Parallel Algorithms of Linear Systems And Initial Value Problems

Salman Hassan Abbas

Ph. D. 1990

GIFT هَدِيَّة

PARALLEL ALGORITHMS OF LINEAR SYSTEMS AND INITIAL VALUE PROBLEMS

BY

SALMAN HASSAN ABBAS
(B.SC,M.SC)

جامعة البحرين
المكتبة
LIBRARY
University of Bahrain

THESIS

submitted in accordance with the requirements of

THE UNIVERSITY OF LIVERPOOL

for the degree of

DOCTOR IN PHILOSOPHY

NOVEMBER 1990



To:

My wife, and my children, Anam, Ali and Huda.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my supervisor Professor L.M. Delves, for his invaluable help, suggestions and encouragement which made the task of completing this work possible. Many thanks also to Dr C. Pursglove for his help and advice with this research.

I would like to thank the University of Bahrain for awarding me the scholarship which enabled me to carry out the present research.

My deep and great appreciation to my family in Bahrain for their encouragement and constant support. I am grateful to Mrs Shirley Gilbertson for her skilled and speedy typing of this thesis.

ABSTRACT

This work is concerned with the study and implementation of parallel algorithms for solution of linear system and initial value ordinary differential equations on MIMD machine.

In Chapter 1, we discuss the available types of parallel computers, showing the need for higher performance and describe a summary of Ada language. In Chapter 2, we present a survey of parallel algorithms for solving a system of linear equations. In Chapter 3, a summary of basic material is given, together with the recent parallel methods on the numerical solution of intial value problems. Chapter 4 gives a full description of two parallel algorithms to solve a system of linear equations together with the theoretical and actual costs. In Chapter 5, we introduce three types of parallel second-order variable block length methods for solution of initial value problems; a stability analysis shows that all the methods are stable, while method three and four retain satisfactory properties in a parallel environment. We also give the experimental and the theoretical costs. Finally, Chapter 6 contains a description of an algorithm for solving initial value problems using a new fourth order method.

All the computational results we present in this work are carried out on Sequent Balance 8000.

CONTENTS

		Page No
CHAPTER 1	Introduction	
1.1	The Need for Parallel Computers	1
1.2	Classification of Computer Architectures	2
1.3	SIMD Processors	3
1.3.1	Massively Parallel Machines	6
1.4	MIMD Processors	17
1.5	Processor Organizations	22
1.6	Complexity	25
1.7	The Use of Parallelism for Evaluating Arithmetic Expression	27
1.8	The Algorithm Must Fit the Architecture	31
1.9	Parallel Processing in Ada	32
CHAPTER 2	A Survey of Parallel Methods in Numerical Linear Algebra	
2.1	Introduction	39
2.2	Gauss Elimination Method	40
2.3	A Parallel Algorithm for Positive Definite Symmetric Matrix	44
2.4	Gauss-Jordan Method	47
2.5	Cholesky Method	50
2.6	WZ Factorization	53
2.7	Linear Recurrence Relations and Algorithms for their Solution	57
2.8	Dense Triangular Systems	62
2.9	Tridiagonal Systems	66
2.10	Parallelisation of Iterative Algorithms	79
CHAPTER 3	A Survey of Parallel Algorithms for Ordinary Differential Equations	
3.1	Introduction	82
3.2	Initial Value Problem	82
3.3	Discrete Variable Methods	83
3.4	Runge-Kutta Methods	86
3.5	Linear Multistep Methods	89
3.6	Predictor-corrector Methods	91
3.7	Conditional Stability Theory	93
3.8	Stiff Equations	98
3.9	Overview of Work on Parallel Methods	101
3.10	Predictor-Corrector Methods	102
3.11	Block Methods	104
3.11.1	Block One-Step Methods	105
3.11.2	Predictor-Corrector Block Methods	109
3.12	Multi-Block Methods	110
3.13	Multiplexing and Interpolation	114
3.14	One-Parameter Method	115

		Page No.
CHAPTER 4	The Solution of Sets of Simultaneous Linear Equations on Parallel Processors	
4.1	Introduction	119
4.2	Elimination Method: A Parallel Row Oriented Algorithm	120
4.2.1	Row Oriented Algorithm	120
4.2.2	Backward Substitution	121
4.3	The Predicted Cost of Row Elimination Method	124
4.4	Test Problem	126
4.5	The Computation Results	127
4.6	Speed Up and Efficiencies	130
4.7	Solving Block Systems of Equations by Block LU Decomposition followed by Forward and Backward Substitution	136
4.7.1	LU Decomposition	136
4.7.2	Forward and Backward Substitution	140
4.8	The Predicted Cost of Block System	145
4.9	Computational Results	149
4.10	Optimum Number of Processors	157
4.11	Conclusion	158
CHAPTER 5	Parallel Solution of Initial Value Problems Using Second Order Block Methods	
5.1	Introduction	160
5.2	Methods 1 and 2	160
5.2.1	Principal Local Truncation Errors	161
5.2.2	Stability Analysis	162
5.2.3	Parallel Algorithms	167
5.2.4	Theoretical Costs	173
5.2.5	Test Problems	175
5.2.6	The Results for Algorithm 1	176
5.3	Method 3	180
5.3.1	Principal Local Truncation Error	181
5.3.2	Stability Analysis	182
5.4	Method 4	183
5.4.1	Principal Local Truncation Error	184
5.4.2	Stability Analysis	185
5.5	The Parallel Algorithms for Method 3 and 4	187
5.5.1	The Comparison between Algorithm 3 and 4	192
5.6	Theoretical Costs for Algorithms 3 and 4	193
5.7	The Results for Algorithms 3 and 4	196
5.8	Conclusion	216
CHAPTER 6	Parallel Solution of Initial Value Problems Using Fourth Order Block Methods	
6.1	Introduction	220
6.2	The Outline of a Fourth Order Method	220
6.3	Principal Local Truncation Error	223
6.4	Stability Analysis	223

		Page No
6.5	Parallel Algorithm	227
6.6	Theoretical Cost	229
6.7	Comparsion With Other Methods	230
6.7.1	Predicted Cost of Adams Methods	232
6.8	Numerical Results	233
6.10	Discussion and Conclusion	239
CHAPTER 7	Suggestion for Future Work	
7.1	Future Work on Linear Systems	241
7.1.1	Block Gaussian Elimination	241
7.1.2	Direct Factorization With Partial Pivoting	245
7.1.3	Block Gauss-Jordan Method	247
7.1.4	Choleski's Method	252
7.2	Future Work on Initial Value Problem	253
7.2.1	Block Methods	253
7.2.1	Runge-Kutta Methods	262
REFERENCES		266
APPENDIX A		271
APPENDIX B		274
APPENDIX C		278
APPENDIX D		282
APPENDIX E		293
APPENDIX F		297
APPENDIX G		304
APPENDIX H		312
APPENDIX I		314

CHAPTER 1

Introduction

In this chapter we present a survey on the subject of parallel computers and parallel computation.

1.1 The Need for Parallel Computers

There are many reasons for the use of parallel computers, some of the most important are:-

1) The increase of computing speed to obtain faster results for existing problems;
2) The ability to solve problems that require more data and take a long time to solve on sequential computers;
3) Solving problems with inherent parallelism.

Computers of this type have indeed been available for some time, examples are the ILLIAC IV, STARAN, the CRAY-1 and, more recently, the Sequent Balance 8000. In the next section, the classification of computer architectures will be described.

1.2 Classification of Computer Architectures

In [9], a taxonomy of parallel computers is given. This taxonomy is particularly useful for describing the features of multi-processors, and parallel computers in this taxonomy are characterized by the number of independent paths memory along which data can move in parallel, and by the number of independent instructions which can be executed in parallel.

Thus, four classes of computers are given as:-

1) Single Instruction Stream - Single Data Stream (SISD) which represents most sequential computers. An SISD Computer may have multiple functional units, but these are under the direction of a single control unit.

2) Single Instruction Stream - Multiple Data Stream (SIMD) which includes most array processors. A processor array executes a single stream of instructions, but contains a number of arithmetic processing units, each capable of fetching and manipulating its own data.

3) Multiple Instruction Stream - Single Data Stream (MISD) which includes specialized streaming organizations using multiple instruction streams on a single sequence of data.

4) Multiple Instruction Stream - Multiple Data (MIMD) which includes most multiprocessors.

The following figures show the classification of above computers.

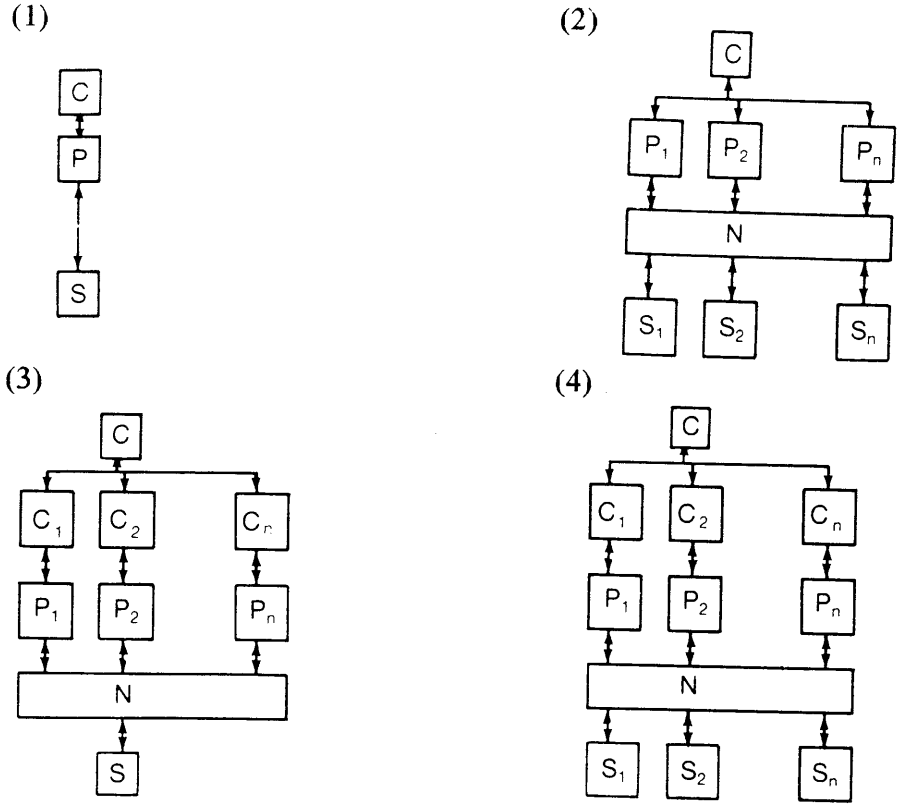


Figure 1.1 - where C,P,N and S denote to the Control Unit, processor, data organization and the store respectively (U. Schendel [1984]).

1.3 SIMD Processors

There are three basic types of SIMD processors which are mentioned in [10]. That is, processors characterized by a master instruction applied over a vector of related operands, the following types are:

1) The Array Processor

This type is most frequently chosen as a model for an SIMD computer. It consists of n processors which simultaneously execute the same instructions. The control unit analyses the program contained in the main store. The two most well-known examples of this computer are ILLIAC IV and ICL-DAP.

We shall give a brief look at the design of ICL-DAP. The ICL-DAP

consists of an array 64 x 64 one-bit processors instructed to perform the same operation by a single master processor, the master control unit. Each processor can communicate with four neighbours. In Great Britain ICL manufactures the DAP, also delivered the first production models in 1980. The DAP is designed to be integrated into an ICL-2900 computer. Figure 1.2 shows a typical 2900 system, which consists of an order code processor and store

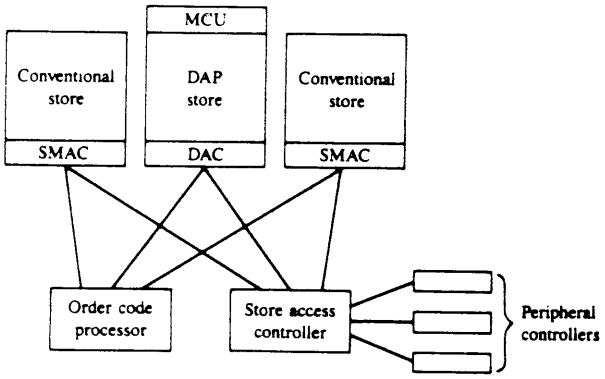


Figure 1.2 SMAC, store multiple access controller. DAC, DAP access controller and MCU, master control unit. (Hockney and Jesshope [1981])

access controller, both cross connected with a number of memory units. One or more of these memory units may be a DAP, which provides memory and can also be instructed by a processor to execute its own DAP code. The store access controller provides access to memory and transfer blocks of data between memory units.

The main components of the DAP and data highways are shown in Figure 1.3. The processors are arranged into a 64 x 64 array. Each 64-bit word in

2900 corresponds to a row across the DAP memory. Increasing the 2900 address first increases down the columns of the DAP array and then through the 4k DAP address space. The column highway fetches data words from DAP memory, fetches rows to a master control unit register and fetches DAP instructions from the DAP store.

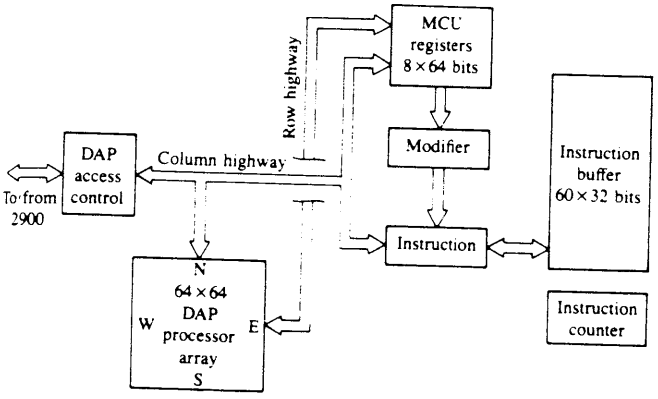


Figure 1.3 - Major components of the DAP. (Hockney and Jesshope [1981])

DAP instructions are stored two per word. The row highway has one-bit for each row of processors in the array and is used for transferring data to and from master control unit registers. Finally, the DAP single processing element is illustrated in Figure 1.4. One of the objectives of the system designs was to use inexpensive, good quality technology to achieve high processing power. A processing element contains two multiplexers, a 1-bit full adder, three 1-bit registers and 4k 1-bit memory.

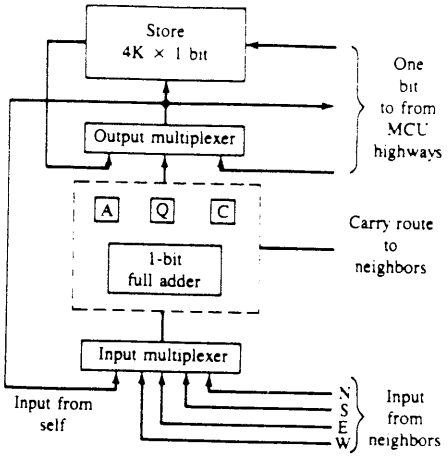


Figure 1.4 - single DAP processing element. (Hockney and Jesshope [1981])

The A register gives programmable control. The other registers are accumulator Q and the C register is for carry store. The adder adds Q, C and the input to the processing element, giving sum and carry outputs, which are stored in the Q and C registers respectively. The ICL-DAP is fully described in [18].

1.3.1 Massively Parallel Machines

In this section we shall give a brief description of two super-computer machines.

1. The DAP 610

As we know that the DAP (Distributed Array of Processors) technology was manufactured in Britain by ICL, before being taken over and further developed by AMT (Active Memory Technology) in Irvine, California.

AMT's first production was DAP 510, which has 1,024 processors and fits under a desk. The new DAP 610 has 4,096 processors, and thus has four times the performance of the original machine. The DAP is hosted by a SUN workstation, and powerful Fortran development tools are provided, permitting

the DAP to be programmed from the SUN under UNIX.

The performance of the DAP 610 is 200 million floating point operations - or 1,600 million integer operations - per second.

More information can be found in [46].

2. The Connection Machine

The Connection Machine was manufactured by Thinking Machines Corporation (TMC) in Cambridge, Massachusetts. The Connection Machine is fine grained, it was designed to contain up to 4 modules of 16,384 processing elements each, for a maximum configuration of 65,536 processing elements. Each of these elements has 4,096 bits of memory and 1-bit-wide ALU (Arithmetic and Logical Units), so that adding two 32-bit numbers takes 32 cycles. Each module is controlled by a single microcontroller which executes macroinstructions originating in a host computer. A machine cycle (3 microcontroller states) takes about 750 ns, yielding a raw system throughput exceeding 2 billion 32-bit operation per second. Communication is allowed by three independent communication networks:

1) A 1-bit-wide global-OR network, 2) A 2-dimensional grid; and

3) A 12-dimensional hypercube with 16 processors at each vertex.

A global-OR network permits such operation as global minimum, global-OR to be executed in a time span proportional to the length of the operands: on the order of microseconds per bit. The architecture of the Connection Machine is illustrated in the following figure:

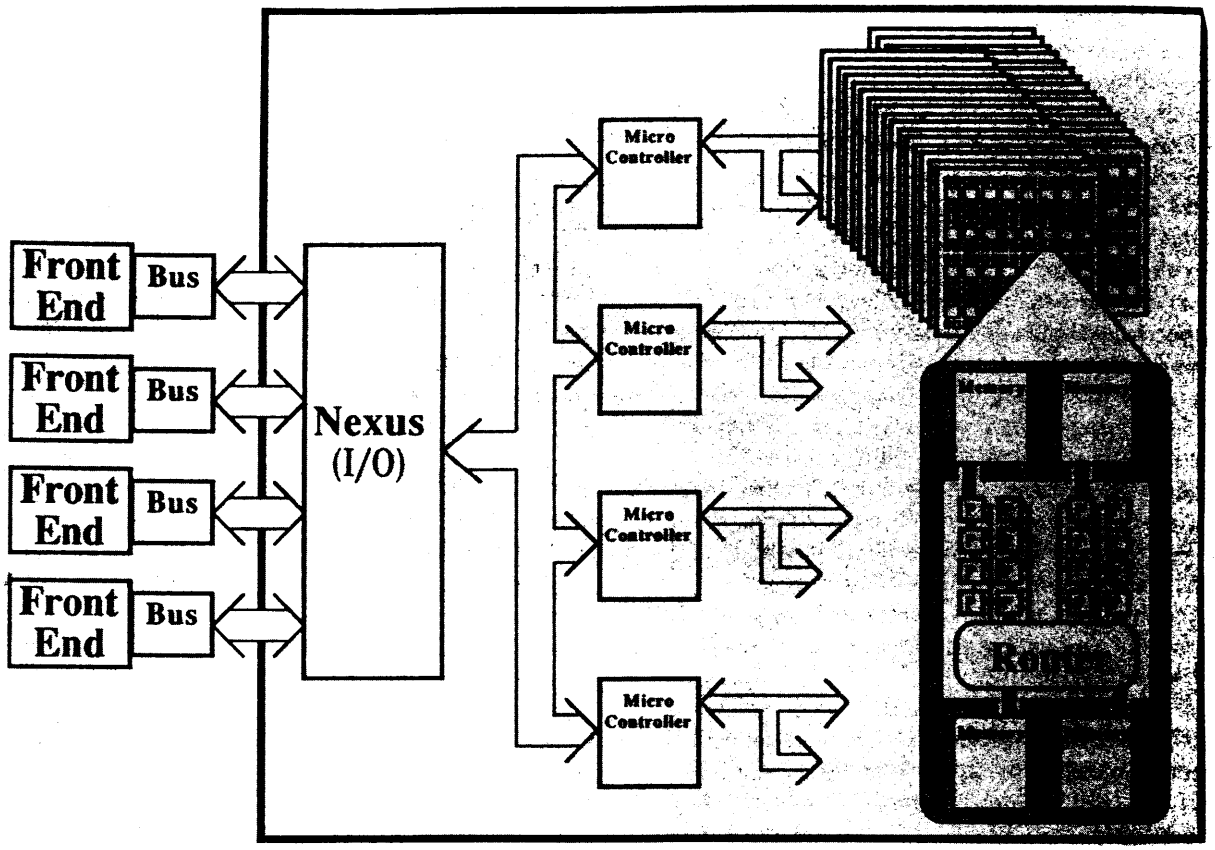


Figure 1.5 - The Connection Machine and one of 65,536 processing elements (K.A. Frenkel [1986])

The grid permits instantaneous communication between nearest neighbours. The hypercube network supports full packet-switched communication between arbitrary processors. More details can be found in [12].

2) The Piplined Processor

One of the most successful ways of speeding up arithmetic is known as the

pipeline. The basic idea is to introduce some simultaneity processing by breaking a complex, time consuming function into a series of simpler, faster operations. Figure 1.6 shows that instead of having many processors working in parallel, it just has one extremely elaborate processor. Pipelines are designed to speed up the evaluation of vector loops, as a simple example

$$C_i = A_i + B_i \ , \quad i = 1, \ldots, n \quad ,$$

Initially, the A and B arrays are fetched from memory to the general registers. Processing begins as A_1 and B_1 sent from registers to segment 1 of a pipeline processor. When segment 1 processing of these two operands is complete, the results are sent to segment 2, and so on until the result C_1 emerges from segment K. But when the intermediate results of $A_1 + B_1$ passed to segment 2. A_2 and B_2 can enter the pipeline, through segment 1. Thus, by the time C_1 emerges from the pipeline, the result of A_2 and B_2 are nearly completed and entering segment K and so on until the pipe to segment 1 where A_{k+1} and B_{k+1} are entering. Notice that the process is very much like an assembly line, when each segment does a little further processing of the intermediate results, and segment K is given the completed summuation.

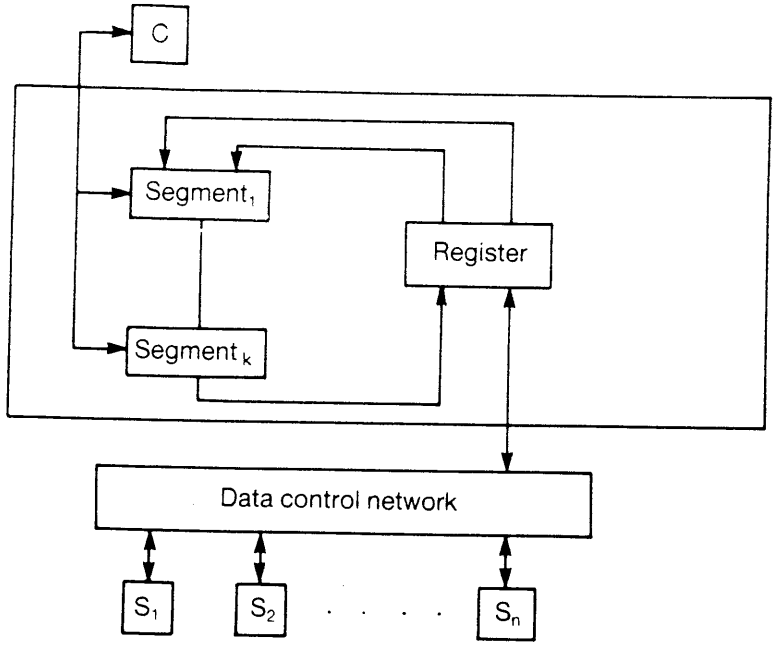


Figure 1.6 - model of pipeline computer C, S is control unit and store. (U. Schendel [1984])

Pipelined processors can be classified as a memory to memory or a register to register, depending upon how vector instructions are executed. In memory to memory, vectors are streamed directly from memory to arithmetic units and back to memory. In a register to register, internal registers are used to store operands and results. The CYBER-205 has a memory to memory, while the CRAY-1 has register to register.

We now present a brief description of the above computers.

a) **The CRAY-1**

The CRAY-1 is the most famous pipelined processor ever built. The first CRAY-1 went to Los Alamos National Laboratory in April 1976. The diagram of CRAY-1 is shown in Figure 1.7. Since the CRAY-1 is a register to register architecture, the functional units take data from and put results into register. Memory is divided into 16 memory banks that may work simultaneously. There are twelve functional units, organized in four groups; address, scalar, vector and floating point. Each functional unit is

pipelined into single clock segments. Figure 1.7 shows the CRAY-1 registers in relationship to the functional units instruction buffers, input/output channel control registers and memory. The basic set of programmable registers are as follows:

Eight 24-bit address (A) registers, eight 64-bit scalar (S) registers and eight 64-bit word vector (V) registers. There are sixty-four 24-bit buffer registers between the address registers and main memory and sixty-four 64-bit registers between the scalar registers and main memory.

The CRAY-1 can chain together a series of vector operations, allowing a second vector operation to begin as soon as results begin streaming out of the first function unit's pipeline. Useful discussions of the CRAY-1 computer system are given in [37].

b) **The CYBER-205**

Control Data Corporation manufactures the CYBER-205. The first computer delivered to Great Britain was in 1981.

The CYBER-205 has either two or four memory units, each containing one million 64-bit words. The architecture of the CYBER-205 is shown in Figure 1.8. The scalar part of the Central Processor Unit consists of a load/store and five arithmetic functional units. The functional units take arguments and return results to a set of 256 registers of 64-bits.

All the functional units are pipelined and may take new arguments every clock period. Concurrency is limited by the fact that registers can supply at most one pair of arguments to the functional units and accept one result every

clock period. However, the result of any functional unit can be passed directly to another functional unit without first being sent to a register. The CYBER-205 has no vector registers, argument vectors are fetched from memory, and the resultant vectors are stored in memory. A stream unit feeds vectors into arithmetic section of the central processor unit, which consists of one, two or four floating point pipelines and a string unit. Each floating point pipeline consists of five separate pipelined functional units which are as follows: an addition unit, multiplication unit, shift unit, logical unit and delay unit. Each unit can generate results at the rate of 50 million results per second for 64-bit operations and 100 million results per second for 32-bit operations. More details about CYBER-205 can be found in [18].

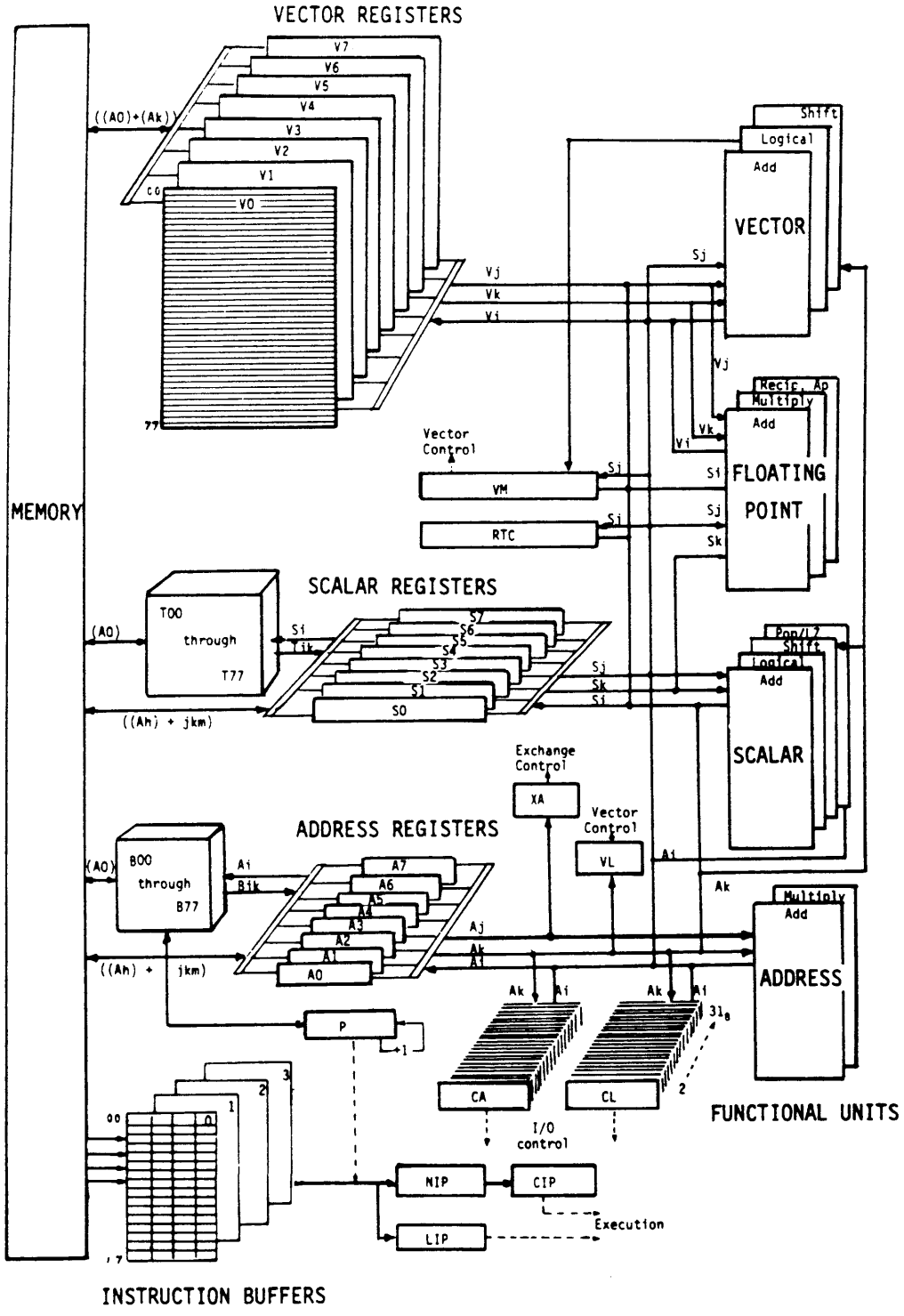


Figure 1.7 - Architecture of the CRAY-1 (R.M. Russell [1978])

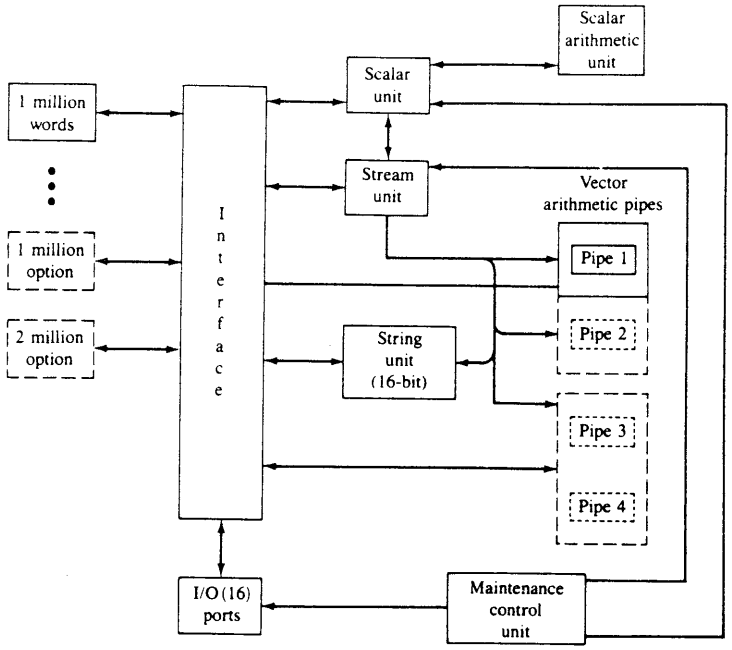


Figure 1.8 - Architecture of the CYBER-205 (M.J. Quinn [1988])

3) The Associative Processor

This is a variation of array processor. Processing elements are not directly addressed. The processing elements of this type of computer are activated when a generalized match relation is satisfied between an input register and characteristic data contained in each processing elements. An example for such a computer is STARAN IV.

In the following, we give a short description of the above computer. The STARAN computer contains a large number of processors working simultaneously under a single control. Figure 1.9 illustrates the basic architecture of STARAN which consists of traditionally addressed control memory for program storage and data buffering, a control unit for sequencing and decoding instructions from control memory. Also there are up to 32 array modules. Figure 1.10 shows a simplified block diagram of an array module. Each array module contains 256 simple processing elements. Each processing element is

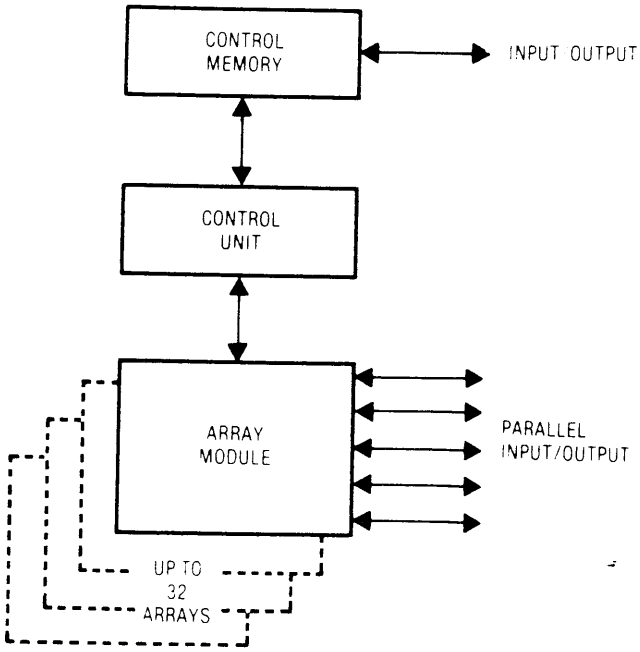


Figure 1.9 - STARAN modularity (Rohrbacher and Potter [1977])

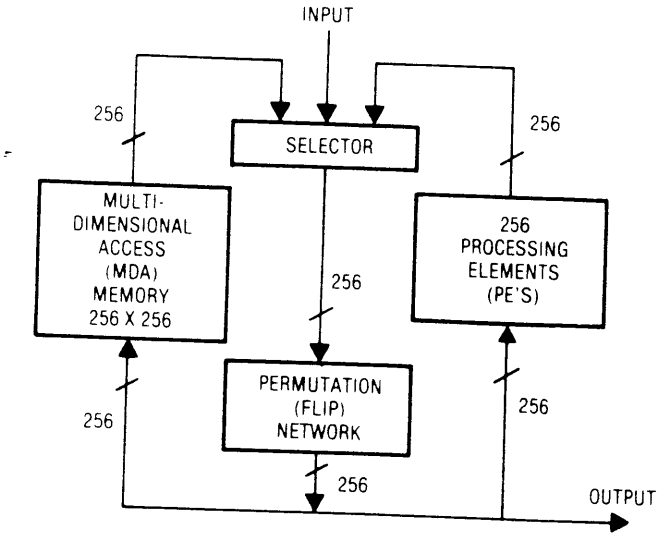


Figure 1.10 - An Array Module (Rohrbacher and Potter [1977])

given enough logic and storage to perform bit-serial searches and arithmetic functions. The processing element communicates with a multidimensional access memory through a "flip" network which can permute a set of operands to allow interprocessing element communication.

The multidimensional access memory contains a square array of bits, 256 words with 256 bits in each word.

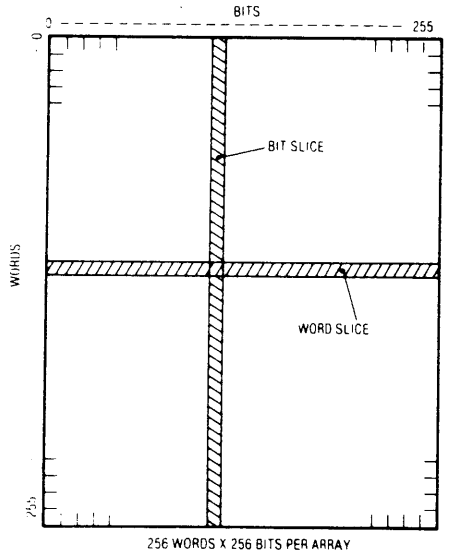


Figure 1.11 - Two access modes. (Rohrbacher and Potter [1977])

The bit slice access mode, which is illustrated in Figure 1.11, is used in the associative operation to access one bit in all words in parallel, while the word access mode is used in input/output operations to access all bits of one word in parallel. The processing elements must be able to access more than one word in the memory. The permutation network is capable of permutating the 256-bit vector read from the multidimensional access memory in a wide variety of ways. More about the design may be found in [35].

1.4 MIMD Processor

MIMD computers consists of n independent processors P_1, P_2,..., P_n each having its own control unit. These processors may share, among other things, a number of input and output units and a main store. At every instant the processor can carry out different instructions in parallel. Examples are Sequent Balance 8000, Transputer and Hep.

Sequent Balance is available in the Statistics and Computational Mathematics Department at the University of Liverpool, and all the experimental work in this thesis is carried out using this computer.

In [4], it is mentioned that the Sequent Balance 8000 has a large global memory accessible in unit time to all processors, data (eg. Large arrays) may then be stored in this memory and supposed to be available to all processors.

Transputer array has no shared memory. Each processor has its own local memory, the communications between the processors must be through message passing.

We give here a short brief description of the architectures of the Sequent Balance 8000, Transputer and Hep.

1) The Sequent Balance 8000

The Sequent Balance 8000 first computer was delivered in 1984 [31], and manufactured by Sequent Balance Computer Inc, Beaverton, Oregon, United States of America. Sequent Balance offers general purpose computing which is suitable for both academic and industrial research.

The Sequent Balance 8000 system, as in Figure 1.12, is a true multiprocessor not an array processor or an ensemble computer, and can contain two to twelve 32-bit processors, cover a performance up to 8.4 MIPs and support up to 96 users.

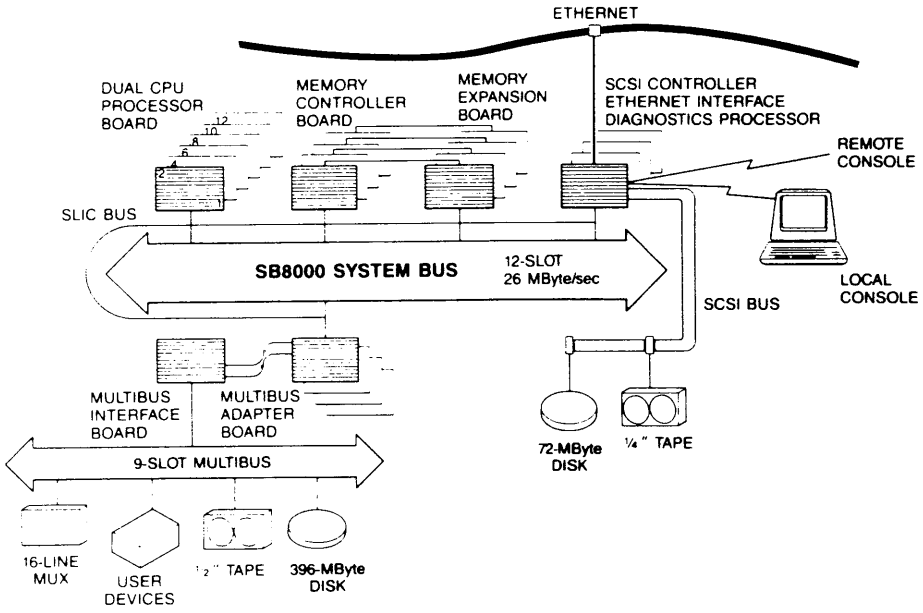


Figure 1.12 - Architecture of the Sequent Balance 8000. (Oseterhauge, A)

All the processors share a single common 28 Mbytes. They also contain a single copy of DYNIX operating system supports 4.2 bsd and system V application in the same time. In addition, all processors automatically schedule themselves to ensure that all of them are kept busy as long as there are executable processes available. Each processor has 8 kbytes of local RAM and 8 kbytes of small and fast memory units, both of which greatly reduce the number of times the processor must access system memory. For secondary mass storage, the Balance system offers both disks and tape drives. Balance

systems are available with $5\frac{1}{4}$ inch, 8 inch or 10 inch disks and can be configured with up to 32 disks of any one type. Also the system contain up to reel-to-reel tape drives. Both 6250-bpi and 1600-bpi drives are available. On the Balance, parallel programs execute simultaneously. At any instant, the system can be executing multiple instructions from multiple processors.

Sequent's parallel programming library provides a powerful set of services to support parallel programming in C, Fortran, pascal and Ada, where the Balance is the first available system to offer Ada Language facilities in a truly parallel environment.

2) **The Transputer**

The transputer is a micro-supercomputer. It is a product of INMOS, designed and manufactured in the United Kingdom, for building high performance computer systems. The transputer is one of the most powerful processors available.

Transputers are the first range of VLSI (Very Large Scale Integration) device [32]. The development of VLSI has made it possible to contain wide amounts of memory and processing ability on a single wafer of silicon. All the transputers use local memory and point-to-point communication links, figure 1.13, which allow networks of any size and topology to be created.

The first transputer available was T414, a 32-bit microprocessor with throughout of 10 MIPs (Million instruction per second). The T414 has 2 kilobytes of on-chip RAM and four INMOS serial links. Also the transputer can address up to 4 gigabytes of memory over a 32-bit local memory. The T800 is

compatible with T414 but includes floating point hardware and 4k of internal RAM.

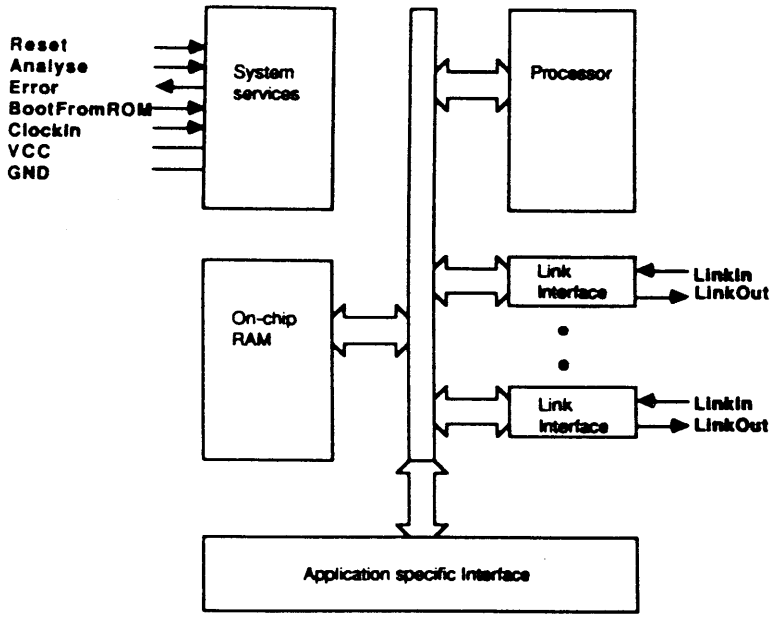


Figure 1.13 - Transputer Architecture (Packer, J.)

Occam is the transputer language, since the transputer arrived in 1985. Also transputers are designed to be programmed in high level language such as Fortran, Ada, C and Pascal.

3) The Hep

The Hep was manufactured by Denelcor of Denver, Colorado. It was designed to contain up to 16 processors execution modules (PEM's), up to 128 data memory modules, connected by a high-speed switching network. The architecture of a four processor Hep [34] is shown in Figure 1.14. Every PEM has its own local program memory, constant memory and 2048 general registers and 4096 special registers. Each processor can execute ten

million instructions per second, it achieves this speed only when at least 8 processors are being executed, and a single process can never execute faster than 1.25 MIPs.

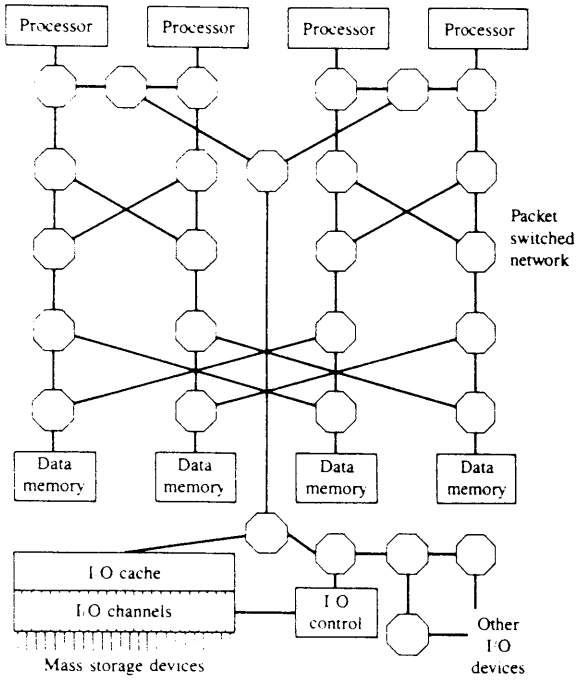


Figure 1.14 - Architecture of a four-processor Denelcor HEP. (M.J. Quinn [1988])

1.5 Processor Organizations

In this section, we shall define five important processor organizations of connecting processors in a parallel computer.

1) **Mesh**

In a mesh network, the nodes are arranged into a q-dimensional lattice. Communication is allowed only between neighbouring nodes, hence the interior nodes communicate with 2q other processors. Figure 1.15a illustrates a two-dimensional mesh. Some variants of the mesh model allow wrap around connections between processors on the edge of the mesh. These connections may connect processors in the same row or column (Figure 1.15b), or they may toroidal (Figure 1.15c).

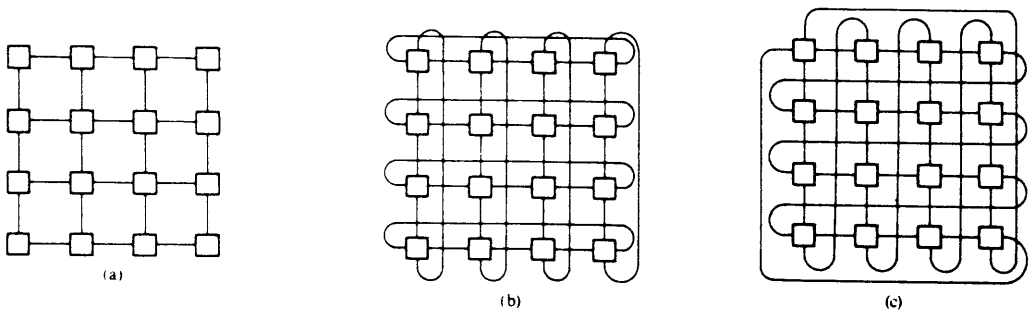


Figure 1.15 - Two dimensional meshes. (M.J. Quinn [1988])
This connectivity is popular on transputer arrays.

2) **Pyramid**

A pyramid network is used to organize processors. A pyramid of size 16 which is illustrated in figure 1.16 has, at its base, a two-dimensional mesh network containing 16 processors. The levels of the pyramid are numbered as follows: the base has level number 0, the processor at the apex of the pyramid has level number 2. In general, the total number of processors in a pyramid of size p is $(4^{p+1}-1)/3$.

A machine with this type of connectivity is believed to be under development at Warwick. The nodes consist of transputer tied to a small DAP.

3) **Shuffle - Exchange Network**

A shuffle-exchange network consists of $n = 2^k$ nodes, numbered 0, 1, ..., n-1, and two kinds of connections, called shuffle and exchange. Exchange connections link pairs of nodes whose numbers differ in their last significant bit. The perfect shuffle connection links node i with node 2i modules n-1, with the exception that node n-1 is connected to itself. See figure 1.17 for a drawing of an eight-node shuffle exchange network. Shuffle connections are indicated by solid arrows, dashed arrows represent exchange links. To see the meaning of perfect shuffle, consider shuffling a deck of eight cards, numbered 0,1,2,3,4,5,6,7. If the deck is divided into two exact halves and shuffled perfectly, then the result is the following order: 0,4,1,5,2,6,3,7. The final position of the card that began at index i can be determined by following the shuffle link from node i.

4) **Butterfly**

A butterfly network consists of $(k+1)2^k$ nodes divided into k + 1 rows, or ranks, containing $n=2^k$ nodes each (Figure 1.18). The ranks are labelled 0 through k, although the ranks 0 and k are sometimes identified, giving each node four connections to other nodes.

Let node (l,m) refer to the mth node on the lth rank, where $0 \leq l \leq k$ and $0 \leq m < n$. Then node (l,m) on rank $l > 0$ is connected to two nodes on rank l-1; node (l-1,m) and node (l-1,r), where r is the integer found by inverting the lth most significant bit in the binary representation of m. The Bolt, Beranek and Newman Butterfly [34] is an example of a machine using this connectivity.

5) **Hypercube**

A hypercube network is a butterfly with its columns collapsed into single nodes. The network consists of $n=2^k$ nodes forming a k-dimensional hypercube. The nodes are labelled 0,1, ..., 2^k-1; two nodes are adjacent if their labels differ in exactly one bit position. Figure 1.19 illustrates a four dimensional hypercube. The hypercube has become popular in recent years; the Intel iPSC is an example of a machine using this connecting. More details can be found in [34].

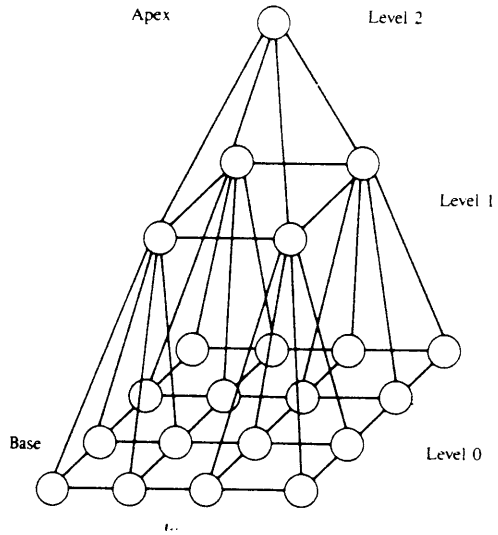


Figure 1.6 - A pyramid network of size 16 (M.J. Quinn [1988])

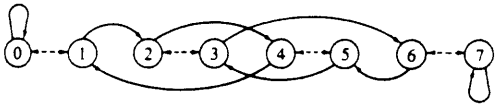


Figure 1.17 - Shuffle exchange network with eight nodes (M.J. Quinn [1988])

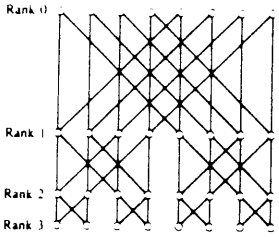


Figure 1.18 - A Butterfly network with 32 nodes (M.J. Quinn [1988])

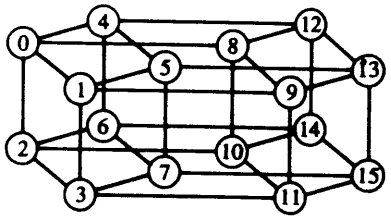


Figure 1.19 - Four-dimensional (16 nodes) hypercube M.J. Quinn [1988])

1.6 Complexity

In this section we shall take a brief look at questions of complexity in the context of numerical methodology. Two important measures of the quality of parallel algorithms implemented on multiprocessors are speedup and efficiency.

The speedup of parallel algorithms is defined by

$$S_p = T_1/T_p$$

where t_1 is the execution time of the algorithm using a single processor, T_p is the execution time of the algorithm using p processors. One hopes that $S_p > 1$, the corresponding efficiency is given by

$$E_p = S_p/p$$

where E_p measures the utilisation of parallel computers. One expects $E_p < 1$ rather that $S_p < 1$, but there are cases where because the algorithm takes different paths when run on more than one processor, we find $E_p > 1$.

We shall give an example to illustrate the concepts of the speedup and the efficiency. Suppose we wish to sum $A = \sum_{i=1}^{20} a_i$. Using a single processor requires 19 additions. Let the number of additions be used as a measure of the time needed. Then $T_1 = 19$ time units. If $p = 2$, then we have two sums

$$b_1 = a_1 + a_2 + \ldots + a_{10} \quad \text{and} \quad b_2 = a_{11} + a_{12} + \ldots + a_{20}$$

which can be calculated simultaneously, and then form $c_1 = b_1 + b_2$. Thus A can be obtained in $T_2 = 10$ time units. When $p = 10$, the time required is $T_{10} = 5$ time units.

The following table measures the speedups and the efficiencies:

Table 1.6.1

Number of Processors p	Time Units T_p	Speedup S_p	Efficiency E_p
1	19	1	1
2	10	1.9	0.95
4	6	3.17	0.79
10	5	3.8	0.38

We should take account of the communication costs in evaluating the complexity of a parallel algorithm. Sometimes the communication cost is too much compared with the computational cost. In other words, more time is spent in sending the data among processors than in carrying out useful work.

In [34] can be found useful notes about communication and synchronization. It is noted that the communication is achieved either through shared variables or through message passing. The type of the communication available may be dictated by architectures. For example, in multiprocessors which do not have shared memory, the communication is achieved through message passing. The message passing is a form of communication, since a processor receiving a message is receiving values from another process. The message passing is also a form of synchronization, since a message can be received only after it has been sent.

In the next section we shall give the important and typical case of time and processor bounds in the evaluation of arithmetic expressions.

1.7. The Use of Parallelism for Evaluating Arithmetic Expression

In [21], [22] some examples of arithmetic expression evaluations are given. For any arithmetic expression, how much time reduction can be acheived? Can general bounds and algorithms for the achievable speedups? How many processors are needed? To answer these questions, we will give some results concerning the evaluation of expression.

Definition 1.1

An arithmetic expression is any well-formed string consisting of at least one of four operations (+,-,*,/), left and right brackets and atoms which are constants or variables. An arithmetic expression A of n different atoms will be denoted by $A < n >$.

The time required to evaluate $A < n >$ in one processor is n-1 time units. With any number of processors $A < n >$ can be evaluated in $\log_2 n$ time steps. Denote $\lceil x \rceil$ is the smallest integer greater than or equal to x. The following lemma provides the bound on the time required to compute a given expression.

Lemma 1.1

Given any arithmetic expression $A < n >$ we have

$$T(A < n >) \geq \lceil \log_2 n \rceil$$

On the other hand, the expression whose evaluation requires O(n) time units independent of how many processors are available can be constructed.

Consider, for example, the Horner scheme for the evaluation of the polynomial of degree n

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + ,\ldots , + a_n x^n$$
$$= a_0 + x(a_1 + x(a_2 + ,\ldots , + x(a_{n-1} + xa_n) \ldots).$$

It requires O(n) operations. The availability of more than a single processor has, therefore, no power to speed up the calculation. However, by using associativity, commutativity and distributivity we may be able to evaluate the above expression more quickly.

EXAMPLE 1

Consider the following expression

$$A < 4 > = (((a+b) + c) + d) \quad , \quad a,b,c,d \; \varepsilon \; R$$

Figure 1.20 shows the parse tree. This tree requires three steps for its equation

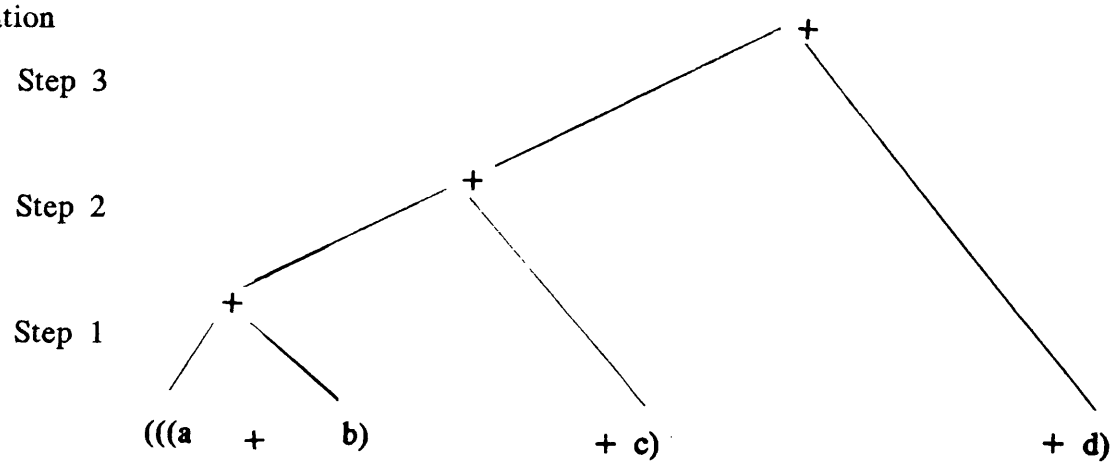


Figure 1.20

with $T_1 = 3$. We refer to this as a tree height of three. However, by using the associative property of addition, the expression can be written as

$$A < 4 > = (a+b) + (c+d)$$

which can be evaluated as in the following figure

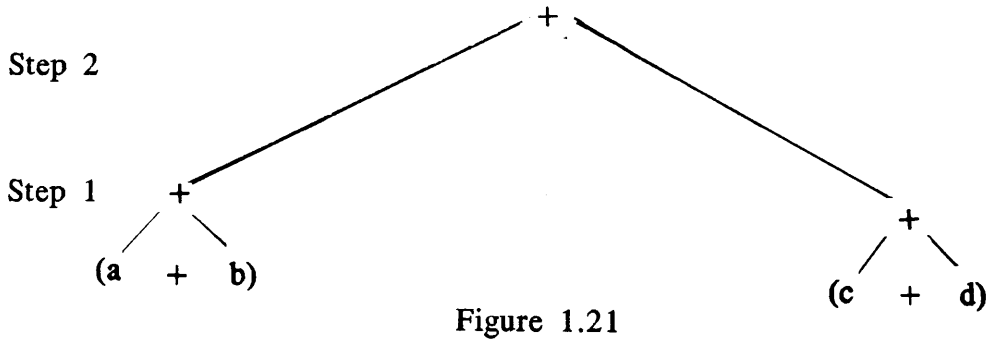


Figure 1.21

with $T_2 = 2$, $S_2 = T_1/T_2 = 3/2$, $E_2 = 3/4$.

EXAMPLE 2

Consider the expression

$$A < 4 > = a + bc + d \quad , \quad a,b,c,d \; \varepsilon \; R$$

we have a tree of height three. In this case the tree is not unique, but no lower height tree for the expression can be found by using associativity.

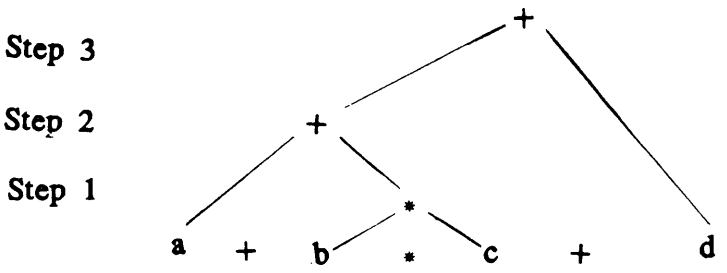


Figure 1.22

The time required $T_1 = 3$. However, by using the commutative property of addition we have

$$A < 4 > = a + d + bc \; ,$$

and the tree of Figure 1.23, whose height is only two

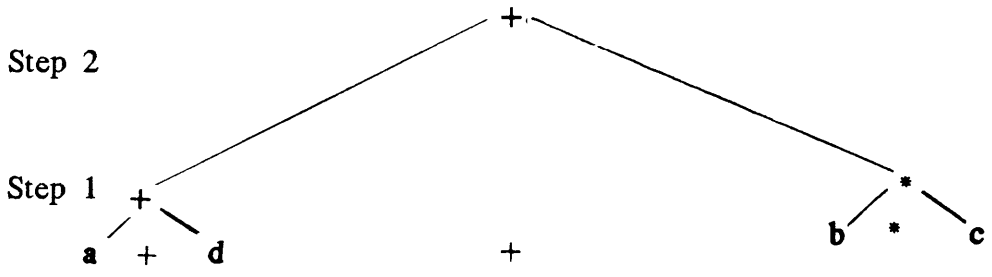


Figure 1.23

with $T_2 = 2$, $S_2 = 3/2$, $E_2 = 3/4$.

An upper bound on the reduced tree height, assuming that only associativity

and commutativity are used, can be given.

In [20] the following theorem was given:-

THEOREM

Let $A < n \mid d >$ be an arithmetic expression with a depth d of parenthesis nesting. By the use of associativity and commutativity only, $A < n \mid d >$ can be transformed in such a way that

$T_p(A < n \mid d >) \leq \lceil \log_2 n \rceil + 2d + 1$,

with $p \leq \lceil n/2 - d \rceil$.

Note: if the depth of parenthesis nesting d is small, then this bound is quite close to lower bound of $\lceil \log_2 n \rceil$. More details can be found in [22].

1.8 The Algorithm Must Fit the Architecture

The performance of an algorithm depends on the architecture and the language [1]. Sometimes this is due to communication overhead, and sometimes other factors are involved. Let us define the concept of the granularity which is the size of the pieces into which the problem is divided into equivalent numbers of instructions. Then, usually, algorithms that perform relatively few operations between synchronization, for example, have small granularity, give poor efficiency on MIMD computers. This yields to the range of the granularity which depends also on the application.

1.9 Parallel Processing in Ada

Ada language has been used in this thesis in order to implement our algorithms in parallel processing. Here we shall give some background on Ada and its facilities.

In the mid 1970's the American Department of Defence (D.O.D.) became worried at the amount of money spent on computing [5]. Investigation was carried out to find where the money was spent. They found too much money was spent on software. More than one hundred languages were in use. They observed that the huge number of languages was itself a problem, then they decided to put forward a new programming language which has facilities for parallel programming.

The D.O.D. set up a working group to study programming languages. This group included representatives from the Army, Navy and Air Force and had members from U.K., France and West Germany. It was decided that the language should be called Ada, in honour of Ada Byron, Countess of Lovelace, who assisted Charles Babbage on his analytic engine in the nineteenth century. Ada is now seen as an international language, not just an American idea, and it is almost certain that Ada will become the word standard language for embedded computer system programming.

Now we shall discuss Ada's facilities for parallelism which are given in [1] and [5].

1) **Ada Tasks**

In Ada, parallel activities are described using tasks. A task is a program unit that can be executed simultaneously with other program units.

The task consists of specifications such as

```
Task T is
- - - - - -
End T;
```

and the body which is specified seperately as follows

```
Task body T is
---------------
End T;
```

Consider, for example, a family consisting of three members, mother, father and son going shopping. They need meat, salad and bread whose purchase is to be represented by procedures buy-meat, buy-salad and buy-bread. If each member of the family gets just one item and meet together in the car park, then the parallel solution is as follows:

```
Procedure Shopping is
    Task get-salad;
    Task body get-salad is
    Begin
    Buy-salad;
    End get-salad;
    Task get-bread;
    Task body get-bread is
    Begin
    Buy-bread;
    End get-bread;
    Begin
    Buy-meat;
End Shopping;
```

The mother is the main processor and calls by-meat directly. The father and the son are subservient processors and perform locally declared tasks get-salad and get-bread. The sequence of the actions in the above example are as follows:

First, the tasks get-salad and get-bread are declared to be part of main task, then they all start working in parallel when the moment that begins the main task is reached. The two tasks continue to execute at the same time until terminated. The main task does not terminate until the dependent tasks terminate.

2) **The Rendezvous**

Two tasks are said to rendezvous when the first task calls an entry declared in the second. An entry is declared in the task specification:

```
Task T is
Entry E(...);
End;
```

The declaration of the entry is like procedure specification. It is called in the similar way to a procedure

```
T . E (...);
```

In the body of the T task the entry point appears in an accepted statement and looks like the following:

```
Accept E(...)do
- - - sequence of statements
End E;
```

When a task accepts an entry it executes a sequence of code on behalf of the calling task. This is sometimes referred to as remote procedure calling. There may be a time delay between the call of an entry and its acceptance. When one task calls the entry, the task containing the entry executes it. The accept statement cannot be executed until a task calls the entry and the entry task containing it reaches the entry. One of these will happen first and the task concerned is suspended until the other catches up. When the entry is completed, the rendezvous is complete and the tasks once again continue in parallel.

Return again to the shopping procedure. Suppose the father is not there and they wanted only bread and meat. The mother gives her son the money and buy-bread and buy-meat are replaced by functions that take in money and return the item wanted. Then the example would be:

```
Procedure Shopping is
    Task get-bread is
    Entry pay (M:in money);
    Entry collect (S:out bread);
    End get-bread;
    Task body get-bread is
    Cash : money
    Eat : bread
    Begin
    Accept pay (M: in money) do
    Cash : = M;
```

```
        End pay;
        Eat : = buy-bread (cash);
        Accept collect (S: out bread) do
        S : = eat;
        End collect;
        End get-bread;
```

The main task is as follows:

```
    Begin
        Get-bread . pay (100);
        MM : = buy-meat (200);
        Get-bread . collect (BB);
    End;
```

The above example shows that the meat and bread end up in variables MM and BB whose declaration would appear somewhere in the program. When get-bread becomes active, it meets an accept statement and waits until the main task calls the entry pay. After calling the function buy-meat, the main task calls the collect entries.

3) **Task Types**

If we find ourselves writing the same task structure for several tasks, then we should use a task type. A task type is specified by the form

```
        Task type T is
        Entry E;
        End;
```

The task body is as before, we treat the task type as a template. For example, if the number of T tasks are N then we could have an array of tasks as follows:

S: array (1..N) of T;

introduce a new task referred to as S, and S(i), as an example, refers to the ith task.

4) The Select Statement

The select statement allows a task to select from one of several rendezvous. Suppose we have within a task

```
loop
    select
        Accept read (X : out item) do
        X : = V;
    End
or
        Accept write (X : in item) do
        V : = X;
        End;
    End select;
    End loop;
```

The select statement begins with reserves word select and finishes with end select. In Ada, systems will make an arbitrary choice when deciding which one, read or write, is to be executed first. In this example there are three

possibilities:

a) If neither read nor write has been called then the task is suspended, one or other is called and then the corresponding accept statement is obeyed.

b) If calls of both read or write are queued then only one of them is chosen

c) If one or more calls of read are queued but there are no queued calls of write, then the first call of read is accepted and vice versa with the roles of read and write reversed.

5) Shared Variables

Shared variables are variables that are used in more than one task. It is important to note that when using shared variables, sometimes these variables are changing in two different tasks at the same time.

CHAPTER 2

A Survey of Parallel Methods in Numerical Linear Algebra

2.1 Introduction

In this chapter we shall review some work done on parallel solution of linear systems of the form

$$A \underline{x} = \underline{b} \qquad (2.1.1)$$

where A is n x n matrix, $\underline{x}$ and $\underline{b}$ are vectors of order n. Direct techniques are considered to solve (2.1.1). Gauss Elimination method, Gauss-Jordan method, Cholesky method and WZ factorization, are particularly suited to parallel computation. We then describe column sweep algorithm, a recurrent product algorithm and two algorithms for solving triangular systems of equations. Also, we describe some algorithms for solving tridiagonal systems.

Most sequential techniques reduce the system of equations to a trianglar form which can easily be solved for unknowns, using backward substitution. Solving the linear system, using Gauss elimination method, for example, or its variant LU_decomposition, requires $O(2n^3/6)$ multiplications and divisions, whereas Gauss-Jordan requires $O(3n^3/6)$ multiplications and divisions. Throughout this chapter, we give the speedup and the corresponding efficiency of most parallel algorithms which are mentioned above as much as we can. Note that the communication cost of these algorithms is ignored by the literature. However, the cost in our algorithms, which are given later in Chapter 4, has been calculated.

2.2 Gauss Elimination Method

A linear system (2.1.1) can be solved in three steps:-

a) Decompose A into the product of lower triangular matrix L and upper triangular matrix U,

b) Solve L $\underline{y}$ = $\underline{b}$ and,

c) Solve U $\underline{x}$ = $\underline{y}$

The first step has the highest computational cost. We shall first describe a parallel algorithm developed by [26].

The description that follows in [26], and is idealised, is not taking any account of communication costs.

Consider the sequential algorithm for perfoming an LU-decomposition of n x n dense matrix A by using pivoting.

LU_decomposition

For k ⟵ 1 to n-1 do

Find l such that |a(l,k)| = max (|a(k,k)|,... ,|a(n,k)|)

piv (k) ⟵ l {l is the pivot row}

a(piv(k),k) ⟷ a(k,k)

c ⟵ 1/a(k,k)

For i ⟵ k+1 to n do

a(i,k) ⟵ a(i,k) * c (elements of L)

Endfor

For j ⟵ k+1 to n do

a(piv(k),j) ⟷ a(k,j)

For i ⟵ k+1 to n do

a(i,j) ⟵ a(i,j) - a(i,k) * a(k,j)

Endfor

Endfor

Endfor

Define a task T_k^j to be a portion of the algorithm that computes a particular column of j for a given value k. The set of tasks is given by

$J = \{ T_k^j \mid 1 \le k \le j \le n;\ k \le n-1 \}$.

Noted that task T_k^j must be performed before task T_m^l if either

a) $j < l$ and $k = m$ or

b) $k < m$.

The range and domain of these tasks are:

$R(T_k^j) = \{a(i,j) \mid k \le i \le n\}$

$D(T_k^j) = \{a(i,j) \mid k \le i \le n\} \cup \{a(i,k) \mid k \le i \le n\}$ respectively.

We can observe that, for example,

$\{T_k^{k+1}, T_k^{k+2}, \ldots, T_k^n\}$ are independent of each other and can be executed in parallel.

Assume that a time step consists of either a multiply and a substract or a multiply and a compare. Neglecting any overhead for loop control, we notice the execution time $W(T_k^j)$ for each task is :

$$W(T_k^j) = \begin{cases} n+1-k & \text{if } k = j \\ n-k & \text{if } k < j \end{cases}$$

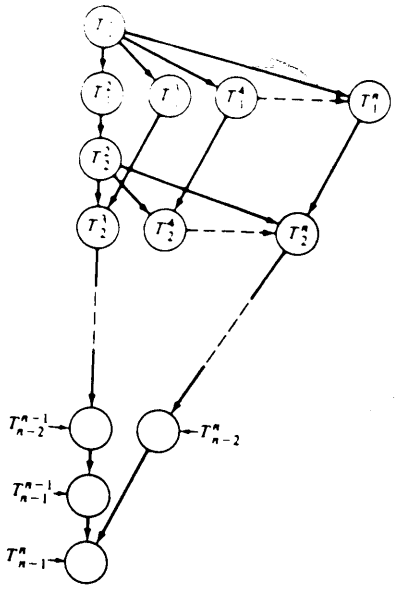


Figure 2.1 is a maximally parallel graph that upholds all the precedence relations. (Lord, Kowalik and Kumar [1983])

The longest path in the graph traverses the nodes

$$T_1^1 , T_1^2 , T_2^2 , T_2^3 , T_3^3 , \ldots , T_{n-1}^{n-1} , T_{n-1}^{n} .$$

The length of path is

$$n + 1 + 2 \sum_{j=2}^{n-1} j = n^2 - 1.$$

This path is so long that it is easy to come up with a schedule that assigns remaining tasks to $\lceil n/2 \rceil - 1$ processors so no tasks finish after task T_{n-1}^{n}. The first processor is assigned the tasks along the critical path. The second processor will execute the tasks

$$T_1^3, T_1^4 , T_2^4 , T_2^5 , T_3^5 , \ldots , T_{n-2}^{n} .$$

The third processor will execute the tasks

$$T_1^5 , T_1^6 , T_2^6 , T_2^7 , T_3^7 , \ldots , T_{n-4}^{n}$$

and the jth processor will execute the tasks

$$T_1^{2j-1} , T_1^{2j} , T_2^{2j} , T_2^{2j+1} , \ldots , T_{n-2(j-1)}^{n} .$$

Note that these tasks are not paths through the graph. When n is even, the following figure illustrates the schedule.

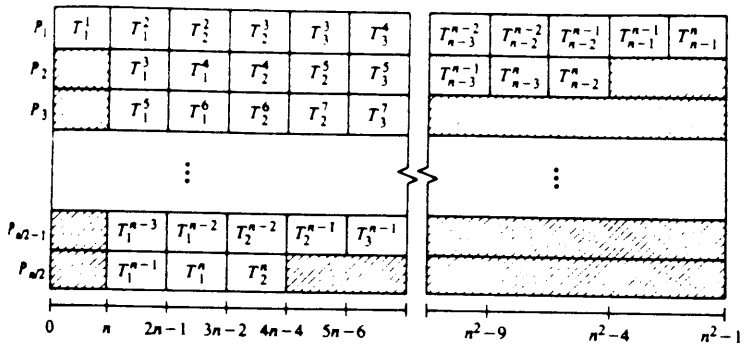


Figure 2.2 - Schedule using n/2 processors (n even).
(Lord, Kowalik and Kumar [1983])

The sequential LU_decomposition algorithm requires $n^3/3 + O(n^2)$ time unit to execute. Since the schedule in Figure 2.2 has length n^2-1, then this schedule is optimal for n/2 processors. The parallel algorithm requires $(n^2-1)(n/2)/p$ time units to execute. Hence, the efficiency of this algorithm using p processors, is

$$E = \frac{n^3/3 + O(n^2)}{(n^2 - 1)n/2}$$ which is equal to 2/3 as $n \longrightarrow \infty$.

The following table gives the actual and the predicted efficiency together where the program is run with $n \leq 35$ and $1 \leq p \leq 8$, using the Hep parallel computer which is described in Chapter 1.

Table 2.1

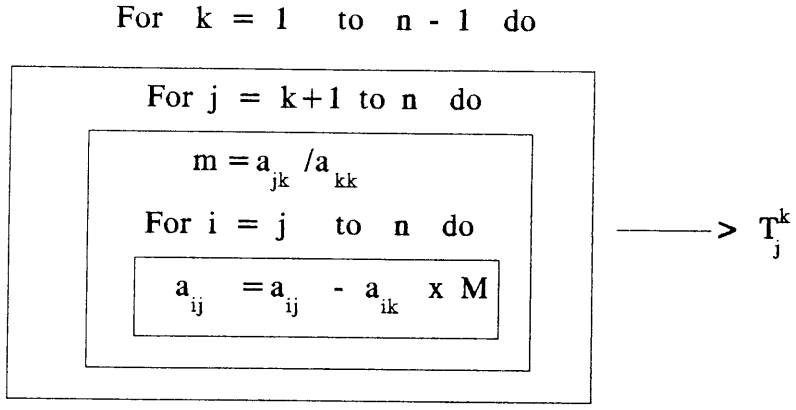
(Lord, Kowalik and Kumar [1983])

n	Number of Processors, P 2	3	4	5	6	7	8	Actual or Predicted
10	0.833	0.719	0.642	0.633				A
	0.852	0.739	0.678	0.685				P
15	0.888	0.794	0.740	0.651	0.618	0.625	0.581	A
	0.900	0.815	0.766	0.679	0.652	0.681	0.633	P
20	0.921	0.843	0.774	0.758	0.670	0.623	0.605	A
	0.931	0.863	0.798	0.789	0.703	0.656	0.640	P
25	0.934	0.878	0.830	0.763	0.755	0.692	0.642	A
	0.944	0.896	0.855	0.739	0.788	0.726	0.675	P
30	0.942	0.892	0.844	0.818	0.757	0.744	0.710	A
	0.949	0.911	0.863	0.843	0.783	0.777	0.745	P
35	0.948	0.901	0.862	0.819	0.790	0.747	0.741	A
	0.956	0.918	0.880	0.843	0.827	0.779	0.769	P

2.3 A Parallel Algorithm for Positive Definite Symmetric Matrix

In [23], a parallel decomposition by elimination method for solving (2.1.1), where A is a real, symmetric and positive definite matrix, is described.

The serial version of the algorithm is just Gauss elimination:



This algorithm produces the factor LD of $A = (LD)L^T$ by Gaussian elimination method, where L is a unit lower triangular and D is diagonal matrix with positive element.

Define the task T_k^j as the code segment

$$T_k^j = \begin{cases} \{m = a_{jk}/a_{kk} \\ \{ \\ \{a_{ij} = a_{ij} - m \times a_{ik} \quad , i = j , j+1 , \ldots , n. \end{cases}$$

The total work t_1 of the above algorithm or uniprocessor computer is

$$t_1 = \sum_{k=1}^{n-1} \sum_{j=k+1}^{n} W(T_j^k) = n^3/3 + n^2/2 - 5n/6 .$$

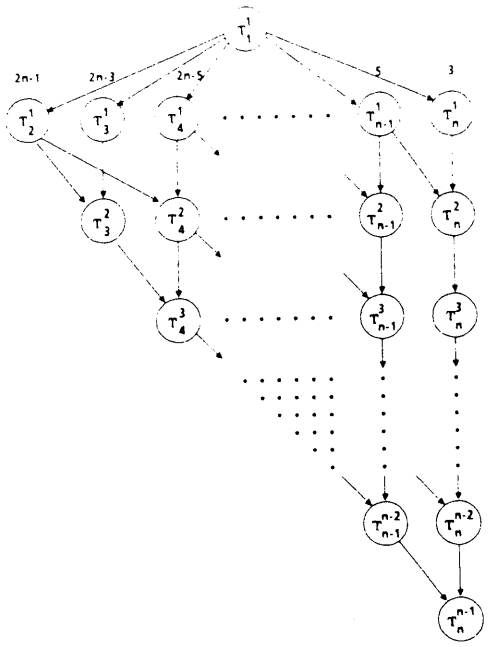


Figure 2.3 - Procedence graph G′ of the maximally system C′ (Kumar and Kowalik [1986])

The longest path LP is

$$LP = (T_2^1 , T_3^2 , T_4^3 , \ldots , T_n^{n-1})$$

and has the minimum execution time

$$W(LP) = \sum_{k=1}^{n-1} W(T_{n+1}^{k}) = n^2 - 1.$$

The maximum number of processors which can be used in parallel is n-1. Then the speedup is given by

$$S_p = \frac{n^3/3 + n^2/2 - 5n/6}{n^2 - 1} \approx \frac{n}{3}$$

for larger n when compared with Gauss elimination.

The corresponding efficiency is

$E_p = S_p/p \approx 0.3333$ for larger n.

2.4 Gauss - Jordan Method

This method can also be used to solve linear system (2.1.1). Suppose we wish to solve a linear system of 4 x 4, the four equations are:

$$\begin{aligned}
E_1&: \; x_1 + 3x_2 + 7x_3 + x_4 = 32\\
E_2&: \; x_1 + 4x_2 + 9x_3 + 2x_4 = 44\\
E_3&: \; 2x_1 + 8x_2 + 19x_3 + 3x_4 = 87\\
E_4&: \; 3x_1 + 13x_2 + 33x_3 + 4x_4 = 144
\end{aligned}$$

which will be solved for unknowns x_1, x_2, x_3 and x_4. The first step is to use equation E_1 to eliminate the unknown x_1 from E_2, E_3 and E_4 by performing $(E_2 - E_1) \longrightarrow (E_2)$, $(E_3 - 2E_1) \longrightarrow E_3$ and $(E_4 - 3E_1) \longrightarrow E_4$. The resulting system is:

$$\begin{aligned}
E_1&: \; x_1 + 3x_2 + 7x_3 + x_4 = 32\\
E_2&: \; x_2 + 2x_3 + x_4 = 12\\
E_3&: \; 2x_2 + 5x_3 + x_4 = 23\\
E_4&: \; 4x_2 + 12x_3 + x_4 = 48
\end{aligned}$$

In the new system, E_2 is used to eliminate x_2 from E_3 and E_4 by the operations $(E_3 - 2E_2) \longrightarrow (E_3)$ and $(E_4 - 4E_2) \longrightarrow (E_4)$ to give

$$\begin{aligned}
E_1&: \; x_1 + 3x_2 + 7x_3 + x_4 = 32\\
E_2&: \; x_2 + 2x_3 + x_4 = 12\\
E_3&: \; + x_3 - x_4 = -1\\
E_4&: \; + 4x_3 - 3x_4 = 0
\end{aligned}$$

In the Gauss-Jordan method we eliminate the elements above the diagonal as well as below. Perform the following step $(E_1\text{-}3E_2)\longrightarrow(E_1)$ which eliminates x_2 from E_1 to give:

$$\begin{aligned} E_1&: x_1 + x_3 - 2x_4 = -4 \\ E_2&: x_2 + 2x_3 + x_4 = 12 \\ E_3&: x_3 - x_4 = -1 \\ E_4&: 4x_3 - 3x_4 = 0 \end{aligned}$$

Next perform the following operations $(E_4\text{-}4E_3)\longrightarrow(E_4)$, $(E_2\text{-}2E_3)\longrightarrow(E_2)$ and $(E_1 - E_3) \longrightarrow (E_1)$ to give:

$$\begin{aligned} E_1&: x_1 - x_4 = -3 \\ E_2&: x_2 + 3x_4 = 14 \\ E_3&: x_3 - x_4 = -1 \\ E_4&: x_4 = 4 \end{aligned}$$

Finally, perform the following operations $(E_1 + E_4) \longrightarrow (E_1)$, $(E_3 + E_4) \longrightarrow (E_3)$ and $(E_2 - 3E_4) \longrightarrow (E_2)$ to give:

$$\begin{aligned} E_1&: x_1 = 1 \\ E_2&: x_2 = 2 \\ E_3&: x_3 = 3 \\ E_4&: x_4 = 4 \end{aligned}$$

It should be noted that backsubstitution is not needed. In [27], a computational network of Gauss-Jordan algorithm is presented. In the following algorithm, no pivoting is used and each diagonal element in the

matrix A is used to eliminate all the elements in the corresponding column. The right-hand side $\underline{b}$ is considered as column n+1 of A and upon completion of the elimination, the solution $\underline{x}$ is stored in the last column. The algorithm is as follows:

For i = 1,... , n do

$$a_{ij} = a_{ij}/a_{ii} \quad , \quad j = i + 1,\ldots , n + 1$$

For k = 1,... , i - 1 and k = i + 1,... , n do

$$a_{kj} = a_{kj} - a_{ki} * a_{ij} \quad , \quad j = i + 1,\ldots , n + 1$$

Assuming that the matrix A is dense, we denote the operations for the multipliers n - i + 1 by D_i and U_{ik} the n - i + 1 operations for the update of row k caused by contributions from row i during the elimination of a_{ki}.

The network used in [27], consists of n stages, s = 1,... , n each consisting of n - s + 2 cells. The network is shown in the following figure where each cell is labelled by a pair (r,s), indicating its position with respect to the shown axes.

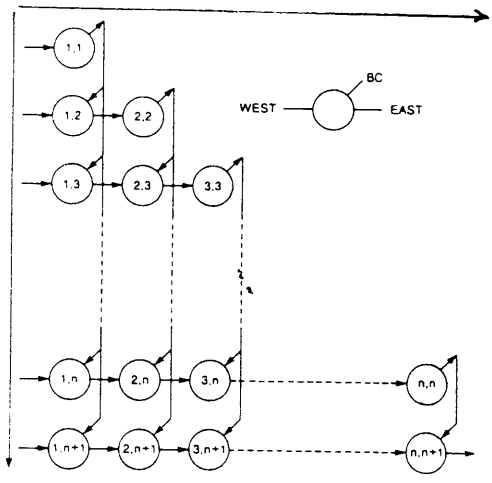


Figure 2.4 - The computational network. (R. Melhem [1987])

Assuming that any diagonal cell (s,s) may broadcast data to the other cells in the same stage. However, it should be clear that this broadcast may be easily replaced by local vertical interconnections, if the input one is skewed properly.

The element of a row i of the matrix A enters the network at cycle i and crosses stages s = 1,2,... , i-1, where at each stage s the operation $U_{s,i}$ takes place. When row i arrives at stage i (at cycle 2i-1), cell (i,i) broadcasts a_{ii} to the other cells and the operation D_i is executed.

The updated elements of row i are then stored in stage i for n-1 cycles during which rows k = i + 1,... , n followed by rows k = 1, 2,... , i-1 cross stage. At the end of cycles 2i+n-1, row i comes out of stage n after 2i+n-1 cycles with $a_{ii} = 1$, $a_{ij} = 0$ for j = 1,2,... , n, $j \neq i$ and $a_{i,n+1} = x_i$. In other words, the outputs $x_1,...,x_n$ are produced at cycles 2n,... , 3n-1.

The network which is described in [27], uses n(n+3)/2 cells and the network completes execution in 3n-1 cycles. There are no timings produced in [27].

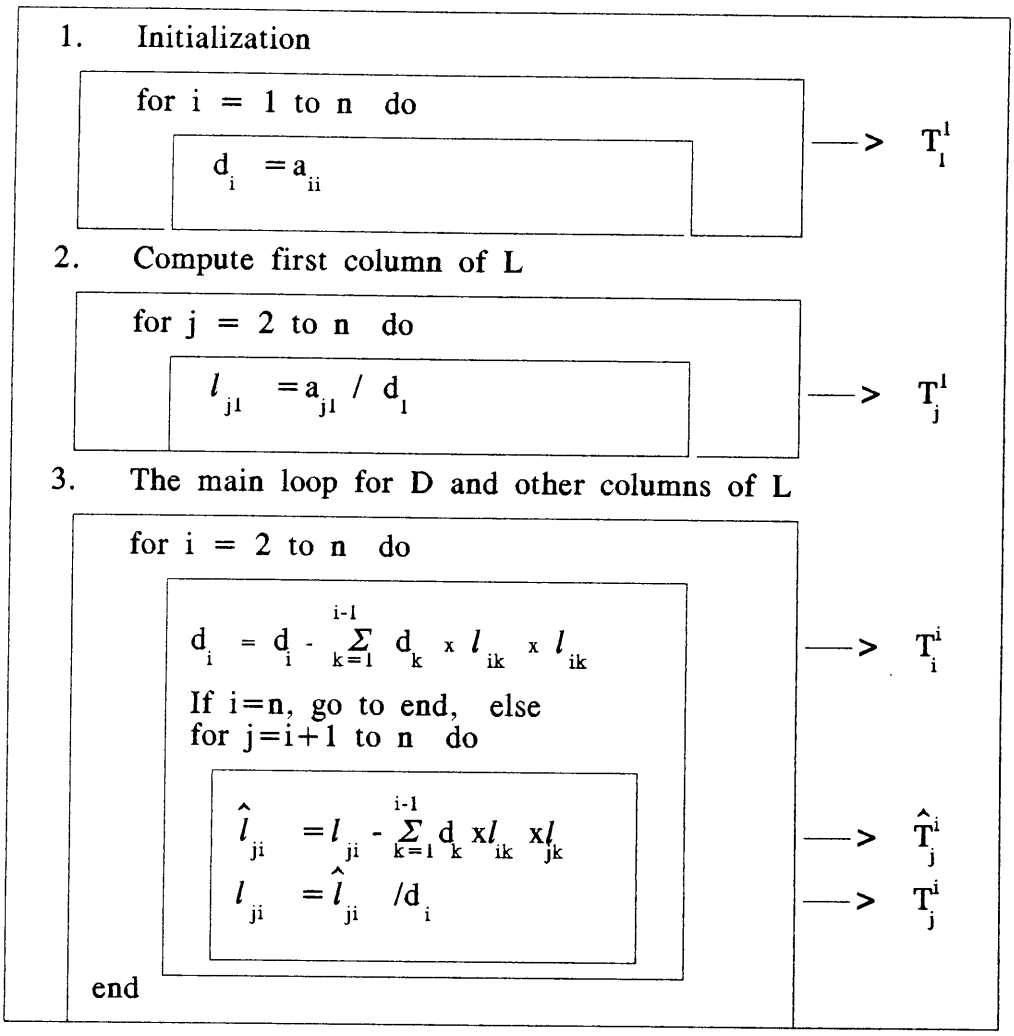
2.5 Cholesky Method

In [23], the implementation of Cholesky method is described. When A is a symmetric positive definite matrix, it can be factorized in the form:

$$A = L\,D\,L^T = (L\,D^{1/2})\,(D^{1/2}L^T) = L'\,L'^T ,$$

where L is a lower triangular matrix with identity elements and D is a diagonal matrix with positive elements. To avoid the quantity $D^{1/2}$, we prefer to compute the factors L and D rather than L′.

The algorithm LDL^T is as follows:



The total time for LDL^T using a single processor is

$$t_1 = \sum_{i=2}^{3} [3(i-1) + 3(i-1)(n-i) + (n-i)] + (n-1)$$

$$= 3n(n^2+n-2/2) - 2(n^2-n) - 3(2n^3+2n^2+n-6/6) + 5(n^2+n-2/2) - 2(n-1)$$

$$= (9n^3+9n^2-18n-12n^2+12n-6n^3-9n^2-3n+18+15n^2+15n-30n-12n+12)/6$$

$$= (3n^3+3n^2-6n)/6$$

$= n^3/2 + n^2/2 - 6$ steps

But, actually when we factor the matrix A into LL^T, using the following algorithm:

$$l_{ii} = (a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2)^{1/2} \quad , \quad i=1,\ldots,n$$

$$l_{ji} = \frac{1}{l_{ii}} (a_{ji} - \sum_{k=1}^{i-1} l_{jk} \; l_{ik}) \quad , \quad j=i+1,\ldots,n.$$

the cost is $O(n^3/6)$ (see appendix A).

The above algorithm divides the program LDL^T into a set of computational tasks, T, where a task is an indivisible collection of computational activities. The above algorithm gives the definition of the tasks T_i^i, $\hat{T}_j^i$ and T_j^i.

Consider the parallelism in LDL^T when the number of processors is 0(n). Let

$$J = \{ T_j^i : 1 \le i \le n \; ; \; 1 \le j \le n, \; \hat{T}_j^i : 2 \le i \le n \; ; \; i \le j \le n \}$$

be the set of these tasks.

The length of the longest path

$LP = (T_1^1, T_2^1, T_2^2, T_3^2, T_3^3, \ldots , T_n^{n-1}, T_n^n)$ is given by

$$W(LP) = 3/2n^2 - 1/2n - 1.$$

The execution time of schdl, using n-1 processors, is given by

$t_p = 3/2n^2 - n/2 - 1 \approx 3/2n^2$ for larger n.

The speedup and efficiency are given by

$$S_p = t_1/t_p = \frac{n^3/2 + n^2/2 - n}{3/2n^2 - n/2 - 1} = n/3 + O(n) + O(1/n)$$

$$E_p = S_p/p = 1/3 + O(1/n^2)$$

which for large n becomes

$$S_p \approx n/3 \quad , \quad E_p \approx 0.3333 \ ,$$

But, if we define the speedup of parallel algorithm to be ratio between the execution time of the most efficient sequential algorithm running on a serial machine and the execution time of parallel algorithm running on parallel computer, we might find

$$S_p = t_1/t_p = \frac{n^3/6 + O(n^2)}{3/2n^2 - n/2 - 1} \approx n/9$$

for larger n. The corresponding efficiency is

$$E_p = S_p/p \approx 0.1111$$

2.6 WZ Factorization

We shall give a brief description of WZ factorization method for the solution of (2.1.1). The method is due to [7]. The idea is to factorize A into the form WZ, where A is nxn nonsingular matrix. A has the form

$$A = WZ$$

where

$$W = \begin{bmatrix} 1 & & & & & 0 \\ w_{2,1} & 1 & & & 0 & w_{2,n} \\ w_{3,1} & w_{3,2} & 1 & 0 & w_{3,n-1} & \vdots \\ \vdots & \vdots & 1 & & \vdots & \vdots \\ \vdots & w_{n-2,0} & 0 & 1 & w_{n-2,n-1} & \vdots \\ w_{n-1,1} & 0 & & & 1 & w_{n-1,n} \\ 0 & & & & & 1 \end{bmatrix} \quad \text{and}$$

$$Z = \begin{bmatrix} z_{1,1} & z_{1,n} & \cdots & z_{1,n-1} & z_{1,n} \\ 0 & z_{2,2} & \cdots & z_{2,n-1} & 0 \\ \vdots & 0 & & 0 & \vdots \\ 0 & z_{n-1,2} & & z_{n-1,n-1} & 0 \\ z_{n,1} & z_{n,2} & \cdots & z_{n,n-1} & z_{n,n} \end{bmatrix}$$

The elements of W and Z are given by

$$w_{ij} = \begin{cases} 1, \ i = j \\ \begin{cases} 0, \ i = 1(1)[n/2] \ , \ j = (i+1)(1)(n-i+1), \\ 0, \ i = m(1)n \ , \ j = (n-i+1)(1)(i-1), \end{cases} \\ w_{ij} \ , \text{ otherwise} \end{cases}$$

$$z_{ij} = \begin{cases} z_{ij} \ , \ i = 1(1)[(n+1)/2] \ , \ j = i(1)(n-i+1) \ , \\ z_{ij} \ , \ i = m(1)n \ , \ j = (n-i+1)(1)i \ , \\ 0 \ , \text{ otherwise} \end{cases}$$

where $m = \{n+1 - \lfloor n/2 \rfloor\}$ and the symbol $\lfloor l \rfloor$ denotes the largest integer not greater than the real value of l.

Assuming A is positive definite and symmetric, we let $W = [W_1, W_2, \ldots, W_n]$ and $Z^t = [Z_1, Z_2, \ldots, Z_n]$ where

$$W_i = \begin{cases} [\underset{i-1}{0,\ldots}, 0, 1, W_{i+1,i}, \ldots, W_{n-i,i}, \ldots, 0, \ldots, 0]^t \ , \ i=1(1)\lfloor (n-1)/2 \rfloor \\ [\underset{i-1}{0,\ldots}, 0, 1, 0, \ldots, 0]^t = \begin{cases} n/2 \\ (n+2)/2 \end{cases} , \quad n \text{ even} \quad i=(n+1)/2 \ , \ n \text{ odd} \\ [\underset{n-i+1}{0,\ldots}, 0, W_{n-i-2,i}, \ldots, W_{i-1,i}, 1, 0, \ldots 0]^t \quad i = \lfloor (n+4)/2 \rfloor (1)n \end{cases}$$

and

$$Z_i = \begin{cases} [0,\ldots,0,\ \underset{i-1}{}\ z_{i,i},\ldots,z_{i,n-i+1},\ 0,\ldots,0]^t & i = 1(1)\lfloor (n+1)/2 \rfloor \\ [0,\ldots,0,\ \underset{n-i}{}\ z_{i,n-i+1},\ldots,z_{i,i},0,\ldots 0]^t & i = \lfloor (n+3)/2 \rfloor (1)n. \end{cases}$$

Therefore

$$A = WZ = \sum_{i=1}^{n} W_i Z_i^t$$

To compute the elements of the matrices W and Z, at the beginning of the kth stage,we have the column vectors

$[w_1, w_2,\ldots, w_{k-1}, w_{n-k+2},\ldots, w_n]$ and the matrix

$$A_k = A - \sum_{i=1}^{k-1} W_i Z_i^t - \sum_{i=n-k+2}^{n} W_i Z^t{}_i$$

whose first and last (k-1) rows and columns are zero.

Thus

$$\left.\begin{aligned} z_{k,j} &= a_{k,j}^{(k)} \\ z_{n-k+1,j} &= a_{n-k+1,j}^{(k)} \end{aligned}\right\} \quad j = k,\ n-k+1$$

and

$$\left.\begin{aligned} z_{k,k}\ w_{j,k} + z_{n-k+1,k}\ w_{j,n-k+1} &= a_{j,k}^{(k)} \\ z_{k,n-k+1}\ w_{j,k} + z_{n-k+1,\ n-k+1}\ w_{j,\ n-k+1} &= a_{j,\ n-k+1}^{(k)} \end{aligned}\right\} \quad (2.6.1)$$

for $j = k + 1(1)\ n-k$.

Then A_{k-1} becomes

$$A_{k+1} = A_k - W_k Z_k^t - W_{n-k+1}\ Z_{n-k+1}^t .$$

At least $\lfloor (n-2)/2 \rfloor$ such stages are required to compute all the elements of W and Z. The coefficients $w_{j,k}$ and $w_{j,n-k+1}$ for $j = k+1(1)(n-k)$ can be obtained

from (2.6.1) in 3 time steps using 2(n-2k+2)+1 processors.

To solve (2.1.1), rewrite $A\underline{x} = \underline{b}$ as

$WZ\underline{x} = \underline{b}$, so that we first solve an n element vector $\underline{q}$ where

$$W\,\underline{q} = \underline{b} \qquad (2.6.2)$$

and

$$Z\,\underline{x} = \underline{q} \qquad (2.6.3)$$

To solve (2.6.2), let

$$W\,\underline{q} = \underline{b} = b^{(1)} \longrightarrow \sum_{i=1}^{n} q_i\, w_i = b^{(1)}$$

hence,

$$\left.\begin{aligned} q_k &= b_k^{(k)} \\ q_{n-k+1} &= b_{n-k+1}^{(k)} \end{aligned}\right\} \quad \text{for } k = 1,2,\ldots, \lfloor n/2 \rfloor$$

where

$$b^{(k)} = b^{(k-1)} - q_{k-1}\, w_{k-1} - \qquad q_{n-k+2}\, w_{n-k+2} \,. \qquad (2.6.4)$$

At the kth stage, compute the quantity (2.6.4) in one multiplication using 2(n-2k) processors and two subtractions using (n-2k) processors. Therefore, 3 time steps are required at the kth stage and then the system (2.6.2) requires $3\lfloor (n-2)/2 \rfloor$ steps to be solved. The system (2.6.3) can be solved as follows:

At k^{th} stage, solving 2x2 system

$$z_{m-k+1,m-k+1} x_{m-k+1} + z_{m-k+1,n-1+k} x_{n-m+k} = q_{m-k+1}$$

$$z_{n-m+k,m-k+1} x_{m-k+1} + z_{n-m+k,n-m+k} x_{n-m+k} = q_{n-m+k}$$

where $m = \lfloor (n-1)/2 \rfloor$, and this system requires 3 time steps using Cramer's rule. Therefore, the solution of (2.6.3) can be solved in a total of $6\lfloor (n-1)/2 \rfloor$ time steps using no more than 2(n-2) processors.

2.7 Linear Recurrence Relations and Algorithms for their Solution

We shall, in the course of this section, define the linear recurrence, describe the column sweep algorithm and the product recurrent algorithm [21] for solving dense systems. Defining a linear recurrence system R < n, m > of order m for n equations by

$$R < n, m >: x_k = \begin{cases} 0 & \text{for} \quad k \leq 0 \\ c_k + \sum_{j=k-m}^{k-1} a_{kj} x_j & \text{for} \quad 1 \leq k \leq n \end{cases} \tag{2.7.1}$$

where $m \leq n-1$. This system is called a general linear recurrence system and is denoted by R < n >.

Rewrite (2.7.1) in matrix-vector notation as

$$\underline{x} = \underline{c} + A \underline{x} \tag{2.7.2}$$

where $\underline{x} = (x_1, x_2, \ldots, x_n)^t$, $\underline{c} = (c_1, c_2, \ldots, c_n)^t$ and $A = a_{ik}$, i, k=1,2,...n is strictly a lower triangular matrix with $a_{ik} = 0$ for $i \leq k$ or $i - k > m$. A is a band matrix for $m < n-1$.

For example, an R < 4,2 > has the form

$$\begin{aligned} x_1 &= c_1 \\ x_2 &= c_2 + a_{21} x_1 \\ x_3 &= c_3 + a_{31} x_1 + a_{32} x_2 \\ x_4 &= c_4 + \quad + a_{42} x_2 + a_{43} x_3 \ . \end{aligned} \tag{2.7.3}$$

It should be noted that the constant vector and the coefficients matrix generally contain values which can be calculated before the recurrence computation starts.

The column sweep algorithm is as follows:

Given any R < n > system, we initially know the value x_1. Then

Step 1: The expressions $a_{i1}x_1 + c_i$ can be evaluated in parallel. Now x_2 is known and there remains an R < n-1 > system.

Step 2: Evaluate the expressions $a_{i2}x_2 + c_i^{(1)}$ with $c_i^{(1)} = a_{i1}x_1 + c_i$ are evaluated in parallel. Hence x_3 is known.

Step k: x_k is known and the expressions $a_{ik}x_k + c_i$ are evaluated where $c_i^{(k-1)} = a_{i,k-1} + c_i^{(k-2)}$ giving x_{k+1}.

Clearly for nxn system requires n-1 steps and n-1 processors at step 1 and fewer thereafter. So $T_p = 2(n-1)$ with $p = n-1$. The time required to solve the system using a single processor is $T_1 = n(n-1)$. Hence, the speedup is given by

$$S_p = n(n-1)/2(n-1) = n/2$$

with an efficiency of

$$E_p = S_p/p = n/2(n-1) > \frac{1}{2}$$

It can be seen that the column sweep algorithm for an R < n,m > system has speedup factor $S_m = O(m)$.

One of the fastest methods for solution R < n,m > system is called the recurrent product algorithm. The description of the algorithm is as follows:

Given R < n > system, then (2.7.2) can be written as

$$\underline{x} = (I - A)^{-1}\underline{c} = L^{-1}\underline{c}$$

where A is strict lower triangular matrix, and L^{-1} can be expressed in the product form

$$L^{-1} = M_n M_{n-1} M_{n-2} \ldots M_1$$

with

$$M = \begin{bmatrix} 1 & & & & 0 \\ & 1 & & & \\ & & 1/a_{ii} & & \\ & & -a_{i+1}/a_{ii} & 1 & \\ 0 & & -a_{ni}/a_{ii} & & 1 \end{bmatrix}$$

Thus

$$\underline{x} = M_{n-1}M_{n-2}\ldots M_1\underline{c}$$

is the product of n-1 lower triangular matrix. The solution vector x can be evaluated in parallel.

From (2.7.3), we can write the matrix A as

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 \\ a_{21} & 0 & 0 & 0 \\ a_{31} & a_{32} & 0 & 0 \\ 0 & a_{42} & a_{43} & 0 \end{bmatrix} \qquad L = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -a_{21} & 1 & 0 & 0 \\ -a_{31} & -a_{32} & 1 & 0 \\ 0 & -a_{42} & -a_{43} & 1 \end{bmatrix}$$

To find the L^{-1},

1st column:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -a_{21} & 1 & 0 & 0 \\ -a_{31} & -a_{32} & 1 & 0 \\ 0 & -a_{42} & -a_{43} & 1 \end{bmatrix} \begin{bmatrix} l_{11} \\ l_{21} \\ l_{31} \\ l_{41} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$\Rightarrow l_1 = 1,\ l_{21} = a_{21}\ ,\ l_{31} = a_{31} + a_{32}a_{21}$

and $l_{41} = a_{42}a_{21} + a_{43}(a_{31} + a_{32}a_{21})$.

2nd column:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -a_{21} & 1 & 0 & 0 \\ -a_{21} & -a_{32} & 1 & 0 \\ 0 & -a_{42} & -a_{43} & 1 \end{bmatrix} \begin{bmatrix} l_{12} \\ l_{22} \\ l_{32} \\ l_{42} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

$\Rightarrow l_{12} = 0,\ l_{22} = 1,\ l_{32} = a_{32}$ and $l_{42} = a_{42} + a_{43}a_{32}$.

3rd column:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -a_{21} & 1 & 0 & 0 \\ -a_{31} & -a_{32} & 1 & 0 \\ 0 & -a_{42} & -a_{43} & 1 \end{bmatrix} \begin{bmatrix} l_{13} \\ l_{23} \\ l_{33} \\ l_{43} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$\Rightarrow l_{13} = 0,\ l_{23} = 0,\ l_{33} = 1$ and $l_{43} = a_{43}$, finally,

4th column:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -a_{21} & 1 & 0 & 0 \\ -a_{31} & -a_{32} & 1 & 0 \\ 0 & -a_{42} & -a_{43} & 1 \end{bmatrix} \begin{bmatrix} l_{14} \\ l_{24} \\ l_{34} \\ l_{44} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$\Rightarrow l_{14} = 0,\ l_{24} = 0,\ l_{34} = 0$ and $l_{44} = 1$.

$$\therefore L^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ a_{21} & 1 & 0 & 0 \\ a_{31}+a_{32}a_{21} & a_{32} & 1 & 0 \\ a_{42}a_{21}+a_{43}(a_{31}+a_{32}a_{21}) & a_{42}+a_{43}a_{32} & a_{43} & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & a_{43} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & a_{32} & 1 & 0 \\ 0 & a_{42} & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ a_{21} & 1 & 0 & 0 \\ a_{31} & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

$$= M_3 M_2 M_1.$$

Hence,

$$\underline{x} = L^{-1}\,\underline{c} = M_3 M_2 M_1\,\underline{c}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & a_{43} & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & a_{32} & 1 & 0 \\ 0 & a_{42} & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ a_{21} & 1 & 0 & 0 \\ a_{31} & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}$$

so we have

$$\begin{bmatrix} c_1 \\ a_{21}\,c_1 + c_2 \\ a_{32}\,(a_{21}\,c_1 + c_2) + a_{31}\,c_1 + c_3 \\ (a_{43}\,a_{32} + a_{42})(a_{21}c_1 + c_2) + a_{43}\,(a_{31}\,c_1 + c_3) + c_4 \end{bmatrix}$$

Then any linear recurrence system R < n,m > can be computed in

$$T_p < (2 + \log_2 m) \log_2 n - (\log_2^2 m + \log_2 m)/2$$

with

$$p \leq \begin{cases} m(m+1)n/2 + O(m^3) & \text{for } 1 \leq m \leq n/2 \\ n^3/68 + O(n^2) & \text{for } n/2 < m \leq n-1 \end{cases}$$

In the partiuclar case of an R < n,2 > system with large n the necessary time and number of processors for the evaluation of the system can be shown by an exact calculation to satisfy

$$T_p \leq (2 + \log_2 2) \log_2 n - \log_2 2 (\log_2 2+1)/2 = 3 \log_2 n - 1$$

with

$$p \leq 3n-8.$$

Hence, the parallel evaluation for the above example with m = 2, n = 4 gives

$$T_p \leq 3\log_2 4 - 1 = 5$$

time unit steps and p ≤ 4. More details can be found in [21].

2.8 Dense Triangular Systems

In this section we present two algorithms which are given in [41] solving the system L $\underline{x}$ = $\underline{c}$ for dense triangular system and give bounds on the time and the number of required processors. In [3] and [16] they have shown that, given p = $O(n^3)$ processors, a triangular system L $\underline{x}$ = $\underline{c}$ can be solved in $O(\log_2^2 n)$ steps. The two algorithms are as follows:

a) Algorithm 1

It is assumed that the system to be solved is of the form L $\underline{x}$ = $\underline{c}$ with L a lower triangular matrix defined as L = $[\alpha_{ij}]$. The inverse is given by

$$L^{-1} = M_n M_{n-1} \ldots M_1$$

where

$$M_i = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & 1/\alpha_{ii} & & \\ & & & -\alpha_{i+1,i}/\alpha_{ii} & \ddots & \\ & 0 & & -\alpha_{n,i}/\alpha_{ii} & & 1 \end{bmatrix} \quad , \; i = 1,2, \ldots, n$$

and the solution is given by

$$\underline{x} = M_n M_{n-1} \ldots M_1 \underline{c}$$

Evaluating the product $M_{n-1} M_{n-2} \ldots M_1 \underline{c}$ in parallel requires $\mu = \log_2 n$ steps.

Let $M_i^{(0)} = M_i \qquad i = 1,2,\ldots, n-1$

$c^{(0)} = c$ and $\qquad s = 2^j \quad , j = 0,1,\ldots, \mu = \log_2 n$

Then at each stage $(j+1)$ we calculate the matrices

$$M_i^{(j+1)} = M_{2i+1}^{(j)} M_{2i}^{(i)} \text{ for } j = 0,1,\ldots, \mu-2 \text{ and } i = 1,2,\ldots,(n/2s) - 1$$

and the vector

$$c^{(j+1)} = M_1^{(j)} c^{(j)} \quad , \quad j = 0,1,\ldots, \mu-1.$$

Each matrix $M_i^{(j)}$ has the form

$$M_i^{(j)} = \begin{bmatrix} I_i^{(j)} & 0 & 0 \\ 0 & \hat{L}_i^{(j)} & 0 \\ 0 & S_i^{(j)} & \hat{I}_i^{(j)} \end{bmatrix}$$

where $\hat{L}_i^{(j)}$ is lower triangular matrix of order s, $I_i^{(j)}$ and $\hat{I}_i^{(j)}$ are unit matrices of order $q_i^{(j)} = is-1$ and $r_i^{(j)} = (n+1)-(i+1)s$ respectively. Clearly, $\hat{L}_i^{(0)} = \frac{1}{\alpha_{ii}}$ and $S_i^{(0)} = (-\frac{1}{\alpha_{ii}})\ (\alpha_{i+1,i},\dots\ ,\ \alpha_{n,i})^t$.

Let

$$S_i^{(j)} = \begin{bmatrix} U_i^{(j)} \\ V_i^{(j)} \end{bmatrix}, \quad i = 1,2,\dots\ ,\ (n/s) - 2$$

where $U_i^{(j)}$ is an sxs. Thus

$$\hat{L}_i^{(j+1)} = \begin{bmatrix} \hat{L}_{2i}^{(j)} & 0 \\ \hat{L}_{2i+1}^{(j)}\ U_{2i}^{(j)} & \hat{L}_{2i+1}^{(j)} \end{bmatrix} \text{ and } S_i^{(j+1)} = [S_{2i+1}^{(j)}\quad U_{2i}^{(j)} + V_{2i}^{(j)}, S_{2i+1}^{(j)}\ ]$$

$j = 0,1,\dots\ ,\ \mu\text{-}2$ and $i = 1,2,\dots\ ,\ (n/s) - 1$.

If $\underline{c}^{(j)t} = (b_1^{(j)},\ b_2^{(j)},\ b_3^{(j)})^t$ where $b_1^{(j)}$, $b_2^{(j)}$ and $b_3^{(j)}$ of orders s-1, s and $n + 1 - 2s$. Then

$$c^{(j+1)} = M_1^{(j)}c^{(j)} = \begin{bmatrix} b_1^{(j)} \\ \hat{L}_1^{(j)}b^{(j)}{}_2 \\ R_1^{(j)}b^{(j)}{}_2 + b_3^{(j)} \end{bmatrix}, \quad j = 0,1,\dots\ ,\ \mu - 1.$$

The triangular system of equation $L\ \underline{x} = \underline{c}$ where L is lower triangular matrix of order n, can be solved in $T = (\log_2^2 n)/2 + 3(\log_2 n)/2 + 3$ steps using no more than $p = (15/1024)n^3 + O(n^2)$ processors. Also L^{-1} can be calculated in

the same time T using no more than $p = (21\ n^3 + 60\ n^2)/128$ processors.

b) Algorithm 2

In [41], is given another algorithm for the solution (2.1.1) in which A is a complete lower triangular unit matrix. The solution of $L\ \underline{x} = \underline{c}$ is carried out by

$$\underline{x} = L^{-1}\underline{c} = D_{\mu-1}D_{\mu-2}\cdots\ D_0\ \underline{c}$$

where $\mu = \log_2 n$. Starting with $L^{(0)} = L$ we can then choose intermediate matrices D_i such that

$$L^{(j)} = D_{j-1}\ D_{j-2}\cdots\ D_0\ L^{(0)} \qquad , j = 1,2,\ldots\ , \mu\ .$$

Each matrix $L^{(j)}$ has the form

$$L^{(j)} = \begin{bmatrix} I_s & & & \\ G_{21}^{(j)} & I_s & & \\ G_{31}^{(j)} & G_{32}^{(j)} & I_s & \\ G_{n/s,1}^{(j)} & G_{n/s,2}^{(j)} & G_{n/s,n/s-1} & I_s \end{bmatrix}$$

where $s = 2^j$, and $G_{ij}^{(0)} = \alpha_{ij}/\alpha_{ii}$, therefore

$$(D^{(j)})^{-1} = \text{diag}\ (L_1^{(j)}, L_2^{(j)},\ldots\ , L_{n/s}^{(j)}\ )$$

where

$$L_i^{(j)} = \begin{bmatrix} I_s & 0 \\ G_{2i,2i-1}^{(j)} & I_s \end{bmatrix} \qquad i = 0,1,\ldots\ , n-1 \text{ and } j = 0,1,\ldots\ , n/s.$$

Thus, for stage (j+1) we have,

$$G_{ik}^{(j+1)} = \begin{bmatrix} I_s & 0 \\ -G_{2i,2i-1}^{(j)} & I_s \end{bmatrix} \begin{bmatrix} G_{2i-1,2k-1}^{(j)} & G_{2i-1,2k}^{(j)} \\ -G_{2i,2k-1}^{(j)} & G_{2i,2k}^{(j)} \end{bmatrix}$$

for $j = 0,1,\ldots, \mu-1$, $k = 1,2,\ldots, (n/2s) - 1$ and $i = k + 1,\ldots, n/s + 1$

and

$$C_i^{(j+1)} = \begin{bmatrix} I_s & \\ -G_{2i,2i+1} & I_s \end{bmatrix} \begin{bmatrix} c_{2i-1}^{(j)} \\ c_{2i}^{(j)} \end{bmatrix}$$

$j = 0,\ldots, \mu - 1$, $i = 1,\ldots, n/s+1$.

Hence, the total time for solving the linear system is given in [41] as

$$T_p = \log_2^2 n/2 + 3 \log_2^2 n/2 + 1$$

and the required number of processors is

$$p = n^3/32 + n^2/8.$$

2.9 Tridiagonal Systems

Stone [43] has developed an algorithm for solving a tridiagonal system of linear system in time $O(\log_2 n)$ using $O(n)$ processors. The explanation of Stone's algorithm is given in [38] as follows:

$$\text{Let } A = \begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & a_3 & b_3 & c_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}$$

be non-singular tridiagonal matrix where b_i, i = 1,2,... ,n are diagonal elements, and a_{j+1},c_j, j = 1,... , n-1 are the off diagonal elements. Let D be a diagonal matrix with elements α_i, L and U be identity lower and upper bidiagonal matrices with off diagonal elements β_{j+1} and γ_j , j = 1,2,... , n-1 respectively. Then A can be written as

$$A = L\,D\,U = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ \beta_2 & 1 & \ddots & \vdots \\ 0 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \\ 0 & 0 & \beta_n & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 & 0 & \cdots & 0 \\ 0 & \alpha_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \alpha_n \end{bmatrix} \begin{bmatrix} 1 & \gamma_1 & 0 & 0 \\ 0 & 1 & \gamma_2 & \\ \vdots & \ddots & \ddots & 0 \\ 0 & 0 & 1 & \gamma_{n-1} \\ & & & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \alpha_1 & \alpha_1\gamma_1 & 0 & \\ \beta_2\alpha_1 & \beta_2\alpha_1\gamma_1 + \alpha_2 & \alpha_2\gamma_2 & \\ 0 & \beta_3\alpha_2 & \beta_3\alpha_2\gamma_2 + \alpha_2 & \alpha_3\gamma_3 \\ & & \ddots & \ddots \\ & & \beta_{n-1}\alpha_{n-2}\gamma_{n-2} + \alpha_{n-1} & \alpha_{n-1}\gamma_{n-1} \\ & & \beta_n\alpha_{n-1} & \beta_n\alpha_{n-1}\gamma_{n-1} + \alpha_n \end{bmatrix}$$

Hence, we obtain the recurrence relation

$$b_1 = \alpha_1 \ , \ \gamma_1 = c_1/b_1 \ ,$$

$$b_i = \alpha_i + a_i \, c_{i-1}/\alpha_{i-1} \ , \ i = 2,3,\ldots \ , \ n \ ,$$

$$\beta_i = a_i/\alpha_{i-1} \ , \ i = 2,\ldots \ , \ n \ ,$$

$$\gamma_i = c_i/\alpha_i \ , \ i = 1,\ldots \ , \ n.$$

If we take the non-linear recurrence

$$b_i = \alpha_i + a_i \, c_{i-1}/\alpha_{i-1} \qquad , \qquad i = 2,3,\ldots \ , \ n$$

and multiplying both sides by

$$\mu_{i-1} = \alpha_1 \alpha_2 \ldots \alpha_{i-1} \qquad , \qquad \text{we have the linear recurrence}$$

$$\mu_i - b_i \mu_{i-1} + a_i c_{i-1} \mu_{i-2} = 0 \ , \ i = 2,3,\ldots \ , \ n$$

with $\mu_0 = 1$ and $\mu_1 = b_1$.

The evaluation of μ_i , i = 2,3,... , n requires 2 $\log_2$ n+O(1) steps using 2n-4 processors [38]. Hence the total time for solving the system is 6 $\log_2$ n+O(1) steps using 2n-4 processors. The resulting speedup over the sequential algorithm is of O(n/$\log_2$n) with an efficiency of O(1/$\log_2$n).

In [38], it is mentioned that the number of operations required by this parallel algorithm is O(n $\log_2$n), yielding a redundancy of O($\log_2$n). However, the algorithm fails if any leading principal submatrix of A is singluar. Observe that if pivoting is necessary to complete the factorization, the parallel algorithm reduces to its serial counterpart, requiring time O(n). Even if pivoting is not necessary, this parallel algorithm suffers from the possibility of over or underflow. Also, the bound on the absolute error in the calculating of the second linear recurrence in μ_i grows exponentially with n.

In [40], a parallel algorithm for solving tridiagonal systems is given, with discussion of its shortcomings and presents another algorithm to avoid them. The algorithm is based on the orthogonal factorization Q A = R, where A is a tridiagonal matrix given by

$$A = \begin{bmatrix} b_1 & c_1 & & & & \\ a_2 & b_2 & c_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & & a_n & b_n \end{bmatrix}$$

a) Algorithm 1

We annihilate the subdiagonal elements of A by applying the orthogonal transformation

$$Q = P^{(n-1)}_{n-1,n} , \ldots , P^{(2)}_{2,3} \; P^{(1)}_{1,2}$$

where $P^{(j)}_{i,i+1}$ rotates rows i and i+1 annihilating the elements in the position (i+1, j),

$$P_{i,i+1} = \text{diag}\,(I_{i-1} , U_{i,i+1} , I_{n-i-1})$$

and

$$U_{i,i+1} = \begin{pmatrix} C_i & S_i \\ -S_i & C_i \end{pmatrix}. \quad \text{where } C_i = \text{Cos } \theta_i \text{ and } S_i = \text{Sin } \theta_i, \; [39].$$

Therefore, $P^{(r-1)}_{r-1,r} \ldots P^{(2)}_{2,3} P^{(1)}_{1,2} A$ is given by

$$\begin{bmatrix} \alpha_1 & \beta_1 & \gamma_1 & & & \\ & \alpha_2 & \beta_2 & \gamma_2 & & \\ & \cdots\cdots & & & & \\ & & \alpha_{r-1} & \beta_{r-1} & \gamma_{r-1} & \\ & & & \lambda_r & t_{r-1} & c_r \\ & & & & a_{r+1} & b_{r+1} & c_{r+1} \\ & & & & & \cdots\cdots \end{bmatrix}$$

It follows that

$$t_r = \lambda_r/\alpha_r \quad , \quad s_r = a_{r+1}/\alpha_r \quad , \quad r \geq 1 \qquad (2.9.1)$$

with

$$\left.\begin{aligned} \lambda_r &= t_{r-1} b_r - t_{r-2} s_{r-1} c_{r-1} \quad , \\ \beta_r &= s_r b_{r+1} + t_{r-1} t_r c_r \quad , \qquad r \geq 1 \\ \gamma_r &= s_r c_{r+1} \quad , \end{aligned}\right\} \qquad (2.9.2)$$

where $t_0 = 1$, $S_0 = 0$. From (2.9.1) and the first of (2.9.2), we obtain the recurrence relation in α_r ,

$$t_r \alpha_r = t_{r-1} b_r - t_{r-2} (c_{r-1} a_r / \alpha_{r-1}). \qquad (2.9.3)$$

Let $w_k = t_k \prod_{i=1}^{k} \alpha_i$, then (2.9.3) reduces to

$$w_r = b_r w_{r-1} - (c_{r-1} a_r) w_{r-2} \quad , \quad r \geq 2$$

where $W_0 = 1$ and $W_1 = b_1$. This can be expressed as unit lower triangular system of equations,

$$\begin{bmatrix} 1 & & & & \\ -b_1 & 1 & & & \\ c_1 a_2 & -b_2 & 1 & & \\ & \cdots\cdots & \cdots\cdots & \cdots\cdots & \\ & & c_{n-2}\, a_{n-1} & -b_{n-1} & 1 \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{n-1} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad (2.9.4)$$

The system can be solved [39] in $2\log_2 n$ steps using $(2n-4)$ processors, $n \geq 4$. Obtaining the vector $(w_0, \ldots, w_{n-1})^t$ requires $T = 1 + 2\log_2 n$ steps with $P_1 = 2n-4$ processors. Defining

$$\theta_k = \prod_{i=1}^{k} \alpha_i^2$$

and using (2.9.1) and the fact that $t_k^2 + s_k^2 = 1$, we get

$$\theta_k = a_{k+1}^2 \theta_{k-1} + w_k^2 \quad , \quad k \geq 1$$

where the corresponding systems of equations is given by

$$\begin{bmatrix} 1 & & & \\ -a_2^2 & 1 & & \\ & -a_3^2 & 1 & \\ & & -a_n^2 & 1 \end{bmatrix} \begin{bmatrix} \theta_0 \\ \theta_1 \\ \vdots \\ \theta_{n-1} \end{bmatrix} = \begin{bmatrix} w_0^2 \\ w_2^2 \\ \vdots \\ w_{n-1}^2 \end{bmatrix}$$

where $\theta_0 = w_0 = 1$.

This system is solved in [39] in $2\log_2 n$ steps with n-1 processors. θ_i, $i=1,\ldots, n-1$ can be obtained in $T = 1+2\log_2 n$ steps using no more than $2(n-1)$ processors.

Obtain t_j, α_j and s_j from

$$t_j^2 = w_j^2/\theta_j \quad , \quad \alpha_j^2 = \theta_j/\theta_{j-1} \quad \text{and} \quad s_j^2 = a_{j+1}^2/\alpha_j^2$$

in two steps and one square root using 3(n-1) processors. Once these are obtained, β_j, γ_j, $j \geq 1$ and $\alpha_n = \lambda_n$ are obtained in 3 steps using 2n processors.

From [14], Q is lower Hessenberg with elements

$$\begin{aligned} \kappa_{ij} &= g_i \eta_i \qquad \text{if} \quad i \geq j \\ &= s_i \qquad \text{if} \quad j = i+1 \\ &= 0 \qquad \text{otherwise} \end{aligned}$$

where $g_i = t_i \zeta_{i-1}$, $\eta_i = t_{i-1}/\zeta_{i-1}$, $i=1,2,\ldots,n$,

with $\zeta_0 = (-1)^i \prod_{k=1}^{i} s_k$, $i=1,2,\ldots,n-1$

in which $\zeta_0 = t_0 = t_n = 1$. The matrix Q can be expressed as

$$Q = G\ L\ Y + S\ E$$

where $G = \text{diag}(g_1,\ldots,g_n)$, $Y = \text{diag}(\eta_1,\ldots,\eta_n)$, $S = \text{diag}(s_1,\ldots,s_{n-1},0)$ and $E = (0, e_1,\ldots,e_{n-1})$, where e_i is the ith column of the identity. L is a lower triangular matrix whose elements are all one. The evaluation of $(\zeta_1,\ldots,\zeta_{n-1})^t$ requires $\log_2 n$ steps using (n-2)/2 processors. The matrices G, Y and S are obtained in one step with 2n-3 processors. Consequently, the factorization QA = R can be obtained in $8+5\log_2 n$ steps. Hence, the solution $\underline{x}$ is obtained by solving $R\ \underline{x} = Q\ \underline{f}$. The algorithm requires $11+9\log_2 n$ steps and have square roots with 3n processors. The speedup is of $O(n/\log_2 n)$ with efficiency of $O(1/\log_2 n)$.

In [40], it is noted that the algorithm (1) does not fail if any of the principal minors vanishes, it still shares the two disadvantages of Stones algorithm, namely the possibility of over and underflow in computing $w_k = t_k \prod_{i=1}^{k} \alpha_j$ and $\theta_k = \prod_{i=1}^{k} \alpha_i^2$ for $1 \leq k \leq n-1$. Second, an error analysis of the parallel algorithm for solving (2.9.4) [39] shows that if the computed solution $\tilde{v}$ then

$$\| \tilde{v} - v \| \leq 2(n-1)\, \delta^n + \varepsilon \ , \qquad (2.9.5)$$

where $\| \ . \ \|$ denotes the ∞ - norm, ε is the unit of roundoff error, and $\delta \leq \phi^3 + 2\phi^2$ in which $\phi = \max \{b_i, c_i, a_i\}$.

The absolute error in the solution (2.9.5) can be extremely large for large n. To avoid the backdraws, algorithm (1) should be applied to systems of order no larger than $\log_2 n$. Hence, the probability of over or underflow in calculating w_k and θ_k is reduced. Also if $\delta \leq 2^l$, for an integer l, then (2.9.5) is reduced to

$$\| \tilde{v} - v \| \leq 2\delta\, n^l\, (\log(n/2))\, \varepsilon \ .$$

Now, we shall give algorithm (2) which is suitable when the number of processors is limited.

b) Algorithm (2)

Assuming that the tridiagonal matrix $A = A^{(0)}$ as above, such that $c_i a_{i+1} > 0$. First we shall give the following Lemmas.

Lemma 2.1

Let A, as above, satisfy the assumption $c_i a_{i+1} > 0$, i = 1,2,... , n-1.

Then A is of rank no less than n-1. Moreover, if A is of rank n-1, this means if $\alpha_n = \lambda_n = 0$ then $t_{n-1} \neq 0$.

Proof:

The proof of this first part of lemma is as follows:

Since $c_i a_{i+1} > 0$ there exits a nonsingular diagonal matrix D, such that DAD^{-1} is a symmetric tridiagonal matrix with positive off-diagonal element $\sqrt{c_i a_{i+1}}$; i = 1,... , n-1.

For the second part:

Let $t_{n-1} = 0$; since $\alpha_n = \lambda_n = 0$ then from (2.9.2) we have $t_{n-2} = b_n t_{n-1} / c_{n-1} s_{n-1}$. But $s_r = a_{r+1} / \alpha_r$ and c_r are both different from zero, for all r.

Thus t_{n-2} must be zero. From (2.9.1)

$$\lambda_{n-1} = t_{n-1}\alpha_{n-1} \text{ then } \lambda_{n-1} = 0.$$

Note that $\alpha_i > 0$ for i = 1,2,... , n-1. Continuing this process we get

$$t_1 = 0 \text{ and } \lambda_2 = b_2 t_1 - s_1 c_1 = 0.$$

This give the contradiction that either $s_1 = 0$ or $c_1 = 0$, proving the lemma.

Lemma 2.2

Let $V = \left[\gamma_0, G, g_0 \right]$ where G is an n x n symmetric tridiagonal matrix with positive off-diagonal elements and rank n-1, $\gamma_0 = ae_1$ and $g_0 = be_n$ in which a, b > 0.

Let Q G = R and let

$$P = \begin{bmatrix} 1 & & \\ h_1 & I_n & \\ 0 & h_2^t & 1 \end{bmatrix}$$

where $h_1 = e_1$ and $h_2 = e_n$.

If $QVP = [\ \gamma_1,\ \tilde{R},\ g_1\ ]$

then $\tilde{R}$ is a nonsingular upper triangular matrix and

$$W = \left[e_n^t \ \tilde{R}^{-1} \gamma_1 \right] \left[e_1^t \ \tilde{R}^{-1} g_1 \right] \geq 0.$$

Proof:

$\tilde{R}$ and R have the same diagonal elements except the last. The last diagonal element of $\tilde{R}$ is $\alpha_n = ct_{n-1}$ (from algorithm (1)), where $t_{n-1} \neq 0$. This means $\tilde{R}^{-1}$ exists. Also $\tilde{R} = Q(G+Ce_n e_n^t) = Q\tilde{W}$, and $\tilde{G}^{-1} = \tilde{R}^{-1}Q$. Also $\gamma_1 = aQe_1 + ce_1$, $g_1 = bQe_n$, where $\alpha_1 > 0$ is the first diagonal element of $\tilde{R}$. Thus $e_n^t \tilde{R}^{-1}\gamma_1 = ae_n^t \tilde{R}^{-1}Qe_1 = ae_n^t \tilde{G}^{-1}e_1$, and

$$e_1^t \tilde{R}^{-1}g_1 = be_n^t \tilde{R}^{-1}Qe_n = be_1^t \tilde{G}^{-1}e_n .$$

Since $\tilde{G}$ is a symmetric matrix, then $W = e_1^t \tilde{G}^{-1}e_n = e_n^t \tilde{G}^{-1}e_1$ and $W = acw^2 \geq 0$.

The results of lemma 2.2 hold if:

i) G is singular and $h_1 = 0$,

ii) G is nonsingular and $h_2 = 0$

iii) G is nonsingular and $P = I_{n+2}$.

Then the various stages of the algorithm are as follows:

Stage 1: From $A^{(1)} = DA^{(0)}D^{-1}$, where the elements of D are given by $\delta_1 = 1$, $\delta_{i+1} = \delta_i C_i / \sqrt{c_i a_{i+1}}$, $1 \leq i \leq n-1$. Hence $A^{(1)}$ is symmetric with elements

$[\tilde{c}_{j-1}, \tilde{b}_j, \tilde{c}_j]$, where $\tilde{b}_j = b_j$ and $\tilde{c}_j = \sqrt{c_j a_{j+1}}$.

Stage 2: Partition $A^{(1)}$ in the form

$$\begin{bmatrix} A_1^{(1)} & B_1^{(1)} & & & \\ B_1^{(1)t} & A_2^{(1)} & B_2^{(1)} & & \\ & \cdots\cdots\cdots & & & \\ & & B_{m-2}^{(1)t} & A_{m-1}^{(1)} & B_{m-1}^{(1)} \\ & & & B_{m-1}^{(1)t} & A_m^{(1)} \end{bmatrix}$$

where $A_i^{(1)}$ are symmetric tridiagonal matrices with positive off-diagonal elements, and of order $q=\log n$, except possibly $A_m^{(1)}$, and $m = [n/q]$. The columns of each $B_i^{(1)}$ are all zero except the first one $\tilde{c}_{iq}\tilde{e}_q$, where $\tilde{e}_i$ is the ith column of the identity I_q. Let $Q = \text{diag}\,(Q_1, Q_2, \ldots, Q_m)$ where $Q_i A_i^{(1)} = R_i$ is upper triangular and Q_i is orthogonal matrix; hence Q is also orthogonal. Now if $A^{(2)} = QA^{(1)}$,

$$A^{(2)} = \begin{bmatrix} R_1 & B_1^{(2)} & & & \\ L_1^{(2)} & R_2 & B_2^{(2)} & & \\ & \cdots\cdots\cdots & & & \\ & & L_{m-2}^{(2)} & R_{m-1} & B_{m-1}^{(2)} \\ & & & L_{m-1}^{(2)} & R_m \end{bmatrix}$$

All the columns of each $B_i^{(2)}$ are zero except the first one, $\tilde{c}_{iq} Q_i \tilde{e}_q$, and each $L_i^{(2)}$ has all its columns zero except the last, $\tilde{c}_{iq} Q_{i+1} \tilde{e}_1$.

Stage 3: Let P be an nxn matrix that differs from the identity I_n in, at most, the positions $(iq + 1, iq)$, $i=1,2,..,m-1$, with elements $\theta=1$ when the last diagonal element of R_i is zero, $\alpha_q^{(i)} = 0$. Form $A^{(3)} = A^{(2)}p$,

$$A^{(3)} = \begin{bmatrix} \tilde{R}_1 & B_1^{(2)} & & & \\ L_1^{(3)} & \tilde{R}_2 & B_2^{(2)} & & \\ & \cdots\cdots\cdots & & & \\ & & L_{m-2}^{(3)} & \tilde{R}_{m-1} & B_{m-1}^{(2)} \\ & & & L_{m-1}^{(3)} & R_m \end{bmatrix}$$

where $\tilde{R}_i = R_i$ and $L_i^{(3)} = L_i^{(2)}$ if R_i is nonsingular; otherwise $\tilde{R}_i = R_i + \tilde{c}_{iq} Q_i \tilde{e}_q \tilde{e}_q^t$, $i = 1,2,..,m-1$, and the last and only non zero column of $L_i^{(3)}$ is $C_{iq} Q_i + 1\tilde{e}_1 + \alpha_1^{(i+1)} \tilde{e}_1$, $\alpha_1^{(i)}$ being the first diagonal element of R_i. From algorithm 1, we can see that the last element of $\tilde{c}_{iq} Q_i \tilde{e}_q$ is given by $\tilde{c}_{iq} t_{q-1}^{(i)}$. If R_i is singular then $t_{q-1}^{(i)} \neq 0$ by lemma 2.2, and since $\tilde{c}_{iq} > 0$ by assumption, then $\tilde{C}_{iq} t_{q-1}^{(i)} \neq 0$ and $\tilde{R}_i^{-1}$ exists.

Stage 4:

$$\text{Let } R_m = \begin{bmatrix} U & u \\ o & \mu \end{bmatrix}$$

and $R = \text{diag}(\tilde{R}_1^{-1},\ldots, \tilde{R}_{m-1}^{-1}, \tilde{U}^{-1})$, where $\tilde{U}^{-1} = R_m^{-1}$ if R_m is non singular, otherwise $\tilde{U}^{-1} = \text{diag}\,(U^{-1},1)$. Then $RA^{(3)}$ is of the form

$$R\,A^{(3)} = \begin{bmatrix} I_q & B_1^{(3)} & & & \\ L_1^{(4)} & I_q & B_2^{(3)} & & \\ & \ddots & \ddots & \ddots & \\ & & L_{m-2}^{(4)} & I_q & B_{m-1}^{(3)} \\ & & & L_{m-1}^{(4)} & K \end{bmatrix}$$

where the first and only non zero column of $B_i^{(3)}$ is now $\tilde{c}_{iq}\tilde{R}_i^{-1}Q_i\tilde{e}_q$. $L_i^{(4)} = \tilde{R}_{i+1}^{-1}L_i^{(3)}$, $i=1,2,..,m-2$, has all its columns zero except the last one which is given either by $(\tilde{C}_{iq}\tilde{R}_{i+1}^{-1}Q_{i+1}\tilde{e}_1 + \alpha_1^{(i+1)}\tilde{R}_{i+1}^{-1}\tilde{e}_1)$ if $\tilde{R}_i \equiv R_i$ is non singular. K is the identity of order $\tilde{q} = n - (m-1)q$ if R_m is non singular; otherwise

$$K = \begin{bmatrix} I_{\tilde{q}-1} & \tilde{u} \\ & 0 \end{bmatrix}$$

with $\tilde{u} = U^{-1}u$. Finally, the last and only non zero colum of $L_{m-1}^{(4)}$ can be obtained. Let $\tilde{p}$ be a permutation that exchange columns iq and iq+1 for $i=1,2,..,$ m-1, and form $A^{(4)} = R\,A^{(3)}\tilde{p}$.

Stage 5: Note that rows q, q+1, 2q, 2q+1,..., (m-1)q, (m-1)q+1, and n form a tridiagonal matrix G of order 2m-1. Then there ixists a permutation M such that

$$A^{(s)} = MA^{(4)}\,M^4 = \begin{bmatrix} G & \\ S & I_l \end{bmatrix}, \quad \text{where } l = n-2m+1.$$

Since A is a nonsingular matrix then G is also singular. Let H = M R Q D and $J = D^{-1} p \tilde{p} M$; then $A^{(s)} = HAJ$. The system $A\underline{x} = \underline{f}$ can therefore be written as $A^{(s)}\underline{y} = \tilde{\underline{f}}$ where $\underline{y} = J^{-1}\underline{x}$ and $\tilde{\underline{f}} = \underline{f}$ which gives the form

$$\begin{bmatrix} G & \\ S & I_1 \end{bmatrix} \begin{bmatrix} \underline{y}_1 \\ \underline{y}_2 \end{bmatrix} = \begin{bmatrix} \tilde{f}_1 \\ \tilde{f}_2 \end{bmatrix}$$

Assuming that the tridiagonal $G\,\underline{y}_1 = \tilde{f}_1$ can be solved via algorithm 1 without difficulty, then

$\underline{y}_2 = \tilde{\underline{f}}_2 - s\underline{y}_1$, and $\underline{x} = J\underline{y}$.

The above process, stages 1 to 5, may be repeated until the reduced tridiagonal system is of order q. Once this last tridiagonal system is solved via algorithm 1, then it is easy to find the solution. Finally, algorithm 2 requires a time O((log q)(log n)) using less than 5n processors.

For more details see [40].

2.10 Parallelisation of Iterative Algorithms

We shall look for parallel elements in the Jocabi, Gauss-Seidel and SOR processes for solving (2.1.1). We shall describe the ideas which are mentioned in [42]. Consider the matrix A to be tridiagonal, decompose A into lower and upper triangular matrices L and R respectively so that A = LR. Assume without loss of generality, that the diagonal elements a_{kk} of A are all unity, then A can be written as

$$A = LR = A_L + I + A_R$$

where

$$L = A_L + D, R = I + J$$

and

D is the diagonal matrix, I is unit matrix and

$$A_L = \begin{bmatrix} 0 & & & & 0 \\ a_{21} & 0 & 0 & & \vdots \\ 0 & a_{32} & 0 & & \\ 0 & 0 & a_{43} & \ddots & \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & & a_{n,n-1} & 0 \end{bmatrix} \quad \text{and} \quad A_R = \begin{bmatrix} 0 & a_{12} & 0 & & \\ \vdots & \ddots & \ddots & \ddots & \\ & \ddots & \ddots & a_{23} & 0 \\ & & \ddots & \ddots & \ddots \\ & & & \ddots & 0 \\ & & & a_{n-1,n} & \\ 0 & & 0 & & 0 \end{bmatrix}$$

J has the same structure as A_R. (See appendix B).

a) The Jacobi algorithm

The solution of (2.1.1) is given by

$$x^{k+1} = -(A_L + A_R)\, x^k + b \quad , \qquad k = 0,1, \ldots, \qquad (2.10.1)$$

since this is a matrix vector iteration, the procedure is inherently parallel.

b) The Gauss-Seidel algorithm

The outer iteration is

$$(I + A_R)\, x^k = -A_L\, x^{k-1} + b \qquad (2.10.2)$$

$$\Rightarrow \quad x^k = -(I+A_R)^{-1} A_L\, x^{k-1} + (I+A_R)^{-1} b$$

$$x^k = -T x^{k-1} + U$$

where $T = (I+A_R)^{-1} A_L$ and $U = (I+A_R)^{-1} b$

The specific form of the components arising from (2.10.2) is

$$x_m^k = -t_m x_{m-1}^{k-1} + u_m$$
$$x_i^k = -t_i x_{i-1}^{k-1} + u_i - e_i x_{i+1}^k \quad , \quad i = m-1, m-2, \ldots, 2$$
$$x_1^k = u_1 - e_1 x_2^k \quad ,$$

the above components of x^k can only be calculated serially. But if we substitute

$$z = -A_L x^{k-1} + b \quad , \quad y = x^k$$

then the components of Z can be calculated in parallel and (2.10.2) can be written as

$$(I + A_R)\, y = z \qquad (2.10.3)$$

Then (2.10.3) is solved by the following iteration

$$y^i = z - A_R y^{i-1} \quad , \quad i = 1,2,\ldots$$

c) SOR algorithm

Introducing a factor W, then rewrite (2.10.2) as

$$(I + wA_R)\, x^k = [(1-w)I - wA_L]\, x^{k-1} + wb \qquad (2.10.4)$$
$$w > 0,\ k = 1,2,\ldots$$

This converges for $0 < w < 2$.

Now we substitute

$$z_w = [(1-w)I - wA_L]\, x^{k-1} + wb,$$
$$y_w = x^k \ ,$$

and (2.10.4) is reduced to the form

$$(I + wA_R)\, y = z \ . \qquad (2.10.5)$$

Hence, the system (2.10.5) is solved by the iteration

$$y_w^i = z_w - wA_R\, y_w^{i-1} \quad , \quad i = 1,2,\ldots$$

CHAPTER 3

A Survey of Parallel Algorithms for Ordinary Differential Equations

3.1 Introduction

A second part of the work carried out has been on the solution of ordinary differential equations. This chapter has two aims. The first is to present some basic theory of the numerical solution of initial value problems. The second is to review some work done on parallel methods for solving the initial value problems.

3.2 Initial Value Problem

We consider the initial value problem

$$y' = f(x,y) \quad , \quad y(a) = y_o \quad , \quad a \leq x \leq b \qquad (3.2.1)$$

Before attempting to solve (3.2.1), we would like to know whether a unique solution exists. The existence of a unique solution is given by the following theorem, whose proof may be found in [17].

Theorem 3.2

For the initial value problem (3.2.1), let f(x,y) be continuous for all points in the region D defined by $a \leq x \leq b$, $-\infty < y < \infty$, and let there exist a constant $L < \infty$ such that

$$\| f(x,y) - f(x,z) \| \leq L \| y - z \| \qquad (3.2.2)$$

for any $(x,y), (x,z) \varepsilon D$. Then there exists a unique function y(x) which satisfies (3.2.1). The condition on f(x,y) is called Lipschitz Condition and L is called Lipschitz Constant.

From the mean-value theorem, it is easy to see that f(x,y) satisfies Lipschitz Condition (3.2.2) if the partial derivatives $\frac{\partial f}{\partial y}$ is bounded in absolute value for x in [a,b] and all y. To see this, assume $|\frac{\partial f(x,y)}{\partial y}| \leq L$; then $f(x,y) - f(x,z) = \frac{\partial f(x,\upsilon)}{\partial y}(y-z)$, where υ is between y and z. Hence if $|\frac{\partial f}{\partial y}(x,y)| \leq L$ for $a \leq x \leq b$, then $\|f(x,y)-f(x,z)\| \leq L\,|y-z|$.

In the following sections, we shall introduce some of the methods for solving (3.2.1.) with respect to a constant stepsize h, and give concepts of the error.

3.3 Discrete Variable Methods

Discrete variable methods yield a sequence of approximation $y_n \approx y(x_n)$, where $x_{n+1} = x_n + h_n$, and $h_n > 0$ is the step size, $x_o = a$, $x_N = b$; $n = 0,1,\ldots,N-1$. Conceptually, the simplest way of advancing the solution from the point x_n to x_{n+1} is by Taylor series

$$y(x_{n+1}) = y(x_n) + h\,\Delta(x_n, y_n, h)$$

where

$$\Delta(x_n, y_n, h) = y'(x) + \frac{h}{2}y''(x) + \frac{h^2}{3!}y'''(x_n) + \ldots.$$

The Taylor series method is obtained by truncating the above series to give

$$y(x_{n+1}) = y(x_n) + h\,\phi(x_n, y_n, h) \quad , \quad n = 0,1,\ldots.$$

where

$$\phi(x,y,h) = f(x,y) + \frac{h}{2!}f'(x,y) + \ldots + \frac{h^{p-1}}{p!}f^{p-1}(x,y)$$

y_n will always denote an approximation to $y(x_n)$ and if y(x) is a solution of $y' = f(x,y)$, where we assume f(x,y) is infinitely differentiable, then the higher derivatives of y(x) are completely determined by the function f(x,y). We shall write $y'' = f'(x,y)$, $y''' = f''(x,y)$ and generally $y^{(p+1)} = f^p(x,y)$, $p = 1, 2, \ldots.$

$\phi(x,y,h)$ is called the increment function, $\phi(x,y,h)$ also satisfies a Lipschitz condition,

$$\| \phi(x,y^*,h) - \phi(x,y,h) \| \leq L \| y^* - y \|,$$

where y^*, y are any numbers.

For $p = 1$, we have Euler's method which is given by

$$y_{n+1} = y_n + hf(x_n,y_n). \qquad (3.3.1)$$

For $p = 2$, we have

$$y_{n+1} = y_n + h\left[f(x_n,y_n) + \frac{h}{2}\left[f_x(x_n,y_n) + f_y(x_n,y_n).f(x_n,y_n)\right]\right], \qquad (3.3.2)$$

from which, given y_0, approximations $\{y_n\}$ can be obtained, one step at a time.

Consider the general class of methods defined by

$$\sum_{i=0}^{k} \alpha_i y_{n+i} = h\phi(x_n;y_{n+k},\ldots , y_n; h) \qquad (3.3.3)$$

for $0 \leq n \leq N-k$, $b-a = Nh$, where $\{\alpha_i\}$ are constant, $\alpha_k \neq 0$ with given starting values $y_0,y_1,\ldots ,y_{k-1}$. If ϕ is independent of y_{n+k} the method is explicit, otherwise the method is implicit. This general class of methods contains the well-known methods which are Runge-Kutta methods and linear multistep methods, eg.

$$\text{RK: } k=1,\ \phi = \sum_{r=0}^{m} c_r k_r$$

$$k_r = f(x_n + ha_r\ ,\ y_n + \sum_{s=0}^{m} b_{rs}\ k_s)$$

$$\text{LMM: } \phi = \sum_{i=0}^{k} \beta_i\ f(x_{n+i}\ ,\ y_{n+i}).$$

Definition 3.1

The local truncation error of (3.3.1) at $x_{n+k} \varepsilon [a,b]$ is given by

$$t_{n+k} = \sum_{i=0}^{k} \alpha_i y(x_{n+i}) - h\ \phi\ (x_n;\ y(x_{n+k}),\ldots ,\ y(x_n);\ h)$$

The quantity t_{n+k} is the amount by which the true solution of (3.2.1) fails to satisfy (3.3.3) and may be regarded as a first measure of the accuracy of

the formula. For Euler's method the local truncation error at i^{th} step for (3.2.1) is given by

$$t_i = y_i - y_{i-1} - hf(x_{i-1},y_{i-1}) \quad , \quad i = 1,2,\ldots , N$$

where $y_i = y(x_i)$ denotes the exact value of the solution at x_i. The local truncation error measures the accuracy of the method at a specific step, assuming that the method was exact at the previous step. As such, it depends on the differential equation, the stepsize, and the particular step in the approximation. The local truncation error in Euler's method is $t_i = \frac{h^2}{2!}y''(\xi_i)$ for some ξ_i, $x_i < \xi_i < x_{i+1}$, and when $y''(x)$ is known to be bounded by a constant M on [a,b], this gives

$$| t_i | \leq \frac{h^2}{2!} M.$$

Definition 3.2

The quantity $\| y_n - y(x_n) \|$ is called the global error at $x = x_n$, $0 \leq n \leq N$ where y_n is the numerical approximation to the solution $y(x_n)$ at the point x_n. The global error is the error over the whole range. In the previous definition, we have seen that the local truncation error in Euler's method is $O(h^2)$, then the global error is O(h).

Definition 3.3

A method of class (3.3.3) is convergent if, when applied to any problem of class (3.2.1), $y_n \longrightarrow y(x)$ as $h \longrightarrow 0$ where $nh = x-a$ for any $x\varepsilon[a,b]$. The condition for a method of class (3.3.3) to be convergent is that it must be consistent and zero-stable.

3.4 Runge-Kutta Methods

The general form of explicit Runge-Kutta methods is

$$y_{n+1} = y_n + h\,\phi\,(x_n, y_n, h) \qquad (3.4.1)$$

$$\phi\,(x,y,h) = \sum_{r=1}^{m} b_r k_r$$

where

$$k_1 = f(x,y)$$

$$k_r = f\left(x + h\,c_r,\ y + h \sum_{s=1}^{r-1} a_{rs}\,k_s\right)$$

$$c_r = \sum_{s=1}^{r-1} a_{rs}\ ,\ r = 2,3,\ldots,m$$

where b_r, a_{rs} and c_r are real constants. The m-stage Runge-Kutta method involves m evaluations of f per step. These methods are easy to use since they are self-starting and permit a change of step length without difficulty during the computation. The parameters for the above formula can be put in a tableau as follows:

0	0				
c_2	a_{21}				
c_3	a_{31}	a_{32}			
:	:	:			
c_m	a_{m1}	a_{m2}	.....	$a_{m,m-1}$	
	b_1	b_2	.....	b_{m-1}	b_m

Here we assume

$$c_r = \sum_{s=1}^{m-1} a_{rs} \quad \text{and} \quad \sum_{r=1}^{m} b_r = 1$$

Implicit Runge-Kutta method is defined by

$$y_{n+1} = y_n + h\,\phi\,(x_n, y_n, h)$$

$$\phi\,(x_n, y_n, h) = \sum_{r=1}^{m} b_r\, k_r \qquad (3.4.2)$$

$$k_r = f(x + c_r\, h,\ y + h \sum_{s=1}^{m} a_{rs}\, k_s)\ ,\ r = 1,2,\ldots\ ,\ m$$

represented by the tableau:

c_1	a_{11}	a_{12}	.....	a_{1m}
c_2	a_{21}	a_{22}	.....	a_{2m}
c_3	⋮			
c_m	a_{m1}	a_{m2}	.....	a_{mm}
	b_1	b_2	.....	b_m b_{m+1}

The method (3.4.2) is fully implicit if $a_{rs} \neq 0$ for some $s > r$, the method is semi-implicit or semi-explicit if $a_{rs} = 0\ \forall\ s > r$, and the method is explicit if $a_{rs} = 0\ \forall\ s \geq r$. Implicit Runge-Kutta formulae are intended for stiff problems. The local truncation error is $O(h^{p+1})$ for sufficiently smooth problems. The minimum value of the number stages s, for given p is shown as follows:

p	1	2	3	4	5	6	7	8
s	1	2	3	4	6	7	9	11

Some examples of explicit Runge-Kutta methods:

For $s = p = 1$

This is the first method of Euler.

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array}$$

For $s = p = 2$.

From (3.4.1) we get

$$\begin{aligned} \phi(x,y,h) &= \sum_{r=1}^{2} b_r k_r \\ &= b_1 k_1 + b_2 k_2 \\ &= b_1 f(x,y) + b_2 f(x+hc_2,\ y+ha_{21}f(x,y)) \end{aligned}$$

where b_1, b_2, c_2 and a_{21} are constant to be determined.

Actually,

$$\begin{aligned} k_2 &= f(x+hc_2,\ y+ha_{21}f(x,y)) \\ &= f(x,y) + c_2 h \frac{\partial f(x,y)}{\partial x} + a_{21} hf(x,y). \frac{\partial f(x,y)}{\partial y} . \end{aligned}$$

Then we have,

$$\begin{aligned} \phi(x,y,h) &= b_1 f(x,y) + b_2 f(x,y) + b_2 c_2 h \frac{\partial f}{\partial x} + b_2 a_{21} h \frac{\partial f}{\partial y} . f(x,y) \\ &= (b_1 + b_2) f(x,y) + hb_2 c_2 f_x + hb_2 a_{21} f_y . f(x,y) \\ &= (b_1 + b_2) f(x,y) + hb_2 (c_2 f_x + a_{21} f_y . f(x,y)) \end{aligned}$$

from (3.3.2) we have

$$\phi(x,y,h) = f(x,y) + h\left[f_x + f_y . f(x,y)\right]/2$$

macth these two expansions, we get

$$b_1 + b_2 = 1 \quad , \quad b_2 c_2 = {}^1/_2 \quad .$$

This is a set of two equations in three unknowns and thus there exists a one parameter family of solutions.

For $c_2 = {}^1/_2, {}^2/_3$ and 1, we obtain the following forms

0		
1/2	1/2	
	0	1

For $c_2 = 1/2$

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

For $c_2 = 2/3$

0		
1	1	
	1/2	1/2

For $c_2 = 1$

More details about different types of Runge-Kutta methods can be found in [15].

3.5 Linear Multistep Methods

This is the second major class of methods used in practice. The general k-step linear multistep is defined by

$$\sum_{j=0}^{k} \alpha_j y_{n+j} = h \sum_{j=0}^{k} \beta_j y'_{n+j} \qquad (3.5.1)$$

we say that the method (3.5.1) is explicit if $\beta_k = 0$, and implicit if $\beta_k \neq 0$. For an explicit method equation (3.5.1) yields the current value y_{n+k} directly in terms of y_{n+j}, y'_{n+j}, $j = 0,1,\ldots, k-1$.

An implicit method, however, will call for the solution at each stage of the computation of the equation

$$y_{n+k} = h\beta_k y'(x_{n+k}, y_{n+k}) + g \qquad (3.5.2)$$

where g is a known function of previously calculated values y_{n+j} and y'_{n+j}, j = 0,1,... ,k-1.

When the differential equation in (3.2.1) is linear, then (3.5.2) is linear in y_{n+k}. When y′ is non-linear, there exists a unique solution of (3.5.2) for y_{n+k}, which can approach arbitrarily closely by iteration $y_{n+k}^{[s+1]} = h\,\beta_k\, y'(x_{n+k}, y_{n+k}^{[2]}) + g,\ y_{n+k}^{[0]}$ arbitrarily provided that $0 \leq L \leq 1$, where L is the Lipschitz constant with respect to y_{n+k} of the right hand side of (3.5.2). If the Lipschitz constant of y′ with respect to y is L*, then clearly we take L to have the value $L^*h|\beta_k|$. Thus a unique solution for y_{n+k} exists, and the above iteration convergent to it, if

$$h < 1/L^*|\beta_k| \qquad (3.5.3)$$

when implementing the method (3.5.1) one needs to supply starting values by some auxiliary starting procedure usually a one-step method such as Runge-Kutta method or a sequence of linear multistep formulae for different k (see appendix C).

An example of an explicit linear multistep formulae is the Adams-Bashforth method,

$$y_{n+3} = y_{n+2} + \frac{h}{12}(23y'_{n+2} - 16y'_{n+1} + 5y'_n)$$

and an implicit method is Adams-Moulton method,

$$y_{n+3} = y_{n+2} + \frac{h}{12}(5y'_{n+3} + 8y'_{n+2} - y'_{n+1}).$$

3.6 Predictor-Corrector Methods

In using an implicit linear multistep method to solve the initial value problem (3.2.1), at each step we must solve for y_{n+k} the equation:

$$y_{n+k} + \sum_{j=0}^{k-1} \alpha_j y_{n+j} = h\beta_k f(x_{n+k}, y_{n+k}) + h \sum_{j=0}^{k-1} \beta_j f_{n+j} \qquad (3.6.1)$$

where y_{n+j}, f_{n+j}, $j=0,1,\ldots,k-1$, are known. In general this equation is non-linear. As mentioned in the previous section, a unique solution for y_{n+k} exists and can be approached arbitrary closely by the iteration

$$y_{n+k}^{[s+1]} + \sum_{j=0}^{k-1} \alpha_j y_{n+j} = h\beta_k f(x_{n+k}, y_{n+k}^{[s]}) + h \sum_{j=0}^{k-1} \beta_j f_{n+j} \qquad (3.6.2)$$

where $y_{n+k}^{[0]}$ is arbitrary, provided that condition

$L < 1/L|\beta_k|$ is satisfied.

To make the initial guess $y_{n+k}^{[0]}$ as accurate as we can, we use an explicit method to estimate $y_{n+k}^{[0]}$. The explicit method is called the predictor, and the implicit is called the corrector. After getting the first corrected value $y_{n+k}^{[1]}$ we can continue the iteration till convergence. Let p denote an application of the predictor, c a single application of the corrector and E an evaluation of f in terms of known values of its argument. Let the linear multistep method used as predictor be defined by the characteristic polynomials

$$\rho^*(\mu) = \sum_{j=0}^{k} \alpha_j^* \mu^j \quad , \quad \delta^*(\mu) = \sum_{j=0}^{k-1} \beta_j^* \mu_j^j \qquad (3.6.3)$$

and that used as corrected by

$$\rho(\mu) = \sum_{j=0}^{k} \alpha_j \mu^j \quad , \quad \delta(\mu) = \sum_{j=0}^{k} \beta_j \mu^j \quad , \tag{3.6.4}$$

Then the modes $P(EC)^mE$ and $P(EC)^m$ are defined as follows for $m=1,2,...$

i) $P(EC)^mE$:

$$\left.\begin{aligned}
&P: \quad y_{n+k}^{[0]} + \sum_{j=0}^{k-1} \alpha_j^* y_{n+j}^{[m]} = h \sum_{j=0}^{k-1} \beta_j^* f_{n+j}^{[m]} \\
&E: \quad f_{n+k}^{[s]} = f(x_{n+k}, y_{n+k}^{[s]}) \\
&C: \quad y_{n+k}^{[s+1]} + \sum_{j=0}^{k-1} \alpha_j y_{n+j}^{[m]} = h\beta_k f_{n+k}^{[s]} + h \sum_{j=0}^{k-1} \beta_j f_{n+j}^{[m]}
\end{aligned}\right\} \quad s=0,1,..m-1 \tag{3.6.5}$$

$$E: \quad f_{n+k}^{[m]} = f(x_{n+k}, y_{n+k}^{[m]}) \quad .$$

ii) $P(EC)^m$:

$$\left.\begin{aligned}
&P: \quad y_{n+k}^{[0]} + \sum_{j=0}^{k-1} \alpha_j^* y_{n+j}^{[m]} = h \sum_{j=0}^{k-1} \beta_j^* f_{n+j}^{[m-1]} \\
&E: \quad f_{n+k}^{[s]} = f(x_{n+k}, y_{n+k}^{[s]}) \\
&C: \quad y_{n+k}^{[s+1]} + \sum_{j=0}^{k-1} \alpha_j y_{n+j}^{[m]} = h\beta_k f_{n+k}^{[s]} + h \sum_{j=0}^{k-1} \beta_j f_{n+j}^{[m+1]}
\end{aligned}\right\} \quad s=0,1,...,m-1 \tag{3.6.6}$$

Note that as $m \longrightarrow \infty$, the result of computing with either of the above modes will tend to those given by the mode of correcting to convergence. When $m=1$, the mode which is described in (ii) will be PEC, in which the final derivative evaluation after the correction is omitted (comparing to mode PECE), and in which the f_{n+1} value using the predicted y_{n+1} is used to next cycle of computation. Such methods save one derivative evaluation per step.

In order to start the computation, we need to used some auxiliary method to obtain the starting values.

3.7 Conditional Stability Theory

We have seen that for convergent method the numerical solution y_n tends to the exact solution $y(x_n)$ as $h \longrightarrow 0$, $h \longrightarrow \infty$, nh fixed. In practice the limit cannot be taken, so we investigate the behaviour of global discretization error as $h \longrightarrow \infty$, and h is fixed. This stability of the method (3.3.3) makes use of the scalar equation

$$y' = \lambda y \quad , \quad \lambda \text{ complex} \tag{3.7.1}$$

known as the test equation, $Re(\lambda) \leq 0$ implies any solution of (3.7.1) remains bounded, since if, e.g. $y(0) = 0$ is the initial condition, the exact solution is $y(x) = e^{\lambda x}$. For a method to be stable we require the behaviour of the approximate solution to mirror that of the exact solution.

Definition 3.4

The method (3.3.3) is said to be absolutely stable for a given λh if, for that λh, all the roots of the stability polynomial lie within unit circle. A region R of the complex plane is said to be a region of absolute stability of (3.3.3) if (3.3.3) is absolutely stable for all $\lambda h \varepsilon R$.

The condition for absolute stability for Runge-Kutta methods is $|\mu(\lambda h)| < 1$. Where $\mu(\lambda h)$ is polynomial in λh if the method is explicit, and a rational function if it is implicit. In figure (3.7.1) are given sketches

of R for these examples:

i) Euler's method:

$$y_{n+1} = y_n + h\, f(x_n, y_n)$$

when applied to (3.6.1) yields

$$y_{n+1} = (1+\lambda h)\, y_n$$

where $\mu(\lambda h) = 1+\lambda h$ and the condition for absolute stability is

$$|1+\lambda h| \leq 1 \qquad \Rightarrow \lambda h \in (-2,0)$$

which ensures that $y_{n+1} = (1+\lambda h)^n y_0 \to 0$ as $n \to \infty$.

ii) Backward Euler method:

$$y_{n+1} = y_n + h\, f(x_{n+1}, y_{n+1}) \quad ; \quad \mu(\lambda h) = \frac{1}{1-\lambda h} \quad \Rightarrow \lambda h \in (-\infty, 0).$$

iii) Any 4th order 4-stage explicit Runge-Kutta methods; where

$$\mu(\lambda h) = 1 + \lambda h + \frac{(\lambda h)^2}{2} + \frac{(\lambda h)^3}{6} + \frac{(\lambda h)^4}{24}.$$

Note that for each of m = 1,2,3,4, all m-stage explicit Runge-Kutta methods of order m have the same region of absolute stability $\Rightarrow \lambda h \in (-2.8, 0)$.

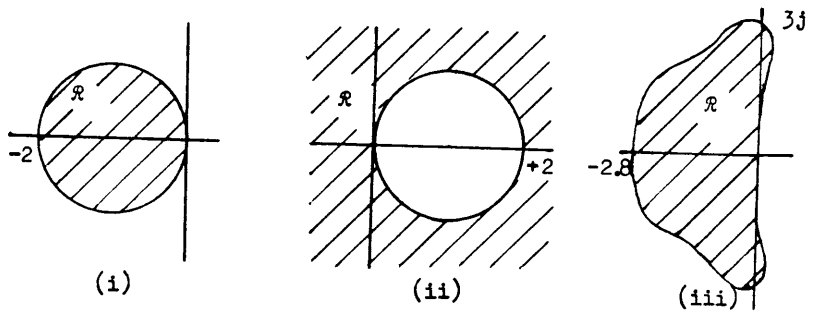


Figure 3.7.1 - (Hall and Watt [1976])

The most usual method for finding the region of absolute stability is the so-called boundary Locus method [15].

The midpoint rule is given by

$$y_{n+1} = y_{n-1} + 2h\, f(x_n, y_n) \quad .$$

Apply this method to test equation $y' = \lambda y$,

we have

$$\mu^2 - 2\lambda h\mu - 1 = 0$$

whose roots are

$$\mu_1 = \lambda h + \sqrt{\lambda^2 h^2 + 1} \quad \text{and} \quad \mu_2 = \lambda h - \sqrt{\lambda^2 h^2 + 1}$$

and we can see that

$$|\mu_1| = |\lambda h + \sqrt{\lambda^2 h^2 + 1}| > |\lambda h - \sqrt{\lambda^2 h^2 + 1}| = |\mu_2| \quad .$$

For $\lambda < 0$,

$$|\mu_2| = |\lambda h - \sqrt{\lambda^2 h^2 + 1}| > 1.$$

From the definition of the absolute stability (3.4) ensure that midpoint rule is absolutely unstable.

We wish to make further points. First, we shall give the coefficients of Adams-Bahsforth methods and Adams-Moulton methods together with local truncation errors. Second, we shall list the stability intervals of some predictor-corrector methods [25], which are used in practice. Table 3.7.1 illustrates the coefficients for Adams-Bahsforth methods for k=1,2,3 and 4.

Table 3.7.1

k	β_0	β_1	β_2	β_3	Order	L.T.E
1	1				1	$(h^2/2)\ y_n^{(2)}$
2	3/2	-1/2			2	$(5h^3/12)\ y_n^{(3)}$
3	23/12	-6/12	5/12		3	$(9h^4/24)\ y_n^{(4)}$
4	55/24	-59/24	37/12	-9/24	4	$(251h^5/720)\ y_n^{(5)}$

The following table illustrates the coefficients for Adams-Moulton methods for k=1,2,3 and 4.

Table 3.7.2

k	β_0	β_1	β_2	β_3	Order	L.T.E
1	1/2	1/2			2	$(-h^3/12)\ y_n^{(3)}$
2	5/12	8/12	-1/12		3	$(-h^4/24)\ y_n^{(4)}$
3	9/24	19/24	-5/24	1/24	4	$(-19h^5/720)\ y_n^{(5)}$
4	251/720	646/720	-264/720	106/720	5	$(-27h^6/1440)y_n^{(6)}$

The following table gives a stability intervals [25].

Table 3.7.3

Method	Stability Intervals
1. Euler : P Modified Euler : C $PE(CE)^s$	(-2,0) $(-\infty,0)$ (-2,0) , s=1 (-2,0) , s=2 (-2,0) , s=3
2. Nystrom : P Modified Euler : C $PE(CE)^s$	Unstable $(-\infty,0)$ (-2,0) , s=1 (-2,0) , s=2
3. Third Order Adams P C $PE(CE)^s$	(-0.55,0) (-6,0) (-1.8,0) , s=1 (-1.3,0) , s=2
4. Fourth Order Adams P C $PE(CE)^s$	(-0.3,0) (-3,0) (-1.3,0) , s=1 (-0.9,0) , s=2
5. Fifth Order Adams P C PE(CE)	(-0.2,0) (-1.8,0) (-0.95,0)

Definition 3.5

A method of class (3.3.3) is said to be strongly stable if it is consistent and satisfies the strong root condition ie. 1st characteristic polynomial of the method has only one root with magnitude one.

Some methods may have more than one root on the unit circle. These methods are usually called weakly stable. For example the characteristic

equation of midpoint rule is

$\mu^2 - 2h\lambda\mu - 1 = 0$ which has the roots

$\mu_1 = 1 + \lambda h + \frac{1}{2}(\lambda h)^2 + 0(h^4)$ and $\mu_2 = -1/\mu_1$.

If we consider the limit of h —> 0 then the two roots are $\mu_1 = +1$ and $\mu_2 = -1$. Thus the midpoint rule is a weakly stable method.

Definition 3.6

The method (3.3.3) is said to be relatively stable for a given λh, if for that λh, all the roots of the stability polynomial are less than $|e^{\lambda h}|$ in modulus. A region R of the complex plane is said to be a region of relative stability of (3.3.3) if (3.3.3) is relatively stable for all $\lambda h \varepsilon R$.

3.8 Stiff Equations

Several engineering and scientific fields such as chemical engineering, control theory, chemical kinetics, circuit theory, yield initial value problems with a property known as stiffness.

Definition 3.7

The initial value problem

$\underline{y}' = \underline{f}(x,\underline{y})$, $\underline{y}(a) = y_0$, $x \varepsilon [a,b]$

is said to be stiff in the interval IC[a,b] if, for x ε I.

If the derivatives $\frac{\partial f}{\partial y}$ are continuous and bounded, the Lipschitz constant

$L = \|\partial f/\partial y\| > \rho(\partial f/\partial y)$

where the spectral radius ρ is defined to be

$$\rho = \max_{i=1,\ldots,s} |\lambda_i| \quad .$$

Definition 3.8 (Dahlquist, 1963)

The numerical method is said to be A-stable if its region of absolute stability contains the whole of the half-plane $Re(\lambda h) < 0$. The A-stability property was originally formulated for linear multistep methods, and it proves to be a very restrictive requirement for such methods.

Definition 3.9 (Widlund, 1967)

A numerical method is said to be A(α)-stable, $\alpha\varepsilon(0,\pi/2)$, if its region of absolute stability contains the infinite wedge,

$$w_\alpha = \{\lambda h | -\alpha < \pi - \text{ang } \lambda h < \alpha\} ;$$ it is said to be A(0)-stable if it is A(α)-stable for some $\alpha\varepsilon(0,\pi/2)$.

(i) $Re(\lambda_i) < 0$, $i=1,2,\ldots,s$ and

(ii) $s(x) = \max_{i=1,..,s} Re(-\lambda_i) / \min_{i=1,..,s} Re(-\lambda_i)$ is large,

where λ_i are the eigen values of $\frac{\partial f}{\partial y}$ evaluated on the solution y(x) at x. The quantity s(x) is called the stiffness ratio and it provides a measure of the severity of the stiffness of the initial value problem.

For stability on such a problem we require $h\lambda\varepsilon R$, $i=1,\ldots,s$. This is a severe restriction on h for explicit methods, as shown in figure (3.8.1). The efficient solution of stiff systems requires the use of implicit formulae. In the initial rapid transient phase accuracy consideration dominate and an explicit method can be used efficiently.

The stiffness ration may be used as a measure of the stiffness of the

problem. A stiff problem is often referred to in the literature as a problem with "widely differing time constant" or a system with Large Lipschitz constant.

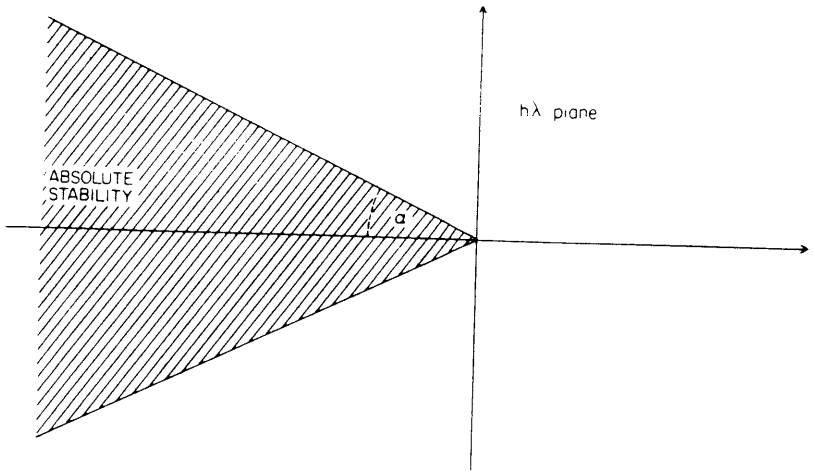


Figure 3.8.1 - A(α)-stability (Hall and Watt [1976])

Note that in the above definitions a given eigen value λ either lies inside or outside the wedge, regardless of the positive stepsize used. Widlud proved that no explicit linear multistep method could be A(0)-stable, and showed that implicit linear multistep methods of order three and four existed that were A(α)-stable for any $\alpha < \pi/2$.

In some respects the A-stability property is not stringent enough, and studies of A-stable one-step methods have produced a number of additional stability concepts. A one method applied to the equation

$y' = \lambda y$ gives an approximation

$y_{n+1} = \mu(\lambda h) y_n$ to the solution

$y(x_{n+1}) = e^{\lambda h}\, y(x_n)$, and A-stability ensures that $|\mu(\lambda h)| < 1$ for all $Re(\lambda h) < 0$. For many A-stable one-step methods, however, $\mu(\lambda h)$ is such that

$$|\mu(\lambda h) \longrightarrow 1 \quad \text{as} \quad Re(\lambda h) \longrightarrow -\infty$$

so that numerical appoximations to the rapidly decaying eigen solutions with very small time constants may decay only very slowly.

Definition 3.10

A one-step method is said to be L-stable if it is A-stable, and when applied to the equation

$y' = \lambda y$ with $Re(\lambda) < 0$, it gives

$y_{n+1} = \mu(\lambda h)\, y_n$, where

$|\mu\lambda h)| \longrightarrow 0$ as $Re(\lambda h) \longrightarrow \infty$.

There are some basic results on multistep methods:

1) An explicit k-step method cannot be A-stable.

2) The order of an A-stable linear multistep cannot exceed two.

3) There is only one A(o)-stable method with $p \geq k+1$, the trapezoidal rule.

More details can be found in [24].

3.9 Overview of Work on Parallel Methods

The second aim in this chapter is to review some work done on parallel methods for solving the initial value problem (3.2.1).

We have to ask ourselves why we use parallel computers for ODES? The answer is if the function is very expensive to evaluate, for example if

(3.2.1) refers to large sets of coupled equations, the problem may not be solvable without parallel computers; or the response time may be so important the problem is not solvable in a short enough time using conventional computers. Furthermore, the parallel computers can be cost effective.

There are two major means in which parallelism can be applied to ODES namely 'across the method' (different parts of the method are executed on different processors), and 'across the system' (each equation or group of equations in a large system is assigned to different processors)[13]. There is already significant literature on parallel methods for ordinary differential equations. We shall mention them in the following sections.

3.10 Predictor-corrector Methods

The ideas of parallel algorithm of [28] are most easily described by means of simple example. Let us consider the scalar ordinary differential equation

$$y' = f(x,y) \text{ where } x > 0 \text{ and } y(0) = y_0 \qquad (3.10.1)$$

At first sight, it seems that there is little scope of parallelism in solving (3.10.1) since the usual integration methods are essentially serial. There exists, however, parallel versions of serial predictor-corrector methods.

Fix a mesh size h and let $x_n = (n-1)h$, $n = 1,2,\ldots$, where $h > 0$. Let y_n be an approximation to the solution y at x_n, then the serial predictor-corrector formulas are:

$$y^p_{n+1} = y^c_n + \frac{h}{2}(3f^c_n - f^c_{n-1})$$

$$y^c_{n+1} = y^c_n + \frac{h}{2}(f^p_{n+1} + f^c_n)$$

where y^p_n and y^c_n are predicted and corrected values of y_n, f^c_n denotes $f(x_n,y^c_n)$ and f^p_n denotes $f(x_n,y^p_n)$.

The sequence of computations proceeds as follows:

$$\ldots \longrightarrow y^p_{n+1} \longrightarrow f^p_{n+1} \longrightarrow y^c_{n+1} \longrightarrow f^c_{n+1} \longrightarrow \ldots$$

and is clearly sequential calculations.

A parallel mode of integration is given by the following pair of formulas:

$$y^p_{n+1} = y^c_{n-1} + 2hf^c_n$$

$$y^c_n = y^c_{n-1} + \frac{h}{2}(f^p_n + f^c_{n-1})$$

where the corrector formula does not depend on the result of the corresponding predictor formula. The sequence of computation is divided into two parts:

$$\ldots \longrightarrow y^p_{n+1} \longrightarrow f^p_{n+1} \longrightarrow \ldots$$

$$\ldots \longrightarrow y^c_n \longrightarrow f^c_n \longrightarrow \ldots$$

which may be executed simultaneously on separate processors.

One possible parallel method has been presented by [11], [28] is the fourth-order, two processors predictor-corrector method

$$y^p_{n+1} = y^p_{n-1} + \frac{h}{3}(8f^p_n - 5f^c_{n-1} + 4f^c_{n-2} - f^c_{n-3})$$

$$y^c_n = y^c_{n-1} + \frac{h}{24}(9f^p_n + 19f^c_{n-1} - 5f^c_{n-2} + f^c_{n-3})$$

where the superscripts p and c refer to the predicted and corrected values and subscript n indicates the step number.

Suppose that the system to be solved is described by N first order differential equations. The predictor and corrector formulas are each solved once for each of the N equations. In the two-processor case, the predictor and corrector formulas are solved in parallel. One set in each processor. The fact is that the function evaluations f_n^c and f_n^p are also performed in parallel. The parallelism of the method guarantees that the function evaluation time per step decreases directly as the number of processors increases. The overall time for a single step is given by

$$T = Nt_{Lc} + t_f + 3Nt_{com} + 2t_{syn},$$

where t_{Lc} is the time to solve the predictor (corrector) formula for one component given the function values, t_f, is the function evaluation time for all N differential equations, t_{com} and t_{syn} are the communication and synchronization times respectively. Communication must occur between the processors at each step. At each step y^c and f^c, values must be passed from the corrector processor to predictor processor, and all of f^p values must be passed from the predictor processor to corrector processor. In this case, therefore, the question of optimal allocation of equations does not occur.

3.11 Block Methods

The property of these methods is produce simultaneously approximation to the solution of initial value problem at block points x_{n+1}, x_{n+2},... ,x_{n+k} in

each computational step. In [45], block method is introduced and may be applied either in one-step mode or in a predictor-corrector mode.

3.11.1 Block One-Step Methods

In one-step mode, a sequence of open and closed Newton-cotes formulas is applied to generate inital approximations to the solution y(x) at each point in the current block. Then implicit formulas are applied iteratively until convergence is obtained to the maximum order of accuracy obtainable. Only the solution at the last point in the block is used to start the next block. Let k be the number of points in a block. Assuming equally spaced nodes within each block, thus

$$x_{n+r} = x_n + rh, \qquad r = 1,2,\ldots,k,$$

where h is the stepsize.

The formulas for the sequential two-point block scheme are:

$$\begin{aligned}
y_{n+1,0} &= y_n + h\, y'_n \\
y_{n+1,1} &= y_n + \tfrac{h}{2}\left[y'_n + y'_{n+1,0}\right] \\
y_{n+2,1} &= y_n + 2h\, y'_{n+1,1} \qquad (3.11.1) \\
y_{n+1,2} &= y_n + \tfrac{h}{12}\left[5y'_n + 8y'_{n+1,1} - y'_{n+2,1}\right] \\
y_{n+2,s+1} &= y_n + \tfrac{h}{3}\left[y'_n + 4y'_{n+1,2} + y'_{n+2,s}\right], \quad s = 1 \text{ or } 2.
\end{aligned}$$

The method (3.11.1) is used to produce the approximations y_{n+1} and y_{n+2} to $y(x_{n+1})$ and $y(x_{n+2})$ respectively. The order and local truncation error for each formula of (3.11.1) are shown in the following table

Formula Number	Order	Local truncation error
1	1	$h^2/2 \quad y_n^{(2)}(\xi)$
2	2	$-h^3/12 \quad y_n^{(3)}(\xi)$
3	2	$h^3/3 \quad y_n^{(3)}(\xi)$
4	3	$-h^4/24 \quad y_n^{(4)}(\xi)$
5	4	$-h^5/90 \quad y_n^{(5)}(\xi)$

Table 3.2

Let $Y_{m,s}$ denote an approximation to $y(x_{n+m})$, $m = 1,2$. Let $Y_{o,s} = Y_n$ is the initial approximation, $Y'_{o,s} = y'(x_n,y_n)$, $s \geq 0$, where $Y_{0,s}$ and $Y'_{o,s}$ will not change with s.

The sequence of the computations for the algorithm uses (3.11.1) is described in [24], [36] as follows:

$$
\begin{aligned}
Y_{1,0} &= y_n + h\, Y'_{0,0} \\
Y'_{1,0} &= y'\left[x_{n+1}, Y_{1,0} \right] \\
Y_{1,1} &= y_n + \tfrac{h}{2}\left[Y'_{0,0} + Y'_{1,0} \right] \\
Y'_{1,1} &= y'\left[x_{n+1}, Y_{1,1} \right] \\
Y_{2,1} &= y_n + 2h\, Y'_{1,1} \\
Y'_{2,1} &= y'\left[x_{n+2}, Y_{2,1} \right] \\
Y_{1,2} &= y_n + \tfrac{h}{12}\left[5Y'_{o,1} + 8Y'_{1,1} - Y'_{2,1} \right] \\
Y'_{1,2} &= y'\left[x_{n+1}, Y_{1,2} \right] \\
Y_{2,2} &= y_n + \tfrac{h}{3}\left[Y'_{0,2} + 4Y'_{1,2} + Y'_{2,1} \right]
\end{aligned}
\qquad (3.11.2)
$$

$$Y'_{2,2} = y'\left[x_{n+2}, Y_{2,2} \right]$$

$$Y_{2,3} = y_n + \frac{h}{3}\left[Y'_{0,2} + 4Y'_{1,2} + Y'_{2,2} \right]$$

The computation is completed by setting

$$y_{n+1} = Y_{1,2} \quad \text{and} \quad y_{n+2} = Y_{2,3} .$$

This procedure has five function evaluations for two-step block. However, in order to apply the algorithm at the next block to obtain y_{n+3}, y_{n+4}, the only starting value is y_{n+2}, and the loss of accuracy in y_{n+1} does not affect subsequent points and the algorithm remains of order four.

The corresponding formulas for parallel two-point block scheme are

$$\begin{aligned} y_{n+r,0} &= y_n + rh\, y'_n , \qquad r = 1,2 \\ y_{n+1,s+1} &= y_n + \frac{h}{12}\left[5y'_n + 8y'_{n+1,s} - y'_{n+2,s} \right] \\ y_{n+2,s+1} &= y_n + \frac{h}{3}\left[y'_n + 4y'_{n+1,s} + y'_{n+2,s} \right] \end{aligned} \tag{3.11.3}$$

where $s = 0,1,2,(3)$ as mentioned in [45].

The first formula is Euler's method which has order one, and the local truncation errors are

$$\frac{h^2}{2} y_n^{(2)}(\xi_{n+1}) \text{ and } 2h^2 y_n^{(2)}(\xi_{n+2}) \text{ for } r = 1 \text{ and } r = 2$$

respectively.

The local truncation errors for the second and third formulas of (3.11.3) are

$$\frac{-h^4}{24} y_n^{(4)}(\xi_{n+1}) \text{ and } \frac{-h^5}{90} y_n^{(5)}(\xi_{n+2})$$

respectively.

To apply the above scheme, we need to compute $y'_{n+j,s}$; j = 0,1,2. So the scheme might be written as follows:

1) Compute y'_n

2) Processor r computes $y_{n+r,0}$ using Euler's method r = 1,2.

3) Send the value of $y_{n+r,0}$ to all processors. (3.11.4)
   Set s = 0.

4) Processor r computes $y'_{n+r,0}$, r = 1,2

5) Send $y'_{n+r,0}$ to all processors.

6) Processor r computes $y_{n+r,s}$, r = 1,2
   Set s = s + 1.

7) Go to 4.

In this case, y_{n+1} and y_{n+2} are obtained simultaneously for each step. The number of function evaluations is reduced from five in (3.11.2) when s = 2, to three or four in the parallel algorithm (3.11.4) when s = 0 or s = 1 respectively. Note the iteration may be stopped at s = 1 in (3.11.2) and s = 2 in (3.11.3).

An algorithm for the four-point case is represented by the sequence, as in [45],

$$y_{n+r,0} = y_n + rh\, y'_n,\ r=1,2,3,4, \qquad (3.11.5)$$

$$y_{n+1,s+1} = y_n + \frac{h}{720}\left[251y'_n + 646y'_{n+1,s} - 264y'_{n+2,s} + 106y'_{n+3,s} - 19y'_{n+4,s}\right]$$

$$y_{n+2,s+1} = y_n + \frac{h}{90}\left[29y'_n + 124y'_{n+1,s} + 24y'_{n+2,s} + 4y'_{n+3,s} - y'_{n+4,s}\right],$$

$$y_{n+3,s+1} = y_n + \frac{3h}{80}\left[9y'_n + 34y'_{n+1,s} + 24y'_{n+2,s} + 14y'_{n+3,s} - y'_{n+4,s}\right],$$

$$y_{n+4,s+1} = y_n + \frac{2h}{45}\left[7y'_n + 32y'_{n+1,s} + 12y'_{n+2,s} + 32y'_{n+3,s} + 7y'_{n+4,s}\right]$$

where s = 0,1,2,3,4 or 5.

In this case, y_{n+1}, y_{n+2}, y_{n+3} and y_{n+4} are obtained simultaneously for each step. The first formula of (3.11.5) is Euler's method which is of order one as mentioned before. The second, third and fourth formulas of (3.11.5) are of order 5, and the fifth is of order 6. The local truncation errors corresponding to these formulas are

$$\frac{3h^6}{160} y_n^{(6)}, \quad \frac{h^6}{90} y_n^{(6)}, \quad \frac{3h^6}{160} y_n^{(6)} \text{ and } -\frac{8h^7}{945} y_n^{(7)}$$

respectively.

To apply the method (3.11.5), it might be use the sequence in (3.11.4), where r = 1,2,3,4.

In [45], it is noted that the process of increasing the number of block points k or of changing the stepsize is straightforward and may be implemented in a manner similar to the sequential approach. In practice, however, k would not exceed 16. The speedup ratio when k varies between 2 and 10 might vary between 1.7 and 5.9.

3.11.2 Predictor-Corrector Block Methods

Block schemes are readily adaptable to a predictor-corrector mode. The following algorithm, mentioned in [11], [45] is an example which is suitable for parallel computers with k = 2.

The predictor equations are

$$y_{n+1}^p = \frac{1}{3}\left[y_{n-2}^c + y_{n-1}^c + y_n^c \right] + \frac{h}{6}\left[f_{n-2}^c - 4f_{n-1}^c + 13f_n^c \right] \qquad (3.11.6)$$

$$y_{n+2}^p = \frac{1}{3}\left[y_{n-2}^c + y_{n-1}^c + y_n^c \right] + \frac{h}{12}\left[29f_{n-2}^c - 72f_{n-1}^c + 79f_n^c \right]$$

The corrector equations are

$$y^c_{n+1} = y^c_n + \frac{h}{12}\left[5f^c_n + 8f^c_{n+1} - f^p_{n+2} \right] \qquad (3.11.7)$$

$$y^c_{n+2} = y^c_n + \frac{h}{3}\left[f^c_n + 4f^p_{n+1} + f^p_{n+2} \right]$$

For both equations (3.11.6) and (3.11.7) the values of y_{n+1} and y_{n+2} are obtained simultaneously. The predictor equations (3.11.6) are dependent on values taken from the previous block (n,n-1,n-2). The corrector equations (3.11.7) need a single value from the previous block (n) and predicted values of current block (n+1,n+2). Both equations within each part are independent of each other. They can easily be mapped onto two processors, the processors must exchange information twice per block. Once after predictor equations and function evaluations are completed and once after the corrector equations and function evaluations are completed.

The solution of each block provides two values with the function evaluations, and y variable calculation for two performed in parallel. If we use more than two processors, then it is possible to increase the number of steps per block and change the block size such that each processor is allocated to one or more points per block. We presume this idea later; but aim to produce families of methods which may use very large k.

3.12 Multi-Block Methods

A multi-block method is one in which several blocks of points are considered simultaneously. Methods of this type are considered in [29], and

are described as k-block, r-points method. Time is divided into a series of blocks with each block containing r steps at which solutions to differential systems are to be found. The computation proceeds in blocks and is based on the computed values at the earlier k blocks.

Let $x_k = x_o + kh$ and y_k represents a numerical approximation to the exact solution $y(x_k)$ and $f_k = f(x_k,y_k)$. Denote the block number by m=0,1,2... and r the block size. Let Y_m and F_m be vectors defined by

$$Y_m = \left[y_n, y_{n+1},\dots ,y_{n+r-1} \right]^T$$

$$F_m = \left[f_n, f_{n+1},\dots , f_{n+r-1} \right]^T$$

then the general k-block, r-point method is a matrix finite difference equation of the form

$$Y_m = \sum_{i=1}^{k} A_i Y_{m-i} + h \sum_{i=0}^{k} B_i F_{m-i} \qquad (3.12.1)$$

where the A_i's and B_i's are chosen to satisfy basic accuracy and the zero-stability. Note that when r=1, then (3.12.1) is just the classical k-step method, and we say the method is explicit if $B_o = 0$; otherwise it is implicit. The analogies of predictor-corrector modes for multi-block methods are given in [29]. Then the p(EC)^{t}E mode, for example, is given by

$$\text{P:}\quad Y_m^{(0)} = \sum_{i=1}^{k^*} A_i^* Y_{m-i}^t + h \sum_{i=1}^{k^*} B_i^* F_{m-i}^{(t)} \ ;$$

$$\text{E:}\quad F_m^{(s)} = \left[f(x_n,y_n^{(s)}),\dots , f(x_{n+r-1}, y_{n+r-1}^{(s)}) \right]^T \ ;$$

$$\text{C:}\quad Y_m^{(s+1)} = \sum_{i=1}^{k} A_i Y_{m-i}^{(t)} + h \sum_{i=1}^{k} B_i F_{m-i}^{(t)} + h B_o F_m^{(s)} \text{ for } s=0,\dots , t-1,$$

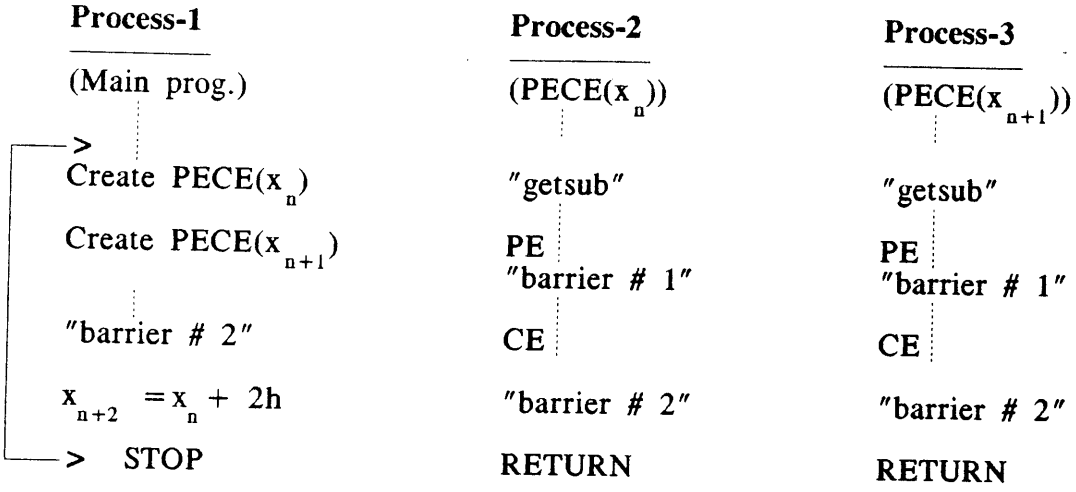
$$E: \quad F_m^{(t)} = \left[f(x_n, y_n^{(t)}), \ldots, f(x_{n+r-1}, y_{n+r-1}^{(t)}) \right]^T,$$

where the mark "*" denotes to the predicted values.

It is noted that for economical purposes, one should use a corrector in which each component has equal or one order higher accuracy than the predictor. Here the local truncation error of PECE mode is equal to the order of the corrector. The machine used was the Hep, which was described in chapter 1. The Hep supported an enhanced version of Fortran 77 which allows for parallel execution of multiple task within a single program. These extensions are implemented in Hep-Fortran, but are commonly used through higher level monitors (macros) in Argone. The macros allow for multiple task schedulling and other associated concepts useful in multiprocessing. Various tasks to be done on the Hep are referred to as "processes".

In [29], the macros "getsub" and "barrier" schedule the execution of the processes, and provide the necessary synchronization between these processes.

Consider the case when k = 2 and r = 2, where the predictor predicts at x_n and x_{n+1} and the corrector corrects at x_n and x_{n+1}. The PECE at x_n is handled by one process while PECE at x_{n+1} is handled by a second process. The main program represents a third process. The location of PECE at x_n or x_{n+1} to a process is handled by the "getsub" macro. Since the corrector step requires the function evaluation from the predictor step, the "barrier" macro is used to synchronize the process after PE has been completed by both processes. The "barrier" is again used to synchronize processes after CE has been completed. This is represented as follows:



In the above, process-2 and 3 may be viewed as subroutines running simultaneously with the main program, process-1. The "create" statements are equivalent to the FORTRAN "call" statement except the execution continues after the statement is encountered. This provides the concurrent execution of the process. The "getsub" is contained in subroutine PECE and assigns the x_n or x_{n+1} step. The "barrier" statements cause the processes to "wait" until all other processes in that category have completed their work before continuing. The "RETURN" statements cause both processes 2 and 3 to "die" off and the main program to prepare for another block of two steps. The problem tested on the HEP was

$$y' = -2y \ , \ y(0) = 1.$$

Theoretically, with three processes, there is an expected speedup of 66%, but actually only two processes are heavily loaded, hence the speedup is closer to 50%.

In [29], it is mentioned that since the algorithm is highly vectorizable, it is desirable to experiment with a multiprocessor computer which has vector instructions, such as the Cray X-Mp, which is an improved version of the Cray-1.

3.13 Multiplexing and Interpolation

A class of methods for parallelizing integration methods for initial value problems are considered in [30].

The problem considered is to integrate

$$y' = f(x,y) \quad , \quad x \in [a,b] \qquad (3.13.1)$$
$$y(a) = y_o \ .$$

The idea is to divide [a,b] into N equal subintervals $[x_{i-1},x_i]$, $x_o = a$, $x_n = b$, $i = 1,2,\ldots, N$. Then make a prediction $y_i^{(0)}$ of the solution $y(x_i)$, $i = 1,2,\ldots, N$. After that at each x_i, M_i of values y_{ij}, $j = 1,2,\ldots,M_i$ are chosen. For each i, the y_{ij} are in the vicinity of $y_i^{(o)}$. Now with some fixed numerical method M, all of the following initial value problems are to be solved simultaneously

$$y' = f(x,y) \ , \ y(a) = y_o \ , \qquad x \in [a,x_1] \qquad (3.13.2)$$
$$y' = f(x,y) \ , \ y(x_i) = y_{ij} \ , \qquad x \in [x_i,x_{i+1}] \qquad (3.13.3)$$
$$j = 1,2,\ldots, M_i; \quad i = 1,2,\ldots, N\text{-}1.$$

In the time it takes to apply the method M to (3.13.2) we solve all of the problem (3.13.2) and (3.13.3), Suppose $Y(x,y_{ij})$ is the result of solving (3.13.3), where $Y(x,y_{ij}) = y_{ij}$ for each i and j. The approximations u_i to

$y(x_i)$, i = 1,..., N are computed as follows:

u_1 is taken to be the value at x_1 calculated by applying the method M to (3.13.2), u_2 is obtained by interpolating the function $Y(x_2,y)$ of y at $y = u_1$ using the pairs $(y_{1j}, Y(x_2,y_{1i}))$, j = 1,2,..., M_1.

Similarly u_i is obtained by interpolating $Y(x_i,y)$ at $y = u_{i-1}$ using the pairs $(y_{i-1}, Y_i(x_i, y_{i-1,j}))$, j = 1,2,..., M_{i-1}, i = 3,..., N.

In [30], it is mentioned that the result will have been computed in one N^{th} the time required for the serial integration from x_o to x_N using the method M, together with the time required for the initial predictions y_i^0, for the interpolation, and for some additional bookkeeping due to the parallelism of the method. Also it is noted in [30], the methods unfortunately are inefficient in the sense that, for equal accuracy, they involve more work. As we known these methods were produced in 1964, and Nievergelt suggested that if he had powerful computer at that time, he will compute the result faster.

3.14 One-Parameter Method

A one-parameter method is mentioned in [8], for the parallel high-speed low-precision integration of a system of S ordinary differential equations. Given a fixed step size h, the integration parameter is evaluated a new at each iteration so to provide an optimal method which gives the minimum of the maximum error on S simultaneous equations and allows a complete integration step in no more than h units of time.

Consider the initial value problem for a first order system

$$\underline{y}' = \underline{f}(x,\underline{y}) \quad , \quad \underline{y}(x_o) = \underline{y}_o \tag{3.14.1}$$

where $\underline{y} = (y_1, y_2, \ldots, y_s)^t$.

Then consider

$$\underline{y}' = A\,\underline{y} \quad , \quad \underline{y}(x_o) = \underline{y}_o \tag{3.14.2}$$

where A is the Jacobian of f at a particular point, and it is s x s real matrix, where the Jacobian of f is J = df/dy . The one-step method of order one is often referred to as the "θ-method" and is given by

$$y_{n+1} - y_n = h \left[(1-\theta)\, y'_{n+1} + \theta\, y'_n \right]$$

which is A-stable if and only if $\theta \leq \frac{1}{2}$ [24]. Furthermore, the method is implicit for every non-unitary value of θ.

To minimize the maximum error over S equations, applying the θ-method to the equations (3.14.1) and (3.14.2) yields after one step of integration

$$y_{i+1} = \left[\frac{1 + \theta h\alpha}{1 - (1-\theta) h\alpha} \right] y_i \tag{3.14.3}$$

where α represents one of the distinct eigen values of A. Equation (3.14.3) represents just one of uncoupled approximate ODE solutions. Since the actual solution at $x_o + h$ is given by

$$u_1 = u_o\, e^{\alpha h} \tag{3.14.4}$$

then θ be chosen to minimize the maximum value of the error

$$e^{\alpha h} - \frac{1+\theta h\alpha}{1-(1-\theta)h\alpha}$$

The algorithm for a system S of equations consists of three tasks, first task

concerns with (3.14.4) in order to evaluate the error given by the method, estimating the actual solution which is required for every equation. Secondly, evaluate the value θ which minimizes (3.14.4). Thirdly, compute the approximate solution by using (3.14.3) and the value of θ computed by the previous task.

The particular problem solved was a system of 6 equations with the matrix A of (3.14.2) given by

$$A = \begin{bmatrix} -10 & 0 & 0 & 0 & 0 & 0 \\ 0 & -8 & 0 & 0 & 0 & 0 \\ 0 & 0 & -5 & 0 & 0 & 0 \\ 0 & 0 & 0 & -3 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.1 \end{bmatrix}$$

with a steplength of 0.01 seconds.

The algorithm was run on the transputer which was described in chapter 1. The programming language was Occam.

The optimization method first is running on a single processor which requires 1.43 seconds to carry out 100 iterations. The maximally implicit and "θ = 0.122" methods require 63.4 milliseconds.

The first task takes an average of 7.6% of the total time. The second task requires an average of 87% of the total time, the third task takes about 5.4% of the time which is spent calculating the approximate solution. So the

method spends much of its time choosing how to carry out the next step.

It is mentioned in [8], that two modifications will add a little overhead but are certainly worthwhile to reduce the total time of computation. During the first phase of the integration, check on the size of the error curves; and not all searches are required to find the optimum value of θ, but each eigen value should appear at least once in the different pairs investigated. Secondly, we need to check the magnitude of θ to make a decision on when to switch from the optimization method to the maximally implicit method. More details can be found in [8].

CHAPTER 4

The Solution of Sets of Simultaneous Linear Equations on Parallel Processors

4.1 Introduction

In this chapter we shall describe the parallel implementation in Ada multi-tasking of a linear equations solver on the Sequent Balance.

Consider the problem of solving a set of linear equations

$$A \underline{x} = \underline{y} \qquad (4.1.1)$$

where A is a square and dense matrix of size n, $\underline{x}$ and $\underline{y}$ are nx1 column vectors. Throughout this chapter, we give two algorithms to solve (4.1.1). The first algorithm uses Gauss elimination followed by back substitution with individual rows treated via parallel tasks. The second solves a block system of equations using block LU-decomposition followed by forward and backward substitution. In this algorithm the treatment of different blocks is carried out in parallel as far as possible; the individual blocks are processed serially, and the block size determines the granularity of the parallelism.

We also present the predicted and actual costs for these algorithms noting that the communication cost is associated with the computation in evaluating the predicted cost.

Finally, we give the speedups and the corresponding efficiencies.

4.2 Elimination Method: A Parallel Row Oriented Algorithm

Consider the system (4.1.1) where

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{23} & a_{33} & \cdots & a_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} \end{bmatrix} \qquad \underline{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix}$$

A of order nxn, $\underline{x}$ and $\underline{y}$ of order nx1.

For convenience of computation we perform the operations on the augmented matrix nx(n+1) by attaching the column y to the matrix A, so that y is the (n+1)st column of the system.

Consider the sequential algorithm for forward elimination

```
For i in 1..n-1 loop
  For j in i+1..n loop
  a_ji: = a_ji/a_ii;
  For k in i+1..n+1 loop
  a_jk: = a_jk-a_ji*a_ik;
  End loop;
  End loop;
End loop;
```

4.2.1 Row Oriented Algorithm

We can rewrite this algorithm using parallel multi-tasking to handle single rows in parallel, as follows:

Task required:

1) Task type forward elimination:

Accept pivot-row(i), elimination-row(j), i;

Compute the multiplier

For j>i,k>i do

a_{jk} <—— a_{jk}-a_{ji} * a_{ik};

Return row(j).

2) Main procedure:

For i in 1..n-1 loop

For j in i+1..n loop

send row(i), row(j), i to forward elimination task j

End loop;

For j in i+1..n loop

collect modified row(j);

End loop;

End loop;

The parallelism involved here is very straightforward; but we note that the number of tasks is determined wholly by n, and no control over the granularity is available.

4.2.2 Backward Substitution

Solve $U\underline{x} = a_{i,n+1}$, i=1, 2,..., n; where U is upper triangular system.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ & \dots\dots\dots\dots & & & \\ 0 & 0 & 0 & \dots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ .. \\ x_n \end{bmatrix} = \begin{bmatrix} a_{1,n+1} \\ a_{2,n+1} \\ a_{3,n+1} \\ ..... \\ a_{n,n+1} \end{bmatrix}$$

We have

$$x_n = a_{n,n+1} / a_{n,n}$$

$$x_{n-1} = a_{n-1,n+1} / a_{n-1,n-1} - \left[a_{n-1,n} / a_{n-1,n-1}\right] x_n$$

$$x_{n-2} = a_{n-2,n+1} / a_{n-2,n-2} - \left[a_{n-2,n} / a_{n-2,n-2}\right] x_n - \left[a_{n-2,n-1} / a_{n-2,n-2}\right] x_{n-1}$$

$$\vdots$$

$$x_1 = a_{1,n+1} / a_{1,1} - \left[a_{1,n} / a_{1,1}\right] x_n - \left[a_{1,n-1} / a_{1,1}\right] x_{n-1} .. \left[a_{1,2} / a_{1,1}\right] x_2$$

The procedure then takes the following form:

```
parallel  For i in reverse 1...n loop
              x_i := a_{i,n+1} / a_{i,i};
          End loop;
serial    For j in reverse 2...n loop
parallel  For i in reverse 1...j-1 loop
              x_i := x_i - (a_{i,j} / a_{i,i}) x_j;
          End loop;
          End loop;
```

Now, we present the tasks required to compute $\underline{x}$:

```
Task type backward substitution:
Accept row i,i;
```

Calculate $x_i := a_{i,n+1} / a_{i,i}$;

Collect the result;

Serial-loop j:

Accept x_j;

Calculate x_i;

Collect the result;

The main procedure is

For i in reverse 1...n loop

Send row(i),i to backward substitution task i;

Collect x_i;

End loop;

serial For j in reverse 2...n loop

parallel For i in reverse 1...j-1 loop

Send x_j to backward subst task i;

End loop;

parallel For i in reverse 1...j-1 loop

Collect x_i;

End loop;

End loop;

4.3 The Predicted Cost of Row Elimination Method

1) Forward Elimination:
Consider the matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} & a_{1,n+1} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} & a_{2,n+1} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_{nn} & a_{n,n+1} \end{bmatrix}$$

using serial machine to triangulate A requires (n-1) stages with cost approximately $n^3/3$ multiplications and additions.

We shall now describe forward elimination using the parallel algorithm. Let the matrix A as before contain n rows, and assume there is a task for each row at each stage (but not necessarily a processor for each task).

The algorithm is as follows:

Send pivot_row(i), elimination_row(j),i;

Calculate the multiplier a_{ji}/a_{ii};

which needs to be computed once only per row.

For $j > i, k > i$ do

$a_{jk} <-- a_{jk} - a_{ji} * a_{ik}$

Update the row(j).

We shall give the predicted time of the above algorithm in terms of arithmetic operations count, communication cost and data sending. We have one rendezvous to send the data and one rendezvous to receive the update, and

$3(n+1)$ data items are sent, so we have $2(n-i)$ rendezvous at each stage i; altogether $\sum_{i=1}^{n-1} ?(n-i)$ equal to $n(n-1)$ rendezvous. A total of $3(n+1)n(n-1)/2$ data items are sent which is approximately $3n^3/2$ if the whole row is sent each time. The total number of arithmetic operations is $\sum_{i=1}^{n-1} (n-i) = n^2/2$.

If we assume that the number of processors $p < n$, then the total number of arithmetic operations should be multiplied by n/p. Therefore, the number of arithmetic operations is $n^3/2p$. Assume the time to set up one rendezvous is r, the time to send one item is s and the time to do one multiplication or addition is m, then the predicted time of forward elimination is approximately

$$\left[n^3/2p \right] m + \left[n(n-1) \right] r + \left[3n^3/2 \right] s \; . \qquad (4.3.1)$$

A test program was written to measure these quantities. The estimate of m is 0.25 millisec, r = 2 millisec and s = 0.02 millisec (see appendix D, E, G).

2. Backward Substitution

We shall now evaluate the predicted cost of backward substitution.

The algorithm is as follows:

Send row(i),i

Calculate $x_i := a_{i,n+1} / a_{i,i}$ (Here x_n is computed)

Collect x_i, i = 1,..., n.

For j in n to 2 do

Send x_j (communicate x_j into each remaining tasks so as to modify the right hand side).

Compute $x_i := x_i - (a_{ij}/a_{ii})\, x_j$.

Update the result.

In the first part there are two rendezvous, one to send the data and one to receive the result, altogether 2n rendezvous. For the second part, we have one rendezvous to send x_j to n-i tasks and one to receive the result. Altogether n(n-1) rendezvous. Then the total rendezvous is n(n+1). For the first part we have n+2 data items which are sent and 2 data items which are sent for the second part. Altogether n(2n+1) data items are sent.

Finally, the arithmetic operations for the first part is n/p, and for the second part is n(n-1)/p. In case that the number of processors $p < n$, then the arithmetic operations is approximately n^2/p.

Thus, the predicted time for backward substitution is

$$[n^2/p]m + [n(n+1)]r + [n(2n+1)]s. \qquad (4.3.2)$$

4.4 Test Problem

The following test problem [20] solves the system $A\,\underline{x} = \underline{y}$, where the coefficients of the matrix A are given by

$a_{1j} = 1.0, \quad i = 1,\ldots, n$,

$a_{ii} = 1.0, \quad i = 2,\ldots, n$,

$a_{i1} = \text{float}(i)$, $i = 2,\ldots, n$,

$a_{nj} = 1.0$, $i = 2,\ldots, n\text{-}1$.

The right hand side is given by

$a_{i,n+1}$ = float (2i) , i = 2,..., n-1 ,

$a_{1,n+1}$ = float (n(n+1)/2) ,

$a_{n,n+1}$ = float ((n+2)(n-1)/2 + n).

The solution is $\underline{x}^T$ = (1,2,3,...,n).

4.5 The Computation Results

We now present the computation results of the system which were mentioned in (4.4). Table 4.1 shows the actual and predicted costs (with s = 0) for the parallel row elimination, where the predicted costs are found using (4.3.1) and (4.3.2). Denote that the actual and predicted costs by t(A) and t(P) respectively. The program was run for p = 1,2,3,..., 7 for each value of n.

Table 4.5.1

p\n	60	80	120	140	160	t(A)or t(P)
1	55.4	115.4	355.5	585.6	880.9	t(A)
	42.3	91.2	227.2	426.3	620.8	t(P)
2	38.0	75.7	215.5	332.6	496.5	t(A)
	28.3	58.4	167.4	252.3	361.6	t(P)
3	33.6	64.4	174.4	264.5	388.6	t(A)
	23.7	47.5	130.8	194.3	275.2	t(P)
4	32.3	61.3	158.2	238.8	346.3	t(A)
	21.3	42.0	112.5	165.3	232.0	t(P)
5	33.7	60.5	151.8	228.4	327.9	t(A)
	20.0	38.7	101.5	148.0	206.1	t(P)
6	34.7	61.1	153.2	227.1	326.5	t(A)
	19.0	36.5	94.2	136.3	188.8	t(P)
7	35.2	63.8	156.7	232.3	332.9	t(A)
	18.3	35.0	89.0	128.1	176.5	t(P)

where the time is measured in seconds.

The above table illustrates the comparison of the actual and predicted times. We notice that the differences between them, which might be expected when the times of the data transfer are neglected in the predicted cost, are high. We also observe that for fixed n the actual time decreases to four processors when n is small, then increases slightly when $p > 4$. But if n is large, the actual time decreases to six processors.

If we take account of the quantity of data sending, then the above predicted time will be increased. So from figure 4.6.3, where $n = 120$ we compute s as follows:

$$\left[3\ n^3/2 + n(2n + 1) \right] * s = 52.4 \text{ seconds} \qquad (4.5.1)$$

Note that the quantity in the left-hand side is the whole data sending in both forward elimination and back substitution. Then (4.5.1) becomes

$$\left[3\ (120)^3/2 + (120)(241) \right] * s = 52.4 \text{ seconds}$$

gives

$$s = 0.02 \text{ milliseconds.}$$

This is an evidence to ensure that the experimental test on estimating the time of sending single item is correct.

Taking data transfer into account using this figure, the predicted times are shown by the table below.

Table 4.5.2

p \ n	60	80	120	140	160
1	48.9	106.8	329.6	509.4	744.7
2	34.9	74.0	219.8	335.4	485.5
3	30.3	63.1	183.2	277.4	399.1
4	27.9	57.6	164.9	248.4	355.9
5	26.6	54.3	153.9	231.1	330.0
6	25.6	52.1	146.6	219.4	312.7
7	24.9	50.6	141.4	211.2	300.4

From the above table, we observe that the predicted times are improved and the differences between the predicted times in table 4.5.2 and the actual times in 4.5.1 are slightly better than the diferences between the predicted and the actual times in table 4.5.1. However, there is a significant change

in the predicted times when we add the time of the data sending to the computation cost.

The following table shows the predicted times as a percentage which are required for the computation, rendezvous and data sending for n = 60, 120 and 160 and p = 1,2 and 5.

Table 4.5.3

n	Number of Processors	Computation Time	Rendezvous Time	Data Sending Time
60	1	57.1 %	29.4 %	13.5 %
	2	39.8 %	41.3 %	18.9 %
	5	20.9 %	54.2 %	24.8 %
120	1	66.6 %	17.5 %	15.9 %
	2	49.9 %	26.2 %	23.8 %
	5	28.5 %	37.4 %	34.0 %
160	1	69.6 %	13.8 %	16.6 %
	2	53.4 %	21.1 %	25.5 %
	5	31.4 %	31.0 %	37.5 %

From the above table, we notice that the computation requires much more time when p = 1, but the time decreases by increasing the number of processors. However, the communication times which consist of the rendezvous and data sending times, will have more influence when the number of processors is more than one.

4.6 Speed Up and Efficiencies

Table 4.6.1 shows the actual and predicted speedups and efficiencies for the values of n equal to 60, 80, 120, 140 and 160. The predicted time are those which are obtained from table 4.5.2.

Table 4.6.1

n \ p	1	2	3	4	5	6	7	Actual or Predicted
60	1.0	1.46	1.65	1.75	1.64	1.60	1.57	S(A)
	1.0	0.73	0.55	0.43	0.33	0.27	0.22	E(A)
	1.0	1.40	1.61	1.75	1.84	1.91	1.96	S(P)
	1.0	0.70	0.54	0.44	0.37	0.32	0.28	E(P)
80	1.0	1.52	1.79	1.88	1.91	1.89	1.81	S(A)
	1.0	0.76	0.59	0.47	0.38	0.31	0.25	E(A)
	1.0	1.44	1.69	1.85	1.97	2.05	2.11	S(P)
	1.0	0.72	0.56	0.46	0.39	0.34	0.30	E(P)
120	1.0	1.65	2.04	2.25	2.34	2.32	2.27	S(A)
	1.0	0.82	0.68	0.56	0.47	0.39	0.32	E(A)
	1.0	1.50	1.80	2.00	2.14	2.25	2.33	S(P)
	1.0	0.75	0.60	0.50	0.43	0.37	0.33	E(P)
140	1.0	1.76	2.21	2.45	2.56	2.58	2.52	S(A)
	1.0	0.88	0.74	0.61	0.51	0.43	0.36	E(A)
	1.0	1.52	1.84	2.05	2.20	2.32	2.41	S(P)
	1.0	0.76	0.61	0.51	0.44	0.39	0.34	E(P)
160	1.0	1.77	2.27	2.54	2.69	2.70	2.65	S(A)
	1.0	0.89	0.76	0.63	0.54	0.45	0.38	E(A)
	1.0	1.60	1.87	2.09	2.26	2.38	2.48	S(P)
	1.0	0.80	0.62	0.52	0.45	0.40	0.35	E(P)

We notice that the actual speedup increased by increasing the

calculation performed on each processor. In the case when n = 120, p varies between 2 and 5, the actual speedup increases from 1.65 to 2.32 and the corresponding efficiency decreases from 0.82 to 0.47 which is around the predicted value. Also we observe that the time spent on communication is not very cheap compared to the computation time and will affect the speedup.

Figures 4.6.1 to 4.6.5 show the actual and predicted times plot for the number of processors p.

The following notation is used in the figures.

t(A): The actual time.

t(P): The predicted time in terms of arithmetic operations and the time of the rendezvous.

t'(P): The predicted time in terms of arithmetic operation, the time of the rendezvous and the time of data distribution.

Figure 4.6.6 shows the behaviour of the actual time as n increases for fixed P.

1: t(P), 2: t′(P), 3: t(A)

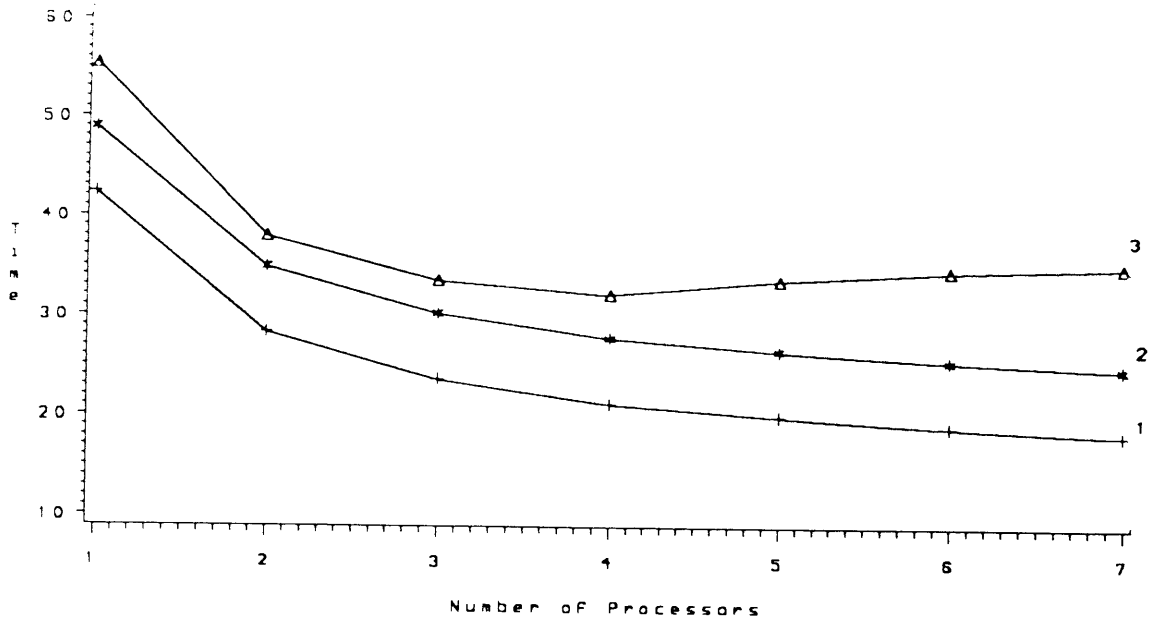


Figure 4.6.1__ n=60

1: t(P), 2: t′(P), 3: t(A)

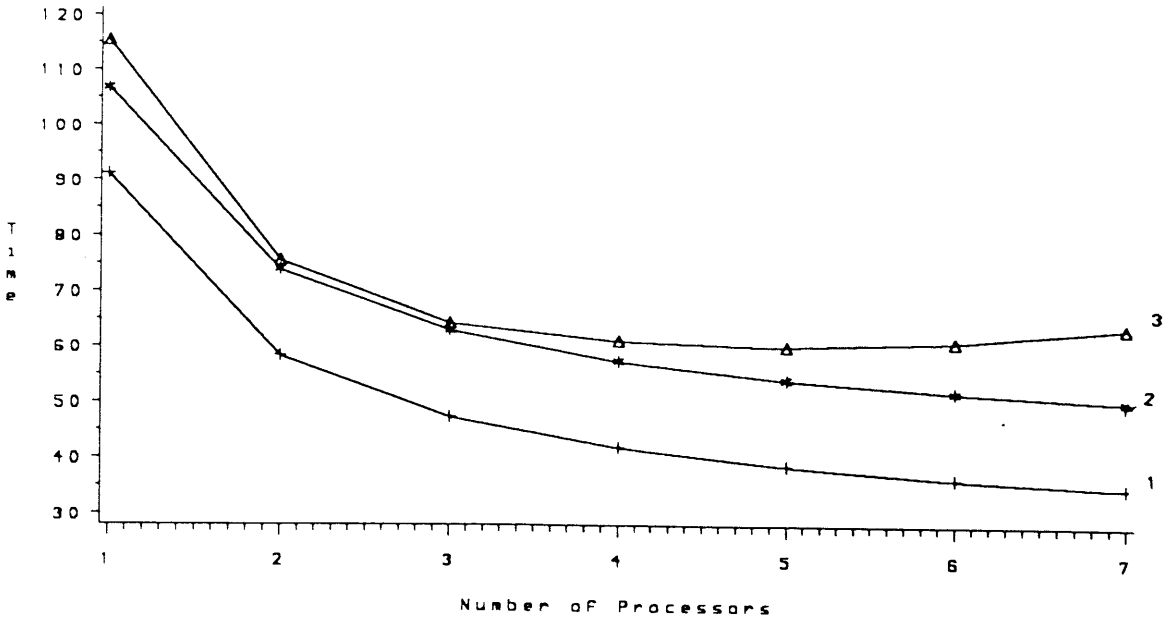


Figure 4.6.2__ n=80

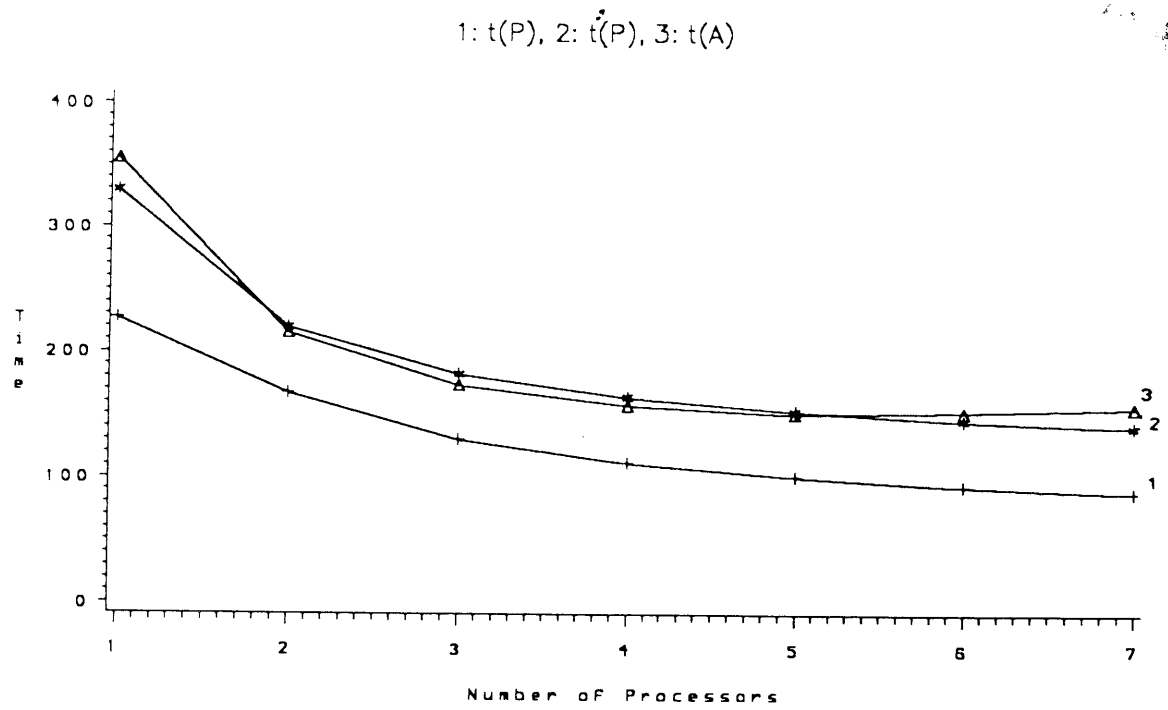


Figure 4.6.3_ n=120

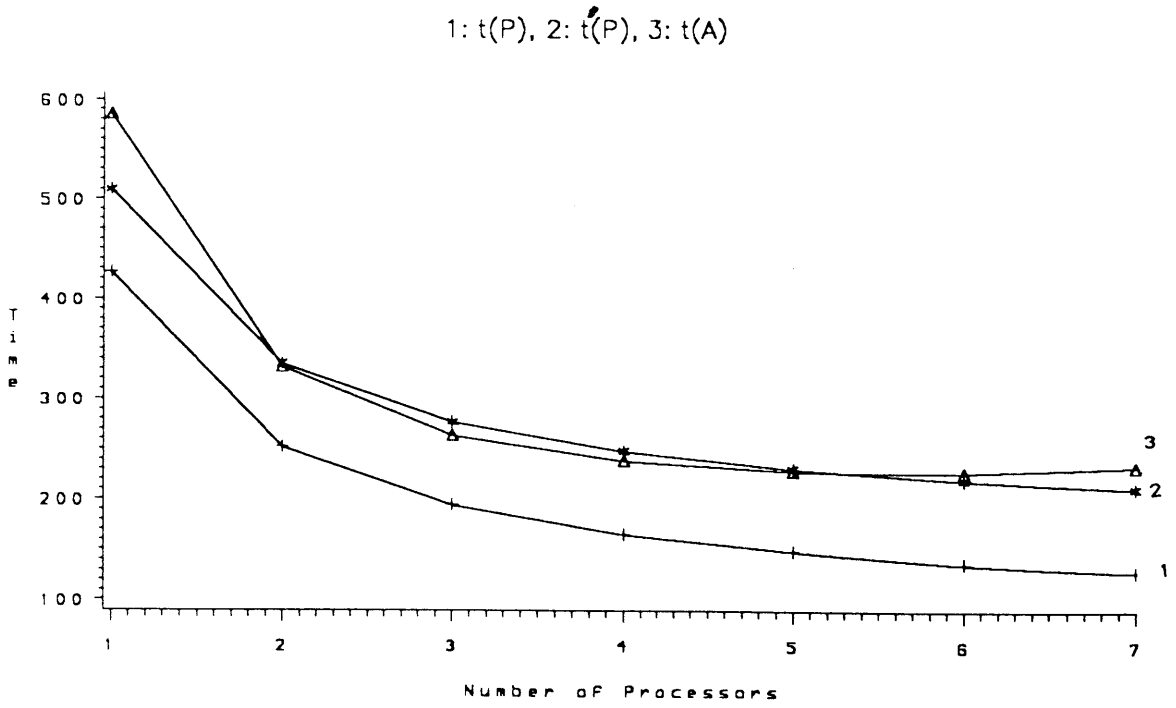


Figure 4.6.4_ n=140

1: t(P), 2: t'(P), 3: t(A)

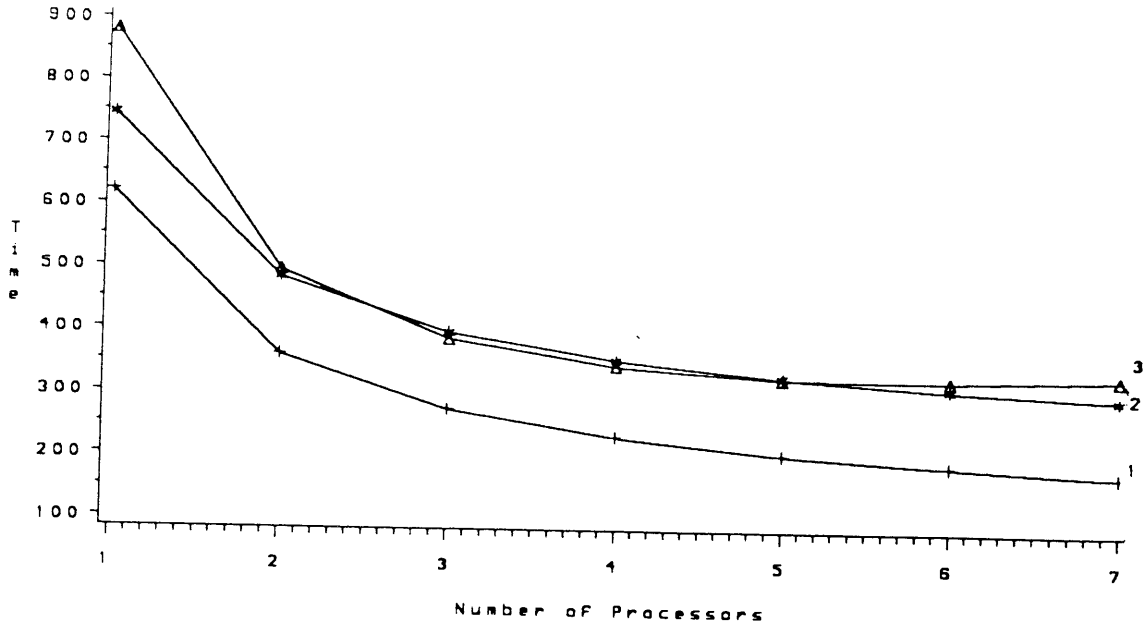


Figure 4.6.5__ n=160

1: One Processor
2: Two Processors
3: Three Processors

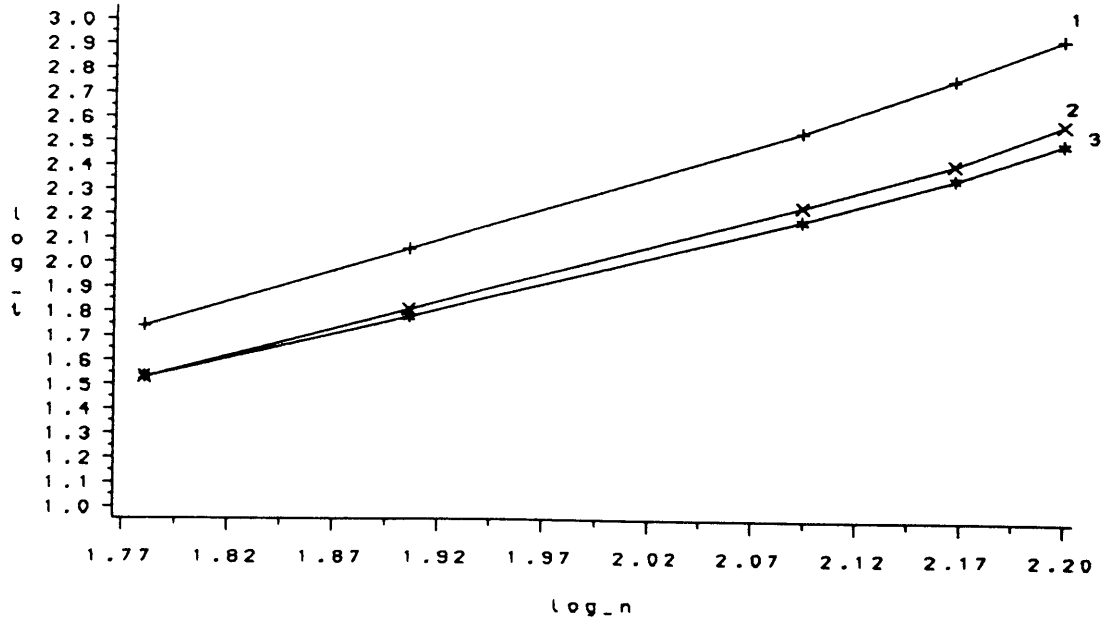


Figure 4.6.6

4.7 Solving Block System of Equations by Block LU Decomposition followed by Forward and Backward Substitution

Alternatively, we consider an algorithm based on sectioning the matrix into blocks.

Consider the system in (4.1.1) where A of order n*n and $\underline{X}$, $\underline{Y}$ are column vectors n*1. Partition A into q^2 of blocks order wxw each.

Also partition the vectors $\underline{X}$, $\underline{Y}$ into q blocks of order wx1, where q=n/w. In Ada, the matrix A may be stored in block_row (1..q) of blocks. $\underline{Y}$ stored in block_vector (1..q) of vectors.

4.7.1 LU Decomposition

Consider the sequential algorithm for block LU_decomposition without pivoting:-

```
For i in 1..q-1 loop
   Transpose (A_ii,pivot);      - store transpose of A_ii in pivot
                                  variable
   Triangulate (pivot);
For j in i+1..q loop
   Transpose (A_ji, Elim);      - store transpose of A_ji in Elim
                                  variable
   Solve (pivot, Elim, Mult);
   A_ji := Mult;
```

```
For k in i+1..q loop
  A_jk := A_jk - Mult * A_ik;
End loop;
End loop;
End loop;
```

As mentioned earlier, the individual blocks are processed serially. So, we need the following procedures:

1. Procedure to transpose a block A;
2. Procedure to triangulate a block A;
3. Procedure to multiply two blocks;
4. Procedure to subtract two blocks;
5. Procedure to solve B A = C where A and C are given. Where the block B denotes to the multiplier block.

We shall describe the computational scheme for evaluating the matrix B as follows:

First, transpose both sides of B A = C, we get

$$A^T B^T = C^T$$

Let A^T = pivot and C^T = Elim. Second, factoring the pivot into L, lower triangular matrix and U, upper triangular matrix, that is pivot = LU.

The process involves writing the system $A^T B^T = C^T$ as L U B^T = C. So, we solve

$$L\ C = C \quad \text{and} \quad U\ B^T = C.$$

To solve the system L C = C, we have

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ a_{21} & 1 & 0 & 0 & 0 \\ a_{31} & a_{32} & 1 & & 0 \\ \vdots & \vdots & a_{43} & & \\ a_{w1} & a_{w2} & a_{w3} & a_{w,w-1} & 1 \end{bmatrix} \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1w} \\ c_{21} & c_{22} & \cdots & c_{2w} \\ \vdots & \vdots & & \vdots \\ c_{w1} & c_{w2} & & c_{ww} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & \cdots & c_{1w} \\ c_{21} & c_{22} & c_{32} & \cdots & c_{2w} \\ \vdots & \vdots & \vdots & & \vdots \\ c_{w1} & c_{w2} & c_{w3} & & c_{ww} \end{bmatrix}$$

where w is the size of these blocks.

From the above scheme, we have

```
For i in 2..w loop
  For j in 1..w loop
  Sum = 0.0;
  For k in 1..i-1 loop
  Sum = sum + a_ik * c_kj
  End loop;
  c_ij := c_ij - sum;
  End loop;
End loop;
```

Now solve $U B^T = C$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1w} \\ 0 & a_{22} & a_{32} & & a_{2w} \\ \vdots & 0 & a_{33} & & \vdots \\ & \vdots & 0 & & \\ 0 & 0 & 0 & & a_{ww} \end{bmatrix} \begin{bmatrix} b_{11} & b_{21} & \cdots & b_{w1} \\ b_{12} & b_{22} & & b_{w2} \\ b_{13} & b_{23} & & \vdots \\ \vdots & \vdots & & \\ b_{1w} & b_{2w} & & b_{ww} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1w} \\ c_{21} & c_{22} & & c_{2w} \\ c_{31} & c_{23} & & c_{3w} \\ \vdots & \vdots & & \vdots \\ c_{w1} & c_{2w} & & c_{ww} \end{bmatrix}$$

which gives,

For i in 1..w loop

$b_{iw} := c_{wi} / a_{w,w}$;

End loop;

For i in reverse 1..w-1 loop

For j in 1..w loop

sum:= 0.0;

For K in reverse i+1..w loop

sum:= sum + $a_{i,k}$ * $b_{j,k}$;

End loop;

End loop;

$b_{ji} := (c_{ij}$ - sum)/a_{ii} ;

End loop;

Then we use the results in eliminating blocks in the other columns.

To get new blocks, we use

A_{jk} <—— A_{jk} - Mult * A_{jk} , i = 1,..., q-1,

j = i + 1,..., q and

k = i + 1,..., q.

Assume that q processors are available, then the update blocks in each block row can be carried in parallel, on this basis, we need a task type to compute the new blocks as follows:

1) Task type (elim block:= elim block - Mult x pivot block):

Accept A,P,B: in blocks;

Compute A:= A - P * B;

Return A;

The major task type for LU-decomposition is

2) Task type (LU-decomposition);

Serial loop

Accept row (i), row(j), i ;

Call procedure to transpose A_{ii};

Call procedure to triangulate A_{ii};

Call procedure to transpose A_{ji};

Solve A_{ii}^T * $Mult^T$:= A_{ji}^T;

Call the above task (1) to update the blocks;

Collect the results;

Return row(j);

The main procedure is

For i in 1...q-1 loop

For j in i + 1...q loop

Send row(i), row(j), i to LU-decomposition task j;

End loop;

For j in i + 1...q loop

Collect the updated row(j);

End loop

End loop;

4.7.2 Forward and Backward Substitution

1) Forward Substitution:

Solve L$\underline{Z}$:= $\underline{Y}$;

$$\begin{bmatrix} I & 0 & 0 & 0 & 0 \ldots & 0 \\ A_{21} & I & 0 & 0 & 0 \ldots & 0 \\ A_{31} & A_{32} & I & 0 & 0 \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & 0 \\ A_{q1} & A_{q2} & \ldots & \ldots & A_{q,q-1} & I \end{bmatrix} \begin{bmatrix} Z_1 \\ Z_2 \\ Z_3 \\ \vdots \\ Z_q \end{bmatrix} = \begin{bmatrix} Y_1 \\ Y_2 \\ Y_3 \\ \vdots \\ Y_q \end{bmatrix}$$

We have

$Z_1 := Y_1$

$Z_2 := Y_2 - A_{21} * Z_1$

$Z_3 := Y_3 - A_{31} * Z_1 - A_{32} \times Z_2$

.

.

$Z_q := Y_q - A_{q1} * Z_1 - A_{q2} \times Z_2 - \ldots - A_{qq-1} * Z_{q-1}$

Then we can write

```
Parallel   For i in 1..q loop
             Z_i := Y_i;
             End loop;
Serial     For i in 1..q-1 loop
Parallel   For j in i+1..q loop
             Z_j := Z_j - A_ji * Z_i;
             End loop;
             End loop;
```

In this case, we need the following procedures:

1) Procedure to multiply block by block vector,

2) Procedure to subtract two block vectors.

To compute $\underline{Z}$ using $Z_j = Z_j - A_{ji} * Z_i$, i = 1,..., q-1; j = i+1,...,q, we assume that q processors are available, then $\underline{Z}$ can be calculated in parallel using the following task type:

1) Task type (Vector:=Vector - Matrix * Vector),

   Accept X,Y: in vector; A: in matrix;

   Compute X:= X-A*Y;

   Return X;

The forward substitution task type is given by

2) Task type (Forward Substitution):

   1. Evaluate the first column of intermediate vector;

      Accept Y_i; Return Z_i;

   2. Update the vector Z;

      Serial_loop

      Accept Z_j,Z_i,A_{ji};

      Call task(1);

      Collect the result;

      Return vector Z_j;

Then the main procedure takes the following form:

```
Parallel   For i in 1..q loop
           Send Y_i; Collect Z_i;
           End loop;
Serial     For i in 1..q-1 loop
Parallel   For j in i+1..q loop
              Send Z_j,Z_i,A_ji to forward subst task j;
              End loop;
```

Parallel For j in i+1..q loop

Collect updated Z_j;

End loop;

End loop;

2) Backward Substitution

Finally, we solve the system U $\underline{X}$ = $\underline{Z}$ for $\underline{X}$. Consider,

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & \dots & A_{1q} \\ 0 & A_{22} & A_{23} & \dots & A_{2q} \\ 0 & 0 & A_{33} & \dots & A_{3q} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \dots & A_{qq} \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ \\ X_q \end{bmatrix} = \begin{bmatrix} Z_1 \\ Z_2 \\ Z_3 \\ \\ Z_q \end{bmatrix}$$

We have

$$A_{qq} * X_q = Z_q$$

$$A_{q-1q-1} * X_{q-1} = Z_{q-1} - A_{q-1q} * X_q$$

.

.

$$A_{11} * X_1 = Z_1 - A_{1q} * X_q - \dots - A_{12} * X_2.$$

Then we can write:

Triangulate A_{qq};

Solve forward & backward Substitution for X_q;

Serial For i in reverse 2..q loop

Parallel For j in reverse 1..i-1 loop

$Z_j = Z_j - A_{ji} * X_i$;

End loop;

Triangulate A_{i-1i-1};

Solve Forward & Backward substitution for X_{i-1};

End loop;

We may summarize the above procedure as follows:

1) Compute $\underline{X}_q$

2) The expression $Z_j = Z_j - A_{ji} * X_i$, for i =q, j = i-1,..., 1 can be calculated in parallel.

3) Compute $\underline{X}_{q-1}$, then compute $Z_j = Z_j - A_{ji} * X_i$ for i = q-1, j = i-1,...,1.

4) This procedure continues and in general ....

The calculation of $\underline{X}_1$ completes the calculation of the vector X. Now, we shall give the task required for backward substitution.

1) Task type (Vector=Vector - Matrix * Vector):

As mentioned in forward substitution.

2) Task backward substitution

Accept block_vector Z;

Triangulate A_{qq};

Solve forward & backward substitution for X_q;

Serial_loop i:

Call task (1);

Collect the result;

Triangulate A_{i-1i-1};

Solve forward & backward substitution for X_{i-1};

End loop;

Return block_vector X;

The main procedure is:

Send block_vector Z to backward substitution task;

Collect the solution X;

4.8 The Predicted Cost of Block System

1) LU_decomposition:

Consider the matrix A of order nxn and partition into q^2 blocks as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1q} \\ A_{21} & A_{22} & A_{23} & \cdots & A_{2q} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ A_{q1} & A_{q2} & A_{q3} & \cdots & A_{qq} \end{bmatrix}$$

which contains q block_rows, and each block_row contains q blocks of order wxw, where w = n/q.

The Algorithm to triangulate the matrix A is as follows:

Serial loop

Send block_row(i), block_row(j),i;

Transpose (A_{ii},pivot);

Triangulate(pivot);

Transpose(A_{ji},Elim);

Calculate the multiplier A_{ji};

Call task to compute;

A_{jk} <—— A_{jk} - A_{ji} * A_{ik};

Collect the result;

Collect the update block_row(j);

End loop;

As before, we shall give the predicted time of the above algorithm in terms of arithmetic operations count, communication cost and data sending. Here we have one rendezvous to send the data and one rendezvous to receive the update. So we have 2(q-i) rendezvous at each stage i. And $3q^2(q-1)w^2/2$ data items are sent. On the other hand, there is one rendezvous to send three blocks and one rendezvous to receive the update block. So we have $\sum_{i=1}^{q-1} 2(q-i)^2$ equal to $q(q-1)(2q-1)/3$ rendezvous and $2q(q-1)(2q-1)w^2/3$ data items are sent.

Table 4.8.1 describes the number of multiplication and addition, and the number of steps which are required for each operation; it is assumed that there are at least q processors available.

Table 4.8.1

Operations	**No. of Multiplications and Additions**	**No. of Steps**
Transpose A_{ii}	-	(q-1)
Transpose A_{ji}	-	(q-1)
Triangulate A_{ii}	$w^3/3$	(q-1)
Compute the multiplier	$2w^3$	(q-1)

Compute:

A_{jk} <—— A_{jk} - M * A_{ik} $2w^3$ q(q-1)/2

If the number of processors p < q, then the predicted time for LU_Decomposition is approximately

$[2q(q-1)(q+1)]$ r/3 + $[q(q-1)(17q-4)w^2]$ s/6 + $[(q-1)(3q+7)w^3q]$ m/3p (4.8.1)

where r is the time to set up one rendezvous, s the time to send one item and m the time to do one multiplication or addition.

2) Forward Substitution

The first part of the algorithm is:

Send the right hand side Y_i; Return Z_i

The second part is:

Serial_loop

Call a task to compute:

Z_j ——> Z_j - A_{ji} * Z_i;

Collect the result;

Update the vector Z_j;

In the first part we have one rendezvous to send the right hand side and one to collect the result, and two data items are sent each time. This gives qr + 2qws.

On the other hand, there is one rendezvous to send Z_j,Z_i,A_{ji} and one to receive Z_j, and four data items are sent, three of them are vectors of order w * 1, and one block of order w * w. The same quantity required for the

calling task. This gives $[2q(q-1)]r + [q(q-1)(3+w)w]s$.

Finally the algorithm requires (q-1) steps to update the vector Z. Then the predicted time for forward substitution is approximately

$$[q(2q-1)]r + [qw(3q-1) + (q-1)w^2]s + [q(q-1)w^2]m/p. \qquad (4.8.2)$$

3) Backward Substitution

The algorithm is as follows:

Send the right hand side block_vector Z;

Triangulate A_{qq};

Compute X_q;

Serial_loop

Call task to compute:

Z_j ——> $Z_j - A_{ji} * X_i$;

Collect the result.

Triangulate A_{i-1i-1};

Compute X_{i-1};

Receive the solution X.

Here we have one rendezvous to send Z and one rendezvous to collect X, and 2q data items are sent. This gives $2r + (2qw)s$. Also we have q(q-1) rendezvous, and four data items are sent. So the total cost is

$[q(q-1)]r + [q(q-1)(3+w)w]s/2$.

The number of multiplications and additions together with number of steps are shown in the following table 4.8.2:

Table 4.8.2

Operations	No. of Multiples and Additions	No. of Steps
Triangulate A_{qq}	$w^3/3$	1
Compute X_q	$2w^2$	1
Update Z_j	w^2	(q-1)
Triangulate $A_{i-1,i-1}$	$w^3/3$	(q-1)
Compute X_{i-1}	$2w^2$	(q-1)

Then the total number of arithmetic operations is

$(w^3/3)q + w^2(3q-1)$

The predicted time for backward substitution is approximately

$$[q(q-1)+2]\ r + [q(q+3)w/2 + 3q(q-1)w^2/2]\ s + q[qw^3/3+w^2(3q-1)]m/p. \quad (4.8.3)$$

4.9 Computational Results

In this section we present the actual and predicted time, speedup and efficiencies for values of n equal to 60, 80 and 100.

Table 4.9.1: n=60

q	p	t(A)	t(P)	S(A)	S(P)	E(A)	E(P)
10	1	65.6	69.0	1.0	1.0	1.0	1.0
	2	37.5	36.4	1.75	1.89	0.88	0.95
	3	28.8	25.5	2.28	2.71	0.76	0.90
	4	25.0	20.1	2.62	3.43	0.66	0.85
	5	23.3	16.4	2.82	4.21	0.56	0.84
	6	22.2	14.7	2.95	4.70	0.49	0.78
	7	22.1	13.1	2.97	5.30	0.42	0.75
12	1	71.9	69.6	1.0	1.0	1.0	1.0
	2	40.9	37.6	1.75	1.85	0.88	0.93
	3	31.3	26.9	2.29	2.59	0.77	0.87
15	1	82.9	71.7	1.0	1.0	1.0	1.0
	2	47.2	40.3	1.76	1.78	0.88	0.89
	3	36.0	29.8	2.30	2.41	0.77	0.80
20	1	108.4	78.7	1.0	1.0	1.0	1.0
	2	61.1	47.8	1.77	1.63	0.89	0.82
	3	46.6	37.5	2.33	2.09	0.78	0.69

The above table shows the actual and predicted time, speedups and efficiencies for n = 60, when q equal to 10, 12, 15, 20. We observe that the actual time decreases when the number of processors increases. However, for block LU_decomposition more than five processors used and the actual time

continue to decrease, and this is the expected behaviour for the actual time when p > 1. We again observe, for fixed n and variable q of the actual time increases by increasing q. Clearly, a large q requires much more computation time.

There are essentially no differences between the predicted time and the corresponding actual time when the number of processors used is 3, but the differences increase slightly and reach the maximum value when p = 7. In this case, when n = 60, q = 10 and p = 7, the difference is equal to 9.0 seconds. But the actual speedup increases from 1.75 to 2.97 when p = 2 and p = 7 and the efficiency decrease from 0.88 to 0.42.

The following table illustrates the results for n = 80 when q is equal to 16 and 20.

Table 4.9.2: n=80

q	p	t(A)	t(P)	S(A)	S(P)	E(A)	E(P)
16	1	164.3	158.9	1.0	1.0	1.0	1.0
	2	88.7	85.7	1.85	1.85	0.93	0.93
	3	64.7	61.3	2.54	2.59	0.85	0.86
	4	53.2	49.1	3.09	3.24	0.77	0.81
	5	47.2	41.8	3.46	3.80	0.69	0.76
	6	44.5	36.9	3.69	4.31	0.62	0.72
	7	42.8	33.4	3.84	4.76	0.55	0.68
20	1	189.4	164.2	1.0	1.0	1.0	1.0
	2	102.0	92.1	1.85	1.78	0.93	0.89
	3	74.4	68.1	2.55	2.41	0.85	0.80

The attempts to enlarge the value n will affect the actual time. This is a fortunate consequence and gives a better speedup. In this case, when q = 16 the actual speedup increases from 1.85 to 3.84 when p = 2 and 7.

The table below shows the results for n = 100 when q is equal to 20 and 25.

Table 4.9.3: n=100

q	p	t(A)	t(P)	S(A)	S(P)	E(A)	E(P)
20	1	314.8	303.1	1.0	1.0	1.0	1.0
	2	165.9	163.6	1.90	1.85	0.95	0.93
	3	118.2	117.0	2.66	2.59	0.89	0.86
	4	95.6	93.8	3.29	3.23	0.82	0.81
	5	83.2	79.8	3.78	3.79	0.76	0.76
	6	76.4	70.5	4.12	4.30	0.69	0.72
	7	72.6	63.9	4.34	4.74	0.62	0.68
25	1	363.9	313.9	1.0	1.0	1.0	1.0
	2	192.1	176.2	1.89	1.78	0.95	0.89
	3	138.7	130.2	2.62	2.41	0.87	0.80

From the above table we observe that the behaviour is similar to that of the previous tables. Exactly the same remarks apply to table 4.9.3.

The following tables give the predicted times as a percentage of the total time for the computation, rendezvous and data distribution.

Table 4.9.4: n=60

q	p	Computation Time	Rendezvous Time	Data Distribution Time
10	1	94.4%	2.8%	2.8%
	2	89.6%	5.2%	5.2%
	3	85.0%	7.5%	7.5%
	7	70.9%	14.5%	14.5%
20	1	78.5%	16.4%	5.1%
	2	64.6%	26.9%	8.4%
	3	54.9%	34.4%	10.7%
	7	34.2%	50.2%	15.6%

Table 4.9.5: n=100

q	p	Computation Time	Rendezvous Time	Data Distribution Time
20	1	92.1%	4.2%	3.7%
	2	85.3%	7.9%	6.8%
	3	79.5%	11.0%	9.5%
	7	62.4%	20.2%	17.4%
25	1	87.8%	7.8%	4.4%
	2	78.2%	13.9%	7.9%
	3	70.5%	18.8%	10.7%
	7	50.6%	31.5%	17.9%

Tables 4.9.4 and 4.9.5 show the predicted times for n = 60, q = 10 and 20 and for n = 100, q = 20 and 25 respectively when p = 1,2,3,7.

From the above tables, we observe that the rendezvous time dominates the computation time when q is large enough and w is small. Also the time taken to send and collect the results goes up as more processors are used. However, the communication cost is too large when compared to the cost for the computation.

The following figures 4.9.1, 4.9.2 and 4.9.3 show the actual and predicted times plot for the number of processors.

The figures 4.9.4 and 4.9.5 show the speedups and efficiencies plot for the number of processors respectively.

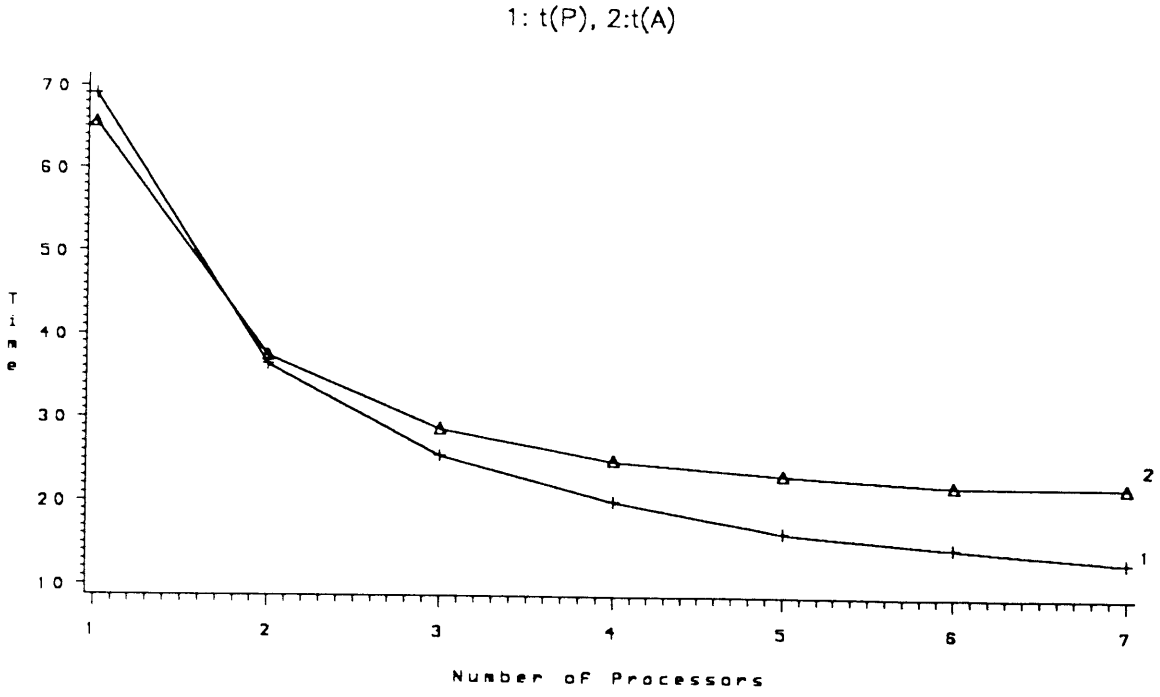


Figure 4.9.1 _ n=60, q=10, w=6

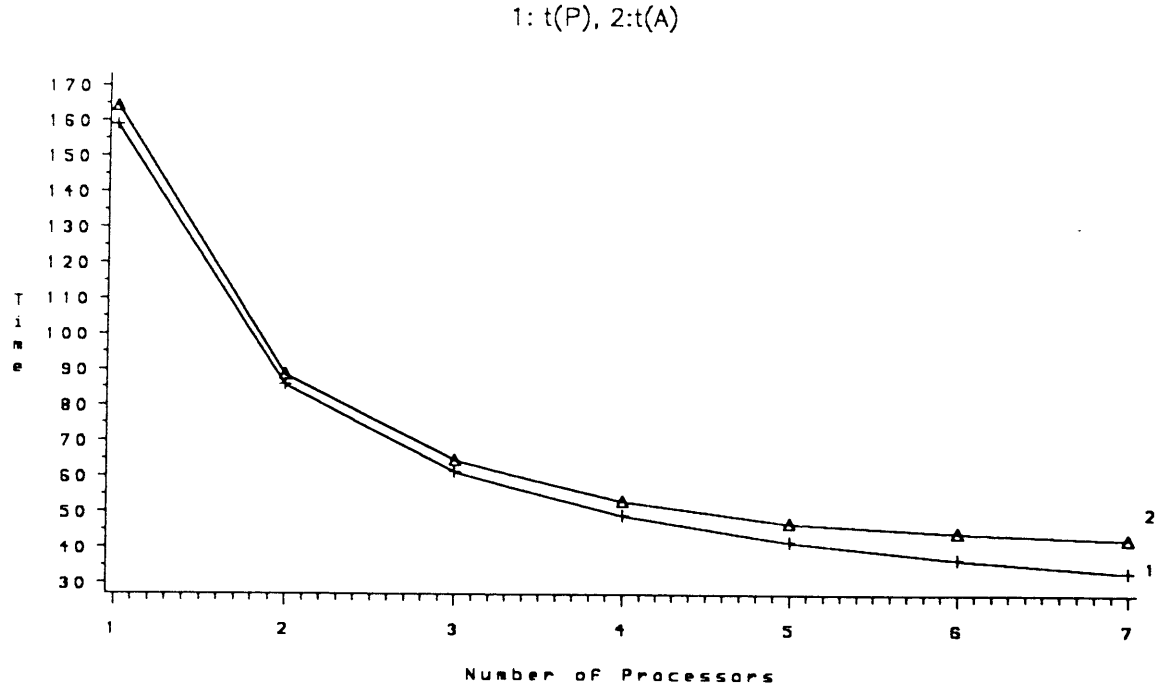


Figure 4.9.2 _ n=80, q=16, w=5

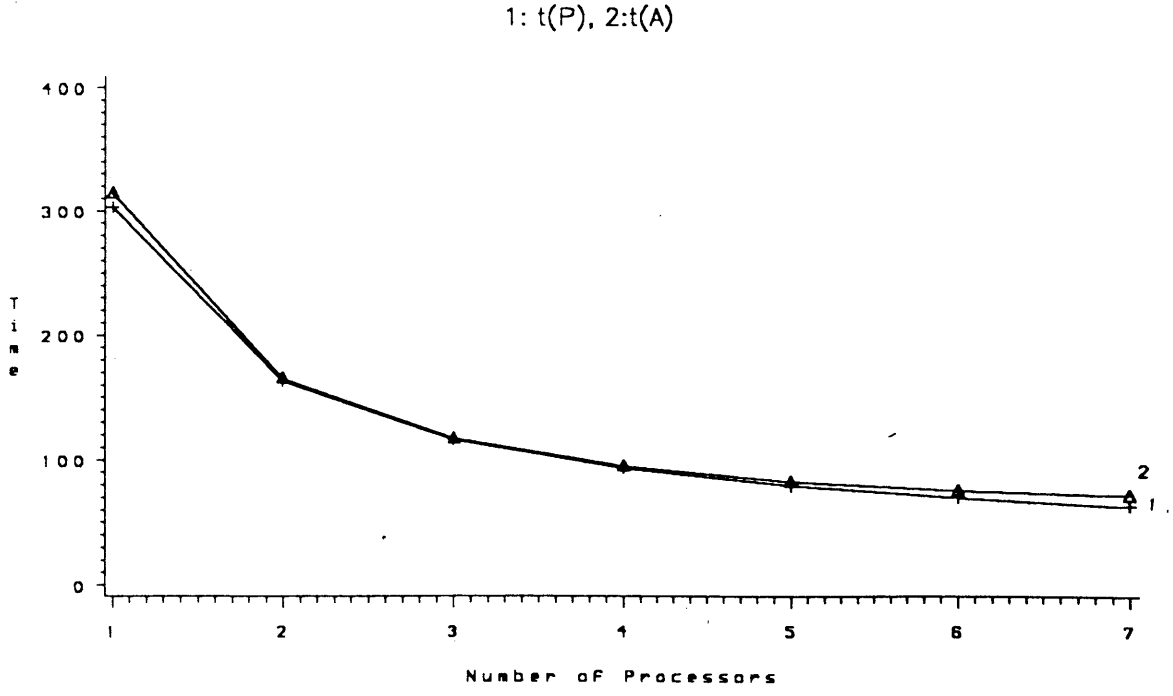


Figure 4.9.3 _ n=100, q=20, w=5

1: n=60, q=10
2: n=80, q=16
3: n=100, q=20

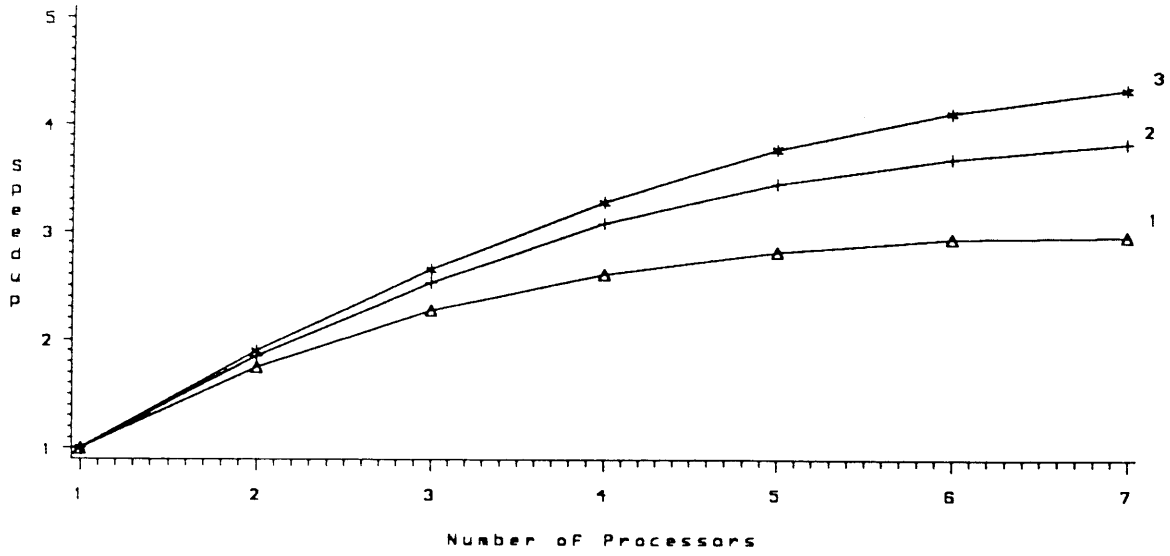


Figure 4.9.4

1: n=60, q=10
2: n=80, q=16
3: n=100, q=20

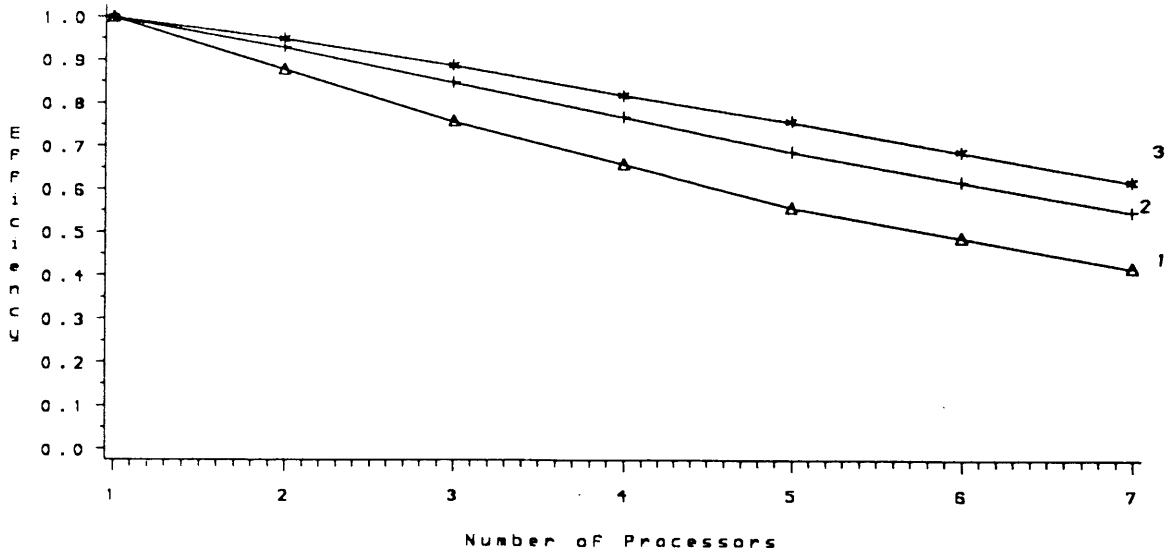


Figure 4.9.5

4.10 Optimum Number of Processors

The total estimated for LU decomposition is

$$T = (2q^3/3 + 3q^2 - 8q/3 + 2)r + (17q^3w^2/6 + 7q^2w/2 + qw/2 - 4qw^2/3 - w^2 - 2q^2w^2 - 4qw^2/3)s + (q^3w^3 - q^2w^3 - 7qw^3/3 - 2qw^2 + 4w^2q^2)m/p \quad (4.10.1)$$

where r, s and m denote the time required to set up one rendezvous, the time required to send one data item and the time required to do one multiplication or addition.

As mentioned before, $w = n/q$, we assume that there are q processors available, so, formula (4.10.1) becomes

$$T = (2p^3/3 + 3p^2 - 8p/3 + 2)r + (17n^2p/6 + 7np/2 + n/2 - 4n^2/p - n^2/p^2 - 2n^2 - 4n^2/6p)s + (n^3/p - n^3/p^2 - 7n^3/3p^3 - 2n^2/p^2 + 4n^2/p)m \quad (4.10.2)$$

By differentiating (4.10.2) with respect to p we can find a theoretical optimum number of processors when $dT/dP = 0$.

We have

$$(2p^2 + 6p - \tfrac{8}{3})r + (17n^2/6 + 7n/2 + 4n^2/p^2 + 2n^2/p^3 + 2n^2/3p^2)s + (-n^3/p^2 + 2n^3/p^3 + 7n^3/p^4 + 4n^2/p^3 - 4n^2/p^2)m = 0$$

Taking the dominant terms, we get

$$2p^2 r + 17n^2s/6 - n^3m/p^2 = 0 \quad (4.10.3)$$

suppose we ignored the data sending time, then for large n we get

$$P = (n^3m/2r)^{1/4}$$

On the Sequent Balance, the ratio m/r is equal to 0.125. Then the optimum number of processors is

$$P = 0.5n^{3/4}.$$

If n = 60, we can say that the number of processors in this case is 10. Taking account of s, then the optimum number of processrs is

$$P^2 = \frac{-(17n^2s) \pm \sqrt{(17n^2s)^2 + 288n^3mr}}{24r} = \sqrt{n^2s^2/2r^2 + n^3m/2r}$$

In this case the optimum number of processors is

$$P = \left[n^4s^2/sr^2 + n^3m/2r \right]^{1/4} .$$

4.11.1 Conclusion

From the results of the computation carried out using row-oriented algorithm and block LU-decomposition algorithm we can give the following remarks:

1) The idea which is used to implement each of them is clearly acceptable in the case of using multiprocessors.
2) The speedups and the efficiencies of the calculation of row-oriented algorithm is much less than block LU-decomposition for fixed n and fixed number of processors. This can be shown that when n=80, the actual speedups for row-oriented algorithm and block LU-decomposition when p=7 are 1.81 and 3.84 respectively. This shows that the speedup of block LU-decomposition algorithm is about 2 times the speedup of row-oriented algorithm. The corresponding efficiencies are 0.25 and 0.55.
3) The optimum number of processors for row-oriented algorithm is 5, this means that the speedup for a given problem size increases to the maximum

when p=5 and then decreases. This is due to the high cost of the communication.

4) Finally, the block LU-decomposition gives a better speedup which is 4.34 with the efficiency falling only to about 0.62 for a problem of size 100 using seven processors.

CHAPTER 5

Parallel Solution of Initial Value Problems Using Second Order Block Methods

5.1 Introduction

In this chapter, we describe four new parallel algorithms using a fixed stepsize to solve ordinary differential equations defined by

$$y' = f(x,y), \quad y(x_0) = y_0, \quad x_0 \le x \le x_{end} \tag{5.1.1}$$

using one-step block methods, defined in general as

$$y_{n+r,o} = y_n + rh\, y'_n \tag{5.1.2}$$

$$y_{n+r,s+1} = y_n + h \sum_{j=0}^{k} a_{rj}\, y'_{n+j,s}, \qquad r=1,2,..,k;\ s=0,1 \tag{5.1.3}$$

where k is the block length.

We shall give three types of methods to solve (5.1.1), each of which is second order and applicable for arbitrary block length. For each method we compute the local truncation error and give a stability analysis. Finally, we present the description of parallel algorithms based on these methods, the theoretical cost and some experimental timings.

5.2 Methods 1 and 2

The requirement that the methods be second order for any block length k complicates the search for good methods. Initially, we looked for methods which had only very few nonzero coefficients; the aim was to produce methods which might be very efficient to implement. We found two classes of very simple second order methods, and we describe these first.

The first method considered is defined as

$$y_{n+r,s+1} = y_n + h(a_{r,r-1}\ y'_{n+(r-1),s} + a_{r,r}y'_{n+r,s}) \quad , r=1,2,..,k; \quad s=0,1 \quad (5.2.1)$$

where $a_{r,r-1} = \frac{r^2}{2}$, $a_{r,r} = r(1-r/2)$.

The second method is defined as

$$y_{n+r,s+1} = y_n + h(a_{r,o}y'_n + a_{r,r}y'_{n+r,s}) \quad , r=1,2,..,k; \quad s=0,1 \qquad (5.2.2)$$

where $a_{r,o} = a_{r,r} = r/2$

Note that the methods will be second order if the coefficients of the methods satisfy the first and the second characteristic polynomials, which are given as:

$$\sum_{j=0}^{k} a_{rj} = r \ , \quad \sum_{j=0}^{k} ja_{rj} = r^2/2 \ , \quad r=1,2,..,k. \qquad (5.2.3)$$

5.2.1 Principal Local Truncation Error

The rth component of local truncation error of (5.1.3) is given by

$$\varepsilon_r = y_{n+r,s+1} - y_n - h\sum_{j=0}^{k} a_{rj}\ y'_{n+j,s} \qquad (5.2.4)$$

where $y_{n+r} = y(x_{n+r})$ denotes the exact value of the solution at x_{n+r}.

ε_r can be expressed as a power series in h, and the remainder term of Taylor's series expansion used to derive ε_r.

Therefore, the kth component of local truncation error of (5.2.1) is

$$\varepsilon_k = (2k^3 - 3k^2)h^3y_n^{(3)}/12.$$

Then we can express $\underline{\varepsilon}$ as

$$\left[-h^3y_n^{(3)}/12, \quad h^3y_n^{(3)}/3, \quad 9h^3y_n^{(3)}/4, \ldots, \quad (2k^3-3k^2)h^3y_n^{(3)}/12 \right]^T,$$

for r=1,2,3,...,k.

The local truncation error of (5.2.2) is given by

$$\underline{\varepsilon} = \left[-h^3 y_n^{(3)}/12, \quad -2h^3 y_n^{(3)}/3, \quad -9h^3 y_n^{(3)}/4, \ldots, \quad -k^3 h^3 y_n^{(3)}/12 \right]^T .$$

So the local errors of the methods (5.2.1) and (5.2.2) estimated depend on k^3h^3. The global errors of these methods are of order k^2h^2.

5.2.2. Stability Analysis

The following analysis is carried out according to the stability of a fixed stepsize studied in [44]. Consider the general block method

$$y_{n+r,s+1} = y_n + h \sum_{j=0}^{k} a_{rj} \, y'_{n+j,s} , \quad r=1,2,\ldots,k; \; s=0,1.$$

Apply the method to the test equation $y' = \lambda y$, we get

$$y_{n+r} = y_n + H \sum_{j=0}^{k} a_{rj} \, y_{n+j} , \quad r=1,2,..,k.$$

where $H = \lambda h$.

Hence,

$$y_{n+1} = y_n + H(a_{1,0} \, y_n + a_{1,1} \, y_{n+1} + \ldots\ldots\ldots + a_{1,k} \, y_{n+k})$$

$$y_{n+2} = y_n + H(a_{2,0} \, y_n + a_{2,1} \, y_{n+1} + \ldots\ldots\ldots + a_{2,k} \, y_{n+k})$$

$$y_{n+3} = y_n + H(a_{3,0} \, y_n + a_{3,1} \, y_{n+1} + \ldots\ldots\ldots + a_{3,k} \, y_{n+k})$$

$$\vdots$$

$$y_{n+k} = y_n + H(a_{k,0} \, y_n + a_{k,1} \, y_{n+1} + \ldots\ldots\ldots + a_{k,k} \, y_{n+k})$$

Then the general form of the method in this representation is given by

$$\begin{bmatrix} 1-a_{1,1}H & -a_{1,2}H & \cdots & -a_{1,k}H \\ -a_{2,1}H & 1-a_{2,2}H & \cdots & -a_{2,k}H \\ -a_{3,1}H & -a_{3,2}H & 1-a_{3,3}H\ldots & -a_{3,k}H \\ \vdots & & & \\ -a_{k,1}H & -a_{k,2}H & \cdots & 1-a_{k,k}H \end{bmatrix} \begin{bmatrix} y_{n+1} \\ y_{n+2} \\ y_{n+3} \\ \vdots \\ y_{n+k} \end{bmatrix} = \begin{bmatrix} 1+a_{1,0}H \\ 1+a_{2,0}H \\ 1+a_{3,0}H \\ \vdots \\ 1+a_{k,0}H \end{bmatrix} y_n \quad (5.2.5)$$

Let

$$A = \begin{bmatrix} 1-a_{1,1}H & -a_{1,2}H \ldots & -a_{1,k}H \\ -a_{2,1}H & 1-a_{2,2}H \ldots & a_{2,k}H \\ & \vdots & \\ -a_{k,1}H & \cdots & 1-a_{k,k}H \end{bmatrix} \text{ and } \underline{B} = \begin{bmatrix} 1+a_{1,0}H \\ 1+a_{2,0}H \\ \vdots \\ 1+a_{k,0}H \end{bmatrix}$$

It is necessary that A to be invertible for all H. Let us define q(H), $q_k(H)$ by

$$q(H) = \det(A) \quad \text{and} \quad q_k(H) = \det(A_k)$$

where A_r is the matrix A with the rth column replaced by the vector B. Using the above definitions and Cramer's rule in (5.2.3) we find that

$$y_{n+r} = \frac{q_r(H)}{q(H)} y_n \quad , \quad r=1,2,\ldots,k;$$

where q(H) and $q_r(H)$ are polynomials with degree k.

For $n=mk$, $m=0,1,2,\ldots,$

we have

$$y_{n+r} = \left[\frac{q_r(H)}{q(H)} \right] \left[\frac{q_k(H)}{q(H)} \right]^m , \quad r=1,2,\ldots,k.$$

Hence, the block method is said to be A-stable if and only if

$$\left| \frac{q_k(H)}{q(H)} \right| < 1 \text{ for all } H, \text{ Re}(H) < 0.$$

In the case when k = 2, (5.2.5) becomes

$$\begin{bmatrix} 1-a_{1,1}H & -a_{1,2}H \\ -a_{2,1}H & 1-a_{2,2}H \end{bmatrix} \begin{bmatrix} y_{n+1} \\ y_{n+2} \end{bmatrix} = \begin{bmatrix} 1+a_{1,0}H \\ 1+a_{2,0}H \end{bmatrix} y_n$$

$$\Rightarrow \begin{bmatrix} y_{n+1} \\ y_{n+2} \end{bmatrix} = \begin{bmatrix} 1-a_{1,1}H & -a_{1,2}H \\ -a_{2,1}H & 1-a_{2,2}H \end{bmatrix}^{-1} \begin{bmatrix} 1+a_{1,0}H \\ 1+a_{2,0}H \end{bmatrix} y_n$$

where

$$A = \begin{bmatrix} 1-a_{1,1}H & -a_{1,2}H \\ -a_{2,1}H & 1-a_{2,2}H \end{bmatrix}$$

$$\begin{bmatrix} y_{n+1} \\ y_{n+2} \end{bmatrix} = \frac{\begin{bmatrix} H^2(a_{1,2}a_{1,0} - a_{2,2}a_{1,0}) + H(a_{1,2} - a_{2,2}) + (Ha_{1,0}+1) \\ H^2(a_{2,1}a_{1,2} - a_{1,1}a_{2,0}) + H(a_{2,1} - a_{1,1}) + (Ha_{2,0}+1) \end{bmatrix}}{H^2(a_{1,1}a_{2,2} - a_{1,2}a_{2,1}) + H(a_{1,1}+a_{2,2}) + 1} y_n$$

$$= \frac{\begin{bmatrix} q_1(H) \\ q_2(H) \end{bmatrix}}{q(H)} \quad y_n$$

where $q_1(H) = \det \begin{bmatrix} 1+a_{1,0}H & -a_{1,2}H \\ 1+a_{2,0}H & 1-a_{2,2}H \end{bmatrix}$,

$q_2(H) = \det \begin{bmatrix} 1-a_{1,1}H & 1+a_{1,0}H \\ -a_{2,1}H & 1+a_{2,0}H \end{bmatrix}$, and $q(H) = \det A$.

Hence, $q_r(H) = \det(A_r)$

where A_r is the matrix A with the rth column replaced by the vector $\begin{bmatrix} 1+a_{1,0}H \\ 1+a_{2,0}H \end{bmatrix}$. Then we get

$$y_{n+r} = \frac{q_r(H)}{q(H)} \; y_n \quad , \quad r=1,2.$$

where $q(H)$ and $q_r(H)$ are polynomials with degree 2. Then we can say the method is A-stable (k=2) if and only if

$$\left| \frac{q_2(H)}{q(H)} \right| = \left| \frac{H^2(a_{2,1}\,a_{1,0} - a_{1,1}\,a_{2,0}) + H(a_{2,1} - a_{1,1}) + H(a_{2,0}+1)}{H^2(a_{1,1}\,a_{2,2} - a_{1,2}\,a_{2,1}) + H(a_{1,1}+a_{2,2}) + 1} \right| < 1$$

for all H, Re(H) < 0.

Note that the stability analysis depends only on the Corrector formula.

To discuss the stability of the methods 1 and 2, we apply the first method (5.2.1) to the test equation $y' = \lambda y$, we find

$$\begin{bmatrix} 1-H/2 & 0 & 0 & 0 & 0 & 0 \\ -2H & 1 & 0 & 0 & 0 & 0 \\ 0 & -9H/2 & 1+3H/2 & 0 & 0 & 0 \\ 0 & 0 & -8H & 1+4H & 0 & 0 \\ \vdots & \vdots & \vdots & & & \vdots \\ 0 & 0 & 0 & & & 0 \\ 0 & 0 & 0 & 0 & -k^2H/2 & 1-k(1-k/2)H \end{bmatrix} \begin{bmatrix} y_{n+1} \\ y_{n+2} \\ y_{n+3} \\ y_{n+4} \\ \vdots \\ y_{n+k} \end{bmatrix} = \begin{bmatrix} 1+H/2 \\ 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

The $q(H)$ and $q_k(H)$ for the above scheme are given by

$$q(H) = \begin{bmatrix} 1-H/2 & 0 & 0 & 0 & 0 \\ -2H & 1 & 0 & 0 & 0 \\ 0 & -9H/2 & 1+3H/2 & 0 & \\ 0 & 0 & -8H & 1+4H & \\ \vdots & \vdots & \vdots & & 0 \\ 0 & 0 & 0 & & \\ 0 & 0 & 0 & -k^2H/2 & 1-k(1-k/2)H \end{bmatrix},$$

$$q_k(H) = \begin{bmatrix} 1-H/2 & 0 & 0 & 0 & 0 & 1+H/2 \\ -2H & 1 & 0 & 0 & 0 & 1 \\ 0 & -9H/2 & 1+3H/2 & 0 & 0 & 1 \\ 0 & 0 & -8H & 1+4H & & 1 \\ \vdots & \vdots & 0 & & & \vdots \\ 0 & 0 & 0 & & -k^2H/2 & 1 \end{bmatrix}$$

The amplification factor is

$$\left| \frac{q_k(H)}{q(H)} \right| = \left| \frac{k(k-1)H^2 + (2k-1)H + 2}{-H+2} \right|$$

which is less than one for all values of $H \varepsilon \left(\frac{-2}{k-1}, 0 \right)$. So the method is A-stable for all $H \varepsilon \left(\frac{-2}{k-1}, 0 \right)$. In general, as the block length increases, the stability bounds decreases.

We want now to examine the stability of method 2. To illustrate this by applying method (5.2.2) to the test equation $y' = \lambda y$, we have

$$\begin{bmatrix} 1-H/2 & 0 & 0 & 0 \ldots\ldots & 0 \\ 0 & 1-H & 0 & 0 \ldots\ldots & 0 \\ 0 & 0 & 1-3H/2 & 0 \ldots\ldots & 0 \\ \vdots & \vdots & 0 & & \vdots \\ 0 & 0 & 0 \ldots\ldots\ldots 0 & & 1-kH/2 \end{bmatrix} \begin{bmatrix} y_{n+1} \\ y_{n+2} \\ y_{n+3} \\ \vdots \\ y_{n+k} \end{bmatrix} = \begin{bmatrix} 1+H/2 \\ 1+H \\ 1+3H/2 \\ \vdots \\ 1+kH/2 \end{bmatrix} y_n \quad .$$

The amplification factor is given by

$$\left| \frac{q_k(H)}{q(H)} \right| = \left| \frac{1+kH/2}{1-kH/2} \right| < 1 \quad \text{for all } H < 0, \text{ then}$$

we can say that the method is A-stable for all $H<0$.

5.2.3 Parallel Algorithms

We shall, in this section, describe the implementation of parallel algorithms of methods 1 and 2.

Let k be the block length, for parallel implementation the block length is divided into p=k/q sub-blocks. In these algorithms the treatments of different sub-blocks is carried out in parallel, and the size of sub-block q determines the granularity of the parallelism. Thus, the procedure can be applied using k/q tasks, where each task is allocated to do the calculations for q points.

a) Algorithm 1

We shall first present the parallel implementation of method 1, using the formula (5.1.2) as a predictor and (5.2.1) as the corrector-formula. The mode $P(EC)^2$ which will be used is described as follows:

$$P: \quad y^{P}_{n+r} = y_n + r * h * y'_n$$

$$E: \quad y'^{P}_{n+r} = f\left[x_{n+r},\ y^{P}_{n+r}\right]$$

$$C: \quad y^{C}_{n+r} = y_n + h\left[(r^2/2)y'^{P}_{n+(r-1)} + r(1-r/2)y'^{P}_{n+r} \right]$$

$$E: \quad y'^{C}_{n+r} = f\left[x_{n+r},\ y^{C}_{n+r} \right]$$

$$C: \quad y^{CC}_{n+r} = y_n + h\left[(r^2/2)y'^{C}_{n+(r-1)} + r(1-r/2)y'^{C}_{n+r} \right]$$

$$r = 1,2,..,k$$

where n initially takes the value zero, then increases by the value of k until the end of the number of steps.

From the above sequence, we can say that the predicted values for k/q tasks can be obtained in parallel, where each task computes the predicted

values for q points, similarly for the function evaluations and the corrected values. On this basis the algorithm needs three task types. The first task is as follows:

```
Task type predictor-formula:
Send-initial (i,q,step-size) to task i;
Declare;
  ynew:vector (1..q);
Begin
Serial-loop
Send data (initial-value, function-value);
function-value:=stepsize * function value;
ynew(1):=initial-value + float(1+(i-1)q) * function-value;
  for r in 2..q loop
ynew(r):=ynew(r-1) + function-value;
End loop;
Collect the values (ytempi);
End Serial-loop;
End;
End predictor-formula;
```

The above task computes the predicted values for k/q tasks. For example, task 1 computes the predicted values $y_{n+1},\ldots,y_{n+q}$, task 2 computes $y_{n+(q+1)},\ldots,y_{n+2q}$ and so on until task k/q computes $y_{n+(k-q+1)},\ldots,y_{n+k}$.

Now, we present the task required for the function evaluations.

Task type function-evaluations:

```
Send-initial (i,q) to task i;
Declare;
  xnew,ynew,znew:vector (1..q);
Begin
Serial-loop
Send-data (xnew,ynew) to task i;
For r in 1..q loop
znew (r):= dy(xnew(r),ynew(r));   --call the function dy;
End loop;
Collect the function (ztempi);
End;
End function-evaluations;
```

The remaining task is the task to compute the corrected values.

```
Task-type corrected-formula:
Send-initial (i,q) to task i;
Declare
   anew : column (1..q);
   bnew : diagonal (1..q);
   ynew,znew : vector (1..q);
Begin
Serial-loop
Send-data (anew,bnew,function-value,znew);
Call a procedure to compute the corrected values;
Collect the values (ytempi);
```

End loop;

End;

End corrector-formula;

where anew and bnew denote the

$a_{r,r-1}$ and $a_{r,r}$ for $r=(i-1)q+1,\ldots,iq$; $i=1,\ldots,k/q$

respectively.

The main procedure is as follows:

Declare;

x,y,z : vector (1..k);

a : column (1..k);

b : diagonal (1..k);

Begin

For i in 1..k/q loop

pv(i).send_initial (i,q,step_size) to predictor_formula task i;

fv(i).send_initial (i,q) to function_evaluation task i;

cr(i).send_initial (i,q) to corrector_formula task i;

End loop;

n:=0;

loop

Exit when n > number of steps;

zz:= dy(xstart, initial_value);

--call predicted task

For i in 1..k/q loop

pv(i).send_data(initial_value,zz);

End loop;

```
For i in 1..k/q loop
pv(i).collect(y((i-1)*q+1..i*q));
End loop;

For s in 0..1 loop -- correction loop
-- call function evaluation task

For i in 1..k/q loop
fv(i).send_data (x((i-1)*q+1..i*q), y((i-1)*q+1..i*q));
End loop;

For i in 1..k/q loop
fv(i).collect (z((i-1)*q+1..i*q));
End loop;
-- call corrector task

For i in 1..k/q loop
cr(i).collect (a((i-1)*q+1..i*q), b((i-1)*q+1..i*q), zz, z(((i-1)*q+1..i*q))
End loop;

For i in 1..k/q loop
cr(i).collect (y((i-1)q+1..i*q));
End loop;
End loop;  -- end correction loop;
-- the values xstart and initial_value for the next block;
initial_value:=   y(k);
xstart:=  x(k);
```

n:= n+k;-- increment the n_value;

Exit when n=number of steps;

End loop;

End procedure;

where zz denotes to y'_n, z((i-1)*q+1,..,i*q) denotes y'_{n+r}; r=(i-1)*q+1,..,i*q; i=1,..,k/q.

b) **Algorithm 2**

We present the parallel implementation of method 2. The algorithm here is exactly similar to the algorithm (1) except we use the corrector-formula (5.2.2) which is different from the (5.2.1).

The description of the algorithm is as follows:

1. A task type to compute the predicted values.
2. A task type to evaluate the functions.
3. A task type to compute the corrected values.

Note that all the algorithms in this chapter were implemented on Sequent Balance using Ada multitasking. All these algorithms require for each run the specification of p, the number of processors, k, the block length, q, the sub-block length and Nsteps, the number of steps to be used and hence the stepsize h can be computed.

5.2.4 Theoretical Cost

We shall give the theoretical cost of the algorithms in terms of arithmetic operations count, communication costs, data sending and function

evaluations. In the Sequent Balance, we assume that the time to do one multiplication or one addition is m=0.25 millisec, the time to send one item is s=0.02 millisec, the time to set one rendezvous is r=2 millisec and the time to do one function evaluation depends on the type of the function (see appendix F).

As mentioned above, the algorithm 2 is exactly similar to algorithm 1. So we shall give the theoretical cost of algorithm 1 only. Also we give the total cost of the sequential mode of algorithm 1.

1) The Parallel Cost of Algorithm 1

The first part of this algorithm is to compute the predicted values, so we have three rendezvous to send 3+q data items, the total number of arithmetic operations is q. Assume the number of processors $p \lesssim k/q$, then the total number of arithmetic operations is k/p. So the total cost of the predicted task per block is given by

$$(r+s)k/q + \left[2kr/q + (2+q)ks/q + km/p \right].$$

On the other hand, the cost of evaluating the functions is

$$kr/q + \left[4kr/q + 6ks + 2kf/p \right].$$

Finally, the cost of computing the corrected values is

$$kr/q + \left[4kr/q + (8q+2)ks/q + 10km/p \right].$$

Hence, the total cost of algorithm 1 is

$$(3r+s)k/q + \left[10kr/q + (15q+4)ks/q + (11m+2f)k/p \right] * c/kh \qquad (5.2.6)$$

where c is the end of the interval and h is the stepsize.

2) The Sequential Cost of Algorithm 1

The total cost of the sequential algorithm is given by

$$\left[\ 11km+2kf\ \right]\ c/kh.$$

Then the speedup is given by

$$S_1 = \frac{[\ 11km+2kf\ ]\ c/kh}{(3r+s)k/q\ +[\ 10kr/q\ +(15q+4)ks/q\ +(11m+2f)k/p]\ c/kh}$$

Assuming the number of tasks is equal to the number of processors, then

$$S_1 = \frac{[\ 11km+2kf\ ]\ c/kh}{(3r+s)p\ +[10pr\ +(15q+4)ps\ +(11m+2f)q]\ c/kh} \qquad (5.2.7)$$

5.2.5 Test Problems

In this section we present four select test problems given by:

Problem (i):

$$y'=y,\quad y(0)=1\ ,\quad 0\le x\le 1\ .$$

The analytic solution is $y=e^x$.

Problem (ii): we use the following problem [6],

$$y'=-y^3/2\ ,\quad y(0)=1\ ,\quad 0\le x\le 20\ .$$

The analytic solution is $y = 1/\sqrt{1+x}$.

Problem (iii):

$$y'=(20y-y^2)/80\ ,\quad y(0)=1\ ,\quad 0\le x\le 20$$

The analytic solution is

$$y\ =20/(1+19\ \bar{e}^{\,x/4}).$$

Problem (iv):

$$y'=y\ \cos\ x\ ,\quad y(0)=1\ ,\quad 0\le x\le 1$$

The analytic solution is

$$y = e^{\sin x} \quad .$$

5.2.6 The Results for Algorithm 1

We shall present the theoretical costs obtained by applying algorithm 1 to problem (i) where the cost is measured in seconds and the number of steps = 10,000.

Table 5.2.1

Blocklength k	Sub-block q	p	Number of Tasks	Sequential Cost	Parallel Cost	Speed Up S_1
500	250	2	2	27.9	17.5	1.59
500	100	5	5	27.9	10.2	2.74
200	100	2	2	27.9	18.7	1.49
200	50	4	4	27.9	13.5	2.07
200	40	5	5	27.9	13.2	2.11

The above table shows the theoretical costs of the sequential algorithm and parallel algorithm but only when the number of tasks is equal to the number of processors. We observe that as the number of processors is increased the speedup increases to the maximum value. In this case, $S_1 = 2.74$ when $k=500$, $q=100$ and $p=5$. Note that the speedups are obtained from (5.2.7).

It is important to know the space of time needed for the computation and the communication. The following table illustrates the theoretical cost for the computation, function evaluation, rendezvous and data distribution as a percentage, when solving problem (ii).

Table 5.2.2

Block Length k	Sub-block q	p	Computation Cost	Function Evaluation Cost	Rendezvous Cost	Data Distribution Cost
200	100	1 2	68.9% 61.7%	19.5% 17.5%	6.5% 11.6%	5.0 % 9.0 %
200	50	1 3 4	65.6% 49.8% 44.6%	18.6% 14.1% 12.7%	9.5% 21.7% 25.9%	6.2% 14.1% 16.9%
200	40	1 2 5	64.1% 54.3% 37.4%	18.2% 15.4% 10.6%	11.7% 19.8% 34.0%	6.1% 10.3% 17.7%

From the above table we observe that the time of function evaluations dominates the computation and the communication time. The function evaluations time and the computation time decrease by increasing the number of processors, while the communication time increases. However, for k=200, q = 50 and p = 4, the total of the communication cost is 42.8% which is so expensive compared to the time of both function evaluations and the computation. Also we can deduce that for small q, the cost of the rendezvous is very effective.

The experimental timings, t(A), the theoretical cost, t(p), and the corrresponding speedups S(A) and S(p) for algorithm 1 solving problems (i) and (ii) are given in tables 5.2.3 and 5.2.4 respectively.

Table 5.2.3

Block Length k	Sub-Block q	Number of Tasks k/q	Number of Blocks	Number of Processors p	t(A)	t(p)	s(A)	s(p)
500	250	2	20	1 2	31.3 20.4	30.8 17.5	1.0 1.53	1.0 1.76
500	100	5	20	1 2 3 4 5	32.5 21.5 17.7 17.3 17.5	32.5 18.7 14.0 11.5 10.2	1.0 1.51 1.84 1.89 1.86	1.0 1.74 2.32 2.86 3.19
200	100	2	50	1 2	32.6 21.6	32.5 18.7	1.0 1.51	1.0 1.74
200	50	4	50	1 2 3 4	34.5 23.8 18.9 17.2	34.4 20.7 15.8 13.5	1.0 1.45 1.83 2.00	1.0 1.66 2.18 2.55
200	40	5	50	1 2 3 4 5	36.4 23.1 19.0 18.5 18.8	35.5 22.9 16.8 14.5 13.2	1.0 1.58 1.92 1.97 1.94	1.0 1.55 2.11 2.45 2.69

The above table illustrates the comparison of the experimental and theoretical times. We observe that the experimental time decreases to a minimum and then increases slightly when $p > 4$. The number of processors required is quite small. This depends on the total number of arithmetic operations which are not too much and the high cost of the communication. Also we notice that the actual speedup for $k=500$ and $q=100$ increases from

1.51 to 1.89 when p varies between 2 and 4, whereas the corresponding theoretical speedup increases from 1.74 to 2.86.

Table 5.2.4

Block Length k	Sub-Block q	Number of Tasks k/q	Number of Blocks	Number of Processors p	t(A)	t(p)	s(A)	s(p)
500	250	2	20	1 2	38.9 23.8	38.7 21.1	1.0 1.63	1.0 1.83
500	100	5	20	1 2 3 4 5	40.0 24.5 19.6 18.2 18.3	39.9 22.3 16.3 13.4 11.7	1.0 1.63 2.04 2.20 2.19	1.0 1.78 2.45 2.98 3.41
200	100	2	50	1 2	40.0 24.7	39.9 22.3	1.0 1.62	1.0 1.79
200	50	4	50	1 2 3 4	42.0 25.6 20.4 20.3	41.9 24.3 18.4 15.4	1.0 1.64 2.06 2.07	1.0 1.72 2.28 2.72
200	40	5	50	1 2 3 4 5	43.2 25.8 21.0 19.8 20.1	42.9 25.3 19.3 16.4 14.7	1.0 1.67 2.06 2.18 2.15	1.0 1.70 2.22 2.62 2.92

Table 5.2.4 illustrates that the time taken to run algorithm 1 solving problem (ii) is longer than the time taken for problem (i). The time difference between these problems $\left[(i),(ii)\right]$ is 7.4 seconds (if the number of processors is one). This is because the function evaluation cost if very

expensive compared to the function evaluation cost for problem (i), (see appendix G).

As mentioned in section 5.2.1, the truncation errors of methods 1 and 2 depend on k^3h^3. This means that, if k is too large, as determined in our results, the local truncation errors are too big. To avoid this problem, we seek methods for which the local truncation error has a lower power of k. At this point we have two alternative methods. The first alternative is described in the following section.

5.3 Method 3

The third method considered is defined as

$$y_{n+r,s+1} = y_n + h(a_{r,o}\ y'_n + a_{r,1}\ y'_{n+1,s} + a_{r,k}\ y'_{n+k,s}) \qquad (5.3.1)$$

$$r=1,2,..,k \ ; \ s=0,1$$

where

$$a_{r,0} = r(2+k)/3 - r^2/2 \ , \ a_{r,1} = r^2/2 - rk/2 \text{ and } a_{r,k} = r/3 \ .$$

Note that these coefficients satisfy (5.2.3) for r=1,2,..k.

We note that the values of $a_{r,0}$ and $a_{r,1}$ are dependent on k for r=1,2,..,k.

The coefficients for this method are shown in the table below, where k=2,3 and 4.

Table 5.3.1

k	r	$a_{r,0}$	$a_{r,1}$	$a_{r,k}$
2	1	5/6	-1/6	1/3
	2	2/3	2/3	2/3
3	1	7/6	-1/2	1/3
	2	4/3	0	2/3
	3	1/2	3/2	1
4	1	3/2	-5/6	1/3
	2	2	-2/3	2/3
	3	3/2	1/2	1
	4	0	8/3	4/3

Clearly the coefficients $a_{r,0}$ and $a_{r,1}$ take different values for different k, whereas $a_{r,k}$ does not change.

5.3.1 Principal Local Truncation Error

The method (5.3.1) can be written in the form

$$y_{n+1} = y_n + h\left[\left(\frac{1}{6}+\frac{k}{3}\right)y'_n + \left(\frac{1}{2}-\frac{k}{3}\right)y'_{n+1} + \frac{1}{3}y'_{n+k}\right]$$

$$y_{n+2} = y_n + h\left[\left(-\frac{2}{3}+\frac{2k}{3}\right)y'_n + \left(2-\frac{2}{3}k\right)y'_{n+1} + \frac{2}{3}y'_{n+k}\right]$$

$$y_{n+3} = y_n + h\left[\left(-\frac{5}{2}+k\right)y'_n + \left(\frac{9}{2}-k\right)y'_{n+1} + y'_{n+k}\right] \qquad (5.3.2)$$

$$y_{n+4} = y_n + h\left[\left(-\frac{16}{3}+\frac{4k}{3}\right)y'_n + \left(8-\frac{4}{3}k\right)y'_{n+1} + \frac{4}{3}y'_{n+k}\right]$$

$$y_{n+5} = y_n + h\left[\left(-\frac{55}{6}+\frac{5}{3}k\right)y'_n + \left(\frac{25}{2}-\frac{5k}{3}\right)y'_{n+1} + \frac{5}{3}y'_{n+k}\right]$$

$$\vdots$$

$$y_{n+k} = y_n + h\left[\left(\frac{2k}{3}-\frac{k^2}{6}\right)y'_n + \frac{k^2}{6}y'_{n+1} + \frac{k}{3}y'_{n+k}\right]$$

Using (5.2.4), we get

$$\underline{\varepsilon} = \left[(2(k-k^2)-1)h^3y_n^{(3)}/12 \ , \quad ((k-k^2)+1)h^3y_n^{(3)}/3 \ ,\ldots, \ -k^2h^3y_n^{(3)}/12 \right]^T$$

We note that the local truncation error for method (5.3.1) depends on k^2h^3 which is smaller than the local truncation errors for methods 1 and 2. Thus, we may expect higher accuracy than that of previous methods and when h is small enough, the results will always be stable. Actually this is what we preferred.

5.3.2 Stability Analysis

To analyse the stability of (5.3.1), apply (5.3.1) to the test equation $y'=\lambda y$, then the scheme is

$$\begin{bmatrix} 1-(3-2k)H/6 & 0 & 0 & 0 & \ldots & -H/3 \\ -(6-2k)H/3 & 1 & 0 & 0 & & -2H/3 \\ -(9-2k)H/2 & 0 & 1 & 0 & & -H \\ -(16+4k)H/3 & 0 & 0 & 1 & & -4H/3 \\ & & & 0 & & \\ -k^2H/6 & 0 & 0 & 0 & & 1-kH/3 \end{bmatrix} \begin{bmatrix} y_{n+1} \\ y_{n+2} \\ y_{n+3} \\ y_{n+4} \\ \\ y_{n+k} \end{bmatrix} = \begin{bmatrix} 1+(1+2k)H/6 \\ 1+(-2+2k)H/3 \\ 1+(-5+2k)H/2 \\ \\ \\ 1+(4k-k^2)H/6 \end{bmatrix} y_n$$

which has the values of the determinants as follows:

$$q(H) = \left\{ 1 - \frac{(3-2k)H}{6} \right\} \left[1 - \frac{kH}{3} \right] - \left[\frac{H}{3} \right] \left[\frac{k^2 H}{6} \right]$$

$$= 1 - \frac{H}{2} + \frac{(k-k^2)}{6} H \qquad \text{and}$$

$$q_k(H) = \left\{ 1 - \frac{(3-2k)H}{6} \right\} \left\{ 1 + \frac{(4k-k^2)H}{6} \right\} + \left\{ 1 + \frac{(1+2k)H}{6} \right\} \frac{(k^2 H)}{6}$$

$$= 1 + \frac{(2k-1)}{2} H + \frac{2(k^2-k)H^2}{6} \quad .$$

Then the amplification factor is given by

$$\left| \frac{q_k(H)}{q(H)} \right| = \left| \frac{6 + (6k-3)H + 2(k^2-k)H^2}{6 - 3H + (k-k^2)H^2} \right| < 1 \quad \text{for all value of}$$

$H \varepsilon \left[\frac{-2}{k-1}, 0 \right]$, so the method is A-stable for all $H \varepsilon \left[\frac{-2}{k-1}, 0 \right]$.

5.4 Method 4

The fourth method considered is given by

$$y_{n+r,s+1} = y_n + h \sum_{j=0}^{r} a_{rj} \, y'_{n+j,s} \quad , \quad r=1,2,..,k \quad , \quad s=0,1 \qquad (5.4.1)$$

where $a_{r,o} = a_{r,r} = \frac{1}{2}$, $r=1,2,3,..,k$;

$a_{r,j} = 1$, $r=2,3,..,k$; $j=1,2,..,r-1$.

The following table illustrates the coefficients for the above method.

Table 5.4.1

r	$a_{r,0}$	$a_{r,1}$	$a_{r,2}$	$a_{r,3}$	$a_{r,4}$	$a_{r,5}$	.....	$a_{r,k}$
1	1/2	1/2						
2	1/2	1	1/2					
3	1/2	1	1	1/2				
4	1/2	1	1	1	1/2			
5	1/2	1	1	1	1	1/2		
⋮	⋮							
k	1/2	1	1	1	1	1	.....	1/2

It is easy to show that these coefficients satisfy (5.2.3) when r=1,2,..,k;

1) For r=1

$$a_{1,0} + a_{1,1} = 1/2 \quad , \quad a_{1,1} = 1/2$$

2) For r=2

$$a_{2,0} + a_{2,1} + a_{2,2} = 2 \quad , \quad a_{2,1} + 2a_{2,2} = 2$$

3) For r=k

$$a_{k,0} + a_{k,1} + \ldots + a_{k,k-1} + a_{k,k} = 1 + \sum_{r=1}^{k-1} 1 = k;$$

$$a_{k,1} + 2a_{k,2} + 3a_{k,3} + \ldots + (k-1)\, a_{k,k-1} + k a_{k,k} = \sum_{r=1}^{k-1} r + k/2$$

$$= k(k-1)/2 + k/2 = k^2/2 \quad .$$

5.4.1 Principal Local Truncation Error

To start this discussion, we first give the kth formula of (5.4.1)

$$y_{n+k} = y_n + \frac{h}{2}\left[y'_n + 2y'_{n+1} + 2y'_{n+2} + \ldots + 2y'_{n+(k-1)} + y'_{n+k} \right] .$$

Using (5.2.4), we get

$$\varepsilon_k = y_{n+k} - y_n - \frac{h}{2}\left[y'_n + 2y'_{n+1} + 2y'_{n+2} + \ldots\ldots + 2y'_{n+(k-1)} + y'_{n+k} \right]$$

$$= h^3 y_n^{(3)} \left[-k^3/6 - 1/2 \left\{ \left[1+4+9+\ldots + (k-1)^2 \right] + k^2/2 \right\} \right]$$

$$= h^3 y_n^{(3)} \left[k^3/6 - \left[2k^3 - 3k^2 + k + 3k^2 \right]/12 \right]$$

$$= h^3 y_n^{(3)} \left[k^3/6 - k^3/6 - k/12 \right]$$

Therefore,

$$\varepsilon_k = -kh^3 y_n^{(3)}/12 \quad .$$

Thus, the local truncation error, ε, can be written as

$$\underline{\varepsilon} = \left[-\frac{h^3}{12} y_n^{(3)} \;,\; -\frac{h^3}{6} y_n^{(3)} \;,\; -\frac{h^3}{4} y_n^{(3)} \;,\ldots,\; -\frac{k}{12} h^3 y_n^{(3)} \right]^T .$$

In this case, we expect the method will be more accurate than the previous methods and give better results.

5.4.2 Stability Analysis

Finally, the scheme of method (5.4.1) may be written in matrix form

$$\begin{bmatrix} 1-H/2 & 0 & 0 & 0\ldots\ldots & 0 \\ -H & 1-H/2 & 0 & 0\ldots\ldots & 0 \\ -H & -H & 1-H/2 & 0\ldots\ldots & 0 \\ -H & -H & -H & & 0 \\ \vdots & \vdots & \vdots & & 0 \\ -H & -H & -H & -H & 1-H/2 \end{bmatrix} \begin{bmatrix} y_{n+1} \\ y_{n+2} \\ y_{n+3} \\ y_{n+4} \\ \vdots \\ y_{n+k} \end{bmatrix} = \begin{bmatrix} 1+H/2 \\ 1+H/2 \\ 1+H/2 \\ 1+H/2 \\ \vdots \\ 1+H/2 \end{bmatrix} y_n \qquad (5.4.2)$$

Consider, for example, the case k=2, to find the stability region. The above scheme (5.4.2) becomes

$$\begin{bmatrix} 1-H/2 & 0 \\ -H & 1-H/2 \end{bmatrix} \begin{bmatrix} y_{n+1} \\ y_{n+2} \end{bmatrix} = \begin{bmatrix} 1+H/2 \\ 1+H/2 \end{bmatrix} y_n$$

The value of the determinates $q(H)$ and $q_2(H)$ are as follows:

$$q(H) = (1-H/2)^2 \quad \text{and}$$

$$q_2(H) = (1-H/2)(1+H/2) + H(1+H/2) = (1+H/2)(1+H/2) = (1+H/2)^2.$$

Therefore, the amplification factor is given by

$$\left| \frac{q_2(H)}{q(H)} \right| = \left| \frac{(1+H/2)^2}{(1-H/2)^2} \right| < 1 \quad \text{for all values of } H < 0,$$

thus showing this method is A-stable for k=2.

Similarly for k=3,

$$q(H) = (1-H/2)^3,$$

$$q_3(H) = (1-H/2)\{(1-H/2)(1+H/2) + H(1+H/2)\} + (1+H/2)\{H^2 + H(1-H/2)\}$$

$$= (1-H/2)(1+H/2)^2 + H(1+H/2)^2 = (1+H/2)^3.$$

So, the amplification factor is

$$\left| \frac{q_3(H)}{q(H)} \right| = \left| \frac{(1+H/2)^3}{(1-H/2)^3} \right| < 1.$$

In general, we have

$$\left| \frac{q_k(H)}{q(H)} \right| = \left| \frac{(1+H/2)^k}{(1-H/2)^k} \right|$$ which is less than one for all $H < 0$, so the method is A-stable.

5.5 The Parallel Algorithms for Methods 3 and 4

Before we go directly into the parallel algorithms, we want to discuss how these methods work in practice. We start first with the third method which was mentioned in (5.3). The method is given by

$$y_{n+r,s+1} = y_n + h\left[(r(2+k)/3 - r^2/2)\ y'_{n,s} + (r^2/2 - rk/2)\ y'_{n+1,s} + (r/3)\ y'_{n+k,s}\right]$$

$r=1,2,..,k; \quad s=0,1.$

In particular, three coefficients matrix will be considered. We may summarize the procedure in the following manner:

1. Compute y'_n, then compute the predicted values y_{n+1} and y_{n+k} using (5.1.2).
2. Evaluate the functions y'_{n+1} and y'_{n+k}.
3. Using method 3 to compute the corrected values y_{n+1}, y_{n+2},..., y_{n+k}.

In fact, for each block the corrector formula requires only three function evaluations which are y'_n, y'_{n+1} and y'_{n+k}, also these are needed for each step. On this basis we shall calculate both of the predicted values and the functions serially, while the corrected values can be calculated in parallel.

We now turn to method 4 which was mentioned in (5.4). The corrector formula (5.4.1) can be written in the matrix form as

$$\begin{bmatrix} y_{n+1} \\ y_{n+2} \\ y_{n+3} \\ y_{n+4} \\ \vdots \\ y_{n+k} \end{bmatrix} = \begin{bmatrix} y_n \\ y_n \\ y_n \\ \vdots \\ y_n \end{bmatrix} + h \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & & & & & \\ \frac{1}{2} & 1 & \frac{1}{2} & & & & \\ \frac{1}{2} & 1 & 1 & \frac{1}{2} & & & \\ \frac{1}{2} & 1 & 1 & 1 & \frac{1}{2} & & \\ \vdots & & & & & & \\ \frac{1}{2} & 1 & 1 & 1 & \cdots\cdots & 1 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} y'_n \\ y'_{n+1} \\ y'_{n+2} \\ \vdots \\ y'_{n+k} \end{bmatrix} \qquad (5.5.1)$$

In order to start the calculation, we need y'_n to compute the predicted values y_{n+1}, y_{n+2} ,..., y_{n+k} using (5.1.2). In this case, we need to evaluate all the functions y'_{n+1}, y'_{n+2} ,..., y'_{n+k} using the predicted values, then using method (5.4.1) to compute the corrected values. Here we can implement both the predictor and the corrector formulae in parallel. On this basis, this method would appear to be an excellent one compared to the previous methods. There is one point to note about the above scheme (5.5.1). As we mentioned before in section (5.2.3), in an attempt to divide the blocklength k into several sub-blocks, each of size q, the valuesy$'_{n+r}$,r=1,2,..,k; should be available for each sub-block.

Next we shall describe the parallel algorithms for methods 3 and 4 using Ada multitasking.

a) Algorithm 3

We shall first give the parallel implementation for method 3. The idea is to use (5.1.2) as a predictor formula and (5.3.1) as a corrector formula. As we mentioned above, the predicted values and the function evaluations can be obtained serially, this means no need to create any tasks for these values. The only task required in this algorithm is the one to compute the corrected values. The task required is as follows:

```
Task type corrected_formula:
Send_initial (i,q) to task i;
Declare:
```

```
    ynew: vector (1,,q);
    col_0, col_1, col_k:   column (1..q);
Begin
Serial_loop
Send_data (col_0, col_1, col_k, zz, z(1), z(k));
Call procedure to evaluate the corrected values;
Collect the corrected values (ytempi);
End loop;
End corrector_formula;
```

where q is the sub-block length as mentioned in (5.2), the columns col_0, col_1, col_k denote the coefficients $a_{r,0}$, $a_{r,1}$ and $a_{r,k}$ respectively and zz, z(1) and z(k) denote y'_n, y'_{n+1} and y'_{n+k} respectively. To illustrate the above task we shall give the description of the main procedure.

The main procedure is as follows:

```
Declare
   a,b,c:  column (1..k);
Begin
For i in 1..k/q loop
cr(i).  Send_initial (i,q) to corrector_formula task i;
End loop;
n:= 0;
Loop
Exit when n > number of steps;
zz:= dy(xstart, initial_value);
```

```
-- Evaluate the predicted values
 y(1):= initial_value + h * zz;
 y(k):= initial_value + float (k) * h * zz;
For S in 0..1 loop  -- correction_loop
-- compute the function
 z(1):= dy(x(1), y(1));
 z(k):= dy(x(k), y(k));
-- call corrected_formula
For i in 1..k/q loop
 cr(i). send_initial (a((i-1) * q+1..i * q), b((i-1) * q+1..i * q)),
     c((i-1) * q+1..i * q), zz, z(1), z(k);
End loop;
For i in 1..k/q loop
 cr(i). collect (y((i-1) * q+1..i * q)),
End loop;
End loop;
 y(k):= initial_value;
 x start:= x(k);
 n:= n+k;
 Exit when n = number of steps;
End loop;
End procedure;
```

Let us now discuss the above procedure.

1. The first thing required is to send two integers i and q to the

corrected_formula task i, where i is the specific task number and q is the sub-block length. Set n=0.

2. Calculate the predicted values y_{n+1} and y_{n+k}.
3. For s = 0,1.
4. Compute y'_{n+1} and y'_{n+k}.
5. Send $a_{r,0}$, $a_{r,1}$ and $a_{r,k}$; y'_n , y'_{n+1} and y'_{n+k} to corrector-formula task i, where r = (i-1)*q+1,..., i*q and i = 1,..., k/q.
6. Collect the corrected values y((i-1) * q+1 ,.., i * q) ;
7. Go to 3.
8. Increase the value of n by k, repeat the calculation to the end of the number of steps.

b) Algorithm 4

We present the parallel implementation of method 4. The algorithm here is similar to the algorithm (1) except the corrector_formula task which is different. The description of the algorithm is as follows:

1) A task type to compute the predicted values
2) A task type to evaluate the function
3) A task type to compute the corrected values

where 1 and 2 are the same as algorithm (1).

The task to compute the corrected values is given by:

```
Task type corrector_formula is
Declare
  z: vector (1..k);
Begin
```

```
Send_initial (i,q) to task i;
Declare
  ynew: vector (1..q);
Begin
Serial loop
Send (function_value, z);
Call procedure to compute the corrected values;
Collect the values (ytempi);
End loop;
End corrector_formula;
```

The main procedure is similar to that mentioned in algorithm (1) except the corrector task calling, which is given by

```
For i in 1..k/q loop
 cr(i). send (zz,z);        --send whole vector z(1..k)
End loop;                   --to task i;
```

5.5.1. The Comparison between Algorithm 3 and 4

There are some important differences to note between algorithm 3 and algorithm 4. The first is the amount of work which is required for each of them. Algorithm 3 requires for each block two predicted values, three function evaluations and k corrected values, whereas algorithm 4 requires for each block k predicted values, k function evaluations and k corrected values. Thus, it seems that algorithm 4 is slightly more expensive than algorithm 3

and the amount of work is great enough to keep all the processors busy (see appendix H), while algorithm 3 has less work to do. Second, algorithm 4 requires a large number of data to distribute and collect the results. We will see later the theoretical and experimental costs of these algorithms.

5.6 Theoretical Costs of Algorithms 3 and 4

In this section we shall give the sequential and parallel costs of algorithms 3 and 4 in terms of the three parameters which were mentioned in (5.2.4). We present first the theoretical cost of algorithm 3.

a) The Cost of Algorithm 3

1. Cost of Sequential Algorithm:

The total cost of the sequential algorithm is:

$$\left[(12k+6)m+5f \right] c/kh \ .$$

2. Cost of Parallel Algorithm

Since there is only one task type to compute the corrected values, then the time required for each block is:

$$kr/q \ + \left[\ 4kr/q \ + (8q+6)\ ks/q \ + 12km/p) \right] \ .$$

The predicted values and the function evaluations can be obtained in times 6m and 5f respectively. Hence, the total cost of the algorithm 3 is approximately

$$kr/q \ + \left[\ 4kr/q \ + (8q+6)\ ks/q \ + (6m+5f+12km/p) \right] \ c/kh \ .$$

Then the theoretical speedup is given by

$$S_3 = \frac{[(12k+6)m+5f]\ c/kh}{kr/q + [4kr/q + (8q+6)\,ks/q + (6m+5f+12km/p)]c/kh}$$

Assuming that the number of tasks is equal to the number of processors, hence

$$S_3 = \frac{[(12k+6)m+5f]\ c/kh}{pr + [4pr + (8q+6)ps + (6m+5f+12qm)]\ c/kh} \qquad (5.6.1)$$

which gives the speedup when $p=k/q$.

b) The Cost of Algorithm 4

1. Cost of Sequential Algorithm

The total cost of the sequential algorithm is

$$[\ (7k + (k^2 + kq)/q^2)m + 2kf\ ]\ c/kh\ .$$

2. Cost of Parallel Algorithm

First, the cost of computing the predicted values is

$$(r+s)k/q + [\ 2kr/q + (2+q)ks/q + km/p\ ]\ .$$

Similarly, the cost of evaluating the functions is

$$kr/q + [\ 4kr/q + 6ks + 2kf/p\ ]\ .$$

Finally, the corrector_formula task requires three rendezvous to send $(k+q+1)$ data items, and the procedure which used to compute the corrected values is given by:

```
sum:=0.0;
For r in 1..(i-1)q loop
sum:=sum+z(r);
Endloop;
```

$w(1) := y_n + h(zz/2 + sum + z((i-1)q+1)/2);$

For r in 2..q loop

$w(r) := w(r-1) + h(z(r+(i-1)q-1)+z(r+(i-1)q))/2;$

End loop;

$y(r) := w(r), \quad r=1,..,q.$

The above procedure requires $(6+2i)q$ multiplications and additions. This indicates that the number of operations differs from one task to another; this means the first task requires the smallest number of operations, whereas the k/q^{th} task requires the largest number of operations. On this basis, task 1 executes first, while k/q^{th} task executes last. The following table shows the arithmetic operations which are required for each task:

Table 5.6.1

Task Number	Number of Operations
1	8q
2	10q
3	12q
⋮	⋮
i	(6+2i)q
⋮	⋮
k/q	6q+2k

Therefore, if the $p=k/q$ (number of processors = number of tasks) then the number of arithmetic operations considered is the number which is required for the k/q^{th} task (takes the longest time). So, if $p \leq k/q$ then the number of operations which will be chosen depends on the number of processors which is used.

Assume that the number of operations is O(q) then the total cost of the corrected values is kr/q

$$kr/q + [4kr/q + (k+q+1)ks/q + O(q)m] .$$

So, the total cost of algorithm 4 is

$$(S+3r)k/q + \left[10kr/q + (2k + 9q + 4)\, ks/q + (k/p + O(q))m + 2kf/p \right] c/kh$$

If the number of tasks is equal to the number of processors, then the expected speedup is given by

$$S_4 = \frac{[\ (7k + (k^2 + kq)/q^2)m + 2kf\ ]\ c/kh}{(s+3r)p + [10pr + (2k + 9q + 4)ps + (7q + 2k)m + 2qf\ ]\ c/kh} \qquad (5.6.2)$$

5.7 The Results of Algorithms 3 and 4

In this section we present the theoretical and the experimental costs of algorithms 3 and 4, solving the test problems which were mentioned in (5.2.6).

a) The Results of Algorithm 3

The following table shows the time required for the computation, function evaluations, rendezvous and data sending when applying algorithm 3 to test problem (ii).

Table 5.7.1

Block Length k	Sub Block q	p	Computation Time	Function Evaluations	Rendezvous Time	Data Sending Time
500	250	1 2	93.3% 88.2%	0.12% 0.23%	0.93% 1.76%	5.0% 9.4%
500	100	1 2 3 4 5	92.4% 85.8% 80.1% 75.5% 71.1%	0.12% 0.23% 0.32% 0.40% 0.47%	2.5 % 4.6 % 6.4 % 8.0 % 9.5 %	4.9% 9.1% 12.8% 16.1% 19.0%

From the above table we observe that most of the calculation time is involved in the computation. The time spent on the function evaluations is very cheap.

Table 5.7.2 presents the sequential cost and the parallel cost of algorithm 3 when solving problem (iv), but only when the number of processors is equal to the number of tasks. The theoretical speedups S_3 are those which were in 5.6.1.

Table 5.7.2

Block Size k	Sub Block q	p	k/q	Sequential Cost	Parallel Cost	S_3
500	250	2	2	30.1	17.0	1.77
500	100	5	5	30.1	8.5	3.54
200	100	2	2	30.3	17.6	1.72
200	50	4	4	30.3	10.9	2.86
200	40	5	5	30.3	9.8	3.09

The following tables show the experimental timings, t(A), the theoretical cost, t(p) and the corresponding speedups S(A) and S(p) of method 3 solving problems (ii) and (iv).

Table 5.7.3

Block Length k	Sub Block q	Number of Tasks k/q	Number of Blocks	Number of Processors	t(A)	t(p)	S(A)	S(p)
500	250	2	20	1	32.0	31.9	1.0	1.0
				2	22.8	17.0	1.40	1.87
500	100	5	20	1	33.1	32.5	1.0	1.0
				2	20.7	17.5	1.59	1.86
				3	18.2	12.5	1.82	2.60
				4	17.9	9.9	1.85	3.28
				5	18.1	8.4	1.83	3.87
200	100	2	50	1	32.0	32.0	1.0	1.0
				2	20.7	17.0	1.55	1.88
200	50	4	50	1	33.5	33.4	1.0	1.0
				2	22.3	18.4	1.50	1.82
				3	17.5	13.8	1.91	2.42
				4	17.2	10.8	1.95	3.09
200	40	5	50	1	34.0	33.8	1.0	1.0
				2	22.9	18.8	1.55	1.80
				3	17.8	13.8	1.91	2.45
				4	17.4	11.2	1.95	3.01
				5	18.0	9.7	1.89	3.48

The experimental timings in the above table decrease as the number of processors increases, the speedup ratio varies between 1.59 to 1.85, when k=500 and q=100 and the number of processors varies between 2 and 4.

Table 5.7.4

Block Length k	Sub Block q	Number of Tasks k/q	Number of Blocks	Number of Processors	t(A)	t(p)	S(A)	S(p)
500	250	2	20	1	34.8	32.0	1.0	1.0
				2	24.5	17.0	1.42	1.94
500	100	5	20	1	33.3	32.5	1.0	1.0
				2	23.3	17.5	1.45	1.86
				3	18.3	12.5	1.82	2.60
				4	17.9	10.0	1.86	3.25
				5	18.7	8.5	1.78	3.82
200	100	2	50	1	32.7	32.6	1.0	1.0
				2	23.5	17.6	1.39	1.82
200	50	4	50	1	34.0	33.4	1.0	1.0
				2	23.8	18.4	1.43	1.82
				3	18.7	13.4	1.82	2.49
				4	18.4	10.9	1.85	3.06
200	40	5	50	1	34.2	33.8	1.0	1.0
				2	23.9	18.8	1.43	1.80
				3	18.8	13.8	1.82	2.45
				4	18.5	11.3	1.85	2.99
				5	19.0	9.8	1.80	3.45

Comparing the timings in tables 5.7.3 and 5.7.4, it is very clear that the timings in table 5.7.4 are slightly longer than the timings in table 5.7.3. This is due to the cost of problem (iv) which, as we believe, is approximately twice the cost of problem (ii) (see appendix G).

The following tables show the experimental timings, where the timings are measured in seconds, speedup, efficiencies and the accuracy for problems (i), (iii) and (iv) using method (3) for fixed k and q and a variable

number of steps, where the accuracy of the solution was measured by evaluating the l_∞ norm of the differences between the approximate solution and the exact solution at k equally spaced points.

Table 5.7.5

The block length = 200, q = 20 and k/q = 10

Test Problem (i)

Number of Steps	Number of Processors	t	S	E	Maximum Error
2000	1	11.0	1.0	1.0	9.43×10^{-5}
	2	8.7	1.26	0.63	
	3	8.4	1.31	0.44	
	4	8.1	1.36	0.34	
3000	1	13.2	1.0	1.0	2.71×10^{-5}
	2	10.5	1.26	0.63	
	3	9.8	1.35	0.45	
	4	9.2	1.43	0.36	
4000	1	14.0	1.0	1.0	1.09×10^{-5}
	2	11.2	1.25	0.63	
	3	10.5	1.33	0.44	
	4	10.1	1.39	0.35	
5000	1	16.9	1.0	1.0	5.70×10^{-6}
	2	13.7	1.23	0.62	
	3	12.6	1.34	0.45	
	4	12.3	1.37	0.34	
6000	1	20.0	1.0	1.0	3.49×10^{-6}
	2	16.1	1.24	0.62	
	3	15.0	1.33	0.44	
	4	14.8	1.35	0.34	
7000	1	23.1	1.0	1.0	2.32×10^{-6}
	2	18.5	1.25	0.63	
	3	17.9	1.29	0.43	
	4	16.6	1.39	0.35	
8000	1	24.9	1.0	1.0	1.64×10^{-6}
	2	20.3	1.23	0.62	
	3	19.1	1.30	0.43	
	4	18.2	1.37	0.34	

Table 5.7.6

The block length = 200, q = 20 and k/q = 10

Test Problem (iii)

Number of Steps	Number of Processors	t	S	E	Maximum Error
2000	1	12.3	1.0	1.0	8.96×10^{-3}
	2	9.9	1.24	0.62	
	3	9.4	1.31	0.44	
	4	9.0	1.37	0.34	
3000	1	13.8	1.0	1.0	2.40×10^{-3}
	2	10.7	1.28	0.64	
	3	10.3	1.34	0.45	
	4	10.2	1.35	0.34	
4000	1	16.2	1.0	1.0	9.88×10^{-4}
	2	12.9	1.26	0.63	
	3	12.5	1.30	0.43	
	4	12.4	1.31	0.33	
5000	1	18.5	1.0	1.0	5.01×10^{-4}
	2	14.9	1.24	0.62	
	3	14.4	1.28	0.43	
	4	14.0	1.32	0.33	
6000	1	21.3	1.0	1.0	2.89×10^{-4}
	2	17.6	1.21	0.61	
	3	16.2	1.31	0.43	
	4	16.1	1.32	0.33	
7000	1	23.6	1.0	1.0	1.82×10^{-4}
	2	20.8	1.13	0.57	
	3	18.2	1.30	0.43	
	4	17.8	1.33	0.33	
8000	1	25.4	1.0	1.0	1.22×10^{-4}
	2	21.8	1.17	0.58	
	3	20.1	1.26	0.42	
	4	19.8	1.28	0.32	

Table 5.7.7

The block length = 200, q = 20 and k/q = 10

Test Problem (iv)

Number of Steps	Number of Processors	t	S	E	Maximum Error
2000	1	13.3	1.0	1.0	4.62×10^{-4}
	2	11.2	1.19	0.59	
	3	10.3	1.29	0.43	
	4	9.9	1.34	0.34	
3000	1	14.8	1.0	1.0	1.41×10^{-4}
	2	12.0	1.23	0.62	
	3	10.9	1.36	0.45	
	4	10.1	1.47	0.37	
4000	1	17.2	1.0	1.0	6.04×10^{-5}
	2	13.7	1.26	0.63	
	3	12.2	1.41	0.47	
	4	11.9	1.45	0.36	
5000	1	19.6	1.0	1.0	3.14×10^{-5}
	2	15.9	1.23	0.62	
	3	14.3	1.37	0.46	
	4	13.7	1.43	0.36	
6000	1	21.4	1.0	1.0	1.85×10^{-5}
	2	17.3	1.24	0.62	
	3	16.3	1.31	0.44	
	4	15.6	1.37	0.34	
7000	1	24.7	1.0	1.0	1.18×10^{-5}
	2	19.8	1.25	0.63	
	3	18.7	1.32	0.44	
	4	17.6	1.40	0.35	
8000	1	27.5	1.0	1.0	7.99×10^{-6}
	2	22.3	1.23	0.62	
	3	20.4	1.35	0.45	
	4	20.2	1.36	0.34	

From the above tables we observe that the experimental timings decrease by increasing the number of processors. The speedup for problem (iv), as an example, increases from 1.23 to 1.43 when p varies between 2 and 4 and the corresponding efficiency decreases from 0.62 to 0.36 when the number of steps = 5000. We also notice that the accuracy of problem (iii) is high with h=0.01 and k = 200. Thus, for small h we can say that the algorithm may give a better accuracy. The following table shows the results of problem (iii) when k=10 and h varies.

Table 5.7.8

Step Size h	Maximum Error	Computed Solution
0.01	1.21×10^{-6}	$1.773017 \times 10^{+1}$
0.005	1.79×10^{-7}	$1.773012 \times 10^{+1}$
0.0025	2.88×10^{-8}	$1.773012 \times 10^{+1}$
0.000125	5.18×10^{-9}	$1.77302 \times 10^{+1}$

We see that the method is more accurate as k and h are decreased. Figures 5.7.1 and 5.7.2 show the |log max error| plot for the experimental timing for a fixed number of processors, where the results are obtained from tables 5.7.6 and 5.7.7.

We can see that the maximum number of processors which are used here is 4. Also we notice that the maximum and the minimum values of |log max error| are 3.91 and 2.05 and 5.10 and 3.34 for problems (iii) and (iv) respectively.

b) The Results of Algorithm 4

We now turn to computational results of algorithm 4. The following table shows the theoretical cost of arithmetic operations of algorithm 4 when solving any problem and the number of steps is 10,000.

Table 5.7.9

Block length k	Sub block q	Number of Processors			
		1	2	4	5
500	250	25.0	13.8		
500	100	32.5	21.3	15.6	8.5
200	100	25.0	13.8		
200	50	30.0	20.0	18.1	-
200	40	32.5	21.3	14.6	8.5

The above table shows that for fixed k and variable q the cost is increasing when q is decreasing.

The following tables are meant to show the theoretical cost of function evaluations of algorithm 4, if problems (ii), (iii) and (iv) are used.

Table 5.7.10

Block length k	Sub block q	Number of Processors				
		1	2	3	4	5
500	250	7.8	3.9			
500	100	7.8	3.9	2.6	1.95	1.56
200	100	7.8	3.9			
200	50	7.8	3.9	2.6	1.95	1.56
200	40	7.8	3.9	2.6	1.95	1.56

Table 5.7.11

Block length k	Sub block q	Number of Processors				
		1	2	3	4	5
500	250	8.2	4.1			
500	100	8.2	4.1	2.7	2.0	1.6
200	100	8.2	4.1			
200	50	8.2	4.1	2.7	2.0	
200	40	8.2	4.1	2.7	2.0	1.6

Table 5.7.12

Block length k	Sub block q	Number of Processors				
		1	2	3	4	5
500	250	16.0	8.0			
500	100	16.0	8.0	5.3	4.0	3.2
200	100	16.0	8.0			
200	50	16.0	8.0	5.3	4.0	
200	40	16.0	8.0	5.3	4.0	3.2

In the light of the above tables, we notice that the function evaluations cost of problem (iv) is approximately twice the cost of function evaluations of problems (ii) and (iii).

The following tables show the experimental timings, t(A), the theoretical timings, t(P) and the corresponding speedups S(A) and S(P) of method 4, solving problems (iii) and (iv)

Table 5.7.13

Block Length k	Sub Block q	Number of Tasks k/q	Number of Blocks	Number of Processors	t(A)	t(p)	S(A)	S(p)
500	250	2	20	1 2	39.4 24.6	36.2 19.9	1.0 1.60	1.0 1.82
500	100	5	20	1 2 3 4 5	48.9 33.1 26.5 24.2 23.1	46.0 29.8 -- 23.0 15.5	1.0 1.48 1.85 2.02 2.11	1.0 1.54 -- 2.00 2.96
200	100	2	50	1 2	40.2 24.9	36.9 22.1	1.0 1.61	1.0 1.67
200	50	4	50	1 2 3 4	49.0 32.7 28.9 27.5	45.2 31.1 -- 27.1	1.0 1.50 1.70 1.78	1.0 1.45 -- 1.67
200	40	5	50	1 2 3 4 5	52.6 36.8 28.9 26.7 25.0	49.1 32.8 -- 25.0 18.5	1.0 1.45 1.82 1.97 2.10	1.0 1.50 -- 1.96 2.65

The actual timings in the above table decrease as the number of processors increases, the speedup ratio varies between 1.48 to 2.11 for k=500, q=100 and the number of processors varies between 2 and 5. We also notice that for fixed k, the time increases as q decreases; this means for small q we get more tasks which require more communication and data distribution.

Table 5.7.14

Block Length k	Sub Block q	Number of Tasks k/q	Number of Blocks	Number of Processors	t(A)	t(p)	S(A)	S(p)
500	250	2	20	1 2	48.9 29.3	44.4 25.2	1.0 1.67	1.0 1.76
500	100	5	20	1 2 3 4 5	58.4 37.9 30.5 28.4 27.2	54.3 35.1 -- 25.4 17.5	1.0 1.54 1.91 2.06 2.15	1.0 1.55 -- 2.14 3.10
200	100	2	50	1 2	52.9 30.7	45.6 26.4	1.0 1.72	1.0 1.73
200	50	4	50	1 2 3 4	56.9 37.2 31.4 30.9	53.4 35.4 -- 29.5	1.0 1.52 1.81 1.84	1.0 1.51 -- 1.81
200	40	5	50	1 2 3 4 5	59.0 38.8 35.4 32.5 28.6	57.3 38.1 -- 27.4 20.5	1.0 1.52 1.67 1.81 2.06	1.0 1.50 -- 2.09 2.79

It is clear that the timings in table 5.7.14 are longer than the timings in table 5.7.13 and the time difference is approximately 8 seconds (if the number of processors is one) and 4 seconds (if the number of processors is two).

The following tables give the experimental results, speedup, efficiency and the accuracy of problems (iii) and (iv) using algorithm 4.

Table 5.7.15

k=200, q=20, k/q=10

Number of Steps	Number of Processors	t	S	E	Maximum Error
2000	1	22.4	1.0	1.0	6.89×10^{-3}
	2	15.6	1.44	0.72	
	3	12.6	1.78	0.59	
	4	12.0	1.87	0.47	
3000	1	27.2	1.0	1.0	1.89×10^{-3}
	2	20.1	1.35	0.68	
	3	15.3	1.78	0.59	
	4	14.6	1.86	0.47	
4000	1	34.6	1.0	1.0	7.68×10^{-4}
	2	25.3	1.37	0.68	
	3	19.0	1.82	0.60	
	4	18.2	1.90	0.48	
5000	1	41.1	1.0	1.0	3.85×10^{-4}
	2	30.3	1.36	0.68	
	3	22.7	1.81	0.60	
	4	21.7	1.89	0.47	
6000	1	46.9	1.0	1.0	2.20×10^{-4}
	2	34.7	1.35	0.68	
	3	26.2	1.79	0.60	
	4	24.8	1.89	0.47	
7000	1	53.4	1.0	1.0	1.38×10^{-4}
	2	38.9	1.37	0.68	
	3	29.5	1.81	0.60	
	4	27.8	1.92	0.48	
8000	1	58.9	1.0	1.0	9.16×10^{-5}
	2	42.1	1.40	0.70	
	3	32.7	1.80	0.60	
	4	30.5	1.93	0.48	

Table 5.7.16

Block length = 200, q = 20 and k/q = 10

Number of Steps	Number of Processors	t	S	E	Maximum Error
2000	1 2 3 4	23.5 16.2 13.1 12.3	1.0 1.45 1.79 1.91	1.0 0.73 0.60 0.48	2.98×10^{-5}
3000	1 2 3 4	28.1 19.5 15.7 14.8	1.0 1.44 1.79 1.90	1.0 0.72 0.60 0.48	8.78×10^{-6}
4000	1 2 3 4	34.9 24.4 19.3 18.3	1.0 1.43 1.81 1.91	1.0 0.72 0.60 0.48	3.69×10^{-6}
5000	1 2 3 4	43.1 29.9 23.9 22.8	1.0 1.44 1.80 1.89	1.0 0.72 0.60 0.42	1.88×10^{-6}
6000	1 2 3 4	46.8 32.7 26.1 24.8	1.0 1.43 1.79 1.89	1.0 0.72 0.60 0.60	1.08×10^{-6}
7000	1 2 3 4	56.0 40.0 30.9 28.7	1.0 1.40 1.81 1.95	1.0 0.70 0.60 0.49	6.85×10^{-7}
8000	1 2 3 4	62.1 43.7 34.7 32.2	1.0 1.42 1.79 1.93	1.0 0.71 0.60 0.48	4.59×10^{-7}

Figures 5.7.3 and 5.7.4 show the |log max error| plot for the experimental timing for a fixed number of processors, where the results are obtained from tables 5.7.15 and 5.7.16.

The above tables show the results of method 4, using different values of h. We observe that the accuracy increases as h decreases which might be expected. As mentioned earlier, because the global error is of $O(h^2)$, we can say that the best accuracy may be obtained when h is small enough and the behaviour like ch^2.

For large k and/or h, the principal truncation error may, however, be misleading, and the next term of the error which is given may dominate,

$$\varepsilon_k = \frac{(kh)^4}{4!} y_n^{(4)} - \frac{h}{2}\left[2\left(\frac{h^3}{3!} + \frac{8h^3}{3!} + \ldots + \frac{(k-1)^3}{3!} h^3\right) + \frac{k^3h^3}{3!}\right] y_n^{(4)}$$

$$= \left[\frac{k^4h^4}{24} - \frac{h^4}{6} \left(1+8+27+\ldots+(k-1)^3\right) - \frac{k^3h^4}{12}\right] y_n^{(4)}$$

$$= \left[\frac{k^4h^4}{24} - \left[\frac{h^4}{6}\right] \frac{k^2(k-1)^2}{4} - \frac{k^3h^4}{12}\right] y_n^{(4)}$$

$$= \left[\frac{k^4h^4}{24} - \frac{k^4h^4}{24} + \frac{k^3h^4}{12} - \frac{k^2h^4}{24} - \frac{k^3h^4}{12}\right] y_n^{(4)}$$

$$= -\frac{k^2h^4}{24} = O(k^2h^4) \quad .$$

For fixed k and sufficiently large k, this will always essentially be larger than the principal term kh^3. For this reason, we will increase the number of iterations to obtain good results and better accuracy.

Tables 5.7.17 and 5.7.18 show numerical results of problem (i) and (ii) using method 4.

Table 5.7.17

Number of Steps	Step Size	"1" Correction	
		Computed Solution	Max Error
1000	0.001	2.702708163	1.55×10^{-2}
2000	0.002	2.714080847	4.20×10^{-3}
3000	0.00033	2.716366958	1.91×10^{-3}
4000	0.00025	2.717191054	1.09×10^{-3}
5000	0.0002	2.717578493	7.03×10^{-4}
6000	0.0001667	2.717790904	4.91×10^{-4}
7000	0.0001429	2.717919853	3.62×10^{-4}
8000	0.000125	2.718003443	2.77×10^{-4}
Number of Steps	Step Size	"2" Corrections	
		Computed Solution	Max Error
1000	0.001	2.717509562	7.72×10^{-4}
2000	0.002	2.718177314	1.05×10^{-4}
3000	0.00033	2.718250034	3.18×10^{-5}
4000	0.00025	2.718268239	1.36×10^{-5}
5000	0.0002	2.718274817	7.01×10^{-6}
6000	0.0001667	2.718277499	4.08×10^{-6}
7000	0.0001429	2.718279251	2.58×10^{-6}
8000	0.000125	2.718280097	1.73×10^{-6}

Table 5.7.17 (Cont)

Number of Steps	Step Size	"3" Corrections	
		Computed Solution	Max Error
1000	0.001	2.718251359	3.05×10^{-5}
2000	0.002	2.718278005	2.03×10^{-6}
3000	0.00033	2.718281430	3.98×10^{-7}
4000	0.00025	2.718281707	1.22×10^{-7}
5000	0.0002	2.718281781	4.70×10^{-8}
6000	0.0001667	2.718281807	2.09×10^{-8}
7000	0.0001429	2.718281834	1.01×10^{-8}
8000	0.000125	2.718281823	5.13×10^{-9}
Number of Steps	Step Size	"4" Corrections	
		Computed Solution	Max Error
1000	0.001	2.718281037	7.92×10^{-7}
2000	0.002	2.718281850	2.29×10^{-8}
3000	0.00033	2.718281849	2.05×10^{-8}
4000	0.00025	2.718281841	1.30×10^{-8}
5000	0.0002	2.718281837	8.69×10^{-9}
6000	0.0001667	2.718281834	6.14×10^{-9}
7000	0.0001429	2.718281833	4.55×10^{-9}
8000	0.000125	2.718281832	3.50×10^{-9}

Table 5.7.18

Number of Steps	Step Size	"1" Correction	
		Computed Solution	Max Error
1000	0.02	$2.325877653 \times 10^{-1}$	1.68×10^{-2}
2000	0.01	$2.212278396 \times 10^{-1}$	3.29×10^{-3}
3000	0.0067	$2.194162629 \times 10^{-1}$	1.26×10^{-3}
4000	0.005	$2.18837903 \times 10^{-1}$	6.43×10^{-6}
5000	0.004	$2.185941407 \times 10^{-1}$	3.87×10^{-6}
6000	0.0033	$2.18469392 \times 10^{-1}$	2.57×10^{-6}
7000	0.002857	$2.183975826 \times 10^{-1}$	1.83×10^{-4}
8000	0.0025	$2.183525823 \times 10^{-1}$	1.37×10^{-6}
Number of Steps	Step Size	"2" Corrections	
		Computed Solution	Max Error
1000	0.02	$2.1112851019 \times 10^{-1}$	8.18×10^{-3}
2000	0.01	2.1760429×10^{-1}	6.59×10^{-4}
3000	0.0067	$2.180585862 \times 10^{-1}$	1.67×10^{-4}
4000	0.005	$2.1815572839 \times 10^{-1}$	6.44×10^{-5}
5000	0.004	$2.181876770 \times 10^{-1}$	3.11×10^{-5}
6000	0.0033	$2.1820104538 \times 10^{-1}$	1.72×10^{-5}
7000	0.002857	$2.1820757428 \times 10^{-1}$	1.05×10^{-5}
8000	0.0025	$2.1821112723 \times 10^{-1}$	6.88×10^{-6}

Table 5.7.18 (Cont)

Number of Steps	Step Size	"3" Corrections	
		Computed Solution	Max Error
1000	0.02	$2.19864506 \times 10^{-1}$	1.91×10^{-3}
2000	0.01	$2.1833006659 \times 10^{-1}$	1.21×10^{-4}
3000	0.0067	$2.1823858766 \times 10^{-1}$	2.17×10^{-5}
4000	0.005	$2.18224049 \times 10^{-1}$	6.38×10^{-6}
5000	0.004	$2.182202930 \times 10^{-1}$	2.47×10^{-6}
6000	0.0033	$2.1821900445 \times 10^{-1}$	1.14×10^{-6}
7000	0.002857	$2.1821847227 \times 10^{-1}$	5.94×10^{-7}
8000	0.00257	$2.1821822183 \times 10^{-1}$	3.38×10^{-7}
Number of Steps	Step Size	"4" Corrections	
		Computed Solution	Max Error
1000	0.02	$2.17705056 \times 10^{-1}$	5.94×10^{-4}
2000	0.01	$2.181991003 \times 10^{-1}$	2.02×10^{-5}
3000	0.0067	$2.1821538714 \times 10^{-1}$	2.62×10^{-6}
4000	0.005	$2.182172910 \times 10^{-1}$	6.21×10^{-7}
5000	0.004	$2.182176872 \times 10^{-1}$	2.09×10^{-7}
6000	0.0033	$2.182780323 \times 10^{-1}$	8.91×10^{-8}
7000	0.002857	$2.182178459 \times 10^{-1}$	4.51×10^{-8}
8000	0.00257	$2.82176465 \times 10^{-1}$	2.60×10^{-8}

In table 5.7.17 we present the results of problem (i) when k=200 and q=20 for different values of h. We can observe that when using more iteration we expect the solution will be quite accurate. Exact solutions are also shown. These figures show quite nicely the effect of increased accuracy associated with decreasing values of h.

In table 5.7.18 we give the results of problem (ii) when k=100 and q=10 for different values of h. Exactly the same remarks apply to table 5.7.18. Finally, the time required will be increased.

Figures 5.7.5 and 5.7.6 show the |log max error| plot for |log h| for fixed block length and sub-block length, where the results are obtained from tables 5.7.17 and 5.7.18 respectively.

5.8 Conclusion

We have shown in this chapter three types of methods, each of which has different local truncation error and easily adapted to parallel implementation.

From the results of the computation we can give the following remarks:

1. Method 3 is preferable to use when k is small. Also the time required is slightly cheaper when compared to the other methods.
2. Method 1 is not good enough, because the local truncation error depends on k^3h^3.
3. The best method we recommended is the fourth method. It is more accurate and gives better speedups. Two corrector iterations are more efficient for small stepsize and four iterations for large stepsize.

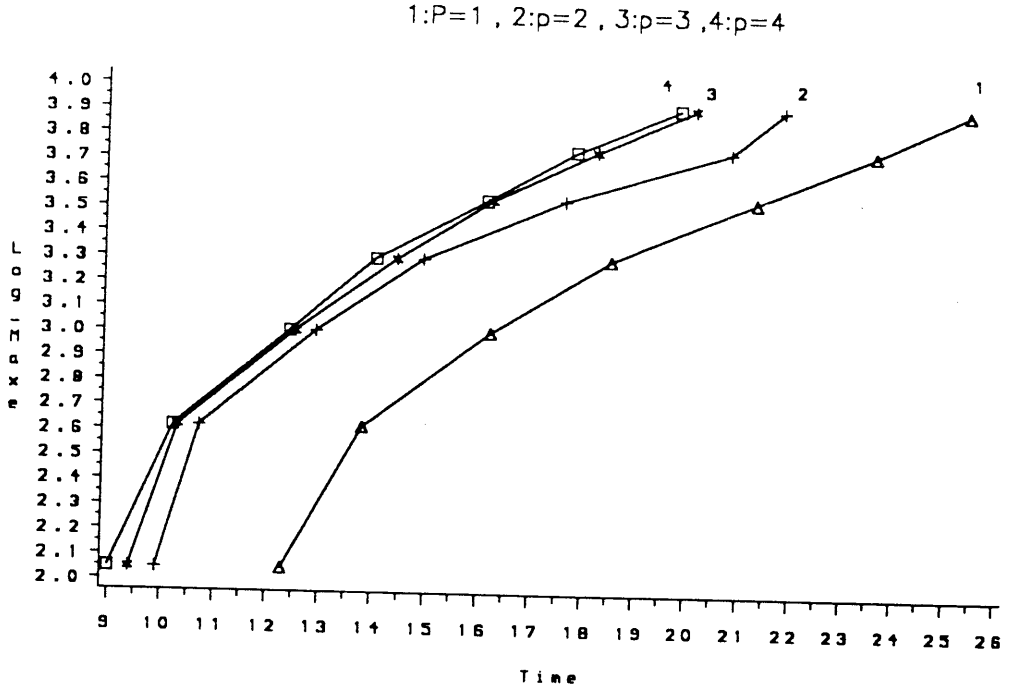


Figure 5.7.1 _ k=200, q=20

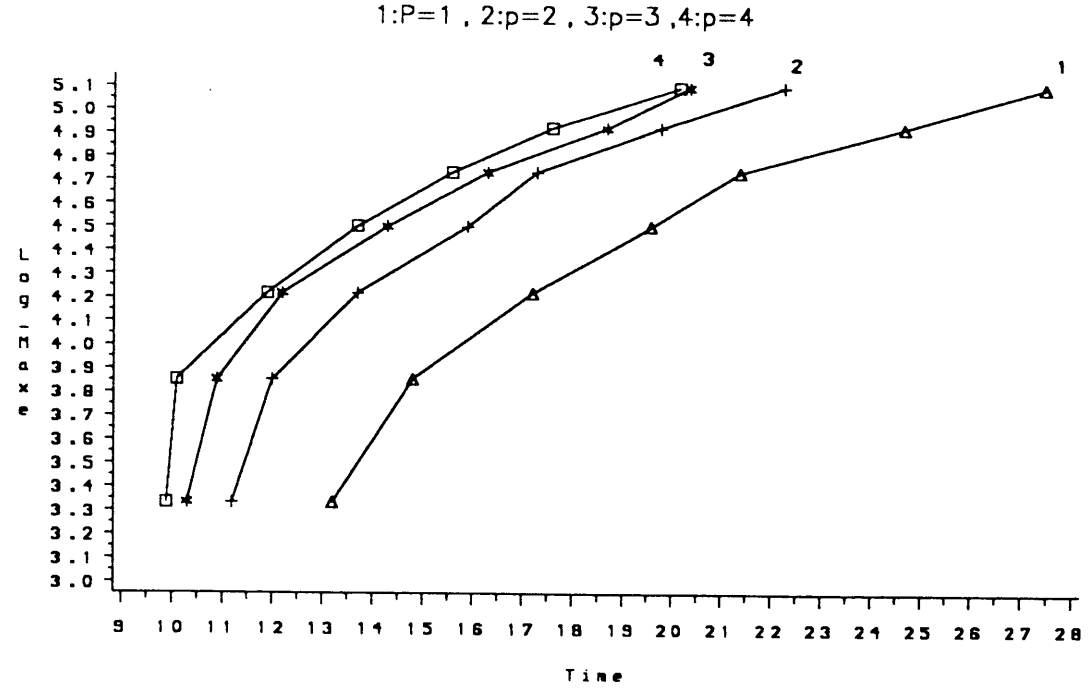


Figure 5.7.2 _ k=200, q=20

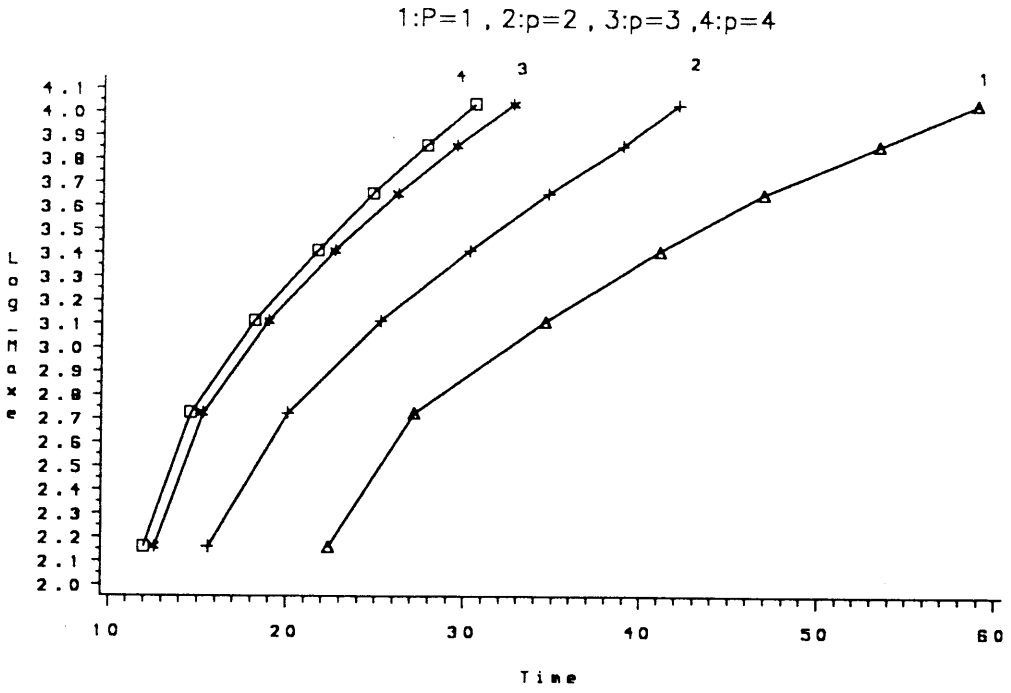


Figure 5.7.3 __ k=200, q=20

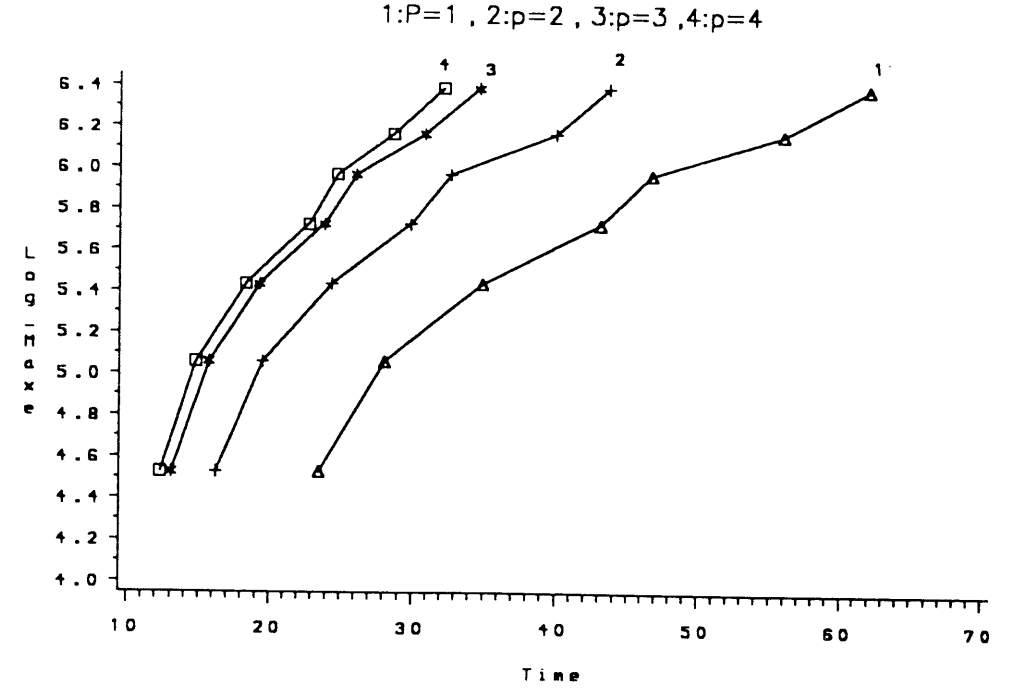


Figure 5.7.4 __ k=200, q=20

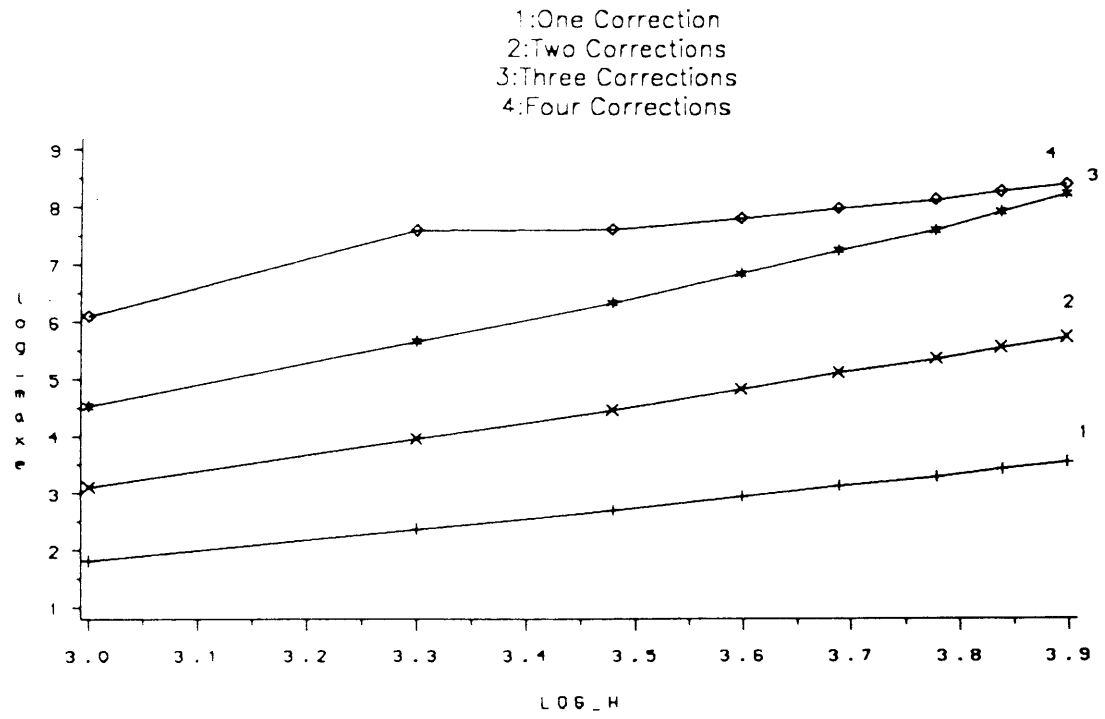


Figure 5.7.5 _ k=200, q=20

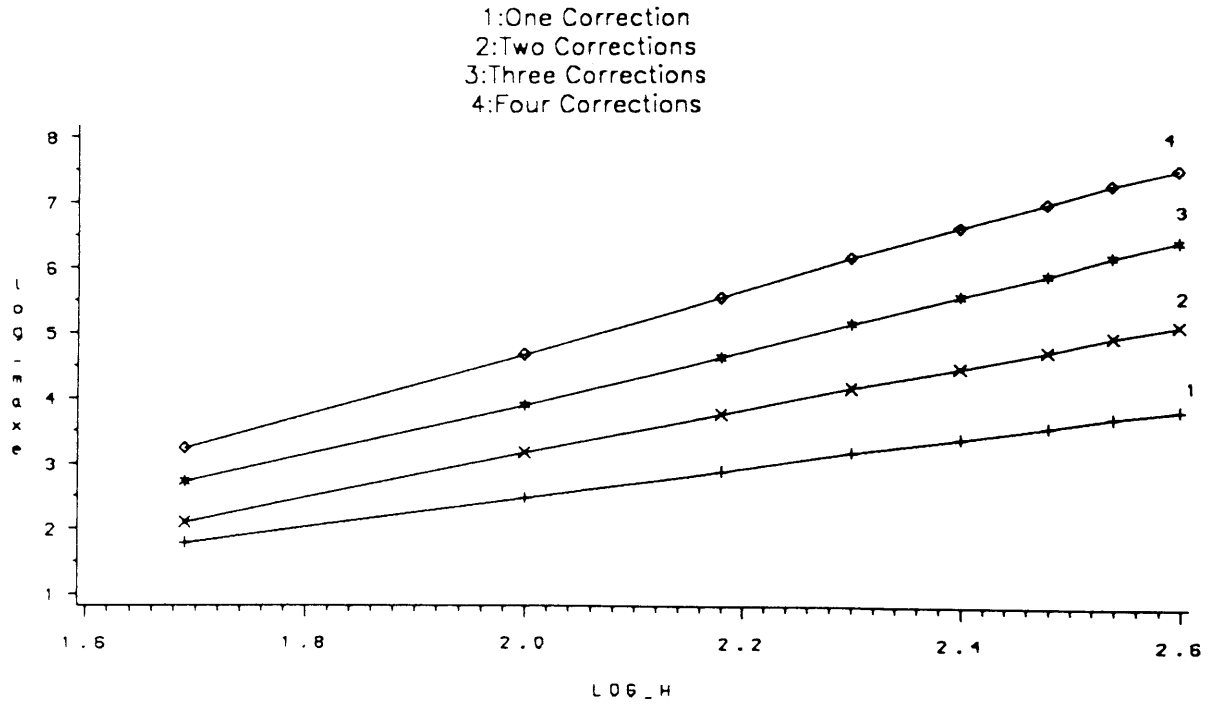


Figure 5.7.6_ k=100, q=10

CHAPTER 6

Parallel Solution of Initial Value Problems Using Fourth Order Block Methods

6.1 Introduction

In this Chapter, we shall present a fourth order method which is applicable for arbitrary block length to solve ordinary differential equation (refer to 5.1.1). The coefficients of the method satisfy the characteristic polynomials, which are given as:

$$\sum_{j=0}^{k} a_{rj} = r, \quad \sum_{j=0}^{k} j a_{rj} = r^2/2 , \quad \sum_{j=0}^{k} j^2 a_{rj} = r^3/3$$

and $\sum_{j=0}^{k} j^3 a_{rj} = r^4/4 ; \quad r = 1,2,..,k.$

Our aim was to produce fourth order methods which might be very efficient to implement. As far as order of the accuracy and the stability are concerned, if we require a method of order S to be achieved then the local truncation error will normally be of order $k^{s+1}h^{s+1}$ which is a very high dependence upon the block length k. We seek some methods where the local truncation errors have a lower power of k, as example, $k^s h^{s+1}$, but then over experience with order 2 methods suggests that the stability problem will be possibly serious.

6.2 The Outline of a Fourth Order Method

Consider the fourth order method

$$y_{n+r,s+1} = y_n + h(a_{r,o}y'_n + a_{r,1}y'_{n+1,s} + a_{r,k-1}y'_{n+k-1,s} + a_{r,k}y'_{n+k,s}) \quad (6.2.1)$$

$$, \ r = 1,2,..,k, \qquad s = 0,1.$$

where the coefficients of the method is given by

$$a_{r,k} = (3r^4 - 4kr^3 + 6kr^2 - 6r^2)/12k(k-1) \ ,$$

$$a_{r,k-1} = \left[(r^3/3 - r^2/2) - k(k-1)a_{r,k} \right] / (k-1)(k-2) \ ,$$

$$a_{r,k} = r^2/2 - (k-1)\, a_{r,k-1} - k\, a_{r,k} \quad \text{and}$$

$$a_{r,o} = r - a_{r,1} - a_{r,k-1} - a_{r,k}; \qquad r = 1,2,..,k.$$

We shall give the values of the coefficients for r = 1,2,3,k-1 and k for any k, as follows:

1) When r = 1:

$$a_{1,0} = (6k^3 - 22k^2 + 23k - 6) / 12k\,(k-1)(k-2) \ ,$$

$$a_{1,1} = (6k^3 - 14k^2 + 7k) / 12k(k-1)(k-2) \ ,$$

$$a_{1,k-1} = (k-2k^2) / 12k(k-1)(k-2) \quad \text{and}$$

$$a_{1,k} = (2k^2 - 7k + 6) / 12k(k-1)(k-2).$$

2) When r = 2:

$$a_{2,0} = (16k^2 - 56k + 48) / 12k(k-1)(k-2) \ ,$$

$$a_{2,1} = (24k^3 - 88k^2 + 80k) / 12k(k-1)(k-2) \ ,$$

$$a_{2,k-1} = (8k^2 - 16k) / 12k(k-1)(k-2) \quad \text{and}$$

$$a_{2,k} = (-8k^2 + 40k - 48) / 12k(k-1)(k-2) \ .$$

3) When r = 3:

$$a_{3,0} = (-18k^3 + 162k^2 - 441k + 378) / 12k(k-1)(k-2) \ ,$$

$$a_{3,1} = (54k^3 - 270k^2 + 351k) / 12k(k-1)(k-2) \ ,$$

$$a_{3,k-1} = (54k^2 - 135k) / 12k(k-1)(k-2) \quad \text{and}$$

$$a_{3,k} = (-54k^2 + 297k - 378) / 12k(k-1)(k-2) \ .$$

4) When r = k-1

$a_{k-1,0} = (-k^5 + 8k^4 - 18k^3 + 10k^2 + 7k - 6) / 12k(k-1)(k-2)$,

$a_{k-1,1} = (k^5 - 2k^4 + 2k^2 - k) / 12k(k-1)(k-2)$,

$a_{k-1,k-1} = (k^5 - 2k^4 - 6k^3 + 14k^2 - 7k) / 12k(k-1)(k-2)$ and

$a_{k-1,k} = (-k^5 + 8k^4 - 24k^3 + 34k^2 - 23k + 6) / 12k(k-1)(k-2)$.

5) When r = k:

$a_{k,0} = (-k^5 + 8k^4 - 18k^3 + 12k^2) / 12k(k-1)(k-2)$,

$a_{k,1} = (k^5 - 2k^4) / 12k(k-1)(k-2)$,

$a_{k,k-1} = (k^5 - 2k^4) / 12k(k-1)k-2)$ and

$a_{k,k} = (-k^5 + 8k^4 - 18k^3 + 12k^2) / 12k(k-1)(k-2)$.

We notice from the above values that the coefficients of the method depend on k. This means the coefficients take different values for different k.

As an example, we should give the formulae of the method when k = 3 and 4.

1) When k = 3, the formulae are

$$\left.\begin{aligned} y_{n+1} &= y_n + \tfrac{h}{24}\left[9y'_n + 19y'_{n+1} - 5y'_{n+2} + y'_{n+3} \right] \\ y_{n+2} &= y_n + \tfrac{h}{3}\left[y'_n + 4y'_{n+1} + y'_{n+2} \right] \\ y_{n+3} &= y_n + \tfrac{3h}{8}\left[y'_n + 3y'_{n+1} + 3y'_{n+2} + y'_{n+3} \right] \end{aligned}\right\} \qquad (6.2.2)$$

2) When k = 4, the formulae are:

$$\left.\begin{aligned}
y_{n+1} &= y_n + \frac{h}{144}\left[\, 59y'_n + 94y'_{n+1} - 14y'_{n+3} + 5y'_{n+4} \,\right] \\
y_{n+2} &= y_n + \frac{h}{144}\left[\, 40y'_n + 224y'_{n+1} + 32y'_{n+3} - 8y'_{n+4} \,\right] \\
y_{n+3} &= y_n + \frac{h}{144}\left[\, 27y'_n + 270y'_{n+1} + 162y'_{n+3} - 27y'_{n+4} \,\right] \\
y_{n+4} &= y_n + \frac{h}{144}\left[\, 32y'_n + 256y'_{n+1} + 256y'_{n+3} + 32y'_{n+4} \,\right]
\end{aligned}\right\} \qquad (6.2.3)$$

6.3 Principal Local Truncation Error

The kth component of local truncation error of the above method (6.2.1) is given by:

$$\varepsilon_k = 2k^4(k^4 - 8k^3 + 22k^2 - 25k + 10)h^5y_n^{(5)}/\ 1440k(k-1)(k-2)$$
$$= k^3(k^2-5k+5)h^5y_n^{(5)}/\ 720 \quad = O(k^5h^5).$$

The local truncation error $\underline{\varepsilon}$ for (6.2.2) and (6.2.3) are

$$\underline{\varepsilon} = \left\{ -19h^5y_n^{(5)}/720\ ,\ -h^5y_n^{(5)}/90\ ,\ -3h^5y_n^{(5)}/80 \right\} \text{ and}$$

$$\underline{\varepsilon} = \left\{ -11h^5y_n^{(5)}/180\ ,167h^5y_n^{(5)}/4320\ ,3h^5y_n^{(5)}/20\ ,8h^5y_n^{(5)}/90 \right\}$$

respectively.

6.4 Stability Analysis

To analyse the stability of (6.2.1) we will use the concept of the stability which is discussed in Chapter 5 when the method (6.2.1) is applied to the test equation, the results are as follows:

The three cofactors for q(H) are

$$q^{(1)}(H) = 1 - \frac{(-6k^4-18k^3+12k^2)H}{12k(k-1)(k-2)}$$

$$+\frac{(12k^8-60k^7+66k^6+102k^5-268k^4+196k^3-49k^2)H^2}{144k^2(k-1)^2(k-2)^2}$$

$$-\frac{(72k^{11}-744k^{10}+3192k^9-738k^8+9942k^7-7812k^6+3318k^5-388k^4)H^3}{1728k^3(k-1)^3(k-2)^3}$$

$$q^{(2)}(H) = (-1)^{2k-2}\left[\frac{(2k^7-5k^6+2k^5+4k^4-4k^3+k^2)H^2}{144k^2(k-1)^2(k-2)^2}\right.$$

$$\left.+\frac{(-2k^{10}+78k^9-204k^8+276k^7-204k^6+78k^5-12k^4)H^3}{1728k^3(k-1)^3(k-2)^3}\right]$$

and

$$q^{(3)}(H) = (-1)^{2k-2}\left[\frac{(-2k^7+11k^6-20k^5+12k^4)H^2}{144k^2(k-1)^2k-2)^2} -\right.$$

$$\left.-\frac{(12k^{10}-90k^9+264k^8-378k^7+264k^6-72k^5)H^2}{1728k^3(k-1)^3(k-2)^3}\right]$$

The three cofactors for $q_k(H)$ are

$$q_k^{(1)}(H) = 1 + \frac{(-k^5+8k^4-18k^3+12k^2)H}{12k(k-1)(k-2)}$$

$$+\frac{(-2k^7-25k^6+148k^5-268k^4+196k^3-49k^2)H^2}{144k^2(k-1)^2(k-2)^2}$$

$$+\frac{(36k^{11}-444k^{10}-2178k^9-5604k^8+8244k^7-6972k^6+3150k^5-588k^4)H}{1728k^3(k-1)^3(k-2)^3}$$

$$q_k^{(2)}(H) = (-1)^{2k-2}\left[\frac{(4k^4-4k^3+k^2)H^2}{144k^2(k-1)^2k-2)^2} + \frac{(12k^8-54k^7+84k^6-54k^5+12k^4)H^3}{1728k^3(k-1)^3(k-2)^3}\right]$$

and

$$q_k^{(3)}(H) = (-1)^{2k-2}\left[\frac{(k^5-2k^4)H}{12k(k-1)(k-2)} + \frac{(12k^8-58k^7+97k^6-64k^5+12k^4)H^2}{144k^2(k-1)^2k-2)^2} + \frac{(36k^{11}-276k^{10}+846k^9-1320k^8+1098k^7-456k^6+72k^5)H^3}{1728k^3(k-1)^3(k-2)^3}\right]$$

Therefore, the polynomials q(H) and $q_k(H)$ are

$$q(H) = (a_1-a_2H+a_3H^2-a_4H^3)/a_1 \quad \text{and}$$

$$q_k(H) = (a_1+a_2H+a_3H^2+a_4H^3)/a_1 \quad \text{where}$$

$a_1 = 1728k^3(k-1)^3(k-2)^3,$

$= 1728k^9-1552k^8+57024k^7-108864k^6+114048k^5-62208k^4+13824k^3,$

$a_2 = 144k^2(k-1)^2(k-2)^2(6k^4-18k^3+12k^2),$

$= 864k^{10}-7776k^9+28512k^8-54432k^7+57024k^6-31104k^5+6912k^4,$

$a_3 = 12k(k-1)(k-2)(12k^8-60k^7+72k^6-84k^5-252k^4+192k^3-48k^2),$

$= 144k^{11}-1152k^{10}+3312k^9-3024k^8-4320k^7+13392k^6-13536k^5+6336k^4-1152k^3,$

$a_4 = 72k^{11}-720k^{10}+3024k^9-6912k^8+9288k^7-7344k^6+3168k^5-576k^4.$

Hence, the amplification factor is

$$\left|\frac{q_k(H)}{q(H)}\right| = \left|\frac{(a_1+a_2H) + H^2(a_3+a_4H)}{(a_1-a_2H) + H^2(a_3-a_4H)}\right|$$

which is less than one for all $H < 0$. So the method is A-stable for all k.

We check the cases when $k = 3$ and $k = 4$.

Apply the system (6.2.2) to the test equation, then the values of q(H) and $q_3(H)$ are as follows:

$$q(H) = (576-864H+528H^2-144H^3)/576 \quad \text{and}$$

$$q_3(H) = (576+864H+528H^2+144H^3)/576.$$

Hence, the amplification factor is

$$\left|\frac{q_3(H)}{q(H)}\right| = \left|\frac{576+864H+528H^2+144H^3}{576-864H+528H^2-144H^3}\right| < 1,$$

then the method is A-stable when k=3.

Similarly, apply the system (6.2.3) to the test equation, we get

$$\begin{bmatrix} 1-94H/144 & 0 & 14H/144 & -5H/144 \\ -22H/144 & 1 & -32H/144 & +8H/144 \\ -270H/144 & 0 & 1-162H/144 & +27H/144 \\ -256H/144 & 0 & -256H/144 & 1-32H/144 \end{bmatrix} \begin{bmatrix} y_{n+1} \\ y_{n+2} \\ y_{n+3} \\ y_{n+4} \end{bmatrix} = \begin{bmatrix} 1+59H/144 \\ 1+40H/144 \\ 1+27H/144 \\ 1+32H/144 \end{bmatrix}$$

Therefore, q(H) is given by

$$q(H) = (1-94H/144)\left[(1-162H/144)(1-32H/144)-(27H/144)(-256/144)\right]$$
$$-(14H/144)\left[(-270H/144)(1-32H/144)+(256H/144)(27H/144)\right]$$
$$-(5H/144)\left[(-270H/144)(-256H/144)+(256H/144)(1-162H/144)\right]$$
$$q(H) = (2985984-4575168H+4727808H^2-1492992H^3)/2985984$$

q (H) is given by

$$q_4(H) = \begin{bmatrix} 1-94H/144 & 0 & 14H/144 & 1+59H/144 \\ -224H/144 & 1 & -32H/144 & 1+40H/144 \\ -270H/144 & 0 & 1-162H/144 & 1+27H/144 \\ -256H/144 & 0 & -256H/144 & 1+32H/144 \end{bmatrix}$$

$$= (1-94H/144)\left[(1-162H/144)(1+32H/144H)-(1+27H/144)(-256H/144)\right]$$
$$- (14H/144)\left[(-270H/144)(1+32H/144)+(256H/144)(1+27H/144)\right]$$
$$+ (1+59H/144)\left[(-270H/144)(-256H/144)+(256H/144)(1-162H/144)\right]$$

$= (2985984+663552H-1456704H^2-162432H^3)/2985984$

$+ (28224H^2+24192H^3)/2985984 + (5308416H+6156288H^2+1631232H^3)/2985984.$

Therefore,

$q_4(H) = (2985984+5971968H+4727808H^2+1492992H^3)/2985984.$

Hence, the amplification factor is

$$\left|\frac{q_4(H)}{q(H)}\right| = \left|\frac{2985984+4575168H+4727808H^2+1492992H^3}{2985984-4575168H+4727808H^2-1492992H^3}\right| < 1$$

for all $H < 0$. Then the method is A-stable when $k=4$.

6.5 Parallel Algorithm

The following algorithm is based on the above method. The description of parallel algorithm which is given here is similar to algorithm(2) which is given in Chapter 5.

The algorithm is as follows:

1. Compute y'_n, then compute the predicted values y_{n+1}, y_{n+k-1} and y_{n+k} using the formula $y_{n+r} = y_n + rhy'_n$, where r = 1, k - 1 and k.
2. Evaluate the function y'_{n+1}, y'_{n+k-1} and y'_{n+k}.
3. Computation of the corrected values is carried out by the following task:

```
Task type corrected-formula:
Send-initial (i,q) to task i;
    Declare
    ynew:vector (1..q);
    col-0,col-1,col-(k-1),col k; column (1..q);
```

```
        Begin
        Serial-loop
        Send-data (col-0,col-1,col-(k-1),col k,zz,z(1),z(k-1),z(k);
        Call procedure to evaluate the corrected values;
        Collect the corrected values (ytempi);
        End loop;
    End Corrector-formula;
```

The main procedure is as follows:

```
        Declare
        a,b,c,d:column(1..k);
    Begin
        For i in 1..k/q loop
        Cr(i).send-initial (i,q) to corrector-formula task i;
        End loop;
        n:=0;
        loop
        Exit when n > number of steps;
        zz:=dy(xstart,initial-value);
    --Evaluate the predicted values
        y(1):=initial-value + h * zz;
        y(k-1):=initial-value + h * float (k-1) * zz;
        y(k):=initial-value + h * float (k) * zz;
```

Since the predictor formula of order one and the corrector formula of order four, then it is important to use the mode $p(EC)^3$.

```
For s in 0..2 loop -- correction loop
    z(1):=   dy(x(1),y(1));
    z(k-1):= dy(x(k-1),y(k-1));
    z(k):=   dy(x(k),y(k));
--Call the corrector-formula
For i in 1..k/q loop
    Cr(i).send-initial (a(i-1) * q+1..i*q), b((i-1) * q+1..i*q),
    c((i-1) * q+1..i*q), d((i-1) * q+1..i*q), zz, z(1), z(k-1), z(k));
End loop;
For i in 1..k/q loop
    Cr(i).collect(y((i-1) * q+1...i*q));
End loop;
End loop; --end correction loop;
    Initial-value:= y(k);
    xstart:= x(k);
    n:= n+k;
    Exit when n = number of steps;
End loop;
End procedure.
```

6.6 Theoretical Cost

In this section we shall present the parallel cost of the above algorithm in terms of the four parameters which were mentioned in previous

chapters (4,5).

For each block we need three predicted values and four function which can be obtained in times 8m and 10f respectively. Since there is only one task type to compute the corrected values, then the time required is

$$kr/q + [6kr/q + (15q + 12)\ ks/q + 27km/p].$$

This shows that, there is one rendezvous to send i,q to k/q task and two rendezvous to send and collect 5q + 4 data items. The total number of arithmatics operations is 9q. If we assume that the number of processors $p \leq k/q$, then the number of arithmatics operation is 9k/p. Thus the time required for each block is

$$kr/q + [\ 2kr/q + (5q + 4)\ ks/q + 9km/p\ ]\ x\ 3$$

where 3 is the number of iterations.

Hence, the total cost of this method is

$$kr/q + [6kr\ /q + (15q + 12)ks/q + 8m + 27km/p + 10f]\ c/kh \qquad (6.6.1)$$

where c is the end of the interval and h the stepsize. But if we assume the number of tasks is equal to the number of processors, then the formula (6.6.1) becomes

$$kr/q + [6kr/q + (15q + 12)ks/q + (8 + 27q)m + 10f]\ c/kh \qquad (6.6.2)$$

6.7 Comparison with Other Methods

In this section we compare between our 4th order block method derived in this chapter and four order Adams predictor-corrector method [33] in two ways:

a) The behaviour of global error

b) Time taken to run each method

The Adams predictor-corrector method is given by:

$$p:\ y^{p}_{n+1} = y_n + \frac{h}{24}\left[\, 55y'_n - 59y'_{n-1} + 37y'_{n-2} - 9y'_{n-3} \,\right]$$

$$c:\ y^{c}_{n+1} = y_n + \frac{h}{24}\left[\, 9y'_{n+1} + 19y'_n - 5y'_{n-1} + y'_{n-2} \,\right]$$

The truncation errors for these formulas are:

$251h^5y^{(5)}(\zeta)/720$ and $-19h^5y^{(5)}(\zeta)/720$ respectively.

The above formulas have the same order, then it is necessary to make one application of the corrector for each step, provided h is small.

To implement the above formulas we shall use fourth order Runge-Kutta method which is defined by

$$k_1 = h\, f(x_n, y_n),$$

$$k_2 = h\, f(x_n + h/2\ ,\ y_n + k_1/2)\ ,$$

$$k_3 = h\, f(x_n + h/2\ ,\ y_n + k_2/2)\ ,$$

$$k_4 = h\, f(x_n + h\ ,\ y_n + k_3)\ ,$$

$$y_{n+1} = y_n + (k_1 + 2k_2 + 2k_3 + k_4)/6\ ,$$

to obtain the starting values y_1, y_2 and y_3(see appendix c). So, for each step, compute the predicted value y^{p}_{n+1}, evaluate y'^{p}_{n+1} and compute the corrected value y^{c}_{n+1}. The serial algorithm of the above methods can be written as

- - Compute the starting values using R-k methods

For i in 1...3 loop

$k_1 := h\ f(x_{i-1}, y_{i-1})$;

$k_2 := h\ f(x_{i-1} + h/2,\ y_{i-1} + k_1/2)$;

$k_3 := h\ f(x_{i-1} + h/2,\ y_{i-1} + k_2/2)$;

$k_4 := h\ f(x_{i-1} + h,\ y_{i-1}\ k_3)$;

$y_i := y_{i-1} + (k_1 + 2.0 * k_2 + 2.0 * k_3 + k_4)/6.0$;

End loop

- - compute the function evaluations

$Z_0 := f(x_0, y_0)$;

$Z_1 := f(x_1, y_1)$;

$Z_2 := f(x_2, y_2)$;

$Z_3 := f(x_3, y_3)$;

For i in 4 .. n loop

- - compute the predicted values

$y_p := y_{i-1} + h * (55.0 * Z_{i-1} - 59.0 * Z_{i-2} + 37.0 * Z_{i-3} - 9.0 * Z_{i-4})/24.0;$

- - Evaluate the function

$Z_i := f(x_i, y_p)$;

- - Compute the corrected values

$y_i := y_{i-1} + h * (9.0 * Z_i + 19.0 * Z_{i-1} - 5.0 * Z_{i-2} + Z_{i-3})/24.0$;

End loop;

6.7.1 Predicted Cost of Adams Methods

The Adams method used here requires four starting values (including initial value), so using 4th order Runge-Kutta method requires four function

evaluations per solution value. Before we start the predictor-corrector procedure, four function evaluations have to be used to compute the predicted value. Thus with PEC mode, one function evaluation needs to be calculated after the predicted value. Hence the number of function evaluations is

$$\sum_{i=1}^{3} 4 + 4 + \sum_{i=4}^{n} 1 = n + 13$$

where n is the number of steps.

The number of additions and multiplications is approximately 19 n.

Therefore, the total time is

$$T = (n+13)\, f + 19n\, m$$

where f is the time to do one function evaluation and m is the time to do one addition or multiplication.

6.8 Numerical Results

The numerical results presented here are those produced by the Adams predictor-corrector method and our fourth order block method when applied to test problems (i), (ii), (iii) and (iv).

Table 6.8.1 - Test Problem (i)

Number of Steps	Stepsize	Global Error	
		Adams p-c	Block Method
100	0.01	5.87×10^{-10}	5.61×10^{-8}
200	0.005	4.07×10^{-11}	3.56×10^{-9}
300	0.003333	8.30×10^{-12}	7.08×10^{-10}
400	0.0025	2.67×10^{-12}	2.25×10^{-10}
500	0.002	1.11×10^{-1}	9.22×10^{-11}
600	0.001667	5.38×10^{-13}	4.45×10^{-11}
700	0.0014286	2.92×10^{-13}	2.40×10^{-11}
800	0.00125	1.72×10^{-13}	1.41×10^{-11}
900	0.001111	1.00×10^{-13}	8.81×10^{-12}
1000	0.001	6.62×10^{-14}	5.78×10^{-12}

Table 6.8.2 - Test Problem (ii)

Number of Steps	Stepsize	Global Error	
		Adams p-c	Block Method
100	0.2	1.71×10^{-4}	5.05×10^{-5}
200	0.1	1.22×10^{-5}	2.72×10^{-6}
300	0.066667	2.44×10^{-6}	4.99×10^{-7}
400	0.05	7.64×10^{-7}	1.51×10^{-7}
500	0.04	3.08×10^{-7}	6.03×10^{-8}
600	0.03333	1.46×10^{-7}	2.85×10^{-8}
700	0.02857	7.80×10^{-8}	1.52×10^{-8}
800	0.025	4.54×10^{-8}	8.80×10^{-9}
900	0.02222	2.80×10^{-8}	5.45×10^{-9}
1000	0.02	1.81×10^{-8}	3.55×10^{-9}

Table 6.8.3 - Test Problem (iii)

Number of Steps	Stepsize	Global Error	
		Adams p-c	Block Method
100	0.2	4.73×10^{-7}	2.14×10^{-5}
200	0.1	4.18×10^{-8}	1.38×10^{-6}
300	0.06667	9.14×10^{-9}	2.74×10^{-7}
400	0.05	3.03×10^{-9}	8.72×10^{-8}
500	0.04	1.28×10^{-9}	3.58×10^{-8}
600	0.03333	6.28×10^{-10}	1.73×10^{-8}
700	0.02857	3.43×10^{-10}	9.34×10^{-9}
800	0.025	2.03×10^{-10}	5.48×10^{-9}
900	0.0222	1.28×10^{-10}	3.42×10^{-9}
1000	0.02	8.44×10^{-11}	2.25×10^{-9}

Table 6.8.4 - Test Problem (iv)

Number of Steps	Stepsize	Global Error	
		Adams p-c	Block Method
100	0.01	6.35×10^{-10}	1.55×10^{-8}
200	0.005	3.79×10^{-11}	9.57×10^{-10}
300	0.00333	7.37×10^{-12}	1.89×10^{-10}
400	0.0025	2.31×10^{-12}	5.96×10^{-11}
500	0.002	9.41×10^{-13}	2.44×10^{-11}
600	0.001667	4.53×10^{-13}	1.18×10^{-11}
700	0.0014286	2.44×10^{-13}	6.33×10^{-12}
800	0.00125	1.46×10^{-13}	3.72×10^{-12}
900	0.001111	8.84×10^{-14}	2.32×10^{-12}
1000	0.001	5.51×10^{-14}	1.52×10^{-12}

6.9 Timing Results

The experimetnal timings in this section are those produced by the Adams method and 4th order block method when applied to the test problem (i), (ii), (iii) and (iv).

First, we shall give the actual and predicted times which were obtained using the Adams method, when the program was run on a single processor.

Table 6.9.1

Test Problem	Number of Steps	t(A)	t(P)
(i)	500 1000	2.40 3.80	1.61 3.45
(ii)	500 1000	3.0 4.0	1.80 3.62
(iii)	500 1000	2.80 4.10	1.81 3.64
(iv)	500 1000	3.50 4.10	2.04 4.10

The following table illustrates the timings when 4th order block method is used.

Table 6.9.2

Test Problem	Number of Steps	Number of Processors 1	2	3	t(A) or t(P)
(i)	500	8.6	8.3		t(A)
		6.7	5.2		t(P)
	1000	15.5	13.6	14.3	t(A)
		13.7	10.3	9.2	t(P)
(ii)	500	8.9	8.3		t(A)
		7.3	5.6		t(P)
	1000	16.0	13.7	13.8	t(A)
		14.6	11.3	10.1	t(P)
(iii)	500	8.8	8.3		t(A)
		7.3	5.7		t(P)
	1000	16.0	14.6	14.3	t(A)
		14.7	11.3	10.2	t(P)
(iv)	500	10.7	9.7		t(A)
		7.9	6.2		t(P)
	1000	16.3	15.2	15.4	t(A)
		15.7	12.4	11.2	t(P)

6.10 Discussion and Conclusion

Tables 6.8.1 to 6.8.4 show the global errors of the Adams method and the block method when block length k=4 and sub-block q=2.

We observe from these tables that once the stepsize h has changed to h/2, the value of the ratio (between the current error and previous error) is near 16, which is the theoretical ratio that would be expected since these methods are fourth order. Comparing the global errors in table 6.8.1, we can

see that the Adams method has smaller errors than block method, for example, the Adams method has error 0.01 times that of the block method. In table 6.8.2, block method has global error 0.01 times that of the Adams method. In table 6.8.3, the global error of problem (iii), using the Adams method, is 0.01 times the global error of the block method. Finally, exactly the same remarks apply to table 6.8.4.

Tables 6.9.1 and 6.9.2 show the time taken to run the Adams method and the block method respectively. The Adams method seems to be much cheaper than our block method, and one of the reasons is in the Adams method mode PEC was used, while in the block method $P(EC)^3$ was used, but the numerical results are satisfactory since no real difference in accuracy between the two methods is shown.

CHAPTER 7

Suggestions for Future Work

This chapter has two aims. The first is to present some modified methods used in solving linear system (2.1.1) and it would be interesting to carry out more theoretical and computation study in order to find the best method suitable for parallel computers. The second is to present new methods for solving the initial value problems.

7.1 Future Work on Linear Systems

In this section, we present a description of some modified methods used in solving linear system (2.1.1). Block Gauss elimination with partial pivoting. It is slightly different from algorithm 2 which was mentioned in chapter 4. Also it is useful to compare the cost of this algorithm with algorithm 2. Concerning direct factorization with partial pivoting, the matrix A is to be factored using LU decomposition. It is interesting to compare the cost here with the cost of algorithm (1) which was mentioned in chapter 4. Block Gauss-Jordan and Choleski's methods are given.

7.1.1 Block Gaussian Elimination

In this section we shall give an algorithm to solve a linear system $A \underline{x} = \underline{b}$ using block Gauss elimination with partial pivoting, consider a matrix A of order n x n and the vectors $\underline{x}$ and $\underline{b}$ are of order n x 1.

If we partition the matrix A into q^2 blocks of order m x m each and partition $\underline{x}$ and $\underline{b}$ into q x 1, where q = n/m then the forward elimination algorithm can be written as

```
For i in 1..q loop
  q row (i):= i
-- denotes the row numbers
End loop;
For i in 1..q-1 loop
  let α be the smallest integer with
  i ≤ α ≤ q
| A (q row(α),i) | := max   | A (q row(j),i) |
                    i ≤ j ≤ q
if   A(q row(α),i) := 0
     then  no unique solution exists.
if   q row(i) ≠ q row(α)
     then interchange rows.
For j in i + 1 .. q loop
     m_j := A_ji  A_ii^-1  ;
For k in i + 1 .. q loop
     A_jk := A_jk  -  m_j A_ik  ;
     b_k := b_k - m_j b_i
End loop;
End loop;
End loop;
```

The individual blocks are processed serially, so we need the following procedures:

1. Procedure to compute A^{-1};
2. Procedure to multiply two blocks;
3. Procedure to subtract two blocks.

As we know Cramer's method is used to compute the inverse of a non singular matrix, involving the computation of q+1 determinants, so the computation of these determinants is fairly expensive. A much more efficient method for finding A^{-1} is as follows:

let e_i denote the ith column of identity matrix I; that is:

$$e_i = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad i^{th} \text{ component}$$

let c_i denote the ith column of the matrix A^{-1}. Then

$$A\ A^{-1} = I$$

$$\Rightarrow \quad A\ c_i = e_i, \quad i = 1,2,\ldots, m$$

Thus the solution to the equation

$$A\ Z = e_i \qquad (7.1.1)$$

is just the ith column of A^{-1}. Hence, A^{-1} can be computed by solving (7.1.1) for i=1,2,..,m. This can be done using Gaussian elimination.

To solve the system of equations, we shall use backward substitution which is given as follows:

$$\underline{x}_q = A^{-1}_{q,q} \; \underline{b}_q$$

$$\underline{x}_{q-1} = A^{-1}_{q-1,q-1} \; (\underline{b}_{q-1} - A_{q-1,q} \; \underline{x}_q)$$

$$\underline{x}_{q-2} = A^{-1}_{q-2,q-2} \; (\underline{b}_{q-2} - A_{q-2,q} \; \underline{x}_q - A_{q-2,q-1} \; \underline{x}_{q-1})$$

$$\vdots$$

$$\underline{x}_1 = A^{-1}_{11} \; (\underline{b}_1 - A_{1,q}\underline{x}_q - A_{1,q-1} \; \underline{x}_{q-1} - A_{1,q-1}\underline{x}_{q-1} \cdots A_{1,2}\underline{x}_2).$$

We assume that A^{-1}_{ii}, i = q,.., 1 are known then we can compute the following:

$$B^{(1)}_i = A^{-1}_{ii} * (\underline{b}_i - A_{i,q} \; \underline{x}_q) \quad , \quad i = q\text{-}1,.., \; 1$$

where

$$\underline{B}^{(1)}_{q-1} = A^{-1}_{q-1,q-1} \; (\underline{b}_{q-1} - A_{q-1,q} \; \underline{x}_q) \quad \Leftrightarrow \quad \underline{x}_{q-1}$$

Then

$$\underline{B}^{(2)}_i = \underline{B}^{(1)}_i - A^{-1}_{ii} * A_{i,q-1} \; \underline{x}_{q-1}$$

$$i = q\text{-}2,.., \; 1$$

where

$$\underline{B}^{(2)}_{q-2} = \underline{B}^{(1)}_{q-2} - A^{-1}_{q-2,q-2} * A_{q-2,q-1} \; \underline{x}_q \quad \Leftrightarrow \quad \underline{x}_{q-2}$$

and so on. Then the final step is

$$\underline{B}^{(q-1)}_1 = \underline{B}^{(q-2)}_1 - A^{-1}_{11} * A_{12} \; \underline{x}_2 \quad \Leftrightarrow \quad \underline{x}_1 \; .$$

Note that the pivoting that was done during the elimination should, however, also be included in the backward substitution.

7.1.2 Direct Factorization with Partial Pivoting

We are interesting to solve the linear system $A\underline{x}=\underline{b}$ using LU-decomposition with partial pivoting. The system of linear equations can be solved in three stages.

a) decompose A into the product of a lower triangular matrix L and upper triangular matrix U;

b) solve $L\ \underline{Z} = \underline{b}$ and,

c) solve $U\ \underline{x} = \underline{Z}$.

The following task is used to factor the coefficients of the matrix A into L and U, where the entries $a_{ij} = l_{ij}$ for each i = 2,3,.., n and j = 1,2,..,i-1; $l_{ii} = 1$, i = i,i+1,.., n.

The task required to triangulate the matrix A is

Accept pivot row (i), elimination row (j), i;

-- to find the first pivot

let q be the smallest integer such that $1 \leq q \leq n$

and $|a_{q1}| = \max_{1 \leq j \leq n} |a_{j1}|$;

if $|a_{q1}| = 0$, then is no unique solution exists.

if $q \neq 1$ then interchange rows q and 1 in A.

Compute the multipliers

For j > i , k > i do

$a_{jk} <\!\!- a_{jk} - a_{ji} * a_{ik}$;

Return row j ;

To complete the solution of linear system, the following tasks are required:

The forward substitution task is given as

Accept the vector b;

$Z_i = b_i$, i=1,..,n

Accept a_{ij} , Z_j ,

$Z_i = Z_i - a_{ij} * Z_j$

Return Z_i;

The task is started by copying $\underline{b}$ into $\underline{Z}$ for i= 1,.., n. At this point Z_1 is computed. To compute Z_i, i=2,..,n, Z_1 and $b_i(=Z_i)$ are needed. So both Z_1 and Z_2 are computed. The next step Z_1, Z_2 and Z_i, i=3,..,n are needed. Then the calculation of Z_n completes the calculation of the vector Z.

The backward substitution task is:

Accept a_{ii} , Z_i ;

Compute $x_i = Z_i/a_{ii}$, i=1,..,n

Return x_i;

Accept a_{ij} , x_j ;

Compute $x_i = x_i - (a_{ij}/a_{ii})x_j$

Return x_i .

The task is started by evaluating the value of x_n, then communicate x_n into each remaining tasks, so as to modify the right hand side. The above implementation is carried out in a manner similar to the forward substitution task.

In order to compare this algorithm with algorithm 1, which was mentioned in chapter 4, we expect that the cost will be slightly more than the

previous algorithm, this is due to the time consumed to interchange rows and the forward substitution.

7.1.3 Block Gauss-Jordan Method

Solving a linear system $A\ \underline{x} = \underline{b}$ using Gauss-Jordan method converts $[A \mid \underline{b}]$ to $[I \mid b^{(n)}]$, so that at the completion of the elimination $\underline{x} = \underline{b}^{(n)}$. To solve $A\ \underline{x} = \underline{b}$ by this method on sequential machine requires $n^3/2$ which is more expensive than regular Gauss elimination. If we consider a matrix A of order n x n and $\underline{x}$ and $\underline{b}$ are of order n x 1, and partition the matrix A into q^2 and the vectors $\underline{x}$ and $\underline{b}$ into q x 1, then the system will be

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & \cdots & A_{1n} \\ A_{21} & A_{22} & A_{23} & \cdots & A_{2n} \\ A_{31} & A_{32} & A_{33} & \cdots & A_{3n} \\ \vdots & \vdots & \vdots & & \vdots \\ A_{n1} & A_{n2} & A_{n3} & & A_{nn} \end{bmatrix} \begin{bmatrix} \underline{x}_1 \\ \underline{x}_2 \\ \underline{x}_3 \\ \vdots \\ \underline{x}_n \end{bmatrix} = \begin{bmatrix} \underline{b}_1 \\ \underline{b}_2 \\ \underline{b}_3 \\ \vdots \\ \underline{b}_n \end{bmatrix}$$

In Gauss-Jordan method every block is eliminated from each row of the matrix except the diagonal block, which is made equal to unity. However, we can write a simple algorithm as follows:

1) Multiply row (1) by A_{11}^{-1}
2) Multiply row (1) by A_{k1} , $k = 2,3,..,q$
   row (k) <— row (k) - row (1) * $A_{k1}A_{11}^{-1}$
3) Multiply row (2) by A_{22}^{-1}, multiply row (2) by A_{k2} and compute,
   row (1) <— row (1) - row (2) * $A_{12}^{(1)}\ A_{22}^{-1}$ and
   row (k) <— row (k) - row (2) * $A_{k2}^{(1)}\ A_{22}^{-1}$, $k=3,4,..,q$.

This places a zero block in the second column position for every row except row (2), which contains unit block. Also we do the same thing for the right hand side. Repeat step (3) for rows j = 3,4,.., q, noting that k is the range over all rows except row j.

Now, suppose q = 4, we have

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & b_1 \\ A_{21} & A_{22} & A_{23} & A_{24} & b_2 \\ A_{31} & A_{32} & A_{33} & A_{34} & b_3 \\ A_{41} & A_{42} & A_{43} & A_{44} & b_4 \end{bmatrix}$$

Applying the first two steps from the above algorithm, we get

$$A_{22} \leftarrow A_{22} - A_{12}A_{21}A_{11}^{-1} \quad A_{32} \leftarrow A_{32} - A_{12}A_{31}A_{11}^{-1} \quad A_{42} \leftarrow A_{42} - A_{12}A_{41}A_{11}^{-1}$$

$$A_{23} \leftarrow A_{23} - A_{13}A_{21}A_{11}^{-1} \quad A_{33} \leftarrow A_{33} - A_{13}A_{31}A_{11}^{-1} \quad A_{43} \leftarrow A_{43} - A_{13}A_{41}A_{11}^{-1}$$

;

;

$$A_{24} \leftarrow A_{24} - A_{14}A_{21}A_{11}^{-1} \quad A_{34} \leftarrow A_{34} - A_{14}A_{31}A_{11}^{-1} \quad A_{44} \leftarrow A_{44} - A_{14}A_{41}A_{11}^{-1}$$

$$b_2 \leftarrow b_2 - b_1A_{21}A_{11}^{-1} \quad b_3 \leftarrow b_3 - b_1A_{31}A_{11}^{-1} \quad b_4 \leftarrow b_4 - b_1A_{41}A_{11}^{-1}$$

The system becomes

$$\begin{bmatrix} I & A_{12} & A_{13} & A_{14} & b_1 \\ 0 & A_{22} & A_{23} & A_{24} & b_2 \\ 0 & A_{32} & A_{33} & A_{34} & b_3 \\ 0 & A_{42} & A_{43} & A_{44} & b_4 \end{bmatrix}$$

Applying the 3rd step,

$A_{12} <— A_{12} - A_{22}A_{12}A_{22}^{-1}$ $A_{32} <— A_{32} - A_{22}A_{32}A_{22}^{-1}$ $A_{42} <— A_{42} - A_{22}A_{42}A_{22}^{-1}$

$A_{13} <— A_{13} - A_{23}A_{12}A_{22}^{-1}$ $A_{33} <— A_{33} - A_{23}A_{32}A_{22}^{-1}$ $A_{43} <— A_{43} - A_{23}A_{42}A_{22}^{-1}$

;

;

$A_{14} <— A_{14} - A_{24}A_{12}A_{22}^{-1}$ $A_{34} <— A_{34} - A_{24}A_{32}A_{22}^{-1}$ $A_{44} <— A_{44} - A_{24}A_{42}A_{22}^{-1}$

$b_1 <— b_1 - b_2A_{12}A_{22}^{-1}$ $b_3 <— b_3 - b_2A_{32}A_{22}^{-1}$ $b_4 <— b_4 - b_2A_{42}A_{22}^{-1}$

This will give

$$\begin{bmatrix} I & 0 & A_{13} & A_{14} & b_1 \\ 0 & I & A_{23} & A_{24} & b_2 \\ 0 & 0 & A_{33} & A_{34} & b_3 \\ 0 & 0 & A_{43} & A_{44} & b_4 \end{bmatrix}$$

Repeat step 3 for row 3,

$A_{13} <— A_{13} - A_{33}A_{13}A_{33}^{-1}$ $A_{23} <— A_{23} - A_{33}A_{23}A_{33}^{-1}$ $A_{43} <— A_{43} - A_{33}A_{43}A_{33}^{-1}$

;

;

$A_{14} <— A_{14} - A_{34}A_{13}A_{33}^{-1}$ $A_{24} <— A_{24} - A_{34}A_{23}A_{33}^{-1}$ $A_{44} <— A_{44} - A_{34}A_{43}A_{33}^{-1}$

$b_1 <— b_1 - b_3A_{13}A_{33}^{-1}$ $b_2 <— b_2 - b_3A_{23}A_{33}^{-1}$ $b_4 <— b_4 - b_3A_{43}A_{33}^{-1}$

This gives

$$\begin{bmatrix} I & 0 & 0 & A_{14} & b_1 \\ 0 & I & 0 & A_{24} & b_2 \\ 0 & 0 & I & A_{34} & b_3 \\ 0 & 0 & 0 & A_{44} & b_4 \end{bmatrix}$$

The final step is

$$A_{14} \leftarrow A_{14} - A_{44}A_{14}A_{44}^{-1} \rightarrow 0$$

$$A_{24} \leftarrow A_{24} - A_{44}A_{24}A_{44}^{-1} \rightarrow 0 \quad ; \quad A_{34} \leftarrow A_{34} - A_{44}A_{34}A_{44}^{-1} \rightarrow 0 \quad .$$

The final form of the matrix is

$$\begin{bmatrix} I & 0 & 0 & 0 & b_1 \\ 0 & I & 0 & 0 & b_2 \\ 0 & 0 & I & 0 & b_3 \\ 0 & 0 & 0 & I & b_4 \end{bmatrix}$$

which yields to

$$\underline{x}_1 = \underline{b}_1 \quad , \quad \underline{x}_2 = \underline{b}_2 \quad , \quad \underline{x}_3 = \underline{b}_3 \quad , \quad \underline{x}_4 = \underline{b}_4 \quad .$$

We can say that in case q=4, we should have 3 processors available. The first step (finding the miltipliers)

1. Compute the multipliers

$$m_i = A_{i1}A_{11}^{-1} \quad , \quad i=2,3,4 \quad \text{in parallel}$$

Compute the following blocks

$$\begin{matrix} A_{22} & A_{32} & A_{42} \\ A_{23} & A_{33} & A_{43} \\ A_{24} & A_{34} & A_{44} \\ b_2 & b_3 & b_4 \end{matrix}$$

in parallel. In this case we can create a task defined as

block(1) <— block(1) - block(2) * Multiplier

2. Compute the multipliers

$$m_1 = A_{12}A_{22}^{-1} \quad , \quad m_3 = A_{32}A_{22}^{-1} \quad , \quad m_4 = A_{42}A_{22}^{-1}$$

Update the blocks

$$\begin{matrix} A_{12} & A_{32} & A_{42} \\ A_{13} & A_{33} & A_{43} \\ A_{14} & A_{34} & A_{44} \\ b_1 & b_3 & b_4 \end{matrix}$$

3. Compute the multipliers

$$m_1 = A_{13}A_{33}^{-1} \quad , \quad m_2 = A_{23}A_{33}^{-1} \quad , \quad m_4 = A_{43}A_{33}^{-1}$$

Update the blocks

$$\begin{matrix} A_{13} & A_{23} & A_{43} \\ A_{14} & A_{24} & A_{44} \\ b_1 & b_2 & b_4 \end{matrix}$$

4. Compute: $m_1 = A_{13}A_{33}^{-1}$, $m_2 = A_{23}A_{33}^{-1}$, $m_3 = A_{43}A_{33}^{-1}$

Update

$$A_{13} \quad A_{23} \quad A_{43}$$
$$A_{14} \quad A_{24} \quad A_{44}$$
$$b_1 \quad b_2 \quad b_4 \quad .$$

5. Then the solution is obtained by setting

$$\underline{x}_i = \underline{b}_i \ , \quad i = 1,2,3,4 \quad .$$

The above procedure can be implemented on parallel machine using three processors.

7.1.4 Choleski's Method

It is interesting to solve the linear system $A\ \underline{x} = \underline{b}$ using Choleski's method, because the cost is about $n^3/6$ using sequential computer. Let A be a symmetric and positive definite matrix of order n, then the algorithm might be written as follows:

1) Find $l_{11} = \sqrt{a_{11}}$

2) Find $l_{ji} = a_{ji}/l_{11}$, j=2,..,n in parallel

3) For i in 2 .. n-1 loop

Compute $l_{ii} = \left[a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2 \right]^{1/2}$

Compute $l_{ji} = \left[a_{ji} - \sum_{k=1}^{i-1} l_{jk}\ l_{ik} \right] / l_{ii}$

, $j = i + 1,.., n$ in parallel.

4) Compute $l_{nn} = \left[a_{nn} - \sum_{k=1}^{n-1} l_{nk}^2 \right]^{1/2}$.

5) Solve the system

$$L\ \underline{Z} = \underline{b} \quad \text{and} \quad L^T \underline{x} = \underline{b} \quad .$$

An important work has to be carried out theoretically and computationally in future and this is a continuation of chapter 4.

7.2 Future Work on Initial Value Problem

The primary aim of this work was to produce second and fourth order blocks methods for initial value ordinary differential equations which are applicable for arbitrary blocklength. In addition,the knowledge of the stability properties allows us to choose the efficient methods which are A-stable. Moreover, we prefer the methods where the local truncation errors have a lower power of k (blocklength).

Clearly, much additional work on block methods can be carried out. In particular, there are some methods we have thought about, but for which there has not been sufficient time for a complete analysis.

7.2.1 Block Methods

The first method suggested is given by

$$y_{n+r,s+1} = y_n + h \sum_{j=0}^{r} a_{rj}\, y'_{n+j,s}$$

$$, \; r = 1,2,.., \; k \; ; \quad s=0,1$$

where the coefficients can be put in tableau as follows:

Table 7.2.1

r	$a_{r,o}$	$a_{r,1}$	$a_{r,2}$	$a_{r,3}$	$a_{r,4}$	$a_{r,5}$	....	$a_{r,k-1}$	$a_{r,k}$
1	1/2	1/2							
2	2/5	6/5	2/5						
3	2/5	11/10	11/10	2/5					
4	2/5	11/10	1	11/10	2/5				
5	2/5	11/10	1	1	11/10	2/5			
⋮									
k	2/5	11/10	1	1	1	1	 1	11/10	2/5

where the coefficients satisfy the following:

a) when r = 1:

$$1/2 + 1/2 = 1,\ 1/2 = 1/2$$

b) when r = 2:

$$2/5 + 6/5 + 2/5 + 10/5 = 2\ ,\ 6/5 + 4/5 = 10/5 = 2$$

c) when r = 3:

$$2/5 + 11/10 + 11/10 + 2/5 = \frac{30}{10} = 3\ ,$$

$$11/10 + 22/10 + 12/10 = 45/10 = 9/2$$

d) when r = k:

$$2/5 + 11/10 + \sum_{j=2}^{k-2} 1 + 11/10 + 2/5 = (4+11+10k-30+11+4)/10 = k\ ,$$

$$11/10 + \sum_{j=2}^{k-2} j + 11(k-1)/10 + 2k/5 = (11+5k^2-15k+11k-11+4k)/10 = k^2/2.$$

The kth component of the local truncation error is

$$\varepsilon_k = \left[(kh)^3/6 - h^3 11+10(2^2+3^2+\ldots+(k-2)^2 + 11(k-1)^2 + 4k^2)/20\right] y'''_n$$

$$= \left[\frac{k^3}{6} - (11+10(2k^3-9k^2+13k-12)/6 + 11(k^2-2k+1) + 4k^2)/20\right] h^3 y'''_n$$

Therefore

$$\varepsilon_k = (k/60 - 1/10)h^3 y'''_n = O(kh^3).$$

To check that, suppose k=3, then the 3rd component of local truncation error is given by

$$\varepsilon_3 = \left[27/6 - (11+44+36)/20\right] h^3 y'''_n$$

$$= (27/6 - 91/20)h^3 y'''(x_n) = -h^3 y'''_n/20.$$

Also when k=4, the 4th component of the local truncation error is given by

$$\varepsilon_4 = \left[64/6 - (11+40+99+64)/20\right] h^3 y'''_n$$

$$= -h^3 y'''_n/30.$$

The second method suggested is given by

$$y_{n+r,s+1} = y_n + h \sum_{j=0}^{r} a_{rj}\, y'_{n+j,s}$$

$$r=1,2,\ldots,k \quad ; \quad s=0,1$$

where the coefficients represented in the tableau below:

Table 7.2.2

r	$a_{r,0}$	$a_{r,1}$	$a_{r,2}$	$a_{r,3}$	$a_{r,4}$	$a_{r,5}$	...	$a_{r,k-1}$	$a_{r,k}$
1	1/2	1/2							
2	2/5	6/5	2/5						
3	2/5	11/10	11/10	2/5					
4	2/5	16/15	16/15	16/15	2/5				
5	2/5	21/20	21/20	21/20	21/20	2/5			
⋮									
k	2/5	$\frac{(5k-4)}{5(k-1)}$	$\frac{(5k-4)}{5(k-1)}$	$\frac{(5k-4)}{5(k-1)}$	$\frac{(5k-4)}{5(k-1)}$	.....		$\frac{(5k-4)}{5(k-1)}$	2/5

The coefficients satisfy the first and the second characteristic polynomials, which are given by:

$$\sum_{j=0}^{k} a_{rj} = r \quad \text{and} \quad \sum_{j=0}^{k} j a_{rj} = r^2/2 \quad , \quad r=1,\ldots,k.$$

The kth component of the local truncation error is given by:

$$\varepsilon_k = h^3\left\{k^3/6 - (5k-4)\left(\sum_{j=1}^{k-1} j^2\right)/10(k-1) - k^2/5\right\} y'''_n$$

$$= (k^2-4k)h^3 y'''_n/60 = O(k^2h^3).$$

The third method suggested is given by

$$y_{n+r,s+1} = y_n + \sum_{j=0}^{r} a_{rj}\, y'_{n+j,s}$$

$$r=1,2,..,k \quad ; \quad s=0,1$$

where the coefficients represented in the tableau below:

Table 7.2.3

r	$a_{r,0}$	$a_{r,1}$	$a_{r,2}$	$a_{r,3}$	$a_{r,4}$	.....	$a_{r,k}$
1	1/2	1/2					
2	2/3	2/3	2/3				
3	3/4	3/4	3/4	3/4			
4	4/5	4/5	4/5	4/5	4/5		
⋮	⋮						
k	k/k+1	k/k+1	.....				k/k+1

The coefficients satisfy the first and the second characteristic polynomials, which are given as

$$\sum_{j=0}^{k} a_{rj} = r \quad \text{and} \quad \sum_{j=0}^{k} a_{rj} = r^2/2 \quad , \quad r=1,2,\ldots,k$$

when r = k, the local truncation error is given by

$$\varepsilon_k = h^3\left\{k^3/6 - k(\sum_{j=1}^{k} j^2)/2(k+1)\right\} y'''_n$$

$$= h^3\left\{k^3/6 - k^2(k+1)(2k+1)/12(k+1)\right\} y'''_n$$

$$= (-k^2h^3/12)\, y'''_n = O(k^2h^3)$$

The fourth suggested method is given by

$$y_{n+r,s+1} = y_n + h \sum_{j=0}^{r} a_{rj}\, y'_{n+j,s}$$

$$, \; r=1,2,\ldots,k \; ; \; s=0,1$$

where the coefficients are shown in the table below:

Table 7.2.4

r	$a_{r,0}$	$a_{r,1}$	$a_{r,2}$	$a_{r,3}$	$a_{r,4}$	$a_{r,5}$	.....	$a_{r,k-1}$	$a_{r,k}$
1	1/2	1/2							
2	1/2	1	1/2						
3	1/2	5/6	4/3	1/3					
4	1/2	1	1	1	1/2				
5	1/2	5/6	4/3	2/3	4/3	1/3			
⋮									
k-1	1/2	5/6	4/3	2/3	4/3	2/3	.....	4/3	1/3
k	1/2	1	1	1	1	1	.....	1	1/2

where the coefficients satisfy the first and the second characteristic polynomials, which are given by

$$\sum_{j=0}^{k} a_{rj} = r \quad , \qquad \sum_{j=0}^{k} j^2 a_{rj} = r^2/2.$$

So, the kth component of the local truncation error when k is even given by

$$\varepsilon_k = (-kh^3)y_n'''/12 \quad ,$$

and when k is odd ,

$$\varepsilon_k = -h^3 y_n'''/12 \quad \text{which does not depend on k.}$$

The fifth method suggested is given by

$$y_{n+r} = y_n + h \sum_{j=0}^{r} a_{r,j}\, y'_{n+j,s}$$

$$r = 2,\ldots, k \quad ; \quad s=0,1.$$

In this case we can choose the coefficients when r=1 as follows

$a_{1,0} = 9/24$, $a_{1,1} = 19/24$, $a_{1,2} = -5/24$, $a_{1,3} = 1/24$.

So, the coefficients for this method are shown in the table below when k is even.

Table 7.2.5

r	$a_{r,0}$	$a_{r,1}$	$a_{r,2}$	$a_{r,3}$	$a_{r,4}$	$a_{r,5}$	$a_{r,6}$	.....	$a_{r,k}$
1	9/24	19/24	-5/24	1/24					
2	1/3	4/3	1/3						
3	3/8	9/8	9/8	3/8					
4	1/3	4/3	2/3	4/3	1/3				
5	1/3	4/3	17/24	9/8	9/8	3/8			
6	1/3	4/3	2/3	4/3	2/3	4/3	1/3		
⋮									
k	1/3	4/3	2/3	4/3	2/3	4/3	.....	2/3	4/3 1/3

The coefficients of the method satisfy the characteristic polynomials, which are given as:

$$\sum_{j=0}^{k} a_{rj} = r \ , \quad \sum_{j=0}^{k} j a_{rj} = r^2/2 \ , \quad \sum_{j=0}^{k} j^2 a_{rj} = r^3/3$$

and $\sum_{j=0}^{k} j^3 a_{rj} = r^4/4$, $r=1,\ldots,k.$

The local truncation error when r=k is given by

$$\varepsilon_k = \left[\frac{k^5 h^5}{120} - \frac{h^5}{72}\left\{4(1^4+3^4+5^4+7^4+..+(k-1)^4)+2(2^4+4^4+6^4+..+(k-2)^4\right\} - \frac{k^4 h^5}{72}\right] y_n^{(5)}$$

$$= \left[\frac{k^5h^5}{120} - \frac{h^5}{72}\left\{\frac{4k(k-1)(3k^3+3k^2-7k-7)}{30} + \frac{2k(k-1)(3k^3-12k^2+8k+8)}{30}\right\} - \frac{k^4h^5}{72}\right]y_n^{(5)}$$

$$= \left[\frac{k^5}{120} - \frac{(k)(k-1)}{2160}(12k^3+12k^2-28k-28) - \frac{k(k-1)}{2160}(6k^3-24k^2+16k+16) - \frac{30k^4}{2160}\right]h^5y_n^{(5)}$$

$$= \left\{18k^5-18k^5+30k^4-12k-30k^4\right\}h^5y_n^{(5)}/2160 \quad = \frac{-kh^5}{180}\,y_n^{(5)}$$

$$= O(kh^5)$$

To check that, suppose k=2, then the formula when r=2 becomes

$$y_{n+2} = y_n + \frac{h}{3}\left[y'_n + 4y'_{n+1}\, y'_{n+2}\right]$$

where

$$\varepsilon_2 = \left\{\frac{2^5}{120} - \frac{1}{72}\left[4+2^4\right]\right\}h^5y_n^{(5)} = \left\{\frac{32}{120} - \frac{20}{72}\right\}h^5y_n^{(5)}$$

$$= -h^5y_n^{(5)}/90.$$

Also, in case k=4, then the 4th component of the local truncation error is given by

$$\varepsilon_4 = \left\{\frac{4^5}{120} - \frac{1}{72}(4+2.2^4+4.3^4+4^4)\right\}h^5y_n^{(5)} = -h^5y_n^{(5)}/45.$$

When k is odd, the kth formula can be written as

$$y_{n+k} = y_n + h\left[y'/3+4y'_{n+1}/3+2y'_{n+2}/3 + \ldots + 4y'_{n+(k-4)}/3+17y'_{n+(k-3)}/24\right.$$
$$\left.ay'_{n+(k-2)}/8+9y'_{n+(k-1)}/8+3y'_{n+k}/8\right].$$

Therefore, the local truncation error is

$$\varepsilon_k = \left\{\frac{k^5h^5}{120} - \frac{h^5}{24}\left[\frac{4}{3} + \frac{2.2^4}{3} + \frac{4.3^4}{3} + \frac{2.4^4}{3} + \ldots + \frac{4(k-4)^4}{3} + \frac{17(k-3)^4}{24}\right.\right.$$
$$\left.\left. + \frac{9(k-2)^4}{8} + \frac{9(k-1)^4}{8} + \frac{3k^4}{8}\right]\right\}y_n^{(5)}$$

$$= \left\{ \frac{k^5}{120} - \frac{1}{72} \left[4(1^4+3^4+5^4+\ldots+(k-4)^4 \right] + 2(2^4+4^4+6^4+\ldots+(k-5)^4) \right]$$

$$- \frac{17(k-3)^4}{576} - \frac{9(k-2)^4}{192} - \frac{9(k-1)^4}{192} - \frac{3k^4}{192} \Big\} h^5 y_n^{(5)}$$

$$= \Big\{ 72k^5/8640 - (72k^5 - 1200k^4 + 7920k^3 - 25920k^2 + 42168k - 27360)/8640$$

$$- (255k^4 - 3060k^3 + 13770k^2 - 27540k + 20655)/8640$$

$$- (405k^4 - 3240k^3 + 9720k^2 - 12960k + 6480)/8640$$

$$- (405k^4 - 1620k^3 + 2430k^2 - 1620k + 405)/8640$$

$$- 135k^4/8640 \Big\} h^5 y_n^{(5)}.$$

Hence,

$$\varepsilon_k = -(180+48k)h^5 y_n^{(5)}/8640 = O(kh^5).$$

To check that, in case k=3, the formula for r=3 becomes

$$y_{n+3} = y_n + \frac{3h}{8} (y'_n + 3y'_{n+1} + 3y'_{n+2} + y'_{n+3})$$

which has

$$\varepsilon_3 = -3h^5 y_n^{(5)}/80.$$

The formula, in case when r=5 (k=5), is given by

$$y_{n+5} = y_n + h(y'_n/3 + 4y'_{n+1}/3 + 17y'_{n+2}/24 + 9y'_{n+3}/8 + 9y'_{n+4}/8 + 3y'_{n+5}/8,$$

and

$$\varepsilon_5 = = -\frac{7}{144} h^{(5)} y_n^{(5)} \quad .$$

This work on these methods is currently in progress, but still needs some more work to report about it; and we think it is important to investigate the stability analysis of such methods. Also, for fourth and higher order methods we should seek some methods to use as predicted formulae.

7.2.2 Runge-Kutta Methods

As far as this work is concerned with parallel solution of ordinary differential equations and because the Runge-Kutta methods have been used to produce high equality ordinary differential equations software, we should mention some ideas about parallel Runge-Kutta methods.

Consider the general form of an explicit Runge-Kutta method which consists of evaluating the quantities

$$k_1 = f(x_n, y_n)$$

$$k_r = f(x_n + hc_r,\ y_n + h \sum_{s=1}^{r-1} a_{rs} k_s)$$

$$, \ r = 2,\ldots,m$$

and

$$y_{n+1} = y_n + \sum_{r=1}^{m} b_r k_r$$

using a sequential machine, $k_1, k_2, \ldots, k_m$ are evaluated in sequence and in order.

It is noted in [13], that such methods can easily be used either across the method or across the system. In case of solving a system of equations, at each value of $x_n + hc_r$ there are nk values which depend upon n derivative equations. The calculation of the derivative functions can be carried out in parallel, and each set of k_r, $r = 1,2,\ldots,m$ is assigned to a different processor. The idea here is how to compute the derivative functions simultaneously. However, if we try to think for a new Runge-Kutta method,

where the value of k_r, r=2,..,m depend only on k_1 (say), this will be fine and it is suitable to use.

Now, we shall describe a new technique which might be helpful for parallel computation.

Let us start with fourth order Runge-Kutta method

$$k_1 = f(x_n, y_n)$$
$$k_2 = f(x_n + hc_2, y_n + ha_{21}k_1)$$
$$k_3 = f(x_n + hc_3, y_n + ha_{31}k_1 + ha_{32}k_2)$$
$$k_4 = f(x_n + hc_4, y_n + ha_{41}k_1 + ha_{42}k_2 + ha_{43}k_3)$$

and

$$y_{n+1} = y_n + h(a_{51}k_1 + a_{52}k_2 + a_{53}k_3 + a_{54}k_4).$$

Apply the above formulae to the test problem

$$y' = \lambda y \ , \quad \lambda < 0$$

then we have,

$$k_1 = \lambda y_n$$
$$k_2 = \lambda(1 + a_{21}H)y_n$$
$$k_3 = \lambda(1 + (a_{31} + a_{32})H + a_{32}a_{21}H^2)y_n$$
$$k_4 = \lambda(1 + (a_{41} + a_{42} + a_{43})H + (a_{42}a_{21} + a_{43}(a_{31} + a_{32}))H^2 + a_{43}a_{32}a_{21}H^3)y_n,$$

where $\lambda h = H$.

Let $v_{11} = 1, \ v_{21} = 1, \ v_{31} = 1, \ v_{41} = 1$

$$v_{22} = a_{21}v_{11}, \quad v_{32} = a_{31}v_{11} + a_{32}v_{21}$$
$$v_{33} = a_{32}v_{22}$$
$$v_{42} = a_{41}v_{11} + a_{42}v_{21} + a_{43}v_{31}$$

$$v_{43} = a_{42}v_{22} + a_{43}v_{32}$$
$$v_{44} = a_{43}v_{33}.$$

Now let $k_1 = \lambda v_{11} y_n$

$$k_j = \lambda(v_{j1} + v_{j2}H + \ldots + v_{jj}H^{j-1})y_n \quad , \quad j=2,3,4.$$

We derive the following recurrence relation for v_{ji},

$$v_{j1} = 1$$

$$v_{j2} = a_{j1}v_{11} + a_{j2}v_{21} + \ldots + a_{jj-1}v_{j-1,1}$$

$$v_{j3} = a_{j2}v_{22} + a_{j3}v_{32} + \ldots + a_{jj-1}v_{j-1,2}$$

Generally,

$$v_{ji} = \sum_{k=i-1}^{j-1} a_{jk}\, v_{k,j-1}$$

$j=2,3,4$, $i=2,3,j$.

Then

$$y_{n+1} = \mu(H)y_n$$

where

$$y_{n+1} = y_n + a_{51}hk_1 + a_{52}hk_2 + a_{53}hk_3 + a_{54}hk_4.$$

$$= y_n + a_{51}Hy_n + a_{52}H(1+a_{21}H)y_n + a_{53}H(1+(a_{31}+a_{32})H + a_{32}a_{21}H^2)y_n$$
$$+ a_{54}H(1+(a_{41}+a_{42}+a_{43})H + (a_{42}a_{21}+a_{43}(a_{31}+a_{32}))H^2 + a_{43}a_{32}a_{21}H^3)y_n$$

$$\Rightarrow y_{n+1} = y_n + (a_{51}+a_{52}a_{53}+a_{54})Hy_n + (a_{52}a_{21}+a_{53}(a_{31}+a_{32})$$
$$+ a_{54}(a_{41}+a_{42}+a_{43}))H^2y_n + (a_{53}a_{32}a_{21}+a_{54}a_{41}+a_{54}(a_{31}+a_{32}))H^3y_n$$
$$+ a_{54}a_{43}a_{32}a_{21}H^4y_n.$$

$$\Rightarrow y_{n+1} = y_n + \left[H \sum_{k=1}^{4} a_{5,k} v_{k,1} + H^2 \sum_{k=2}^{4} a_{5,k} v_{k,2} + H^3 \sum_{k=3}^{4} a_{5,k} v_{k,3} + H^4 a_{54} v_{44}\right]$$

$$\therefore \mu(H) = 1 + H \sum_{k=1}^{4} a_{5,k} v_{k,1} + H^2 \sum_{k=2}^{4} a_{5,k} v_{k,2} + H^3 \sum_{k=3}^{4} a_{5,k} v_{k,3} + H^4 a_{54} v_{44}$$

$$= 1 + H v_{5,1} + H^2 v_{5,2} + H^3 v_{5,3} + H^4 v_{5,4}$$

where

$$v_{5,i} = \sum_{k=1}^{4} a_{5,k} v_{k,i} \quad , \quad i=1,2,3,4.$$

The idea is to evaluate all k_i simultaneously,

$$k_1 = \lambda_1 v_{1i} y_n$$

$$k_j = \lambda(v_{j1} + v_{j2} H + \ldots + v_{jj} H^{j-1}) y_n \quad , \quad j=2,3,\ldots,m$$

Then

$$y_{n+1} = \mu(H) y_n.$$

Again, a considerable amount of work seems to be done before it is clear whether such methods are better than or less good than block methods via parallel environment.

REFERENCES

[1] Almasi, C.S., "Overview of Parallel Processing", Parallel Computer 2, North Holland, pp191-203, (1985).

[2] Babb II, R.G., "Programming parallel processors", orgen Graduate Center, Addison-Wesley Publishing Company, INC, (1988).

[3] Chen, S.C. and Kuck, D.J., "Time and Parallel Processor Bounds for Linear Recurrence Systems", IEEE Trans. on Comp., C24, 7, pp701-717, July, (1975).

[4] Delves, L.M., "Parallel Algorithms on Transputer Arrays", problem overview, working paper, Centre for Mathematical Software Research, University of Liverpool, (1987).

[5] Delves, L.M. and Pursglove, C.J., "Ada Short Course Notes", Centre for Mathematical Software Research, University of Liverpool, (1986).

[6] Enright, W.H. and Pryce, J.D., "Two Fortran Packages for Assessing Initial Value methods", ACM Transcation on Mathematical Software, vol 13, No. 1, pp1-27, March, (1987).

[7] Evans, D.J. and Hatzopoulos, M., "A Parallel Linear System Solver", Intern. J. Computer. Math., Section B, vol. 7, pp227-238, (1979).

[8] Favre, J.B., Coray, C. and Stiles, G.S., "A Parallel High-speed, Low-precision Method of a System of Ordinary Differential Equations", submitted to the Journal of Parallel Processing, June (1988).

[9] Flynn, M.J., "Very High-speed Computing Systems", proceedings of IEEE, 54, 12 (December), pp1901-1909, (1966).

[10] Flynn, M.J., "Some Computer Organizations and their Effectiveness", IEEE TRANION Computers, vol. C-21, No. 9, September, (1972).

[11] Franklin, M.A., "Parallel Solution of Ordinary Differential Equations", IEEE Trans. Computers, vol. C-27, pp413-420, (1978).

[12] Frenkel, K.A., "Evaluating Two Massivley Parallel Machines", Communication of ACM, vol. 29, no. 8, pp750-758, (1986).

[13] Gear, C.W., "The Potential for Parallelism in Ordinary Differential Equations", Report No UIUCDCS-r-86-1246, Dept of Computer Science, University of Illinois, Urbana-Champaign, Urbana, (1986).

[14] Gill, P.E., Golub, G.H., Murray, W. and Saundres, M.A., "Methods for Modifying Matrix Factorization", Math. Comput. 28, pp505-535, (1974).

[15] Hall, G. and Watt, J.M., "Modern Numerical Methods for Ordinary Differential Equations", Oxford University Press, (1976).

[16] Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra", SIAM Review, 20, 4, pp740-776, (1978).

[17] Henrici, P., "Discrete Variable Methods in Ordinary Differential Equations", New York, John Wiley and Sons, (1962).

[18] Hockney, R.W. and Jesshope, C.R., "Parallel Computers:Architecture Programming and Algorithms", Adam Hilger, Bristol, England, (1981).

[19] Howard, P.G.N., "Timing Parallel Ada Programs on the Sequent Balance", Parallel Processing Research Group Working Paper, Centre for Mathematical Software Research, University of Liverpool, December, (1987).

[20] Howard, P.G.N., "Simultaneous Linear Equations on Transputer Arrays", Parallel Processing Group Working Paper, Centre for Mathematical Software Research, University of Liverpool, December, (1987).

[21] Kuck, D.J., "Structure of Computers and Computations", John Wiley & Son, New York, (1978).

[22] Kuck, D.J. and Maruyama, "Time Bounds on the Parallel Evaluation of Arithmetic Expressions", SIAM, J. Comput. Vol 4, No 2, (1975).

[23] Kumar, S.P. and Kowalik, J.S., "Triangularization of Positive Definite Matrix on Parallel Computer", J. of Parallel and Distributed Computing, 3, pp450-460, (1986).

[24] Lambert, D., "Computational Methods in Ordinary Differential Equations", New York, Wiley, (1973).

[25] Lapidus, L. and Seinfeld, J.H., "Numerical Solution of Ordinary Differential Equations", Academic Press, (1971).

[26] Lord, R.E., Kowalik, J.S. and Kumar, S.P., "Solving Linear Algebraic Equations on MIMD Computer", J. Assoc. Comput. Mach., 30, 1, pp103-117, (1983).

[27] Melhem, R., "Parallel Gauss-Jordan Elimination for the Solution of Dense Linear Systems", Parallel Computing, 4, pp339-343, (1987).

[28] Miranken, W.L. and Linginer, W., "Parallel Methods for the Numerical Integration of Ordinary Differential Equations", Math. Comp., vol. 21, pp303-320, (1967).

[29] Moody, T.C. and Hans, H., "Parallel Solution of ODE's by Multi-block Methods", SIAM, J. Sci., STAT. Comput., vol. 8, pp342-353, (1987).

[30] Nievergelt, J., "Parallel Methods for Integration Ordinary Differential Equations", Commu. Ass. Comput. Mach., vol. 7, pp731-733, (1964).

[31] Oseterhaug, A., "Guide to Parallel Programming on Sequent Computer Systems", Sequent Technical Publication, published by Sequent Computer Systems Inc.

[32] Packer, J., "Exploiting Concurrency:a Ray Tracing Example", INMOS publication.

[33] Phillips, G.M. and Taylor, P.J., "Theory and Applications of Numerical Analysis", Academic Press INC, (1973.

[34] Quinn, M.J., "Designing Efficient Algorithms for Parallel Computers", McGraw-Hill International Editions, Computer Science Series, (1988)

[35] Rohrbacher, D. and Potter, J.L., "Image Processing with the STARAN Parallel Computer", Computer pp54-59, (1977).

[36] Rosser, J.B., "A Runge-Kutta for all Seasons", SIAM Rev., vol. 9, pp417-452, (1967).

[37] Russell, R.M., "The CRAY-1 Computer Systems", Communication of ACM, pp63-72, January, (1978).

[38] Sameh, A.H., "Numerical Parallel Algorithms - A Survey", Highspeed Computers and Algorithm Organization, pp207-228, (1977).

[39] Sameh, A. and Kuck, D.J., "A Parallel QR Algorithm for Symmetric Tridiagonal Matrices", IEEE Trans. Computers, C-26, pp147-153, (1977).

[40] Sameh, A. and Kuck, D.J., "On Stable Linear System Solvers", J. Assoc. Comput. Mach. 25, pp81-89, (1978).

[41] Sameh, A. and Brent, R.P., "Solving Triangular Systems on Parallel Computer", SIAM J. Numerical Anaysis, 14, 6, pp1101-1113, (1977).

[42] Schendel, U., "Introduction to Numerical Methods for Parallel Computers", John Wiley & Sons, New York, (1984).

[43] Stone, H.S., "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations", J. of ACM., vol. 20, pp27-38, (1973).

[44] Watts, H.A. and Shampine, L.F., "A Stable Block Implicit Methods", BIT, vol. 12, pp252-266, (1972).

[45] Worland, P.B., "Parallel Methods for the Numerical Solution of Ordinary Differential Equations", IEEE Trans. Computers, vol 25, pp1045-1048, (1976).

[46] Active Memory Technology, INC., "DAPS: A Massively Parallel Boost for Sun", OPEN Network, (1988).

APPENDIX A

This is to show that the Choleski's method applied to the augmented matrix

$$[\ A : \underline{b}\ ]$$

requires

$$\frac{n^3 + 9n^2 + 2n}{6} \quad \text{multiplications/divisions,}$$

$$\frac{n^3 + 6n^2 - 7n}{6} \quad \text{additions/subtractions}$$

and n square roots.

Proof:

Considering the case in which the matrix A is symmetric and positive definite, we can solve the linear system (2.1.1) using Choleski method, which states that we can find a lower triangular matrix L such that $A = LL^T$.

We have

$$A\ \underline{x} = LL^T\ \underline{x} = \underline{b}$$

Let $L^T\ \underline{x} = \underline{Z}$, so we have to solve

(i) $L\ \underline{Z} = \underline{b}$ by forward substitution and

(ii) $L^T\ \underline{x} = \underline{Z}$ by backward substitution.

The major part of Choleski method is as follows:

$$l_{ii} = (a_{ii} - \sum_{k=1}^{i-1} l_{ik}^2)^{1/2} \quad , \quad i=1,2,..,n$$

$$l_{ji} = \frac{1}{l_{ii}} (a_{ji} - \sum_{k=1}^{i-1} l_{jk} l_{ik}) \ , \quad j=i+1,..,n.$$

The total number of multiplications and divisions is

$$\sum_{i=1}^{n} \left[(i-1) + \sum_{j=i+1}^{n} \right] = \sum_{i=1}^{n} \left[(i-1) \quad + ni - i^2 \right]$$

$$= \frac{n(n+1)}{2} - n + \frac{n^2(n+1)}{2} - \frac{n(n+1)(2n+1)}{6}$$

$$= \frac{3n^2 + 3n - 6n + 3n^3 + 3n^2 - 2n^3 - 3n^2 - n}{6}$$

$$= \frac{n^3 + 3n^2 - 4n}{6}$$

The total number of additions is

$$\sum_{i=1}^{n} \left[(i-1) + \sum_{j=i+1}^{n} (i-1) \right] = \sum_{i=1}^{n} \left[(i-1) + (n-i)(i-1) \right]$$

$$= \sum_{i=1}^{n} (ni + 2i - i^2 - n-1)$$

$$= \frac{n^2(n+1)}{2} + \frac{2n(n+1)}{2} - \frac{n(n+1)(2n+1)}{6} - n^2 - n$$

$$= \frac{3n^3 + 3n^2 + 6n^2 + 6n - 2n^3 - 3n^2 - n - 6n^2 - 6n}{6}$$

$$= \frac{n^3 - n}{6}$$

Also there is n square roots.

Hence, to solve the system of equations

$L \underline{Z} = \underline{b}$ and $L^T \underline{x} = \underline{b}$, we can write the forward substitutions algorithm as

```
Z(1) = b(1)/l(1,1);
For i in 2..n loop
    sum:= b(i)/l(i,i) ;
For j in 1..i-1 loop
    sum:= sum-l(i,j) * Z(j) ;
```

End loop;

Z(i) = sum ;

End loop;

Then the total number of multiplications and divisions is

$$1 + \sum_{i=2}^{n} (1 + i-1) = 1 + \sum_{i=2}^{n} i$$

$$= 1 + \frac{n(n+1)}{2} - 1 = \frac{2 + n^2 + n - 2}{2} = \frac{n(n+1)}{2} .$$

The total number of additions

$$\sum_{i=2}^{n} (i-1) = \frac{n(n+1)}{2} - 1 - (n-1) = \frac{n^2 + n - 2 - 2n + 2}{2}$$

$$= \frac{n(n-1)}{2} .$$

The same amount of operations is required for backward substitution. Thus the total number of arithmetic operations required to solve $A \underline{x} = \underline{b}$, using Choleski's method, is

$$\frac{n^3 + 3n^2 - 4n}{6} + n(n+1) = \frac{n^3 + 9n^2 + 2n}{6} \text{ multiplication,}$$

$$\frac{n^3 - n}{6} + n(n-1) = \frac{n^3 + 6n^2 - 7n}{6} \text{ additions and n square roots.}$$

This is about half the number of arithmetic operations that are required using Gauss elimination method.

APPENDIX B

Ths is to show that the matrix J (refer to section 2.10 in Chapter 2) has the same structure as A_R.

If the tridiagonal matrix A, where the diagonal elements are all unity, is considered, then A can be written as

$$A = A_L + I + A_R = L_R$$

where

$$A_L = \begin{bmatrix} 0 & 0 & 0 & \cdots\cdots & 0 \\ a_{21} & 0 & 0 & & \vdots \\ 0 & a_{32} & 0 & & \\ 0 & 0 & a_{43} & & \\ \vdots & \vdots & 0 & & \\ 0 & 0 & 0 & a_{n-1,n} & 0 \end{bmatrix}$$

$$A_R = \begin{bmatrix} 0 & a_{12} & 0 & 0 \cdots\cdots & 0 \\ 0 & 0 & a_{23} & 0 & \vdots \\ \vdots & \vdots & 0 & a_{34} & \\ & & \vdots & 0 & \\ & & & \vdots & a_{n-1,n} \\ 0 & 0 & 0 & 0 \cdots\cdots & 0 \end{bmatrix}$$

and $L = A_L + D$, $R = I + J$.

Now suppose that

$$A = \begin{bmatrix} 1 & a_{12} & 0 & \cdots\cdots & 0 \\ a_{21} & 1 & a_{23} & & \vdots \\ 0 & a_{32} & 1 & & \\ 0 & 0 & a_{43} & & \\ \vdots & \vdots & 0 & & a_{n-1,n} \\ 0 & 0 & 0 & a_{n,n-1} & 1 \end{bmatrix}$$

and that

$$L = A_L + D = \begin{bmatrix} d_1 & 0 & 0 & \cdots\cdots & 0 \\ a_{21} & d_2 & 0 & & \vdots \\ 0 & a_{32} & d_3 & & \\ \vdots & 0 & a_{43} & & 0 \\ 0 & 0 & 0 & a_{n,n-1} & d_n \end{bmatrix},$$

$$R = I + J = \begin{bmatrix} 1 & e_1 & 0 & \cdots\cdots & 0 \\ 0 & 1 & e_2 & & \vdots \\ 0 & 0 & 1 & & \\ \vdots & \vdots & 0 & & e_{n-1} \\ 0 & 0 & 0 & & 1 \end{bmatrix}$$

So, we have

$$A = L\,R \Rightarrow$$

$$\begin{bmatrix} 1 & a_{12} & 0 & 0 \ldots\ldots & 0 \\ a_{21} & 1 & a_{23} & 0 & \vdots \\ 0 & a_{32} & 1 & a_{34} & 0 \\ 0 & 0 & a_{43} & 1 & a_{n-1,n} \\ \vdots & \vdots & 0 & & \\ 0 & 0 & 0 & a_{n,n-1} & 1 \end{bmatrix} =$$

$$\begin{bmatrix} d_1 & d_1 e_1 & 0 & 0 \ldots\ldots\ldots & 0 \\ a_{21} & a_{21} e_1 + d_2 & d\ e & 0 \ldots\ldots\ldots & 0 \\ 0 & a_{32} & a_{32} e_2 + d_3 & d_3 e_3 \ldots\ldots & 0 \\ \vdots & & & & d_{n-1}\ e_{n-1} \\ 0 & 0 & 0 \ldots\ldots & a_{n-1,n} & a_{n-1,n}\ e_{n-1} + d_n \end{bmatrix}$$

Therefore,

$$d_1 = 1,\quad e_1 = a_{12},\quad d_2 = 1 - a_{21}a_{12},\quad e_2 = \frac{a_{23}}{1 - a_{21}a_{12}},\quad d_3 = \left[1 - \frac{a_{23}a_{23}}{1 - a_{21}a_{12}} \right],$$

$$e_3 = a_{34} / \left[1 - \frac{a_{32}a_{23}}{1 - a_{21}a_{12}} \right]$$

etc., where $e_1, e_2, \ldots$ are the elements of the matrix J.

Now, we give numerical example to show the above theory:

Consider $A = \begin{bmatrix} 1 & -2 & 0 & 0 \\ -2 & 1 & -2 & 0 \\ 0 & -2 & 1 & -2 \\ 0 & 0 & -2 & 1 \end{bmatrix}$

then from the above, we have

$$L = \begin{bmatrix} d_1 & 0 & 0 & 0 \\ -2 & d_2 & 0 & 0 \\ 0 & -2 & d_3 & 0 \\ 0 & 0 & -2 & d_4 \end{bmatrix} \quad \text{and} \quad R = \begin{bmatrix} 1 & e_1 & 0 & 0 \\ 0 & 1 & e_2 & 0 \\ 0 & 0 & 1 & e_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Hence,

$$e_1 = -2, \quad e_2 = \frac{-2}{1-(-2)(-2)} = 2/3 \quad \text{and}$$

$$e_3 = (-2)/\left[1 - \frac{(-2)(-2)}{-3} \right] = -2/\left[1+4/3 \right] = -6/7 \ .$$

This gives

$$J = \begin{bmatrix} 0 & -2 & 0 & 0 \\ 0 & 0 & 2/3 & 0 \\ 0 & 0 & 0 & -6/7 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Therefore J has the same structure as A_R.

APPENDIX C

This appendix is intended to show the reason why we do not use lower order multistep methods as starting procedures.

The starting values y_1, y_2, y_3,... are usually found by one-step methods. These methods are used to compute the starting values because they require no other values than the initial value y_0 [24].

It is mentioned in [33], if the order of predictor formula is m + 1, a one-step method of order m + 1 to evaluate the starting values should be chosen.

If the initial value problem is

$$y' = f(x,y) \quad , \quad y(a) = y_0 \, , \quad x \, \varepsilon \, [a,b] \tag{1}$$

then we use the following formula as predictor

$$y_{n+1} = y_n + h \, [b_0 f_n + b_1 \nabla f_n + \ldots + b_m \nabla^m f_n] \tag{2}$$

for $n = m, m+1, \ldots$, where $f_r = f(x_r, y_r)$.

In practice (2) can be written in form

$$y_{n+1} = y_n + h[\beta_0 f_n + \beta_1 f_{n-1} + \ldots + \beta_m f_{n-m}] = y_n + h \sum_{i=0}^{m} \beta_i f_{n-i} \tag{3}$$

From (2), we have

$$\frac{y_{n+1} - y_n}{h} = h \sum_{j=0}^{m} b_j \nabla^j f_n \tag{4}$$

and consider this as an approximation to (1).

Now,

$$\int_{x_n}^{x_{n+1}} f(x,y) = \int_{x_n}^{x_{n+1}} y'(x) \, dx$$

$$= h \sum_{j=0}^{m} b_j \nabla^j y'(x_n) + h^{m+2} b_{m+1} y^{(m+2)}(\xi_n)$$

where $x_{n-m} < \xi_n < x_{n+1}$

In the same time,

$$\frac{y(x_{n+1}) - y(x_n)}{h} = \int_{x_n}^{x_{n+1}} f(x,y)\, dx$$

$$\Rightarrow \frac{y(x_{n+1}) - y(x_n)}{h} = \sum_{j=0}^{m} b_j \nabla^j f(x_n, y(x_n)) + h^{m+1} b_{m+1} y^{(m+2)}(\xi_n) \qquad (5)$$

where

$$\nabla^j f(x_n, y(x_n)) = \nabla^{j-1} f(x_n, y(x_n)) - \nabla^{j-1} f(x_{n-1}, y(x_{n-1})).$$

The truncation error is $h^{m+1} b_m y^{(m+2)}(\xi_n)$ which is of order $m+1$.

From (2), we have

$$y(x_{n+1}) = y(x_n) + h[\beta_0 f(x_n, y(x_n)) + \beta_1 f(x_{n-1}, y(x_{n-1})) + \ldots$$

$$+ \beta_m f(x_{n-m}, y(x_{n-m})] + h^{m+2} b_{m+1} y^{(m+2)}(\xi_n). \qquad (6)$$

Assume that $y^{(m+2)}(x)$ is bounded for $x \varepsilon [x_0, b]$, so there exists M_{m+2} such that

$$\max_{x \varepsilon [x_0, b]} \left| y^{(m+2)}(x) \right| \leq M_{m+2}$$

Let $e_n = |y(x_n) - y_n|$, and by Lipschitz condition on f (where e_n is global error)

$$| f(x_n, y(x_n)) - f(x_n, y_n) | \leq L | y(x_n) - y_n | = L e_n.$$

Subtracting (2) from (6), we obtain

$$y(x_{n+1}) - y_{n+1} = y(x_n) - y_n + h[\beta_0 (f(x_n, y(x_n) - f_n) + \ldots + \beta_m (f(x_{n-m}, y(x_{n-1,m})) - f_{n-m}]$$

$$+ h^{m+2} b_{m+1} y^{(m+2)}(\xi_n).$$

$$\Rightarrow \quad e_{n+1} \leq e_n + h L \left[|\beta_0| e_n + |\beta_1| e_{n+1} + .. + |\beta_m| e_{n-m} \right]$$

$$+ h^{m+2} b_{m+1} |M_{m+2}|.$$

Suppose that the starting values $y_0, \ldots, y_m$ are such that

$$e_r \leq \delta \text{ for } n = 0,1,.., m.$$

Lemma: If the sequence $\{u_n\}_{n=0}^{\alpha}$ of positive needs satisfies the recurrent inequality

$$u_{n+1} \leq u_n + \alpha_0 u_n + \alpha_1 u_{n-1} + \ldots + \alpha_m u_{n-m} + b$$

for $n=m, m+1, \ldots$, where $\alpha_r \geq 0$, $r = 0,1,\ldots,m$ and $b \geq 0$, then

$$u_n \leq (\delta + \frac{b}{A}) e^{nA} - \frac{b}{A}$$

for $n = 0,1,2,\ldots$, where δ is any real number satisfying

$$\delta \geq u_r \quad , \quad r = 0,1,\ldots, m$$

and $A = \alpha_0 + \alpha_1 + \alpha_2 + \ldots + \alpha_m \neq 0.$

Applying the above lemma with

$$\alpha_r = h L |\beta_r| \quad \text{and} \quad b = h^{m+2} |b_{m+1}| M_{m+2},$$

so that $A = \alpha_0 + \alpha_1 + ,\ldots, + \alpha_m = h L B_m,$

where $B_m = |\beta_0| + |\beta_1| + ,\ldots, + |\beta_m| .$

Then

$$e_n \leq \left(\delta + \frac{h^{m+1} |b_{m+1}| m_{m+2}}{L B_m} \right) e^{(x_n - x_o) L B_n} - \frac{h^{m+1} |b_{m+1}| m_{m+2}}{L B_m} \quad (7)$$

where $nh = x_n - x_0.$

The above inequality (7) provides a global error for Adams-Bashforth method, it tells us the behaviour of the error as h is decreased.

From the first term on right hand-side, it can be seen if the method is to be of order $m+1$, then we require δ, a bound for the error in the starting values to decrease like h^{m+1} as h tends to zero.

Thus we should choose a one-step method of order $m+1$ to determine starting values.

The best known one step methods are Runge-Kutta methods, and they are usual means for calculating the initial values. They are fairly simple to program and self-starting.

APPENDIX D

Timing Arithmetic Operations

We present here a selection of the results of a series of tests that have been run on Sequent Balance. The set of programs that were used can be divided into three parts:

1) program with scalar operations,

2) program with linear algebra code as follows:

vector add, vector subtract, vector multiply and vector divide,

3) a program to copy each component of the vector into scalar.

All the computations are done on floating points numbers, using one processor. Each vector contains an order set of n elements, where n is called length of the vector.

The scalar operation program is written to do fifty-one operations and they construct like

For i in 1..n rept loop

$x := x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x_{11}$;

$x := x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x$;

$x := x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x$;

$x := x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x$;

$x := x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 + x_9 + x_{10} + x$;

$x := x + x$;

End loop;

where n rept is the outer loop.

We give the timing results which are obtained by reading the clock just before and after the calculation of the numbers.

1) Time to do fifty-one additions

N rept	**Total time (seconds)**		**Time to do one operation(microsec)**	
	1st Run	**2nd Run**	**1st Run**	**2nd Run**
100	0.0299	0.0199	5.86	3.90
1000	0.1500	0.1400	2.94	2.75
10000	1.3699	1.3609	2.68	2.67
100000	13.640	12.910	2.67	2.53

2) Time to do fifty-one subtractions

100	0.01996	0.0300	3.91	5.88
1000	0.14996	0.1399	2.94	2.74
10000	1.29998	1.2899	2.55	2.53
100000	12.7899	12.789	2.51	2.51

3) Time to do fifty-one multiplications

100	0.03292	0.0299	6.45	5.86
1000	0.14001	0.1400	2.75	2.75
10000	1.32000	1.2990	2.59	2.55
100000	12.9000	13.120	2.53	2.57

4) Time to do fifty-one divisions

100	0.03003	0.03003	5.89	5.89
1000	0.1699	0.1699	3.33	3.33
10000	1.5599	1.5800	3.06	3.10
100000	15.559	15.5100	3.05	3.04

Addition and subtraction are slightly cheaper than multiplication, and multiplication in turn is cheaper than division.

The second program is a linear algebra code to time vector add, vector subtract, vector multiply and vector division.

Where the procedures vary only in the line of code specifying, the operation timed as follows:

$$c(i) := a(i) + b(i) \quad ; \quad i = 1, \ldots, n.$$

There are two variables:

n the vector length, n rept the number or repetition of the outer loop.

The programs run four times to ensure that the timings are consistent.

The results are given as follows:

1) Timing Vector Add

Vector Length	Number of rept	Total Time (seconds)	Millisec per Operation
200	100	5.109	0.255
		5.099	0.254
		5.101	0.255
		5.109	0.254
300	100	7.550	0.251
		7.590	0.253
		7.579	0.252
		7.590	0.253
400	100	10.199	0.254
		10.210	0.255
		10.210	0.255
		10.199	0.254
500	100	12.699	0.253
		12.659	0.253
		12.699	0.253
		12.699	0.253

2) Timing Vector Subtract

Vector Length	Number of Rept	Total Time (Seconds)	Millisec per Operation
200	100	5.119	0.255
		5.099	0.254
		5.101	0.255
		5.099	0.254
300	100	7.579	0.253
		7.600	0.253
		7.599	0.253
		7.620	0.254
400	100	10.210	0.255
		10.212	0.255
		10.199	0.254
		10.290	0.254
500	100	12.690	0.254
		12.789	0.255
		12.699	0.254
		12.681	0.254

3) Timing Vector Multiply

Vector Length	Number of rept	Total Time (seconds)	Millisec per Operation
200	100	5.119	0.255
		5.119	0.255
		5.101	0.255
		5.099	0.254
300	100	7.679	0.255
		7.686	0.255
		7.610	0.254
		7.679	0.255
400	100	10.210	0.255
		10.220	0.255
		10.229	0.255
		10.229	0.255
500	100	12.690	0.254
		12.710	0.254
		12.699	0.254
		12.700	0.254

4) Timing Vector Division

Vector Length	Number of Rept	Total Time (Seconds)	Millisec per Operation
200	100	5.120	0.256
		5.130	0.257
		5.129	0.256
		5.129	0.256
300	100	7.689	0.256
		7.690	0.256
		7.699	0.257
		7.689	0.256
400	100	10.240	0.256
		10.310	0.257
		10.242	0.256
		10.300	0.257
500	100	12.850	0.257
		12.832	0.257
		12.820	0.257
		12.841	0.257

The third program is to copy each component of the vector into variable x The procedure is as follows:

```
For i in 1..n loop
    x:= a(i);
End loop;
```

The timings results are given in the following table:

Vector Length	Number of Rept	Total Time (Seconds)	Millisec to Copy One Component
200	100	1.659	0.083
		1.669	0.083
		1.670	0.083
		1.670	0.083
300	100	2.489	0.083
		2.489	0.083
		2.489	0.083
		2.489	0.083
400	100	3.309	0.083
		3.299	0.083
		3.320	0.083
		3.320	0.083
500	100	4.119	0.083
		4.139	0.083
		4.130	0.083
		4.130	0.083

The number of arithmetic operations in the second program is n, so the average time to do one opeation is approximately 0.25 millisec. The time to copy one component is 0.083 millisec. So, we have three components which need to copy each time and one operation which gives 0.083 * 3 + 0.003 = 0.25 millisec.

```
--Timing Arithmetic (Scalar) Operations
with text_io,float_io,integer_io,math;
use text_io,float_io,integer_io,math;
with calendar;use calendar;
procedure test_arith is
subtype real is float;
nrept:integer;
next_time:calendar.time;
time_interval:duration:=0.0;
begin
get(nrept);
declare
x:float;
x1:constant float:=1.1;
x2:constant float:=2.2;
x3:constant float:=3.3;
x4:constant float:=4.4;
x5:constant float:=5.5;
x6:constant float:=6.6;
x7:constant float:=7.7;
x8:constant float:=8.8;
x9:constant float:=9.9;
x10:constant float : =10.1;
x11:constant float : =11.11;
begin
new_line;
put("ready to start.");new_line;
next_time:=calendar.clock;
for k in 1..nrept loop
--Fifty_one additions
x:=x1+x2+x3+x4+x5+x6+x7+x8+x9+x10+x11;
x:=x1+x2+x3+x4+x5+x6+x7+x8+x9+x10+x;
x:=x1+x2+x3+x4+x5+x6+x7+x8+x9+x10+x;
x:=x1+x2+x3+x4+x5+x6+x7+x8+x9+x10+x;
```

```
x:=x1+x2+x3+x4+x5+x6+x7+x8+x9+x10+x;
x:=x+x;
--Fifty_one divisions
--x:=x1/x2/x3/x4/x5/x6/x7/x8/x9/x10/x11;
--x:=x1/x2/x3/x4/x5/x6/x7/x8/x9/x10/x;
--x:=x1/x2/x3/x4/x5/x6/x7/x8/x9/x10/x;
--x:=x1/x2/x3/x4/x5/x6/x7/x8/x9/x10/x;
--x:=x1/x2/x3/x4/x5/x6/x7/x8/x9/x10/x;
--x:=x/x;
--Fifty_one multiplications
--x:=x1*x2*x3*x4*x5*x6*x7*x8*x9*x10*x11;
--x:=x1*x2*x3*x4*x5*x6*x7*x8*x9*x10*x;
--x:=x1*x2*x3*x4*x5*x6*x7*x8*x9*x10*x
--x:=x1*x2*x3*x4*x5*x6*x7*x8*x9*x10*x;
--x:=x1*x2*x3*x4*x5*x6*x7*x8*x9*x10*x;
--x:=x*x;
 --Fifty_one     subtractions
--x:=x1-x2-x3-x4-x5-x6-x7-x8-x9-x10-x11;
--x:=x1-x2-x3-x4-x5-x6-x7-x8-x9-x10-x;
--x:=x1-x2-x3-x4-x5-x6-x7-x8-x9-x10-x;
--x:=x1-x2-x3-x4-x5-x6-x7-x8-x9-x10-x;
--x:=x1-x2-x3-x4-x5-x6-x7-x8-x9-x10-x;
--x:=x-x;
end  loop;
time_interval:=(calendar.clock-next_time)+time_interval;
put("time_interval");
put(real(time_interval));new_line;
end;
end test_arith;
```

```
--Program With Linear Algebra Code
with text_io,float_io,integer_io;
use text_io, float_io,integer_io;
with calendar;use calendar;
procedure test_mat is
subtype real is float;
n,nrept,i:integer;
next_time:calendar.time;
time_interval:duration:=0.0;
begin
get(n); --size of the vector
get(nrept); --number of repetition;
declare
      x:float;
      type vector is array(1..n) of float;
      a,b,c:vector;
procedure compute (a,b:in vector;c:out vector) is
begin
for i in 1..n loop
c(i):=a(i)+b(i);
--c(i):=a(i)-b(i);
--c(i):=a(i)*b(i);
--c(i):=a(i)/b(i);
end loop;
end compute;
--The following procedure is to copy each element of the vector
--into variable x
--procedure check_vec(a:in vector;x:in out float) is
--begin
--for i in 1..n loop
--x:=a(i);
--end loop;
--end check_vec;
```

```
begin
for i in 1..n loop
a(i):=float(i);
b(i):=float(i*2);
end loop;
new_line;
put("ready to start");new_line;
next_time:=calendar.clock;
for k in 1..nrept loop
compute(a,b,c); --this is for add two vectors;
--check_vec(a,x); --this is to copy the vector components in the var x;
end loop;
time_interval:=(calendar.clock-next_time)+time_interval;
put("time_interval");new_line;
put(real(time_interval));new_line;
end;
end test_mat;
```

APPENDIX E

Timings for Rendezvous

This appendix investigates the rendezvous timings, where a parallel program was written which contains m identical tasks, each has an accept statement which is in a loop.

The main procedure is given by

```
For i m 1...n loop
    For j in 1...m loop
        E V (j), send;
    End loop;
End loop;
```

where n is the number of rendezvous.

In the Sequent Balance it is easy to specify the number of processors to be used to run a parallel program containing tasks. In our experiment the number of rendezvous and the number of repetitions are fixed and equal to 100, whereas the number of tasks varys from 1 to 8 and the corresponding number of processors is 2 to 9; this is to allow each task and main procedure work in separate processor.

The results are displayed in the following table:

Timing the Rendezvous

Number of Tasks	Number of Processors	Number of Rendezvous	Number of Repetition	Total Time (Seconds)	Time to Set One Rendezvous (Millisec)
1	2	100	100	20.709	2.07
2	3	100	100	42.959	2.14
3	4	100	100	67.229	2.24
4	5	100	100	95.300	2.38
5	6	100	100	127.880	2.56
6	7	100	100	164.790	2.75
7	8	100	100	209.909	2.99
8	9	100	100	265.399	3.32

Therefore, the average time to set one rendezvous is approximately 2 millisec. This is the same result which was obtained by [19].

```
--Timings For Rendezvous
with text_io,float_io,integer_io,math;
use text_io,float_io,integer_io,math;
with calendar;use calendar;
procedure test_rend is
subtype real is float;
n,m,nrept:integer;
begin
get(n); --number of rendezvous
get(m); --number of tasks
get(nrept); --number of repetition;
declare
next_time:calendar.time;
time_interval:duration:=0.0;
procedure evaluate(n,m:integer) is
task type evaluation is
entry send;
end;
EV:array(1..m) of evaluation;
task body evaluation is
count:integer:=0;
begin
loop
   select
      accept send;
      count:=count+1;
  or
  terminate;
  end select;
end loop;
end evaluation;
begin
for i in 1..n loop
for j in 1..m loop
EV(j).send;
end loop;
```

```
end loop;
end evaluate;
begin
put("ready to start");new_line;
next_time:=calendar.clock;
for s in 1..nrept loop
evaluate(n,m);
end loop;
time_interval:=(calendar.clock-next_time)+time_interval;
put("time_interval");new_line;
put(real(time_interval));new_line;
end;
end test_rend;
```

APPENDIX F

Timing Function Evaluations

This appendix is to investigate the time of functions which are used in chapters 5 and 6. The sequential programs which were written to time these functions vary only in the line of code, specifying the function timed, as an example test problem (iv):

```
function dy(x,y:vector) return vector is
    b:vector (1..k);
    For i in 1..k loop
    b(i):= y(i) * cos(x(i));
    End loop;
    Return b;
End dy;
```

where k is the block length.

Calling the above function which is given by

```
For m in 1..n rept loop
    For l in 1..n loop
        Z:= dy(x,y)
    End loop;
End loop;
```

where n rept is the number of repetitions and n is the number of calling the function.

As we know, test problem (i) contains no arithmetic operations, so to keep the cost of this function consistent we need to make the number of call $n \geq 10$. Moreover, a test for $y' = \cos x$ is carried out. The timing results are given as follows:

Test Problem (i)

k(block length)	n rept	n	Total Time (Seconds)	Microsec per function evaluation
200	1	10	0.0500	25.0
200	2	10	0.0999	25.0
200	3	10	0.14001	23.3
200	10	10	0.4299	21.5
200	1	100	0.4299	21.5
200	2	100	0.8400	21.0
200	3	100	1.26001	21.0
200	10	100	4.1700	20.9
400	1	10	0.0999	25.0
400	2	10	0.1799	22.5
400	3	10	0.2599	21.7
400	10	10	0.84003	21.0
400	1	100	0.84003	21.0
400	2	100	1.64996	20.6
400	3	100	2.48999	20.7
400	4	100	8.20012	20.5
500	1	10	0.1199	24.0
500	2	10	0.20996	21.0
500	3	10	0.330017	22.0
500	10	10	1.03997	20.8
500	1	100	1.0400	20.8
500	2	100	2.0599	20.6
500	3	100	3.0999	20.7
500	10	100	10.2600	20.5

Test Problem (ii)

k(block length)	n rept	n	Total Time (Seconds)	Millisec Per Function
200	1	10	0.79004	0.395
200	2	10	1.55999	0.390
200	3	10	2.33002	0.388
200	10	10	7.73999	0.387
400	1	10	1.599976	0.399
400	2	10	3.1099853	0.389
400	3	10	4.6699829	0.389
400	10	10	1.555999	0.389
500	1	10	1.94000	0.388
500	2	10	3.88000	0.388
500	3	10	5.829956	0.388
500	10	10	19.320007	0.386

Test Problem (iii)

k(block length)	n rept	n	Total Time (Seconds)	Millisec Per Function
200	1	10	0.81994	0.409
200	2	10	1.620056	0.405
200	3	10	2.429993	0.405
200	10	10	8.049987	0.403
400	1	10	1.630004	0.408
400	2	10	3.2200317	0.403
400	3	10	4.829956	0.403
400	10	10	16.090026	0.402
500	1	10	2.019958	0.404
500	2	10	4.029968	0.403
500	3	10	6.049987	0.403
500	10	10	2.0049987	0.401

Timing the Function y′ = cos x

k(block length)	n rept	n	Total Time (Seconds)	Millisec per Function Evaluation
200	1	1	0.1500	0.750
200	2	1	0.2899	0.725
200	3	1	0.41998	0.700
200	10	1	1.39000	0.695
200	1	10	1.3700	0.685
200	2	10	2.7300	0.683
200	3	10	4.0900	0.681
200	10	10	13.5700	0.679
400	1	1	0.290039	0.725
400	2	1	0.570007	0.713
400	3	1	0.840027	0.700
400	10	1	2.729980	0.683
400	1	10	2.7300	0.683
400	2	10	5.46999	0.684
400	3	10	8.15002	0.679
400	10	10	27.14002	0.679
500	1	1	0.35998	0.720
500	2	1	0.70001	0.700
500	3	1	1.03997	0.693
500	10	1	3.41998	0.684
500	1	10	3.440002	0.688
500	2	10	6.80999	0.681
500	3	10	10.19000	0.679
500	10	10	33.92999	0.679

Test Problem (iv)

k (block length)	n rept	n	Total Time (seconds)	Multisec Per Function
200	1	10	1.6029	0.801
200	2	10	3.2049	0.801
200	3	10	4.8061	0.801
200	10	10	16.040	0.802
400	1	10	3.2083	0.802
400	2	10	6.4249	0.803
400	3	10	9.6002	0.800
400	10	10	32.000	0.800
500	1	10	4.000	0.800
500	2	10	8.010	0.801
500	3	10	12.000	0.800
500	10	10	40.000	0.800

```
--Timing Function Evalutions
with text_io,float_io,integer_io , math;
use text_io,float_io,integer_io,math;
with calendar;use calendar;
procedure test_function is
subtype real is float;
k,nrept,n:integer;
begin
get(k);  --block length
get(nrept);
get(n);
declare
next_time:calendar.time;
time_interval:duration:=0.0;
     type vector is array(integer range<>) of float;
     x,y,z:vector(1..k);
--Problem (1)
function dy(y:vector) return vector is
begin
return y;
end dy;
--problem (4)
--function dy(x,y:vector)return vector is
--b:vector(1..k);
--begin
--for i in 1..k loop
--b(i):=y(i)*cos(x(i));
--end loop;
--return b;
--end dy;
begin
for i in 1..k loop
x(i):=float(i/2);
y(i):=float(i);
end loop;
```

```
put("ready to start.");new_line;
next_time:=calendar.clock;
for m in 1..nrept loop
for l in 1..n loop
z:=dy(y);
--z:=dy(x,y);
end loop;
end loop;
time_interval:=(calendar.clock-next_time)+time_interval;
put("time_interval");new_line;
put(real(time_interval));new_line;
end;
end test_function;
```

APPENDIX G

Timing Data Sending

In this appendix we present two programs as follows:

i) A program to give a total time of the rendezvous and data sending.

ii) A program to give a total time of the rendezvous, data sending and function evaluations.

Both of them are written in parallel code.

Timing the first program, so the total time is the summation of total number of rendezvous plus the data sending, where the time to set one rendezvous is already known. Therefore the value of s can be estimated as

number of rendezvous * r + number of data sending * s = total time

$$\Rightarrow s = \frac{\text{total time - time of rendezvous}}{\text{number of data sending}}$$

Second program is tested when test problem IV was used. Here we calculate the predicted time which consists of the time of rendezvous, time of function evaluation and time of data sending. By substituting the quantities r,f,s (s=0.02 millisec) in the predicted time and comparing the predicted time with the actual time of program 2, we find them similar.

The timings results are given as follows:

Total time of programs (i) and (ii)

Number of Processors	Number of Tasks	Sub block q	Number of rept	Total time of (rend.+ data sending)	Total time of (rend. + data sending + function eval)
3	2	200	100	2.669 2.690 2.659	20.07 20.01 20.04
4	3	200	100	4.320 4.330 4.309	22.21 21.98 22.08
5	4	200	100	6.309 6.280 6.309	24.13 24.13 24.20

The predicted time of (i) is

2 * number of tasks * r + 3q * number of tasks * s

We know that r = 2 millisec, so the value of s can be estimated as follows:

1) 8 + 1200 s = 26.69 $\Rightarrow$ s = 0.02 millisec

2) 12 + 1800 s = 43.20 $\Rightarrow$ s = 0.02 millisec

3) 16 + 2400 s = 63.09 $\Rightarrow$ s = 0.02 millisec

Now, the predicted time of (ii) is

2 * number of tasks * r + 3q * number of tasks * s + q * f

where r = 2 millisec. f = 0.800 millisec and s = 0.02 (from above)

	Predicted (seconds)	Actual (seconds)
1) 8 + 24 + 160=	0.192	0.201
2) 12 + 36 + 160=	0.208	0.221
3) 16 + 48 + 160=	0.224	0.241

```
--This Program Gives a Total Time of Rendezvous and Data Sending
with text_io, float_io, integer_io , math;
use text_io,float_io,integer_io , math;
with calendar ; use calendar;
procedure data_send is
subtype real is float;
i,k,q,nrept,m:integer;
begin
get(k);  --the blocklength
get(q);  --the sub block;
get(nrept);  --the number of reptition;
declare
next_time:calendar.time;
time_interval:duration:=0.0;
      type vector is array(integer range<>) of float;
      x,y:vector(1..k);
      type integer_vector is array(1..k/q) of integer;
      min,max:integer_vector;
--procedure put (x:in vector) is
--begin
--for i in 1..q loop
--put(x(i));new_line;
--new_line;
--end loop;
--end put;
procedure check(i,k,q:integer) is
task type check_send is
entry       send_initial(itemp,qtemp:integer);
entry   send_vec(xtempi,ytempi:in vector);
entry   collect(ytempi:out vector);
end;
T:array(1..k/q) of check_send;
task body check_send is
   i,q:integer;
begin
accept   send_initial(itemp,qtemp:integer)   do
```

```
i:=itemp;q:=qtemp;
end;
loop
select
accept send_vec(xtempi,ytempi:in vector) do
xnew:=xtempi;ynew:=ytempi;
end;
accept collect(ytempi:out vector) do
ytempi:=ynew;
end;
or
terminate;
end select;
end loop;
end check_send;
begin
for i in 1..k/q loop
min(i):=(i-1)*q+1;
max(i):=i*q;
end loop;
for i in 1..k/q loop
T(i).send_initial(i,q);
end loop;
for i in 1..k/q loop
T(i).send_vec(x(min(i)..max(i)),y(min(i)..max(i)));
end loop;
for i in 1..k/q loop
T(i).collect(y(min(i)..max(i)));
end loop;
end check;
begin
for i in 1..k loop
x(i):=float(i);
y(i):=float(i*2);
```

```
end loop;
put("ready to start");new_line;
next_time:=calendar.clock;
for s in 1..nrept loop
check(i,k,q);
end loop;
time_interval:=(calendar.clock-next_time)+time_interval;
put("time_interval");new_line;
put(real(time_interval));new_line;
end;
end data_send;
```

```
--This Program Gives a Total Time of The Rendezvous,Data Sending
--and Function Evaluation for Problem (4)
with text_io,float_io,integer_io,math;
use text_io,float_io,integer_io,math;
with calendar;use calendar;
procedure rend_func is
subtype real is float;
i,k,q,nrept:integer;
begin
get(k); --block length
get(q); --sub block length
get(nrept); --number of repetition
declare
next_time:calendar.time;
time_interval:duration:=0.0;
   type vector is array(integer range<>) of float;
   x,y,z:vector(1..k);
function dy(x,y:vector) return vector is
b:vector(1..q);
begin
for i in 1..q loop
b(i):=y(i)*cos(x(i));
end loop;
return b;
end dy;
--procedure put(x:in vector) is
--begin
--for i in 1..q loop
--put(x(i));new_line;
--new_line;
--end loop;
--end put;
procedure evaluate(k,q,i:integer) is
task type function_evaluation is
```

```
entry send_initial(itemp,qtemp:integer);
entry send(xtempi,ytempi:in vector);
entry collect(ztempi:out vector);
end;
Fv:array(1..k/q) of function_evaluation;
task body function_evaluation is
i,q:integer;
begin
accept send_initial(itemp,qtemp:integer) do
i:=itemp;q:=qtemp;
end;
declare
xnew,ynew,znew:vector(1..q);
begin
accept send(xtempi,ytempi:in vector) do
xnew:=xtempi;ynew:=ytempi;
end;
znew:=dy(xnew,ynew);
accept collect(ztempi:out vector) do
ztempi:=znew;
end;
end;
end function_evaluation;
begin
for i in 1..k/q loop
FV(i).send_initial(i,q);
end loop;
for i in 1..k/q loop
FV(i).send(x((i-1)*q+1..i*q),y((i-1)*q+1..i*q));
end loop;
for i in 1..k/q loop
FV(i).collect(z((i-1)*q+1..i*q));
end loop;
end evaluate;
```

```
begin
for i in 1..k loop
x(i):=float(i);


y(i):=float(i*2);
end loop;
put("ready to start");new_line;
next_time:=calendar.clock;
for s in 1..nrept loop
evaluate(k,q,i);
end loop;
time_interval:=(calendar.clock-next_time)+time_interval;
put("time_interval");new_line;
put(real(time_interval));new_line;
end;
end rend_func;
```

APPENDIX H

This appendix describes the load balancing in the algorithms mentioned in Chapters 5 and 6 in detail.

First, we start with algorithm (1) which was mentioned in Chapter 5.

The predictor and corrector formulae are given by:

$$p: \quad y_{n+r} = y_n + hy'_n$$

$$c: \quad y_{n+r,s+1} = y_n + h(a_{r,r-1} \quad y'_{n+r-1,s} + a_{r,r} \quad y'_{n+r,s}); \; s=0,1.$$

Therefore, there are three task types : predictor_formula, function_ evaluations and corrector_formula.

Each task type consists of k/q tasks working in parallel and no data are passed from one task to another. The partition for the data allows each processor to perform calculation for q points, where q is the grain size (make q as large as possible). Each task does not depend on the results of a previous task. In this case all the predictor, function_evaluationd tasks and corrector tasks have the same amount of work.

We can say now, algorithm (1) is an example of fully independent tasks, no data is passed from one task to another. Data partitioning, which is very important to keep all the processors busy and tend to lead to load balancing [2].

Thus, algorithm (3) and algorithm of the 4th order method, mentioned in Chapters 5 and 6 respectively, have the same properties, even if there is only one task type and less amount of work to do.

We turn now to algorithm (4), which was mentioned in Chapter 5. This algorithm has three tasks, the first two, predictor_formula and function evaluation, have the same properties as in algorithm 1. The third, corrector_formula, which includes task 1, requires little effort, whereas task k/q requires the largest effort of all. Unfortunately, the corrector_formula task type of algorithm (4) contains k/q nonhomogenous tasks. This does not allow these tasks to perform simultaneously. This means that task 1, which has little effort finishes first and task k/q, which has the largest effort of all, finishes last.

As results, although all these tasks work at the same time, every one of them finishes before the next, but the last one remains busy.

APPENDIX I

Parallel Programming for the Solution of Initial Value Problems

```
--This program solves Initial Value problems Using Method (1)
with text_io, float_io,integer_io, math;
use text_io,float_io,integer_io,math;
procedure method_1 is
k,q,nsteps : integer;
begin
get(k);
get(q);
get(nsteps);
declare
    n:integer;
    type vector is array(integer range< >) of float;
    x,y,z,true_solution,error:vector(1..k);
    type col is array(integer range< >) of float;
    a,b:col(1..k);
    type integer_vector is array(1..k/q) of integer;
    min,max:integer_vector;
    xstart,xend,initial_value,zz,h,max_error:float;
function dy(y:float) return float is
begin
--return y;
return -0.5*(y**3);
end dy;
procedure mult(zz:in float;z:in vector;col_0,col_r:in col;
                i,q:integer;ynew:out vector) is
begin
ynew(1):=initial_value+h*(col_0(1)*zz+col_r(1)*z(1));
for r in 2..q loop
ynew(r):=initial_value+h*(col_0(r)*z(r-1)+col_r(r)*z(r));
end loop;
end mult;
--the following task evaluates the predicted_y
 task type predicted_value is
 entry      send_initial(itemp,qtemp:integer;step_size:in      float);
 entry     send_data(initial,value:in      float);
 entry    collect(ytempi:out    vector);
```

```
 end;
 task body predicted_value is
 initial_value,h,zz:float;
 i,q:integer;
 begin
      accept send_initial(itemp,qtemp:integer;step_size:in float) do
      i:=itemp;q:=qtemp;h:=step_size;
      end;
      declare
      ynew,w:vector(1..q);
      begin
   loop
     select
      accept send_data(initial,value:in float) do
      initial_value:=initial;zz:=value;
      end;
     zz:=h*zz;
     w(1):=initial_value+float(1+(i-1)*q)*zz;
     for r in 2..q loop
     w(r):=w(r-1)+zz;
     end loop;
     for r in 1..q loop
     ynew(r):=w(r);
     end loop;
     accept collect(ytempi:out vector) do
     ytempi:=ynew;
     end;
     or
     terminate;
    end select;
   end loop;
   end;
end predicted_value;
--the following task computes the y'(n+j)
 task type function_evaluation is
```

```
 entry send_initial(itemp,qtemp:integer);
 entry   send_data(ytempi:in   vector);
 entry collect(ztempi:out vector);
 end;
  task  body  function_evaluation  is
  i,q:integer;
  begin
       accept    send_initial(itemp,qtemp:integer)    do
       i:=itemp;q:=qtemp;
       end;
     declare
     ynew,znew:vector(1..q);
     begin
     loop
      select
      accept  send_data(ytempi:in  vector)  do
      ynew:=ytempi;
      end;
      for  r  in  1..q  loop
      znew(r):=dy(ynew(r));
      end  loop;
     accept  collect(ztempi:out   vector)   do
     ztempi:=znew;
     end;
    or
   terminate;
   end   select;
 end  loop;
end;
end    function_evaluation;
 task  type  corrected_value  is
 entry        send_initial(itemp,qtemp:integer);
 entry   send_data(atempi,btempi:in   col;value:in   float;ztempi:in   vector);
 entry   receive(ytempi:out   vector);
 end;
```

```
 task body corrected_value is
 zz:float;
 i,q:integer;
begin
      accept    send_initial(itemp,qtemp:integer)    do
      i:=itemp;q:=qtemp;
      end;
    declare
    anew,bnew:col(1..q);
    znew,ynew:vector(1..q);
  begin
    loop
    select
accept send_data(atempi,btempi:in col;value:in float;ztempi:in vector)
                              do
     anew:=atempi;bnew:=btempi;zz:=value;znew:=ztempi;
     end;
    mult(zz,znew,anew,bnew,i,q,ynew);
     accept  receive(ytempi:out  vector)  do
     ytempi:=ynew;
     end;
    or
    terminate;
    end   select;
  end  loop;
  end;
  end   corrected_value;
  begin
    declare
    pv:array(1..k/q)    of    predicted_value;
    fv:array(1..k/q)    of    function_evaluation;
    cv:array(1..k/q)    of    corrected_value;
begin
for  r  in  1..k  loop
a(r):=float(r**2)/2.0;
```

```
b(r):=float(r)-(float(r**2)/2.0);
end loop;
        xstart:=0.0;
      xend:=20.0;
   --   xend:=1.0;
        h:=(xend-xstart)/float(nsteps);
   --     put("the step_size is");new_line;
   --     put(h);new_line;
        initial_value:=1.0;
  for i in 1..k/q loop
  pv(i).send_initial(i,q,h);
  fv(i).send_initial(i,q);
  cv(i).send_initial(i,q);
  end loop;
  for i in 1..k/q loop
  min(i):=(i-1)*q+1;
  max(i):=i*q;
  end loop;
  n:=0;
   loop          --serial loop
   exit when n>nsteps;
  for r in 1..k loop
  x(r):=xstart+float(r)*h;
  end loop;
   zz:=dy(initial_value);
 --call predicted_value task
  for i in 1..k/q loop
  pv(i).send_data(initial_value,zz);
  end loop;
  for i in 1..k/q loop
  pv(i).collect(y(min(i)..max(i)));
  end loop;
  for s in 0..1 loop      --serial loop for iteration
  --call function_evaluation task
```

```
  for i in 1..k/q loop
  fv(i).send_data(y(min(i)..max(i)));
  end loop;
  for i in 1..k/q loop
  fv(i).collect(z(min(i)..max(i)));
  end loop;
 --call corrected_value task
  for i in 1..k/q loop
  cv(i).send_data(a(min(i)..max(i)),b(min(i)..max(i)),zz,
                  z(min(i)..max(i)));
  end loop;
  for i in 1..k/q loop
  cv(i).receive(y(min(i)..max(i)));
  end loop;
 end loop;   --serial itreation end loop
 for r in 1..k loop
--      true_solution(r):=exp(x(r));
 true_solution(r):=1.0/sqrt(x(r)+1.0);
  error(r):=abs(true_solution(r)-y(r));
  end loop;
 max_error:=0.0;
 for r in 1..k loop
 if max_error<error(r) then
 max_error:=error(r);
 end if;
 end loop;
  xstart:=x(k);
  initial_value:=y(k);
    n:=n+k;
 -- put("the number n is ");new_line;
 --   put(n);new_line;
  exit when n=nsteps;
  end loop;       --end serial loop
     put("the true solution is");new_line;
     put(true_solution(k));new_line;
```

```
  put(y(k));new_line;
   put(max_error);new_line;
  end;
  end;
end method_1;
```

```
--This Program solves Initial Value Problems Using Method_3
with text_io, float_io,integer_io, math;
use text_io,float_io,integer_io,math;
procedure method_3 is
k,q,nsteps : integer;
begin
get(k);
get(q);
get(nsteps);
declare
    iv,h,xx,z1,z2,z3,xnsteps,max_error:float;
    n:integer;
    type vector is array(integer range< >) of float;
    x,y,error,true_solution:vector(1..k);
    type col is array(integer range< >) of float;
    a,b,c:col(1..k);
--define the function
function dy(x,y:float) return float is
begin
--return y;
return (-y**3)/2.0;
--return (20.0*y-y**2)/80.0;
--return y*cos(x);
end dy;
procedure mult(a1,b1,c1:in col;z1,z2,z3:float;
                i,q:integer;ynew:out vector) is
begin
for r in 1..q loop
ynew(r):=iv+h*(a1(r)*z1+b1(r)*z2+c1(r)*z3);
end loop;
end mult;
task type corrected is
entry send_initial(itemp,qtemp:integer);
```

```
entry    send(atempi,btempi,ctempi:in    col;
                        value_1,value_2,value_3:in        float);
entry    collect(ytempi:out    vector);
end;
task body corrected is
     i,q:integer;
     z1,z2,z3:float;
begin
    accept    send_initial(itemp,qtemp:integer)    do
    i:=itemp;q:=qtemp;
    end;
    declare
    anew,bnew,cnew:col(1..q);
    ynew:vector(1..q);
    begin
    loop
    select
    accept    send(atempi,btempi,ctempi:in    col;
                        value_1,value_2,value_3:in    float)    do
  anew:=atempi;bnew:=btempi;cnew:=ctempi;
   z1:=value_1;z2:=value_2;z3:=value_3;
  end;
  mult(anew,bnew,cnew,z1,z2,z3,i,q,ynew);
  accept    collect(ytempi:out    vector)    do
  ytempi:=ynew;
  end;
  or
  terminate;
  end    select;
  end  loop;
  end;
  end    corrected;
  begin
  declare
  cr:array(1..k/q)    of    corrected;
```

```
   begin
   for r in 1..k loop
   --put("here is the coffec");new_line;
   a(r):=(float(r)/3.0)*(2.0+float(k))-float(r**2)/2.0;
   b(r):=float(r**2)/2.0-float(r*k)/3.0;
   c(r):=float(r)/3.0;
   end loop;
iv:=1.0;
   xx:=0.0;
   --xnsteps:=1.0;
   xnsteps:=20.0;
   h:=(xnsteps-xx)/float(nsteps);
   --put("The step size is");new_line;
   --put(h);new_line;
   for i in 1..k/q loop
   cr(i).send_initial(i,q);
   end loop;
   n:=0;
   loop
   exit when n>nsteps;
   z1:=dy(xx,iv);
   --put("the predictor y is");new_line;
   y(1):=iv+h*z1;
   --put(y(1));new_line;
   y(k):=iv+float(k)*h*z1;
   --put(y(k));new_line;
   for r in 1..k loop
   x(r):=xx+float(r)*h;
   end loop;
   for s in 0..1 loop
   --put("here the function evalution");new_line;
   z2:=dy(x(1),y(1));
   z3:=dy(x(k),y(k));
   --call corrected task
```

```
for i in 1..k/q loop
cr(i).send(a((i-1)*q+1..i*q),b((i-1)*q+1..i*q),c((i-1)*q+1..i*q),
                                                    z1,z2,z3);
end loop;
for i in 1..k/q loop
cr(i).collect(y((i-1)*q+1..i*q));
end loop;
end loop;
for r in 1..k loop
--true_solution(r):=exp(x(r));
true_solution(r):=1.0/sqrt(x(r)+1.0);
--true_solution(r):=20.0/(1.0+19.0*exp(-x(r)/4.0));
--true_solution(r):=exp(sin(x(r)));
error(r):=abs(true_solution(r)-y(r));
end loop;
max_error:=0.0;
for r in 1..k loop
if max_error<error(r) then
max_error:=error(r);
end if;
end loop;
xx:=x(k);
iv:=y(k);
n:=n+k;
exit when n=nsteps;
end loop;
put("the true solution is");new_line;
put(true_solution(k));new_line;
put(y(k));new_line;
put(max_error);new_line;
end;
end;
end method_3;
```

```
--This Program solves Initial Value Problem Using Method (4)
with text_io, float_io,integer_io, math;
use text_io,float_io,integer_io,math;
procedure method_4 is
k,q,nsteps:integer;
begin
get(k);
get(q);
get(nsteps);
declare
     n:integer;
     type vector is array(integer range< >) of float;
     x,y,z,true_solution,error:vector(1..k);
     type integer_vector is array(1..k/q) of integer;
     min,max:integer_vector;
     xstart,xend,initial_value,zz,h,max_error:float;
function dy(y:float) return float is
begin
--return   y;
--return    -y**3/2.0;
return    (20.0*y-y**2)/80.0;
--return    y*cos(x);
end dy;
procedure mult(zz:in float;z:in vector;
                  i,q:integer;ynew:out vector) is
  sum:float;
  w:vector(1..q);
begin
sum:=0.0;
   for r in 1..(i-1)*q loop
   sum:=sum+z(r);
   end loop;
   w(1):=initial_value+h*(zz/2.0+sum+z((i-1)*q+1)/2.0);
   for r in 2..q loop
   w(r):=w(r-1)  +  h*(z(r+(i-1)*q-1)/2.0+z(r+(i-1)*q)/2.0);
```

```
    end loop;
    for r in 1..q loop
    ynew(r):=w(r);
    end loop;
end mult;
--the following task evaluates the predicted_y
 task type predicted_value is
 entry      send_initial(itemp,qtemp:integer;step_size:in      float);
 entry     send_data(initial,value:in      float);
 entry    collect(ytempi:out    vector);
 end;
 task body predicted_value is
   initial_value,h,zz:float;
   i,q:integer;
 begin
 accept     send_initial(itemp,qtemp:integer;step_size:in     float)     do
 i:=itemp;q:=qtemp;h:=step_size;
 end;
   declare
      ynew,w:vector(1..q);
 begin
   loop
     select
      accept    send_data(initial,value:in    float)    do
      initial_value:=initial;zz:=value;
      end;
     zz:=h*zz;
     w(1):=initial_value+float(1+(i-1)*q)*zz;
     for r in 2..q loop
     w(r):=w(r-1)+zz;
     end loop;
     for r in 1..q loop
     ynew(r):=w(r);
     end loop;
     accept   collect(ytempi:out   vector)   do
```

```
      ytempi:=ynew;
      end;
      or
      terminate;
     end  select;
    end  loop;
    end;
end   predicted_value;
--the  following  task  computes  the  y'(n+j)
 task  type  function_evaluation  is
 entry       send_initial(itemp,qtemp:integer);
 entry   send_data(ytempi:in   vector);
 entry   collect(ztempi:out   vector);
 end;
  task  body  function_evaluation  is
  i,q:integer;
 begin
 accept   send_initial(itemp,qtemp:integer)   do
 i:=itemp;q:=qtemp;
 end;
  declare
        xnew,ynew,znew:vector(1..q);
  begin
    loop
     select
     accept  send_data(ytempi:in  vector)  do
     ynew:=ytempi;
     end;
     for  r  in  1..q  loop
     znew(r):=dy(ynew(r));
     end  loop;
    accept  collect(ztempi:out  vector)  do
    ztempi:=znew;
    end;
   or
```

```
  terminate;
  end  select;
 end  loop;
end;
end   function_evaluation;
 task  type  corrected_value  is
 entry       send_initial(itemp,qtemp:integer);
 entry   send_data(value:in   float;ztemp:in   vector);
 entry   receive(ytempi:out   vector);
 end;
 task  body  corrected_value  is
 zz:float;
 i,q:integer;
 z:vector(1..k);
 begin
      accept   send_initial(itemp,qtemp:integer)   do
      i:=itemp;q:=qtemp;
      end;
   declare
   ynew:vector(1..q);
  begin
    loop
    select
    accept  send_data(value:in  float;ztemp:in  vector)  do
    zz:=value;z:=ztemp;
    end;
    mult(zz,z,i,q,ynew);
    accept  receive(ytempi:out  vector)  do
    ytempi:=ynew;
    end;
   or
   terminate;
   end  select;
 end  loop;
 end;
```

```
   end   corrected_value;
   begin
    declare
    pv:array(1..k/q)    of    predicted_value;
    fv:array(1..k/q)    of     function_evaluation;
    cv:array(1..k/q)    of    corrected_value;
begin
   xstart:=0.0;
--xend:=1.0;
   xend:=20.0;
   h:=(xend-xstart)/float(nsteps);
--put("the    step_size    is");new_line;
--      put(h);new_line;
   initial_value:=1.0;
   for i in 1..k/q loop
   pv(i).send_initial(i,q,h);
   fv(i).send_initial(i,q);
   cv(i).send_initial(i,q);
   end loop;
   for i in 1..k/q loop
   min(i):=(i-1)*q+1;
   max(i):=i*q;
   end loop;
     n:=0;
     loop          --serial loop
       exit when n>nsteps;
       for r in 1..k loop
      x(r):=xstart+float(r)*h;
      end loop;
       zz:=dy(initial_value);
     --call   predicted_value    task
      for i in 1..k/q loop
      pv(i).send_data(initial_value,zz);
      end loop;
```

```
      for i in 1..k/q loop
      pv(i).collect(y(min(i)..max(i)));
      end loop;
      for s in 0..1 loop      --serial loop for iteration
      --call   function_evaluation   task
      for i in 1..k/q loop
--            fv(i).send_data(x(min(i)..max(i)),y(min(i)..max(i)));
      fv(i).send_data(y(min(i)..max(i)));
      end loop;
      for i in 1..k/q loop
      fv(i).collect(z(min(i)..max(i)));
      end loop;
      --call   corrected_value   task
      for i in 1..k/q loop
      cv(i).send_data(zz,z);
      end loop;
      for i in 1..k/q loop
      cv(i).receive(y(min(i)..max(i)));
      end loop;
      end loop;    --serial itreation end loop
      for r in 1..k loop
    --       true_solution(r):=exp(x(r));
--            true_solution(r):=1.0/sqrt(x(r)+1.0);
   true_solution(r):=20.0/(1.0+19.0*exp(-x(r)/4.0));
--            true_solution(r):=exp(sin(x(r)));
       error(r):=abs(true_solution(r)-y(r));
       end loop;
    max_error:=0.0;
    for r in 1..k loop
    if max_error<error(r) then
    max_error:=error(r);
    end if;
    end loop;
   xstart:=x(k);
```

```
   initial_value:=y(k);
   n:=n+k;
  -- put("the number n is ");new_line;
  --    put(n);new_line;
  exit when n=nsteps;
  end loop;        --end serial loop
  put("the true solution is");new_line;
  put(true_solution(k));new_line;
  put(y(k));new_line;
  put(max_error);new_line;
end;
end;
end method_4;
```

```
--This Program solves Initial Value Problems Using Fourth_Order
--Method
with text_io, float_io,integer_io, math;
use text_io,float_io,integer_io,math;
procedure fourth_order_block is
k,q,nsteps:integer;
begin
get(k);
get(q);
get(nsteps);
declare
    iv,h,xx,z1,z2,z3,z4,xnsteps,max_error:float;
    n:integer;
    type vector is array(integer range<>) of float;
    x,y,error,true_solution:vector(1..k);
    type col is array(integer range<>) of float;
    a,b,c,d:col(1..k);
--define the function
function dy(x,y:float) return float is
begin
--return y;
return (-y**3)/2.0;
--return (20.0*y-y**2)/80.0;
--return y*cos(x);
end dy;
procedure mult(a1,b1,c1,d1:in col;z1,z2,z3,z4:float;
                i,q:integer;ynew:out vector) is
begin
for r in 1..q loop
ynew(r):=iv+h*(a1(r)*z1+b1(r)*z2+c1(r)*z3+d1(r)*z4);
end loop;
end mult;
task type corrected is
entry send_initial(itemp,qtemp:integer);
```

```
entry    send(atempi,btempi,ctempi,dtempi:in    col;
                        value_1,value_2,value_3,value_4:in        float);
entry    collect(ytempi:out    vector);
end;
task body corrected is
     i,q:integer;
     z1,z2,z3,z4:float;
begin
    accept    send_initial(itemp,qtemp:integer)    do
    i:=itemp;q:=qtemp;
    end;
    declare
    anew,bnew,cnew,dnew:col(1..q);
    ynew:vector(1..q);
    begin
    loop
    select
    accept    send(atempi,btempi,ctempi,dtempi:in    col;
                        value_1,value_2,value_3,value_4:in    float)    do
  anew:=atempi;bnew:=btempi;cnew:=ctempi;dnew:=dtempi;
   z1:=value_1;z2:=value_2;z3:=value_3;z4:=value_4;
  end;
  mult(anew,bnew,cnew,dnew,z1,z2,z3,z4,i,q,ynew);
  accept    collect(ytempi:out    vector)    do
  ytempi:=ynew;
  end;
  or
  terminate;
  end    select;
  end  loop;
  end;
  end    corrected;
  begin
  declare
  cr:array(1..k/q)    of    corrected;
```

```
   begin
   for r in 1..k loop
   --put("here is the coffec");new_line;
   d(r):=3.0*float(r**2)-4.0*float(k*r)+6.0*float(k-1);
   d(r):=d(r)*float(r**2)/float(12*k*(k-1));
   --put(d(r));new_line;
   c(r):=float(r**3)/3.0-float(r**2)/2.0-float(k)*(float(k)-1.0)*d(r);
   c(r):=c(r)/(float(k**2)-3.0*float(k)+2.0);
   --put(c(r));new_line;
   b(r):=float(r**2)/2.0-(float(k)-1.0)*c(r)-float(k)*d(r);
   --put(b(r));new_line;
   a(r):=float(r)-b(r)-c(r)-d(r);
   --put(a(r));new_line;
   end loop;
iv:=1.0;
   xx:=0.0;
   --xnsteps:=1.0;
   xnsteps:=20.0;
   h:=(xnsteps-xx)/float(nsteps);
   --put("The step size is");new_line;
   --put(h);new_line;
   for i in 1..k/q loop
   cr(i).send_initial(i,q);
   end loop;
   n:=0;
   loop
   exit when n>nsteps;
   z1:=dy(xx,iv);
   --put("the predictor y is");new_line;
   y(1):=iv+h*z1;
   --put(y(1));new_line;
   y(k-1):=iv+float(k-1)*h*z1;
   --put(y(k-1));new_line;
   y(k):=iv+float(k)*h*z1;
   --put(y(k));new_line;
```

```
n:=n+k;
exit when n=nsteps;
end loop;
put("the true solution is");new_line;
put(true_solution(k));new_line;
put(y(k));new_line;
put(max_error);new_line;
end;
end;
end fourth_order_block;
```