Winter 1987

Implementation and Performance Analysis of Numerical Algorithms on the MPP, Flex/32, and Cray/2

Raad A. Fatoohi
Old Dominion University

IMPLEMENTATION AND PERFORMANCE ANALYSIS OF NUMERICAL ALGORITHMS ON THE MPP, FLEX/32, AND CRAY/2

by

Raad A. Fatoohi
B.S. June 1976, Mosul University
M.S. December 1982, Syracuse University

A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY
IN
ENGINEERING

OLD DOMINION UNIVERSITY
December 1987

Approved by:

John W. Stoughton (Director)

Abstract

IMPLEMENTATION AND PERFORMANCE ANALYSIS OF NUMERICAL ALGORITHMS ON THE MPP, FLEX/32, AND CRAY/2

Raad A. Fatoohi
Old Dominion University, 1987

This dissertation presents the results of the implementation of a number of numerical algorithms on three parallel/vector computers. The object of this research is to determine how well, or poorly, a number of numerical algorithms would map onto three different architectures and to analyze the performance of these architectures using these algorithms. These algorithms are: a relaxation scheme for the solution of the Cauchy-Riemann equations, an ADI method for the solution of the diffusion equation, and a compact difference scheme for the solution of two-dimensional Navier-Stokes equations. The computers were chosen so as to encompass a variety of architectures. They are: the MPP, an SIMD machine with 16K bit serial processors; Flex/32, an MIMD machine with 20 processors; and the Cray/2. The machine architectures are briefly described. The implementation of these algorithms is discussed in relation to these architectures and measures of the performance on each machine are given. The basic comparison is among SIMD instruction parallelism on the MPP, MIMD process parallelism on the Flex/32, and vectorization of a serial code on the Cray/2. Simple performance models are used to describe the performance. These models highlight the bottlenecks and limiting factors for these algorithms on these architectures. Finally conclusions are presented.

Dedication

For Rakiya and Wisam

Acknowledgements

I would like to take this opportunity to extend a special thanks to Professor Chester Grosch, without his unique offer for supervision this research would not have been possible. In addition to the technical expertise that he provided, I am grateful to Professor Grosch for his patience, encouragement, and expeditious feedback.

I would also like to thank the other members of my committee, Dr. John Stoughton, Dr. Stephen Zahorian, Dr. Sharad Kanetkar, and Dr. Robert Voigt for their help and advice.

I would like to express my appreciation to NASA Langley Research Center, NASA Goddard Space Flight Center, and NASA Ames Research Center for allowing me access to the Flex/32, MPP, and Cray/2. I am also grateful to Tom Crockett of ICASE, Tor Opsahl and David Wildenhain of Science Applications Research, and Dan Nagle of Cray Research, Inc., for their help in using these machines.

Finally, I would like to thank Dr. Roland Mielke, Chairman of the Electrical and Computer Engineering Department, and Dr. Robert Voigt, Director of ICASE, for their valuable support during my Ph.D. study. This research was supported by the National Aeronautics and Space Administration under NASA Contracts No. NAS1-17070 and NAS1-18107 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

Table of Contents

	Page
List of Tables	vi
List of Figures	viii
List of Symbols	ix
Chapter	
1. Introduction	1
2. The Architectures and Programming Languages	5
2.1. The MPP architecture and MPP Pascal	5
2.2. The Flex/32 architecture and Concurrent Fortran	8
2.3. The Cray/2 architecture and CFT/2 compiler	11
3. Performance Models	13
3.1. The MPP	13
3.2. The Flex/32	15
3.3. The Cray/2	17
4. Solving the Cauchy-Riemann Equations	19
4.1. The numerical method	19
4.2. Implementation on the MPP	25
4.3. Implementation on the Flex/32	31
4.4. Implementation on the Cray/2	34
5. Solving the Diffusion Equation	39
5.1. The numerical method	39
5.2. Implementation on the MPP	44
5.3. Implementation on the Flex/32	47
5.4. Implementation on the Cray/2	50

6. Solving Navier-Stokes Equations 54

6.1. The numerical method 54
6.2. Implementation on the MPP 58
6.3. Implementation on the Flex/32 61
6.4. Implementation on the Cray/2 64

7. Comparisons and Concluding Remarks 67

List of References 73

List of Tables

Table		Page
1	Measured execution times (in machine cycles) of the elementary operations on the MPP.	15
2	Measured execution times for one iteration and processing rates for the relaxation algorithm on the MPP.	29
3	Estimated times (in milliseconds) of the relaxation algorithm on the MPP.	30
4	Speedup and efficiency of the relaxation algorithm on the Flex/32 using the MMOS locks.	33
5	Speedup and efficiency of the relaxation algorithm on the Flex/32 using the Local locks.	33
6	Measured execution times for one iteration and processing rates for the relaxation algorithm using 16 processors of the Flex/32.	33
7	Measured execution times for one iteration and processing rates for the relaxation algorithm on one processor of the Cray/2.	36
8	Estimated and measured execution times (in milliseconds) of the relaxation algorithm on the Cray/2.	38
9	Measured execution time per time step and processing rate for the ADI procedure on the MPP.	45
10	Operation counts for one pass of the ADI method using the cyclic elimination algorithm for solving the tridiagonal systems.	46
11	Estimated and measured times (in milliseconds) of the ADI method on the MPP.	46
12	Speedup and efficiency of the ADI method on the Flex/32.	48

13	Measured execution times per time step and processing rates for the ADI method using 16 processors of the Flex/32.	48
14	Estimated and measured times (in milliseconds) of the ADI method on the Flex/32.	49
15	Measured execution times per time step and processing rates for the ADI method on one processor of the Cray/2.	51
16	Operation counts for one pass of the ADI method using the Gaussian elimination algorithm for solving the tridiagonal systems.	52
17	Estimated and measured execution times (in milliseconds) of the ADI method on one processor of the Cray/2.	53
18	Measured execution time and processing rate of the Navier-Stokes subprograms on the MPP.	60
19	Estimated times (in milliseconds) of the Navier-Stokes subprograms on the MPP.	61
20	Speedup and efficiency of the Navier-Stokes algorithm on the Flex/32.	63
21	Measured execution times for ten time steps and processing rates for the Navier-Stokes algorithm using 16 processors of the Flex/32.	63
22	Measured execution time and processing rate of the Navier-Stokes subprograms on the Cray/2.	65
23	Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 64×64 problem on one processor of the Cray/2.	66
24	Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 128×128 problem on one processor of the Cray/2.	66
25	Summary of the results for the 128×128 problem.	68
26	Clock cycles (in million cycles) for the 128×128 problem.	70

List of Figures

Figure		Page
1	Block diagram of the MPP.	6
2	Block diagram of the Flex/32 architectures.	9
3	Typical computational cell and the data associated with it.	21
4	Computational domain and the four color cells.	24
5	Spectral radius as a function of the acceleration parameter.	26

List of Symbols

C_a Number of cycles in add operation on the MPP (section 3.1),

C_m Number of cycles in multiply operation on the MPP (section 3.1),

C_d Number of cycles in divide operation on the MPP (section 3.1),

C_{sh} Startup cost of shift operation on the MPP (section 3.1),

C_{st} Number of cycles for each step in shift operation on the MPP (section 3.1),

$C_{i,j}$ Matrix of coefficients (section 4.1),

CP Clock Period of the Cray/2 (section 3.3),

$f_{bc}(p)$ Busses contention factor on the Flex/32 (section 3.2),

f_{ld} Load distribution factor of an algorithm on the Flex/32 (section 3.2),

$F_{i,j}$ Forcing term of tridiagonal equations (section 5.1),

$G_{i,j}$ Forcing term of tridiagonal equations (section 5.1),

k_{cld} Number of times a vector is stored in common memory on the Flex/32 (section 3.2),

k_{clm} Number of times a vector is copied to local memory on the Flex/32 (section 3.2),

k_{cma} Number of times a shared vector is referenced on the Flex/32 (section 3.2),

k_{lck} Number of times a shared variable is locked on the Flex/32 (section 3.2),

L_f Length of a floating point functional unit of the Cray/2 (section 3.3),

L_m Length of data path between main memory and registers on the Cray/2 (section 3.3),

L_{vcr} Length of the vector (section 3.3),

n Number of grid points in each dimension of the computational domain (section 3.2),

N_a Number of additions of an algorithm on the MPP (section 3.1),

N_d Number of divisions of an algorithm on the MPP (section 3.1),

N_{f1} Number of floating point operations with stride of 1 on the Cray/2 (section 3.3),

N_{f2} Number of floating point operations with stride of 2 on the Cray/2 (section 3.3),

N_m Number of multiplications of an algorithm on the MPP (section 3.1),

N_{m1} Number of memory access operations with stride of 1 on the Cray/2 (section 3.3),

N_{m2} Number of memory access operations with stride of 2 on the Cray/2 (section 3.3),

N_{sh} Number of shift operations of an algorithm on the MPP (section 3.1),

N_{st} Number of steps in shift operations of an algorithm on the MPP (section 3.1),

p Number of processors (section 3.2),

$\vec{P}$ Box variables (section 4.1),

R_1 Data transfer rate with stride of 1 through main memory on the Cray/2 (section 3.3),

R_2 Data transfer rate with stride of 2 through main memory on the Cray/2 (section 3.3),

$R^{(k)}$ Residual after k'th iteration (section 4.1),

Re Reynolds number (section 6.1),

t_c Machine cycle time for the MPP (section 3.1),

t_{cma} Time to access an element of shared vector on the Flex/32 (section 3.2),

t_{lck} Total time to lock and unlock a shared variable on the Flex/32 (section 3.2),

t_{lma} Time to access an element of vector in local memory of the Flex/32 (section 3.2),

t_{spn} Time to spawn one process on the Flex/32 (section 3.2),

T_{cld} Time difference between common and local memories on the Flex/32 (section 3.2),

T_{clm} Total time to copy shared variables to local memory on the Flex/32 (section 3.2),

T_{cma} Total common memory access time of an algorithm on the Flex/32 (section 3.2),

T_{cmm} Communication cost of an algorithm on the MPP (section 3.1),

T_{cmo} Total common memory overhead time of an algorithm on the Flex/32 (section 3.2),

T_{cmp} Computation cost of an algorithm (section 3.1),

T_{f1} Time to do floating point operations with stride of 1 on the Cray/2 (section 3.3),

T_{f2} Time to do floating point operations with stride of 2 on the Cray/2 (section 3.3),

T_{m1} Time to do memory access operations with stride of 1 on the Cray/2 (section 3.3),

T_{m2} Time to do memory access operations with stride of 2 on the Cray/2 (section 3.3),

T_{ovr} Overhead time of an algorithm on the Flex/32 (section 3.2),

T_p Execution time of an algorithm on the Flex/32 (section 3.2),

T_{spn} Spawning time of p processes on the Flex/32 (section 3.2),

T_{syn} Total synchronization time of an algorithm on the Flex/32 (section 3.2),

$\vec{u}$ Velocity field (section 4.1),

$\vec{U}_{i,j}$ Velocity field in discrete approximation notation (section 4.1),

$Z_{i,j}$ Matrix of coefficients (section 4.1),

α Coefficient of tridiagonal equations (section 5.1),

β Coefficient of tridiagonal equations (section 5.1),

γ Coefficient of tridiagonal equations (section 6.1),

δ_x Centered difference operator (section 4.1),

δ_x^2 Centered second difference operator (section 5.1),

Δt Full time step (section 5.1),

ζ Vorticity (section 4.1),

$\lambda_{i,j}$ Matrix of coefficients (section 4.1),

μ_x Average operator (section 4.1),

σ Spectral radius (section 4.1),

ω Acceleration parameter (section 4.1),

$\lceil x \rceil$ Next integer greater than or equal to x,

CHAPTER ONE

Introduction

The objective of the research reported here is to determine how well, or poorly, a number of numerical algorithms would map onto three different parallel architectures and to analyze the performance of these architectures using these algorithms.

It appears that single processor computers, whether scalar or vector, are nearing the ultimate limit of their performance. Certainly, the circuit clock period will decrease and circuit density will increase in the future, but it appears unlikely that major and rapid gains are in prospect. The latest supercomputer, Cray/2, has a clock period of 4.1 nanoseconds, and the Cray/1, introduced about 10 years ago, had a clock period of 12.5 nanoseconds. Thus there has been only a factor of 3 increase in clock speed in the last decade. A reduction of the clock period to one nanosecond seems possible soon. This development, while increasing the processing rate, will impose rather stringent constraints on the packaging density and architecture of a single processor computer. An alternative way of achieving greater processing power is to use computers consisting of multiple processors.

There are a number of important, unresolved questions concerning multiprocessor computers. Among these issues are: should they consist of a few, rather powerful processors or many, simple processors, or something in between? Should the new computers be SIMD (Single Instruction Multiple Data) or MIMD (Multiple Instruction Multiple Data)? There is a natural expectation that the multiprocessors with a few, powerful processors will have an MIMD architecture and that the others will have SIMD architectures. Another

issue is the communication among the processors; how the memory is connected to the processors and how these processors are connected to each other. Should the interconnection scheme be a lattice, a bus, a switch, or something else? It is sterile at this point to argue what is the "best" combination of number of processors and power per processor and what is the "best" interconnection scheme. Rather, carrying out experiments with existing multiprocessor computers would appear to be of greater value.

If the parallel computers can be used effectively, very large gains in overall processing power are possible. There are three conditions which must be met if an algorithm is to execute at high efficiency on a multiprocessor computer: (1) it must have many operations which are executable in parallel, (2) the amount of communication required between the processors must be small compared to the amount of calculations which are required, and (3) each processor must have roughly the same amount of work to do. High performance will actually be obtained from parallel architectures when the algorithms executed map efficiently to the architecture. Efficient mapping must be based on a thorough and detailed understanding of the resource requirements of the algorithms and the ability of a given architecture to deliver these resources. Mappings of algorithms onto parallel architectures is a problem of extensive dimensionality and great complexity.

Despite the fact that the most powerful existing parallel/vector computers can perform at peak rates of several hundred MFLOPS (million floating point operations per second), the average processing rates of many codes are in the range of 20 to 30 MFLOPS [9]. In part, this can be explained by invoking Amdahl's law [2],

$$S = \frac{1}{1 + f(R^{-1} - 1)},$$

where S is the speedup, f is the fraction of the code that can be parallelized/vectorized, and R is the parallel/vector to scalar speed ratio. Calculations with this formula show that

nearly all the code must be parallelized/vectorized in order to achieve a substantial speedup. It is also clear that increasing R will only give a very modest improvement for a fixed f.

The problem of measuring the performance of parallel computers is a difficult one and, as yet, does not have a solid theoretical foundation [22]. Performance is highly dependent on the architecture of the multiprocessor, the computational algorithm, and the software environment, particularly the programming language used. One abstract approach would be to use a model of the concurrent processor and analyze the execution of a particular algorithm or class of algorithms [12], [16]. The degree of abstraction of the model and the depth to which the algorithm is analyzed might very well influence the results. Another approach is to calculate an upper bound on performance by considering the time to perform a single arithmetic operation, together with the number of processors. Such a measure is widely held to be unrealistic because it does not include any of the omnipresent overhead.

A different approach is to program in some language and run a specific algorithm or class of algorithms on a particular multiprocessor computer. Despite the fact that this would be a very specific experiment, this approach has some distinct advantages. Measurement of the performance gives an objective measure of the cost of computation, although for a specific class of algorithms, expressed in a specific language and executed on a specific architecture. This kind of experiment also can yield subjective evidence as to how well the class of algorithms fits the architecture, how difficult it was to program in the particular language, and so on. This approach has been considered by Grosch [17] in adapting a Navier-Stokes code to the ICL-DAP (an SIMD machine with 4096 one-bit processors) and it will be adapted in this work.

There have been a substantial number of theoretical studies of the performance of algorithms on parallel computers but far fewer actual experimental studies [19], [21]. Until

recently, most of the experimental studies had been concentrated on the use of the Cyber 200 and Cray series of computers. Beside the works by Gallopoulos [11] on the MPP and Bokhari [6] and Crockett [8] on the Flex/32, there have been a few actual studies in using the machines chosen for this research.

In this research we describe the implementation of three numerical algorithms on three different parallel architectures and analyze the performance of these architectures using these algorithms. These algorithms are: a relaxation scheme for the solution of the Cauchy-Riemann equations, an ADI method for the solution of the diffusion equations, and a compact difference scheme for the solution of the incompressible, two-dimensional, time-dependent Navier-Stokes equations. Both the relaxation scheme and the ADI method are used in the solution of the Navier-Stokes equations. These algorithms are described in Gatski et al. [13] and Grosch [16]. The architectures chosen for this study are: the MPP, an SIMD machine with 16K serial one-bit processors; the Flex/32, an MIMD machine with 20 processors based on 32-bit NSC 32032 microprocessor; and Cray/2, an MIMD machine with four powerful vector processors. The basic comparison is between SIMD instruction parallelism on the MPP, MIMD process parallelism on the Flex/32, and vectorization of a serial code on the Cray/2.

The basic features of the three architectures and the programming languages used are described in Chapter two. Chapter three presents general performance models of these architectures. The implementation of the three algorithms on these architectures is described in Chapters four through six; each chapter contains a brief description of the algorithm and the implementation and performance analysis of that algorithm on each machine. Finally, Chapter seven contains a comparison of performance of these machines at the problem solving level and some concluding remarks.

CHAPTER TWO

The Architectures and Programming Languages

A brief description of the three architectures and the programming languages used is given in this chapter in order to clarify the way in which the three algorithms, described in the Chapters four through six, were adapted to these architectures. Both the hardware and software of each computer are described in order that one can appreciate some of the most important features of each machine that contribute to its limitations and advantages.

2.1. The MPP architecture and MPP Pascal

The Massively Parallel Processor (MPP) is a large-scale SIMD processor developed by Goodyear Aerospace Co. for NASA Goddard Space Flight Center [5], [15]. The MPP is a back-end processor for a VAX-11/780 host, which supports its program development and I/O needs.

The block diagram of the hardware elements of the MPP is shown in Fig. 1. The Array Unit (ARU) consists of a square array of 128×128 bit-serial Processing Elements (PE's). Each PE has a local 1024 bit random access memory and is connected to its four nearest neighbors with programmable edge connections. Arithmetic in each PE is performed in bit serial fashion using a serial-by-bit adder. The PE also contains a shift register which is used in multiplication and division. The ARU is controlled by the Array Control Unit (ACU). The ACU supervises the PE array processing, performs scalar arithmetic, and shifts data across the PE array. Items of data are sent to the ARU and taken from the ARU

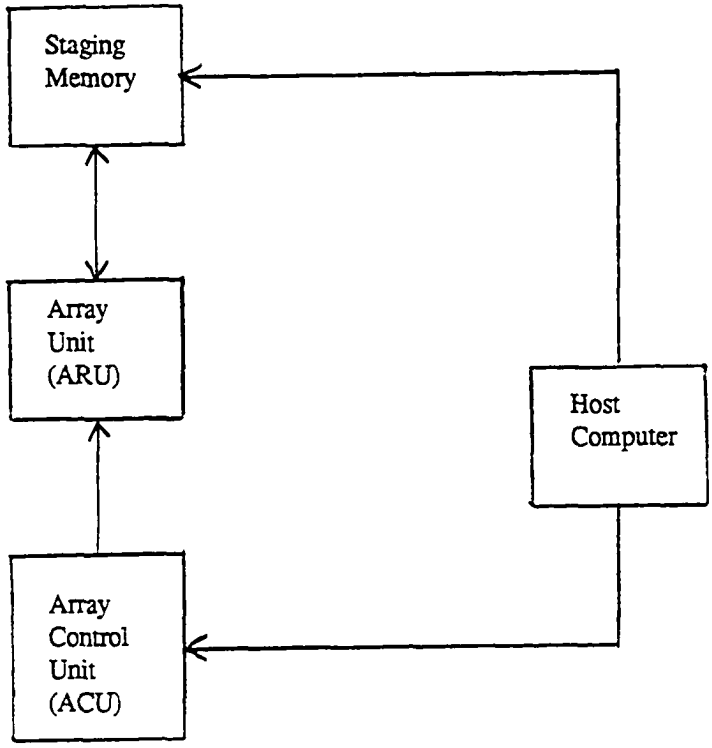

Fig. 1. Block diagram of the MPP.

through the staging memory. The staging memory can buffer arrays of data transmitted over this path and it can also reformat them. The MPP has a cycle time of 100 nsec.

With 16,384 PE's operating in parallel, the array has very high processing speed. Despite the bit-slice nature of each PE, the floating-point speeds compare favorably with other high performance machines. For example, the processing rate for the addition of two 128×128 arrays, each consisting of 32 bit floating-point numbers, is 430 MFLOPS and for multiplication it is 216 MFLOPS.

Three programming languages are currently implemented on the MPP. They are MPP Pascal and two assembly languages, Main Control Language (MCL) and PE Array Language (PEARL). MPP Pascal [14] is a machine-dependent language which has evolved directly from the language Parallel Pascal defined by Reeves [24]. Parallel Pascal is an extended version of the conventional serial Pascal programming language with a convenient syntax for specifying array operations.

MPP Pascal provides a new intrinsic type of data structure termed a parallel array. This type directs the compiler to store the array in the array memory. The last two dimensions of a parallel array must be 128×128. In addition to arithmetic operations and function evaluations on parallel arrays, MPP Pascal provides two fundamental classes of operations on array data which are not available in conventional programming languages. The first class provides operations which reduce a parallel array to a scalar. These arithmetic reduction functions are: maximum, minimum, sum, and product. The second class provides operations which permute a parallel array. The permutation functions are: shift (end off shift), rotate (end around shift), and transpose. The extensions also provide a single parallel control statement, the where-do-otherwise statement. It is similar to an if-then-else statement but with an array control variable. MPP Pascal also includes two system-defined

arrays, row-index and col-index, that give each PE its location in the array. Their major use is in masking out a particular set of PE's for a given operation. MPP Pascal programs can execute on the host, on the MPP, or on a combination of both machines. Through the use of compiler switches, the programmer specifies, at the procedure level, the system on which the code will execute.

MPP Pascal's I/O system consists of several different modules that handle each of the I/O communication links on the MPP/VAX system. There are two techniques for controlling data transfer to and from the array memory through the staging memory. These are virtual channel I/O and bit-plane I/O. In virtual channel I/O, data exist in the stager in an "unknown" address; retrieving data from the stager depends on knowing how it was stored. Virtual channel I/O software views the staging memory as a permuting channel through which data move and are reformated. Bit-plane I/O treats the stager as a memory not as a permuting channel. Bit-plane I/O allows users to access data in the stager by variable name, simply by specifying a bit plane address. For this reason, the stager is configured to look like the array memory, i.e., a 16K array of a 128×128 bit planes (the staging memory size is 32 Mbytes).

2.2. The Flex/32 architecture and Concurrent Fortran

The Flex/32 is an MIMD shared memory multiprocessor based on 32 bit National Semiconductor 32032 processor [10]. The Flex/32 cabinet can hold up to 20 of any combination of processor and memory cards. The results presented in this research were obtained using the 20 processor machine that is now installed at NASA Langley Research Center.

As shown in Fig. 2, there are ten local buses; each connects two processors. These local buses are connected together and to the common memory by a common bus. The 2.25 Mbytes of the common memory is accessible to all processors. Each of processors 1,

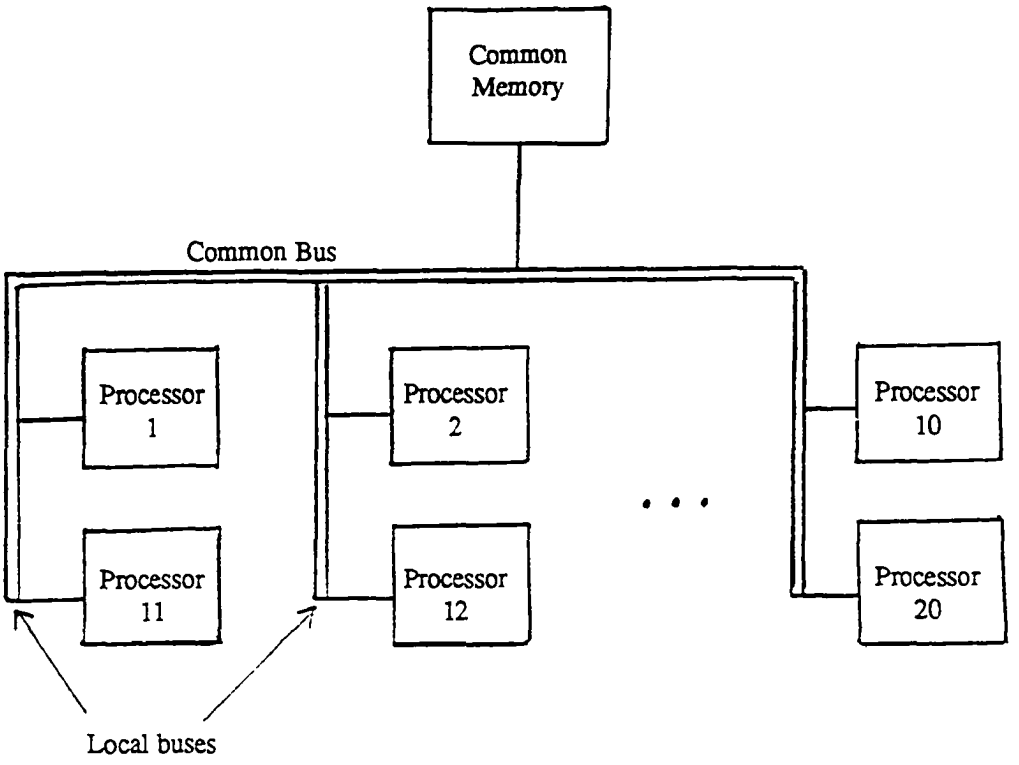

Fig. 2. Block diagram of the Flex/32 architecture.

2, and 3 contains 4 Mbytes of local memory. All other processors contain 1 Mbyte each. Each processor has a cycle time of 100 nsec.

The UNIX operating system is resident in processors 1 and 2. These processors are also used for software development and for loading and booting the other processors. Processors 3 through 20 run the Multicomputing Multitasking Operating System (MMOS) and are available for parallel processing.

The Flex/32 software provides two methods of synchronizing communications between processes on separate processors. The first method is via the common memory; 8192 locks are provided to lock variables in the common memory. The second method is a message sending technique; processes can communicate by sending and receiving messages.

The Flex/32 system software has special concurrent versions of C and Fortran 77. Concurrent C and Concurrent Fortran are extensions to C and Fortran 77 programming languages with all the standard definitions and features of the languages preserved. Both introduce new constructs for implementing parallel programs. Among these constructs are:

lock — a shared variable can be locked if it is not locked by any other process. The locking process will then be able to access that variable while other processes attempting to lock it or access it will wait until the lock is released.

process — define and start the execution of a code segment on a specified processor.

shared — variables defined as shared are common data items located in the common memory and are used by several processes and/or processors.

unlock — release a locked variable.

2.3. The Cray/2 architecture and CFT/2 compiler

The Cray/2 is an MIMD supercomputer with four Central Processing Units (CPU), a foreground processor which controls I/O, and a central memory. The central memory has 256 million 64 bit words organized in four quadrants of 32 banks each. Each CPU has access to one quadrant during each clock cycle. Each CPU has an internal structure very similar to the Cray/1, see [19], with the addition of 16K words of local memory available for storage of vector and scalar data. Within each CPU there are eight vector registers (64 words each), eight scalar registers, special purpose registers (vector length and vector mask) and nine pipelined functional units, four of which support vector processing. The clock cycle is 4.1 nanoseconds.

The Cray/2 runs the UNICOS operating system which is based on UNIX system V. The four processors can operate independently on separate jobs, multiprogramming, or concurrently on a single job, multitasking.

The Cray/2 Fortran compiler (CFT/2) [7] attempts to vectorize the innermost DO loops. This is the only place where vectorization is attempted. This process is automatic, but certain loops can not be vectorized and programmer intervention is frequently required. Among the conditions preventing vectorization are I/O, CALL, IF, and GOTO statements; dependency involving an array; and ambiguous index expressions in the innermost DO loop. By default, the compiler generates 'safe' code; it assumes the worst about ambiguous situations. Some of these situations can be resolved by inserting compiler directives, using system libraries, or rewriting a program segment.

A vector operation in Cray/2 is performed by loading a group of up to 64 elements into a vector register and moving it, one element per clock period, to the functional unit performing the operation. Once it is loaded in the vector register, none of the elements can

be changed; all the elements are treated the same.

All Crays interleave words in memory so that consecutive elements of an array are stored in consecutive banks in memory. The bank cycle for the Cray/2 is 57 clock periods, i.e., accessing any bank in memory creates a 'bank busy' condition for that bank for 57 cycles. This problem is called 'memory bank conflict'. In addition to the bank conflict, array accesses with even-numbered strides (stride means the memory increment between successive elements stored or fetched) will suffer quadrant delays, which are a consequence of the four CPU's of the Cray/2 taking turns accessing the four quadrants of memory. Even-numbered strides that are not divisible by four will result in more than 50% slowdown in data transfer rate and strides that are divisible by four will result in more than 75% slowdown [4].

CHAPTER THREE

Performance Models

In this chapter, general performance models for each of the three computers, the MPP, Flex/32, and Cray/2, are presented. These models reflect the architecture of the computers, as described in Chapter two. In subsequent chapters these models are applied to the performance of a number of algorithms on the three computers.

3.1. The MPP

The following model is based on counting the number of arithmetic and data transfer operations of an algorithm and multiplying these numbers by the measured cost of each operation in order to estimate the execution time of that algorithm. Only operations on parallel arrays are considered here.

The execution time of an algorithm on the MPP, T, can be modeled as follows:

$$T = T_{cmp} + T_{cmm}, \tag{3.1}$$

$$T_{cmp} = t_c \, (N_a \, C_a + N_m \, C_m + N_d \, C_d), \tag{3.2}$$

$$T_{cmm} = t_c \, (N_{sh} \, C_{sh} + N_{st} \, C_{st}), \tag{3.3}$$

where

T_{cmp} Computation cost,

T_{cmm} Communication cost,

t_c Machine cycle time = 100 nanoseconds,

N_a Number of additions,

N_m Number of multiplications,

N_d Number of divisions,

N_{sh} Number of shift operations,

N_{st} Number of steps involved in all shift operations,

C_a Number of cycles to add two arrays of 32 bit floating point numbers,

C_m Number of cycles to multiply two arrays of 32 bit floating point numbers,

C_d Number of cycles to divide two arrays of 32 bit floating point numbers,

C_{sh} Startup cost (in cycles) of shifting an array of 32 bit floating point numbers,

C_{st} Number of cycles to perform a one step shift within a shift operation.

Table 1 contains the estimated values of C_a, C_m, C_{sh}, and C_{st} for two sets of primitives, IBM format and VAX format. The IBM format primitives are provided by the Goodyear Aerospace Co. and used mainly in the assembly language programs. The IBM format primitives were unavailable for MPP Pascal programs at the time this research was conducted. The peak performance rates of the arithmetic operations are computed based on the IBM format primitives. The VAX format primitives called by the MPP Pascal compiler version 2 were in use until July 1987. The ones called by the MPP Pascal compiler version 3 are currently used and supposed to be improved. Both of these VAX format primitives were programmed at NASA Goddard. The values corresponding to the VAX format were obtained using a simple test problem; the execution time of each operation was measured by using a loop of length 1000. Note that the VAX format primitives take considerably longer than the IBM format primitives to perform an operation. The ratio of execution times

ranges from 1.07 for multiplication to 2.53 for addition.

Table 1. Measured execution times (in machine cycles) of the elementary operations on the MPP.

Operation	IBM format primitives	VAX format primitives	
		version 2 compiler	version 3 compiler
addition	381	824	965
multiplication	758	877	811
division	1031	1130	1225
one step shift	96	166	168
k step shift	64 + 32 k	134 + 32 k	136 + 32 k

3.2. The Flex/32

The following model is based on estimating the values of various overheads resulting from running an algorithm on more than one processor. Also, the time to do the real computation on each processor is estimated.

The execution time of an algorithm on p processors of the Flex/32, T_p, can be modeled as follows:

$$T_p = T_{cmp} + T_{ovr}, \tag{3.4}$$

where T_{cmp} is the computation time and T_{ovr} is the overhead time. Let f_{ld} be a load distribution factor where $f_{ld} = 1$ if the load is distributed evenly between the processors and $f_{ld} > 1$ if at least one processor has less work to do than the other processors. Then the computation time on p processors can be computed by

$$T_{cmp} = f_{ld}\, T_1 / p, \tag{3.5}$$

where T_1 is the computation time using a single processor.

The overhead time can be modeled by:

$$T_{ovr} = T_{spn} + T_{cmo} + T_{syn}, \tag{3.6}$$

where

T_{spn} Spawning time of p processes,

T_{cmo} Total common memory overhead time,

T_{syn} Total synchronization time.

These times can be estimated as follows:

$$T_{spn} = p\ t_{spn}, \tag{3.7}$$

$$T_{syn} = p\ k_{lck}\ t_{lck}, \tag{3.8}$$

$$T_{cmo} = T_{cma} + T_{clm} + T_{cld}, \tag{3.9}$$

$$T_{cma} = n\ k_{cma}\ f_{bc}(p)\ t_{cma}, \tag{3.10}$$

$$T_{clm} = n\ k_{clm}\ (f_{bc}(p)\ t_{cma} + t_{lma}), \tag{3.11}$$

$$T_{cld} = n\ k_{cld}\ (f_{bc}(p)\ t_{cma} - t_{lma}), \tag{3.12}$$

where

n Length of the vector,

T_{cma} Total common memory access time,

T_{clm} Total time required for copying shared vectors to local memory,

T_{cld} Total time difference between storing vectors in common and local memories; i.e., Overhead time of storing vectors in common memory instead of local memory,

t_{spn} Time to spawn one process; a reasonable value is 13 msec,

t_{lck} Total time to lock and unlock a shared variable; a reasonable value is 47 μsec,

t_{cma} Time to access an element of a vector in common memory; a reasonable value is 6 μsec,

t_{lma} Time to access an element of a vector in local memory; a reasonable value is 5 μsec,

k_{lck} Number of times a shared variable is locked and unlocked for each process.

k_{cma} Number of times a shared vector is referenced,

k_{clm} Number of times a shared vector is copied to local memory,

k_{cld} Number of times a vector is stored in common memory instead of local memory,

$f_{bc}(p)$ Busses contention factor. This contention results from having more than one processor trying to access the common memory at the same time; it is a function of p.

The values of t_{spn}, t_{lck}, t_{cma}, and t_{lma} are estimated based on timing experiments performed by Bokhari [6] and Crockett [8]. It is assumed that all common memory access operations are performed on vectors of length n.

3.3. The Cray/2

In order to estimate the cost of arithmetic and memory access operations on the Cray/2, the following timing values are used:

Clock Period (CP) = 4.1 nanoseconds,

Length of data path between the main memory and the registers, L_m = 56 CPs,

Length of each floating point functional unit, L_f = 23 CPs,

Data transfer rate with stride of 1 through main memory, R_1 = 1 CP/word,

Data transfer rate with stride of 2 through main memory, R_2 = 2 CPs/word.

A lower bound on the values of R_1 and R_2 is assumed here. Competition for memory banks from other processors causes a lower transfer rate and hence increased values of R_1 and R_2. The actual values are difficult to estimate.

Based on the fact that Cray vector operations are "stripmined" in sections of 64 elements, the time required to perform arithmetic and memory access operations on vectors of length L_{vcr} can be modeled as follows:

$$T_{f1} = (\left\lceil \frac{L_{vcr}}{64} \right\rceil L_f + L_{vcr})\, N_{f1}\, CP, \tag{3.13}$$

$$T_{f2} = (\left\lceil \frac{L_{vcr}}{128} \right\rceil L_f + \frac{L_{vcr}}{2})\, N_{f2}\, CP, \tag{3.14}$$

$$T_{m1} = (\left\lceil \frac{L_{vcr}}{64} \right\rceil L_m + R_1\, L_{vcr})\, N_{m1}\, CP, \tag{3.15}$$

$$T_{m2} = (\left\lceil \frac{L_{vcr}}{64} \right\rceil L_m + R_2\, \frac{L_{vcr}}{2})\, N_{m2}\, CP, \tag{3.16}$$

where

$\lceil x \rceil$ Next integer greater than or equal to x,

T_{f1} Time to perform floating point operations on vectors with stride of 1,

T_{f2} Time to perform floating point operations on vectors with stride of 2,

T_{m1} Time to perform main memory access operations on vectors with stride of 1,

T_{m2} Time to perform main memory access operations on vectors with stride of 2,

N_{f1} Number of floating point operations on vectors with stride of 1,

N_{f2} Number of floating point operations on vectors with stride of 2,

N_{m1} Number of main memory access operations on vectors with stride of 1,

N_{m2} Number of main memory access operations on vectors with stride of 2.

The divide operation on the Cray/2 is performed by using the reciprocal approximation look-up table and the floating point vector multiply unit (Newton approximation method). It is assumed in this model that each division takes three times the multiplication time.

CHAPTER FOUR

Solving the Cauchy-Riemann Equations

This chapter is the first of three chapters dealing with implementing numerical algorithms on the three architectures. The algorithm considered in this chapter is a relaxation scheme for the solution of the Cauchy-Riemann equations. The numerical method and the adaptation of the algorithm to parallel computers are described in section 4.1. The implementation of the algorithm on the MPP, Flex/32, and Cray/2 is described in sections 4.2 through 4.4; each section contains details of the implementation, the results, and the application of the performance model of the machine, developed in Chapter three, to this algorithm.

4.1. The numerical method

Consider the following differential equations, the Cauchy-Riemann equations:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \tag{4.1}$$

$$\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = \zeta. \tag{4.2}$$

The numerical method used to approximate these equations is based on compact differencing schemes developed by Rose [25] and Philips and Rose [23]. The method is briefly described here for the sake of completeness.

Consider approximating the solution of equations (4.1) and (4.2) in a rectangular domain on whose boundary one component of $\vec{u}$, where $\vec{u} = (u, v)$, is prescribed. Subdivide

the domain into rectangular cells. This array of cells can be nonuniform. A typical cell and the location of the variables on that cell are shown in Fig. 3. A variable associated with the side of a cell is to be interpreted as the average of that variable over the side of the cell and one associated with the center of a cell is an average over the cell. The centered difference and average operators are defined on a cell by:

$$\delta_x U_{i,j} \equiv \frac{(U_{i+1/2,j} - U_{i-1/2,j})}{\Delta x}, \tag{4.3}$$

$$\mu_x U_{i,j} \equiv \frac{(U_{i+1/2,j} + U_{i-1/2,j})}{2}. \tag{4.4}$$

Suppose that $\zeta_{i+1/2,j+1/2}$ is prescribed. Then equations (4.1) to (4.2) are approximated by,

$$\delta_x U_{i+1/2,j+1/2} + \delta_y V_{i+1/2,j+1/2} = 0, \tag{4.5}$$

$$\delta_x V_{i+1/2,j+1/2} - \delta_y U_{i+1/2,j+1/2} = \zeta_{i+1/2,j+1/2}, \tag{4.6}$$

$$\mu_x U_{i+1/2,j+1/2} - \mu_y U_{i+1/2,j+1/2} = 0, \tag{4.7}$$

$$\mu_x V_{i+1/2,j+1/2} - \mu_y V_{i+1/2,j+1/2} = 0. \tag{4.8}$$

The adaptation of this algorithm to different parallel architectures can be simplified by the introduction of box variables to represent $\vec{U}$. The center of a cell is at $(i+1/2,j+1/2)$. The box variables, $\vec{P}$, are defined at the corners of the cells, as shown in Fig. 3. They are related to $\vec{U}$ by:

$$\vec{U}_{i,j+1/2} = \frac{(\vec{P}_{i,j+1} + \vec{P}_{i,j})}{2}, \tag{4.9}$$

$$\vec{U}_{i+1/2,j} = \frac{(\vec{P}_{i+1,j} + \vec{P}_{i,j})}{2}, \tag{4.10}$$

and similarly for $\vec{U}_{i+1,j+1/2}$ and $\vec{U}_{i+1/2,j+1}$.

It is easy to see that equations (4.7) and (4.8) are satisfied identically for any set of box variables. For the cell $(i+1/2,j+1/2)$, equations (4.5) and (4.6) become,

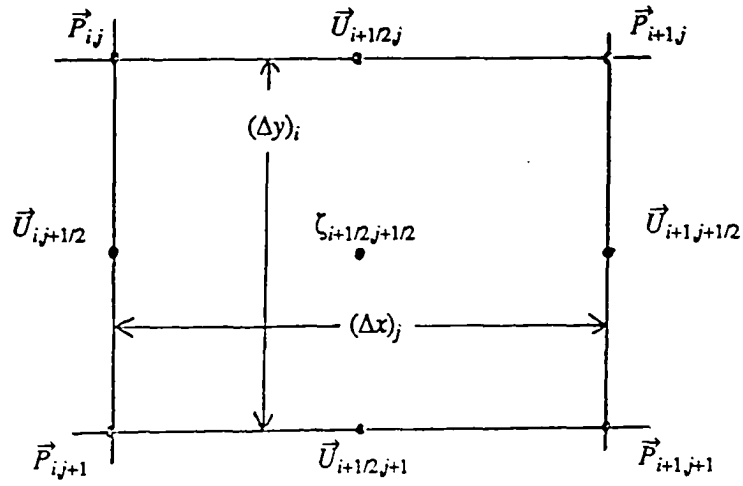

Fig. 3. Typical computational cell and the data associated with it.

$$AP = Z, \tag{4.11}$$

where

$$A = \begin{bmatrix} -\lambda_{i,j} & -1 & \lambda_{i,j} & -1 & \lambda_{i,j} & 1 & -\lambda_{i,j} & 1 \\ 1 & -\lambda_{i,j} & 1 & \lambda_{i,j} & -1 & \lambda_{i,j} & -1 & -\lambda_{i,j} \end{bmatrix},$$

$$P = (\vec{P}_{i,j} , \vec{P}_{i+1,j} , \vec{P}_{i+1,j+1} , \vec{P}_{i,j+1})^T,$$

$$Z = (0 , Z_{i,j})^T,$$

$$\vec{P}_{i,j} = (P_{i,j} , Q_{i,j}),$$

$$Z_{i,j} = 2(\Delta y)_i \zeta_{i+1/2,j+1/2},$$

$$\lambda_{i,j} \equiv \frac{(\Delta y)_i}{(\Delta x)_j}.$$

Equation (4.11) is solved by an iteration scheme which was originally proposed by Kaczmarz [20] and was generalized by Tanabe [26]. If $P^{(k)}$ is the value after the k'th iteration, then the residual after the k'th iteration, $R^{(k)}$, is given by:

$$R^{(k)} = AP^{(k)} - Z. \tag{4.12}$$

The next iteration is

$$P^{(k+1)} = P^{(k)} - \omega A^T (AA^T)^{-1} R^{(k)}, \tag{4.13}$$

where ω is an acceleration parameter. In detail, the kernel of the relaxation process has the following equations:

$$R_{i,j}^{(k)} = \lambda_{i,j} (P_{i+1,j+1}^{(k)} + P_{i+1,j}^{(k)} - P_{i,j+1}^{(k)} - P_{i,j}^{(k)}) + Q_{i+1,j+1}^{(k)} + Q_{i,j+1}^{(k)} - Q_{i+1,j}^{(k)} - Q_{i,j}^{(k)}, \tag{4.14}$$

$$S_{i,j}^{(k)} = \lambda_{i,j} (Q_{i+1,j+1}^{(k)} + Q_{i+1,j}^{(k)} - Q_{i,j+1}^{(k)} - Q_{i,j}^{(k)}) - P_{i+1,j+1}^{(k)} - P_{i,j+1}^{(k)} + P_{i+1,j}^{(k)} + P_{i,j}^{(k)} - Z_{i,j}, \tag{4.15}$$

$$P_{i,j}^{(k+1)} = P_{i,j}^{(k)} + C_{i,j} (\lambda_{i,j} R_{i,j}^{(k)} - S_{i,j}^{(k)}), \tag{4.16}$$

$$Q_{i,j}^{(k+1)} = Q_{i,j}^{(k)} + C_{i,j} (R_{i,j}^{(k)} + \lambda_{i,j} S_{i,j}^{(k)}), \tag{4.17}$$

$$P_{i+1,j}^{(k+1)} = P_{i+1,j}^{(k)} - C_{i,j} (\lambda_{i,j} R_{i,j}^{(k)} + S_{i,j}^{(k)}), \tag{4.18}$$

$$Q_{i+1,j}^{(k+1)} = Q_{i+1,j}^{(k)} + C_{i,j} (R_{i,j}^{(k)} - \lambda_{i,j} S_{i,j}^{(k)}), \tag{4.19}$$

$$P_{i+1,j+1}^{(k+1)} = P_{i+1,j+1}^{(k)} - C_{i,j}\,(\lambda_{i,j}\,R_{i,j}^{(k)} - S_{i,j}^{(k)}), \tag{4.20}$$

$$Q_{i+1,j+1}^{(k+1)} = Q_{i+1,j+1}^{(k)} - C_{i,j}\,(R_{i,j}^{(k)} + \lambda_{i,j}\,S_{i,j}^{(k)}), \tag{4.21}$$

$$P_{i,j+1}^{(k+1)} = P_{i,j+1}^{(k)} + C_{i,j}\,(\lambda_{i,j}\,R_{i,j}^{(k)} + S_{i,j}^{(k)}), \tag{4.22}$$

$$Q_{i,j+1}^{(k+1)} = Q_{i,j+1}^{(k)} - C_{i,j}\,(R_{i,j}^{(k)} - \lambda_{i,j}\,S_{i,j}^{(k)}), \tag{4.23}$$

where

$$R = (\,R_{i,j}\,,\,S_{i,j}\,),$$

$$C_{i,j} \equiv \left[\omega(AA^T)^{-1}\right]_{i,j} = \frac{\omega}{4(\lambda_{i,j}^2 + 1)}.$$

This relaxation scheme is equivalent to an SOR method. On a serial computer the array of computational cells is swept over, applying equation (4.13) to each, until the maximum residual is reduced to the desired level.

The key to the adaptation of this relaxation scheme to parallel computers is the realization that each $\vec{P}$ is updated four times in a sequential sweep over the array of cells. This fact is utilized on parallel computers by using the concept of reordering to achieve parallelism [1], [27]; operations are reordered in order to increase the percentage of the computation that can be done in parallel. As shown in Fig. 4, the computational cells are divided into four sets of disjoint cells so that the cells of each set can be processed in parallel. A particular $\vec{P}$ which lies on the corner of a cell (see Fig. 4) is changed during the relaxation of set R first, then of set B, then of set O, and finally set G. In each of these cases, $\vec{P}$ lies at a different corner of the cell being relaxed. It is therefore clear that the cell iteration for the box variables is a four "color" scheme. Also, different linear combinations of the residuals are used to update each $\vec{P}$ and all of the *P's* are updated in each step. Thus the four steps are nccessary for a complete relaxation sweep. This is due to the fact that this is a multicolor cell relaxation scheme in contrast to multicolor point relaxation scheme in which only a fraction of the values are updated at each stage.

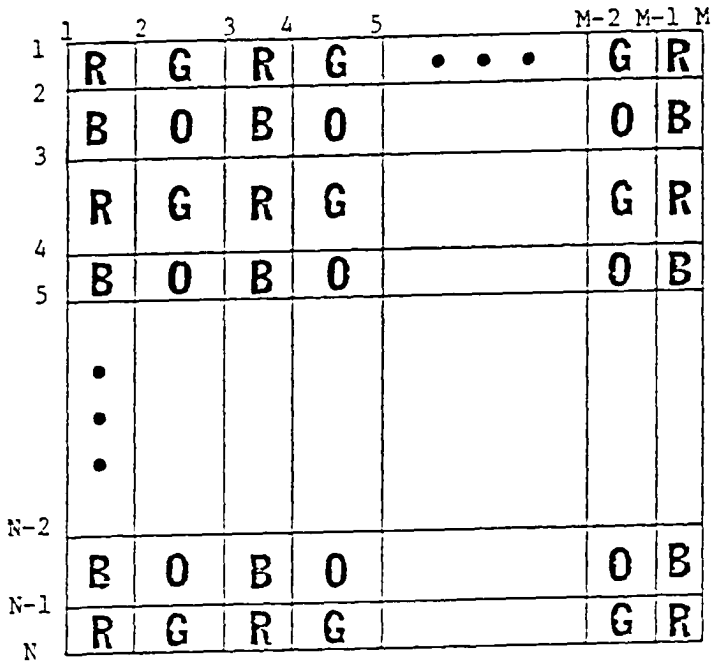

Fig. 4. Computational domain and the four color cells assuming that M and N are even numbers.

In brief, the relaxation algorithm is implemented by computing the residuals, $R^{(k)}$, using equation (4.12) for each set of cells, followed by updating the $P's$ using equation (4.13). This sequence must be completed four times in order to complete a sweep. Finally, the maximum residual is computed and tested against the convergence tolerance. The whole process is repeated until the iteration procedure converges.

A test problem, based on predefined values of ζ and boundary values of u and v, is used to study the behavior of the numerical method. Equations (4.12) and (4.13) are solved with given $\zeta_{i+1/2,j+1/2}$ and the value of one of the box variables on each side prescribed. In order to illustrate the behavior of this relaxation scheme the variation of the spectral radius, σ, with the acceleration parameter is shown in Fig. 5. This shows measured values of σ for three cases in which the number of grid points in the computational domain was increased from 32×32 to 128×128.

4.2. Implementation on the MPP

The computational cells, as described in section 4.1, are mapped onto the array so that the corners of the cells correspond to the processors. The storage pattern used on the MPP is to store $\vec{P}_{i,j}$, $\zeta_{i+1/2,j+1/2}$, and $\lambda_{i,j}$ in the memory of processor (i,j). Thus with a 128×128 array of processors there is an array of 127×127 cells.

The relaxation scheme has been implemented on the MPP for problems which fit on the array, 128×128 grid points, and for problems which are larger than the array, 128×255 grid points.

In detail, the MPP algorithm for a 128×128 problem is implemented as follows:

(1) All of the initialization is done on the VAX. This includes computing the matrices $Z_{i,j}$, $\lambda_{i,j}$, and $C_{i,j}$, calculating the boundary conditions, and initialization of the box variables

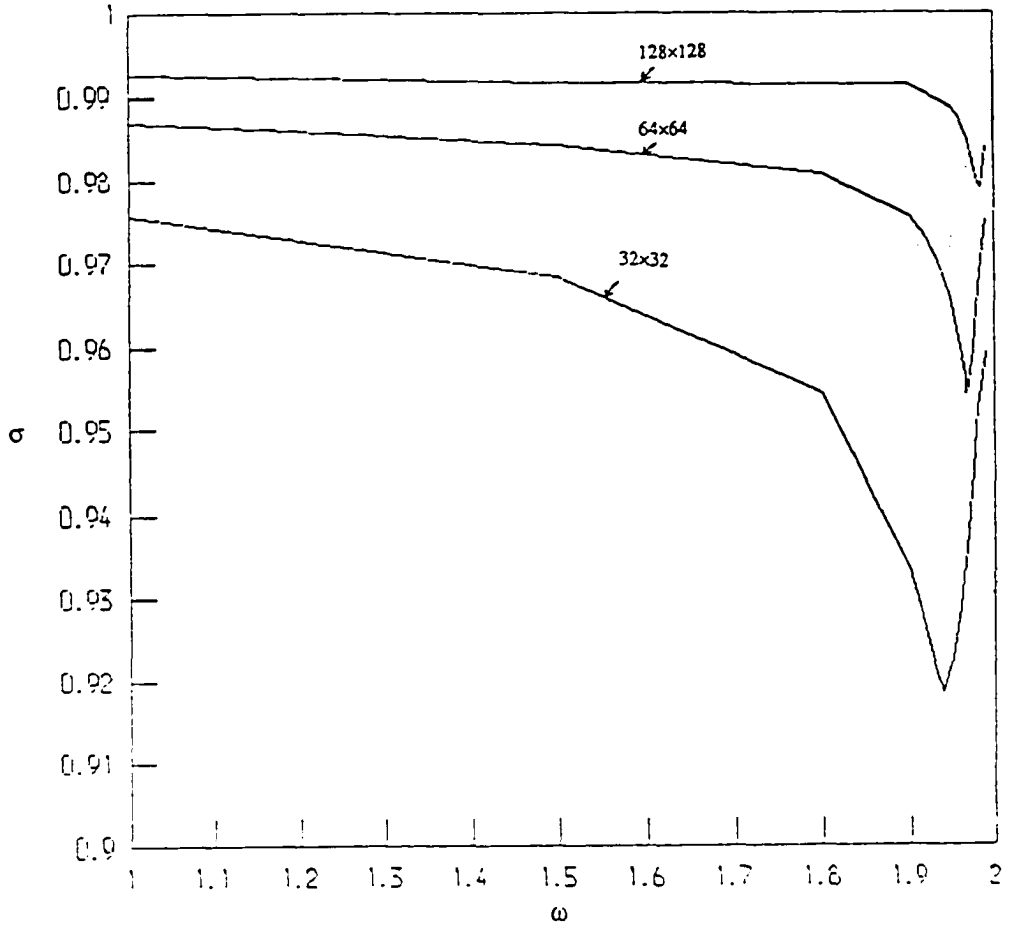

Fig. 5. Spectral radius as a function of the acceleration parameter.

($P_{i,j}$, $Q_{i,j}$ matrices) by setting all values to zero except those determined by the boundary values.

(2) The five matrices, computed in step (1), are moved to the staging memory, then to the array using the bit-plane I/O technique.

(3) The relaxation process is carried out entirely on the MPP. A set of temporary matrices are generated to store the *P's* for each color. The residuals are computed and the *P's* of the same color are updated; each of the four corners of the computational cells is updated by masking out the other corners. This is easily done using the where statement and boolean masks. The boundary values are also masked. The relaxation sequence is implemented four times to complete a sweep. This is followed by the computation of the maximum residual. This step is repeated until the process converges; that is the maximum residual is reduced to the desired value. Note that in updating the *P's* for each color only one forth of the processors do useful work.

(4) Finally, the box variables are moved back to the staging memory and then to the host using the bit-plane I/O method.

The relaxation procedure requires 22 parallel arrays of floating point numbers, all but 5 of which are temporary, and 19 parallel arrays of boolean variables. Each floating point array uses 32 bits and each boolean array uses 1 bit of the array memory. Together these arrays use 723 bits of the 1024 bit PE memory. Most of the remaining bits hold system functions and primitives. If one solves problems which are larger than 128×128, and thus do not "fit" on the array unit, additional memory is required. A total of 5 floating point arrays of data must be stored for each 128×128 sheet. Thus 160 bits of additional memory are needed for each sheet. This additional memory is not available in the PE memory so we must use the staging memory as a backup and move the data arrays in and out of the

array memory when we deal with 128×255 and larger problems.

A 128×255 problem is implemented on the MPP as follows:

(1) Initialization of the whole domain is performed on the VAX.

(2) The domain is decomposed into two regions, left and right. This means that the data is divided into two sheets; each sheet contains five matrices, $P_{i,j}$, $Q_{i,j}$, $Z_{i,j}$, $\lambda_{i,j}$, $C_{i,j}$. The two sheets are moved to the staging memory using bit-plane I/O method.

(3) The relaxation process is implemented on the MPP for each region separately. For each region, the following steps are performed: (a) A sheet of data is moved from the staging memory to the array, (b) the interface points of the box variables are updated from previous iteration of the other region, (c) the relaxation sequence and computation of the maximum residual are implemented as for the 128×128 problem, (d) the box variables are updated and boundary conditions are reset on all sides of the region except the interface side, (e) the interface points are saved in temporary arrays to be used for the next iteration of the other region, and (f) the box variables are moved back to the staging memory. These steps are repeated until convergence is achieved on both regions.

(4) After convergence the box variables of both regions are moved back to the host using bit-plane I/O method.

The host program as well as the MPP relaxation procedure are written in MPP Pascal and run through the MPP Pascal compiler version 2. Bit-plane I/O procedures are MCL routines, and are called from the MPP. A Fortran subroutine is used to initialize the buffer on the VAX for the parallel array transfers. These routines, declared as external procedures, are compiled as separate units and linked with the main program unit for execution, since, unlike standard Pascal, MPP Pascal provides the capability of compiling routines separately.

The relaxation algorithm mapped well onto the MPP because it can be implemented almost entirely with matrix operations. There are no vector operations and only two scalar operations per iteration. The amount of time spent on data transfers is quite small because nearly all data transfers are only between nearest neighbors. This type of transfer is generally inexpensive in machines like the MPP. The local nature of the data transfers is due to the fact that the differencing scheme is a compact second order scheme.

Table 2 contains the execution time and the processing rate for one iteration for a 128×128 and a 128×255 problem. The amount of time spent in the host program is not measured, because there is only a small amount of computation involved in it. The processing rate is determined by taking the ratio of the number of effective arithmetic operations to the total execution time of the relaxation routine. In counting the number of effective arithmetic operations, only pure arithmetic operations, addition and multiplication, are counted. Data transfers as well as computing the absolute and the maximum values are not counted as floating point operations.

Table 2. Measured execution times for one iteration and processing rates for the relaxation algorithm on the MPP.

Problem size (grid points)	Execution time (msec)	Processing rate (MFLOPS)
128×128	13.56	175
128×255	50.55	94

If the addition and the multiplication operations are counted separately and the maximum processing rates are considered, the maximum possible rate for this 128×128 problem will be 365 MFLOPS. The measured processing rate is, therefore, about 48% of the maximum possible rate.

In order to use the model developed in section 3.1 for the cost of implementing the relaxation algorithm on the MPP, the arithmetic and data transfer operations of the algo-

rithm are counted. The values of N_a, N_m, N_{sh}, N_{rt} for one iteration are 119, 26, 63, 84 respectively. The computation cost (equation (3.2)) and the communication cost (equation (3.3)) of the relaxation algorithm are listed in Table 3 using IBM format primitives and VAX format primitives version 2 compiler. We used the VAX format primitives called by the MPP Pascal compiler version 2 in our implementation and their usage in the model gives a reasonable measure of the performance of the algorithm on the MPP. Based on the measured values of these VAX format primitives, the computation cost contributes about 89% of the total cost and the communication cost contributes about 8% of the total cost. The costs of computing the absolute and the maximum values as well as performing the two scalar operations are not included in this model. However, it is estimated that these costs represent less than 3% of the total cost and these operations may overlap with the array operations. Note that this algorithm achieves only about 50% of the peak performance rate of the MPP because of the relative inefficiency of the VAX format primitives.

Table 3. Estimated times (in milliseconds) of the relaxation algorithm on the MPP.

Primitives	Computation time	Communication time	Total estimated time	Measured time
IBM format	6.50	0.67	7.17	-
VAX format	12.09	1.11	13.20	13.56

For the 128×255 problem there is an overhead for transferring the data to and from the staging memory. A total of seven data swaps between the stager and the array are required for each sheet. This swapping adds 11.7 msec to the time for each iteration; yielding an I/O overhead of 86%. This reduces the efficiency of the MPP for oversize problems.

The code can be easily expanded to larger problems using the same numerical algorithm. It is expected that the execution times will be multiples of that for the 128×255 problem.

4.3. Implementation on the Flex/32

The four color cell relaxation scheme, as described in section 4.1, was implemented on the Flex/32 using 64×64 cells (65×65 grid points) and 128×128 cells (129×129 grid points). The main program as well as the relaxation subroutine are written in Concurrent Fortran.

One obvious way to partition the relaxation routine is by color using four processors; each processor handles one color. Although the method is implemented easily, it has a slow convergence rate and no gain is achieved. This is because all of the processors are operating on the same initial data every iteration yielding a relaxation method which is equivalent to the Jacobi method. This method has a slow convergence rate compared to the SOR method.

In order to implement the algorithm in parallel, the domain is decomposed into 1, 2, 4, 8, or 16 strips; each strip contains 64×64, 64×32, 64×16, 64×8, or 64×4 cells for the 65×65 problem. Each strip is given to a process. At the beginning the main program, which is process 'main' running on processor 3, creates and starts (spawns) the execution of the processes on specified processors with each process assigned to a separate processor. Processors 4 to 19 are used for parallel processing.

Data is distributed between the common and local memories, with the intention of doing most of the work locally. The matrices $P_{i,j}$, $Q_{i,j}$, $Z_{i,j}$, $\lambda_{i,j}$, $C_{i,j}$ for each strip are stored in the local memory. These matrices are initialized in parallel by all processes. The values of the boundary points, interface points, and error flags are stored in the common memory. The boundary points are computed for the whole domain in 'main', and used by all processes.

The relaxation process for each strip is performed locally by: fetching the variables from the local memory, computing the residuals, then updating the variables, and finally storing them back in the local memory. After doing these steps four times, the maximum residual is computed and tested against the convergence tolerance. If the iteration has converged the convergence flag of that process is set to unity. After relaxing each set of cells (each color), each process exchanges the values of the interface points with its two neighbors through the common memory. A set of flags are used here to ensure that the updated values of the interface points are used for the next color.

Synchronization is accomplished by setting a variable, 'countr', in the common memory and assigning a lock to it. At the beginning of each iteration, 'countr' is set to zero by process 1. This is a signal to the processes to proceed. When each process completes a sweep it signals back to process 1 by incrementing 'countr'. Finally process 1 tests for global convergence and resets 'countr' if the iteration has not converged.

The performance of the parallel algorithm on the Flex/32 is evaluated by using the speedup and efficiency measures. The speedup is defined as the ratio of the time to solve the problem using one processor to the time to solve the same problem using p processors. Knowing the speedup, we determined the efficiency by taking the ratio of the speedup using p processors to p. Thus in the ideal situation the speedup is p and the efficiency is unity. The speedups and efficiencies as functions of the number of processors of both problems using two types of locks, MMOS and Local, are shown in Tables 4 and 5. The execution time for one iteration and the processing rate for both problems using 16 processors and local locks are listed in Table 6. These results were obtained using a timer with a 20 msec resolution.

Table 4. Speedup and efficiency of the relaxation algorithm on the Flex/32 using the MMOS locks.

Number of processors	65 × 65 grid points		129 × 129 grid points	
	speedup	efficiency	speedup	efficiency
1	1.000	1.000	1.000	1.000
2	1.976	0.988	1.975	0.988
4	3.864	0.966	3.912	0.978
8	7.254	0.907	7.687	0.961
16	10.977	0.686	14.239	0.890

Table 5. Speedup and efficiency of the relaxation algorithm on the Flex/32 using the Local locks.

Number of processors	65 × 65 grid points		129 × 129 grid points	
	speedup	efficiency	speedup	efficiency
1	1.000	1.000	1.000	1.000
2	1.991	0.996	1.978	0.989
4	3.955	0.989	3.941	0.985
8	7.825	0.978	7.834	0.979
16	15.294	0.956	15.437	0.965

Table 6. Measured execution times for one iteration and processing rates for the relaxation algorithm using 16 processors of the Flex/32.

Problem size (grid points)	Execution time (msec)	Processing rate (MFLOPS)
65 × 65	246.53	1.10
129 × 129	964.85	1.12

The results shown in Table 4 were obtained using the MMOS locks and the results shown in Table 5 were obtained using the Local locks. The MMOS locks, used to lock and unlock variables in the common memory, are provided by Flexible Computer Co. while the Local locks were programmed at NASA LaRC. The Local locks are based on the SBITI instruction of the NSC 32032 microprocessor while the MMOS locks are based on some expensive MMOS system calls. As shown in Tables 4 and 5, the Local locks are very efficient compared to the MMOS locks. For the 65 × 65 problem using 16 processors, for example, the speedup is 10.977 using the MMOS locks while it is 15.294 when using the Local locks. This is an increase of about 39%, and shows the impact of the design of

parallel processing primitives on the performance of the parallel machines. It was found that when the MMOS locks were used the synchronization cost of the algorithm represents more than 70% of the overhead cost for large number of processors.

In order to apply the model developed in section 3.2, equations (3.4) to (3.12), in estimating the computation and overhead times of the relaxation algorithm, the parameters f_{ld}, k_{lck}, and k_{cma} were computed. Since the load is distributed evenly between the processors, the load distribution factor is unity. The values of k_{lck} and k_{cma} per iteration are one and eight, respectively. Using these values, it was found that the overhead time represents at most 4% of the execution time of the algorithm. The synchronization time was insignificant because the routines that provide the locking mechanism are very efficient and overlap with the memory access. The spawning time has a minor impact because the processes are spawned only once at the beginning of the program. The busses contention factor is very small even for a large number of processors. The memory access cost dominates the overhead cost.

As the number of processors in use is increased from 2 to 16 the computation cost per processor is decreased while the overhead cost is increased. This causes a degradation in the efficiency of the algorithm. Increasing the number of cells causes an increase by the same ratio in the computation cost; an increase by a smaller ratio in the memory access cost; and no change in the spawning and synchronization costs. This resulted in a slight improvement on the performance of the algorithm for the 129×129 problem using a large number of processors.

4.4. Implementation on the Cray/2

The relaxation algorithm, described in section 4.1, was implemented on the Cray/2 for computational domains of sizes ranging from 64×64 to 1024×1024 grid points. The code,

in each case, is executed as a single job by one of the processors; multitasking was not attempted.

The algorithm is mapped onto the architecture so that columns of each color of the computational cells are processed separately. This mapping removes any recursion because each of these columns contains a disjoint set of cells. The implementation was quite simple. The code, using the reordered form of the algorithm and written in standard Fortran, was run through the CFT/2 compiler. Not all inner loops were vectorized. Two steps were taken to ensure vectorization of all inner loops of the code. First, CFT/2 was told to ignore apparent vector dependencies by using the compiler directive, IVDEP. Second, a segment of the code that computes the maximum value of an array was rewritten in order to be used with an optimized library routine, ISMAX. This resulted in the vectorization of all of the inner loops of the code.

The use of the main memory can be reduced by using scalar temporaries, instead of array temporaries, within inner DO loops. When scalar temporaries are used the CFT/2 compiler stores these variables in the local, rather than the main memory. This reduces memory conflicts and speeds up the calculation. The residuals, equations (4.14) and (4.15), are stored in scalar temporaries.

The measured serial rate of the 64×64 problem is 18 MFLOPS. This rate was obtained when running a serial version of the algorithm on one processor. With this version of the code the compiler was not able to vectorize the inner loops in the relaxation kernel. However, some of the loops in the section where the maximum residual is computed were vectorized. The 18 MFLOPS rate is thus a measure of the processing rate of a serial code; i.e., a code that runs on a serial machine. This result shows that existing codes, written for serial machines, may produce modest performance when they are transferred to

Cray/2.

Table 7 contains the execution time and the processing rate for one iteration using the vectorized code when the domain size is varied from 64×64 through 1024×1024 grid points. Only one processor of the Cray/2 is used. There is up to 20% offset on the results depending on the memory traffic and the number of the active processes on the system. The processing rate is computed by counting the additions and multiplications only, as in section (4.3). As the number of grid points is increased, the processing rate is slightly improved. This is due to the fact that the Cray/2 runs more efficiently on longer vectors; the number of instructions required to complete the loops is relatively smaller for long vectors [4].

Table 7. Measured execution times for one iteration and processing rates for the relaxation algorithm on one processor of the Cray/2.

Problem size (grid points)	Execution time (msec)	Processing rate (MFLOPS)
64×64	2.62	100
128×128	10.47	102
256×256	41.83	103
512×512	164.40	105
1024×1024	639.49	108

The major problem in implementing the relaxation algorithm on the Cray/2 was found to be accessing the main memory. The Cray/2 is a memory bound machine and one of the general rules for writing efficient programs for the Cray/2 is to maximize the number of arithmetic operations per memory access. If this is done the compiler can optimize the use of the functional units, vector registers, and the local memory, thus minimizing the use of the main memory. It has been our experience that a main memory access operation costs at least three times a floating point operation. Another related problem is using a memory stride of 2. This is inherent in the vectorized relaxation algorithm and cannot be avoided. A stride of 2 causes, as described before, a slowdown of more than 50% in data transfer

rate and about a 30% slowdown in the overall algorithm processing rate.

In general the CFT/2 compiler will, if possible, overlap the operation of the addition and multiplication pipeline units. However if there are, as is the case for this algorithm, more additions than multiplications, then only the addition pipeline can be active a portion of the time. Thus one can estimate that most of the time both of these pipelines are active and a portion of the time only the addition pipeline is active. These proportions can be estimated by counting the number of additions and multiplications of the algorithm.

The maximum processing rate of one pipeline in a single processor is about 244 MFLOPS, ignoring the startup time. From this we can compute, using the relative proportions of the time when one or two pipelines are active, the maximum possible processing rate for this algorithm on one processor. It is approximately 350 MFLOPS. The measured processing rate on a single processor (Table 7) is about 27% of the peak rate of one processor. If one includes the startup time for a pipeline, the peak processing rate drops to about 257 MFLOPS. Thus the measured rate is about 40% of the possible peak rate.

The relaxation algorithm has two main costs, the computation cost and the memory access cost. These costs are estimated using equations (3.13) to (3.16). The values of N_{f1}, N_{f2}, N_{m1}, and N_{m2} for the relaxation routine are 15 L_{vcr} additions and 2 L_{vcr} multiplications, 62 L_{vcr} additions and 36 L_{vcr} multiplications, 12 L_{vcr}, and 46 L_{vcr} respectively; where L_{vcr} is equal to the number of cells in each dimension. Table 8 contains the results of applying these values to equations (3.13) through (3.16) for different problem sizes. Also, the measured times are included in the table for comparison. The main memory access cost represents about 55% of the total estimated cost and about 70% of the measured value although a lower bound on the data transfer rates is considered. The total estimated costs exceed the measured values because of the overlapping between memory access and arith-

metic operations. Most of the multiplication operations are running in parallel with other operations so that the multiplication cost has minor impact on the overall cost. It is estimated that about one half of the addition operations can be issued while the system is fetching operands from the main memory.

Table 8. Estimated and measured execution times (in milliseconds) of the relaxation algorithm on the Cray/2.

Problem size	Memory access time	Addition time	Multiplication time	Total estimated time	Measured time
64 × 64	1.78	1.21	0.55	3.54	2.62
128 × 128	7.22	4.14	1.80	13.16	10.47
256 × 256	29.05	16.69	7.26	53.00	41.83
512 × 512	116.53	66.98	29.12	212.63	164.40
1024 × 1024	466.83	268.38	116.68	851.89	639.49

CHAPTER FIVE

Solving the Diffusion Equation

This chapter presents the implementation of an ADI method for solving the diffusion equation on the MPP, Flex/32, and Cray/2. The Gaussian elimination algorithm is used to solve a set of tridiagonal systems on the Flex/32 and Cray/2 while the cyclic elimination algorithm is used to solve these systems on the MPP. The numerical method and both algorithms are described in section 5.1. The implementation on each machine including the application of the performance models, developed in Chapter three, is described in sections 5.2 through 5.4. No reference to Chapter four is made in this chapter.

5.1. The numerical method

Consider the diffusion equation,

$$\frac{\partial u}{\partial t} = \nabla^2 u, \tag{5.1}$$

to be solved in $0 < t \leq T$ and the square region R: $0 \leq x \leq 1$, $0 \leq y \leq 1$ with boundary values at $u(0,y)$, $u(1,y)$, $u(x,0)$, $u(x,1)$. Consider replacing R by a net whose mesh points are denoted by $x_i = (i-1)\Delta x$, $y_j = (j-1)\Delta y$ where $i = 1, 2, \ldots, N+1$; $j = 1, 2, \ldots, M+1$. The numerical method used to approximate equation (5.1) is based on an Alternating Direction Implicit (ADI) method for solving parabolic equations [3]. This method consists of two half steps to advance the solution one full step in time. Let Δt be the full time step and apply the forward difference operator to equation (5.1) for the time derivative, giving

$$U_{i,j}^{n+1/2} - U_{i,j}^{n} = \frac{\Delta t}{2}(\delta_x^2 U_{i,j}^{n+1/2} + \delta_y^2 U_{i,j}^{n}), \tag{5.2}$$

$$U_{i,j}^{n+1} - U_{i,j}^{n+1/2} = \frac{\Delta t}{2}(\delta_x^2 U_{i,j}^{n+1/2} + \delta_y^2 U_{i,j}^{n+1}), \tag{5.3}$$

where only one of the space derivatives is evaluated at the advanced time level; this restriction is imposed in order to produce a set of tridiagonal equations. Define the centered second difference operator by:

$$\delta_x^2 U_{i,j} = \frac{(U_{i+1,j} - 2U_{i,j} + U_{i-1,j})}{(\Delta x)^2}. \tag{5.4}$$

Applying the centered second difference operator to equations (5.2) and (5.3) for the space derivatives, we get

$$\alpha\, U_{i-1,j}^{n+1/2} - (1+2\alpha)\, U_{i,j}^{n+1/2} + \alpha\, U_{i+1,j}^{n+1/2} = F_{i,j}, \tag{5.5}$$

$$\beta\, U_{i,j-1}^{n+1} - (1+2\beta)\, U_{i,j}^{n+1} + \beta\, U_{i,j+1}^{n+1} = G_{i,j}, \tag{5.6}$$

where

$$F_{i,j} = -\beta\, U_{i,j-1}^{n} - (1-2\beta)\, U_{i,j}^{n} - \beta\, U_{i,j+1}^{n}, \tag{5.7}$$

$$G_{i,j} = -\alpha\, U_{i-1,j}^{n+1/2} - (1-2\alpha)\, U_{i,j}^{n+1/2} - \alpha\, U_{i+1,j}^{n+1/2}, \tag{5.8}$$

$$\alpha = \Delta t \,/\, 2(\Delta x)^2, \tag{5.9}$$

$$\beta = \Delta t \,/\, 2(\Delta y)^2, \tag{5.10}$$

for $i = 1, 2, \ldots, N+1$; $j = 1, 2, \ldots, M+1$. Equation (5.5) represents a set of $M+1$ independent tridiagonal systems (one for each vertical line of the net) each of size $N+1$. Similarly, equation (5.6) represents a set of $N+1$ independent tridiagonal systems (one for each horizontal line of the net) each of size $M+1$. Equation (5.5) is solved for the set $\{U_{i,j}^{n+1/2}\}$ using the values of $U_{i,j}^{n}$ while equation (5.6) is solved for the set $\{U_{i,j}^{n+1}\}$ using the computed values of $U_{i,j}^{n+1/2}$. In order to solve these two sets of equations, the following boundary conditions are incorporated:

$$F_{1,j} \equiv U_{1,j}^{n+1/2} = U(0,y_j), \quad F_{N+1,j} \equiv U_{N+1,j}^{n+1/2} = U(1,y_j), \qquad \text{for } j = 1, 2, \ldots, M+1, \text{ and}$$

$$G_{i,1} \equiv U_{i,1}^{n+1} = U(x_i,0), \quad G_{i,M+1} \equiv U_{i,M+1}^{n+1} = U(x_i,1), \qquad \text{for } i = 1, 2, \ldots, N+1.$$

In brief, the ADI method is implemented in two steps. In the first step, the set $\{F_{i,j}\}$ is computed, using equation (5.7), followed by solving $M-1$ tridiagonal systems (excluding the boundary systems), using equation (5.5). With $U^{n+1/2}$ known, the set $\{G_{i,j}\}$ is computed, using equation (5.8), followed by solving $N-1$ tridiagonal systems (excluding the boundary systems), using equation (5.6). These two steps are needed to advance the solution one full time step.

The main issue in implementing the ADI method on a parallel computer is choosing an efficient algorithm for the solution of tridiagonal systems. The selection of the algorithm depends on the amount of hardware parallelism available on the computer, storage requirements, and some other factors. Two algorithms are considered here: Gaussian elimination and cyclic elimination. Although these algorithms are described in the literature (see [19] for details), they are briefly described here for the sake of completeness. Only the standard forms of these algorithms are considered here.

Consider solving a single tridiagonal system of n equations,

$$a_i x_{i-1} + b_i x_i + c_i x_{i+1} = d_i, \tag{5.11}$$

for $i = 2, 3, \ldots, n-1$, and the boundary conditions,

$$x_1 = d_1, \tag{5.12}$$

$$x_n = d_n. \tag{5.13}$$

The Gaussian elimination algorithm, based on an LU decomposition of the tridiagonal matrix, has two stages: the forward elimination and the back substitution. In the forward elimination stage two auxiliary vectors, w and g, are computed as follows:

$$w_1 = 0.0, \tag{5.14a}$$

$$w_i = c_i / (b_i - a_i w_{i-1}), \qquad i = 2, 3, \ldots, n-1; \tag{5.14b}$$

$$g_1 = d_1, \tag{5.15a}$$

$$g_i = (d_i - a_i g_{i-1}) / (b_i - a_i w_{i-1}), \qquad i = 2, 3, \ldots, n-1. \tag{5.15b}$$

The values of x_i are obtained in the back substitution stage as follows:

$$x_n = d_n, \tag{5.16a}$$

$$x_i = g_i - w_i x_{i+1}, \qquad i = n-1, n-2, \ldots, 1. \tag{5.16b}$$

Gaussian elimination is an inherently serial algorithm because of the recurrence relations in both stages of the algorithm. However, if one is faced with solving a set of independent tridiagonal systems, then Gaussian elimination will be the best algorithm to use on a parallel computer. This means that all systems of the set are solved in parallel. In this case we obtain both the minimum number of arithmetic operations and the maximum parallelism.

The cyclic elimination algorithm, also called odd-even elimination [18] or parallel cyclic reduction [19], is a variant of the cyclic reduction algorithm [19] applying the reduction procedure to all of the equations and eliminating the back substitution phase of the algorithm. This makes cyclic elimination most suitable for machines with a large natural parallelism, like the MPP. It can be described as follows. Assume that $n = 2^r$ where r is an integer and

$$x_i = 0, \qquad \text{for } i \le 0 \text{ and } i > n. \tag{5.17}$$

Solving equation (5.11) for x_i and the corresponding equations for x_{i-1} and x_{i+1}, we have:

$$x_{i-1} = (d_{i-1}/b_{i-1}) - (a_{i-1}/b_{i-1})\, x_{i-2} - (c_{i-1}/b_{i-1})\, x_i, \tag{5.18}$$

$$x_i = (d_i / b_i) - (a_i / b_i)\, x_{i-1} - (c_i / b_i)\, x_{i+1}, \tag{5.19}$$

$$x_{i+1} = (d_{i+1}/b_{i+1}) - (a_{i+1}/b_{i+1})\, x_i - (c_{i+1}/b_{i+1})\, x_{i+2}. \tag{5.20}$$

Substitute for x_{i-1} and x_{i+1} in equation (5.19), to get

$$a_i^{(1)} x_{i-2} + b_i^{(1)} x_i + c_i^{(1)} x_{i+2} = d_i^{(1)}, \qquad i = 2, 3, \ldots, n-1, \tag{5.21}$$

where

$$a_i^{(1)} = -(a_i / b_i)(a_{i-1} / b_{i-1}), \tag{5.22}$$

$$b_i^{(1)} = 1 - (a_i / b_i)(c_{i-1} / b_{i-1}) - (a_{i+1} / b_{i+1})(c_i / b_i), \tag{5.23}$$

$$c_i^{(1)} = -(c_i / b_i)(c_{i+1} / b_{i+1}), \tag{5.24}$$

$$d_i^{(1)} = (d_i / b_i) - (a_i / b_i)(d_{i-1} / b_{i-1}) - (c_i / b_i)(d_{i+1} / b_{i+1}). \tag{5.25}$$

The above process eliminates the odd variables in the even equations and the even variables in the odd equations by performing elementary row operations. The resulting system is again tridiagonal of the same form as equation (5.11) but with different coefficients (a_i, b_i, c_i) and forcing terms (d_i). This process can be repeated for r steps ($r = \log_2 n$) until one set of equations remains. These equations are

$$a_i^{(r)} x_{i-2^r} + b_i^{(r)} x_i + c_i^{(r)} x_{i+2^r} = d_i^{(r)}, \qquad i = 2, 3, \ldots, n-1. \tag{5.26}$$

The terms x_{i-2^r} and x_{i+2^r} of equation (5.26) are really zero because they refer to values outside the range $1 \le i \le n$ and by equation (5.17) are zero. Thus equation (5.26) becomes

$$x_i = d_i^{(r)} / b_i^{(r)}, \qquad \text{for } i = 2, 3, \ldots, n-1. \tag{5.27}$$

The cyclic elimination procedure therefore requires the computation of new sets of coefficients and forcing terms, for levels $k = 1, 2, \ldots, r$, from

$$a_i^{(k)} = -A_i A_{i-K}, \tag{5.28}$$

$$b_i^{(k)} = 1 - A_i C_{i-K} - A_{i+K} C_i, \tag{5.29}$$

$$c_i^{(k)} = -C_i C_{i+K}, \tag{5.30}$$

$$d_i^{(k)} = D_i - A_i D_{i-K} - C_i D_{i+K}, \tag{5.31}$$

where

$$A_i \equiv \frac{a_i^{(k-1)}}{b_i^{(k-1)}}, \quad C_i \equiv \frac{c_i^{(k-1)}}{b_i^{(k-1)}}, \quad D_i \equiv \frac{d_i^{(k-1)}}{b_i^{(k-1)}}, \quad K \equiv 2^{k-1}, \quad i = 2, 3, \ldots, n-1, \tag{5.32}$$

followed by computing the values of x_i, using equation (5.27). Note that for $k = 1$ equations (5.28) to (5.31) are equivalent to equations (5.22) to (5.25) with $a_i^{(0)} = a_i$, $b_i^{(0)} = b_i$, $c_i^{(0)} = c_i$, and $d_i^{(0)} = d_i$.

The ADI procedure to solve equation (5.1) can be applied for any set of boundary and initial conditions. A test problem is developed by setting U to zero everywhere at $t = 0$ on the interior and

$$U(0,y) = 1 + 0.25(y - 3y^2), \quad U(1,y) = 1 + 0.25(2y - 3y^2 - 3),$$

$$U(x,0) = 1 + 0.25(3x^2 - 6x), \quad U(x,1) = 1 + 0.25(3x^2 - 5x - 2),$$

for all time. The steady state solution to this problem is,

$$U = 1 + \frac{1}{4}\left[3(x^2 - y^2) + xy - 6x + y\right]. \tag{5.33}$$

5.2. Implementation on the MPP

The ADI method, described in section 5.1, was implemented on the MPP for a 128×128 mesh point problem. In each processor, data corresponding to one mesh point is stored. The main program as well as the ADI procedure were written in MPP Pascal and run through the MPP Pascal compiler version 2. The main program, run on the VAX, handles input and output, initialization of the computational domain, and calling the ADI procedure. The ADI procedure, which was executed entirely on the array, computes the coefficients, forcing terms, and solves two sets of 128 tridiagonal systems. The tridiagonal systems are solved by the cyclic elimination algorithm, described in section 5.1, for all rows and all columns. This is done in parallel on the array with a tridiagonal system of 128 equations being solved on each row or column. In this case the vectors x_i, a_i, b_i, c_i, d_i of equation (5.11) become the matrices $x_{i,j}$, $a_{i,j}$, $b_{i,j}$, $c_{i,j}$, $d_{i,j}$. After solving each set of the tridiagonal systems, all points of the domain are updated except the boundary points. This is

implemented by masking out the boundary columns (or the boundary rows) of the array.

The ADI procedure is reasonably efficient and requires only five parallel arrays of floating point numbers. The procedure contains mostly matrix operations with a few scalar operations (for computing the coefficients of $U_{i,j}$, equations (5.5) to (5.8)) and has no vector operations. However, the cyclic elimination algorithm has some hidden defects. For each level of the elimination process, a set of data is shifted off the array and an equal set of zeros is shifted onto the array. Since all of the processors are executing the same instruction at every cycle, some of these processors may not be doing useful work; here they are either multiplying by zero or adding a zero. This is a problem with many algorithms on SIMD machines.

Table 9 contains the execution time and the processing rate of the ADI procedure for the 128×128 problem. The processing rate is determined by counting only the arithmetic operations (addition, multiplication, and division). Data transfer operations, vital as they are in this work, are not counted as floating point operations. However, all the arithmetic operations including those operations which do not contribute to the solution are counted.

Table 9. Measured execution time per time step and processing rate for the ADI procedure on the MPP.

Problem size (mesh points)	Execution time (msec)	Processing rate (MFLOPS)
128×128	23.698	134

In order to use the model developed in section 3.1 for the cost of implementing the ADI method on the MPP, the data transfer as well as the arithmetic operations of the method are counted. Table 10 contains the operation counts for one pass of the ADI method using the cyclic elimination algorithm. These operations are also required for the second pass of the method and the total number of operations required for both passes will be twice the number that is given in Table 10. The computation cost (equation (3.2)) and

the communication cost (equation (3.3)) of the ADI method on the MPP are listed in Table 11 using VAX format primitives version 2 compiler (Table 1). The computation cost contributes about 75% of the total cost and the communication cost contributes about 25% of the total cost. The scalar operations have very little impact on the performance of the method since they are inexpensive and may overlap with the array operations. It is also found that the cost of solving the tridiagonal systems represents more than 95% of the total cost of the method.

Table 10. Operation counts for one pass of the ADI method using the cyclic elimination algorithm for solving the tridiagonal systems, where $i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, n$, and $r = \log_2 n$.

Operation	Add	Multiply	Divide	Shift	Steps shifted
1. Compute $F_{i,j}$, eq. (5.7)	2	3	-	2	2
2. Compute $A_{i,j}$, $C_{i,j}$, $D_{i,j}$, eqs. (5.32)	-	-	$3r$	-	-
3. Compute $a_{i,j}^{(r)}$, eq. (5.28)	-	r	-	r	$2^r - 1$
4. Compute $b_{i,j}^{(r)}$, eq. (5.29)	$2r$	$2r$	-	$2r$	$2(2^r - 1)$
5. Compute $c_{i,j}^{(r)}$, eq. (5.30)	-	r	-	r	$2^r - 1$
6. Compute $d_{i,j}^{(r)}$, eq. (5.31)	$2r$	$2r$	-	$2r$	$2(2^r - 1)$
7. Compute $x_{i,j}$, eq. (5.27)	-	-	1	-	-
Total operations	$4r + 2$	$6r + 3$	$3r + 1$	$6r + 2$	$6(2^r - 1) + 2$
Total operations for $r = 7$	30	45	22	44	764

Table 11. Estimated and measured times (in milliseconds) of the ADI method on the MPP.

Computation time	Communication time	Total estimated time	Measured time
17.809	6.069	23.878	23.698

If we use the maximum processing rates of the arithmetic operations on the MPP (see section 2.1) and the operation counts from Table 10, the maximum possible rate of the ADI method will be 236 MFLOPS. Therefore, the measured processing rate is about 57% of the

maximum possible rate. The peak performance rate is not achieved because of the data transfer costs and the inefficiency of the VAX format primitives.

5.3. Implementation on the Flex/32

The ADI method, described in section 5.1, was implemented on the Flex/32 using p processors with p = 1, 2, 4, 8, and 16 and for problems of sizes $n \times n$ mesh points with n = 64, 128, and 256. The main program as well as the ADI subroutine were written in Concurrent Fortran.

In order to implement the method in parallel, the domain is decomposed first vertically into n by n / p strips, for the first pass of the method, and then horizontally into n / p by n strips, for the second pass of the method. In the first pass, a set of n / p tridiagonal systems (each of n equations) corresponding to the vertical lines of the net are solved for each strip using the Gaussian elimination algorithm, described in section 5.1, In the second pass, likewise, a set of n / p tridiagonal systems (each of n equations) corresponding to the horizontal lines of the net are solved for each strip using the same algorithm. In our implementation each strip is given to a process. In addition to initialization of the domain and input and output operations, the main program creates and starts the execution of the processes on specified processors with each process assigned to a separate processor. The processes are spawned only once at the beginning of the program and are used for both passes of the method. The data corresponding to the domain is stored in the common memory. Also, the values of the boundary points, computed in the main program, are stored in the common memory. The forcing terms and the temporary matrices $w_{i,j}$ and $g_{i,j}$, equations (5.14) and (5.15), for each strip are computed and stored in the local memory of each processor.

The ADI method was implemented by: computing the coefficients and forcing terms of the tridiagonal systems and solving these systems for all columns first and then for all

rows. Upon completing a pass, each process signals to the other processes by incrementing a counter. All of the processes will wait until each of them has finished the current pass. A lock is assigned to the counter to ensure this sequence of the events.

The speedup and efficiency as functions of the number of processors for problems of sizes 64×64, 128×128, and 256×256 are listed in Table 12. The efficiency of the method ranges from 35%, for the 64×64 problem using 16 processors, to 95%, for the 256×256 problem using two processors. The measured execution times and the processing rates for these problems are listed in Table 13.

Table 12. Speedup and efficiency of the ADI method on the Flex/32.

Number of processors	64×64 points		128×128 points		256×256 points	
	speedup	efficiency	speedup	efficiency	speedup	efficiency
1	1.000	1.000	1.000	1.000	1.000	1.000
2	1.827	0.913	1.874	0.937	1.900	0.950
4	3.393	0.848	3.660	0.915	3.722	0.931
8	5.278	0.660	6.807	0.851	7.271	0.909
16	5.588	0.349	10.486	0.655	13.414	0.838

Table 13. Measured execution times per time step and processing rates for the ADI method using 16 processors of the Flex/32.

Problem size (mesh points)	Execution time (msec)	Processing rate (MFLOPS)
64×64	340	0.36
128×128	740	0.66
256×256	2320	0.85

Table 14 contains the estimated values of the computation time and the overhead times of the ADI method using equations (3.4) to (3.12). Since the ADI method is applied to the interior points of the region only, the first and last processors have less work to do than the other processors. This means that load is not distributed evenly between the processors except when we use two processors only. Therefore, the load distribution factor is

$$f_{ld} = \left\lceil \frac{n-2}{p} \right\rceil \left(\frac{p}{n-2} \right). \tag{5.34}$$

Table 14. Estimated and measured times (in milliseconds) of the ADI method on the Flex/32.

No. of procs.	Computation time	Spawning time	Memory time	Total estimated time	Measured time
64 × 64 points					
1	-	-	-	-	1900
2	981	26	16	1023	1040
4	490	52	8	550	560
8	245	104	4	353	360
16	123	208	2	333	340
128 × 128 points					
1	-	-	-	-	7760
2	3942	26	65	4033	4140
4	1971	52	32	2055	2120
8	985	104	16	1105	1140
16	493	208	8	709	740
256 × 256 points					
1	-	-	-	-	31120
2	15683	26	260	15969	16380
4	7841	52	130	8023	8360
8	3921	104	65	4090	4280
16	1960	208	33	2201	2320

The overhead time of storing vectors in the common memory instead of the local memories is the only overhead that contributes to accessing the common memory. This overhead results from having all variables stored in the local memory when we used one processor while some of these variables were stored in the common memory in the multiprocessing environment. The values of k_{lck} (equation (3.8)) and k_{cld} (equation (3.12)) are two and 8 n / p, respectively. The synchronization time is insignificant because the routines that provide the locking mechanism are very efficient. The spawning time dominates the overhead time for large number of processors. This resulted in the degradation on the performance of the method for large number of processors. However, the overhead time is dominated by the common memory access time for a small number of processors. The total

estimated times are less than the measured times by less than 4%. The busses contention factor, f_{bc}, is not included in the common memory access time. However, it is expected that it has a minor impact on the results. Finally, it was realized that the common memory access operations may overlap with the spawning operations during the first pass of the method.

5.4. Implementation on the Cray/2

The ADI method, described in section 5.1, was implemented on one processor of the Cray/2 for domains of sizes ranging from 64×64 to 1024×1024 mesh points. The code, in each case, was written and run through the CFT/2 compiler. Each set of the tridiagonal systems is solved by the Gaussian elimination algorithm, described in section 5.1, for all systems of the set in parallel. This means that each element of w_i (equations (5.14)) is computed for all the tridiagonal systems before moving to the next element. This process is repeated in computing the elements of g_i (equations (5.15)) and x_i (equations (5.16)). The implementation requires that the vectors w_i, g_i, x_i be changed to the matrices $w_{i,j}$, $g_{i,j}$, $x_{i,j}$. When these are done, all statements of the code vectorize fully and the recursion problem of the algorithm is eliminated.

The ADI method requires that the data is first referenced by vertical lines and then by horizontal lines. The CFT/2 compiler stores arrays by incrementing the leftmost index first (column major order). This means that referencing the data by vertical lines causes no problems because the increment between data elements is unity. However, in referencing the data by horizontal lines the increment between the elements will be equal to the number of variables in each column. In our implementation, this number ranges between 64 and 1024. This could cause a major problem on the Cray/2. As described in section 2.3, the machine has 128 memory banks and consecutive elements of an array are stored in

consecutive banks. For a memory stride of 128, for example, words are drawn from the same bank and each word must wait for 57 clock periods to be moved from main memory, even if there is no competition for resources from other processors. As a result, strides that are divisible by 128 or any large power of two result in major performance reductions. This problem is overcome by storing the data as though it had a column length one greater than its actual length. Thus for a stride of 129 data elements are stored in sequential banks with a stride of one (stride of 129 mod 128 banks = 1).

The number of memory access operations can be reduced by using the local memory whenever that is possible. Once an array is stored in the local memory, none of its elements can be changed; all the elements are treated the same. The forcing terms $F_{i,j}$ and $G_{i,j}$, equations (5.7) and (5.8), are stored in the local memory.

Table 15 contains the execution time and the processing rate of the ADI method when the domain size is varied from 64×64 through 1024×1024 mesh points. The processing rate is computed by counting the additions, multiplications, and divisions only. Division is counted as a single arithmetic operation. The number of arithmetic operations for one pass of the ADI method, using the Gaussian elimination algorithm for solving the tridiagonal systems, are listed in Table 16. The total number of each operation will be twice the number that is given in Table 16.

Table 15. Measured execution times per time step and processing rates for the ADI method on one processor of the Cray/2.

Problem size (mesh points)	Execution time (msec)	Processing rate (MFLOPS)
64×64	1.684	69
128×128	6.474	74
256×256	24.270	80
512×512	94.037	83
1024×1024	364.248	86

Table 16. Operation counts for one pass of the ADI method using the Gaussian elimination algorithm for solving the tridiagonal systems.

Operation	Add	Multiply	Divide
1. Compute $F_{i,j}$, eq. (5.7)	2	3	-
2. Compute $w_{i,j}$, eqs. (5.14)	1	1	1
3. Compute $g_{i,j}$, eqs. (5.15)	2	2	1
4. Compute $x_{i,j}$, eqs. (5.16)	1	1	-
Total operations	6	7	2

Table 17 contains the results of applying equations (3.13) to (3.16) for the ADI method. The number of floating point operations is obtained by multiplying the values given in Table 16 by L_{vcr} ($L_{vcr} = n - 2$) while the number of main memory access operations is 12 L_{vcr} for each pass of the method. The multiplication time includes the time required for division; each division is estimated as three multiplications in this model. The cost of scalar operations are not included in the model. The memory access time represents about 47% of the total estimated time. The estimated time for memory access does not take into account the competition for memory banks from other processors which causes a lower data transfer rate. The total estimated time exceeds the measured time for large domains. This is because for large problems more overlapping between different operations is expected and the impact of scalar operations is reduced. For these reasons, the performance rate of the method, Table 15, increases for large problems.

Because the ADI method has more multiplications (including divisions) than additions, the time to implement the method can be considered as a summation of two portions; a portion with one operational pipeline and a portion with two pipelines. By using Table 16 and assuming that the peak performance rate of one pipeline is 244 MFLOPS, ignoring the vector startup times, the maximum processing rate of the ADI method will be 357 MFLOPS.

Therefore, the measured processing rate for the 128×128 problem is about 20% of the peak performance rate. If the startup time of the floating point units is included, the measured processing rate for the 128×128 problem will be about 28% of the peak processing rate of 262 MFLOPS. The major problems are accessing the main memory and the scalar portion of the code.

Table 17. Estimated and measured execution times (in milliseconds) of the ADI method on one processor of the Cray/2.

Problem size (mesh points)	Memory access time	Addition time	Multiplication time	Total estimated time	Measured time
64×64	0.720	0.259	0.562	1.541	1.684
128×128	2.951	1.066	2.310	6.327	6.474
256×256	11.947	4.324	9.368	25.639	24.270
512×512	48.076	17.414	37.730	103.220	94.037
1024×1024	192.883	69.893	151.430	414.206	364.248

CHAPTER SIX

Solving Navier-Stokes Equations

In this chapter, the implementation of Navier-Stokes equations on the MPP, Flex/32, and Cray/2 is presented. The cell relaxation scheme, described in Chapter four, and the ADI method, described in Chapter five, are used to solve these equations. The numerical method is presented in section 6.1. The implementation on each machine including the application of the performance models, described in Chapter three, is described in sections 6.2 through 6.4.

6.1. The numerical method

The Navier-Stokes equations for the two-dimensional, time dependent flow of a viscous incompressible fluid may be written, in dimensionless variables, as:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0, \tag{6.1}$$

$$\frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} = \zeta, \tag{6.2}$$

$$\frac{\partial \zeta}{\partial t} + \frac{\partial}{\partial x}(u\,\zeta) + \frac{\partial}{\partial y}(v\,\zeta) = \frac{1}{\mathrm{Re}}\nabla^2 \zeta, \tag{6.3}$$

where $\vec{u} = (u,v)$ is the velocity, ζ is the vorticity and Re is the Reynolds number.

The numerical method used to approximate equations (6.1) to (6.3) is based on the compact differencing schemes, described in section 4.1. Gatski et al. [13] applied the compact scheme to solve these Navier-Stokes equations. Grosch [17] adapted the Navier-Stokes code to ICL-DAP. The set of difference equations and boundary conditions resulted from

applying the compact differencing scheme to equations (6.1) and (6.2) are solved using the cell relaxation scheme, see section 4.1. The compact difference approximation to equation (6.3) results in an implicit set of equations for ζ at the next time step. This set of equations are solved by an ADI method which, as described in section 5.1, requires the solution of tridiagonal systems on each sweep. The approximation to equation (6.3) is briefly described here.

Consider the problem of approximating the solution of equations (6.1) to (6.3) in the square domain $0 \le x \le 1$, $0 \le y \le 1$ with the boundary conditions $u = 1$ and $v = 0$ at $y = 1$ and $u = v = 0$ elsewhere. Subdivide the computational domain into rectangular cells, as in section 4.1. Apply the centered difference and average operators, equations (4.3) and (4.4), to get equations (4.5) to (4.8) which are solved using the cell relaxation scheme, as described in section 4.1. Let Δt be a full time step and apply the forward difference operator to equation (6.3) for the time derivative, giving

$$\zeta_{ij}^{n+1/2} - \zeta_{ij}^{n} = \frac{\Delta t}{2}\left[\frac{1}{\mathrm{Re}}\delta_x^2\,\zeta_{ij}^{n+1/2} - \delta_x\,(U_{ij}\,\zeta_{ij}^{n+1/2}) + \frac{1}{\mathrm{Re}}\delta_y^2\,\zeta_{ij}^{n} - \delta_y\,(V_{ij}\,\zeta_{ij}^{n})\right], \tag{6.4}$$

$$\zeta_{ij}^{n+1} - \zeta_{ij}^{n+1/2} = \frac{\Delta t}{2}\left[\frac{1}{\mathrm{Re}}\delta_x^2\,\zeta_{ij}^{n+1/2} - \delta_x\,(U_{ij}\,\zeta_{ij}^{n+1/2}) + \frac{1}{\mathrm{Re}}\delta_y^2\,\zeta_{ij}^{n+1} - \delta_y\,(V_{ij}\,\zeta_{ij}^{n+1})\right]. \tag{6.5}$$

Applying the centered difference operator, equation (4.3), and the centered second difference operator, equation (5.4), for the space derivatives, we get

$$\beta_{ij}^{(x)}\,\zeta_{i-1j}^{n+1/2} - (1 + 2\alpha_j^{(x)})\,\zeta_{ij}^{n+1/2} + \gamma_{ij}^{(x)}\,\zeta_{i+1j}^{n+1/2} = F_{ij}, \tag{6.6}$$

$$\beta_{ij}^{(y)}\,\zeta_{ij-1}^{n+1} - (1 + 2\alpha_i^{(y)})\,\zeta_{ij}^{n+1} + \gamma_{ij}^{(y)}\,\zeta_{ij+1}^{n+1} = G_{ij}, \tag{6.7}$$

where

$$F_{ij} = -\beta_{ij}^{(y)}\,\zeta_{ij-1}^{n} - (1 - 2\alpha_i^{(y)})\,\zeta_{ij}^{n} - \gamma_{ij}^{(y)}\,\zeta_{ij+1}^{n}, \tag{6.8}$$

$$G_{ij} = -\beta_{ij}^{(x)}\,\zeta_{i-1j}^{n+1/2} - (1 - 2\alpha_j^{(x)})\,\zeta_{ij}^{n+1/2} - \gamma_{ij}^{(x)}\,\zeta_{i+1j}^{n+1/2}, \tag{6.9}$$

$$\alpha_j^{(x)} = \frac{\Delta t}{2(\Delta x)_j^2 \, \mathrm{Re}}, \tag{6.10}$$

$$\alpha_i^{(y)} = \frac{\Delta t}{2(\Delta y)_i^2 \, \mathrm{Re}}, \tag{6.11}$$

$$\beta_{i,j}^{(x)} = \alpha_j^{(x)} + \frac{\Delta t}{4(\Delta x)_j} U_{i-1,j}, \tag{6.12}$$

$$\beta_{i,j}^{(y)} = \alpha_i^{(y)} + \frac{\Delta t}{4(\Delta y)_i} V_{i,j-1}, \tag{6.13}$$

$$\gamma_{i,j}^{(x)} = \alpha_j^{(x)} - \frac{\Delta t}{4(\Delta x)_j} U_{i+1,j}, \tag{6.14}$$

$$\gamma_{i,j}^{(y)} = \alpha_i^{(y)} - \frac{\Delta t}{4(\Delta y)_i} V_{i,j+1}. \tag{6.15}$$

The velocity field is not defined at the corners of the cells in this scheme; however, it can be computed as follows:

$$\vec{U}_{i,j} = \frac{1}{2}(\vec{U}_{i+1/2,j} + \vec{U}_{i-1/2,j}). \tag{6.16}$$

Using equation (4.10), to get

$$\vec{U}_{i,j} = \frac{1}{4}(\vec{P}_{i+1,j} + 2\vec{P}_{i,j} + \vec{P}_{i-1,j}). \tag{6.17}$$

Equations (6.6) and (6.7), which are similar to equations (5.5) and (5.6), represent two sets of independent tridiagonal systems. These systems can be solved using one of the algorithms for solving the tridiagonal systems, described in section 5.1.

The ADI method for equation (6.3) is applied to all interior points of the domain. The values of ζ on the boundaries are computed using equation (6.2). On $x = 0$, for example, we have

$$(\zeta = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y})_{x=0} \tag{6.18}$$

but $u = 0$ at $x = 0$, so $(\partial u / \partial y)_{x=0} = 0$ and

$$(\zeta)_{x=0} = (\frac{\partial v}{\partial x})_{x=0}. \tag{6.19}$$

Assume that the length, Δx, of the first two cells bordering the boundary are equal and approximate the derivative of v, to get

$$(\frac{\partial v}{\partial x})_{x=0} \simeq a_0 (v)_{x=0} + a_1 (v)_{x=\Delta x} + a_2 (v)_{x=2\Delta x}. \tag{6.20}$$

Using the Taylor series for v at $x = \Delta x$, $2\Delta x$ about $x = 0$, we get

$$(\frac{\partial v}{\partial x})_{x=0} = a_0 (v)_{x=0} + a_1 \left\{ (v)_{x=0} + \Delta x (\frac{\partial v}{\partial x})_{x=0} + \frac{1}{2}(\Delta x)^2(\frac{\partial^2 v}{\partial x^2})_{x=0} + O\left[(\Delta x)^3\right] \right\}$$

$$+ a_2 \left\{ (v)_{x=0} + 2\Delta x (\frac{\partial v}{\partial x})_{x=0} + \frac{1}{2}(2\Delta x)^2(\frac{\partial^2 v}{\partial x^2})_{x=0} + O\left[(\Delta x)^3\right] \right\}. \tag{6.21}$$

Solving equation (6.21) for the unknowns a_0, a_1, a_2, we have

$$(\zeta)_{x=0} = (\frac{1}{2\Delta x})\left[-3 (v)_{x=0} + 4 (v)_{x=\Delta x} - (v)_{x=2\Delta x}\right]. \tag{6.22}$$

The value of v on $x = 0$ is zero while its value on $x = \Delta x$, $2\Delta x$ is computed using equation (6.17). Similarly,

$$(\zeta)_{x=1} = (\frac{\partial v}{\partial x})_{x=1} \simeq (\frac{1}{2\Delta x})\left[3 (v)_{x=1} - 4 (v)_{x=1-\Delta x} + (v)_{x=1-2\Delta x}\right], \tag{6.23}$$

$$(\zeta)_{y=1} = (-\frac{\partial u}{\partial y})_{y=1} \simeq (\frac{1}{2\Delta y})\left[-3 (u)_{y=1} + 4 (u)_{y=1-\Delta y} - (u)_{y=1-2\Delta y}\right], \tag{6.24}$$

while

$$(\zeta)_{y=0} = (\frac{\partial v}{\partial x})_{y=0} \simeq (\frac{1}{2\Delta x})\left[(v)_{y=0,\, x=\Delta x} - (v)_{y=0,\, x=2\Delta x}\right]. \tag{6.25}$$

The solution procedure for the Navier-Stokes equations can be summerized as follows:

step (1) Assume that ζ is zero everywhere at $t = 0$. The box variables for the relaxation process are initialized and their boundary values are computed.

step (2) The vorticity at the corners of the domain is not defined in this scheme. These values are approximated using the values of their neighboring points. The

values of $\zeta_{i+1/2,j+1/2}$ are computed using the values of ζ at the corners of the cells.

step (3) The relaxation process for the box variables, as described in section 4.1, is performed.

step (4) The coefficients $\alpha_j^{(x)}$, $\alpha_i^{(y)}$, $\beta_{i,j}^{(x)}$, $\beta_{i,j}^{(y)}$, $\gamma_{i,j}^{(x)}$, $\gamma_{i,j}^{(y)}$ (equations (6.10) to (6.15)) for both passes of the ADI method are computed. This includes computing $U_{i,j}$ and $V_{i,j}$, equation (6.17).

step (5) The values of ζ on the boundaries, equations (6.22) to (6.25), are computed.

step (6) The ADI method for ζ is applied to all interior points of the domain, equations (6.6) and (6.7).

The repetition of steps (2) through (6) yields the values of the velocity and vorticity at any later time. These steps were implemented using the following subprograms: *setbc*, step (1); *zcntr*, step (2); *relaxd*, step (3); *cof*, step (4); *zbc*, step (5); and *triied* and *trijed*, step (6). The subprogram *triied* computes $F_{i,j}$, equation (6.8), and solves tridiagonal equations distributed over columns, equation (6.6), while *trijed* computes $G_{i,j}$, equation (6.9), and solves tridiagonal equations distributed over rows, equation (6.7).

6.2. Implementation on the MPP

The Navier-Stokes equations, described in section 6.1, were solved on the MPP using 128×128 grid points (127×127 cells). The computational cells are mapped onto the array so that each corner of the cell corresponds to a processor, as described in section 4.2. The seven subprograms required to solve these equations (see section 6.1) were written in MPP Pascal and run through the MPP Pascal compiler version 3 (this is the only experiment where this version of the compiler was used). These subprograms were executed entirely

on the MPP; only the input and output routines were run on the VAX. The four color relaxation scheme was implemented on the array, as in section 4.2. The tridiagonal systems, equations (6.6) and (6.7), were solved by the cyclic elimination algorithm, described in section 5.1, for all rows and all columns.

One of the problems in solving Navier-Stokes equations on the MPP is the size of the PE memory. As indicated in section 4.2, the relaxation procedure uses almost all of the 1024 bit PE memory. Although the staging memory can be used as a backup memory, this causes an I/O overhead and reduces the efficiency of the machine. This problem was solved by declaring all of the parallel arrays as global variables and using them by different procedures for more than one purpose. This means that temporary arrays were redefined in different parts of the code. Beside this hardware problem, there are problems in using MPP Pascal to perform vector operations and to extract elements of parallel arrays. Operations on vectors are performed on the MPP by expanding them to matrices and performing matrix operations. This means that vector processing rate is 1/128 of that for matrix operations. MPP Pascal does not permit extracting an element of a parallel array on the MPP. This means that scalar operations involving elements of parallel arrays need to be expanded to matrix operations or they should be performed on the VAX.

Table 18 contains the execution time for each subprogram of the Navier-Stokes algorithm, that for one iteration in the case of *relaxd;* the percentage of the total time spent in that subprogram; and the processing rate. The percentage of time spent in each routine determines which routines in the program are using the most time for a given run. It is clear, from Table 18, that the majority of the time was spent in *relaxd* for this particular run. This is because the average time step requires about 270 iterations and the total time spent in the other routines (*zcntr, cof, zbc, triied, trijed*) is only about the time to do two

iterations of *relaxd*. The number of iterations in *relaxd* per time step depends on the particular data used during a given run. A different input data set could result in a smaller number of iterations per time step and relatively less time spent in the relaxation routine.

Table 18. Measured execution time and processing rate of the Navier-Stokes subprograms for the 128 × 128 problem on the MPP.

Subprogram	Execution time (msec)	Percentage of time spent (%)	Processing rate (MFLOPS)
setbc	0.587	0.00	84
zcntr	2.694	0.06	24
relaxd	15.265*	99.23	156
cof	1.933	0.05	136
zbc	1.833	0.04	1.1
triied	12.717	0.31	125
trijed	12.725	0.31	125
overall#	41.597	100.00	155

* per iteration.

\# for ten time steps (execution time is in seconds for this row).

The processing rates in Table 18 are determined by counting only the arithmetic operations which truly contribute to the solution. Scalar and vector operations which were implemented as matrix operations are counted as scalar and vector operations. This is the reason why the routines *zbc* and *zcntr* have low processing rates; *zbc* has only vector operations while *zcntr* has some scalar operations implemented as matrix operations. The routine *setbc* has mostly scalar and data assignment operations which reduce its processing rate. Beside these three routines, the processing rate ranges from 125 to 155 MFLOPS with an average rate of about 140 MFLOPS. The processing rates for *relaxd, triied, trijed* are slightly less than the rates listed in Tables 2 and 9 because two versions of the MPP Pascal compiler, which call different sets of VAX format primitives, were used in these implementations (see table 1). This causes a difference in the processing rate of about 12% for the relaxation routine and about 7% for the tridiagonal solver.

Table 19 contains the estimated computation and communication times of the Navier-Stokes subprograms. These times are computed using equations (3.2) and (3.3) and VAX format primitives version 3 compiler. The cost of scalar operations is not included in this model; this explains the differences between the estimated and measured times for *setbc* and *cof*. The cost of the arithmetic operations is estimated the way these operations were implemented; i.e., scalar and vector operations (in *zcntr* and *zbc*) which were implemented as matrix operations are considered here as matrix operations. The amount of time spent on data transfers is quite modest for these subprograms; from 6% for *relaxd* to 25% for *triied* and *trijed*.

Table 19. Estimated times (in milliseconds) of the Navier-Stokes subprograms on the MPP.

Subprogram	Computation time	Communication time	Total estimated time	Measured time
setbc	0.300	-	0.300	0.587
zcntr	2.177	0.348	2.525	2.694
relaxd	13.592	0.840	14.432	15.265
cof	1.421	0.134	1.555	1.933
zbc	1.540	0.144	1.684	1.833
triied	9.239	3.043	12.283	12.717
trijed	9.239	3.043	12.283	12.725

6.3. Implementation on the Flex/32

The Navier-Stokes algorithm, described in section 6.1, was implemented on the Flex/32 using 64×64 grid points (63×63 cells) and 128×128 grid points (127×127 cells). The main program as well as the seven subprograms of the algorithm were written in Concurrent Fortran.

The parallel implementation of the Navier-Stokes algorithm is done by assigning a strip of the computational domain to a process and performing all the steps of the algorithm by each process. The main program performs only the input and output operations and creates and spawns the processes on specified processors. In our implementation, we used

1, 2, 4, 8, and 16 processors of the machine. The domain is decomposed first vertically for the first six subprograms of the algorithm (*setbc*, *zcntr*, *relaxd*, *cof*, *zbc*, and *triied*) and then horizontally for the subprogram *trijed*. The four color cell relaxation scheme was implemented as described in section 4.3. The tridiagonal systems were solved by the Gaussian elimination algorithm for both passes of the ADI method, as described in section 5.3.

Data is stored in the common memory, in the local memory of each processor, or in both of them. For the relaxation routine, as described in section 4.3, the matrices $P_{i,j}$, $Q_{i,j}$, $Z_{i,j}$, $\lambda_{i,j}$, and $C_{i,j}$ for each strip are stored in the local memory while the interface points and the error flags are stored in the common memory. A copy of the matrix $Q_{i,j}$ is also stored in the common memory to be used in computing the matrix $V_{i,j}$, equation (6.17). The vorticity at the corners of the cells ($\zeta_{i,j}$), the velocity field ($\vec{U}_{i,j}$), and the coefficients of the tridiagonal equations ($\alpha_{i,j}^{(x)}$, $\alpha_{i,j}^{(y)}$, $\beta_{i,j}^{(x)}$, $\beta_{i,j}^{(y)}$, $\gamma_{i,j}^{(x)}$, and $\gamma_{i,j}^{(y)}$) are all stored in the common memory; this is required in order to implement the ADI method. The forcing terms and the temporary matrices for the tridiagonal solvers (see section 5.3) are stored in the local memory of each processor.

In order to satisfy data dependencies between segments of the code running many processes, a counter is used. This counter, which is a shared variable with a lock assigned to it, can be incremented by any process and be reset by only one process. It is implemented as a "barrier" where all processes pause when they reach it. In addition, convergence flag and other flags, described in section 4.3, are used for synchronization in the relaxation routine.

Table 20 contains the speedups and efficiencies as functions of the number of processors for the 64×64 and 128×128 problems for two time steps. The measured execution times for ten time steps and processing rates for these problems using 16 processors are

listed in Table 21.

Table 20. Speedup and efficiency of the Navier-Stokes algorithm on the Flex/32.

Number of processors	64 × 64 points		128 × 128 points	
	speedup	efficiency	speedup	efficiency
1	1.000	1.000	1.000	1.000
2	1.959	0.980	1.976	0.988
4	3.893	0.973	3.941	0.985
8	7.715	0.964	7.850	0.981
16	15.027	0.939	15.483	0.968

Table 21. Measured execution times for ten time steps and processing rates for the Navier-Stokes algorithm using 16 processors of the Flex/32.

Problem size (grid points)	Execution time (sec)	Processing rate (MFLOPS)
64 × 64	268.7	1.09
128 × 128	2587.1	1.13

The performance of the Navier-Stokes algorithm is heavily influenced by the performance of the relaxation routine; about 98% of the total time was spent in this routine for the two time step run. The only difference between the implementation of the relaxation routine in this algorithm and in solving Cauchy-Riemann equations, section 4.3, is that here we used domains of sizes 63 × 63 and 127 × 127 cells while in section 4.3 we used domains of sizes 64 × 64 and 128 × 128 cells. This means that here the number of cells is not divisible by the number of processors used. This also means that the last processor has less work to do than the other processors. Therefore, the load distribution factor, equation (3.5), can be computed by

$$f_{ld} = \left\lceil \frac{n-1}{p} \right\rceil \left(\frac{p}{n-1} \right). \tag{6.26}$$

Using the performance model, developed in section 3.2, the overhead time represents at most 5% of the execution time of the algorithm. The overhead time of the relaxation routine dominates the total overhead time. As described in section 4.3, the spawning and syn-

chronization times are insignificant and the common memory access time dominates the overhead time. The other components of the common memory overhead time, T_{clm} and T_{cld}, have a negligible impact on the total overhead time.

6.4. Implementation on the Cray/2

The Navier-Stokes algorithm, described in section 6.1, was implemented on one processor of the Cray/2 using 64×64 and 128×128 grid points. The codes were written and run through the CFT/2 compiler. The four color cell relaxation scheme was implemented as described in section 4.4. The tridiagonal systems were solved by the Gaussian elimination algorithm for both passes of the ADI method, as described in section 5.4. The inner loops of all of the seven subprograms of the Navier-Stokes algorithm were fully vectorized.

Table 22 contains the execution time for each subprogram of the algorithm, the percentage of the total time spent in that subprogram, and the processing rate for the 64×64 and 128×128 problems. As described in section 6.2, most of the time was spent in *relaxd*. The average time step requires about 110 iterations for the 64×64 problem and about 270 iterations for the 128×128 problem. The subprogram *setbc* has a low processing rate because it has mostly memory access and scalar operations; however, this routine is called only once during the lifetime of the program. Beside this subprogram, the processing rate ranges from 57 to 97 MFLOPS with an average rate of about 70 MFLOPS for the subprograms of both problems. The processing rates for *triied* and *trijed* are slightly less than the rates for the ADI method for solving the diffusion equation (see Table 15) because here the parameters of the tridiagonal systems are matrices (see equations (6.6) and (6.7)) while in solving the diffusion equation these parameters are scalars (see equations (5.5) and (5.6)). This causes an increase of about 40% in the number of main memory access operations and a decrease of at least 16% in the method processing rate.

Table 22. Measured execution time and processing rate of the Navier-Stokes subprograms on the Cray/2.

Routine	64 × 64 grid points			128 × 128 grid points		
	Exec. time (msec)	Perc. of time (%)	Proc. rate (MFLOPS)	Exec. time (msec)	Perc. of time (%)	Proc. rate (MFLOPS)
setbc	0.480	0.02	25	1.651	0.01	29
zcntr	0.252	0.08	63	1.059	0.03	61
relaxd	2.719*	99.02	96	11.001*	99.60	97
cof	0.720	0.24	85	3.036	0.10	84
zbc	0.015	0.01	66	0.034	0.00	59
triied	1.007	0.33	57	4.014	0.13	59
trijed	0.928	0.30	62	3.870	0.13	62
overall#	3.048	100.00	96	30.286	100.00	97

* per iteration.

\# for ten time steps (execution times are in seconds for this row).

Tables 23 and 24 contain the estimated times of the Navier-Stokes subprograms for the 64 × 64 and 128 × 128 problems. These times are obtained using equations (3.13) to (3.16). The main memory access time for each subprogram represents about 50% to 70% of the total estimated time and the measured time. The difference between the total estimated values and the measured values can be contributed to several reasons. Among these reasons are: the memory access and arithmetic operations can overlap, specially for large routines; the time to perform scalar operations is not included in this model; and, as mentioned before, there is up to 20% offset on the results depending on the memory traffic and the number of the active processes on the system.

Table 23. Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 64 × 64 problem on one processor of the Cray/2.

Subprogram	Memory access time	Addition time	Multiplication time	Total estimated time	Measured time
setbc	0.277	0.022	0.089	0.388	0.480
zcntr	0.154	0.067	0.022	0.243	0.252
relaxd	1.783	1.206	0.551	3.540	2.719
cof	0.480	0.173	0.173	0.826	0.720
zbc	0.010	0.002	0.006	0.018	0.015
triied	0.510	0.130	0.281	0.921	1.007
trijed	0.510	0.130	0.281	0.921	0.928

Table 24. Estimated and measured execution times (in milliseconds) of the Navier-Stokes subprograms for the 128 × 128 problem on one processor of the Cray/2.

Subprogram	Memory access time	Addition time	Multiplication time	Total estimated time	Measured time
setbc	1.120	0.090	0.360	1.570	1.651
zcntr	0.622	0.270	0.090	0.982	1.059
relaxd	7.218	4.144	1.802	13.164	11.001
cof	1.967	0.711	0.711	3.389	3.036
zbc	0.019	0.004	0.013	0.036	0.034
triied	2.090	0.533	1.155	3.778	4.014
trijed	2.090	0.533	1.155	3.778	3.870

CHAPTER SEVEN

Comparisons and Concluding Remarks

The architectures, the performance models, and the implementation of the three algorithms on the architectures along with the results were presented in previous chapters. A comparison of the performance of these architectures using the three algorithms is presented in this chapter. Some concluding remarks are also given here.

There are a number of measures that one can use to compare the performance of these parallel computers using a particular algorithm. One is the processing rate and another is the execution time. However it must be borne in mind that both of these measures depend on the architectures of the computers, the overhead required to adapt the algorithm to the architecture, and the technology, that is, the intrinsic processing power of each of the computers.

If we consider a single sized problem for all algorithms, that on a 128×128 grid, then, as shown in Table 25, the processing rate is a maximum for the MPP using the three algorithms; the range of the processing rate for these algorithms is 134 to 175 MFLOPS for the MPP, 74 to 102 MFLOPS for the Cray/2, and only 0.66 to 1.13 MFLOPS on 16 processors of the Flex/32. The low processing rates of these algorithms on the 16 processors of the Flex/32 are simply due to the fact that the National Semiconductor 32032 microprocessor and 32081 coprocessor are not very powerful. Although these algorithms have higher performance rates on the MPP than on the Cray/2, it takes less time to solve these problems on

the Cray/2 than on the MPP. For the relaxation algorithm, this is due to the algorithm overhead involved in adapting the algorithm to the MPP; for each iteration the MPP algorithm has 145 arithmetic operations per grid point compared to 66 operations per grid point on the Cray/2. For the ADI method, the Cray/2 outperformed the MPP with respect to the execution time because the cyclic elimination algorithm, used to solve tridiagonal systems on the MPP, has 97 arithmetic operations per grid point (see Table 10) while the Gaussian elimination algorithm, used on the Cray/2, has only 15 operations per grid point (see Table 16); this is a factor of about 6.5 which makes cyclic elimination the less efficient algorithm. The performance of the Navier-Stokes algorithm is heavily influenced by the performance of the relaxation algorithm; at least 98% of the execution time was spent in the relaxation routine.

Table 25. Summary of the results for the 128 × 128 problem.

Computer	Cauchy-Riemann eqs.		Diffusion eq.		Navier-Stokes eqs.	
	Exec. time (msec)	Processing rate (MFLOPS)	Exec. time (msec)	Processing rate (MFLOPS)	Exec. time (sec)	Processing rate (MFLOPS)
MPP	13.56	175	23.70	134	41.60	155
Flex/32	964.85	1.12	740.00	0.66	2587.10	1.13
Cray/2	10.47	102	6.47	74	30.29	97

The implementation of these algorithms on the Flex/32 has the same number of arithmetic operations per grid point as on the Cray/2; there is only a reordering of the calculations and no additional arithmetic operations in the overhead. The algorithmic overhead for the Flex/32 versions is the cost of exchanging the values of the interface points and setting the synchronization counters for the relaxation algorithm and accessing the common memory for the ADI method. This means that the code on each processor is the serial code plus the overhead code. When the code is run on one processor, it is just the serial code with the overhead portion removed.

These two performance measures can also depend on problem size. For the 128×128 problem, the times to complete one iteration of the relaxation algorithm on the Cray/2 and the MPP are comparable. However, for problems which are larger than the MPP array there is a very large overhead cost because the MPP array memory is not large enough to contain the data and the staging memory must be used. Thus expanding the computational domain caused a degradation on the performance of the MPP for the relaxation algorithm. However, expanding the domain has minor impact on the performance of both the Cray/2 and Flex/32 for the three algorithms. In fact the processing rate of the Cray/2 slightly increased as the problem size increased.

Comparing the measured processing rate with the peak processing rate of an algorithm on each of the computers is also a measure of how well the algorithm has been mapped onto the architecture. This measure is also relatively independent of the basic technology of the implementation of the architecture. For the relaxation algorithm, these relative performance rates are 40% for the Cray/2, 48% for the MPP, and at least 96% for the Flex/32. For the ADI method, these rates are 28% for the Cray/2, 57% for the MPP, and between 64% and 94% for the Flex/32. Accessing the main memory is the major problem on the Cray/2. The poor performance of the VAX format primitives is the major cause of inefficiency on the MPP. If the VAX format primitives were as efficient as IBM format primitives (see Table 1), the performance of the MPP for the relaxation algorithm, for example, could be 315 MFLOPS, about 72% of the peak processing rate. Finally, we have found no major overhead on the Flex/32, except for small problems where the spawning cost could be a factor.

Another measure of performance is the number of machine cycles required to solve a problem. This measure reduces the impact of technology on the performance of the

machine. As shown in Table 26, the MPP outperformed the Cray/2 (by a factor of 7 to 19) and the latter outperformed the Flex/32 (by a factor of about 4) in this measure. This means that one processor of the Cray/2 outperformed 16 processors of the Flex/32 even if we assume that both machines have the same clock cycle. The problem with the Flex/32 is that, although each processor has a cycle time of 100 nsec, the memories (local and common) have access times of about 1 μsec.

Table 26. Clock cycles (in million cycles) for the 128×128 problem.

Computer	Cauchy-Riemann eqs.	Diffusion eq.	Navier-Stokes eqs.
MPP	136	237	415970
Flex/32	9649	7400	25871000
Cray/2	2554	1579	7386859

One simple comparison between the MPP and Cray/2 is the time to perform a single arithmetic operation using the models developed in sections 3.1 and 3.3. Using equation (3.13), the time to perform a single floating point operation (addition or multiplication) on an array of size 128×128 elements on the Cray/2, excluding the memory access cost, is 91.3 μsec. The time to perform the same operation on the MPP using the MPP Pascal version 3 compiler ranges from 81.1 μsec (for multiplication) to 96.5 μsec (for addition). This shows that the processing power of a single functional unit of the Cray/2 is comparable to the processing power of 16384 processors of the MPP. However, much of the overhead is not included in this comparison: memory access cost on the Cray/2, data transfers on the MPP, and so on.

The experiment of solving the Cauchy-Riemann equations showed that by reordering the computations we were able to implement the algorithm on three different architectures with no major modifications. The experiment of solving the diffusion equation showed that two different algorithms, Gaussian elimination and cyclic elimination, were used to solve

the same numerical problem on the three architectures. These two algorithms were chosen to exploit the parallelism available on these architectures. The three algorithms also exploit multiple granularities of parallelism. These algorithms vectorized quite well on the Cray/2. A fine grained parallelism, involving sets of single arithmetic operations executed in parallel, is obtained on the MPP. Parallelism at higher level, large grained, is exploited on the Flex/32 by executing several program units in parallel.

The performance model on the MPP was fairly accurate on predicting the execution times of the algorithms when we used the measured times of the VAX format primitives. The performance model on the Flex/32 showed the impact of the common memory access and spawning overheads on the performance of these algorithms. The performance model on the Cray/2 was based on predicting the execution costs of separate operations. This model is used to identify the major costs of these algorithms and reproduced the measured results with an error of at most 30%.

These experiments showed the impact of software on the performance of parallel machines. The MPP has two sets of primitives and the ratio of execution times of these primitives ranges from 1.07 for multiplication to 2.53 for addition (see Table 1). The Flex/32 has two sets of locks to lock and unlock variables in the common memory. There was an increase of about 39% in the efficiency of the relaxation algorithm when the MMOS locks were replaced by the Local locks (see Tables 4 and 5). The performance of register-to-register vector computers, like the Cray/2, is strongly compiler dependent. The compiler is responsible for overlapping the addition, multiplication and memory access operations.

The ease and difficulty in using a machine is always a matter of interest. The Cray/2 is relatively easy to use as a vector machine. Existing codes that were written for serial machines can always run on vector machines. Vectorizing the unvectorized inner loops will

improve the performance of the code. Unlike parallel machines, vector machines do not have the problem of "either you get it or not". The Flex/32 is not hard to use, except for the unavailability of debugging tools which is a problem for many MIMD machines (a synchronization problem could cause a program to die). On the other hand, the MPP is not a user-friendly system. The size of the PE memory is almost always an issue. MPP Pascal does not permit vector operations on the array nor does it allow extraction of an element of a parallel array. The MCU has 64 Kbytes of program memory. This memory can take up to about 1500 lines of MPP Pascal code. This means that larger codes can not run on the MPP. Finally, input/output is somewhat clumsy on the MPP. However, other machines with architectures similar to the MPP may not have the same problems that the MPP does.

There is one further observation of interest. These algorithms can be implemented concurrently on four processors of the Cray/2 (multitasking). The codes will be similar to the Flex/32 versions except that all of the variables should be stored in the main memory; the local memory on the Cray/2 is used only to store scalar temporaries. Adapting these algorithms to a local memory multiprocessor with a hypercube topology should be relatively easy. A high efficiency is predicted in this case because all data transfers are to nearest neighbors and their cost should be very small compared to the computation cost.

List of References

[1] Adams, L., "Reordering Computations for Parallel Execution," Comm. Appl. Numer. Meths., Vol. 2, No. 3, May-June 1986, pp. 263-272.

[2] Amdahl, G. M., "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," AFIPS Conf. Proc., Vol. 30, Washington, DC, 1967, pp. 483-485.

[3] Ames, W. F., "Numerical Methods for Partial Differential Equations," 2nd ed., Academic Press, New York, 1977, pp. 251-255.

[4] Ames Research Center, "NAS User Guide," Version 3.0, Moffett Field, CA, 1987.

[5] Batcher, K. E., "Design of a Massively Parallel Processor," IEEE Trans. Computers, Vol. C-29, Sept. 1980, pp. 836-840.

[6] Bokhari, S. H., "Multiprocessing the Sieve of Eratosthenes," Computer, Vol. 20, No. 4, April 1987, pp. 50-58.

[7] Cray Research Inc., "Fortran (CFT2) Reference Manual," Publication SR-2007, Mendota Heights, MN, 1986.

[8] Crockett, T. W., "Performance of Fortran Floating-Point Operations on the Flex/32 Multicomputer," ICASE Interim Rep. No. 4, NASA Langley Research Center, Hampton, VA, August 1987.

[9] Dongarra, J. J., "Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment," Tech. Memo. 23, Argonna National Lab., Argonna, IL, Nov. 1985.

[10] Flexible Computer Co., "Flex/32 Multicomputer System Overview," Publication No. 030-0000-002, 2nd ed., Dallas, TX, 1986.

[11] Gallopoulos, E. J., "The Massively Parallel Processor for Problems in Fluid Dynamics," Computer Physics Communications, No. 37, 1985, pp. 311-315.

[12] Gannon, D. B. and Van Rosendale, J., "On the Impact of Communication Complexity on the Design of Parallel Numerical Algorithms," IEEE Trans. Computers, Vol. C-33, No. 12, Dec. 1984, pp. 1180-1194.

[13] Gatski, T. B., Grosch, C. E., and Rose, M. E., "A Numerical Study of the Two-Dimensional Navier-Stokes Equations in Vorticity-Velocity Variables," J. Comput. Phys., Vol. 48, No. 1, 1982, pp. 1-22.

[14] Goddard Space Flight Center, "MPP Pascal Programmer's Guide," Greenbelt, MD, June 1987.

[15] Goodyear Aerospace Co., "General Description of the MPP," Tech. Report GER-17140, Akron, OH, April 1983.

[16] Grosch, C. E., "Performance analysis of Poisson solvers on array computers," in Infotech State of the Art Report: Supercomputers, Jesshope C. R. and Hockney R. W., ed., 2, Infotech International, Maidenhead, England, 1979, pp. 147-181.

[17] Grosch, C. E., "Adapting a Navier-Stokes code to the ICL-DAP," SIAM J. Scientific & Statistical Computing, Vol. 8, No. 1, 1987, pp. s96-s117.

[18] Heller, D., "A Survey of Parallel Algorithms in Numerical Linear Algebra," SIAM Review, Vol. 20, No. 4, 1978, pp. 740-777.

[19] Hockney, R. W. and Jesshope, C. R., "Parallel Computers: Architecture, Programming and Algorithms," Adam Hilger, Bristol, England, 1981.

[20] Kaczmarz, S., "Angenaherte auflosung von systemen linearer gleichungen," Bull. Acad. Polon, Sci Lett. A, 1937, pp. 355-357.

[21] Ortega, J. M. and Voigt, R. G., "Solution of Partial Differential Equations on vector and Parallel Computers," SIAM Review, Vol. 27, No. 2, June 1985, pp. 149-240.

[22] Parkinson, D. and Liddell, H. M., "The Measurement of Performance on a Highly Parallel System," IEEE Trans. Computers, Vol. C-32, No. 1, Jan. 1983, pp. 32-37.

[23] Philips, R. B. and Rose, M. E., "Compact Finite-Difference Schemes for Mixed Initial-Boundary Value Problems," SIAM J. Numerical Analysis, Vol. 19, No. 4, 1982, pp. 698-720.

[24] Reeves, A. P., "Parallel Pascal: An Extended Pascal for Parallel Computers," J. Parallel & Distributed Computing, Vol. 1, 1984, pp. 64-80.

[25] Rose, M. E., "A 'Unified' Numerical Treatment of the Wave Equation and the Cauchy-Riemann Equations," SIAM J. Numerical Analysis, Vol. 18, No. 2, 1981, pp. 372-376.

[26] Tanabe, K., "Projection Method for Solving a Singular System of Linear Equations and its Applications," Numer. Math., Vol. 17, 1971, pp. 203-214.

[27] Voigt, R. G., "Where are the Parallel Algorithms?," 1985 National Computer Conference Proceedings, AFIPS Press, pp. 329-334.

Autobiographical Statement

The author was born in Mosul, Iraq on July 8, 1954. He received his B.S. degree in Electrical Engineering from Mosul University, Mosul, Iraq, in 1976 and his M.S. degree in Computer Engineering from Syracuse University, Syracuse, New York, in 1982. He worked as an Electrical Engineer with the Organization of Technical Industrial, Baghdad, Iraq, from 1977 to 1980.

The author was granted a full scholarship from the Iraqi government for his M.S. study. Also, he was awarded one year doctoral fellowship from Old Dominion University. He is a member of Phi Kappa Phi and the IEEE Computer Society.