figure 5.2 Parallel Integration Algorithm Environment

being simulated. If very large simulations are to be performed, then the data links themselves can be designed using a number of parallel paths. This improves communication time by allowing the large vectors to be broken up before being passed through each path in the link.

A close examination of Figure 5.2 reveals some of the details of the interconnection network. Two separate data links interconnect the 2i+2 and the 2i+1 predictor processors with the 2i corrector processor. Through one link the F^{p}_{2i+2} vector is exchanged, and through the other link the Y^{c}_{2i} vector is passed. There is a separate link between the 2i+1 predictor processor and the 2i corrector processor. Through this link the F^{p}_{2i+1} vector is transferred. The two corrector processors are also interconnected with two separate data links. One data link is dedicated to the transfer of the F^{c}_{2i} vector, and the other is dedicated to transferring the Y^{c}_{2i-1} vector. The Host/Arbitrator processing element is also connected to the corrector clusters by two separate data links. These links provide a means for data to be quickly exchanged with the outside world.

For this environment to be effective, each processing element has to be capable of processing, in parallel, data that passes through several data links. Each predictor and corrector processing elements must also contain a fairly large amount of memory, since copies of the derivative function must be present in each of them.

The Host/Arbitrator cluster performs the sequential portions of the simulation and acts as the central I/O point that connects the simulation with real-world systems. In many situations it may be

desirable to utilize a conventional computer system for this function instead of designing a separate processing element. The Host/Arbitrator processing element must also provide the first few integration points to the predictor and corrector processing elements, since the parallel predictor-corrector method is not self starting. It is also responsible for assigning the initial values to the state variables before the simulation begins.

Similar parallel environments can be designed by extending the parallel predictor-corrector algorithm to incorporate a larger number of processing elements. As discussed previously, this extension of parallel processing comes at the price of slightly increased error and decreased stability. The interconnection network required also increases in complexity as the number of processing elements increases. In addition, the number of precalculated data points required at the beginning of the simulation increases. Therefore, careful consideration should be made before the algorithm is extended to form a massively parallel environment.

One problem with creating any environment that is so closely tied to one particular integration algorithm is that no one algorithm has been developed that works well for each possible application. This means environments that are centered around the parallel predictor-corrector method may work well only for a limited number of continuous systems. A possible solution to this problem is to replace the static network shown with a network that can reconfigure itself dynamically, by the use of switching elements to direct data transfer through a number of possible paths. A number of such network are currently being investigated [41-42].

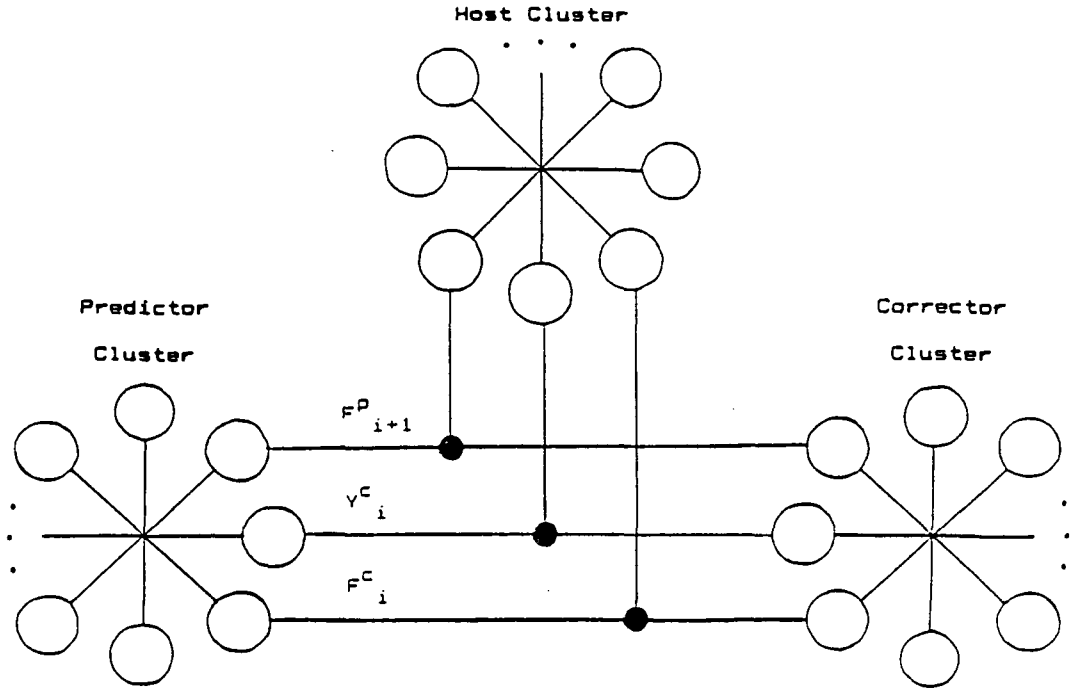
5.3 Environments for Combined Execution

It is also possible with a two-level OPERA type architecture to design a configuration that allows for parallelism to simultaneously exist within the integration algorithm and within the derivative function evaluations. Two examples of direct execution environments that utilize this combined approach will now be discussed.

Figure 5.3a illustrates a direct execution environment that has been specifically designed around the fourth-order two processor parallel predictor-corrector algorithm, that was described by Equation 3.8. The system is made up of three clusters, the Host Cluster, Predictor Cluster, and Corrector Cluster. The clusters are interconnected by three separate bidirectional high-speed data links. Each cluster contains three dedicated I/O processors to separately handle the data that passes through each of these links. The makeup of each of the system clusters is described below.

The Host Cluster is composed of the three special I/O processors and an arbitrary number of general purpose processing elements. Since most of the processing within the Host Cluster is sequential, it may again be possible to utilize a conventional computer system for this task. As in Section 5.1, the Host Cluster is assigned the task of executing the serial sections of continuous simulations that occur at the beginning and end of the simulation run. Furthermore, the Host Cluster must provide the first few integration points to the predictor and corrector clusters, since this parallel predictor-corrector method is not self starting. It is also responsible for assigning the initial

(a) 4th-Order Parallel Predictor-Corrector Combined Environment



(b) 4th-Order Parallel Block Predictor-Corrector Combined Environment

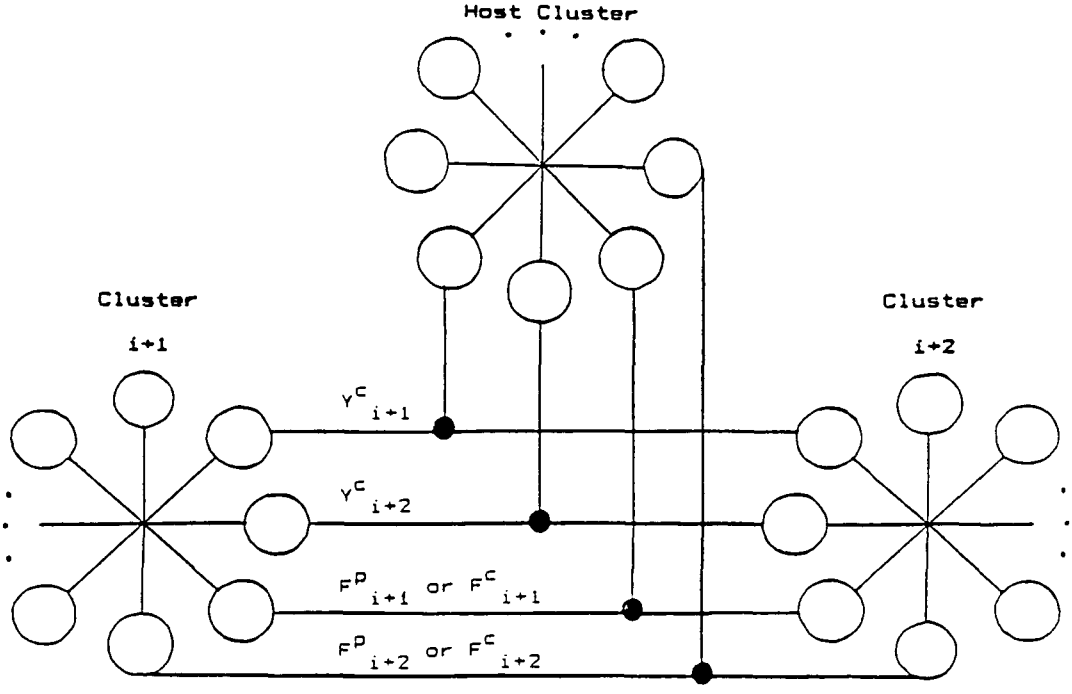


Figure 5.3 Parallel Combined Environments

values to the state variables and allocating the derivative functions among the processing elements of both the predictor and corrector clusters. This allocation problem differs considerably from that described in the first section because the derivative evaluations are spread out among the processing elements within each of the clusters, not among the clusters themselves. Both the predictor and the corrector clusters receive copies of the entire set of derivative evaluations, which are allocated among the processing elements of both clusters in a similar manner. During the dynamic portion of the simulation the Host Cluster acts as the I/O point, allowing the simulation to interact with real-world systems.

The predictor and the corrector portions of the integration algorithm are executed in parallel on two separate clusters. Each cluster executes the appropriate predictor or corrector integration formula and performs one set of derivative evaluations at every step in the integration process. Communication between the clusters occurs at this time, with the state variable vectors being exchanged between the predictor and corrector clusters. As in the previous sections, the size of each of these vectors is equal to the number of differential equations in the system. The predictor cluster passes one vector to the corrector cluster, and the corrector cluster passes two vectors to the predictor cluster. These vectors are passed concurrently through the three high-speed data links that interconnect the clusters. Again, if very large simulations are to be performed, the data links themselves can contain a number of parallel paths.

Both the Predictor Cluster and the Corrector Cluster execute the derivative function evaluations in a parallel manner. This parallel execution is performed either by partitioning the function evaluations in such a way that a certain number of differential equations are processed by each processing element, or by some other method in which the function evaluations are decomposed into several low-level concurrent tasks which are then separately processed by individual processors within the cluster. Regardless of the allocation scheme chosen it should be completely automated, requiring no human interaction, thus being transparent to the user. The type of connection network between the processors within the predictor and the corrector clusters will depend heavily on how the partitioning of the function evaluations is performed. This in turn determines the granularity of processing and tends to suggest an allocation scheme.

The combined processing environment shown in Figure 5.3a is designed specifically to implement a two processor parallel predictor-corrector algorithm. It is possible to expand this processing environment to implement the four, eight, sixteen, or more processor parallel predictor-corrector algorithm, simply by adding an equal number of predictor and corrector clusters to the system. (The integration formula executed by each cluster also has to be altered.) A major drawback to this approach is that a much more complex interconnection network between the clusters is required in order to realize the system. In addition the stability and accuracy of the algorithm has been observed to worsen as the number of processors is increased. This means that the advantages gained by parallelism may be negated by the fact

that a smaller integration step size has to be used to obtain the accuracy desired. Thus a careful evaluation of the feasibility of a proposed multicluster system based around the parallel predictor-corrector integration algorithm should be made before such a system is implemented.

Figure 5.3b shows a direct execution environment that is based around the use of the two-step parallel block predictor-corrector algorithm, described by Equation 3.10. This configuration combines the parallel block algorithm with parallel derivative function evaluations in much the same way as discussed earlier. The major difference in this case is that the system configuration has been changed somewhat to reflect the different integration algorithm used. The system is made up of three clusters, the Host Cluster, i+1 Cluster, and i+2 Cluster. The clusters are interconnected by four separate bidirectional high-speed data links. Each cluster contains four dedicated I/O processors to separately handle the data that passes through each of these links.

The Host Cluster performs the same functions as in the previous case. The i+1 Cluster and the i+2 Cluster combine to separately execute the i+1 and the i+2 time steps that are contained within each block. The clusters concurrently execute one predictor and one corrector formula (each followed by a derivative function evaluation) during each block of execution. Data is passed between clusters twice during each block in the form of state vectors of size N; where N is the number of state variable evaluations (differential equations) in the system. Two vectors are shared between clusters in the middle of the block during

the predictor and corrector evaluations, and four vectors are shared at the end of the block. This is the reason for the four high-speed data links between the clusters. As in the previous case, both clusters execute the derivative function evaluations in parallel, thus fully utilizing the lower level of parallelism.

As with the parallel predictor-corrector algorithm, the parallel block predictor-corrector approach can also be expanded to include more than two clusters. Such expansion tends to increase the amount of parallelism present, but also tends to decrease the stability and accuracy of the simulation. Care must be taken to choose an optimal number of clusters, allowing for the greatest performance in the majority of cases.

5.4 Other Considerations

Regardless of which approach is chosen there are several additional considerations that must be addressed before an intelligent processing environment can be created. One such consideration is the granularity of processing that will be supported. The granularity can be defined as the amount of processing that is performed within each processing node in the system, compared to the amount of data that is transferred between the processing nodes. In a finely grained system, each processing node performs simple operations resulting in a large number of parallel processes being performed. This high degree of parallelism is obtained at the expense of requiring that a large amount of

communication occur between processing elements. In coarsely grained systems, complex processing is performed within each processing node resulting in less parallel processing being performed and less use of the interconnection network. It is important that the relative speeds of the processing elements chosen be compared with the relative speed of the interconnection network, so that an optimal balance between computation and network traffic is achieved.

Another consideration is how the individual processing elements that make up a cluster are designed. It is desirable that every processor efficiently execute its portion of the problem and communicate effectively with other processors in the system. Care should be taken that the processing elements chosen are not only very fast, but also are designed to handle the input and output that is characteristic of their parallel environment. Depending on the granularity of processing and the size of the local memory, it may also be beneficial to utilize such conventional techniques as pipelining and vector processing within the processing elements. The ability to perform multiprocessing within each processing element is also important in situations where the size of the simulation is much larger than can be conveniently handled using the current number of processing elements. In such situations the processing elements will process a number of concurrent tasks sequentially, through task switching, thus allowing very complex simulations to be performed, but at a decreased level of performance.

The interconnection network that connects the processing elements within each cluster, and the network that connects the individual

clusters together must be chosen with care. The type of network chosen needs to be able to handle the communication requirements dictated by the underlining algorithm. Such networks can be either static or dynamic. There is a definite trade-off between having the added control problems and delays associated with dynamic networks and the lack of flexibility associated with static networks. The technology of the network is another consideration. Whether the network will be implemented using standard electrical bussing techniques, or through the use of more advanced technology such as fiber optics [43] must be decided.

Fault tolerance of the environment is also an important consideration. The environment should be able to detect when a faulty condition occurs, isolate the faulty condition, and gracefully recover from the effects of the condition. The proper design of fault tolerant networks is a very broad subject that is currently under much investigation [44-45].

A final consideration is how easily the proposed processing environment can be adapted and expanded to take advantage of improvements in technology. The general environment should be easily expanded by increasing the number of processing elements. In addition, possible changes in technology should have no major effect on the overall structure of the environment.

These are just of few of the considerations that must be investigated before a truly intelligent processing environment can be created. Although the creation of such an environment is very complex, it still appears to be the most practical approach to obtain real-time performance from complex applications [46].

5.5 Summary

The development of an efficient real-time environment for general purpose continuous simulations is a complex process. It requires a thorough understanding of the principles and current problem areas associated with continuous system simulation. It also requires a general knowledge of the type, size, and scope of the simulations that will be processed within that environment. To be efficient the environment must make effective use of the available hardware resources by utilizing configurations that eliminate many or all of the costly translation and compilation steps present in current systems. A direct execution architecture represents one such configuration. In order to guarantee real-time execution speed, the environment must also provide for a near optimal use of parallelism. This can be obtained by exploiting areas of the simulation that lend themselves readily to parallel processing and by implementing parallel algorithms. The parallel processing of derivative function evaluations and the use of parallel integration algorithms are two areas in which parallel processing can be utilized within continuous simulation to provide improved execution. These areas must be mapped to a parallel hardware environment that will fully exploit their structure and parallelism. The direct execution OPERA type environments discussed within this report represent three such environments. Therefore, through the knowledgeable use of hardware resources and parallel processing techniques, an intelligent processing environment can be created that will perform real-time continuous simulations of most real-world problems.

REFERENCES

[1] F. H. Speckhart and W. L. Green, A Guide to Using CSMP, Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1976.

[2] G. Shapiro and M. Rogers, ed., Prospects for Simulation and Simulators of Dynamic Systems, London: Macmillanan Co., Ltd., 1967.

[3] R. E. Stephenson, Computer Simulation for Engineers, New York: Harcourt Brace Jovanovich, Inc., 1971.

[4] R. Bronson, "Computer Simulation: What It Is and How It's Done", BYTE, Vol. 9, No. 3, March 1984, pp. 95-102.

[5] J. J. Lucas and J. V. Wait, "Dare P--A Portable CSSL-Type Simulation Language," Simulation, Vol. 24, No. 1, January 1975, pp. 17-28.

[6] C. A. Guillebeau, "Development of a New Programming Language for Simulation of Dynamic Systems," Ph.D. dissertation, The University of Alabama, College of Engineering, Tuscaloosa, Alabama, 1983.

[7] E. L. Mitchell and J. S. Gauthier, "Advanced Continuous Simulation Language (ACSL)", Simulation, Vol. 26, No. 3, March 1976, pp. 72-78.

[8] E. L. Mitchell and J. S. Gauthier, Advanced Continuous Simulation Language Reference Manual, Concord, Mass.: Mitchell and Gauthier Assoc., 1986.

[9] J. C. Strauss, ed. "The SCi Continuous System Simulation Language (CSSL)", Simulation, Vol. 9, No. 6, December 1967, pp. 281-303.

[10] S. D. Conte, and C. de Boor, Elementary Numerical Analysis: An Algorithmic Approach, New York: McGraw Hill, Inc., 1980.

[11] P. R. Benyon, "A Review of Numerical Methods for Digital Simulation", Simulation, Vol. 11, No. 5, November 1968, pp. 219-238.

[12] J. O. Hamblen, "Parallel Continuous System Simulation Using the Transputer", Simulation, Vol. 49, No. 6, December 1987, pp. 249-253.

[13] G. A. Korn, "Back to Parallel Computation: Proposal for a Completely New On-Line Simulation System Using Standard Minicomputers for Low-Cost Multiprocessing", Simulation, Vol. 19, No. 2, August 1972, pp. 37-45.

[14] J. A. B. Fortes and B. W. Wah, "Systolic Arrays-From Concept to Implementation", Computer, Vol. 20, No. 7, July 1987, pp. 12-17.

[15] H. T. Kung, "Why Systolic Architectures?", Computer, Vol. 15, No. 1, January 1982, pp. 37-46.

[16] M. A. Franklin, "Parallel Solution of Ordinary Differential Equations", IEEE Transaction on Computers, Vol. C-27, No. 5, May 1978, pp. 413-420.

[17] J. B. Sinclair, "Efficient Computation of Optimal Assignments for Distributed Tasks", Journal of Parallel and Distributed Computing, Vol. 4, No. 4, August 1987, pp. 342-362.

[18] C. V. Stewart and C. R. Dyer, "Scheduling Algorithms for PIPE (Pipelined Image-Processing Engine)", Journal of Parallel and Distributed Computing, Vol. 5, No. 2, April 1988, pp. 131-153.

[19] W. Zhao, K. Ramamritham and J. A. Stankovic, "Preemptive Scheduling Under Time and Resource Constraints", IEEE Transactions on Computers, Vol. C-36, No. 8, August 1987, pp. 949-960.

[20] M. Granski, I. Koren, G. M. Silberman, "The Effect of Operation Scheduling on the performance of a Data Flow Computer", IEEE Transactions on Computers, Vol. C-36, No. 9, September 1987, pp. 1019-1029.

[21] C. C. Carroll and K. G. Ananthram, "An Intelligent Allocation Algorithm for Parallel Processing", Bureau of Engineering Research, Report No. 417-17, The University of Alabama, Tuscaloosa, Al, January 1988.

[22] C. C. Carroll, J. N. Youngblood and A. Saha, "Computer Architecture for Efficient Algorithmic Executions in Real-Time Systems: New Technology to Improve Industrial Products", Bureau of Engineering Research, Report No. 409-17, The University of Alabama, University, Alabama, Tuscaloosa, Al, December 1987.

[23] R. B. Asher and J. P. Matuszewski, "Optimal Guidance with Maneuvering Targets", J. Spacecraft, Vol. 11, No. 3, March 1974, pp. 204-206.

[24] L. Stockum and F. C. Weimer, "Optimal and Suboptimal Guidance for a Short Range Homing Missile", IEEE Transactions on Aerospace and Electronic Systems, Vol. AES-12, May 1976, pp. 353-361.

[25] J. N. Youngblood, "Optimal Linear Guidance of Air-to-Air Missiles", Bureau of Engineering Research, Report No. 253-177, The University of Alabama, Tuscaloosa, Al, January 1980.

[26] U. Schendel, Introduction to Numerical Methods for Parallel Computers, Chichester, England: Ellis Horwood Ltd., 1984.

[27] T. R. Martinez and J. J. Vidal, "Adaptive Parallel Logic Networks", Journal of Parallel and Distributed Computing, Vol. 5, No. 1, February 1988, pp. 26-58.

[28] Q. F. Stout, "Supporting Divide-and-Conquer Algorithms for Image Processing", Journal of Parallel and Distributed Computing, Vol. 4, No. 1, February 1987, pp. 95-115.

[29] C. C. Carroll, A. Homaifar and S. Barua, "Efficient Parallel Architecture for Highly Coupled Real-Time Linear System Applications", Bureau of Engineering Research, Report No. 418-17, The University of Alabama, Tuscaloosa, Al, January 1988.

[30] W. L. Miranker and W. Liniger, "Parallel Methods for the Numerical Integration of Ordinary Differential Equations", Math. Comput., Vol.21, pp. 303-320.

[31] L. G. Birta and O. Abou-Rabia, "Parallel Block Predictor-Corrector Methods for ODE's", IEEE Transaction on Computers, Vol. C-36, No. 3, March 1987, pp. 299-311.

[32] P. B. Worland, "Parallel Methods for the Numerical Solution of Ordinary Differential Equations", IEEE Transactions on Computers, Vol. C-25, No. 10, October 1976, pp. 1045-1048.

[33] R. E. Buhrer, H. Brundiers, H. Benz, B. Bron, H. Friess, W. Halg, H. J. Halin, A. Isacson, and M. Tadian, eds., "The ETH-Multiprocessor EMPRESS: A Dynamically Configurable MIMD System", IEEE Transactions on Computers, Vol. C-31, No. 11, November 1982, pp. 1035-1044.

[34] H. J. Halin, R. Buhrer, W. Halg, H. Benz, B. Bron, H. Brundiers, A. Isacson, and M. Tadian, eds., "The ETH Multiprocessor Project: Parallel Simulation of Continuous Systems", Simulation, Vol. 35, No. 4, October 1980, pp. 109-123.

[35] D. I. Rummer, Introduction to Analog Computer Programming, New York: Holt, Rinehart and Winston, Inc., 1969.

[36] J. Walls, M. Greene and W. Teoh, "A Mathematical Model of the Orbital Maneuvering Vehicle", Simulation, Vol. 48, No. 3, March 1987, pp. 98-102.

[37] T. Marshall, "Real-World RISCs", BYTE, Vol. 13, No. 5, May 1988, pp. 263-268.

[38] D. A. Patterson and C. H. Sequin, "RISC 1: A Reduced Instruction Set VLSI Computer", The 8th Annual Symposium on Computer Architecture, May 1981, pp. 443-458.

[39] Y. P. Chiang and M. L. Manwaring, "Direct Execution Lisp and Cell Memory", Technical Report, Department of Electrical and Computer Engineering, Washington State University, Pullman, WA 1987.

[40] A. R. Pleszkun and M. J. Thazhuthaveetil, "The Architecture of Lisp Machines", Computer, Vol. 20, No. 3, March 1987, pp. 35-44.

[41] T. Feng, "A Survey of Interconnection Networks", Computer, Vol. 14, No. 12, December 1981, pp. 12-27.

[42] G. B. Adams III, D. P. Agrawal, and H. J. Seigel, "A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks", Computer, Vol. 20, No. 6, June 1987, pp. 14-27.

[43] L. D. Hutcheson, P. Haugen, A. Husain, "Optical Interconnects Replace Hardwire", IEEE Spectrum, Vol. 24, No. 3, March 1987, pp. 30-41.

[44] D. P. Siewiorek, "Architecture of Fault-Tolerant Computers", Computer, Vol. 17, No. 8, August 1984, pp. 9-18.

[45] O. Serlin, "Fault-Tolerant Systems in Commercial Applications", Computer, Vol. 17, No. 8, August 1984, pp. 19-30.

[46] P. Wiley, "A Parallel Architecture Comes of Age at Last", IEEE Spectrum, Vol. 24, No. 6, June 1987, pp. 46-50.

APPENDIX A

TASK ALLOCATIONS AND PERFORMANCE MEASUREMENTS

```
10 REM                        RANDOM ALLOCATION PROGRAM
20 REM                            (Euler Algorithm)
30 REM                             by B. Earl Wells
40 REM
50 REM                OPTIMAL CONTROL OF GUIDED MISSILE BENCHMARK
60 REM
70 REM             Created February 1988 -- Last Update March 1988
80 REM
90 DIM TIME%(14),PRS%(14),BSTASG%(14),TMS%(14)
100 CLS:PRINT "                                  RANDOM ALLOCATION PROGRAM"
110 PRINT "                                      (Euler Algorithm)"
120 PRINT "                                       by B. Earl Wells":PRINT
130 GOSUB 360:PRINT
140 TMIN%=32000:TMAX%=0
150 ALG%=2
160 ALGPR%=1
170 PRINT
180 INPUT "ENTER NUMBER OF PROCESSORS:  ", NMPRS%
190 INPUT "ENTER NUMBER OF RANDOM ASSIGNMENTS:  ", RNUM%
200 FOR J=1 TO RNUM%
210 FOR I=1 TO NMDE%:PRS%(I)=INT(NMPRS%*RND(1)+1):NEXT I
220 FOR I=1 TO NMPRS%:TMS%(I)=0:NEXT I
230 FOR I=1 TO NMDE%:TMS%(PRS%(I))=2+TMS%(PRS%(I))+TIME%(I):NEXT I
240 TMAX%=0:FOR I=1 TO NMPRS%:IF TMAX%<TMS%(I) THEN TMAX%=TMS%(I)
250 NEXT I
260 IF TMAX%<TMIN% THEN FOR I=1 TO NMDE%:BSTASG%(I)=PRS%(I):NEXT I:
    TMIN%=TMAX%
270 NEXT J
280 PRINT "BEST ALLOCATION FOUND AFTER ";RNUM%;" RANDOM ASSIGNMENTS"
290 FOR I=1 TO 14:PRINT"EQUATION #";I;" ASSIGNED TO PROCESSOR #";
    BSTASG%(I)
300 NEXT I:PRINT"MAX EXECUTION TIME:  ";TMIN%+ALGPR%
310 INPUT " ",X:GOTO 100
320 REM
330 REM
340 REM
350 REM
360  PRINT "                    OPTIMAL CONTROL OF GUIDED ";
     "MISSILE BENCHMARK"
370 NMDE%=14: REM NUMBER OF DIFFERENTIAL EQUATION (PROCESSES)
380  REM EXECUTION TIME OF EACH DE
```

```
390 TIME%(1)=3
400 TIME%(2)=3
410 TIME%(3)=5
420 TIME%(4)=5
430 TIME%(5)=4
440 TIME%(6)=6
450 TIME%(7)=6
460 TIME%(8)=7
470 TIME%(9)=8
480 TIME%(10)=7
490 TIME%(11)=5
500 TIME%(12)=5
510 TIME%(13)=7
520 TIME%(14)=7
530 RETURN
```

Optimal Control of Guided Missile Example
Set of Differential Equations

(EQ 1) $\dot{P1} = -G(P4)(P4)$ [3]

(EQ 2) $\dot{P2} = P1 - G(P4)(P7)$ [3]

(EQ 3) $\dot{P3} = P2 + A(P4) - G(P4)(P9)$ [5]

(EQ 4) $\dot{P4} = P3 + B(P4) - G(P4)(P10)$ [5]

(EQ 5) $\dot{P5} = 2(P2) - G(P7)(P7)$ [4]

(EQ 6) $\dot{P6} = P3 + P5 + A(P7) - G(P7)(P9)$ [6]

(EQ 7) $\dot{P7} = P6 + B(P7) + P4 - G(P7)(P10)$ [6]

(EQ 8) $\dot{P8} = 2(P6) + 2(A)(P9) - G(P9)(P9)$ [7]

(EQ 9) $\dot{P9} = P8 + B(P9) + P7 + A(P10) - G(P9)(P10)$ [8]

(EQ 10) $\dot{P10} = 2(P9) + 2(B)(P10) - G(P10)(P10)$ [7]

(EQ 11) $\dot{P11} = -P2(QT) - G(P4)(P14)$ [5]

(EQ 12) $\dot{P12} = P11-P5(QT)-G(P7)(P14)$ [5]

(EQ 13) $\dot{P13} = P12 + A(P14) - P6(QT) - G(P9)(P14)$ [7]

(EQ 14) $\dot{P14} = P13 + B(P14) - P7(QT) - G(P10)(P14)$ [7]

Allocations for Optimal Control of Guided Missile Example
(based on zero communication delay)

2 Processor Assignment:

Processor 1	Processor 2
EQ 2	EQ 1
EQ 5	EQ 3
EQ 6	EQ 4
EQ 7	EQ 10
EQ 8	EQ 11
EQ 9	EQ 13
EQ 12	EQ 14

Relative Execution Time: 54

3 Processor Assignment:

Processor 1	Processor 2	Processor 3
EQ 1	EQ 9	EQ 2
EQ 3	EQ 12	EQ 4
EQ 6	EQ 13	EQ 5
EQ 10	EQ 14	EQ 7
EQ 11		EQ 8

Relative Execution Time: 37

4 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4
EQ 1	EQ 2	EQ 8	EQ 6
EQ 3	EQ 4	EQ 9	EQ 13
EQ 5	EQ 7	EQ 12	EQ 14
EQ 10	EQ 11		

Relative Execution Time: 28

5 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 2	EQ 4	EQ 10	EQ 3	EQ 1
EQ 7	EQ 5	EQ 13	EQ 6	EQ 9
EQ 14	EQ 8		EQ 12	EQ 11

Relative Execution Time: 23

6 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 2	EQ 7	EQ 5	EQ 8	EQ 6
EQ 3	EQ 14	EQ 13	EQ 9	EQ 10
EQ 12				

Processor 6
EQ 1
EQ 4
EQ 11

Relative Execution Time: 20

7 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 2	EQ 5	EQ 4	EQ 8	EQ 3
EQ 14	EQ 9	EQ 10	EQ 12	EQ 11

Processor 6	Processor 7
EQ 6	EQ 1
EQ 7	EQ 13

Relative Execution Time: 17

8 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 5	EQ 4	EQ 2	EQ 9	EQ 1
EQ 7		EQ 14		EQ 6

Processor 6	Processor 7	Processor 8
EQ 3	EQ 11	EQ 10
EQ 8	EQ 13	EQ 12

Relative Execution Time: 17

9 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 11	EQ 4	EQ 10	EQ 6	EQ 1
	EQ 5		EQ 12	EQ 8

Processor 6	Processor 7	Processor 8	Processor 9
EQ 13	EQ 2	EQ 14	EQ 3
	EQ 9		EQ 7

Relative Execution Time: 16

10 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 4	EQ 1	EQ 8	EQ 6	EQ 14
EQ 12	EQ 11			

Processor 6	Processor 7	Processor 8	Processor 9	Processor 10
EQ 3	EQ 9	EQ 5	EQ 13	EQ 2
		EQ 7		EQ 10

Relative Execution Time: 15

11 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 6	EQ 14	EQ 7	EQ 11	EQ 3
				EQ 5

Processor 6	Processor 7	Processor 8	Processor 9	Processor 10
EQ 4	EQ 1	EQ 9	EQ 10	EQ 8
	EQ 13			EQ 2

Processor 11
EQ 12

Relative Execution Time: 15

12 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 10	EQ 11	EQ 6	EQ 9	EQ 2 EQ 3
Processor 6	**Processor 7**	**Processor 8**	**Processor 9**	**Processor 10**
EQ 5	EQ 13	EQ 1 EQ 7	EQ 14	EQ 8

Processor 11	Processor 12
EQ 12	EQ 4

Relative Execution Time: 14

13 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 1 EQ 2	EQ 3	EQ 4	EQ 5	EQ 6
Processor 6	**Processor 7**	**Processor 8**	**Processor 9**	**Processor 10**
EQ 7	EQ 8	EQ 9	EQ 10	EQ 11

Processor 11	Processor 12	Processor 13
EQ 12	EQ 13	EQ 14

Relative Execution Time: 11

14 Processor Assignment:

Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
EQ 1	EQ 2	EQ 3	EQ 4	EQ 5
Processor 6	**Processor 7**	**Processor 8**	**Processor 9**	**Processor 10**
EQ 6	EQ 7	EQ 8	EQ 9	EQ 10

Processor 11	Processor 12	Processor 13	Processor 14
EQ 11	EQ 12	EQ 13	EQ 14

Relative Execution Time: 11

Table A Performance Measurements cd=0.0

Optimal Control of Guided Missile Benchmark Euler Method of Integration Performance Measurements (Communication Delay = 0.0)					
p	Tp	Cp	Sp	Ep	REp
1	107	107	1.00	1.00	1.00
2	54	108	1.98	0.99	1.96
3	37	111	2.89	0.96	2.79
4	28	112	3.82	0.96	3.65
5	23	115	4.65	0.93	4.33
6	20	120	5.35	0.89	4.77
7	17	119	6.29	0.90	5.66
8	17	136	6.29	0.79	4.95
9	16	144	6.69	0.74	4.97
10	15	150	7.13	0.71	5.09
11	15	165	7.13	0.65	4.63
12	14	168	7.64	0.64	4.87
13	11	143	9.73	0.75	7.28
14	11	154	9.73	0.69	6.76

Table B Performance Measurements cd=0.5

Optimal Control of Guided Missile Benchmark Euler Method of Integration Performance Measurements (Communication Delay = 0.5)					
p	Tp	Cp	Sp	Ep	REp
1	107	107	1.00	1.00	1.00
2	60	119	1.80	0.90	1.62
3	43	129	2.49	0.83	2.06
4	34	136	3.15	0.79	2.48
5	30	150	3.57	0.71	2.54
6	27	159	4.04	0.67	2.72
7	24	168	4.46	0.64	2.84
8	24	192	4.46	0.56	2.48
9	23	207	4.65	0.52	2.40
10	22	220	4.86	0.49	2.37
11	22	237	4.98	0.45	2.25
12	21	252	5.10	0.42	2.16
13	18	234	5.94	0.46	2.72
14	18	252	5.94	0.42	2.52

Table C Performance Measurements cd=1.0

Optimal Control of Guided Missile Benchmark Euler Method of Integration Performance Measurements (Communication Delay = 1.0)					
p	Tp	Cp	Sp	Ep	REp
1	107	107	1.00	1.00	1.00
2	65	130	1.65	0.82	1.35
3	49	147	2.18	0.73	1.59
4	40	160	2.68	0.67	1.79
5	37	185	2.89	0.58	1.67
6	33	198	3.24	0.54	1.75
7	31	217	3.45	0.49	1.70
8	31	248	3.45	0.43	1.49
9	30	270	3.57	0.40	1.41
10	29	290	3.69	0.37	1.36
11	28	308	3.82	0.35	1.33
12	28	336	3.82	0.32	1.22
13	25	325	4.28	0.33	1.41
14	25	350	4.28	0.31	1.31

Table D Performance Measurements cd=3.0

Optimal Control of Guided Missile Benchmark Euler Method of Integration Performance Measurements (Communication Delay = 3.0)					
p	Tp	Cp	Sp	Ep	REp
1	107	107	1.00	1.00	1.00
2	87	174	1.23	0.61	0.76
3	73	219	1.47	0.49	0.72
4	64	256	1.67	0.42	0.70
5	65	325	1.65	0.33	0.54
6	59	354	1.81	0.30	0.55
7	59	413	1.81	0.26	0.47
8	59	472	1.81	0.23	0.41
9	58	522	1.84	0.20	0.38
10	57	570	1.88	0.19	0.35
11	54	594	1.98	0.18	0.36
12	56	672	1.91	0.16	0.30
13	53	689	2.02	0.16	0.31
14	53	742	2.02	0.14	0.29

APPENDIX B

INTEGRATION PROGRAMS

```
/*************************************************************/
/*        ACSL Type Simulation Using the Simple Euler        */
/*                    Method of Integration                  */
/*                                                           */
/*                     by B. Earl Wells                      */
/*                   University of Alabama                   */
/*                                                           */
/*               Last Update:  March 1988                    */
/*************************************************************/


#include <stdio.h>
#include <math.h>
/* Maximum Size allowed for Internal Arrays      */
/* (represents the maximum number of state       */
/*  variables possible in application plus one). */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int t_flg,s_var,der_add,der_mul;
double max_err;

main()
{
     char ch[30];

     int i,cint_lmt,cint_num,int_val();
     double num_calls,time_var;

     double atof();
     double k1[SZ],k2[SZ],k3[SZ],k4[SZ];
     double f[SZ],y[SZ],t,h,cint,lst;

     void initial(),derivative(),dynamic(),output();
     void header(),termt(),st_var(),rel_time(),err();

     printf("\n\n\n");
     printf("            ACSL Type Continuous Simulation using the\n");
     printf("               Simple Euler Method of Integration\n\n");
     printf("                        by B. Earl Wells\n\n");
```

```
header();
lst=0;
time_var=0;
num_calls=0;
max_err=0;
printf("Enter iteration interval:  ");
gets(ch);
h=atof(ch);

printf("Enter communication interval:  ");
gets(ch);
cint=atof(ch);

printf("Enter the number of communication intervals between");
printf(" outputs: ");
cint_lmt=atoi(gets(ch));
cint_num=cint_lmt;

/* Process the INITIAL Section */
initial();

/* Assign Initial Conditions to State Variable(s) */
int_val(y);  /* process special initial value section */


printf("Override the initial conditions of the state ");
printf("variables (Y or N)?");
gets(ch);

if (ch[0]=='y'||ch[0]=='Y') {
     /* override initial conditions specified in initial value */
     /* section and store these results into an array.        */
     for (i=1;i<=s_var;++i) {
          printf("Enter Initial value for y[%d]:  ",i);
          scanf("%s",ch);
          y[i]=atof(ch);
     }
}
printf("\nBeginning Run\n");

t=0;

/* Process the DYNAMIC and DERIVATIVE Sections */
for( ; ; ) {

     num_calls+=1;   /* Increment Derivative counter */
     derivative(t,y,f);
     time_var+=der_add*ADD + der_mul*MUL;

     if (t>=lst-h/2) {
          dynamic(t,y);
          if(t_flg) break;
```

```
                if ((cint_num+=1)>cint_lmt) {
                    cint_num=1;
                    output(t,y);

                }
                lst=lst+cint;
            }
            if (t_flg) break;

            for(i=1;i<=s_var;++i) {
                y[i]=y[i]+h*f[i];
                time_var+=ADD+MUL;
            }
            t=t+h;
            time_var+=ADD;

        }

        output(t,y);

        printf("Run Terminated\n\n");

        printf("Number of calls to the DERIVATIVE Section:  % 1.0f\n",
            num_calls);
        printf("Relative Execution Time:  % 1.0f\n",time_var);
        if (max_err>0)
            printf("Maximum local Error is :  %e\n",max_err);


}

/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
    if (num<0)
        num=-num;

    return(num);
}

/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
    t_flg=flg;
}
```

```
/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}

/* This routine stores the number of addition and  */
/* subtraction operations that are executed every */
/* time the DERIVATIVE section is called into the */
/* global variable "der_add".                     */
void num_add(t_nm)
int t_nm;
{
     der_add=t_nm;
}

/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

```
/*********************************************************************/
/* ACSL Type Simulation Using the Serial 2nd Order Adams-Moulton     */
/*        Predictor-Corrector Method of Integration.                 */
/*                                                                   */
/*                      by B. Earl Wells                             */
/*                     University of Alabama                         */
/*                                                                   */
/*                  Last Update: March  1988                         */
/*********************************************************************/


#include <stdio.h>

/* Maximum Size allowed for Internal Arrays */
/* (represents the maximum number of state  */
/*  variables in application plus one.)     */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int s_var,t_flg,der_add,der_mul;
double max_err;

main()
{
     char ch[30];
     double num_calls,time_var;
     int i,p,cint_lmt,cint_num;
     double atof();
     double yj[SZ],k1[SZ],k2[SZ];
     double f[SZ],y[SZ],fn[4][SZ],t,h,tj,cint,lst;
     void initial(),int_val(),derivative(),dynamic(),output();
     void header(),termt(),st_var(),rel_time(),err();

     printf("\n\n\n");
     printf("      ACSL Type Continuous Simulation using the");
     printf(" serial form\n");
     printf("      of the Second Order Adams-Moulton");
     printf(" Predictor-Corrector\n");
     printf("                      Method of Integration\n\n");
     printf("                         by B. Earl Wells\n\n");

     header();

     tj=0;
     lst=0;
     time_var=0;
     der_add=0;
     der_mul=0;
     num_calls=0;
     max_err=0;
```

```
printf("Enter iteration interval:  ");
h=atof(gets(ch));

printf("Enter communication interval:  ");
cint=atof(gets(ch));

printf("Enter the number of communication intervals between");
printf(" outputs: ");
cint_lmt=atoi(gets(ch));
cint_num=cint_lmt;

/* Process the INITIAL Section */
initial();

/* Assign Initial Conditions to State Variable(s) */
int_val(y);            /* process special initial value  */
for (i=1;i<=s_var;++i) /* and load results into an array */
     yj[i]=y[i];


printf("Override the initial conditions of the state ");
printf("variables (Y or N)?");
gets(ch);

if (ch[0]=='y'||ch[0]=='Y') {
     /* override initial conditions specified in initial value */
     /* section and store these results into an array.        */
     for (i=1;i<=s_var;++i) {
          printf("Enter Initial value for y[%d]:  ";i);
          gets(ch);
          yj[i]=atof(ch);
     }
}
printf("\nBeginning Run\n");


/* Process the DYNAMIC and DERIVATIVE Sections */

/* Find initial four values using the Runge-Kutta Method */
for(p=1;p>=0;--p) {
     t=tj;
     for(i=1;i<=s_var;++i)
          y[i]=yj[i];
     t_flg=0;               /* Clear Termination Flag */
     num_calls+=1;   /* Increment Derivative counter */
     time_var+=der_add*ADD+der_mul*MUL;
     derivative(t,y,f);

     /* Process DYNAMIC Section every Communication Interval */
     if (t>=lst-h/2) {
```

```
        dynamic(t,y);
        if(t_flg) break; /* exit if Termination Flag is set */

        /* Produce Output every "cint_lmt" number of */
        /* of Communication Intervals                */
        if ((cint_num+=1)>cint_lmt) {
             cint_num=1;
             output(t,y);

        }
        lst=lst+cint;
   }
   if (t_flg) break;    /* Exit DERIVATIVE/DYNAMIC loop */
                        /* if Termination Flag is set   */

   for (i=1;i<=s_var;++i)
       fn[p][i]=f[i];

   if (p==0) break;

   /* Process First Order Runge-Kutta Equations */
   for(i=1;i<=s_var;++i) {
        k1[i]=f[i]*h;
        y[i]=k1[i]/2+yj[i];
        time_var+=2*MUL+ADD;
   }
   t=t+tj+h/2;
   time_var+=2*ADD+MUL;

   num_calls+=1;   /* Increment Derivative counter */
   time_var+=der_add*ADD+der_mul*MUL;
   derivative(t,y,f);

   /* Process Second Order Runge-Kutta Equations */
   for(i=1;i<=s_var;++i) {
        k2[i]=f[i]*h;
        yj[i]=k2[i]+yj[i];
        time_var+=ADD+MUL;
   }

   t=tj+h;
   time_var+=ADD;

   tj=t;

}
```

```
for ( ; ; ) {
     t=t+h;
     /* Predictor Equations*/
     for (i=1;i<=s_var;++i) {
       y[i]=yj[i]+h/2*(3*fn[0][i]-fn[1][i]);
       time_var+=2*ADD+3*MUL;
     }

     t_flg=0;
     num_calls+=1;   /* Increment Derivative counter */
     time_var+=der_add*ADD+der_mul*MUL;
     derivative(t,y,f);

     /* Corrector Equations */
     for (i=1;i<=s_var;++i) {
       y[i]=yj[i]+h/2*(f[i]+fn[0][i]);
       time_var+=2*ADD+2*MUL;
     }

     num_calls+=1;   /* Increment Derivative counter */
     time_var+=der_add*ADD+der_mul*MUL;
     derivative(t,y,f);

     /* Process DYNAMIC Section every Communication Interval */
     if (t>=lst-h/2) {

          dynamic(t,y);
          if(t_flg) break;/* exit if Termination Flag is set */

          /* Produce Output every "cint_lmt" number of */
          /* of Communication Intervals                */
          if ((cint_num+=1)>cint_lmt) {
               cint_num=1;
               output(t,y);

          }
          lst=lst+cint;
     }

    if(t_flg) break;  /* exit if Termination Flag is set */

     for (i=1;i<=s_var;++i) {
          fn[1][i]=fn[0][i];
          fn[0][i]=f[i];
          yj[i]=y[i];
     }
}


output(t,y);
printf("Run Terminated\n\n");
```

```
     printf("Number of calls to the DERIVATIVE Section:  % 1.0f\n",
          num_calls);
     printf("Relative Execution Time:  % 1.0f\n",time_var);
     if (max_err>0)
          printf("Maximum local Error is :  %e\n",max_err);


}

/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
     if (num<0)
          num=-num;

     return(num);
}

/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
     t_flg=flg;
}

/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}
```

```
/* This routine stores the number of addition and  */
/* subtraction operations that are executed every */
/* time the DERIVATIVE section is called into the */
/* global variable "der_add".                     */
void num_add(t_nm)
int t_nm;
{
     der_add=t_nm;
}

/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

```
/*****************************************************************/
/* ACSL Type Simulation Using the Serial 4th Order Adams-Moulton */
/*           Predictor-Corrector Method of Integration           */
/*                                                               */
/*                       by B. Earl Wells                        */
/*                     University of Alabama                     */
/*                                                               */
/*                  Last Update: March 1988                      */
/*****************************************************************/


#include <stdio.h>

/* Maximum Size allowed for Internal Arrays */
/* (represents the maximum number of state  */
/*  variables in application plus one.)     */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int s_var,t_flg,der_add,der_mul;
double max_err;

main()
{
     char ch[30];
     double num_calls,time_var;
     int i,p,cint_lmt,cint_num;
     double atof();
     double yj[SZ],k1[SZ],k2[SZ];
     double f[SZ],y[SZ],fn[4][SZ],t,h,tj,cint,lst;
     void initial(),int_val(),derivative(),dynamic(),output();
     void header(),termt(),st_var(),rel_time(),err();

     printf("\n\n\n");
     printf("      ACSL Type Continuous Simulation using the");
     printf(" serial form\n");
     printf("      of the Fourth Order Adams-Moulton");
     printf(" Predictor-Corrector\n");
     printf("                      Method of Integration\n\n");
     printf("                         by B. Earl Wells\n\n");

     header();

     tj=0;
     lst=0;
     time_var=0;
     der_add=0;
     der_mul=0;
     num_calls=0;
     max_err=0;
```

```
printf("Enter iteration interval:  ");
h=atof(gets(ch));

printf("Enter communication interval:  ");
cint=atof(gets(ch));

printf("Enter the number of communication intervals between");
printf(" outputs: ");
cint_lmt=atoi(gets(ch));
cint_num=cint_lmt;

/* Process the INITIAL Section */
initial();

/* Assign Initial Conditions to State Variable(s) */
int_val(y);            /* process special initial value  */
for (i=1;i<=s_var;++i) /* and load results into an array */
     yj[i]=y[i];


printf("Override the initial conditions of the state ");
printf("variables (Y or N)?");
gets(ch);

if (ch[0]=='y'||ch[0]=='Y') {
   /* override initial conditions specified in initial value */
   /* section and store these results into an array.        */
     for (i=1;i<=s_var;++i) {
          printf("Enter Initial value for y[%d]:  ",i);
          gets(ch);
          yj[i]=atof(ch);
     }
}
printf("\nBeginning Run\n");


/* Process the DYNAMIC and DERIVATIVE Sections */

/* Find initial four values using the Runge-Kutta Method */
for(p=3;p>=0;--p) {
     t=tj;
     for(i=1;i<=s_var;++i)
          y[i]=yj[i];
     t_flg=0;               /* Clear Termination Flag */
     num_calls+=1;   /* Increment Derivative counter */
     time_var+=der_add*ADD+der_mul*MUL;
     derivative(t,y,f);

     /* Process DYNAMIC Section every Communication Interval */
     if (t>=lst-h/2) {
```

```
        dynamic(t,y);
        if(t_flg) break; /* exit if Termination Flag is set */

        /* Produce Output every "cint_lmt" number of */
        /* of Communication Intervals                */
        if ((cint_num+=1)>cint_lmt) {
             cint_num=1;
             output(t,y);

        }
        lst=lst+cint;
    }
    if (t_flg) break;    /* Exit DERIVATIVE/DYNAMIC loop */
                         /* if Termination Flag is set   */

    for (i=1;i<=s_var;++i)
        fn[p][i]=f[i];

    if (p==0) break;

    /* Process First Order Runge-Kutta Equations */
    for(i=1;i<=s_var;++i) {
         k1[i]=f[i]*h;
         y[i]=k1[i]/2+yj[i];
         time_var+=2*MUL+ADD;
    }
    t=t+tj+h/2;
    time_var+=2*ADD+MUL;

    num_calls+=1;    /* Increment Derivative counter */
    time_var+=der_add*ADD+der_mul*MUL;
    derivative(t,y,f);

    /* Process Second Order Runge-Kutta Equations */
    for(i=1;i<=s_var;++i) {
         k2[i]=f[i]*h;
         yj[i]=k2[i]+yj[i];
         time_var+=ADD+MUL;
    }

    t=tj+h;
    time_var+=ADD;

    tj=t;


}
```

```
for ( ; ; ) {
     t=t+h;
     /* Predictor Equations */
     for (i=1;i<=s_var;++i) {
       y[i]=yj[i]+h/24*(55*fn[0][i]-59*fn[1][i]+37*fn[2][i]
            -9*fn[3][i]);
       time_var+=4*ADD+6*MUL;
     }

     t_flg=0;
     num_calls+=1;   /* Increment Derivative counter */
     time_var+=der_add*ADD+der_mul*MUL;
     derivative(t,y,f);

     /* Corrector Equations */
     for (i=1;i<=s_var;++i) {
       y[i]=yj[i]+h/24*(9*f[i]+19*fn[0][i]-5*fn[1][i]+fn[2][i]);
       time_var+=4*ADD+5*MUL;
     }

     num_calls+=1;   /* Increment Derivative counter */
     time_var+=der_add*ADD+der_mul*MUL;
     derivative(t,y,f);

     /* Process DYNAMIC Section every Communication Interval */
     if (t>=lst-h/2) {

          dynamic(t,y);
          if(t_flg) break; /* exit if Termination Flag is set */

          /* Produce Output every "cint_lmt" number of */
          /* of Communication Intervals                */
          if ((cint_num+=1)>cint_lmt) {
               cint_num=1;
               output(t,y);

          }
          lst=lst+cint;
     }

    if(t_flg) break;  /* exit if Termination Flag is set */

     for (i=1;i<=s_var;++i) {
          fn[3][i]=fn[2][i];
          fn[2][i]=fn[1][i];
          fn[1][i]=fn[0][i];
          fn[0][i]=f[i];
          yj[i]=y[i];
     }
}
```

```
     output(t,y);
     printf("Run Terminated\n\n");

     printf("Number of calls to the DERIVATIVE Section:  % 1.0f\n",
          num_calls);
     printf("Relative Execution Time:  % 1.0f\n",time_var);
     if (max_err>0)
          printf("Maximum local Error is :  %e\n",max_err);


}

/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
     if (num<0)
          num=-num;

     return(num);
}

/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
     t_flg=flg;
}

/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}
```

```
/* This routine stores the number of addition and  */
/* subtraction operations that are executed every */
/* time the DERIVATIVE section is called into the */
/* global variable "der_add".                     */
void num_add(t_nm)
int t_nm;
{
     der_add=t_nm;
}

/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

```
/*************************************************************/
/*       ACSL Type Simulation Using the Variable Step        */
/*            Trapezoidal Method of Integration              */
/*                                                           */
/*                    by B. Earl Wells                       */
/*                  University of Alabama                    */
/*                                                           */
/*                Last Update: March 11, 1988                */
/*************************************************************/


#include <stdio.h>
#include <math.h>

/* Maximum Size allowed for Internal Arrays      */
/* (represents the maximum number of state       */
/*  variables possible in application plus one). */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int t_flg,s_var,der_add,der_mul;
double max_err;

main()
{
     char ch[30];

     int i,cint_lmt,cint_num,int_val();
     double num_calls,time_var;

     double atof(),abs();
     double hl[SZ],f[SZ],y[SZ],t,h,cint,lst,errr,err_val;

     void initial(),derivative(),dynamic(),output();
     void header(),termt(),st_var(),rel_time(),err();

     printf("\n\n\n");
     printf("     ACSL Type Continuous Simulation using the");
     printf(" Variable Step\n");
     printf("                Trapezoidal Method of Integration\n\n");
     printf("                        by B. Earl Wells\n\n");

     header();

     time_var=0;
     der_add=0;
     der_mul=0;
     num_calls=0;
     max_err=0;
```

```
     printf("Enter iteration interval:  ");
     gets(ch);
     h=atof(ch);

     printf("Enter communication interval:  ");
     gets(ch);
     cint=atof(ch);

     printf("Enter the number of communication intervals between");
     printf(" outputs: ");
     cint_lmt=atoi(gets(ch));

     /* Process the INITIAL Section */
     initial();

     /* Assign Initial Conditions to State Variable(s) */
     int_val(y);  /* process special initial value section */


     printf("Override the initial conditions of the state ");
     printf("variables (Y or N)?");
     gets(ch);

     if (ch[0]=='y'||ch[0]=='Y') {
          /* override initial conditions specified in initial value */
          /* section and store these results into an array.        */
          for (i=1;i<=s_var;++i) {
               printf("Enter Initial value for y[%d]:  ",i);
               scanf("%s",ch);
               y[i]=atof(ch);
          }
     }

     printf("Enter the maximum Estimation Error allowed:  ");
/*   gets(ch);
     err_val=atof(ch);*/
     err_val=1e-8;

     printf("\nBeginning Run\n");

     t=0;

     num_calls+=1;          /* Increment Derivative counter */
     time_var+=der_add*ADD+der_mul*MUL;
     derivative(t,y,f);

     dynamic(t,y);

     cint_num=1;
     lst=cint;
     output(t,y);
```

```
for (i=1;i<=s_var;++i) {
     f[i]=h/2*f[i];
     time_var+=ADD+MUL;
}

/* Process the DYNAMIC and DERIVATIVE Sections */
for( ; ; ) {

     t=t+h;
     time_var+=ADD;
     for (i=1;i<=s_var;++i) {
          hl[i]=f[i]+y[i];
          y[i]=f[i]*2+y[i];
          time_var+=2*ADD+MUL;
     }

     num_calls+=1;          /* Increment Derivative counter */
     time_var+=der_add*ADD+der_mul*MUL;
     derivative(t,y,f);

     if (t>=lst-h/2) {
          dynamic(t,y);
          if(t_flg) break;

          if ((cint_num+=1)>cint_lmt) {
               cint_num=1;
               output(t,y);

          }
          lst=lst+cint;
     }
     if (t_flg) break;

     do  {
          errr=0;

          num_calls+=1;       /* Increment Derivative counter */
          time_var+=der_add*ADD+der_mul*MUL;
          derivative(t,y,f);

          for(i=1;i<=s_var;++i) {
               f[i]=h/2*f[i];
               errr=errr+abs(y[i]-(hl[i]+f[i]));
               y[i]=hl[i]+f[i];
               time_var+=5*ADD+2*MUL;
          }
     } while (errr>err_val);

}
```

```
     output(t,y);
     printf("Run Terminated\n\n");

     printf("Number of calls to the DERIVATIVE Section:  % 1.0f\n",
          num_calls);
     printf("Relative Execution Time:  % 1.0f\n",time_var);
     if (max_err>0)
          printf("Maximum local Error is :  %e\n",max_err);

}

/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
     if (num<0)
          num=-num;

     return(num);
}

/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
     t_flg=flg;
}

/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}
```

```
/* This routine stores the number of addition and  */
/* subtraction operations that are executed every */
/* time the DERIVATIVE section is called into the */
/* global variable "der_add".                     */
void num_add(t_nm)
int t_nm;
{
     der_add=t_nm;
}

/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

```
/*************************************************************/
/*     ACSL Type Simulation Using the Serial 2nd Order       */
/*           Runge-Kutta Method of Integration               */
/*                                                           */
/*                    by B. Earl Wells                       */
/*                  University of Alabama                    */
/*                                                           */
/*              Last Update: March, 1988                     */
/*************************************************************/


#include <stdio.h>

/* Maximum Size allowed for Internal Arrays */
/* (represents the maximum number of state  */
/*  variables in application plus one.)     */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int s_var,t_flg,der_add,der_mul;
double max_err;

main()
{
     double num_calls,time_var;
     char ch[30];
     int i,cint_lmt,cint_num;
     double atof();
     double yj[SZ],k1[SZ],k2[SZ];
     double f[SZ],y[SZ],t,h,tj,tjal,cint,lst;
     void initial(),int_val(),derivative(),dynamic(),output();
     void header(),termt(),st_var(),rel_time(),err();

     printf("\n\n\n");
     printf("      ACSL Type Continuous Simulation using the ");
     printf("Serial form of\n");
     printf("        the Second Order Runge-Kutta Method of ");
     printf("Integration.\n\n");
     printf("                          by B. Earl Wells\n\n");

     header();

     tj=0;
     lst=0;
     der_add=0;
     der_mul=0;
     num_calls=0;
     time_var=0;
     max_err=0;
```

```
printf("Enter iteration interval:  ");
scanf("%s",ch);
h=atof(ch);

printf("Enter communication interval:  ");
scanf("%s",ch);
cint=atof(ch);

printf("Enter the number of communication intervals between");
printf(" outputs: ");
scanf("%d",&cint_lmt);
cint_num=cint_lmt;

/* Process the INITIAL Section */
initial();

/* Assign Initial Conditions to State Variable(s) */
int_val(y);            /* process special initial value  */
for (i=1;i<=s_var;++i) /* and load results into an array */
     yj[i]=y[i];


printf("Override the initial conditions of the state ");
printf("variables (Y or N)?");
scanf("%s",ch);

if (ch[0]=='y'||ch[0]=='Y') {
 /* override initial conditions specified in initial value */
 /* section and store these results into an array.         */
     for (i=1;i<=s_var;++i) {
          printf("Enter Initial value for y[%d]:  ",i);
          scanf("%s",ch);
          yj[i]=atof(ch);
     }
}
printf("\nBeginning Run\n");


/* Process the DYNAMIC and DERIVATIVE Sections */
for( ; ; ) {
     t=tj;
     for(i=1;i<=s_var;++i)
          y[i]=yj[i];
     t_flg=0;               /* Clear Termination Flag */

     num_calls+=1;         /* Increment Derivative counter */
     time_var+=der_add*ADD+der_mul*MUL;
     derivative(t,y,f);

     /* Process DYNAMIC Section every Communication Interval */
     if (t>=lst-h/2) {
```

```
            dynamic(t,y);
            if(t_flg) break; /* exit if Termination Flag is set */

            /* Produce Output every "cint_lmt" number of */
            /* of Communication Intervals                */
            if ((cint_num+=1)>cint_lmt) {
                 cint_num=1;
                 output(t,y);

            }
            lst=lst+cint;
       }
       if (t_flg) break;    /* Exit DERIVATIVE/DYNAMIC loop */
                            /* if Termination Flag is set   */

       /* Process First Order Runge-Kutta Equations */
       for(i=1;i<=s_var;++i) {
            k1[i]=f[i]*h;
            y[i]=k1[i]/2+yj[i];
            time_var+=2*MUL+ADD;
       }
       t=t+tj+h/2;
       time_var+=2*ADD+MUL;

       num_calls+=1;        /* Increment Derivative counter */
       time_var+=der_add*ADD+der_mul*MUL;
       derivative(t,y,f);

       /* Process Second Order Runge-Kutta Equations */
       for(i=1;i<=s_var;++i) {
            k2[i]=f[i]*h;
            yj[i]=k2[i]+yj[i];
            time_var+=ADD+MUL;
       }
       t=tj+h;
       time_var+=ADD;

       tj=t;


  }

  output(t,y);
  printf("Run Terminated\n\n");

  printf("Number of calls to the DERIVATIVE Section:  % 1.0f\n",
       num_calls);
  printf("Relative Execution Time:  % 1.0f\n",time_var);
  if (max_err>0)
       printf("Maximum local Error is :  %e\n",max_err);


}
```

```
/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
     if (num<0)
          num=-num;

     return(num);
}

/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
     t_flg=flg;
}

/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}

/* This routine stores the number of addition and  */
/* subtraction operations that are executed every */
/* time the DERIVATIVE section is called into the */
/* global variable "der_add".                     */
void num_add(t_nm)
int t_nm;
{
     der_add=t_nm;
}
```

```
/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

```
/*************************************************************/
/*     ACSL Type Simulation Using the Serial 4th Order       */
/*     Runge-Kutta Method of Integration.                    */
/*                                                           */
/*                    by B. Earl Wells                       */
/*                  University of Alabama                    */
/*                                                           */
/*              Last Update: September 25, 1987              */
/*************************************************************/


#include <stdio.h>
#include <math.h>
/* Maximum Size allowed for Internal Arrays      */
/* (represents the maximum number of state       */
/*  variables possible in application plus one). */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int t_flg,s_var,der_add,der_mul;
double max_err;

main()
{
     char ch[30];

     int i,cint_lmt,cint_num;
     double num_calls,time_var;

     double atof();
     double yj[SZ],k1[SZ],k2[SZ],k3[SZ],k4[SZ];
     double f[SZ],y[SZ],t,h,tj,cint,lst;

     void initial(),int_val(),derivative(),dynamic(),output();
     void header(),termt(),st_var(),rel_time(),err();

     printf("\n\n\n");
     printf("      ACSL Type Continuous Simulation using the");
     printf(" Serial form of\n");
     printf("        the Fourth Order Runge-Kutta Method of");
     printf(" Integration.\n\n");
     printf("                          by B. Earl Wells\n\n");

     header();

     tj=0;
     lst=0;
     time_var=0;
     num_calls=0;
     max_err=0;
```

```
printf("Enter iteration interval:  ");
gets(ch);
h=atof(ch);

printf("Enter communication interval:  ");
gets(ch);
cint=atof(ch);

printf("Enter the number of communication intervals between");
printf(" outputs: ");
cint_lmt=atoi(gets(ch));
cint_num=cint_lmt;

/* Process the INITIAL Section */
initial();


/* Assign Initial Conditions to State Variable(s) */
int_val(y);            /* process special initial value  */
for (i=1;i<=s_var;++i) /* and load results into an array */
     yj[i]=y[i];

printf("Override the initial conditions of the state ");
printf("variables (Y or N)?");
gets(ch);

if (ch[0]=='y'||ch[0]=='Y') {
     /* override initial conditions specified in initial value */
     /* section and store these results into an array.        */
     for (i=1;i<=s_var;++i) {
          printf("Enter Initial value for y[%d]:  ",i);
          scanf("%s",ch);
          yj[i]=atof(ch);
     }
}
printf("\nBeginning Run\n");


/* Process the DYNAMIC and DERIVATIVE Sections */
for( ; ; ) {
     t=tj;
     for(i=1;i<=s_var;++i)
          y[i]=yj[i];
     t_flg=0;

     num_calls+=1;   /* Increment Derivative counter */
     derivative(t,y,f);
     time_var+=der_add*ADD + der_mul*MUL;

     if (t>=lst-h/2) {
          dynamic(t,y);
          if(t_flg) break;
```

```
        if ((cint_num+=1)>cint_lmt) {
             cint_num=1;
             output(t,y);

        }
        lst=lst+cint;
   }
   if (t_flg) break;

   for(i=1;i<=s_var;++i) {
        k1[i]=f[i]*h;
        y[i]=k1[i]/2+yj[i];
        time_var+=2*MUL+ADD;
   }
   t=t+tj+h/2;
   time_var+=2*ADD+MUL;

   num_calls+=1;   /* Increment Derivative counter */
   derivative(t,y,f);
   time_var+=der_add*ADD + der_mul*MUL;

   for(i=1;i<=s_var;++i) {
        k2[i]=f[i]*h;
        y[i]=k2[i]/2+yj[i];
        time_var+=2*MUL+ADD;
   }


   num_calls+=1;   /* Increment Derivative counter */
   derivative(t,y,f);
   time_var+=der_add*ADD + der_mul*MUL;

   for(i=1;i<=s_var;++i) {
        k3[i]=f[i]*h;
        y[i]=k3[i]+yj[i];
        time_var+=ADD+MUL;
   }
   t=tj+h;
   time_var+=ADD;

   num_calls+=1;   /* Increment Derivative counter */
   derivative(t,y,f);
   time_var+=der_add*ADD + der_mul*MUL;

   for(i=1;i<=s_var;++i) {
        k4[i]=f[i]*h;
        yj[i]=(k4[i]+2*k3[i]+2*k2[i]+k1[i])/6+yj[i];
        time_var+=4*ADD+4*MUL;
   }
   tj=t;


}
```

```
     output(t,y);

     printf("Run Terminated\n\n");

     printf("Number of calls to the DERIVATIVE Section:  % 1.0f\n",
          num_calls);
     printf("Relative Execution Time:  % 1.0f\n",time_var);
     if (max_err>0)
          printf("Maximum local Error is :  %e\n",max_err);


}

/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
     if (num<0)
          num=-num;

     return(num);
}

/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
     t_flg=flg;
}

/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}
```

```
/* This routine stores the number of addition and  */
/* subtraction operations that are executed every  */
/* time the DERIVATIVE section is called into the  */
/* global variable "der_add".                      */
void num_add(t_nm) int t_nm; {
     der_add=t_nm;
}

/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

```
/*************************************************************/
/*    ACSL Type Simulation Using a Adams Parallel 2nd        */
/*    Order Predictor-Corrector Method of Integration        */
/*                (Two Processor Case)                       */
/*                                                           */
/*                  by B. Earl Wells                         */
/*                University of Alabama                      */
/*                                                           */
/*            Last Update: February 1, 1988                  */
/*************************************************************/


#include <stdio.h>

/* Maximum Size allowed for Internal Arrays */
/* (represents the maximum number of state  */
/*  variables in application plus one.)     */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int s_var,t_flg,cint_lmt,cint_num,der_add,der_mul;
double cint,lst,max_err;
double num_calls,time_var;
main()
{
     char ch[30];
     int i,p;
     double atof();
     double yj[SZ],k1[SZ],k2[SZ];
     double f[SZ],y[SZ],f_in[4][SZ],ycn_1[SZ],fcn_1[SZ],fpn_1[SZ];
     double fpn[SZ],fcn[SZ],ycn[SZ],t,h,tj;
     void initial(),int_val(),derivative(),dynamic(),output();
     void predictor(),corrector(),header(),termt(),rel_time(),err();

     printf("\n\n\n");
     printf("   ACSL Type Continuous Simulation using a");
     printf(" parallel version");
     printf(" of the\n");
     printf("    Adam's Second Order Predictor-Corrector Method of");
     printf(" Integration\n");
     printf("                          (Two Processor Case)\n\n");
     printf("                            by B. Earl Wells\n\n");

     header();

     tj=0;
     lst=0;
     time_var=0;
```

```
der_add=0;
der_mul=0;
num_calls=0;
max_err=0;

printf("Enter iteration interval:  ");
h=atof(gets(ch));

printf("Enter communication interval:  ");
cint=atof(gets(ch));

printf("Enter the number of communication intervals between");
printf(" outputs: ");
cint_lmt=atoi(gets(ch));
cint_num=cint_lmt;

/* Process the INITIAL Section */
initial();

/* Assign Initial Conditions to State Variable(s) */
int_val(y);            /* process special initial value */
for (i=1;i<=s_var;++i) /* and load results into an array*/
     yj[i]=y[i];


printf("Override the initial conditions of the state ");
printf("variables (Y or N)?");
gets(ch);

if (ch[0]=='y'||ch[0]=='Y') {
     /* override initial conditions specified in initial value */
     /* section and store these results into an array.        */
     for (i=1;i<=s_var;++i) {
          printf("Enter Initial value for y[%d]:  ",i);
          gets(ch);
          yj[i]=atof(ch);
     }
}
printf("\nBeginning Run\n");


/* Process the DYNAMIC and DERIVATIVE Sections */

/* Find initial four values using the Runge-Kutta Method */
for(p=2;p>=0;--p) {
     t=tj;
     for(i=1;i<=s_var;++i)
          y[i]=yj[i];
     t_flg=0;              /* Clear Termination Flag */
```

```
num_calls+=1;          /* Increment Derivative counter */
time_var+=der_add*ADD+der_mul*MUL;
derivative(t,y,f);


/* Process DYNAMIC Section every Communication Interval */
if (t>=lst-h/2) {

     dynamic(t,y);
     if(t_flg) break;/* exit if Termination Flag is set */

     /* Produce Output every "cint_lmt" number of */
     /* of Communication Intervals                */
     if ((cint_num+=1)>cint_lmt) {
          cint_num=1;
          output(t,y);

     }
     lst=lst+cint;
}
if (t_flg) break;    /* Exit DERIVATIVE/DYNAMIC loop */
                     /* if Termination Flag is set   */

for (i=1;i<=s_var;++i)
    f_in[p][i]=f[i];

if (p==1)
    for (i=1;i<=s_var;++i) {
         ycn_1[i]=y[i];
         fcn_1[i]=f[i];
     }


if (p==0) break;

/* Process First Order Runge-Kutta Equations */
for(i=1;i<=s_var;++i) {
     k1[i]=f[i]*h;
     y[i]=k1[i]/2+yj[i];
     time_var+=2*MUL+ADD;
}

t=t+tj+h/2;
time_var+=2*ADD+MUL;

num_calls+=1;          /* Increment Derivative counter */
time_var+=der_add*ADD+der_mul*MUL;
derivative(t,y,f);
```

```
        /* Process Second Order Runge-Kutta Equations */
        for(i=1;i<=s_var;++i) {
             k2[i]=f[i]*h;
             yj[i]=k2[i]+yj[i];
             time_var+=ADD+MUL;
        }

        t=tj+h;
        time_var+=ADD;

        tj=t;

    }

    /* Initialize Predictor and Corrector Processes */
    predictor(t,ycn_1,fpn_1,f_in,1);
    corrector(t,ycn_1,ycn,f_in,1);

    for (i=1;i<=s_var;++i)
        fpn[i]=f_in[0][i];

    while (t_flg==0) {

        predictor(h,ycn_1,fpn_1,f_in,0);

        corrector(h,fpn,ycn,f_in,0);


        for (i=1;i<=s_var;++i) {
             fpn[i]=fpn_1[i];
             ycn_1[i]=ycn[i];
        }

    }


    printf("Run Terminated\n\n");

    printf("Effective number of calls to the DERIVATIVE Section:");
    printf("  % 1.0f\n",num_calls);
    printf("Relative Execution Time:  % 1.0f\n",time_var);
    if (max_err>0)
        printf("Maximum local Error is :  %e\n",max_err);

}
```

```
void predictor(h,ycn_1,fpn_1,f_in,flg)
double h,ycn_1[],fpn_1[],f_in[4][SZ];
int flg;
{
     static double y[SZ],fpn[SZ],t;
     int i;
     void shift(),derivative();

     if (flg==1) {
          for (i=1;i<=s_var;++i) {
               fpn[i]=f_in[0][i];
          }
          t=h;
     }

     else {

          t=t+h;

          for (i=1;i<=s_var;++i)
               y[i]=ycn_1[i]+2*h*fpn[i];

          derivative(t,y,fpn_1);


          for (i=1;i<=s_var;++i)
               fpn[i]=fpn_1[i];
     }
}

void corrector(h,fpn,ycn,f_in,flg)
double h,fpn[],ycn[],f_in[4][SZ];
int flg;
{
     static double fcn[SZ],ycn_1[SZ],fn_1[SZ],t;
     int i;
     void shift(),derivative(),dynamic(),output();

     if (flg==1) {
          for (i=1;i<=s_var;++i) {
               fn_1[i]=f_in[1][i];
               ycn_1[i]=fpn[i];
          }
          t=h;
     }

     else {
          for (i=1;i<=s_var;++i) {
             ycn[i]=ycn_1[i]+h/2*(fpn[i]+fn_1[i]);
             time_var+=2*ADD+2*MUL;
          }
```

```
          t_flg=0;

          num_calls+=1;          /* Increment Derivative Counter */
          time_var+=der_add*ADD+der_mul*MUL;
          derivative(t,ycn,fcn);

          /* Process DYNAMIC Section every Communication Interval */
          if (t>=lst-h/2) {

               dynamic(t,ycn);
               if(t_flg==0) {

                    /* Produce Output every "cint_lmt" number of */
                    /* of Communication Intervals                */
                    if ((cint_num+=1)>cint_lmt) {
                         cint_num=1;
                         output(t,ycn);
                    }
               }
               lst=lst+cint;
          }

          if (t_flg) output(t,ycn); /* Produce Final Output */


          for (i=1;i<=s_var;++i) {
               fn_1[i]=fcn[i];
               ycn_1[i]=ycn[i];
          }

          t=t+h;
          time_var+=ADD;

     }
}

/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
     if (num<0)
          num=-num;

     return(num);
}

/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
     t_flg=flg;
}
```

```
/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}

/* This routine stores the number of addition and  */
/* subtraction operations that are executed every */
/* time the DERIVATIVE section is called into the */
/* global variable "der_add".                     */
void num_add(t_nm)
int t_nm;
{
     der_add=t_nm;
}

/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

```
/*************************************************************/
/*    ACSL Type Simulation Using a Adams Parallel 4th        */
/*    Order Predictor-Corrector Method of Integration        */
/*                (Two Processor Case)                       */
/*                                                           */
/*                  by B. Earl Wells                         */
/*                University of Alabama                      */
/*                                                           */
/*            Last Update: February 1, 1988                  */
/*************************************************************/


#include <stdio.h>

/* Maximum Size allowed for Internal Arrays */
/* (represents the maximum number of state  */
/*  variables in application plus one.)     */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int s_var,t_flg,cint_lmt,cint_num,der_add,der_mul;
double cint,lst,max_err;
double num_calls,time_var;
main()
{
     char ch[30];
     int i,p;
     double atof();
     double yj[SZ],k1[SZ],k2[SZ];
     double f[SZ],y[SZ],f_in[4][SZ],ycn_1[SZ],fcn_1[SZ],fpn_1[SZ];
     double fpn[SZ],fcn[SZ],ycn[SZ],t,h,tj;
     void initial(),int_val(),derivative(),dynamic(),output();
     void predictor(),corrector(),header(),termt(),rel_time(),err();

     printf("\n\n\n");
     printf("   ACSL Type Continuous Simulation using a parallel");
     printf(" version");
     printf(" of the\n");
     printf("    Adam's Fourth Order Predictor-Corrector Method of");
     printf(" Integration\n");
     printf("                          (Two Processor Case)\n\n");
     printf("                            by B. Earl Wells\n\n");

     header();

     tj=0;
     lst=0;
     time_var=0;
```

```
der_add=0;
der_mul=0;
num_calls=0;
max_err=0;

printf("Enter iteration interval:  ");
h=atof(gets(ch));

printf("Enter communication interval:  ");
cint=atof(gets(ch));

printf("Enter the number of communication intervals between");
printf(" outputs: ");
cint_lmt=atoi(gets(ch));
cint_num=cint_lmt;

/* Process the INITIAL Section */
initial();

/* Assign Initial Conditions to State Variable(s) */
int_val(y);            /* process special initial value section */
for (i=1;i<=s_var;++i) /* and load results into an array.       */
     yj[i]=y[i];


printf("Override the initial conditions of the state ");
printf("variables (Y or N)?");
gets(ch);

if (ch[0]=='y'||ch[0]=='Y') {
     /* override initial conditions specified in initial value */
     /* section and store these results into an array.        */
     for (i=1;i<=s_var;++i) {
          printf("Enter Initial value for y[%d]:  ",i);
          gets(ch);
          yj[i]=atof(ch);
     }
}
printf("\nBeginning Run\n");


/* Process the DYNAMIC and DERIVATIVE Sections */

/* Find initial four values using the Runge-Kutta Method */
for(p=3;p>=0;--p) {
     t=tj;
     for(i=1;i<=s_var;++i)
          y[i]=yj[i];
     t_flg=0;              /* Clear Termination Flag */
```

```
num_calls+=1;          /* Increment Derivative counter */
time_var+=der_add*ADD+der_mul*MUL;
derivative(t,y,f);


/* Process DYNAMIC Section every Communication Interval */
if (t>=lst-h/2) {

     dynamic(t,y);
     if(t_flg) break;  /* exit if Termination Flag is set */

     /* Produce Output every "cint_lmt" number of */
     /* of Communication Intervals                */
     if ((cint_num+=1)>cint_lmt) {
          cint_num=1;
          output(t,y);

     }
     lst=lst+cint;
}
if (t_flg) break;    /* Exit DERIVATIVE/DYNAMIC loop */
                     /* if Termination Flag is set   */

for (i=1;i<=s_var;++i)
    f_in[p][i]=f[i];

if (p==1)
    for (i=1;i<=s_var;++i) {
         ycn_1[i]=y[i];
         fcn_1[i]=f[i];
     }


if (p==0) break;

/* Process First Order Runge-Kutta Equations */
for(i=1;i<=s_var;++i) {
     k1[i]=f[i]*h;
     y[i]=k1[i]/2+yj[i];
     time_var+=2*MUL+ADD;
}

t=t+tj+h/2;
time_var+=2*ADD+MUL;

num_calls+=1;          /* Increment Derivative counter */
time_var+=der_add*ADD+der_mul*MUL;
derivative(t,y,f);
```

```
        /* Process Second Order Runge-Kutta Equations */
        for(i=1;i<=s_var;++i) {
             k2[i]=f[i]*h;
             yj[i]=k2[i]+yj[i];
             time_var+=ADD+MUL;
        }

        t=tj+h;
        time_var+=ADD;

        tj=t;

    }

    /* Initialize Predictor and Corrector Processes */
    predictor(t,ycn_1,fcn_1,fpn_1,f_in,1);
    corrector(t,ycn_1,ycn,fcn,f_in,1);

    for (i=1;i<=s_var;++i)
        fpn[i]=f_in[0][i];

    while (t_flg==0) {

        predictor(h,ycn_1,fcn_1,fpn_1,f_in,0);

        corrector(h,fpn,ycn,fcn,f_in,0);


        for (i=1;i<=s_var;++i) {
             fpn[i]=fpn_1[i];
             ycn_1[i]=ycn[i];
             fcn_1[i]=fcn[i];
        }

    }


    printf("Run Terminated\n\n");

    printf("Effective number of calls to the DERIVATIVE Section:");
    printf("  % 1.0f\n",num_calls);
    printf("Relative Execution Time:  % 1.0f\n",time_var);
    if (max_err>0)
        printf("Maximum local Error is :  %e\n",max_err);

}
```

```
void predictor(h,ycn_1,fcn_1,fpn_1,f_in,flg)
double h,ycn_1[],fcn_1[],fpn_1[],f_in[4][SZ];
int flg;
{
     static double y[SZ],fpn[SZ],fn[3][SZ],t;
     int i;
     void shift(),derivative();

     if (flg==1) {
          for (i=1;i<=s_var;++i) {
               fpn[i]=f_in[0][i];
               fn[0][i]=f_in[2][i];
               fn[1][i]=f_in[3][i];
          }
          t=h;
     }

     else {
          shift(fcn_1,fn);

          t=t+h;

          for (i=1;i<=s_var;++i)
               y[i]=ycn_1[i]+h/3*(8*fpn[i]-5*fn[0][i]+4*fn[1][i]
                    -fn[2][i]);

          derivative(t,y,fpn_1);


          for (i=1;i<=s_var;++i)
               fpn[i]=fpn_1[i];
     }
}

void corrector(h,fpn,ycn,fcn,f_in,flg)
double h,fpn[],ycn[],fcn[],f_in[4][SZ];
int flg;
{
     static double ycn_1[SZ],fn[3][SZ],t;
     int i;
     void shift(),derivative(),dynamic(),output();

     if (flg==1) {
          for (i=1;i<=s_var;++i) {
               fn[0][i]=f_in[1][i];
               fn[1][i]=f_in[2][i];
               fn[2][i]=f_in[3][i];
               ycn_1[i]=fpn[i];
          }
          t=h;
     }
```

```
     else {
          for (i=1;i<=s_var;++i) {
             ycn[i]=ycn_1[i]+h/24*(9*fpn[i]+19*fn[0][i]
                  -5*fn[1][i]+fn[2][i]);
             time_var+=4*ADD+5*MUL;
          }

          t_flg=0;

          num_calls+=1;          /* Increment Derivative Counter */
          time_var+=der_add*ADD+der_mul*MUL;
          derivative(t,ycn,fcn);

          /* Process DYNAMIC Section every Communication Interval */
          if (t>=lst-h/2) {

               dynamic(t,ycn);
               if(t_flg==0) {

                    /* Produce Output every "cint_lmt" number of */
                    /* of Communication Intervals                */
                    if ((cint_num+=1)>cint_lmt) {
                         cint_num=1;
                         output(t,ycn);
                    }
               }
               lst=lst+cint;
          }

          if (t_flg) output(t,ycn); /* Produce Final Output */

          shift(fcn,fn);

          for (i=1;i<=s_var;++i)
               ycn_1[i]=ycn[i];

          t=t+h;
          time_var+=ADD;

     }
}

void shift(x,y)
double x[],y[3][SZ];
{
     int i;

     for (i=1;i<=s_var;++i) {
          y[2][i]=y[1][i];
          y[1][i]=y[0][i];
          y[0][i]=x[i];
     }
}
```

```
/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
     if (num<0)
          num=-num;

     return(num);
}

/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
     t_flg=flg;
}

/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}

/* This routine stores the number of addition and  */
/* subtraction operations that are executed every */
/* time the DERIVATIVE section is called into the */
/* global variable "der_add".                     */
void num_add(t_nm)
int t_nm;
{
     der_add=t_nm;
}
```

```
/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

```
/*************************************************************/
/*    ACSL Type Simulation Using a Adams Parallel 2nd        */
/*    Order Predictor-Corrector Method of Integration        */
/*                (Four Processor Case)                      */
/*                                                           */
/*                  by B. Earl Wells                         */
/*                University of Alabama                      */
/*                                                           */
/*            Last Update: February 1, 1988                  */
/*************************************************************/


#include <stdio.h>

/* Maximum Size allowed for Internal Arrays */
/* (represents the maximum number of state  */
/*  variables in application plus one.)     */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int s_var,t_flg,cint_lmt,cint_num,der_add,der_mul;
double cint,lst,max_err;
double num_calls,time_var;

main()
{
     char ch[30];
     int i,p;
     double atof();
     double yj[SZ],k1[SZ],k2[SZ];
     double f[SZ],y[SZ],f_in[4][SZ],y_in[4][SZ];
     double fp[SZ],fp_1[SZ],yc_2[SZ],yc_3[SZ],fc_2[SZ];
     double fpa_1[SZ],fpa_2[SZ],yc_1[SZ],yc[SZ],fc[SZ];
     double t,h,tj;

     void initial(),int_val(),derivative(),dynamic(),output();
     void predictor(),corrector(),header(),termt(),rel_time(),err();
     void pred_1(),pred_2(),correct_1(),correct_2();

     printf("\n\n\n");
     printf("   ACSL Type Continuous Simulation using a parallel");
     printf(" version");
     printf(" of the\n");
     printf("    Adam's Second Order Predictor-Corrector Method of");
     printf(" Integration\n");
     printf("                          (Four Processor Case)\n\n");
     printf("                             by B. Earl Wells\n\n");
```

```
header();

tj=0;
lst=0;
time_var=0;
num_calls=0;
max_err=0;

printf("Enter iteration interval:  ");
h=atof(gets(ch));

printf("Enter communication interval:  ");
cint=atof(gets(ch));

printf("Enter the number of communication intervals between");
printf(" outputs: ");
cint_lmt=atoi(gets(ch));
cint_num=cint_lmt;

/* Process the INITIAL Section */
initial();

/* Assign Initial Conditions to State Variable(s) */
int_val(y);             /* process special initial value  */
for (i=1;i<=s_var;++i) /* and load results into an array */
     yj[i]=y[i];


printf("Override the initial conditions of the state ");
printf("variables (Y or N)?");
gets(ch);

if (ch[0]=='y'||ch[0]=='Y') {
     /* override initial conditions specified in initial value */
     /* section and store these results into an array.        */
     for (i=1;i<=s_var;++i) {
          printf("Enter Initial value for y[%d]:  ",i);
          gets(ch);
          yj[i]=atof(ch);
     }
}
printf("\nBeginning Run\n");


/* Process the DYNAMIC and DERIVATIVE Sections */

/* Find initial three values using the Runge-Kutta Method */
for(p=0;p<=3;++p) {
     t=tj;
     for(i=1;i<=s_var;++i)
          y[i]=yj[i];
     t_flg=0;               /* Clear Termination Flag */
```

```
num_calls+=1;          /* Increment Derivative counter */
time_var+=der_add*ADD+der_mul*MUL;
derivative(t,y,f);


/* Process DYNAMIC Section every Communication Interval */
if (t>=lst-h/2) {

     dynamic(t,y);
     if(t_flg) break; /* exit if Termination Flag is set */

     /* Produce Output every "cint_lmt" number of */
     /* of Communication Intervals                */
     if ((cint_num+=1)>cint_lmt) {
          cint_num=1;
          output(t,y);

     }
     lst=lst+cint;
}
if (t_flg) break;    /* Exit DERIVATIVE/DYNAMIC loop */
                     /* if Termination Flag is set   */

for (i=1;i<=s_var;++i) {
    f_in[p][i]=f[i];
    y_in[p][i]=y[i];
}

if (p==3) break;

/* Process First Order Runge-Kutta Equations */
for(i=1;i<=s_var;++i) {
     k1[i]=f[i]*h;
     y[i]=k1[i]/2+yj[i];
     time_var+=2*MUL+ADD;
}

t=t+tj+h/2;
time_var+=2*ADD+MUL;

num_calls+=1;          /* Increment Derivative counter */
time_var+=der_add*ADD+der_mul*MUL;
derivative(t,y,f);

/* Process Second Order Runge-Kutta Equations */
for(i=1;i<=s_var;++i) {
     k2[i]=f[i]*h;
     yj[i]=k2[i]+yj[i];
     time_var+=ADD+MUL;
}
```

```
        t=tj+h;
        time_var+=ADD;

        tj=t;

    }

    for (i=1;i<=s_var;++i) {
        fp[i]   = f_in[3][i];
        fp_1[i] = f_in[2][i];
        fc_2[i] = f_in[1][i];
        yc_2[i] = y_in[1][i];
        yc_3[i] = y_in[0][i];
    }

    pred_1(t+2*h,fp,yc_2,fpa_2,0);
    pred_2(t+h,fp_1,fp,yc_2,fpa_1,0);

    correct_1(t,fp_1,fp,yc_3,fc,yc,0);
    correct_2(t-h,fc_2,yc_3,yc_1,0);

    do {
        pred_1(h,fp,yc_2,fpa_2,1);
        pred_2(h,fp_1,fp,yc_2,fpa_1,1);

        correct_2(h,fc_2,yc_3,yc_1,1);
        correct_1(h,fp_1,fp,yc_3,fc,yc,1);

        for (i=1;i<=s_var;++i) {
            fp[i]   = fpa_2[i];
            fp_1[i] = fpa_1[i];
            yc_2[i] = yc[i];
            fc_2[i] = fc[i];
            yc_3[i] = yc_1[i];
        }

    } while (t_flg==0);


    printf("Run Terminated\n\n");

    printf("Effective of calls to the DERIVATIVE Section:");
    printf("  % 1.0f\n",num_calls);
    printf("Relative Execution Time:  % 1.0f\n",time_var);
    if (max_err>0)
        printf("Maximum local Error is :  %e\n",max_err);

}
```

```
/* Predictor #1 (first processor) */
void pred_1(h,fp,yc_2,fpa_2,flg)
double h,fp[SZ],yc_2[SZ],fpa_2[SZ];
int flg;
{
     static double t;
     int i;
     double ypa_2[SZ];
     void derivative();

     /* Initialize time */

     if (flg==0)
          t=h;
     else {
          for(i=1;i<=s_var;++i) {
               ypa_2[i] = yc_2[i] + 4*h*fp[i];
          }

          derivative(t,ypa_2,fpa_2);

          t=t+h*2;
          t_flg=0;
     }

}

/* Predictor #2 (second processor) */
void pred_2(h,fp_1,fp,yc_2,fpa_1,flg)
double h,fp_1[SZ],fp[SZ],yc_2[SZ],fpa_1[SZ];
int flg;
{
     static double t;
     double ypa_1[SZ];
     int i;
     void derivative();

     /* Initialize time */
     if (flg==0)
          t=h;
     else {
          for(i=1;i<=s_var;++i) {
               ypa_1[i]=yc_2[i]+3*h*(fp[i]+fp_1[i])/2;
          }

          derivative(t,ypa_1,fpa_1);

          t=t+h*2;
          t_flg=0;
     }


}
```

```
/* Corrector #1  (third processor) */
void correct_1(h,fp_1,fp,yc_3,fc,yc,flg)
double h,fp_1[SZ],fp[SZ],yc_3[SZ],fc[SZ],yc[SZ];
int flg;
{
     static double t;
     int i;
     void derivative(),dynamic(),output();

     /* Initialize time */
     if (flg==0||t_flg!=0)
          t=h;
     else {
          for (i=1;i<=s_var;++i) {
               yc[i]=yc_3[i]-h*(3*fp[i]-9*fp_1[i])/2;
               time_var+=4*MUL+2*ADD;
          }

          num_calls+=1;          /* Increment Derivative counter */
          time_var+=der_add*ADD+der_mul*MUL;
          derivative(t,yc,fc);

          /* Process DYNAMIC Section every Communication Interval */
          if (t>=lst-h/2) {

               dynamic(t,yc);
               if(t_flg==0) {  /* execute only if Termination Flag */
                               /* is not set                       */

                    /* Produce Output every "cint_lmt" number of */
                    /* of Communication Intervals                */
                    if ((cint_num+=1)>cint_lmt) {
                         cint_num=1;
                         output(t,yc);
                    }
                    lst=lst+cint;
               }
          }
          if (t_flg!=0)
               output(t,yc);
          t=t+2*h;
          time_var+=ADD+MUL;
     }
}
```

```
/* Corrector #2 (fourth processor) */
void correct_2(h,fc_2,yc_3,yc_1,flg)
double h,fc_2[SZ],yc_3[SZ],yc_1[SZ];
int flg;
{
     static double t;
     int i;
     double fc_1[SZ];
     void derivative(),dynamic(),output();

     /* Initialize time */
     if (flg==0||t_flg!=0)
          t=h;
     else {
          for (i=1;i<=s_var;++i) {
               yc_1[i]=yc_3[i]+2*h*fc_2[i];
          }

          derivative(t,yc_1,fc_1);

          /* Process DYNAMIC Section every Communication Interval */
          if (t>=lst-h/2) {
               dynamic(t,yc_1);
               if(t_flg==0) {  /* execute only if Termination */
                               /* Flag is not set             */

                    /* Produce Output every "cint_lmt" number of */
                    /* of Communication Intervals                */
                    if ((cint_num+=1)>cint_lmt) {
                         cint_num=1;
                         output(t,yc_1);
                    }
                    lst=lst+cint;
               }
          }

          if (t_flg!=0)
               output(t,yc_1);

          t=t+2*h;
     }
}

/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
     if (num<0)
          num=-num;

     return(num);
}
```

```
/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
     t_flg=flg;
}

/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}

/* This routine stores the number of addition and  */
/* subtraction operations that are executed every */
/* time the DERIVATIVE section is called into the */
/* global variable "der_add".                     */
void num_add(t_nm)
int t_nm;
{
     der_add=t_nm;
}

/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

```
/*************************************************************/
/*    ACSL Type Simulation Using the Parallel 4th Order      */
/*     Block Predictor-Corrector Method of Integration       */
/*                                                           */
/*                    by B. Earl Wells                       */
/*                 University of Alabama                     */
/*                                                           */
/*             Last Update: February 1, 1988                 */
/*************************************************************/


#include <stdio.h>

/* Maximum Size allowed for Internal Arrays */
/* (represents the maximum number of state  */
/*  variables in application plus one.)     */
#define SZ 21

/* Relative execution time for multiply and add operations */
#define MUL 1
#define ADD 1

/* Declare variables that are global to entire program */
int s_var,t_flg,cint_lmt,cint_num,der_add,der_mul;
double cint,lst,max_err;
double num_calls,time_var;
main()
{
     char ch[30];
     int i,p;
     double atof();
     double yj[SZ],k1[SZ],k2[SZ];
     double f[SZ],y[SZ],f_in[4][SZ],y_in[4][SZ],yc[SZ],fc[SZ];
     double yc_1[SZ],fc_1[SZ],fp_1[SZ],fp_2[SZ];
     double ycp_1[SZ],ycp_2[SZ],fcp_1[SZ],fcp_2[SZ],t,h,tj;

     void initial(),int_val(),derivative(),dynamic(),output();
     void header(),termt(),st_var(),rel_time(),err();
     void pred_1(),pred_2(),correct_1(),correct_2();

     printf("\n\n\n");
     printf("   ACSL Type Continuous Simulation using");
     printf(" a Parallel Block");
     printf(" form of\n");

     printf("     the Fourth Order Predictor-Corrector Method of");
     printf(" Integration\n");
     printf("                        (Two Processor Case)\n\n");
     printf("                          by B. Earl Wells\n\n");

     header();
```

```
tj=0;
lst=0;
time_var=0;
der_add=0;
der_mul=0;
num_calls=0;
max_err=0;

printf("Enter iteration interval:  ");
h=atof(gets(ch));

printf("Enter communication interval:  ");
cint=atof(gets(ch));

printf("Enter the number of communication intervals between");
printf(" outputs: ");
cint_lmt=atoi(gets(ch));
cint_num=cint_lmt;

/* Process the INITIAL Section */
initial();

/* Assign Initial Conditions to State Variable(s) */
int_val(y);            /* process special initial value  */
for (i=1;i<=s_var;++i) /* and load results into an array */
     yj[i]=y[i];


printf("Override the initial conditions of the state ");
printf("variables (Y or N)?");
gets(ch);

if (ch[0]=='y'||ch[0]=='Y') {
     /* override initial conditions specified in initial value */
     /* section and store these results into an array.        */
     for (i=1;i<=s_var;++i) {
          printf("Enter Initial value for y[%d]:  ",i);
          gets(ch);
          yj[i]=atof(ch);
     }
}
printf("\nBeginning Run\n");


/* Process the DYNAMIC and DERIVATIVE Sections */

/* Find initial three values using the Runge-Kutta Method */
for(p=0;p<=2;++p) {
     t=tj;
     for(i=1;i<=s_var;++i)
          y[i]=yj[i];
```

```
t_flg=0;              /* Clear Termination Flag */

num_calls+=1;         /* Increment Derivative counter */
time_var+=der_add*ADD+der_mul*MUL;
derivative(t,y,f);


/* Process DYNAMIC Section every Communication Interval */
if (t>=lst-h/2) {

     dynamic(t,y);
     if(t_flg) break; /* exit if Termination Flag is set */

     /* Produce Output every "cint_lmt" number of */
     /* of Communication Intervals                */
     if ((cint_num+=1)>cint_lmt) {
          cint_num=1;
          output(t,y);

     }
     lst=lst+cint;
}
if (t_flg) break;    /* Exit DERIVATIVE/DYNAMIC loop */
                     /* if Termination Flag is set   */

for (i=1;i<=s_var;++i) {
    f_in[p][i]=f[i];
    y_in[p][i]=y[i];
}

if (p==2) break;

/* Process First Order Runge-Kutta Equations */
for(i=1;i<=s_var;++i) {
     k1[i]=f[i]*h;
     y[i]=k1[i]/2+yj[i];
     time_var+=2*MUL+ADD;
}
t=t+tj+h/2;
time_var+=2*ADD+MUL;

num_calls+=1;         /* Increment Derivative counter */
time_var+=der_add*ADD+der_mul*MUL;
derivative(t,y,f);

/* Process Second Order Runge-Kutta Equations */
for(i=1;i<=s_var;++i) {
     k2[i]=f[i]*h;
     yj[i]=k2[i]+yj[i];
     time_var+=ADD+MUL;
}

t=tj+h;
time_var+=ADD;
```

```
        tj=t;

    }

    for (i=1;i<=s_var;++i) {
        yc[i]   = y_in[2][i];
        yc_1[i] = y_in[1][i];
        fc[i]   = f_in[2][i];
        fc_1[i] = f_in[1][i];
    }

    pred_1(t+h,yc_1,y_in[0],fc_1,f_in[0],fp_1,0);
    pred_2(t+2*h,yc_1,y_in[0],fc_1,f_in[0],fp_2,0);

    correct_1(t+h,yc,fc,fp_1,fp_2,ycp_1,fcp_1,0);
    correct_2(t+2*h,yc,fc,fp_1,fp_2,ycp_2,fcp_2,0);

    do {
        pred_1(h,yc_1,yc,fc_1,fc,fp_1,1);
        pred_2(h,yc_1,yc,fc_1,fc,fp_2,1);

        correct_1(h,yc,fc,fp_1,fp_2,ycp_1,fcp_1,1);
        correct_2(h,yc,fc,fp_1,fp_2,ycp_2,fcp_2,1);

        for (i=1;i<=s_var;++i) {
            fc[i]=fcp_2[i];
            yc[i]=ycp_2[i];
            fc_1[i]=fcp_1[i];
            yc_1[i]=ycp_1[i];
        }

    } while (t_flg==0);


    printf("Run Terminated\n\n");

    printf("Effective number of calls to the DERIVATIVE Section:");
    printf("  % 1.0f\n",num_calls);
    printf("Relative Execution Time:  % 1.0f\n",time_var);
    if (max_err>0)
        printf("Maximum local Error is :  %e\n",max_err);

}

/* Predictor Block i+1 (first processor) */
void pred_1(h,yc_1,yc,fc_1,fc,fp_1,flg)
double h,yc_1[SZ],yc[SZ],fc_1[SZ],fc[SZ],fp_1[SZ];
int flg;
{
    static double t,yc_2[SZ],fc_2[SZ],ypp_1[SZ];
    int i;
    void derivative();
```

```
     /* Initialize time */
     if (flg==0)
          t=h;
     else {
          for(i=1;i<=s_var;++i) {
               ypp_1[i] = (yc_2[i]+yc_1[i]+yc[i])/3
                    + h/6*(3*fc_2[i]-4*fc_1[i]+13*fc[i]);
               time_var+=6*MUL+4*ADD;
          }

          num_calls+=1;        /* Increment Derivative counter */
          time_var+=der_add*ADD+der_mul*MUL;
          derivative(t,ypp_1,fp_1);

          t=t+h*2;
          time_var+=ADD+MUL;
     }

     for (i=1;i<=s_var;++i) {
          yc_2[i]=yc[i];
          fc_2[i]=fc[i];
     }
     t_flg=0;
}

/* Corrector Block i+1 (first processor) */
void correct_1(h,yc,fc,fp_1,fp_2,ycp_1,fcp_1,flg)
double h,yc[SZ],fc[SZ],fp_1[SZ],fp_2[SZ],ycp_1[SZ],fcp_1[SZ];
int flg;
{
     static double t;
     int i;
     void derivative(),dynamic(),output();

     /* Initialize time */
     if (flg==0||t_flg!=0)
          t=h;
     else {
          for (i=1;i<=s_var;++i) {
               ycp_1[i]=yc[i]+h*(5*fc[i]+8*fp_1[i]-fp_2[i])/12;
               time_var+=3*ADD+4*MUL;
          }

          num_calls+=1;        /* Increment Derivative counter */
          time_var+=der_add*ADD+der_mul*MUL;
          derivative(t,ycp_1,fcp_1);

          /* Process DYNAMIC Section every Communication Interval */
          if (t>=lst-h/2) {

               dynamic(t,ycp_1);
               if(t_flg==0) {  /* execute only if Termination */
                               /* Flag is not set             */
```

```
                    /* Produce Output every "cint_lmt" number of */
                    /* of Communication Intervals                */
                    if ((cint_num+=1)>cint_lmt) {
                         cint_num=1;
                         output(t,ycp_1);
                    }
                    lst=lst+cint;
               }
          }
          if (t_flg!=0)
               output(t,ycp_1);
          t=t+2*h;
          time_var+=ADD+MUL;
     }
}


/* Predictor Block i+2 (second processor) */
void pred_2(h,yc_1,yc,fc_1,fc,fp_2,flg)
double h,yc_1[SZ],yc[SZ],fc_1[SZ],fc[SZ],fp_2[SZ];
int flg;
{
     static double t,yc_2[SZ],fc_2[SZ],ypp_2[SZ];
     int i;
     void derivative();

     /* Initialize time */
     if (flg==0)
          t=h;
     else {
          for(i=1;i<=s_var;++i) {
               ypp_2[i] = (yc_2[i]+yc_1[i]+yc[i])/3
                    + h/12*(29*fc_2[i]-72*fc_1[i]+79*fc[i]);
          }

          derivative(t,ypp_2,fp_2);

          t=t+h*2;
     }

     for (i=1;i<=s_var;++i) {
          yc_2[i]=yc[i];
          fc_2[i]=fc[i];
     }

     t_flg=0;
}
```

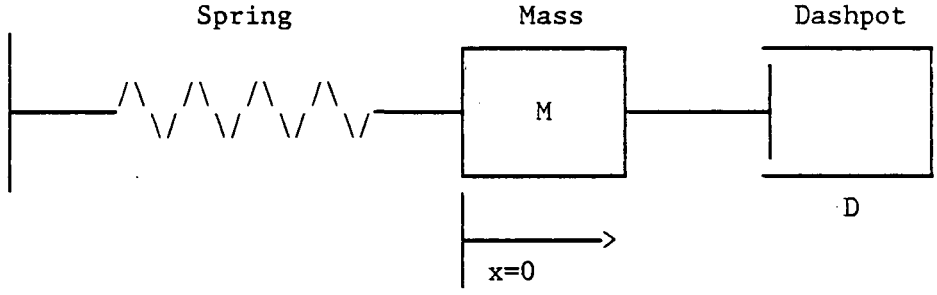
```
/* Corrector Block i+2 (second processor) */
void correct_2(h,yc,fc,fp_1,fp_2,ycp_2,fcp_2,flg)
double h,yc[SZ],fc[SZ],fp_1[SZ],fp_2[SZ],ycp_2[SZ],fcp_2[SZ];
int flg;
{
     static double t;
     int i;
     void derivative(),dynamic(),output();

     /* Initialize time */
     if (flg==0||t_flg!=0)
          t=h;
     else {
          for (i=1;i<=s_var;++i) {
               ycp_2[i]=yc[i]+h*(fc[i]+4*fp_1[i]+fp_2[i])/3;
          }

          derivative(t,ycp_2,fcp_2);

          /* Process DYNAMIC Section every Communication Interval */
          if (t>=lst-h/2) {

               dynamic(t,ycp_2);
               if(t_flg==0) {  /* execute only if Termination */
                               /* Flag is not set             */

                    /* Produce Output every "cint_lmt" number of */
                    /* of Communication Intervals                */
                    if ((cint_num+=1)>cint_lmt) {
                         cint_num=1;
                         output(t,ycp_2);
                    }
                    lst=lst+cint;
               }
          }

          if (t_flg!=0)
               output(t,ycp_2);

          t=t+2*h;
     }
}

/* This routine takes the absolute value of */
/* a double precision argument.             */
double abs(num)
double num;
{
     if (num<0)
          num=-num;

     return(num);
}
```

```
/* This routine sets and resets the global */
/* termination flag.                       */
void termt(flg)
int flg;
{
     t_flg=flg;
}

/* This routine stores the number of state variables */
/* present in the application problem into the       */
/* global variable "s_var".                          */
void st_var(n)
int(n);
{
     s_var=n;
}

/* This routine stores the number of multiplication */
/* and division operations that are executed every  */
/* time the DERIVATIVE section is called into the   */
/* global variable "der_mul".                       */
void num_mul(t_nm)
int t_nm;
{
     der_mul=t_nm;
}

/* This routine stores the number of addition and  */
/* subtraction operations that are executed every */
/* time the DERIVATIVE section is called into the */
/* global variable "der_add".                     */
void num_add(t_nm)
int t_nm;
{
     der_add=t_nm;
}

/* This routine stores the maximum local error */
/* in the global flag "max_err".               */
void err(num)
double num;
{
     double abs();

     if (abs(num)>max_err)
          max_err=abs(num);
}
```

APPENDIX C
BENCHMARK EXAMPLES

Spring Dashpot Example

The Spring Dashpot Example represents a physical system in which a mass is placed in damped harmonic motion. The damping is caused by the dashpot mechanism that converts mechanical energy into heat energy.



This system is described by the second order differential equation

$$M\ddot{X} + D\dot{X} + KX = 0,$$

where M is the mass,
D is the Damping, and
K is the spring constant.

This can be transposed into a set of two first order relationships modeled using the integration operator.

The governing initial conditions are

$$X(0) = 5, \text{ and } \dot{X}(0) = 0.$$

The equation describing the analytical solution used to compute relative error is

```
x=sqrt(3.0)*10.0*exp(-2.0*t)*cos(2.0*sqrt(3.0)*t-PI/6)/3.0.
```

```
/*********************************************/
/*         APPLICATION PROGRAM AREA          */
/*********************************************/


/*         Spring Damping Example            */

double xic,xdic,k,d,m,tstp,xdd;

/*********** HEADER INFORMATION ************/
void header()
{
printf("                ***********************************\n");
printf("                ***** SPRING DASHPOT EXAMPLE *****\n");
printf("                ***********************************\n\n");
}
/*********************************************/

/************** INITIAL SECTION *************/
void initial()
{
/* Define Preset variables  */
     xic=5.0;
     xdic=0.0;
     m=1;
     k=16.0;
     d=4.0;
    tstp=10.00; } /*********************************************/



/**** INITIAL VALUES OF STATE VARIABLES ****/
void int_val(y)
double y[];
{
     y[1]=xic;
     y[2]=xdic;

    /* Specify Number of State Variables  */
    st_var(2);

    /* Specify Relative Execution Time of */
    /* DERIVATIVE section                 */
    num_mul(3);
    num_add(2);

}
/*********************************************/
```

```
/************ DERIVATIVE SECTION ************/
void derivative(t,y,f)
double t,y[],f[];
{

     xdd=(-d*y[2]-k*y[1])/m;
     f[1]=y[2];
     f[2]=xdd;
     termt(t>tstp);
}
/*******************************************/



/************** DYNAMIC SECTION **************/
void dynamic(t,y)
double t,y[];
{
     double sqrt(),exp(),cos(),x;
     x=sqrt(3.0)*10.0*exp(-2.0*t)*cos(2.0*sqrt(3.0)
             *t-3.1415926536/6)/3.0;
     if (x!=0)
       err((y[1]-x)/x);

}
/*******************************************/



/************ OUTPUT STATEMENT(S) ************/
void output(t,y)
double t,y[];
{
     double sqrt(),cos(),exp();
     printf("t=% 12.8f xdd=% 12.8f xd=% 3.8f x=% 3.8f\n",
          t,xdd,y[2],y[1]);
     printf("x=% 12.8f\n",sqrt(3.0)*10.0*exp(-2.0*t)
          *cos(2.0*sqrt(3.0)*t-3.1415926536/6)/3.0);
}
/*******************************************/
```

Table E
Spring Dashpot Example
Effective Number of Derivative Calls

SPRING DASHPOT EXAMPLE Effective Number of Derivative Function Calls			
h	P4	P42	B42
0.4	51	30	29
0.1	203	106	105
0.05	401	205	205
0.01	2003	1006	1005
0.005	4001	2005	2005
0.001	20003	10006	10003
0.0005	40001	20005	20005
0.0001	200003	100006	100003

SPRING DASHPOT EXAMPLE Effective Number of Derivative Function Calls			
h	P2	P22	P24
0.4	51	29	19
0.1	203	105	57
0.05	401	204	107
0.01	2003	1005	507
0.005	4001	2004	1007
0.001	20003	10005	5006
0.0005	40001	20004	10007
0.0001	200003	100005	50006

SPRING DASHPOT EXAMPLE Effective Number of Derivative Function Calls				
h	ER	TP*	R2	R4
0.4	26	1684	51	101
0.1	102	819	203	405
0.05	201	1154	401	801
0.01	1002	3651	2003	4005
0.005	2001	6484	4001	8001
0.001	10002	26446	20003	40005
0.0005	20001	49401	40001	80001
0.0001	100002	223028	200003	400005

*Error coefficient set at 1×10^{-8}

Table F
Spring Dashpot Example
Effective Number of Floating Point Operations

SPRING DASHPOT EXAMPLE Effective Number of Floating Point Operations			
h	P4	P42	B42
0.4	1133	629	629
0.1	4781	2453	2453
0.05	9533	4829	4853
0.01	47981	24053	24053
0.005	95933	48029	48053
0.001	479981	240053	240005
0.0005	959933	480029	480053
0.0001	4799981	2400053	2400005

SPRING DASHPOT EXAMPLE Effective Number of Floating Point Operations			
h	P2	P22	P24
0.4	701	389	305
0.1	2829	1453	1027
0.05	5601	2839	1977
0.01	28029	14053	9577
0.005	56001	28039	19077
0.001	280029	140053	95058
0.0005	560001	280039	190077
0.0001	2800029	1400053	950058

SPRING DASHPOT EXAMPLE Effective Number of Floating Point Operations				
h	ER	TP*	R2	R4
0.4	255	31773	605	1405
0.1	1015	14844	2429	5661
0.05	2005	20516	4805	11205
0.01	10015	62352	24029	56061
0.005	20005	109186	48005	112005
0.001	100015	432457	240029	560061
0.0005	200005	798609	480005	1120005
0.0001	1000015	3537515	2400029	56000061

*Error coefficient set at 1×10^{-8}

Table G
Spring Dashpot Example
Maximum Local Error

SPRING DASHPOT EXAMPLE Maximum Local Error			
h	P4	P42	B42
0.4	1.22e16	1.93e20	1.15e21
0.1	6.27e-1	3.37e-1	2.35e0
0.05	3.59e-2	1.89e-2	2.05e-1
0.01	7.54e-5	4.39e-5	6.67e-4
0.005	6.66e-6	4.06e-6	4.07e-5
0.001	3.63e-8	2.35e-8	5.16e-8
0.0005	4.37e-9	2.87e-9	4.66e-9
0.0001	3.18e-10	3.29e-10	1.83e-10

SPRING DASHPOT EXAMPLE Maximum Local Error			
h	P2	P22	P24
0.4	1.93e11	1.79e19	9.41e17
0.1	1.21e1	8.93e0	8.73e14
0.05	8.52e-1	5.91e-1	7.47e0
0.01	7.55e-3	7.44e-3	5.25e-2
0.005	1.58e-3	1.57e-3	7.66e-3
0.001	5.35e-5	5.35e-5	1.43e-4
0.0005	1.33e-5	1.33e-5	3.07e-5
0.0001	5.33e-7	5.33e-7	1.07e-6

SPRING DASHPOT EXAMPLE Maximum Local Error				
h	ER	TP*	R2	R4
0.4	1.98e12	7.06e3	2.04e2	1.04e2
0.1	1.39e3	4.14e0	1.21e0	8.08e-2
0.05	3.14e1	1.06e0	3.45e-1	5.16e-3
0.01	1.04e1	4.27e-2	1.14e-2	8.25e-6
0.005	4.54e0	1.07e-2	2.70e-3	5.15e-7
0.001	7.99e-1	4.28e-4	1.07e-4	8.39e-10
0.0005	3.93e-1	1.07e-4	2.67e-5	1.26e-10
0.0001	7.74e-2	4.27e-6	1.07e-6	3.50e-10

*Error coefficient set at 1×10^{-8}

Orbital Maneuvering Vehicle

The Orbital Maneuvering Vehicle (OMV) is an unmanned space vehicle that will be deployed from the U.S. Space Shuttle. It is designed to perform various operations by remote control on payloads in space. The transactional equations of motion for the homogeneous case when the vehicle is in low earth orbit are

$$\ddot{X} = -2w\dot{Z},$$

$$\ddot{Y} = -w^2Y,$$

$$\ddot{Z} = 2w\dot{X} + 3w^2Z,$$

where

$$w = 0.00118,$$

If the initial conditions are

$$X=Y=Z=0,\ \dot{X}=0.5,\ \text{and}\ \dot{Y}=\dot{Z}=0,$$

the analytical solution used to compare the accuracy is found to be

$$X=(4*\sin(w*t)-3*w*t)*.05/w, \text{ and}$$

$$Z=(2*(1-\cos(w*t))*.05/w).$$

```
/*********************************************/
/*         APPLICATION PROGRAM AREA          */
/*********************************************/


/*  Orbital Maneuvering Vehicle in Low Earth Orbit */

double w,g,me,ro,hh,tstp;

/************ HEADER INFORMATION ************/
void header()
{
printf("                ****************************************\n");
printf("                ***** ORBITAL MANEUVERING VEHICLE *****\n");
printf("                *****       IN LOW EARTH ORBIT      *****\n");
printf("                ****************************************\n\n");
}
/*********************************************/

/************** INITIAL SECTION **************/
void initial()
{
/* Define Preset variables  */
     g=6.672e-11;
     me=5.98e24;
     ro=6.37e6;
     hh=200e3;

     w=.00118;
     tstp=3600;
}
/*********************************************/

/**** INITIAL VALUES OF STATE VARIABLES ****/ void int_val(y) double
y[]; {
     y[1]=0;
     y[2]=0;
     y[3]=0;
     y[4]=.05;
     y[5]=0;
     y[6]=0;

    /* Specify Number of State Variables  */
    st_var(6);

    /* Specify Relative Execution Time of */
    /* DERIVATIVE section                 */
    num_mul(8);
    num_add(3);
}
/*********************************************/
```

```
/************ DERIVATIVE SECTION *************/
void derivative(t,y,dy)
double t,y[],dy[];
{
     dy[4] = -2*w*y[6];
     dy[1] = y[4];
     dy[5] = -w*w*y[2];
     dy[2] = y[5];
     dy[6] = w*(2*y[4]+3*w*y[3]);
     dy[3] = y[6];

     termt(t>tstp);
}
/*****************************************/



/************** DYNAMIC SECTION *************/
void dynamic(t,y)
double t,y[];
{
     double sin(),cos();
     double x,z;
     void err();
     x=(4*sin(w*t)-3*w*t)*.05/w;
     z=(2*(1-cos(w*t))*.05/w);

     if (x!=0)
          err((x-y[1])/x);

     if (z!=0)
          err((z-y[3])/z);
}
/*****************************************/



/************ OUTPUT STATEMENT(S) ************/
void output(t,y)
double t,y[];
{
     double sin(),cos();
     double x,z;
     printf("t=% 14.8f x=% 12.8f z=% 12.8f\n",t,y[1],y[3]);

     x=(4*sin(w*t)-3*w*t)*.05/w;
     z=(2*(1-cos(w*t))*.05/w);


     printf("                  x=% 12.8f",x);
     printf(" z=% 12.8f\n",z);

}
/*****************************************/
```

Table H
Orbital Maneuvering Vehicle
Maximum Local Error

ORBITAL MANEUVERING VEHICLE Maximum Local Error			
h	P4	P42	B42
300	2.00e-1	2.00e-1	1.25e-1
100	9.77e-3	9.76e-3	6.48e-3
50	1.21e-3	8.00e-4	7.82e-4
10	9.53e-6	6.35e-6	6.32e-6
5	1.19e-7	7.92e-7	7.91e-7
1	9.50e-9	6.33e-9	6.33e-9
0.5	1.19e-9	7.91e-10	7.91e-10
0.1	9.49e-12	6.33e-12	6.35e-12

ORBITAL MANEUVERING VEHICLE Maximum Local Error			
h	P2	P22	P24
300	1.19e-1	1.28e-1	4.40e-1
100	9.15e-3	9.19e-3	3.67e-2
50	2.17e-3	2.19e-3	4.45e-3
10	8.04e-5	8.04e-5	1.23e-4
5	1.98e-5	1.98e-5	3.48e-5
1	7.84e-7	7.84e-7	1.53e-6
0.5	1.96e-7	1.96e-7	3.86e-7
0.1	7.81e-9	7.81e-9	1.56e-8

ORBITAL MANEUVERING VEHICLE Maximum Local Error				
h	ER	TP*	R2	R4
300	3.39e0	1.00e0	2.00e-1	1.30e-3
100	1.14e0	1.10e-1	1.80e-2	1.27e-5
50	5.65e-1	2.72e-2	4.21e-3	7.39e-7
10	1.12e-1	1.08e-3	1.59e-4	1.11e-9
5	5.59e-2	2.69e-4	3.94e-5	6.86e-11
1	1.12e-2	1.08e-5	1.56e-6	1.03e-13
0.5	5.58e-3	2.69e-6	3.91e-7	1.07e-14
0.1	1.12e-3	1.08e-7	1.56e-8	1.64e-12

*Error coefficient set at 1×10^{-8}

Pilot Ejection Example

This example is used to simulate the ejection of a pilot from a high-performance aircraft to determine if he will strike the vertical stabilizer. It is modeled using the logical and nonlinear differential equations

$$X = V \cos W - V_A,$$

$$Y = V \sin W,$$

$$V = 0, \quad \text{when} \quad 0 \leq Y < Y_1,$$

$$V = -D/M - G*\sin W, \quad \text{when} \quad Y \geq Y_1,$$

$$W = 0, \quad \text{when} \quad 0 \leq Y < Y_1,$$

$$W = -(G*\cos W)/V, \quad \text{when} \quad Y \geq Y_1,$$

$$D = 0.5 * p * C_D * S * V^2,$$

where

$M=7.0$ slugs,

$C_D=1.0$,

$G=32.2$ ft/sec^2,

$Y_1=4.0$ ft,

$S=10.0$ ft^2,

$V_A=900.0$,

ymx=30.0,

and initial conditions are set at

$W(0)=15.0$ degrees,

$X(0)=0$,

$Y(0)=0$,

$V(0)=40.0$ ft/sec.

No analytical solution is apparent. Accuracy is determined by periodically referencing a file containing highly accurate values found through other numerical methods.

```
/*********************************************/
/*          APPLICATION PROGRAM AREA         */
/*********************************************/

/*           PILOT EJECTION EXAMPLE          */

#include <stdio.h>

double thedeg,mass,cd,g,ve,xmn,tmx,degrad,yl,s,ro,va,ymx;
double the,vx,vy,vic,thic,d,ygel;

double sin(),cos(),atan2(),sqrt();
FILE *fpl;

/************ HEADER INFORMATION ************/
void header()
{
printf("                ************************************\n");
printf("                ***** PILOT EJECTION PROBLEM *****\n");
printf("                ************************************\n\n");
}
/*********************************************/

/************** INITIAL SECTION **************/
void initial()
{
/* Define all preset variables  */
     thedeg=15.0;
     mass=7.0;
     cd=1.0;
     g=32.2;
     ve=40.0;
     xmn= -60.0;
     tmx=4.0;
     degrad=57.3;
     yl=4.0;
     s=10.0;
     ro=0.0023769;
     va=900.0;
     ymx=30.0;

     /* ejection angle in radians */
     the=thedeg/degrad;

     /* seat initial velocity */
     vx   = va-ve*sin(the);
     vy   = ve*cos(the);
     vic  = sqrt(vx*vx+vy*vy);
     thic = atan2(vy,vx);

}
/*********************************************/
```

```
/**** INITIAL VALUES OF STATE VARIABLES ****/
void int_val(y)
double y[];
{
     y[1]=0;
     y[2]=0;
     y[3]=vic;
     y[4]=thic;

     /* Specify Number of State Variables  */
     st_var(4);

    /* Specify Relative Execution Time of */
    /* DERIVATIVE section                 */
    num_mul(102);
    num_add(201);

     /* Open compare file */
     fp1=fopen("ejt.cmp","rt");
}
/*********************************************/



/************ DERIVATIVE SECTION ************/
void derivative(t,y,f)
double t, y[], f[];
{

    /* compute drag */
     d=0.5*ro*cd*s*y[3]*y[3];
     if (y[2]>=y1) yge1=1;
     else yge1=0;

     /* relative positions  */
     f[1]=y[3]*cos(y[4])-va;
     f[2]=y[3]*sin(y[4]);

    /* space velocity and flight path angle */
    f[3]=yge1*(-d/mass-g*sin(y[4]));
    f[4]=yge1*(-g*cos(y[4])/y[3]);


}
/*********************************************/
```

```
/************* DYNAMIC SECTION *************/
void dynamic(t,y)
double t,y[];
{
     /* specify termination conditions */
     termt(y[1]<=xmn||y[2]>=ymx||t>=tmx);
}
/*******************************************/



/************ OUTPUT STATEMENT(S) ************/
void output(t,y)
double t,y[];
{
     double w,x,yy,z,atof();
     char ch[80];

     w=atof(fgets(ch,80,fp1));
     if (w!=0)
          err((y[4]-w)/w);

     x=atof(fgets(ch,80,fp1));
     if (x!=0)
          err((y[3]-x)/x);

     yy=atof(fgets(ch,80,fp1));
     if (yy!=0)
          err((y[2]-yy)/yy);

     z=atof(fgets(ch,80,fp1));
     if (z!=0)
          err((y[1]-z)/z);

     printf("t=%3.8f th=%3.8f v=%3.8f\n",t,y[4],y[3]);
     printf("t=%3.8f th=%3.8f v=%3.8f\n",t,w,x);

     printf("x=%3.8f y=%3.8f d=%3.8f\n",y[1],y[2],d);
     printf("x=%3.8f y=%3.8f\n\n",z,yy);

}
/*******************************************/
```

Table I
Pilot Ejection Example
Maximum Local Error

PILOT EJECTION EXAMPLE Maximum Local Error			
h	P4	P42	B42
0.05	4.72e-1	4.72e-1	7.34e-1
0.01	3.05e-2	3.05e-2	1.16e-1
0.005	2.02e-2	2.02e-2	1.80e-2
0.001	5.22e-4	5.22e-4	1.72e-3
0.0005	4.42e-3	4.42e-3	5.80e-3
0.0001	4.54e-4	4.54e-4	7.74e-4
0.00005	4.23e-5	4.23e-5	2.05e-4

PILOT EJECTION EXAMPLE Maximum Local Error			
h	P2	P22	P24
0.05	4.72e-1	4.72e-1	1.84e0
0.01	2.37e-2	2.37e-2	4.90e-1
0.005	2.18e-2	2.18e-2	3.65e-2
0.001	5.85e-4	5.85e-4	7.75e-3
0.0005	4.40e-3	4.40e-3	8.95e-3
0.00001	4.54e-4	4.54e-4	1.43e-3
0.000005	4.24e-5	4.24e-5	5.34e-4

PILOT EJECTION EXAMPLE Maximum Local Error				
h	ER	TP*	R2	R4
0.05	7.34e-1	4.72e-1	4.72e-1	1.05e-1
0.01	2.03e-1	4.46e-2	5.32e-2	3.71e-2
0.005	7.43e-2	1.67e-2	2.86e-2	1.28e-2
0.001	1.85e-2	3.84e-4	9.34e-3	6.07e-3
0.0005	1.39e-2	4.46e-3	4.92e-4	1.12e-3
0.0001	2.37e-3	4.56e-4	5.37e-4	2.08e-4
0.00005	9.17e-4	4.19e-5	4.54e-4	2.89e-4

*Error coefficient set at 1×10^{-8}

Optimal Control of Guided Missile Example

This example simulates an optimal guidance strategy used for directing high speed objects such as missiles to their target.

Set of Differential Equations:

$$\dot{P1} = -G(P4)(P4)$$

$$\dot{P2} = P1 - G(P4)(P7)$$

$$\dot{P3} = P2 + A(P4) - G(P4)(P9)$$

$$\dot{P4} = P3 + B(P4) - G(P4)(P10)$$

$$\dot{P5} = 2(P2) - G(P7)(P7)$$

$$\dot{P6} = P3 + P5 + A(P7) - G(P7)(P9)$$

$$\dot{P7} = P6 + B(P7) + P4 - G(P7)(P10)$$

$$\dot{P8} = 2(P6) + 2(A)(P9) - G(P9)(P9)$$

$$\dot{P9} = P8 + B(P9) + P7 + A(P10) - G(P9)(P10)$$

$$\dot{P10} = 2(P9) + 2(B)(P10) - G(P10)(P10)$$

$$\dot{P11} = -P2(QT) - G(P4)(P14)$$

$$\dot{P12} = P11-P5(QT)-G(P7)(P14)$$

$$\dot{P13} = P12 + A(P14) - P6(QT) - G(P9)(P14)$$

$$\dot{P14} = P13 + B(P14) - P7(QT) - G(P10)(P14)$$

Constants:

A=-9,B=17,G=0.5,QT=3

Initial Conditions:

P1=14.5,
P2=P3=P4=P5=P6=P7=P8=P9=P10=P11=P12=P13=P14=0

No analytical solution is apparent. Accuracy is determined by periodically referencing a file containing highly accurate values found through other numerical methods.

```
/*********************************************/
/*         APPLICATION PROGRAM AREA          */
/*********************************************/

#include <stdio.h>

/* Optimal Control of Guided Missile Example */

double a,b,g,qt,ssl;
FILE *fpl;
/*********** HEADER INFORMATION ************/
void header()
{
printf("          ********************************************\n");
printf("          ***** OPTIMAL CONTROL OF GUIDED MISSILE ****\n");
printf("          ********************************************\n\n");
}
/*********************************************/

/************* INITIAL SECTION *************/
void initial()
{
/* Define Constants  */
     a= -9;
     b=17;
     g=.5;
     qt=3;
     ssl=14.5;
}
/*********************************************/



/**** INITIAL VALUES OF STATE VARIABLES ****/
void int_val(p)
double p[];
{
     p[1]=ssl;
     p[2]=0;
     p[3]=0;
     p[4]=0;
     p[5]=0;
     p[6]=0;
     p[7]=0;
     p[8]=0;
     p[9]=0;
     p[10]=0;
     p[11]=0;
     p[12]=0;
     p[13]=0;
     p[14]=0;
```

```
    /* Specify Number of State Variables  */
    st_var(14);

    /* Specify Relative Execution Time of */
    /* DERIVATIVE section                 */
    num_mul(47);
    num_add(31);

     /* Open compare file */
     fp1=fopen("mis.cmp","rt");

}
/*******************************************/


/*********** DERIVATIVE SECTION ************/
void derivative(t,p,dp)
double t, p[], dp[];
{
     dp[1]= -g*p[4]*p[4];                                  /* 3 flops */
     dp[2]=p[1]-g*p[4]*p[7];                               /* 3 flops */
     dp[3]=p[2]+a*p[4]-g*p[4]*p[9];                        /* 5 flops */
     dp[4]=p[3]+b*p[4]-g*p[4]*p[10];                       /* 5 flops */
     dp[5]=2*p[2]-g*p[7]*p[7];                             /* 4 flops */
     dp[6]=p[3]+p[5]+a*p[7]-g*p[7]*p[9];                   /* 6 flops */
     dp[7]=p[6]+b*p[7]+p[4]-g*p[7]*p[10];                  /* 6 flops */
     dp[8]=2*p[6]+2*a*p[9]-g*p[9]*p[9];                    /* 7 flops */
     dp[9]=p[8]+b*p[9]+p[7]+a*p[10]-g*p[9]*p[10];          /* 8 flops */
     dp[10]=2*p[9]+2*b*p[10]-g*p[10]*p[10];                /* 7 flops */
     dp[11]= -p[2]*qt-g*p[4]*p[14];                        /* 5 flops */
     dp[12]=p[11]-p[5]*qt-g*p[7]*p[14];                    /* 5 flops */
     dp[13]=p[12]+a*p[14]-p[6]*qt-g*p[9]*p[14];            /* 7 flops */
     dp[14]=p[13]+b*p[14]-p[7]*qt-g*p[10]*p[14];           /* 7 flops */

     termt(t>1);
}
/*******************************************/


/************* DYNAMIC SECTION *************/
void dynamic(t,p)
double t, p[];
{
}
/*******************************************/
```

```
/*********** OUTPUT STATEMENT(S) ***********/
void output(t,p)
double t,p[];
{
     double pc[14], atof();
     char ch[80];

     int i;
         fgets(ch,80,fp1);

     for (i=1;i<=14;i++) {
         pc[i]=atof(fgets(ch,80,fp1));
         if (pc[i]!=0)
              err((p[i]-pc[i])/pc[i]);
     }
     printf("t=%g\n",t);
     printf("p1=%g p2=%g p3=%g p4=%g\n",p[1],p[2],p[3],p[4]);
     printf("p1=%g p2=%g p3=%g p4=%g\n\n",pc[1],pc[2],pc[3],pc[4]);
     printf("p5=%g p6=%g p7=%g p8=%g\n",p[5],p[6],p[7],p[8]);
     printf("p5=%g p6=%g p7=%g p8=%g\n\n",pc[5],pc[6],pc[7],pc[8]);
     printf("p9=%g p10=%g p11=%g p12=%g\n",p[9],p[10],p[11],p[12]);
     printf("p9=%g p10=%g p11=%g p12=%g\n\n",pc[9],pc[10],pc[11],
          pc[12]);
     printf("p13=%g p14=%g\n",p[13],p[14]);
     printf("p13=%g p14=%g\n\n\n",pc[13],pc[14]);

}
/*****************************************/
```

Table J
Optimal Control of Guided Missile
Maximum Local Error

OPTIMAL CONTROL OF GUIDED MISSILE Maximum Local Error			
h	P4	P42	B42
0.1	9.63e7	3.09e7	2.39e7
0.05	2.02e5	2.53e10	2.25e6
0.01	9.55e2	1.05e3	4.57e3
0.005	1.21e2	8.97e1	4.37e2
0.001	1.10e0	7.28e-1	1.57e0
0.0005	1.40e-1	9.34e-2	1.49e-1
0.0001	1.15e-3	7.63e-4	8.57e-4

OPTIMAL CONTROL OF GUIDED MISSILE Maximum Local Error			
h	P2	P22	P24
0.1	3.46e7	3.28e7	8.13e4
0.05	2.64e5	4.43e6	2.80e7
0.01	1.50e3	2.15e3	6.03e4
0.005	1.20e3	9.25e2	1.10e4
0.001	8.44e1	8.39e1	2.59e2
0.0005	2.24e1	2.24e1	5.65e1
0.0001	9.41e-1	9.41e-1	1.98e0

OPTIMAL CONTROL OF GUIDED MISSILE Maximum Local Error				
h	ER	TP*	R2	R4
0.1	7.97e4	-------	5.38e6	2.26e5
0.05	1.46e6	1.26e5	5.59e5	2.42e4
0.01	2.37e5	9.10e3	1.66e4	9.83e1
0.005	8.79e4	2.35e3	4.40e4	7.04e0
0.001	1.38e4	9.51e1	1.91e2	1.26e-2
0.0005	6.70e3	2.35e1	4.84e1	7.95e-4
0.0001	1.31e3	9.44e-1	1.96e0	4.08e-6

*Error coefficient set at 1×10^{-8}