

1099

PARALLEL SOLUTION OF O D E BY RUNGE-KUTTA METHOD

*Dissertation submitted to The Jawaharlal Nehru University
in partial fulfilment of the requirements
for the award of the degree of*
MASTER OF TECHNOLOGY

IN

COMPUTER SCIENCE

BHUSHAN KUMAR

**SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI-110 067
JANUARY 1994**

TO

NEHA, YAMINI

MADHU, KAVITA,

SAVLEEN, TANPREET,

RUCHIKA & MANSI

CERTIFICATE

This is to certify that the dissertation titled " PARALLEL SOLUTION OF ODE BY RUNGE-KUTTA METHOD " being submitted by **BHUSHAN KUMAR** to Jawaharlal Nehru University, New Delhi in partial fulfillment of the requirements for the award of the degree of **Master of Technology** is a record of the original work done by him under the supervision of **Dr. C. P. Katti**, Associate Professor, School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi during the year 1993, Monsoon Semester.

The results reported in this dissertation have not been submitted in part or in full to any other university or institution for the award of any degree or diploma.



5-1-94

Prof. K.K. Bharadwaj
Dean,
School of Computer and
System Sciences,
Jawaharlal Nehru University,
New Delhi.



Dr. C.P. Katti
Associate Professor,
School of Computer and
System Sciences,
Jawaharlal Nehru University,
New Delhi.

ACKNOWLEDGEMENT

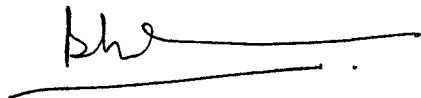
I am immensely indebted to my supervisor **Dr. C. P. Katti**, Associate Professor, School of Computer and System Sciences, Jawaharlal Nehru University, New Delhi for suggesting me this topic. I express my sincere thanks to him for his personal involvement during the period of my work and his guidance which has been indispensable in bringing about a successful completion of the dissertation.

I am also grateful to Dr. C.P. Katti for providing me with his invaluable notes and papers related to the topic.

I extend my sincere gratitude to **Prof. K.K. Bharadwaj**, Dean, School of Computer and System Sciences, Jawaharlal Nehru University for providing me with the environment and all the facilities required for the completion of this dissertation.

Special thanks are due to **Ajay Kumar**, **Diwan H. Khan**, and **Sanjeev Sinha** for helping me in the this project.

I take this opportunity to thank all faculty and staff members and my friends who helped me directly or indirectly.



(**BHUSHAN KUMAR**)

ABSTRACT

The Runge-Kutta methods are widely used for solving initial value problem. These methods provide approximations which converges to true solution as step size tends to zero, and also have the advantage of self-starting.

The drawback of Runge-kutta methods are that they involve considerably more computations per step. The next drawback is that the method is a serial method. But by modifications in its conventional form, it can be converted into a parallel method, and hence computations can be done in parallel to overcome this drawback.

However, due to modifications in original Runge-Kutta method, some of its features, like stability is affected. Hence, to improve the solution, Predictor-Corrector method, which itself will be parallel, are to be used.

CONTENTS

CHAPTER 1	INTRODUCTION TO PARALLEL PROCESSING	1
1.1	MODEL OF PARALLEL COMPUTATION	2
1.2	LEVEL OF PARALLELISM	3
1.2.1	JOB LEVEL PARALLELISM	3
1.2.2	PROGRAM LEVEL PARALLELISM	4
1.2.3	INSTRUCTION LEVEL PARALLELISM	6
1.2.4	ARITHMETIC & BIT LEVEL PARALLELISM	6
1.3	ARCHITECTURES	7
1.4	SYNCHRONOUS ARCHITECTURE	8
1.5	SYSTOLIC ARCHITECTURE	10
CHAPTER 2	ORDINARY DIFFERENTIAL EQUATION	13
2.1	INTRODUCTION	13
2.1.1	INITIAL VALUE PROBLEM	14
2.1.2	TEST EQUATION	16
2.2	NUMERICAL METHODS	18
2.2.1	EULER METHOD	20
2.2.2	BACKWARD EULER METHOD	21
2.2.3	MID POINT FORMULA	21
CHAPTER 3	RUNGE KUTTA METHOD	23
3.1	RUNGE KUTTA METHOD	23
3.1.1	SECOND ORDER RUNGE KUTTA METHOD	25
3.1.2	THIRD ORDER RUNGE KUTTA METHOD	27
3.2	IMPLICIT RUNGE KUTTA METHOD	29

CHAPTER 4	PARALLEL RUNGE KUTTA METHOD	31
4.1	PARALLEL RUNGE KUTTA METHOD	31
4.2	THIRD ORDER PARALLEL RUNGE KUTTA METHOD	33
4.3	STABILITY ANALYSIS	34
4.3.1	STABILITY OF 2nd ORDER RUNGE KUTTA METHOD	37
4.3.2	STABILITY OF 3rd ORDER PARALLEL RUNGE KUTTA METHOD	38
4.4	APPLICATIONS	39
4.4.1	HIGHER ORDER DIFFERENTIAL EQUATIONS	39
4.4.2	SYSTEM OF DIFFERENTIAL EQUATIONS	41
CHAPTER 5	PREDICTOR CORRECTOR METHOD	43
5.1	PREDICTOR CORRECTOR METHOD	43
5.2	PARALLEL CORRECTOR METHOD	44
5.3	HYBRID METHODS	46
5.3.1	PARALLEL VERSION ONE	47
5.3.2	PARALLEL VERSION TWO	47
5.4	ALGORITHM	49
CONCLUSION		52
REFERENCES		

CHAPTER ONE

INTRODUCTION TO PARALLEL PROCESSING

" Mine is a long and sad
tale " said the Mouse,
turning to Alice and
sighing. " It is a long
tail, certainly " said
Alice, looking down with
wonder at the Mouse's
tail, " but why do you
call it sad? "

LEWIS CARROLL.

INTRODUCTION TO PARALLEL PROCESSING

In recent years we have witnessed a tremendous surge in the availability of very fast and inexpensive hardware. This has been made possible partly by the use of faster circuit technologies and smaller feature sizes; partly by novel architectural features such as pipelining, vector processing, cache memories, and systolic arrays; and partly by using novel interconnections between processors and memories such as Hypercube, Omega network, Orthogonal Tree network, and others.

Our ability to design fast and cheap hardware, however, far outstrips our ability to utilize that hardware effectively to solve large problems quickly. This is mainly because a large problem may not be easy to decompose into smaller problems that could be solved in parallel, on account of data dependencies between the subproblems. The intrinsic parallelism of a problem can be defined as the product of the time required for solving it by the fastest parallel algorithm, and the number of processors required by that algorithm, divided by the time required by the best sequential algorithm. For a problem with high internal data dependency, the intrinsic parallelism would be very low. Because of these data dependencies, process solving related subproblems would need to communicate with each other. This concurrent access, via a data bus, or via an interconnection network. Any specific scheme incurs an overhead in terms of

time lost to contention, latency, or both, and additional overhead in hardware costs, which do not reduce dramatically with improved technology as does that of the underlying circuits. Further, a communication scheme that is good for one class of problems may not be good for another class, which only adds to our difficulty in finding a parallel algorithm.

Memory contention can slow down the execution of a parallel algorithm if the various processing elements must access the same variable at the same time, some systems must be devised so that only the processor can access a given variable at any one time (for example, LOCK and UNLOCK on MIMD machines). Also, if the number of logical processors is larger than the number of physical processors in the machine, some sort of scheduling must be done to determine where the extra processes will eventually be handled. The scheduling cost is the resource allocated to do this scheduling. For efficient scheduling, the extra logical processes should be saved until a processor is available, and the internal state of the logical processes should be monitored.

1.1 MODELS OF PARALLEL COMPUTATION

The design of parallel algorithms becomes an important issue as numerous parallel architectures are developed. In fact, a considerable number of very different architectures for parallel computing are in existence. They range from special purpose array processors to tree machines to loosely

coupled networks of processors. Since the design and performance of a parallel algorithm depends very much on the architecture of the parallel machine, it is necessary to keep the architecture in mind when designing parallel algorithms. There is, however, no universal method for designing parallel algorithms. One approach to constructing parallel algorithms is to recognize parallelism in the existing sequential algorithms. This approach has been studied by several researchers (Keller 1973; Lee et al. 1985; Moitra and Iyengar 1986, Nicolau 1985; Shrira et al. 1983; Strom and Yemini 1985).

1.2 LEVELS OF PARALLELISM

Parallelism is not a new concept and has been used to improve the effectiveness of computers since the earliest designs. It can be applied at various levels which can be classified as (a) Job level, (b) Program level, (c) Instruction level and (d) Arithmetic and bit level.

1.2.1 Job Level Parallelism

Job level parallelism is implemented in most computer installations. Viewed in a simplistic manner, every job is divided into several sequential phase each of which requires a different systems program and system resources. For example, an I/O operation is very slow compared to the actual program execution and therefore, any reasonably large computer installation provides several I/O channels or

peripheral processors which can perform the I/O in parallel with program execution.

Usually, several programs reside in the fast memory of a computer and only one of these will be in execution at any given time. As soon as this program, requires a slow I/O facility, e.g. a read from a tape or disk, this I/O operation is initiated in the channel and another program is put into execution. The first program waits until the data is available and its execution restarts only when the other program are similarly obliged to wait.

This sort of (job level) parallelism is not really useful in the context of computationally intensive applications such as CFD. Since CFD programs involve very little I/O time (as compared to the execution time), the system throughput is not likely to increase significantly. In fact, considering the size of these programs, the throughput may actually decrease because of the overheads involved in swapping the massive amounts of data and program from the memory.

1.2.2 Program Level Parallelism

Program level parallelism is the one most important in the context of supercomputers. Within a large program, there could be sections of code which are quite independent and could therefore be executed in parallel on different processors in a multiprocessor environment. This is the central idea in parallel processing. There are several

methods of implementing this architecturally and we will discuss these presently.

Program level parallelism could arise in several ways the most commonly encountered one being a DO loop which can be replaced by one or more vector instructions. This has been exploited in the well-known vector machines such the CRAY, CYBER, etc. The performance of these machine hinges on the ability of the compiler to recognize and take advantage of such vectorization. It also means that the problem must in the first instance permit vectorization.

In certain programs, the different execution of a loop may be completely independent of each other. This arises, for example, in Monte Carlo analysis where the same calculations are repeated many times with different data chosen in a random fashion. In such cases, the full code can be loaded onto each processor in a multi-processor environment and the calculations for each sample done in parallel.

However, things are not usually so simple. While in some instances certain sections of code can be recognized to be independent by logical analysis of the source code or the problem itself, in other cases there may be data dependency and the independence of sections of code may not be known until the program is executed. The onus of recognizing the parallelism then falls on the compiler though the programmer can help to a certain extent by writing his programs in a way that aids the compiler. The hardware and architecture in

general can only provide the necessary computing power while the systems software must actually exploit this capability. There is a considerable amount of work being done on such compilers but the present compilers are far from optimum.

1.2.3 Instruction Level Parallelism

Instruction level parallelism is important in enabling the unit processors in a multiprocessor environment to work faster. The instructions in a unit processor may be divided into several sub-operations, which may then be pipelined to speed up execution. Another approach which is becoming popular in the newer microprocessors is instruction prefetching in an instruction cache. Here, while a particular instruction is being executed and the processor bus is free, new instructions are fetched and loaded into an instruction cache. This overlap of instruction fetch cycles with the execution cycles enhances the processor speed.

1.2.4 Arithmetic and Bit Level Parallelism

Arithmetic and Bit level parallelism is the lowest level of parallelism in computers. This is a self-evident concept. Obviously, using an 8 bit machine to do 64 bit arithmetic is going to be much slower than using a 64 bit machine in the single precision mode. This level of parallelism is directed by considerations such as the requirements of the typical problem to be solved, available technology, the amount of hardware the system designer wants to use etc.

1.3 ARCHITECTURES

Having seen that program level parallelism offers a major increase in computation speed, we will now examine the various architectural concepts which allow for such parallelism.

One can classify computers into four broad categories according to whether the instruction or the data streams are single or multiple. (Fig. 1) (A stream is defined as a sequence of items - instructions or data - as executed or operated on by a processor):

Flynn's taxonomy. Flynn's taxonomy classifies architectures on the presence of single or multiple streams of instructions and data. This yields the four categories below:

SISD (Single instruction, single data stream) - defines serial computers. MISD (Multiple instruction, single data stream) - would involve multiple processors applying different instructions to a single datum; this hypothetical possibility is generally deemed impractical.

SIMD (single instruction, multiple data streams) - involves multiple processors simultaneously executing the same instruction on different data this definition is discussed further prior to examining array processors below).

MIMD (multiple instruction, multiple data streams) - involves multiple processors autonomously executing diverse instructions on diverse data.

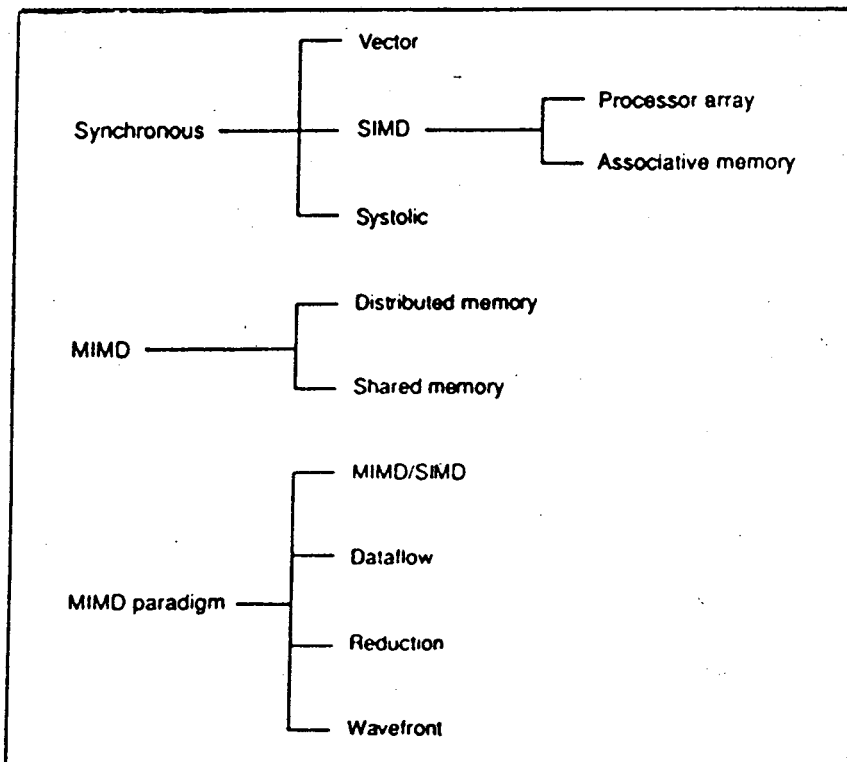


Figure 1. High-level taxonomy of parallel computer architectures.

1.4 SYNCHRONOUS ARCHITECTURES

Pipelined vector processors: Vector processors are characterized by multiple. Pipelined functional units, which implement arithmetic and Boolean operations for both vectors and scalars and which can operate concurrently. Such architectures provide parallel vector processing by sequentially streaming vector elements through a functional unit pipeline and by streaming the output results of one unit into the pipeline of another as input (a process known as "chaining").

A representative architecture might have a vector addition unit consisting of six pipeline stages (Fig. 2).

Recent vector processing supercomputer erst such as the Cray X-MP/4 and ET-A-10) unit four to 10 vector processors through a large shared memory.

SIMD architectures. SIMD architecture (Fig. 3) typically employ a central control unit, multiple processors, and an interconnection network (IN) for either processor-to-processor or processor-to-memory communications. The control unit broadcasts a single instruction to all processors, which execute the instruction in lockstep fashion on local data. The interconnection network allows instruction result calculated at one processor to be communicated to another processor for use as operands in a subsequent instruction. Individual processors may be allowed to disable the current instruction.

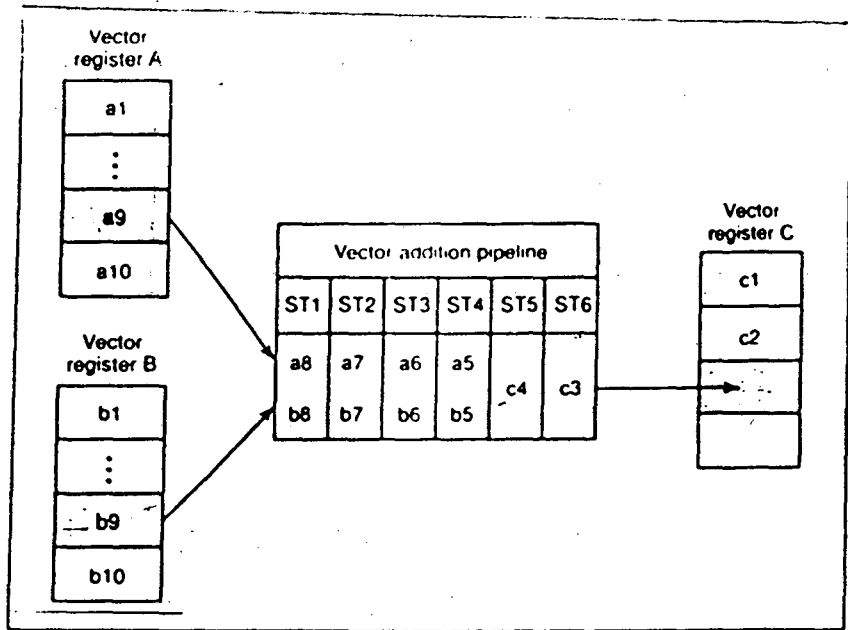


Figure 2. Register-to-register vector architecture operation.

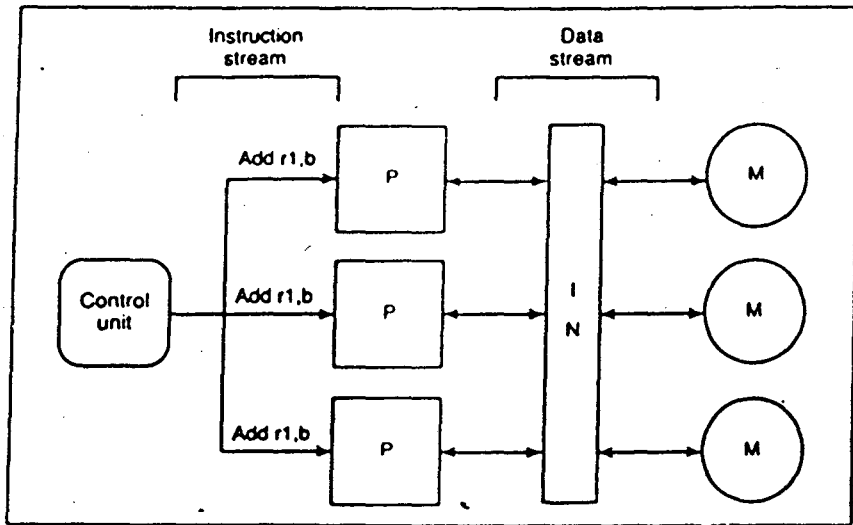


Figure 3. SIMD execution.

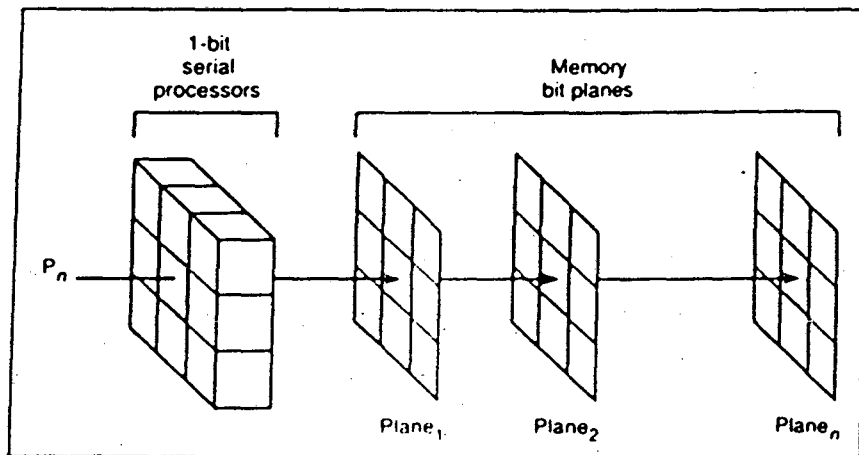


Figure 4. Bit-plane array processing.

Processor array architectures. Processors structured for numerical SIMD execution have often been employed for large-scale scientific calculations. Such as image processing and nuclear energy providing.

Operands are usually floating-point values and typically range in size from 32 to 64 bits. Various IN schemes have been used to provide processor-to-processor or processor - to - memory communications, with mesh and crossbar approaches being among the most popular.

One variant of processor array architectures involves using a large number of one-bit processors. In bit-plane architectures, the array of processors is arranged in a symmetrical grid (such as 64x64) and associated with multiple "planes" of memory bits that correspond to the dimensions of the processor grid (Fig. 4). Processor $n(P_n)$, situated in the grid at location (x, y) , operates on the memory bits at location (x, y) in all the associated memory planes. Usually, operations are provided to copy, mask, and perform arithmetic operations on entire memory planes, as well as on columns and rows within a plane.

Associative memory processor architectures. Computers built around an associative memory constitute a distinctive type of SIMD architecture that uses special comparison logic to access stored data in parallel according to its contents. Research in constructing associative memories began in the late 1950s with the obvious goal of being able to search memory in parallel for data that matched some specified

datum. "Modern" associative memory processors developed in the early 1970s (for example, Bell Laboratories' Parallel Element Processing Ensemble, or PEPE) and recent architectures have naturally been geared to database-oriented applications, such as tracking and surveillance.

Figure 5 shows the characteristic functional units of an associative memory processor. A program controller reads and executes instructions, invoking a specialized array controller when associative memory instructions are encountered. Special registers enable the program controller and associative memory to share data.

Figure 6 depicts a row-oriented comparison operation for a generic bit-serial architecture.

In figure 7 a logical OR operation is performed on a bit-column and the bit-vector in register. A with register B receiving the result. A zero in the mask register indicates that the associated word is not to be included in the current operation.

1.5 SYSTOLIC ARCHITECTURE

In the early 1980s H.T. Kung of Carnegie Mellon University proposed systolic architectures to solve the problems of special-purpose systems that must often balance intensive computations with demanding I/O bandwidths. "Systolic architectures are pipelined multiprocessors in which data is pulsed in rhythmic fashion from memory and through a network of processors before returning to memory. A global clock and explicit timing delays synchronize this

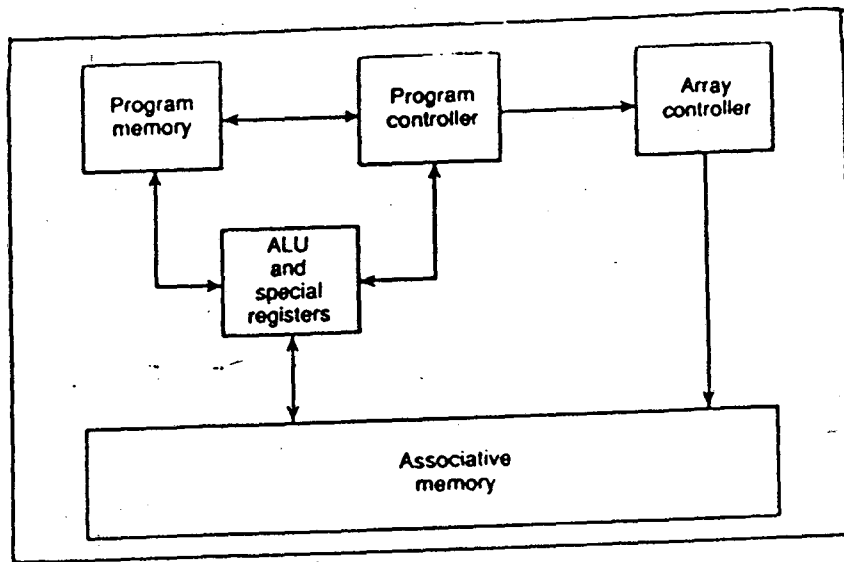


Figure 5. Associative memory processing organization.

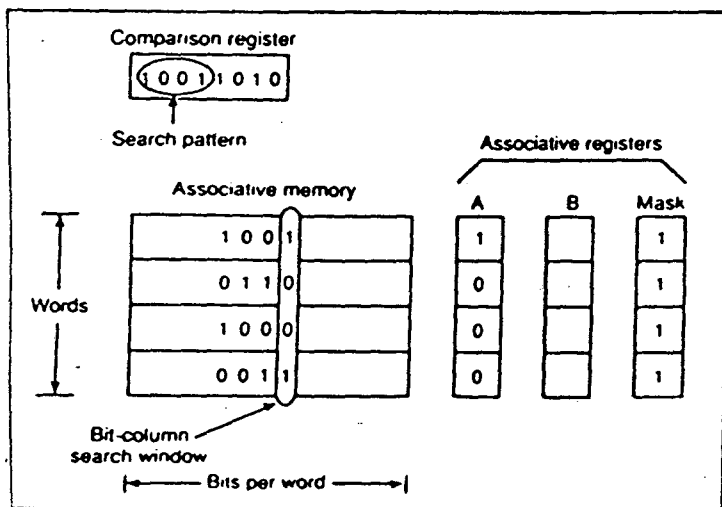


Figure 6. Associative memory comparison operation.

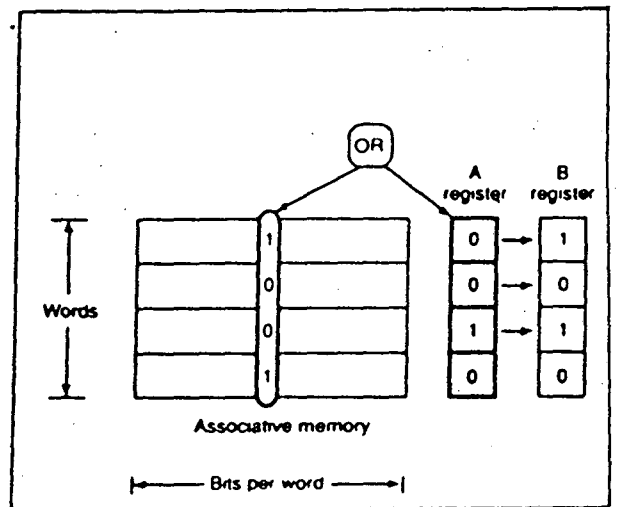


Figure 7. Associative memory logical OR operation.

pipelined data flow, which consists of operands obtained from memory and partial results to local interconnections provide basic building blocks for a variety of special purpose systems. During each time interval, these processors execute a short, invariant sequence of instructions.

A high degree of parallelism is obtained by pipelining data through multiple processors, typically in two-dimensional fashion. Systolic architectures maximize computations performed on a datum once it has been obtained from memory or an external device. Hence, once a datum enters the systolic array, it is passed to any processor that needs it, without an intervening store to memory. Only processors at the topological boundaries of the array perform I/O to and from memory.

Figure 9 a-e shows how a simple systolic array could calculate the outer product of two matrices.

$$A = \begin{vmatrix} a & b \\ c & d \end{vmatrix} \quad \text{and} \quad B = \begin{vmatrix} e & f \\ g & h \end{vmatrix}$$

The zero inputs shown moving through the array are used for synchronization. Each processor begins with an accumulator set to zero and, during each cycle, adds the product of its two inputs to the accumulator. After five cycles the matrix product is complete.

A growing number of special-purpose systems are systolic organization for algorithm-specific architectures, particularly for signal processing. In addition, programmable (reconfigurable) systolic architectures (such as

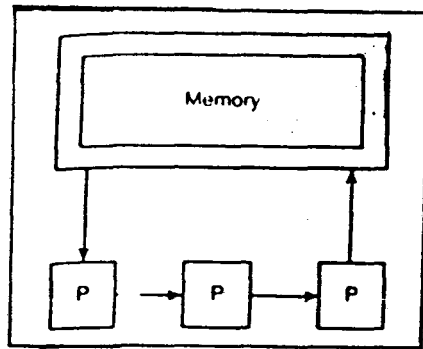


Figure 8. Systolic flow of data from and to memory.

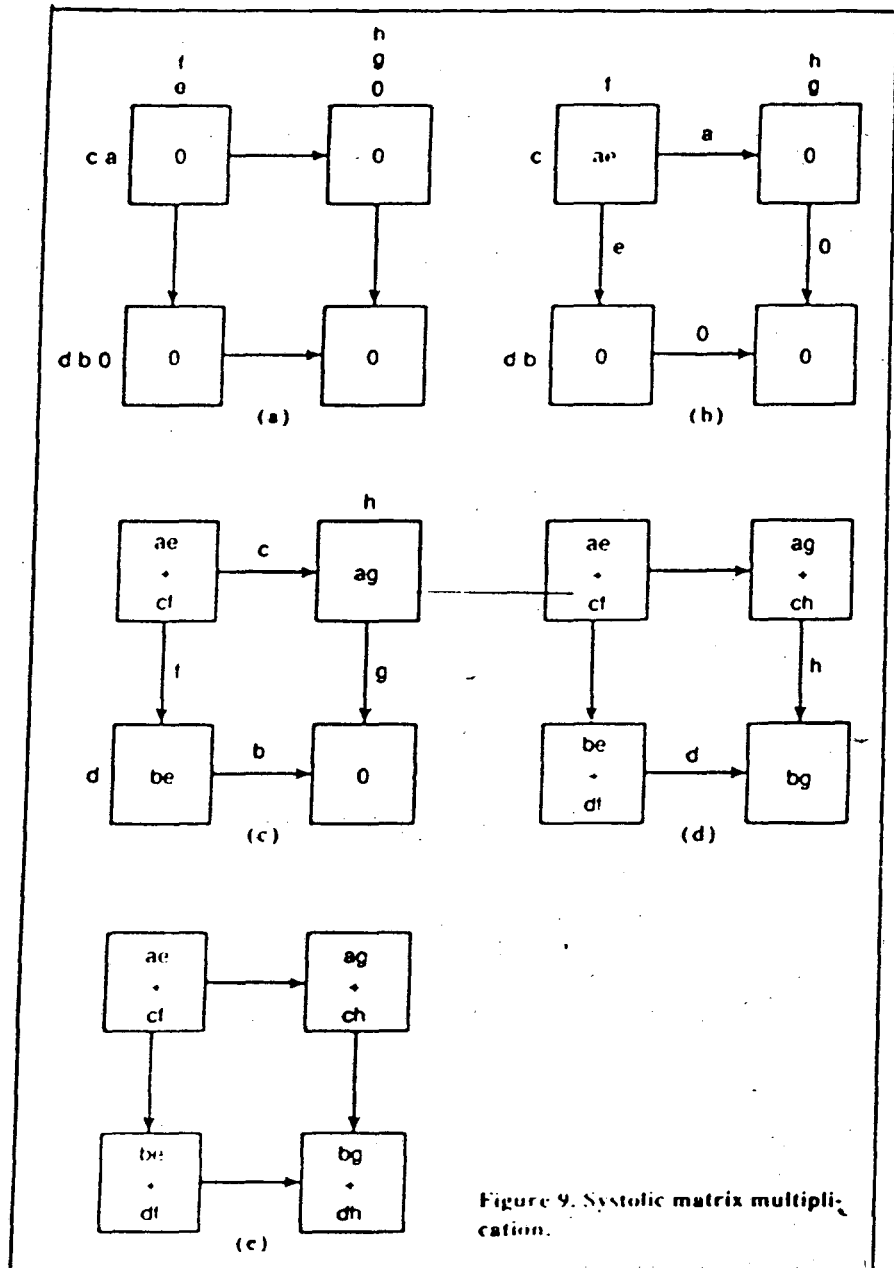


Figure 9. Systolic matrix multiplication.

Carnegie Mellon's Warp and Saxpy's Matrix-1) have been constructed that are not limited to implementing a single algorithm. Although systolic concepts were originally proposed for VLSI-based systems to be implemented at the chip level, systolic architectures have been implemented at a variety of physical levels.

At present, the world of parallel programming is quite primitive. Programmers are made painfully aware of the physical computing environment they are working in and are forced to tailor their code to match the architecture. Software development tools such as debuggers and performance monitors, commonplace in the sequential programming world, are either absent or weak in parallel programming systems. Most important, our understanding of efficient structures and algorithms for parallel systems is still quite shallow. But, we are progressing. Concepts such as distributed shared memory promise to free the programmer from having to structure his or her code to match the underlying architecture. The programmer will be able to develop algorithms using whatever paradigm is natural for the problem being solved. And when the programmer can focus more on the problem than on the computing system, parallel algorithm development and implementation will soar to new heights.

CHAPTER TWO
ORDINARY DIFFERENTIAL EQUATION

*Numerical analysis is a
science-computation is an
art.*

ORDINARY DIFFERENTIAL EQUATIONS

2.1 INTRODUCTION

An ordinary differential equation is a relation between a function, its derivatives, and the variable upon which they depend. The most general form of an ordinary differential equation is given by

$$\phi(t, y, y', y'', \dots, y^{(m)}) = 0 \quad (2.1)$$

where m represents the highest order derivative, and y and its derivatives are functions of t . The order of the differential equation is the order of its highest derivative and its degree is the degree of the derivative of the highest order after the equation has been rationalized. If no product of the dependent variable $y(t)$ with itself or any one of its derivatives occur, then the equation is said to be linear, otherwise it is nonlinear. A linear differential equation of order m can be expressed in the form

$$\sum_{p=0}^m \phi_p(t) y^{(p)}(t) = r(t) \quad (2.2)$$

in which $\phi_p(t)$ are known functions. If the general nonlinear differential equation (2.1) of order m can be written as

$$y^{(m)} = F(t, y, y', \dots, y^{(m-1)}) \quad (2.3)$$

then the equation (2.3) is called a canonical representation of the differential equation (2.1). In such a form, the highest order derivative is expressed in terms of the lower order derivatives and the independent variable.

2.1.1 Initial Value Problem

A general solution of an ordinary differential equation such as (2.1) is a relation between y , t and m arbitrary constants which satisfies the equation, but which contains no derivatives. The solution may be an implicit relation of the form

$$w(t, y, c_1, c_2, \dots, c_m) = 0 \quad (2.4)$$

or an explicit function of t of the form

$$y = w(t, c_1, c_2, \dots, c_m) \quad (2.5)$$

The m arbitrary constants c_1, c_2, \dots, c_m can be determined by prescribing m conditions of the form

$$y^{(v)}(t_0) = n_v, \quad v=0, 1, 2, \dots, m-1 \quad (2.6)$$

at one point $t = t_0$, which are called **initial conditions**. The differential equation (2.1) together with the initial conditions (2.6.) is called **m th order initial value problem**.

The m th order differential equation (2.3) with initial conditions (2.6) may be written as an equivalent system of m first order initial value problems:

$$u_1 = y$$

$$u'_1 = u_2$$

$$u'_2 = u_3$$

⋮

$$u'_{m-1} = u_m$$

$$u'_m = F(t, u_1, u_2, \dots, u_m)$$

$$u_1(t_0) = n_0, \quad u_2(t_0) = n_1, \dots, u_m(t_0) = n_{m-1}$$

which in vector notation becomes

$$\begin{aligned}u' &= f(t, u) \\ u(t_0) &= n\end{aligned}\tag{2.7}$$

where

$$\begin{aligned}u &= [u_1 \ u_2 \ \dots \ u_m]^T \\ f &= [u_2 \ u_3 \ \dots \ u_m \ F]^T \\ n &= [n_0 \ n_1 \ \dots \ n_{m-1}]^T\end{aligned}$$

Thus the methods of solution of the first-order initial value problem

$$du/dt = f(t, u), \quad u(t_0) = n\tag{2.8}$$

may be used to solve the system of first order initial value problems (2.7) and the m th order initial value problem (2.3).

The existence and uniqueness of the solution of the initial value problem (2.8) is guaranteed by the theorem:

Theorem: We assume that $f(t, u)$ satisfies the following conditions:

- (i) $f(t, u)$ is a real function
- (ii) $f(t, u)$ is defined and continuous in the strip
 $t \in [t_0, b], \quad u \in (-\infty, \infty)$

(iii) there exists a constant L such that for any $t \in [t_0, b]$ and for any u_1 and u_2

$$| f(t, u_1) - f(t, u_2) | \leq L | u_1 - u_2 |$$

where L is called the **Lipschitz constant**. Then for any u_0 the initial value problem (2.8) has a unique solution $u(t)$ for $t \in [t_0, b]$.

2.1.2 Test Equations

The behaviour of the solution of the initial value problem (2.8) in the neighbourhood of any point (t,u) can be predicted by considering the linearized form of the differential equation.

$$u' = f(t,u)$$

The nonlinear function $f(t,u)$ can be linearized by expansion of the function about the point (t,u) in the Taylor series truncated after first order term. The resulting linearized form for (2.8) is given by

$$u' = Mu + C \quad (2.9)$$

$$M = (\partial f / \partial u)_{\bar{t}}$$

$$C = f(\bar{t}, \bar{u}) - u(\partial f / \partial u)_{\bar{t}} + (\partial f / \partial t)_{\bar{t}}(t - \bar{t})$$

Further, the equation (2.9) may be written as

$$w' = Mw \quad (2.10)$$

where

$$w = u + C/M$$

Similarly, the test equation for the second order initial value problem

$$y'' = f(t, y, y') \quad (2.11)$$

$$y(t_0) = n_0, \quad y'(t_0) = n_1 \quad (2.12)$$

may also be obtained in the form

$$w'' = -bw' - cw$$

where

$$b = -\partial f / \partial y', \quad \text{and} \quad c = -\partial f / \partial y$$

The differential equation (2.12) is equivalent to the following system of equations.

$$u' = Au \quad (2.13)$$

where

$$u = [u_1 \ u_2]^T \quad A = \begin{bmatrix} 0 & 1 \\ -c & -b \end{bmatrix}$$

$$u_1 = w, \quad u_2 = w'$$

The nature of the solution of (2.12) or (2.13) depends on the roots α_1 and α_2 of the characteristic equation of the matrix A,

$$\alpha^2 + b\alpha + c = 0 \quad (2.14)$$

We now consider the following cases:

(i) $b > 0, c \geq 0, b > 2\sqrt{c}$. The solutions are exponentially decreasing. For $c = 0$, the test equation (2.12) becomes

$$w'' + bw' = 0, \quad b > 0 \quad (2.15)$$

(ii) $b < 0, c \geq 0$ and $|b| > 2\sqrt{c}$. The solution are exponentially increasing. For $c = 0$, the test equation (2.12) becomes

$$w'' + bw' = 0, \quad b < 0 \quad (2.16)$$

(iii) $c > 0$ and $|b| < 2\sqrt{c}$. The solution are oscillating. If $b < 0$ then the solution is an oscillating function whose amplitude becomes unbounded as $t \rightarrow \infty$. If $b > 0$ then the solution is a damped oscillating function as $t \rightarrow \infty$. For $b = 0$, the test equation (2.12) becomes

$$w'' + cw = 0, \quad c > 0 \quad (2.17)$$

whose solution is periodic with period $2\pi/\sqrt{c}$. We observe that α_1 and α_2 ($b=0, c>0$) are pure imaginary numbers. The solution of the test differential equation (2.10) will also be periodic if we allow M to be pure imaginary number.

Thus the nature of the solutions of the systems of equations or higher order equations may be discussed by

using the test equation (2.10) for (i) M pure real, (ii) M pure imaginary, and (iii) M complex.

2.2 NUMERICAL METHODS

The first step in obtaining a numerical solution of the differential equation (2.8) is to partition the interval $[t_0, b]$ on which the solution is desired into a finite number of subintervals by the points

$$t_0 < t_1 < t_2 \dots < t_N = b$$

The points are called the **mesh point** or the **grid points**. The spacing between the points is given by

$$h_j = t_j - t_{j-1}, \quad j = 1, 2, \dots, N$$

which is called the **mesh spacing** or **step length**. For simplicity we assume that the points are spaced uniformly, i.e.

$$h_j = h = \text{constant}, \quad j = 1, 2, \dots, N \quad (2.18)$$

The mesh points are given by

$$t_j = t_0 + jh, \quad j = 0, 1, 2, \dots, N \quad (2.19)$$

In numerical methods we determine a number u_j , which is an approximation to the value of the solution $u(t)$ at the point t_j . The set of numbers $\{u_j\}$, i.e., u_0, u_1, \dots, u_N is the **numerical solution** of the initial value problem. The numbers $\{u_j\}$ are determined from a set of algebraic equations called the **difference equations**. There are many difference approximations possible for a given differential equation. As an example, consider expressions for the first derivative in terms of the forward, backward, and central

difference operators. We assume that the function $u(t)$ may be expanded in a Taylor series in the closed interval $t-h \leq t \leq t+h$. We write as

$$u(t \pm h) = u(t) \pm h u'(t) + \frac{h^2 u''(t)}{2!} \pm \dots \\ \dots + (-1)^p \frac{h^p u^{(p)}(t)}{p!} + \dots \quad (2.20)$$

where a prime denotes differentiation with respect to t . We then have

$$\frac{u(t+h) - u(t)}{h} = u'(t) + \frac{h}{2} u''(t) + O(h^2) \quad (2.21)$$

where the notation $O(h)$ means that the first term neglected is of order h .

Similarly, we obtain

$$\frac{\nabla u(t)}{h} = \frac{du}{dt} + O(h) \quad (2.22)$$

and

$$\frac{\mu \delta u(t)}{h} = \frac{du}{dt} + O(h^2) \quad (2.23)$$

A difference approximation to $u'(t)$ at $t=t_j$ is obtained by neglecting the error term. We have

$$u'(t_j) = \begin{cases} \text{(i)} & (u_{j+1} - u_j)/h \\ \text{(ii)} & (u_j - u_{j-1})/h \\ \text{(iii)} & (u_{j+1} - u_{j-1})/2h \end{cases} \quad (2.24)$$

We use the approximations (2.24) for $u'(t)$ in the differential equation (2.8) at the mesh point t_j . This gives

$$\begin{aligned} \text{(i)} & \quad [u_{j+1} - u_j]/h = f(t_j, u_j) \\ \text{(ii)} & \quad [u_j - u_{j-1}]/h = f(t_j, u_j) \\ \text{(iii)} & \quad [u_{j+1} - u_{j-1}]/2h = f(t_j, u_j) \end{aligned} \quad (2.25)$$

The equations (2.25) may be considered as a relation between differences of an unknown function u_j and may be called **difference equations**. The order of a difference equation is the number of intervals separating the largest and the smallest arguments of the dependent variable. Thus the

difference equations (2.25i) and (2.25ii) are of first order and the difference equation (2.25iii) is of second order. The methods (2.25i) and (2.25ii) are called a **single step methods** and the method (2.25iii) is called a **two-step or multistep method**. The approximate values u_j will contain errors. We must be concerned with the effect of these errors on the solution, and ask what happens as we try to get a more accurate solution, by taking more grid points. A method is **convergent** if, as more grid points are taken or step size is decreased, the numerical solution converges to the exact solution, in the absence of roundoff errors. A method is **stable** if the effect of any single fixed roundoff error is bounded, independent of the number of mesh points.

We now examine some methods in turn, which are used to solve the ordinary differential equations.

2.2.1 Euler Method

We write (2.25i) as

$$u_{j+1} = u_j + hf_j \quad (2.26)$$

This is called the **Euler** or the first order **Adams-Bashforth method**. The method is applied at the mesh points t_j , $j=0(1)N-1$ to get the numerical solution of the differential equation (2.8). We have

$$u_1 = u_0 + hf_0$$

$$u_2 = u_1 + hf_1$$

.

.

.

.

.

$$u_N = u_{N-1} + hf_{N-1}$$

Where $f_j = f(t_j, u_j)$

Choosing h , the value u_1 is determined from the initial condition and the differential equation (2.8), and it is easy to calculate u_2 from u_1 and so on. The method (2.26) is an **explicit method**, since, using u_j , h and f_j we can calculate u_{j+1} from (2.26) directly.

2.2.2 Backward Euler Method

The equation (6.25ii) at the mesh point $t = t_{j+1}$ may be written as

$$u_{j+1} = u_j + hf_{j+1} \quad (2.27)$$

where

$$f_{j+1} = f(t_{j+1}, u_{j+1})$$

This is called the **backward Euler** or the **first order Adams-Moulton method**. The solution values u_1, u_2, \dots, u_N are determined from the following equations:

$$\begin{aligned} u_1 &= u_0 + hf_1 \\ u_2 &= u_1 + hf_2 \\ &\vdots \\ &\vdots \\ &\vdots \\ u_N &= u_{N-1} + hf_N \end{aligned} \quad (2.28)$$



2.2.3 Mid-Point Method

The equation (2.25iii) may be written as

$$u_{j+1} = u_{j-1} + 2hf_j$$

This is called **mid-point** or the **second order Nystrom Method**.

TH-5956

The solution values are given by

$$u_2 = u_0 + 2hf_1$$

$$u_3 = u_1 + 2hf_2$$

⋮

$$u_N = u_{N-2} + 2hf_{N-1}$$

The value u_0 is known from the initial condition.

In this chapter we have given a brief discussion of some well known methods for the numerical solution of an ordinary differential equation satisfying certain given initial conditions. If the solution is required over a wide range, it is important to get the starting values as accurately as possible.

CHAPTER THREE

RUNGE-KUTTA METHOD

When it known that x is same as 6 (which by the way is understood from the pronunciation) all algebraic equations with 1 or 19 unknowns are easily solved by inserting x , substituting 6, elimination of 6 by x , and so on.

FALSTAFF, FAKIR.

RUNGE-KUTTA METHOD

3.1 RUNGE-KUTTA METHOD

We first explain the principle involved in the Runge-Kutta methods. By the Mean Value Theorem any solution of

$$u' = f(t, u), u(t_0) = u_0, t \in [t_0, b]$$

satisfies

$$\begin{aligned} u(t_{j+1}) &= u(t_j) + hu'(t_j + \theta h) \\ &= u(t_j) + hf(t_j + \theta h, u(t_j + \theta h)), \quad 0 < \theta < 1 \end{aligned}$$

for $\theta = 1/2$, we have

$$u(t_{j+1}) = u(t_j) + hf(t_j + h/2, u(t_j + h/2))$$

Euler's method with spacing $h/2$ gives

$$u(t_j + h/2) \equiv u_j + h f_j/2$$

Thus, we have the approximation

$$u_{j+1} = u_j + hf(t_j + h/2, u_j + h f_j/2)$$

which may be written as

$$k_1 = hf_j$$

$$k_2 = hf(t_j + h/2, u_j + K_1/2)$$

$$u_{j+1} = u_j + k_2 \tag{3.1}$$

Alternatively, again using Euler's method, we proceed as follows:

$$\begin{aligned} u'(t_j + h/2) &\equiv (u'(t_j) + u'(t_j + h)) / 2 \\ &\equiv [f(t_j, u_j) + f(t_j + h, u_j + hf_j)] / 2 \end{aligned}$$

and thus we have the approximation

$$u_{j+1} = u_j + [f(t_j, u_j) + f(t_j + h, u_j + hf_j)]h/2 \tag{3.2}$$

which may be written as

$$K_1 = hf(t_j, u_j)$$

$$K_2 = hf(t_j + h, u_j + k_1)$$

$$u_{j+1} = u_j + (k_1 + k_2)/2$$

This method is also called **Euler - Cauchy method**.

Either (3.1) or (3.2) can be regarded as

$$u_{j+1} = u_j + h(\text{average slope})$$

This is the underlying idea of the Runge-Kutta approach. In general, we find the slope at t_j and at several other points, average these slopes, multiply by h and add the result to u_j . Thus the Runge-Kutta method with v slopes can be written as

$$K_1 = hf(t_j, u_j)$$

$$K_2 = hf(t_j + c_2h, u_j + a_{21} K_1)$$

$$K_3 = hf(t_j + c_3h, u_j + a_{31} K_1 + a_{32} K_2)$$

$$K_4 = hf(t_j + c_4h, u_j + a_{41} K_1 + a_{42} K_2 + a_{43} K_3)$$

⋮
⋮
⋮

$$K_v = hf(t_j + c_vh, u_j + \sum_{i=1}^{v-1} a_{vi} k_i)$$

and

$$u_{j+1} = u_j + W_1 K_1 + W_2 K_2 + \dots + W_v K_v \quad (3.3)$$

From (3.3) we may interpret the increment function as the linear combination of the slopes at t_j and at several other points between t_j and t_{j+1} . Further, knowing the values of the quantities on the right hand side of (3.3) the solution value u_{j+1} may be obtained directly. Thus, (3.3) represents the **explicit Runge-Kutta Method** with v slopes. To

determine the parameters c's, a's and W's in (3.3), we expand u_{j+1} in powers of h such that it agrees with the Taylor series expansion of the solution of the differential equation upto a certain number of terms.

3.1.1 Second Order Runge-Kutta Method

Consider the following Runge-Kutta method with two slopes.

$$\begin{aligned} K_1 &= hf(t_j, u_j) \\ K_2 &= hf(t_j + c_2h, u_j + a_{21} K_1) \\ u_{j+1} &= u_j + W_1 K_1 + W_2 K_2 \end{aligned} \quad (3.4)$$

Where the parameters c_2, a_{21}, W_1 and W_2 are chosen to make u_{j+1} closer to $u(t_{j+1})$. Now Taylor's series gives

$$\begin{aligned} u(t_{j+1}) &= u(t_j) + hu'(t_j) + h^2u''(t_j)/2! + h^3u'''(t_j)/3! + \dots \\ &= u(t_j) + hf(t_j, u(t_j)) + h^2(f_{tt} + ff_u)/2! \\ &\quad + h^3[f_{ttt} + 2ff_{tu} + f^2f_{uu} + f_u(f_{tt} + ff_u)] + \dots \end{aligned} \quad (3.5)$$

we also have

$$\begin{aligned} K_1 &= hf_j \\ K_2 &= hf(t_j + c_2h, u_j + a_{21}hf_j) \\ &= h[f_j + h(c_2f_{tt} + a_{21}ff_u) \\ &\quad + h^2(c^2_2f_{ttt} + 2c_2a_{21}ff_{tu} + a^2_{21}f^2f_{uu})/2! + \dots \end{aligned}$$

Substituting the values of K_1 and K_2 in (3.4) we get

$$\begin{aligned} u_{j+1} &= u_j + (W_1 + W_2)hf_j + h^2(W_2c_2f_{tt} + W_2a_{21}ff_u) \\ &\quad + h^3 W_2(c^2_2f_{ttt} + 2c_2a_{21}ff_{tu} + a^2_{21}f^2f_{uu})/2! + \dots \end{aligned} \quad (3.6)$$

Comparing the coefficients of various powers of h in (3.5) and (3.6), we obtain

$$W_1 + W_2 = 2$$

$$c_2 W_2 = 1/2$$

$$a_{21} W_2 = 1/2$$

The solution of this system is

$$a_{21} = c_2, W_2 = 1/c_2, W_1 = 1 - 1/2c_2 \quad (3.7)$$

where $c_2 \neq 0$ is arbitrary. Substituting (3.7) in (3.6), we get

$$u_{j+1} = u_j + hf_j + h^2(f_{tt} + ff_{uu})/2 + h^3 c_2 (f_{ttt} + 2ff_{ttu} + f^2 f_{uuu})/4 + \dots$$

The local truncation error is given by

$$\begin{aligned} T_{j+1} &= u(t_{j+1}) - u_{j+1} \\ &= h^3 [(1/6 - c_2/4)(f_{ttt} + 2ff_{ttu} + f^2 f_{uuu}) + \dots] \end{aligned}$$

which shows that the method (3.4) is of second order. The free parameter c_2 is usually taken between 0 and 1. Sometimes c_2 is chosen such that one of the W's in the method (3.4) is zero or the truncation error is minimum. Such a formula is called an **optimal** formula.

It may be noted that every Runge-Kutta Method should reduce to a quadrature formula when $f(t,u)$ is independent of u with W's as weights and c's as abscissas.

If $c_2 = 1/2$, we get

$$u_{j+1} = u_j + hf(t_j + h/2, u_j + hf_j/2)$$

which is the Euler's method with spacing $h/2$. It reduces to the mid-point quadrature rule when $f(t,u)$ is independent of u .

For $c_2 = 1$ we get

$$u_{j+1} = u_j + h[f(t_j, u_j) + f(t_j + h, u_j + hf_j)]/2$$

which reduces to the trapezoidal rule when $f(t,u)$ is independent of u .

3.1.2 Third Order Runge-Kutta Method

Here we define

$$\begin{aligned} K_1 &= hf(t_j, u_j) \\ K_2 &= hf(t_j + c_2 h, u_j + a_{21} K_1) \\ K_3 &= hf(t_j + c_3 h, u_j + a_{31} K_1 + a_{32} K_2) \\ u_{j+1} &= u_j + w_1 K_1 + w_2 K_2 + w_3 K_3 \end{aligned}$$

Expanding by Taylor series, we get six equations for eight parameters.

$$\begin{aligned} a_{21} &= c_2 & c_2 w_2 + c_3 w_3 &= 1/2 \\ a_{31} + a_{32} &= c_3 & c_2^2 w_2 + c_3^2 w_3 &= 1/3 \\ w_1 + w_2 + w_3 &= 1 & c_2 a_{32} w_3 &= 1/6 \end{aligned} \quad (3.8)$$

Equations (3.8) are typical of all the Runge-Kutta methods; the sum of the a_{ij} in any row equals the corresponding c_i , and the sum for the w_i 's equals 1. Equations (3.8) are linear in w_2 and w_3 and have a solution for w_2 and w_3 if and only if (3.9) holds

$$\begin{vmatrix} c_2 & c_3 & -1/2 \\ c_2^2 & c_3^2 & -1/3 \\ 0 & c_2 a_{32} & -1/6 \end{vmatrix} = 0 \quad (3.9)$$

Simplifying, we get

$$c_2(2-3c_2)a_{32} - c_3(c_3 - c_2) = 0, \quad c_2 \neq 0 \quad (3.10)$$

Thus, we pick c_2 , c_3 and a_{32} to satisfy (3.10). We can do this in most cases by picking c_2 and c_3 arbitrarily and setting

$$a_{32} = c_3(c_3 - c_2) / c_2(2 - 3c_2)$$

However, if $c_3=0$ or $c_2=c_3$, then $c_2=2/3$ and a_{32} is arbitrarily chosen (nonzero). We then calculate w_i 's and a_{ij} 's from Equations (3.8). We display the solution in the form

c_2	a_{21}		
c_3	a_{31}	a_{32}	
	w_1	w_2	w_3

Nystrom

$2/3$	$2/3$		
$2/3$	0	$2/3$	
	$2/8$	$3/8$	$3/8$

Nearly Optimal

$1/2$	$1/2$		
$3/4$	0	$3/4$	
	$2/9$	$3/9$	$4/9$

Classical

$1/2$	$1/2$		
1	-1	2	
	$1/6$	$4/6$	$1/6$

Heun

$1/3$	$1/3$		
$2/3$	0	$2/3$	
	$1/4$	0	$3/4$

The classical Runge-Kutta method is most often used because of its simplicity and moderate order.

3.2 IMPLICIT RUNGE-KUTTA METHODS

The implicit Runge-Kutta method using v slopes is defined as

$$K_1 = hf(t_j + c_i h, u_j + \sum_{m=1}^v a_{im} k_m)$$

$$u_{j+1} = u_j + \sum_{m=1}^v W_m K_m \quad (3.11)$$

where

$$c_i = \sum_{j=1}^v a_{ij}, \quad i = 1, 2, \dots, v$$

and a_{ij} , $1 \leq i, j \leq v$, W_1, W_2, \dots, W_v are arbitrary parameters. The slopes K_m are defined implicitly. The number of unknown parameters are $v(v+1)$. We now give the derivation for the case $v = 1$. We have

$$K_1 = hf(t_j + c_1 h, u_j + a_{11} K_1)$$

$$u_{j+1} = u_j + W_1 K_1 \quad (3.12)$$

The Taylor series gives

$$u(t_{j+1}) = u(t_j) + hu'(t_j) + h^2 u''(t_j)/2 + \dots$$

$$= u(t_j) + hf(t_j, u(t_j)) + h^2 (f_t + ff_u)/2 + \dots$$

and

$$K_1 = h(f(t_j, u_j) + c_1 hf_t + a_{11} K_1 f_u + \dots)$$

$$= (hf + c_1 h^2 f_t + ha_{11} f_u K_1) + O(h^3)$$

$$= hf_j + h^2 (c_1 f_t + a_{11} ff_u) + O(h^3)$$

Substituting (3.13) into (3.12) and comparing the coefficients of h and h^2 , we get

$$c_1 = a_{11}$$

$$W_1 = 1$$

$$W_1 c_1 = 1/2$$

We obtain

$$W_1 = 1, c_1 = a_{11} = 1/2$$

The second order implicit Runge-Kutta method becomes

$$K_1 = hf(t_j + h/2, u_j + K_1/2)$$

$$u_{j+1} = u_j + K_1$$

For $v = 2$, the implicit Runge-Kutta method (3.11) becomes

$$K_1 = hf(t_j + c_1 h, u_j + a_{11} K_1 + a_{12} K_2)$$

$$K_2 = hf(t_j + c_2 h, u_j + a_{21} K_1 + a_{22} K_2)$$

$$u_{j+1} = u_j + W_1 K_1 + W_2 K_2$$

where the parameter values

$$W_1 = 1/2, W_2 = 1/2$$

$$c_1 = (3 - \sqrt{3})/6, c_2 = (3 + \sqrt{3})/6$$

$$a_{11} = 1/4, a_{12} = 1/4 - \sqrt{3}/6$$

$$a_{21} = 1/4 + \sqrt{3}/6, a_{22} = 1/4$$

lead to a fourth order method.

CHAPTER FOUR

PARALLEL RUNGE-KUTTA METHOD

In England the drain pipes are placed outside the houses in order to simplify repair service. Repairs are necessary because the pipes have been placed outside the houses.

PARALLEL RUNGE-KUTTA METHOD

4.1 PARALLEL RUNGE-KUTTA METHOD

The Runge Kutta methods are widely used for solving initial value problem. These methods provide approximations which converges to the true solution as step size (h) tends to zero, and also have the advantage of self-starting.

However, the Runge-Kutta method, in its usual form, is not parallelizable. But, by some modification (which we are going to present) it can be converted into a parallel one, and therefore, computation (per step) can be done in parallel. This is the serious drawback of Runge-Kutta method that it involves considerably more computation per step, and hence by parallelizing the method, this drawback can be minimised effectively.

The Runge Kutta method with v slopes can be written as

$$K_1 = hf(t_j, u_j)$$

$$K_2 = hf(t_j + c_2 h, u_j + a_{21} K_1)$$

$$K_3 = hf(t_j + c_3 h, u_j + a_{31} K_1 + a_{32} K_2)$$

$$K_4 = hf(t_j + c_4 h, u_j + a_{41} K_1 + a_{42} K_2 + a_{43} K_3)$$

⋮
⋮
⋮

$$K_v = hf(t_j + c_v h, u_j + \sum_{j=1}^{v-1} a_{vj} K_j)$$

and

$$u_{j+1} = u_j + W_1 K_1 + W_2 K_2 + \dots + W_v K_v$$

Since K_{i+1} depends upon K_i the method is a serial one, but by assuming certain constants as zero, we can convert the Runge Kutta method to a parallel method. Assuming coefficient of K_{2j} as zero in K_{2j+1} , $j=1,2,\dots,(v-1)/2$, we get

$$K_1 = hf(t_j, u_j)$$

$$K_2 = hf(t_j + c_2h, u_j + a_{21}K_1)$$

$$K_3 = hf(t_j + c_3h, u_j + a_{31}K_1)$$

$$K_4 = hf(t_j + c_4h, u_j + a_{41}K_1 + a_{42}K_2 + a_{43}K_3)$$

$$K_5 = hf(t_j + c_5h, u_j + a_{51}K_1 + a_{52}K_2 + a_{53}K_3)$$

⋮
⋮
⋮

$$K_v = hf(t_j + c_vh, u_j + \sum_{i=1}^{v-2} a_{vi}K_i); \quad v = 2m+1$$

and

$$u_{j+1} = u_j + W_1K_1 + W_2K_2 + \dots + W_vK_v$$

Thus, K_{2j} and K_{2j+1} can be calculated in parallel, since K_{2j+1} does not depend upon K_{2j} . That is the sequence of operations are

First Stage	Calculate K_1
Second Stage	Parbegin
	Calculate K_2
	Calculate K_3
	Parend.
Third Stage	Parbegin
	Calculate K_4
	Calculate K_5
	Parend.

.
 .
 .
 .

(v-2)th stage	Parbegin Calculate K_{v-1} Calculate K_v Parend.
---------------	---

Thus roughly, the time complexity is reduced to half of its original value, since we are calculating K_{2j} and K_{2j+1} in parallel. However due to modification, some of its feature, like stability is affected. To improve the solution we will implement predictor correctors, which itself will be in parallel.

Now, we derive the third order parallel Runge-Kutta method in which k_2 and k_3 can be computed in parallel.

4.2 THIRD ORDER PARALLEL RUNGE KUTTA METHOD

$$\begin{aligned}
 K_1 &= hf(t_j, u_j) \\
 K_2 &= hf(t_j + c_2 h, u_j + a_{21} K_1) \\
 K_3 &= hf(t_j + c_3 h, u_j + a_{31} K_1); \quad a_{32} = 0. \\
 U_{j+1} &= u_j + W_1 K_1 + W_2 K_2 + W_3 K_3 \qquad (4.1)
 \end{aligned}$$

To determine the parameter c 's, a 's and W 's we expand u_{j+1} in power of h such that it agrees with the Taylor series expansion of the solution of the differential equation,

$$\begin{aligned}
 K_1 &= hf_j \\
 K_2 &= hf(t_j + c_2 h, u_j + a_{21} K_1) \\
 &= hf(t_j + c_2 h, u_j + h a_{21} f_j)
 \end{aligned}$$

$$= h[f_i + h(c_2 f_t + a_{21} f f_t) + h^2(c_2^2 f_{tt} + 2c_2 a_{21} f f_{tu} + a_{21}^2 f^2 f_{uu})]$$

$$K_3 = h[f_i + h(c_3 f_t + a_{31} f f_t) + h^2(c_3^2 f_{tt} + 2c_3 a_{31} f f_{tu} + a_{31}^2 f^2 f_{uu})]$$

expanding left hand side of (4.1) and using these K_i 's, we get

$$u(t_j) + hu'(t_j) + h^2 u''(t_j)/2! + h^3 f'''(t_j)/3! + \dots$$

$$= u(t_j) + hf_j + h^2 [f_t + f f_u]/2! + h^3 [f_{tt} + 2ff_{tu} + f^2 f_{uu} + f_u (f_t + f f_u)]/3!$$

$$= W_1 hf_j + W_2 [hf_j + h^2 (c_2 f_t + a_{21} f f_u)] + W_3 [hf_j + h^2 (c_3 f_t + a_{31} f f_u)] + u(t_j)$$

Comparing Coefficients of h and h^2

$$w_1 + w_2 + w_3 = 1$$

$$w_2 c_2 + w_3 c_3 = 1/2$$

$$w_2 a_{21} + w_3 a_{31} = 1/2$$

Solving these equations, we get

$$w_1 = 1 - w_2 - w_3$$

$$w_2 = a_{31} - c_3 / 2 (c_2 a_{31} - c_3 a_{21})$$

$$w_3 = a_{21} - c_2 / 2 (a_{21} c_3 - a_{31} c_2)$$

Thus by assuming suitable value to c_2, c_3 and a_{21}, a_{31} , we can determine the values of w_1, w_2, w_3 . Similarly, we can calculate higher order parallel Runge-kutta method using Taylor series. A n th order parallel Runge-Kutta method is comparable to $(n-1)$ th order Runge-Kutta method. However, parallel Runge-Kutta method is limited to odd orders as we are calculating two k_i 's parallelly.

4.3 STABILITY ANALYSIS

While numerically solving an initial value problem for ordinary differential equations, an error is introduced at

each integration step due to the inaccuracy of the formula. The magnitude of this so called local truncation error is a measure of the accuracy of the integration formula. The magnitude of the total error depends on the magnitude of the local truncation errors and their propagation. Even when the local error at each step is small, the total error may become large due to accumulation and amplification of these local errors. This growth phenomenon is called **numerical instability**. Consider the simple linear first order differential equation

$$u' = Mu, u(t_0) = u_0 \quad (4.2)$$

where M is a constant. It can be seen that, to a first order approximation, the result obtained from a stability analysis on the above linear equation can be extended to a nonlinear case

$$u' = f(t, u), u(t_0) = u_0 \quad (4.3)$$

where df/du from Equation (4.3) plays role similar to that of the constant M in Equation (4.2). The nonlinear function $f(t, y)$ can be linearized by expansion of the function about the point (t_n, u_n) in the Taylor series truncated after first order terms. The resulting linearized form for Equation (4.3) is given in Equation (4.4)

$$u' = Mu + Bt + C \quad (4.4)$$

where

$$M = (\partial f / \partial u)_n,$$

$$B = (\partial f / \partial t)_n,$$

$$C = [f_n - u_n (\partial f / \partial u)_n - t_n (\partial f / \partial t)_n]$$

It can be argued that the stability characteristics of the linear equation (4.4) are very similar to the stability characteristics of the equation of the form given by (4.3). Since the terms Bt and C will give rise to corresponding terms in both numerical and exact solutions which are also linear in t ($M \neq 0$), we conclude that (4.3) exhibits short-range numerical instability in the neighbourhood of (t_n, u_n) , when the corresponding equation (4.2) with $M = f_u(t_n, u_n)$ exhibits numerical instability. Therefore, the stability analysis will be based on the equation

$$u' = f(t, u) = Mu, \quad u(t_0) = u_0 \quad (4.5)$$

where

$$M = \left(\frac{\partial f}{\partial u} \right)_n$$

and it is assumed that (df/du) is relatively invariant in the region of interest. Equation (4.5) has as its solution

$$u(t) = u(t_0) e^{M(t-t_0)}$$

which at $t = t_0 + nh$ becomes

$$u(t_n) = u(t_0) e^{Mnh}$$

A singlestep method when applied to (4.5) will lead to a first order difference equation which has solution of the form

$$u_n = c_1 (E(Mh))^n$$

where c_1 is a constant to be determined from the initial condition and $E(Mh)$ is an approximation to e^{Mh} . We call singlestep method

Absolutely stable if $|E(Mh)| \leq 1$,

Relatively stable if $|E(Mh)| \leq e^{Mh}$

If $M < 0$, the exact solution decreases as t_n increases and the important condition is the absolute stability, since

the numerical solution must also decrease with t_n . If $M > 0$, the exact solution increases with t_n and we do not want $E(Mh) < 1$: so the relative stability is an important condition.

If Euler's method is used, we obtain

$$u_{n+1} = u_n + hf_n = E(Mh)u_n$$

where $E(Mh) = 1 + Mh$

Obviously, Euler's method is absolutely stable if

$$|1 + Mh| < 1 \text{ or } -2 < Mh < 0$$

4.3.1 Stability of 2nd order Runge-Kutta Method

$$k_1 = hf(t_j, u_j)$$

$$k_2 = hf(t_j + c_2h, u_j + a_{21}k_1)$$

$$u_{j+1} = E[Mh]u_j$$

replacing $f(t_j, u_j)$ by Mu_j

$$k_1 = hf(t_j, u_j) = Mhu_j$$

$$k_2 = hf(t_j + c_2h, u_j + a_{21}k_1) = Mh(1 + a_{21}Mh)u_j$$

$$t_{j+1} = t_j + w_1k_1 + w_2k_2$$

$$E[Mh]u_j = u_j [1 + w_1Mh + w_2Mh(1 + a_{21}Mh)]$$

$$E[Mh] = [1 + w_1Mh + w_2Mh(1 + a_{21}Mh)]$$

But

$$w_1 + w_2 = 1$$

$$c_2w_2 = 1/2$$

$$a_{21}w_2 = 1/2$$

for absolute stability

$$|E[Mh]| \leq 1$$

hence,

$$|1 + Mh + M^2h^2/2| \leq 1$$

Solving this equations as quadratic in Mh , we get

$$-2 \leq Mh \leq 0$$

in which this method is absolutely stable.

4.3.2 Stability of 3rd order parallel Runge-Kutta Method

$$k_1 = hf(t_j, u_j) = Mh u_j$$

$$k_2 = hf(t_j + c_2 h, u_j + a_{21} k_1) = Mh(1 + a_{21} Mh) u_j$$

$$k_3 = hf(t_j + c_3 h, u_j + a_{31} k_1) = Mh(1 + a_{31} Mh) u_j$$

or
$$u_{j+1} = u_j + w_1 k_1 + w_2 k_2 + w_3 k_3$$

$$E[Mh] = 1 + w_1 Mh + w_2 Mh(1 + a_{21} Mh) + w_3 (1 + a_{31} Mh) Mh$$

$$E[Mh] = 1 + Mh(w_1 + w_2 + w_3) + M^2 h^2 (w_2 a_{21} + w_3 a_{31})$$

for absolute stability

$$|E[Mh]| \leq 1$$

But,

$$w_1 + w_2 + w_3 = 1$$

$$w_2 c_2 + w_3 c_3 = 1/2$$

$$w_2 a_{21} + w_3 a_{31} = 1/2$$

Thus

$$-2 \leq Mh + M^2 h^2 / 2 \leq 0$$

or

$$-2 \leq Mh \leq 0$$

Hence the stability of 3rd order parallel Runge-Kutta Method is same as that of 2nd order Runge-Kutta Method. Similarly we can determine the stability of higher order parallel Runge-Kutta Method. In general, the stability of a Parallel Runge-Kutta method reduces to a lower order Runge-Kutta method.

4.4 APPLICATIONS

Now, we presents some applications of parallel Runge-Kutta method.

4.4.1 Higher Order Differential equation

Runge-Kutta method is the most widely used method, and it is particularly suitable in cases wher the computation of higher derivatives is complecated. It can be used for equations of arbitrary order by means of a transformation to a system of first-order equations.

Consider the equation

$$u'' = f(t, u, u'),$$

Let $u' = w$ so that,

$$u' = w$$

$$w' = f(t, u, w).$$

This is a special case of

$$u' = F(t, u, w),$$

$$w' = G(t, u, w).$$

Suppose we choose a third order Runge-Kutta method for approximating the value of (t, u, w) . The new set of values $(t+h, u+k, w+l)$ can be calculated as

$$k_1 = hF(t, u, w)$$

$$k_2 = hF(t+c_2h, u+a_{21}k_1, w+b_{21}l_1)$$

$$k_3 = hF(t+c_3h, u+a_{31}k_1+a_{32}k_2, w+b_{31}l_1+b_{32}l_2)$$

$$k = W_1k_1+W_2k_2+W_3k_3$$

and

$$l_1 = hG(t, u, w)$$

$$l_2 = hG(t+c_2h, u+a_{21}k_1, w+b_{21}l_1)$$

$$l_3 = hG(t+c_3h, u+a_{31}k_1+a_{32}k_2, w+b_{31}l_1+b_{32}l_2)$$

$$l = R_1l_1+R_2l_2+R_3l_3$$

Thus the new values are $(t+h, u+k, w+l)$. In case of third order parallel Runge-Kutta method, these equations in which k_1, k_2 and l_1, l_2 can be done in parallel are given by

$$k_1 = hF(t, u, w)$$

$$k_2 = hF(t+c_2h, u+a_{21}k_1, w+b_{21}l_1)$$

$$k_3 = hF(t+c_3h, u+a_{31}k_1, w+b_{31}l_1)$$

$$k = W_1k_1+W_2k_2+W_3k_3$$

and

$$l_1 = hG(t, u, w)$$

$$l_2 = hG(t+c_2h, u+a_{21}k_1, w+b_{21}l_1)$$

$$l_3 = hG(t+c_3h, u+a_{31}k_1, w+b_{31}l_1)$$

$$l = R_1l_1+R_2l_2+R_3l_3$$

The sequence of operations are

Parbegin

 Calculate k_1

 Calculate l_1

parend

parbegin

 Calculate k_2

 Calculate l_2

 Calculate k_3

 Calculate l_3

parend

Similarly, we can choose higher order Runge-Kutta method for better results.

4.4.2 System of Differential Equation

Any nth order initial value problem can be replaced by a system of n first order initial value problems. The system in the vector form may be written as

$$u' = du/dt = f(t,u), t_0 \leq t \leq b$$

$$u(t_0) = u_0$$

where

$$u = \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix}, f(t,u) = \begin{pmatrix} f_1(t, u_1, u_2, \dots, u_n) \\ f_2(t, u_1, u_2, \dots, u_n) \\ \vdots \\ f_n(t, u_1, u_2, \dots, u_n) \end{pmatrix}$$

and

$$u_0 = \begin{pmatrix} u_{1,0} \\ u_{2,0} \\ \vdots \\ u_{n,0} \end{pmatrix}$$

by fifth order Runge-Kutta formula,

$$u(t_{j+1}) = u(t_j) + W_1 k_1 + W_2 k_2 + W_3 k_3 + W_4 k_4 + W_5 k_5$$

where

$$k_i = \begin{pmatrix} k_{1i} \\ k_{2i} \\ \vdots \end{pmatrix}$$

CHAPTER FIVE
PREDICTOR CORRECTOR METHOD

*" I could have done it in
a much more complicated
way " said the red queen,
immensely proud.*

LEWIS CARROLL.

PREDICTOR-CORRECTOR METHOD

5.1 PREDICTOR-CORRECTOR METHOD

To solve a differential equation over a single interval, say from $u=u_n$ to $u=u_{n+1}$, we require information only at the beginning of the interval i.e., at $u=u_n$. Predictor-Corrector methods are methods which require function values at $u_n, u_{n-1}, u_{n-2}, \dots$ for the computation of the function at u_{n+1} . A predictor formula is used to predict the value of u at u_{n+1} and then a corrector formula is used to improve the value of u_{n+1} .

Consider, for example, the following predictor corrector method.

$$\begin{aligned} \text{P: } u_{n+1} &= au_n + bhf_n \\ \text{C: } u_{n+1} &= cu_n + h(df_{n+1} + ef_n)/2 \end{aligned} \quad (5.1)$$

The coefficients a, b, c, d and e can be determined by using Taylor's series. The values f_{n+1} and f_n which are required on the right hand side of (5.1) are obtained by Runge-Kutta method or by some other method. Due to this reason, these methods are called **starter methods**. For practical problems, the Runge Kutta method together with predictor corrector methods have been found to be most successful combination.

5.2 PARALLEL PREDICTOR-CORRECTOR METHOD

In this section, we present two different schemes for parallelizing predictor corrector methods, which can be used by Parallel Runge-Kutta method. However, these predictor-corrector methods are not limited to the parallel Runge Kutta method, and are general in nature. In first scheme u_{n+1}^P and u_n^C are calculated at two different points parallelly. In second scheme, we calculate u_{n+1} for predictor-corrector parallelly at the same point.

First, we present an example of predictor corrector method with its two different parallel versions.

$$\begin{aligned} \text{P: } u_{n+1} &= u_n + hf_n \\ \text{C: } u_{n+1} &= u_n + h(f_{n+1} + f_n)/2 \end{aligned} \quad (5.2)$$

Parallel Version First:

Note that corrector does not depend upon the predictor and hence a corrector of u_n and predictor of u_{n+1} or a predictor of u_{n+1} and a corrector of u_n can be calculated parallelly. Hence by changing the subscript of (5.2) in corrector method, we get

$$\begin{aligned} \text{P: } u_{n+1} &= u_n + hf_n \\ \text{C: } u_n &= u_{n-1} + h(f_n + f_{n-1})/2 \end{aligned}$$

In this case, the sequence of computation can be divided into two parts:

$$\begin{aligned} \dots &\text{----> } u_{i+1}^P \text{----> } f_{i+1}^P \text{----> } \dots \\ \dots &\text{----> } t_i^C \text{----> } f_i^C \text{----> } \dots \end{aligned}$$

These two can be executed parallely on sepearate processors as shown below:

	Predictor	Corrector
	⋮	⋮
$t=r$	u_{i-1} f_{i-1}	u_{i-2} f_{i-2}
$t=r+k$	u_i f_i	u_{i-1} f_{i-1}
$t=r+2k$	u_{i+1} f_{i+1}	u_i f_i
	⋮	⋮

Thus the predictor and corrector can be evaluated at the different points parallely.

Parallel Version Two:

In this scheme, we evaluate Predictor and Corrector parallely at same point. In this scheme Corrector does not depends upon the latest evaluated predicted value but depends upon the predicted value evaluated one cycle before. Hence, both Predictor and Corrector can be evaluated simultaneously.

$$P: u_{n+1} = u_n + hf_n$$

$$C: u_{n+1} = au_n + h(bf_n + cf_{n-1})/2 \tag{5.3}$$

To determine the coefficients, we expand the L.H.S. and R.H.S. of (5.3) by Taylor Series.

L.H.S.

$$u_n + hu'_n + h^2 u''_n / 2! + \dots$$

R.H.S.

$$au_n + h/2 [bu'_n + c(u'_n - hu''_n + h^2 u'''_n / 2! + \dots)]$$

Comparing Coefficients, we get

$$a = 1$$

$$b = 3$$

$$c = -1$$

$$u_{n+1} = u_n + h(3f_n - f_{n-1})/2$$

Hence the parallel predictor corrector becomes

$$P: u_{n+1} = u_n + hf_n$$

$$C: u_{n+1} = u_n + h(3f_n - f_n)/2$$

This is a third order method, since coefficients are matched upto order of h^2 .

5.3 HYBRID METHODS

These methods are also called multistep method with nonstep points. To increase the order of the method, the method is modified by including a linear combination of the slopes at several points between u_n and u_{n+1} .

The K-step method with one non-step point can be written in the form

$$u_{n+1} = \sum_{j=1}^k a_j u_{n-j+1} + h \sum_{j=0}^k b_j f_{n-j+1} + hc_1 f_{n-\theta+1}$$

A Predictor-Corrector using hybrid method is given by

$$\text{Predictor for predictor: } u_{n+1/2} = u_n + hf_n/2$$

$$\text{Predictor: } u_{n+1} = u_n + h(2f_{n+1/2} - f_n)$$

$$\text{Corrector: } u_{n+1} = u_n + h(f_{n+1} + 4f_{n+1/2} + f_n)/6$$

where a_j 's, b_j 's, c_1 and θ are arbitrary $0 < \theta < 1$

5.3.1 Parallel Version One

The parallel predictor - corrector for such scheme can be written as, (replacing n by $n-1$ in corrector)

$$\text{Predictor for predictor: } u_{n+1/2} = u_n + hf_n/2$$

$$\text{Predictor: } u_{n+1} = u_n + h(2f_{n+1/2} - f_n)$$

$$\text{Corrector: } u_n = u_{n-1} + h(f_n + 4f_{n-1/2} + f_{n-1})/6$$

Thus, predictor and corrector can be executed parallelly.

5.3.2 Parallel Version Two

Derivation of corrector: let the Correctors be of the form,

$$u_{n+1} = au_n + h(bf_n + cf_{n-\theta} + df_{n-1})$$

L.H.S.

$$u_{n+1} = u_n + hu'_n + h^2u''_n/2! + h^3u'''_n/3! + \dots$$

where $\theta = hk$ and $u_n = u(t_n)$

R.H.S.

$$au_n +$$

$$hb u'_n +$$

$$h(cu'_n - c\theta u''_n + c\theta^2 u'''_n/2! - c\theta^3 u''''_n/3! + \dots) +$$

$$h(du'_n - dhu''_n + dh^2u'''_n - dh^3u''''_n + \dots)$$

Comparing Coefficients, we get

$$a=1$$

$$b+c+d=1$$

$$ck+d=-1/2$$

$$ck^2+d=1/3$$

$$ck^3+d=-1/4$$

Solving these equations we get

$$a=1$$

$$b=1-c-d$$

$$c=250/357$$

$$d=27/46$$

$$k=-7/10$$

and

$$u_{n+1}=u_n+h(bf_n+cf_{n-7/10}+df_{n-1})$$

Derivation of Predictor:

$$u_{n+1}=u_n+h(af_{n-7/10}+bf_n)$$

L.H.S.

$$u_{n+1}=u_n+hu'_n+h^2u''_n/2!+h^3u'''_n/3!$$

R.H.S.

$$u_n+h[a(u'_n-7hu''_n/10+49h^2u'''_n/100+\dots)+bu'_n]$$

Comparing Coefficients,

$$a+b=1$$

$$-7a/10=1/2$$

Solving this, we get

$$a=-5/7$$

$$b=12/7$$

or

$$u_{n+1} = u_n + h(-5f_{n-7/10} + 12f_{n-1})/7$$

Derivation of Predictor for Predictor

$$u_{n-7/10} = u_n + ahf_n$$

L.H.S.

$$u_n - 7hu'_n/10$$

R.H.S.

$$u_n + ah u'_n$$

From this, we get $a = -7/10$

$$u_{n-7/10} = u_n - 7f_n/10$$

Hence the parallel hybrid predictor-Corrector method becomes

$$u_{n-7/10} = u_n - 7hf_n/10$$

$$u_{n+1} = u_n + h(-5f_{n-7/10} + 12f_{n-1})/7$$

$$u_{n+1} = u_n + h(bf_n + cf_{n-7/10} + df_{n-1})$$

Where $c = 250/357$ $d = 27/46$ $b = 1 - c - d$

In the same manner, we can derive different predictor-Correctors of higher orders.

5.4 ALGORITHM

In this section, we present an algorithm to demonstrate, parallel Runge-Kutta method, with parallel Predictor-Corrector. This is an oversimplified version, to show how parallel Runge-Kutta method and Parallel Predictor-Corrector can be used simultaneously. In this algorithm we choose fifth order parallel Runge-Kutta method. For the Parallel Predictor Corrector method we select the fifth order parallel Adams-Moulton formula,

$$u_{n+1}^p = u_n + h[55f_n - 59f_{n-1} + 37f_{n-2} - 9f_{n-3}]/24$$

and

$$u_{n+1}^c = u_{n-1} + h[9f_n^p + 19f_{n-1} - 5f_{n-2} + f_{n-3}]/24$$

Algorithm

```
Get the values of
    u' = f(t,u),
    u(t0) = u,
    step-size = h,
    t1 = t0 + kh, the value at which u has to be
                    calculated,
    c's, a's and W's.
while t < t1 do
Begin
    Calculate k1 = hf(tj, uj)
    parbegin
        Calculate k2 = hf(tj+c2h, uj+a21k1)
        Calculate k3 = hf(tj+c3h, uj+a31k1)
    parend
    parbegin
        Calculate k4 = hf(tj+c4h, uj+a41k1+a42k2+a43k3)
        Calculate k5 = hf(tj+c5h, uj+a51k1+a52k2+a53k3)
    parend
    /* calculate u(tj+1)=u(tj)+h */
    Calculate u(tj+1) =
        u(tj)+w1k1+w2k2+w3k3+w4k4+w5k5)
    parbegin
        /* Adams-Moulton Predictor-Corrector */
        upj+1=uj+h[55fj-59fj-1+37fj-2-9fj-3]/24
        ucj=uj-1 +h[9fpj+19fj-1-5fj-2+fj-3]/24
    parend
    t = t+h
end
end
```

To obtain better approximation, we can choose a higher order parallel Runge-Kutta method and parallel predictor-corrector of higher order, such as hybrid predictor corrector. This depends upon the accuracy desired to solve the initial value problem.

Although laborious, the Runge-Kutta method is the most widely used one since it gives reliable starting values and is particularly suitable when the computation of higher derivatives is complicated. When the starting values have been found, the computations for the rest of the interval can be continued by means of the predictor-corrector methods. Hence, Predictor-Corrector methods are of special importance.

CONCLUSION

CONCLUSION

The objective of this Dissertation was to develop Parallel Runge-Kutta method. Due to parallelisation, there were some factors, such as stability, truncation error, etc., which were effected. To improve the solution, we presented two different versions of parallel Predictor-Corrector methods. However the parallel Predictor-Corrector method could be used parallely only once. The goal of this work was to present a solution, in which parallel Runge-Kutta method and parallel Predictor-Corrector method can be used simultaneously, to improve the solution.

Due to time limitation and other factors, some features could not be dealt into. The work is still open ended. Suggestions in areas, where there is some scope of improvements are listed below :

Due to parallelisation, stability of the method decreases. No work has been done to improve the stability of the method. This can be done by changing the step size within the iteration i.e., step size is not uniform.

This work is limited to lower order methods. In general, higher order methods provide better approximation.

The multistep methods can be more effectively used and the accuracy will be higher in these methods.

The load balancing factor was not considered which is very important in parallel processing.

REFERENCES

REFERENCES

1. Franklin M.A., " Parallel solution of ordinary differential equations, " Transactions on Computers, VolC-27 nr. 5, May 1978.
2. Worland, P.B. " Parallel methods for the numerical solution of ordinary differential equations ", IEEE Transactions on Computers, October 1976.
3. Rosser, J.B. " A Runge-Kutta for all seasons ", SIAM Rev., Vol 9, 1967.
4. Miranker, W.L., " A survey of parallelism in numerical analysis ", SIAM Review, Vol. 13 nr. 14. October 1971.
6. Hwang, K. & Briggs, F.A., " Computer Architecture & Parallel Processing ", McGraw Hill Book Company 1985.
7. Nicholas Carriger & David Gelernter, " How to write parallel programs ", ACM Computers.
8. Harold, S. Stone, " High performance Computer Architecture ", Addison - Wesley Publishing Company. algorithm ", Prentice Hall, Inc., 1989.
9. Wilf, H.S. , " Algorithms & Complexity ", Prentice Hall, Englewood Cliff, N.J., 1986.
10. Quinn, M.J., " Designing Efficient Algorithms for Parallel Computers ", McGraw Hill, New Delhi, 1987.
11. Fromberg, C.E., " Introduction to Numerical Analysis ", Addison - Wesley Publishing Company, 1972.
12. Ralston, A., " A First Course in Numerical Analysis ", McGraw Hill Book Company , 1965.

13. Duncan, R., " A survey of Parallel Computer Architectures ", IEEE Trans. Computers Vol 23, Feb 1990.
14. Sastry, S.S., " Introductory Methods of Numerical Analysis ", Prentice-Hall of India , 1982.