NASA Contractor Report 178307

ICASE REPORT NO. 87-34

ICASE

SOLVING THE CAUCHY-RIEMANN EQUATIONS ON PARALLEL COMPUTERS

(NASA-CR-178307) SOLVING THE CAUCHY-RIEMANN EQUATIONS ON PARALLEL COMPUTERS Final Report (NASA) 38 p Avail: NTIS HC A03/MF A01 CSCL 09B

N87-24107

Unclas G3/61 0077637

Raad A. Fatoohi

Chester E. Grosch

Contracts No. NAS1-17070, NAS1-18107
May 1987

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

National Aeronautics and Space Administration

Langley Research Center
Hampton, Virginia 23665

SOLVING THE CAUCHY-RIEMANN EQUATIONS ON PARALLEL COMPUTERS

Raad A. Fatoohi and Chester E. Grosch

Old Dominion University

Norfolk, VA 23508

and

Institute for Computer Applications in Science and Engineering

NASA Langley Research Center, Hampton, VA 23665

ABSTRACT

In this paper we discuss the implementation of a single numerical algorithm on three parallel/vector computers. The algorithm is a relaxation scheme for the solution of the Cauchy-Riemann equations; a set of coupled first order partial differential 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 CRAY/2, an MIMD machine with four vector processors. The machine architectures are briefly described. The implementation of the algorithm is discussed in relation to these architectures and measures of the performance on each machine are given. Simple performance models are used to describe the performance. These models highlight the bottlenecks and limiting factors for this algorithm on these architectures. Finally conclusions are presented.

This research was supported by the National Aeronautics and Space Administration under NASA Contracts No. NAS1-17070 and NAS1-18107 while the authors were in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

1. Introduction.

It appears that single processor computers, whether scaler 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. A reduction of the clock period to one nanosecond seems possible within the next decade. 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.

If the parallel computers can be used effectively, very large gains in overall processing power are possible. The major difficulty in obtaining this increase in processing power lies not in hardware development, but in programming development. Concurrent processing is a relatively new and largely unexplored research area. There have been a substantial number of theoretical studies of the performance of algorithms on parallel computers but far fewer actual experimental studies [10], [13].

There are three conditions which must be met if an algorithm is to execute at high efficiency on a concurrent processor: (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. Understanding the execution behavior of one class of algorithms is one of the main issues in this work.

The problem of measuring the performance of parallel computers is a difficult one and, as yet, does not have a solid theoretical foundation [14]. Performance is highly dependent on the architecture of the multiprocessor, the computational algorithm, and 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 [5], [8]. 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 [9] in adapting Navier-Stokes code to the ICL-DAP and it will be adapted in this work.

In this paper we describe the implementation of a single numerical method, a relaxation scheme for the solution of the Cauchy-Riemann equations, on three different parallel architectures. These architectures are: the MPP, an SIMD machine with 16K serial 1-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 algorithm, described in [6], [9], is briefly described in section 2. The implementation on the three computers is described in sections 3 through 5; each section contains a brief description of the machine architecture, the programming language, the implementation, the results, and a simple performance model to describe the execution behavior of the algorithm. Finally, section 6 contains a comparison of performance at the problem solving level and some concluding remarks.

2. *The Algorithm*

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

$$u_x + v_y = 0, \tag{2.1}$$

$$v_x - u_y = \zeta. \tag{2.2}$$

These equations arise as part of Navier-Stokes equations for the two-dimensional, time dependent flow of a viscous incompressible fluid, where $\vec{u} = (u,v)$ is the velocity and ζ is the vorticity. In many cases these equations are reduced to a single second order differential equation, a Poisson equation, by the introduction of a stream function. There can be substantial difficulties with the second order formulation. For example, if the grid is nonuniform a five point stencil is not second order accurate. For this reason, and others, it may be desirable to treat the first order system directly. The numerical method used to approximate these equations is based on compact differencing schemes developed by Rose [17] and Philips and Rose [15]. Gatski, et al. [6] applied these schemes to solve the Navier-Stokes equations in terms of the vorticity and velocity. Grosch [9] adapted the Navier-Stokes code to the ICL-DAP. The method is briefly described here for the sake of completeness.

Consider approximating the solution of equations (2.1) and (2.2) in a rectangular domain on whose boundary one component of the velocity 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. 1. 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:

$$\delta_x U_{i,j} \equiv \frac{(U_{i+1/2,j} - U_{i-1/2,j})}{\Delta x}, \tag{2.3}$$

$$\mu_x U_{i,j} \equiv \frac{(U_{i+1/2,j} + U_{i-1/2,j})}{2}. \tag{2.4}$$

Suppose that $\zeta_{i,j}$ is prescribed. Then equations (2.1) to (2.2) are approximated by,

$$\delta_x U_{i,j} + \delta_y V_{i,j} = 0, \tag{2.5}$$

$$\delta_x V_{i,j} - \delta_y U_{i,j} = \zeta_{i,j}, \tag{2.6}$$

$$\mu_x U_{i,j} - \mu_y U_{i,j} = 0, \tag{2.7}$$

$$\mu_x V_{i,j} - \mu_y V_{i,j} = 0. \tag{2.8}$$

The adaptation of this algorithm to different parallel architectures can be simplified by the introduction of box variables to represent the velocity field. The center of a cell is at (i,j). The box variables, $\vec{P}$, are defined at the corners of the cells, points ($i\pm1/2$,$j\pm1/2$). They are related to the velocity $\vec{U}$ by:

$$\vec{U}_{i,j\pm1/2} = \frac{(\vec{P}_{i+1/2,j\pm1/2} + \vec{P}_{i-1/2,j\pm1/2})}{2}, \tag{2.9}$$

$$\vec{U}_{i\pm1/2,j} = \frac{(\vec{P}_{i\pm1/2,j+1/2} + \vec{P}_{i\pm1/2,j-1/2})}{2}. \tag{2.10}$$

It is easy to see that equations (2.7) and (2.8) are satisfied identically for any set of box variables. For the cell (i,j), equations (2.5) and (2.6) become,

$$AP = Z, \tag{2.11}$$

where

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

$$P = (\vec{P}_{i+1/2,j-1/2}, \vec{P}_{i+1/2,j+1/2}, \vec{P}_{i-1/2,j+1/2}, \vec{P}_{i-1/2,j-1/2})^T,$$

$$Z = (0, 2(\Delta y)_i \zeta_{i,j})^T,$$

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

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

Equation (2.11) is solved by an iteration scheme which was originally proposed by Kaczmarz [11] and was generalized by Tanabe [18]. If $P^{(k)}$ is the value after the k'th iteration, then the residual after the k'th iteration, $R^{(k)}$, is given by:

$$R^{(k)} = AP^{(k)} - Z. \tag{2.12}$$

The next iteration is

$$P^{(k+1)} = P^{(k)} - \omega A^T (AA^T)^{-1} R^{(k)}, \tag{2.13}$$

where ω is an acceleration parameter. This relaxation scheme is equivalent to an SOR method. On a serial computer the array of computational cells is swept over, applying equation (2.13) 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], [19]; operations are reordered in order to increase the percentage of the computation that can be done in parallel. As shown in Fig. 2, 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. 2) 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 necessary 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.

In brief, the relaxation algorithm is implemented by computing the residuals, $R^{(k)}$, using equation (2.12) for each set of cells, followed by updating the $P's$ using equation (2.13). 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. Fig. 3 shows the Fortran code used to compute the residuals, $R^{(k)}$ = (r1, r2), and update the $P's$. The P, Q matrices are the first and the second components of the box variables, the Z matrix is defined by equation (2.11), the ZL matrix contains the values of $\lambda_{i,j}$, and C is a matrix of coefficients defined by:

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

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 (2.12) and (2.13) are solved with given $\zeta_{i,j}$ 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. 4. 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.

3. Implementation on the MPP

3.1. 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 [2], [7]. 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. 5. 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 scaler arithmetic, and shifts data across the PE array. Items of data are sent to the ARU and taken from the ARU 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 32-bit 128×128 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 [12] is a machine-dependent language which has evolved directly from the language Parallel Pascal defined by Reeves [16]. 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. 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).

3.2. Implementation

The computational cells, as described in section 2, 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 $\vec{P}_{i-1/2,j+1/2}$, $\zeta_{i,j}$, 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 Z, ZL, and C matrices, calculating the boundary conditions, and initialization of the P and Q 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 by masking out the others. 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.

(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 arrays of floating point numbers, all but 5 of which are temporary, and 19 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, Q, Z, ZL, and C. 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, (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. 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.

3.3. Results and Discussions

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 scaler 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 I 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.

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.

The execution time for one iteration of the 128×128 problem, T, is computed as follows:

$$T = T_{comp} + T_{comm},$$

$$T_{comp} = t_c \, (N_a \, C_a + N_m \, C_m),$$

$$T_{comm} = t_c \, (N_{s1} \, C_{s1} + N_{s2} \, C_{s2}),$$

where

T_{comp} : Computation cost,

T_{comm} : Communication cost,

t_c : Machine cycle time = 100 nanoseconds,

N_a : Number of additions per iteration = 119,

N_m : Number of multiplications per iteration = 26,

N_{s1} : Number of one step shifts per iteration = 42,

N_{s2} : Number of two step shifts per iteration = 21,

C_a : Number of cycles required to add two arrays of floating point numbers,

C_m : Number of cycles required to multiply two arrays of floating point numbers,

C_{s1} : Number of cycles required to shift a floating point array by one step,

C_{s2} : Number of cycles required to shift a floating point array by two steps.

Table II contains the estimated values of C_a, C_m, C_{s1}, and C_{s2} 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 peak performance rates of the arithmetic operations are computed based on the IBM format primitives. The VAX format primitives were programmed at NASA Goddard and currently used in the MPP Pascal programs. 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 1.15 for multiplication to 2.16 for addition.

The computation and communication costs of the relaxation algorithm are listed in Table III using both sets of primitives. We used the VAX format primitives 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 the 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 scaler 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.

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. Implementation on the Flex/32

4.1. The Flex/32 architecture and Concurrent Fortran

The Flex/32 is an MIMD shared memory multiprocessor based on 32 bit National Semiconductor 32032 processor [3]. The Flex/32 cabinet can hold up to 20 of any combination of processor and memory cards. The results presented here were obtained using the 20 processor machine that is now installed at NASA Langley Research Center.

As shown in Fig. 6, 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, 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. Among the MMOS processors only processor 3 has a terminal attached to it.

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 will wait until the lock is released.

process define and start the execution of a process 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.

4.2. Implementation

The four color cell relaxation scheme, as described in section 2, 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 box variables, the P and Q matrices, the vorticity, the Z matrix, and the matrices of coefficients, ZL and C, 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 for 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.

4.3. Results and Discussion.

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, the efficiency is determined 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 IV and V. The execution time for one iteration and the processing rate for both problems using 16 processors and local locks are listed in Table VI.

The results shown in Table IV were obtained using the MMOS locks and the results shown in Table V 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 IV and V, 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.

The total cost of implementing the relaxation algorithm on p processors using the Local locks is the sum of the computation cost and the overhead cost. Since the load is distributed evenly between the processors with no extra computations, the computation cost can be computed by dividing the execution time using a single processor by the number of processors.

The overhead cost is

$$T_{ovr} = T_{spn} + T_{cma} + T_{syn},$$

where

T_{spn} is the total spawning cost,

T_{cma} is the total common memory access cost per iteration,

T_{syn} is the total synchronization cost per iteration.

These costs can be estimated as follows:

$$T_{spn} = P\ t_{spn},$$

$$T_{cma} = \alpha(P)\ N\ K\ t_{cma},$$

$$T_{syn} = P\ t_{syn},$$

where

P is the number of processors,

N is the number of interface points (N is 65 for first problem and 129 for second problem),

K is the number of times the interface points are referenced for each iteration (K = 8),

t_{spn} is the time to spawn one process; a reasonable value is 13 milliseconds,

t_{cma} is the time to access a location in common memory; a reasonable value is 6 microseconds,

t_{syn} is the time to synchronize one process within p processes for each iteration,

$\alpha(P)$ is the common memory contention factor; it is a function of P.

The overhead cost represents at most 4% of the total cost of the algorithm. The spawning cost has a minor impact because the processes are spawned only once at the beginning of the program. The synchronization cost was insignificant because the routines that provide the locking mechanism are very efficient and overlap with the memory access. It was found that the contention factor ranged from 2.6 to 13.4. 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 large number of processors.

5. Implementation on the Cray/2.

5.1. 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 Cray/1, see [10], with the addition of 16K words of local memory available for storage of vector and scaler data. Within each CPU there are eight vector registers (64 words each), eight scaler 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) [4] 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 subscripts 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 rewritting 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 Cray's 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 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.

5.2. Implementation.

The relaxation algorithm, described in section 2, 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 since each of these columns contains a disjoint set of cells. The implementation was quite simple. The serial version of the algorithm, using the reordered form of the algorithm and written in standard Fortran, was transferred and run through CFT/2 compiler. Not all inner loops were vectorized, but some measure of vector performance was obtained. 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.

The use of the main memory can be reduced by using scaler temporaries, instead of array temporaries, within inner DO loops. This causes CFT/2 to store these variables in the local memory. The residuals, see Fig. 3, are stored in scaler temporaries.

5.3. Results and Discussion.

The measured scaler rate of the 64×64 problem is 30 MFLOPS. This rate is obtained when the serial version of the algorithm is used. This result shows that existing codes, written for serial

machines, produce modest performance when they are transferred to Cray/2.

Table VII 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 (3.3). As the number of grid points is increased, the processing rate is slightly improved. This is due to the fact that more overlapping between different operations is expected for large problems.

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 writting 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, more than 50% slowdown in data transfer rate, and about 30% slowdown in the overall algorithm processing rate.

Because the relaxation routine has more additions than multiplications, the time to complete one iteration can be considered as a summation of two portions; a portion with one operational pipeline and a portion with two operational pipelines. These portions can be estimated by counting the number of additions and multiplications separately. If the peak processing rate of one pipeline is estimated to be 244 MFLOPS, ignoring the vector startup times, the maximum possible rate of the relaxation algorithm will be 350 MFLOPS. Therefore, the measured processing rate for the 128×128 problem is about 29% of the peak processing rate. If the startup time of the vector functional units is included, the measured processing rate for the 128×128 problem will be about 40%

of the peak processing rate of 257 MFLOPS.

The relaxation algorithm has two main costs, the computation cost and the memory access cost. To estimate these costs, 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 floating point functional units, 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 are assumed here. Competition for memory banks from other processors causes lower transfer rates. Their real values are hard to estimate.

The relaxation subroutine has two major parts, performing the relaxation kernel, see Fig. 3, and computing the maximum residual, excluding the cost of the ISMAX routine. These two parts contribute more than 90% of the total cost of the algorithm. The costs of performing these two parts for one iteration are computed as follows:

a. Implementing the relaxation kernel,

$$T_m = \left\{ \left\lceil \frac{N_c - 1}{128} \right\rceil L_m + \frac{N_c - 1}{2} R_2 \right\} \left(\frac{N_c - 1}{2} N_m N_s \right) CP, \tag{5.1}$$

$$T_f = \left\{ \left\lceil \frac{N_c - 1}{128} \right\rceil L_f + \frac{N_c - 1}{2} \right\} \left(\frac{N_c - 1}{2} N_f N_s \right) CP, \tag{5.2}$$

b. Computing the maximum residual,

$$T_m = \left\{ \left\lceil \frac{N_c}{64} \right\rceil L_m + N_c R_1 \right\} (N_m N_c) CP, \tag{5.3}$$

$$T_f = \left\{ \left\lceil \frac{N_c}{64} \right\rceil L_f + N_c \right\} (N_f N_c) CP, \tag{5.4}$$

where

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

T_m : Total main memory access cost per iteration,

T_f : Total floating point operation cost per iteration,

N_c : Number of cells in each dimension,

N_m : Number of memory access operations, 23 for part a and 12 for part b,

N_f : Number of floating point operations, 31 additions and 18 multiplications for part a and 15 additions and 2 multiplications for part b,

N_s : Number of sets (colors) = 4.

Table VIII contains the results of applying equations (5.1) through (5.4) for different problem sizes. Also, the measured times are included in the table for comparison. The main memory access cost represents about 50% of the total estimated cost and at least 60% 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 arithmetic 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.

6. Comparisons and Concluding Remarks.

The processing rate and execution time (see tables I, VI, and VII) have been used to compare the performance of the three architectures for the execution of this relaxation algorithm. Comparing the measured processing rate with the peak processing rate of the algorithm is a useful measure of how well the algorithm has been mapped onto the architecture. These relative performance rates are 48% for the MPP and 40% for the Cray/2. The poor performance of the VAX format primitives is the major cause of inefficiency on the MPP, while accessing the main memory is the major problem on the Cray/2. On the other hand, there is no major overhead on the Flex/32 where an efficiency of at least 96% is obtained even for large number of processors.

A different way of measuring performance is the time taken to solve the problem. Although the algorithm has higher performance rate on the MPP than on the Cray/2, it takes less time to solve the problem on the Cray/2 than on the MPP. This is due to the algorithm overhead involved in adapting the method to the MPP; for each iteration the MPP algorithm has 145 arithmetic operations compared to 66 operations on the Cray/2. For the 128×128 problem, the times to complete one iteration on both machines are comparable. However, for oversize problems, the Cray/2 outperforms the MPP because of the overhead for transferring the data to and from the staging memory. Also, it should be realized that the arithmetic on the Cray/2 is based on 64 bit words while on both the MPP and Flex/32 it is based on 32 bit words.

Another measure of performance is the number of machine cycles required to solve the problem. This measure reduces the impact of technology on the performance of the machine. For the 128×128 problem, for example, each iteration requires 135.6 million cycles on the MPP; 2553.7 million cycles on the Cray/2; and 9648.5 million cycles on 16 processors of the Flex/32 (129×129 grid points was used on the Flex/32).

Expanding the computational domain caused a degradation on the performance of the MPP because of the size of the local memory. However, expanding the domain has minor impact on the performance of both the Cray/2 and Flex/32. Also, it should be realized that the Cray/2 can be

used to solve problems in very large domains because of the large memory available.

This experiment showed that by reordering the computations we were able to implement the algorithm on three different architectures with no major modifications. Also, the algorithm exploits multiple granularities of parallelism. A fine grained parallelism, involving sets of single arithmetic operations executed in parallel, is obtained on the MPP and Cray/2. Parallelism at higher level, large grained, is exploited on the Flex/32 by executing several program units in parallel. Adapting this algorithm 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.

The performance model on the MPP was fairly accurate on predicting the execution times of the relaxation algorithm when we used the measured times of the VAX format primitives. The performance model on the Flex/32 showed that the common memory access cost dominated the overhead cost; although, the overhead cost was less than 5% of the total cost of the algorithm. 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 the algorithm and not to reproduce the measured results. To develop more accurate models of the performance of these machines, a full understanding of the assembly languages may be needed.

In summary we have found that the four color relaxation scheme can be adapted reasonably well to the three different architectures. On the MPP, the processing power of the machine has been fully utilized because the code consists of mostly array operations and data transfers to nearest neighbors. On the Flex/32, the scheme is easily implemented and a speedup of 15.44 on 16 processors was obtained. On the Cray/2, all inner loops were vectorized and processing rate of over 100 MFLOPS was achieved.

Acknowledgements

We would like to thank NASA Goddard Space Flight Center, NASA Ames Research Center, and NASA Langley Research Center for allowing us access to the MPP, Cray/2, and Flex/32. We are grateful to Tor Opsahl of Science Applications Research, Dan Nagle of Cray Research, Inc., and Tom Crockett of NASA Langley Research Center for their help in using the machines. We are also grateful to Dr. R. G. Voigt for his continuing support and for a critical reading of the manuscript.

References

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

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

[3] Flex/32 Multicomputer System Overview, Flexible Computer Co., 1986.

[4] Fortran (CFT2) Reference Manual, Cray Research Inc. Publication SR-2007, 1986.

[5] 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.

[6] 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.

[7] General Description of the MPP, Goodyear Aerospace Co., Tech. Report GER-17140, April 1983.

[8] 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 (1979), Infotech International, pp. 147-181.

[9] 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.

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

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

[12] MPP Pascal Programmer's Guide, NASA Goddard Space Flight Center, Sept. 1986.

[13] Ortega, J. M. and Voigt, R. G., "Solution of Partial Differential Equations on vector and Parallel Computers," SIAM Rev., 27 (1985), pp. 149-240.

[14] 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.

[15] 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.

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

[17] 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.

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

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

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

Table I. Execution times for one iteration and processing rates for the relaxation algorithm on the MPP.

Operation	IBM format primitives	VAX format primitives
addition	381	824
multiplication	758	877
one step shift	96	166
two step shift	128	198

Table II. Estimated execution times (in machine cycles) of the elementary operations on the MPP.

Primitives	Computation time	Communication time	Measured time
IBM format	6.50	0.67	
VAX format	12.09	1.11	13.56

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

Number of processors	65×65 points		129×129 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 IV. Speedup and efficiency on the Flex/32 using the MMOS locks.

Number of processors	65×65 points		129×129 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 V. Speedup and efficiency on the Flex/32 using the Local locks.

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

Table VI. 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)
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

Table VII. Execution times for one iteration and processing rates for the relaxation algorithm on one processor of the Cray/2.

Problem size	Main memory access time	Addition time	Multiplication time	Total estimated time	Measured time
64×64	1.75	1.18	0.53	3.46	2.62
128×128	5.81	4.10	1.78	11.69	10.47
256×256	23.54	16.61	7.22	47.37	41.83
512×512	94.71	66.83	29.03	190.57	164.40
1024×1024	379.93	268.07	116.51	764.51	639.49

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

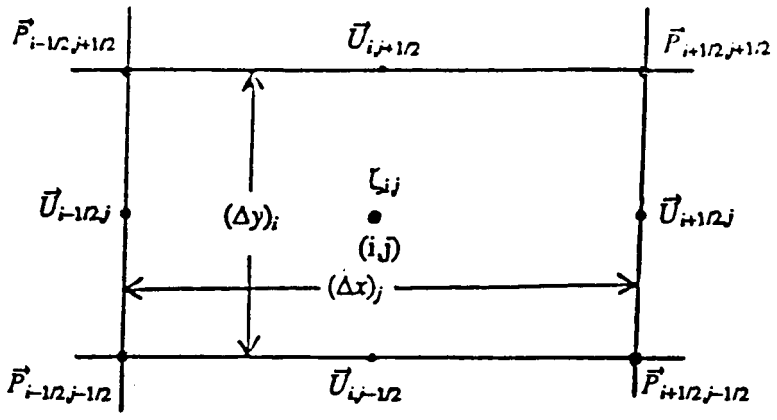

Fig. 1. Typical computational cell and the data associated with it.

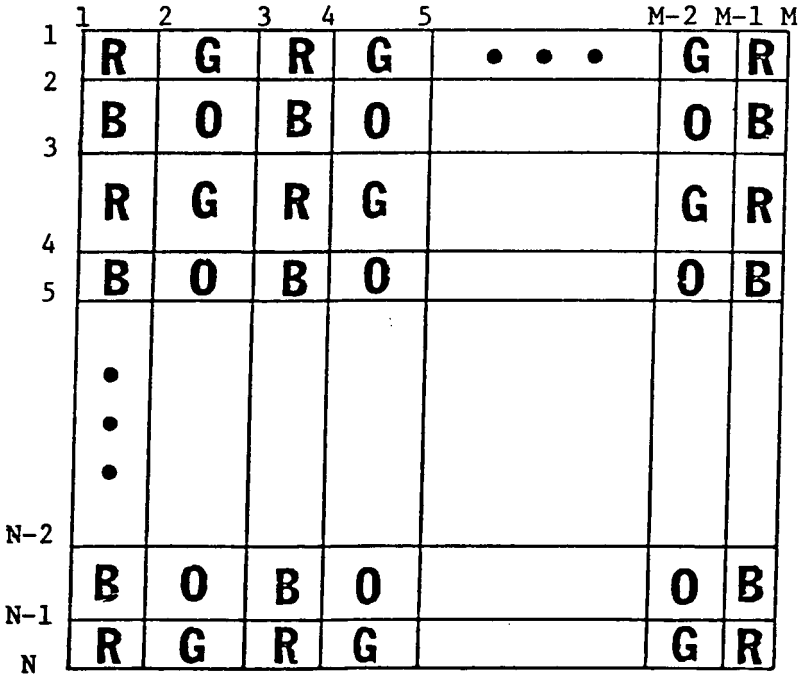

Fig. 2. Computational domain and the four color cells assuming that M and N are even numbers.

```
c
c       compute the residuals
c
        r1(i,j)     = zl(i,j) * (p(i,j+1) + p(i+1,j+1) - p(i,j) - p(i+1,j))
     1                + q(i,j+1) + q(i,j) - q(i+1,j+1) - q(i+1,j)
        r2(i,j)     = zl(i,j) * (q(i,j+1) + q(i+1,j+1) - q(i,j) - q(i+1,j))
     1                + p(i+1,j+1) + p(i+1,j) - p(i,j+1) - p(i,j) - z(i,j)
c
c       compute the relaxation correction and restore the P's
c
        p(i,j)      = p(i,j)      + c(i,j) * ( zl(i,j) * r1(i,j) + r2(i,j) )
        q(i,j)      = q(i,j)      - c(i,j) * ( r1(i,j) - zl(i,j) * r2(i,j) )
        p(i+1,j)    = p(i+1,j)    + c(i,j) * ( zl(i,j) * r1(i,j) - r2(i,j) )
        q(i+1,j)    = q(i+1,j)    + c(i,j) * ( r1(i,j) + zl(i,j) * r2(i,j) )
        p(i+1,j+1)  = p(i+1,j+1)  - c(i,j) * ( zl(i,j) * r1(i,j) + r2(i,j) )
        q(i+1,j+1)  = q(i+1,j+1)  + c(i,j) * ( r1(i,j) - zl(i,j) * r2(i,j) )
        p(i,j+1)    = p(i,j+1)    - c(i,j) * ( zl(i,j) * r1(i,j) - r2(i,j) )
        q(i,j+1)    = q(i,j+1)    - c(i,j) * ( r1(i,j) + zl(i,j) * r2(i,j) )
```

Fig. 3. Kernel of the relaxation subroutine in Fortran.

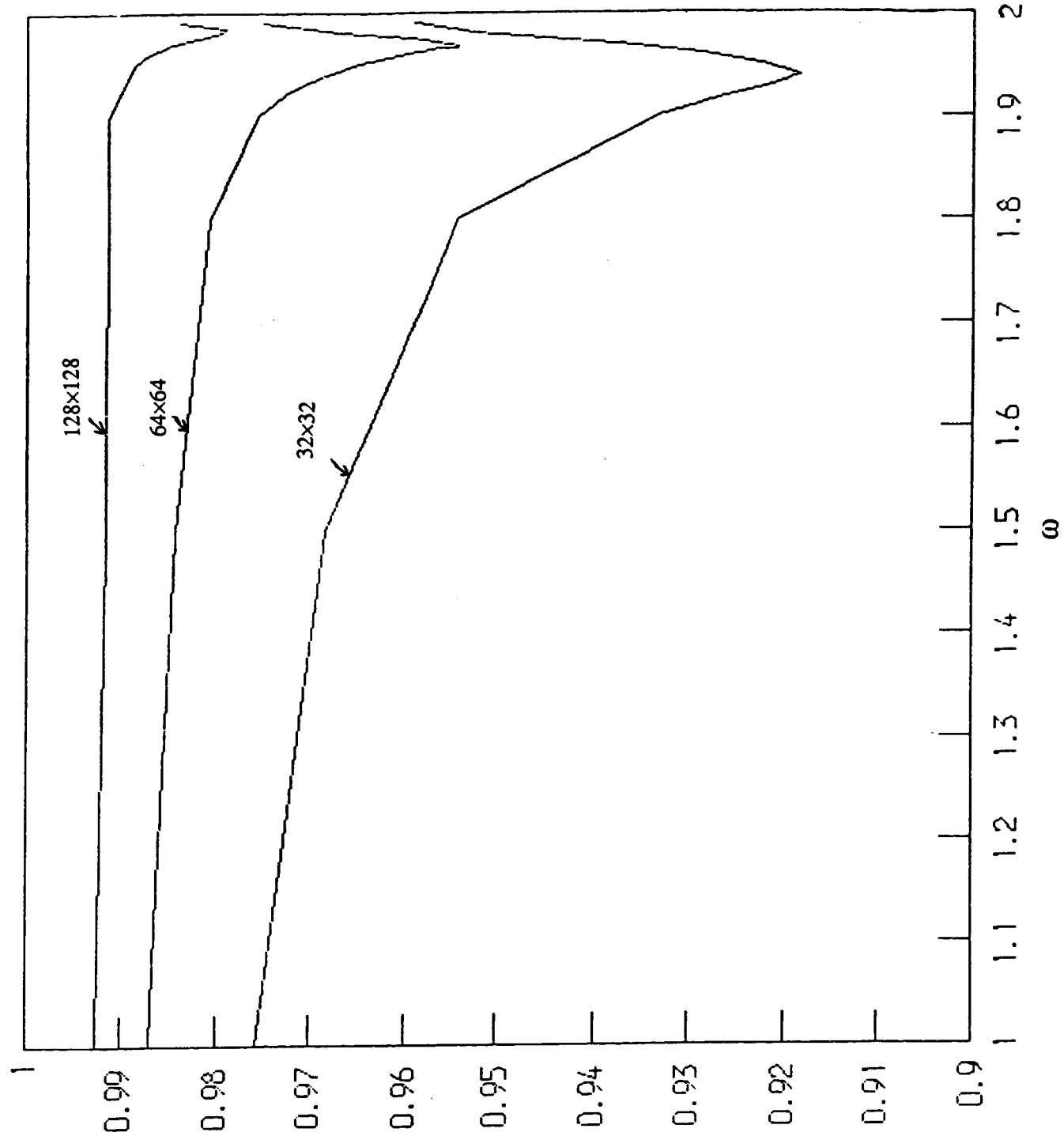

Fig. 4. Spectral radius as a function of the accelaration parameter.

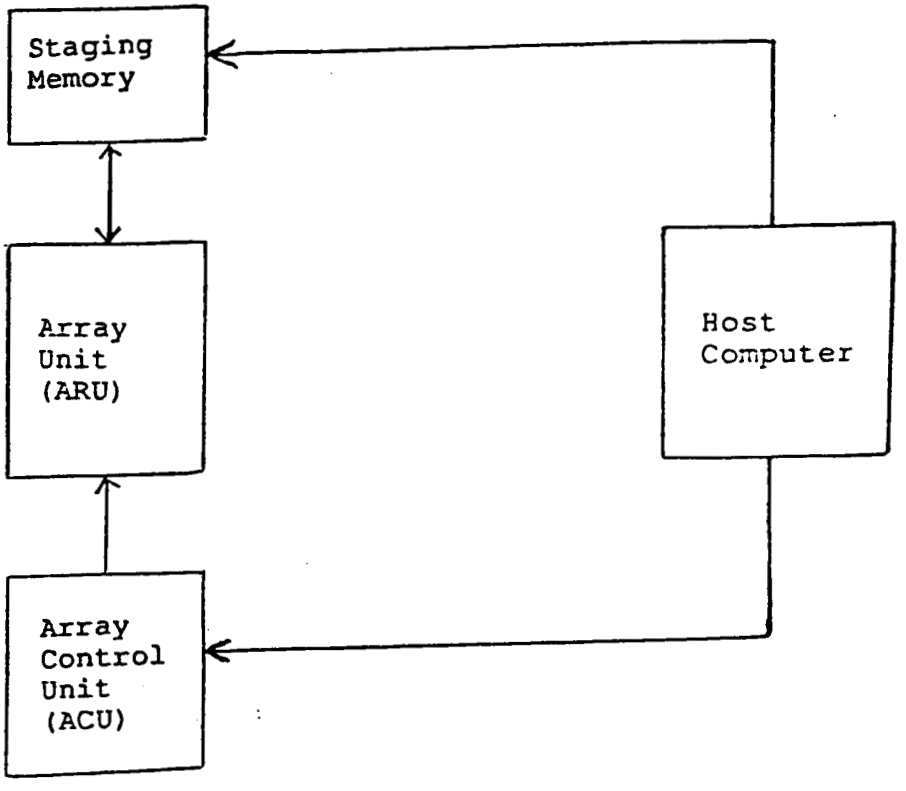

Fig. 5. Block diagram of the MPP.

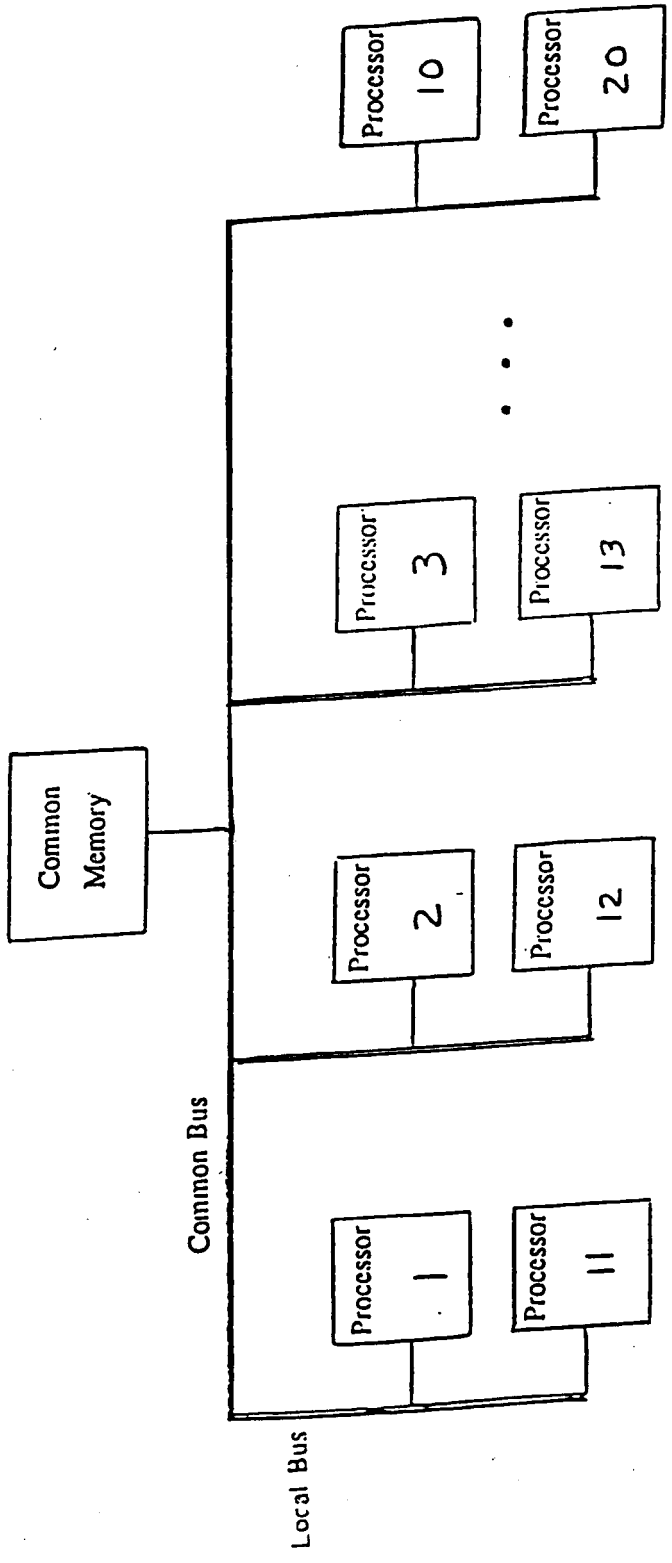

Fig. 6. Block diagram of the Flex/32 architecture.

Standard Bibliographic Page

1. Report No. NASA CR-178307 ICASE Report No. 87-34	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle SOLVING THE CAUCHY-RIEMANN EQUATIONS ON PARALLEL COMPUTERS		5. Report Date May 1987 6. Performing Organization Code
7. Author(s) Raad A. Fatoohi and Chester E. Grosch		8. Performing Organization Report No. 87-34
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225		10. Work Unit No. 505-90-21-01 11. Contract or Grant No. NAS1-17070, NAS1-18107
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report 14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
J. C. South

Submitted to J. Supercomputing

Final Report

16. Abstract

In this paper, we discuss the implementation of a single algorithm on three parallel-vector computers. The algorithm is a relaxation scheme for the solution of the Cauchy-Riemann equations; a set of coupled first order partial differential 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 CRAY/2, an MIMD machine with four vector processors. The machine architectures are briefly described. The implementation of the algorithm is discussed in relation to these architectures and measures of the performance on each machine are given. Simple performance models are used to describe the performance. These models highlight the bottlenecks and limiting factors for this algorithm on these architectures. Finally conclusions are presented.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
SIMD machine, MIMD machine, parallel algorithms, performance analysis, SOR method	61 - Computer Programming and Software 64 - Numerical Analysis Unclassified - unlimited

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	37	A03

For sale by the National Technical Information Service, Springfield, Virginia 22161