SOCIETY OF PETROLEUM ENGINEERS OF AIME
6200 North Central Expressway
Dallas, Texas 75206

PAPER NUMBER SPE 2813

Reservoir Simulation Using a General Processor and an Array Processor in Parallel

By

Olin G. Johnson, IBM Scientific Center

ABSTRACT

In this paper, we describe an array or vector processor that, with only one basic arithmetic operation, can be used in parallel with a central processor for numerical reservoir simulation. In doing so, we demonstrate a partitioning of the problem to balance the computational load between the processors under the constraints imposed by the instruction sets and the relative speeds. We also discuss certain mathematical aspects of the partitioning and the iterative algorithms. Finally, we present results from numerical experiments using the IBM 2938 Array Processor with an IBM 360/85, 75, 65 or 44.

Index Terms for IBM Subject Index

Array Processing	Mathematics
Reservoir Modeling	Numerical Analysis
ADI	Computer Applications
SOR	IBM 2938

INTRODUCTION

Much has been written recently in the area of parallel computation. Some problems have been attacked by using several identical general purpose processing units sharing a common memory bank.[8] Other problems have motivated the design of some highly parallel, special purpose arithmetic units arranged in two-dimensional arrays, each with its own memory and each with the ability to communicate with some of its neighboring units.[1]

In this paper, we consider a very elementary type of parallel computation that is a combination of these two types. The idea is to augment a general purpose computer with an auxiliary arithmetic unit of special type. The computational objective is to place a large part of the computational load arising out of problems such as numerical reservoir simulation in the special unit. This objective dictates that the unit be floating point and operate at high speed on vectors and band matrices. The economic objectives are to create a system that is much less expensive than two general purpose central processing units and that is several times as fast as a stand-alone central processor.

In particular, we show that if the auxiliary arithmetic unit, which we will henceforth call the array processor, contains only one basic vector function, then we can construct a parallel algorithm for reservoir simulation. Further, we have implemented this algorithm on an IBM 360/44, 65, 75, 85 with an IBM 2938 array processor and we present numerical results from these programs. In Section 2, we give a functional description of the array processor. In Section 3, we identify the basic computation kernels in a standard reservoir model. In Sections 4 through 6, we evaluate these kernels

References and illustrations at end of paper.

for array processing and present a parallel algorithm. Finally, in Section 7, we present numerical results using the IBM 2938.

FUNCTIONAL DESCRIPTION OF AN ARRAY PROCESSOR

Basically, we want an auxiliary processor that performs as few operations as possible and that can be used for parallel computation in reservoir simulation. The idea of having only a few operations is, of course, motivated by cost considerations. These same cost considerations, as well as speed considerations, further dictate the type of operations that we can hope to execute on the auxiliary processor. For instance, if we only perform vector operations on the auxiliary processor, we can use "pipelined" hardware. Secondly, if we only add and multiply and not divide, the cost of the processor will be considerably less. Thus, cost and speed factors in hardware design, lead us to a very specialized instruction set.

The object of this paper is to show that this limited instruction set is quite sufficient to increase throughput significantly. In order to accomplish this objective, we first describe an array processor with the above characteristics. For our purposes, a broad functional description of the array processor will be sufficient. We will need the array processor to execute only one arithmetic function:

$$y_i \leftarrow u_i + v_i * x_i \qquad i = 1,\ldots,n. \quad \ldots\ldots\ldots\ldots\ldots\ldots [1]$$

The various subcases of Eq. 1, such as $y_i \leftarrow u_i + x_i$, $y_i \leftarrow v*x_i$ may of course be separate instructions for the purpose of efficiency. An array processor program is a string of these instructions in any order.

The array processor program and data are stored in core along with the program and data for the central processing unit. The array processor and the central processor execute their program simultaneously. Thus the two processors are independent. However, we will need, and we assume there exists a method for syncronizing the two programs when necessary. For instance, if an instruction in the central processor program is not to be executed until the array processor is at a certain point in its program, then the central processor has the ability to determine whether to wait or continue.

The final functional characteristic that concerns us is one that might be called the parallel or recursive index, r, of the array processor. The length, n, of the vectors in Eq. 1 can vary from one instruction to the next, yet the array processor will always perform the operation in bursts of r components. That is, the array processor will fetch the first r components of each of the vectors u, v and x and compute the first r components of y in parallel [essentially]. Each subsequent set of r components is computed until the length n is exhausted.

To illustrate the importance of the index r, consider the following instruction:

$$y_i \leftarrow y_i + u_i * y_{i-s}$$

$$i = s+1,\ldots,n \qquad s \geq 1. \quad \ldots\ldots [2]$$

If this instruction is performed serially, one y_i at a time, then each of the computed values $y_{\nu s+1}$ $\nu = 1,2\ldots$ depends on y_1. The original value of y_{s+1} is not important since its new value is computed before it appears as data for y_{2s+1}. On the other hand, if Eq. 2 is performed completely in parallel, only the new value of y_{s+1} depends on y_1, and the new value of y_{2s+1} depends on the original value of y_{s+1}.

In the case of the array processor, we compute r elements at a time in parallel. Thus if $s \geq r$ the results are exactly the same as if Eq. 2 were performed serially, but if $s < r$, the resulting vector y will not necessarily be either a truly serial or parallel result. This consideration will be important to us in evaluating iterative techniques for array processing.

COMPUTATION KERNELS OF A REPRESENTATIVE MODEL

Before we can identify the array processor uses in reservoir simulation, we must identify what we might call the computational profile of the problem. As we might expect, there is considerable variation in this profile from model to model. Some models are input-output bound on even relatively slow central processing units. Other models perform multicomponent flash calculations in each cell of the model at each time step. However, by and large, the standard model today consists of two or three, coupled, quasi-linear, parabolic equations. The method of solution is also reasonably standardized. The equations are linearized by using the values of the unknowns, pressure and saturation, at the previous time step, in the coefficient calculations.[3] A backward time difference is used for the purpose of stability.[9] Thus, a large band set of simultaneous linear equations must be solved. The model and the method of solution dictate that the resulting computations fall into two distinct categories: [1] coefficient evaluation and [2] solution of linear equations. On most models, coefficient calculation uses 30 to 50 percent of the total computation time since the elements of the matrices must be recomputed each time step. In fact, experience has shown that some of the contributing terms in the matrix elements are so sensitive to

change that if an iterative scheme is used to solve the linear equations, those terms should be recomputed with each iterative step within the time period so that they reflect the functional value corresponding to the best approximation of the present pressures and saturation.

In order to identify the computational steps in more detail, we must now introduce a model. From the above discussion, it is clear that a single, quasilinear, pressure equation will, to a high degree, reflect the computational profile of more complicated models. Our problem, in two-space dimensions, is thus

$$\frac{\partial}{\partial x}\left(P(x,y,p)\,\frac{\partial p}{\partial x}\right) + \frac{\partial}{\partial y}\left(P(x,y,p)\,\frac{\partial p}{\partial y}\right) + Q(x,y) = R(x,y,p)\,\frac{\partial p}{\partial t} \quad \ldots \quad [3]$$

with boundary conditions $p[x,y,t_0]$ given:

$$\alpha p(x,y,t) + \beta\frac{\partial p}{\partial n}(x,y,t) = \text{constant for}$$

$$(x,y)\in\Gamma \text{ and } t \geq t_0,$$

where Γ is the boundary and $\partial/\partial n$ is the normal [outward] derivative.

Since Eq. 3 involves only a [single-phase] pressure, there is no saturation. In practice, however, the variations of the coefficient functions with saturation and the variation of saturation with time are more important than the variations of the coefficient functions with pressure. Thus, in order to keep the model simple and yet retain the computational favor of a full scale model, we will assume that P varies with p in a manner that is computationally similar to the variation of relative permeability, k_r, with saturation divided by the variation of viscosity, μ, with p, and that R varies with p in a manner that is computationally similar to the variation of saturation with time divided by the variation of pressure with time.

More explicitly, we will assume

1. R varies with p more strongly than P so that at the n^{th} time step the left-hand side of Eq. 3 can be linearized by replacing $P[x,y,p_n]$ with $P[x,y,p_{n-1}]$ as in Ref. 2. However, $R[x,y,p_n]$ should, in any iterative solution of the n^{th} time step, be replaced at each step of the iteration by $R[x,y,\tilde{p}_n]$ where $\tilde{p}_n$ is the present approximation to p_n.

2. The coefficient function P has the following form:

$$P(x,y,p) = \frac{\rho(p)k(x,y)k_r(p)}{\mu(p)}$$

where ρ and k vary like density and permeability, and $p = p[x,y,t]$.

3. The function $k[x,y]$ is stored as a table, whereas the functions ρ, k_r and μ are curve fitted as follows:

$$k_r(p) \doteq \sum_{\nu=0}^{3} c_{1\nu}p^{\nu}$$

$$\mu(p) \doteq \sum_{\nu=0}^{1} c_{2\nu}p^{\nu}$$

$$\rho(p) \doteq \begin{cases} \sum_{\nu=0}^{2} c_{3\nu}p^{\nu} & p \leq \bar{c}_3 \\ \sum_{\nu=0}^{2} c_{4\nu}p^{\nu} & p > \bar{c}_3 \end{cases}$$

4. R has the following form.

$$R(x,y,p) \doteq \phi(x,y)\,u(x,y,p),$$

where $$u(x,y,p) \doteq \sum_{\nu=0}^{5} w_{\nu}(x,y)p^{\nu}.$$

The constants $c_{\mu\nu}$ $\mu = 1,\ldots 4$; $\nu = 0,\ldots,$ $\bar{c}_3$ and the functions $\phi(x,y), w_{\nu}(x,y)$ $\nu=0,\ldots,5$ are assumed known.

We do not intend to imply that the above polynomial curve fits are in any way an industry standard. The subject of proper curve fitting techniques cannot be considered here. Thus the above approximations are quite arbitrary.

Besides the advantages of simplicity, and computational similarity to more complicated models, the above model has the third advantage that it can be used [and later is used] for pressure studies of the type found in Refs. 2, 6 and 10.

Let F be the finite difference matrix corresponding to the left side of Eq. 3, in which the points are numbered vertically [say]. Let

$$R = \text{diag}\,\{R(x,y,\tilde{p}_n)\}$$

$$Q = \text{diag}\,\{Q(x,y)\}$$

Then we have

$$\left(F - \frac{1}{\Delta t}R\right)p_n = -\frac{1}{\Delta t}Bp_{n-1} - Q.$$

Let $A = F - \frac{1}{\Delta t} R$,

$B = -\frac{1}{\Delta t} R$,

$b_n = Bp_{n-1} - Q$.

Then the algebraic problem is

$$Ap_n = b_n . \quad \ldots\ldots\ldots\ldots [4]$$

There exist a number of iterative methods for solving Eq. 4. In this paper, we will consider only two of them: Successive line over-relaxation [SLOR] and iterative [as opposed to noniterative[1]] alternating direction implicit [ADI]. Of these two, we will only look at ADI in detail. In using ADI, we have $A = H + V$ and we have n_0 parameters $r_1, \ldots, r_{n_0}$. We compute

$$\begin{cases} (H + r_m)\, p_n^{(m-\frac{1}{2})} = (r_m - V)\, p^{(m-1)} + b_n . \\ (V + r_m)\, p_n^{(m)} = (r_m - H)\, p^{(m-\frac{1}{2})} + b_n \end{cases}$$

$$m = 1, \ldots, n_0 . \quad \ldots [5]$$

Now we are faced with the problem of solving tridiagonal equations on the array processor. Any direct method for doing so is bound to involve a division at each step. Normal Gaussion elimination, for instance, is not an array processor operation, since the matrix must be triangularized sequentially and every third operation is a division. Clearly an algorithm is required in which all the dividing can be done at once and which allows the computation of a good many results in parallel. The LU decomposition thus suggests itself for the first problem, and the fact that Eq. 5 is reducible [i.e. decoupled by lines] solves the second. Thus we have the following computational steps.

I. Steps performed once only or occasionally
   - A. Computation [or read in] of Q
   - B. Computation [or read in] of boundary values
   - C. Evaluation of the ADI parameters r_i $i = 1, \ldots, n_0$

II. Steps performed once per time step
   Computation of F

III. Steps performed each time step [n] for each ADI parameter [m]
   - A. Computation of diagonal of B
   - B. Computation of diagonal of A
   - C. Computation of b
   - D. Computation of $(r_m - V)\, p_n^{(m-1)} + b_n$ and $(r_m - H)\, p_n^{(m-\frac{1}{2})} + b_n$
   - E. Decomposition of $r_m + H$, and $r_m + V$
   - F. Forward and backward H and V sweeps
   - G. Evaluation of convergence criteria

For convenience, in the next three sections, these steps are referred to by the above numbers and letters. We are, of course, only interested in the computations involved in II and III.

COEFFICIENT EVALUATION ON THE ARRAY PROCESSOR

Evaluation of most of the computation kernels of the last section for array processing is straight forward. For instance, consider II, the computation of F. The functions k_r and μ are polynomials in the vector p and can easily be computed on the array processor. On the other hand, the function ρ requires a test on each element of the p vector before the coefficients can be selected. Also, from the assumptions in Sec. 2, the division by μ cannot be performed efficiently with the prescribed instruction set for the array processor.

Let the grid consist of the points $[x_i, y_j]$ $i = 1, \ldots, n_1$; $j = 1, \ldots, n_2$. The horizontal terms in the diagonal elements of F are

$$\frac{p}{(\Delta x)^2} \sum_{k=0}^{1} p(x_{i-\frac{1}{2}+k}, y_j, .5\,(p(x_{i-1+k}, y_j, t) + p(x_{i+k}, y_j, t))).$$

Let $\bar{p}[x]$ be the vector whose elements are the points

$$\{.5(p(x_{i+1}, y_j, t) + p(x_i, y_j, t)) ;$$

$$i = 0, \ldots, n_1 ; \; j = 1, \ldots, n_2\}$$

numbered vertically. Similarly, let $\bar{p}[y]$, be the vector whose elements are the vertical arrangement of the points

$$\{.5(p(x_i, y_{i+1}, t) + p(x_i, y_j, t)) ;$$

$$i = 0, \ldots, n_1 ; \; j = 0, \ldots, n_2\}$$

We define $\bar{k}[x]$ and $\bar{k}[y]$ similarly.

The parallel program in Table 1 is then possible for computing F. The idle CPU time can, of course, be used to output the previous p vector.

EVALUATION OF ITERATIVE METHODS FOR ARRAY PROCESSING

In the third section, we noticed that the tridiagonal set of ADI equations

$$(V + r_m)p_n^{(m)} = (r_m - H)p^{(m-\frac{1}{2})} + b_n \quad \ldots\ldots\ldots\ldots\ldots\ldots [6]$$

are decoupled by lines. Hence, we can simultaneously solve Eq. 6 for $p_n^{(m)}(x_i,y_j,t_n)$ $i=1,2,\ldots,n_1$. Hence we have identified a possible "vector algorithm". Notice that we cannot solve Eq. 6 for $p_n^{(m)}(x_i, y_j, t_n)$ $j=1,\ldots,n_2$ simultaneously by any of the usual direct methods for linear equations unless the index of parallelism, r, is one.

In SLOR, on the other hand, the situation is not as nice. Instead of having a "global" [i.e., $n_1n_2 \times n_1n_2$] decoupled tridiagonal system, we have only a "local" [i.e., $n_2 \times n_2$], coupled, tridiagonal system. In solving for $p_n^{(m)}(x_i,y_j,t_n)$ $j=1,\ldots,n_2$ we use $p_n^{(m)}(x_{i-1},y_j,t_n)$ $j=1,\ldots,n_2$ which we have just computed, and $p_n^{(m-1)}(x_{i+1},y_j,t_n)$ $j=1,\ldots n_2$. Thus neither the x-direction nor the y-direction allows us a vector of non-trivial length, the elements of which can be computed simultaneously. We can correct this situation by using the so-called red-black ordering[9] rather than the successive ordering. With this ordering, we can simultaneously compute $p_n^{(m)}(x_{2i}, y_j,t_n)$ $i=1,\ldots,n_1/2$, then $p_n^{(m)}(x_{2i-1},y_j,t_n)$ $i=1,\ldots,n_1/2$. We note that this ordering is two-cyclic and consistent; hence the asymtotic rate of convergence is the same as the successive ordering.[9] Thus either ADI or red-black SLOR are possible array algorithms. We now look at ADI in detail.

EQUATION SOLVING ON THE ARRAY PROCESSOR

We first consider kernel IIID, computation of $(r_m-V)\,p_n^{(m-1)} + b$ and $(r_m-H)\,p_n^{(m-\frac{1}{2})} + b$. Let r_m-H be stored as an $[n_1n_2,2]$ array, $H^m_{k\nu}$ $k = 1,\ldots,n_1n_2$; $\nu = 1,2$. $H^m_{k,2}$ corresponds to the diagonal of $r_m - H$, and $H^m_{k,1}$ corresponds to the n_2+1st sub- and super-diagonals with zeros in the first n_2 positions. Let p stand for the vector $p_n^{[m-1]}$. We can then compute $[r_m - H]\,p_n^{[m-1]}$ in three steps.

$$q_k \leftarrow H_{k,2} * P_k \qquad k = 1,\ldots,n_1 * n_2$$

$$q_k \leftarrow q_k + H_{k+n_2,1} * p_{m+n_2,1} \qquad k = 1,\ldots,(n_1-1)*n_2$$

$$q_{k+n_2} \leftarrow q_{k+n_2} + H_{k+n_2,1} * P_k \qquad k = 1,\ldots,(n_1-1)*n_2$$

The treatment of the decomposition, III E, is more complicated and is related to the manner in which we perform III F, the forward and backward sweeps.

Let

$$H + r_m = L^m U^m \quad \ldots\ldots\ldots [7]$$

where

$$L^m = \begin{bmatrix} 1 & & & & \\ \ell^m_{n_2+1,1} & & & & \\ & \ell^m_{n_2+2,2} & & & \\ & & \ddots & & \\ & & & \ell^m_{n_3,n_4} & 1 \end{bmatrix}$$

and

$$U^m = \begin{bmatrix} U^m_{11} & & U^m_{1,n_2+1} & & \\ & \ddots & & \ddots & \\ U^m_{2,2} & & \ddots & & U^m_{n_4,n_3} \\ & \ddots & & \ddots & \\ & & U_{n_1n_2,n_1n_2} & & \end{bmatrix}$$

The forward sweep is an array operation. Mathematically, it is accomplished in n_1-1 step

using vectors of length n_2. Let

$$L^m Z = C.$$

Then $Z_i = C_i \qquad i = 1, \ldots, n_2$, and for $j = 1, \ldots, n_1-1$ we compute

$$Z_i \leftarrow C_i - v^m_{i-n_2} * Z_{i-n_2} \qquad i = 1+jn_2, \ldots, (j+1)n_2.$$

where $v^m = (\ell^m_{n_2+1,1}, \ldots, \ell_{n_3,n_4})^t$.

On the array processor, however, we only need two instructions to perform the above n_1-1 steps, provided that the index of parallism, r, is less than or equal to n_2. These instructions are

$$Z_i \leftarrow C_i \qquad i = 1, \ldots, n_1 n_2$$

$$Z_i \leftarrow Z_i - v^m_{i-n_2} * Z_{i-n_2} \qquad i = n_2 + 1, \ldots, n_1 n_2$$

If $r > n_2$, we n_1 instructions of course.

Upon division by the diagonal of U^m, the backward sweep is accomplished in similar fashion. Hence, the decomposition, III E, should supply us with the vectors v^m, and

$$w^m = \left(\frac{1}{U^m_{1,1}}, \ldots, \frac{1}{U^m_{n_1 n_2, n_1 n_2}}\right)^t$$

$$z^m = \left(\frac{U^m_{1,n_2+1}}{U^m_{1,1}}, \ldots, \frac{U^m_{n_4 n_3}}{U^m_{n_, n_2, n_1 n_2}}\right)^t.$$

From Eq. 7 we have

$$U^m_{k,k+n_2} = (r_m + H)_{k,k+n_2}$$

$$k = 1, \ldots, n_3 \quad \ldots\ldots\ldots\ldots [8]$$

and

$$\ell^m_{k+n_2,k} = (r_m + H)_{k+n_2,k} / U^m_{k,k}$$

$$k = 1, \ldots, n_3.$$

By symmetry of $r_m + H$, and Eq. 8 we have

$$v^m_k = Z^m_k = (-H_{k+n_2,1} + 2r_m) * w^m_k$$

$$k = 1, \ldots, n_3.$$

Thus once w^m is known, v^m can be computed on the array processor. The vector w^m itself must be computed with the central processor because of the divisions involved. That is

$$1/U^m_{kk} = [(r_m + H)_{k,k} - (r_m + H)^2_{k-n_2,k} * (1/U_{k-n_2,k-n_2})]^{-1}$$

$$k = n_2 + 1, \ldots, n_1 n_2.$$

We can thus perform the following parallel program in the computation of p^m_n. In the CPU, all data involving R will use p^{m-1}_n, and in the array processor all data involving R will use p^{m-2}_n.

NUMERICAL RESULTS

We implemented the above program on an IBM System 360/44, 65, 75, 85 with an IBM 2938 Array Processor. We wrote two separate programs - one with the 2938 and one without. We kept the details in the coding of the two programs as identical as possible. Our object was to determine the ratio of computation times of the two programs. Our test program used four ADI parameters [after Wachpress].[12] The reservoir was rectangular with zero pressure boundaries and one injection well placed at any interior mesh point in the rectangle. The analytic solution to this problem is given in Ref. 6 in the form of a double series. It can also be determined by the method of reflections.[5] The timing ratios for the IBM 360/75 are given for various grid sizes and six time steps in Table 3.

For those who are unfamiliar with the IBM 2938, we list some of the operating characteristics. More details are available in Ref. 7.

1. The mathematical function of Eq. 1 is implemented in four instructions:

a. $y_i \leftarrow u_i + v_i * x_i$

$u_i = 0$ or $u_i = y_i$

b. $y_i \leftarrow u_i + v * x_i$

$u_i = 0$ or $u_i = y_i$

c. $y_i \leftarrow u_i + x_i \quad x_i$

d. $y_i \leftarrow u_i$

$i = 1, \ldots, n$

2. There is an instruction to take the infinity norm

$$||y||_\infty \leftarrow \max_{i=1,\ldots,n} |y_i|$$

hence much of kernel III G can also be performed in the 2938.

3. All instructions are in single precision floating point.
4. The index of parallelism is 32.
5. A bit is available in each instruction to cause a programmed interrupt in the CPU.

We can now give a general flow chart showing how Table 2 is implemented. For simplicity, assume $p_n^{[0]}$ and $p_n^{[1]}$ are known. We can then compute $p_n^{[m]}$ $m = 2,3, \ldots$ as shown in Fig. 1.

ACKNOWLEDGMENT

The author wishes to thank Mr. Lorenzo Manzanera of the Service Bureau Inc. who programmed tests for this study.

REFERENCES

1. Andrews, D. H.: "An Array Processor Using Large Scale Integration", Computer Design [Jan., 1969] 34-43.
2. Briggs, J. E. and Dixon, T. N.: "Some Practical Considerations in the Numerical Solution of Two-Dimensional Reservoir Problems", Soc. Pet. Eng. J. [June, 1968] 185-194.
3. Douglas, J., Jr.: "A Numerical Method for the Solution of a Parabolic System", Numer. Math. [1960] 2, 91-98.
4. Forsythe, G. and Moler, C.: Computer Solution of Linear Algebraic Systems, Prentice-Hall, Englewood Cliffs, N.J. [1968].
5. Horner, D. R.: "Pressure Build-Up in Wells", Proc., Third World Pet. Cong., Sec. II [1951].
6. Hovanessian, S. A.: "Pressure Studies in Bounded Reservoirs", Soc. Pet. Eng. J. [Dec., 1961] 223-228.
7. IBM Reference Manual: IBM System 360 Custom Feature Description, 2938 Array Processor Model 1 RPQ W24563, Model 2 RPQ 815188, Form A24-3519.
8. Miranker, W. L. and Liniger, W.: "Parallel Methods for the Numerical Integration of Ordinary Differential Equations", Math. of Comp. [1967] 21, No. 99, 303-320.
9. Peaceman, D. W.: "Numerical Solution of the Nonlinear Equations for Two-Phase Flow Through Porous Media", Nonlinear Partial Differential Equations, Academic Press, New York [1961] 171-191.
10. Price, H. S., Varga, R. S. and Warren, J. E.: "Application of Oscillation Matrices to Diffusion-Convection Equations", J. Math. and Phys. [1966] 45, 301-311.
11. Varga, R. S.: Matrix Iterative Analysis, Prentice-Hall, Englewood Cliffs, N. J. [1962].
12. Wachspress, E. L.: "Optimum Alternating-Direction-Implicit Iteration Parameters for a Model Problem", J. Soc. Indust. Appl. Math., 10, 339-350.

Table 1

Time Syncs.	C.P.U.	A.P.
	"Idle"	Compute $\bar{p}(x)$ $\bar{p}(y)$
*		
	Compute $\rho(\bar{p}(x))$ $\rho(\bar{p}(y))$	$k_r(\bar{p}(x))$ $k_r(\bar{p}(y))$ $\mu(\bar{p}(x))$ $\mu(\bar{p}(y))$
*		
	$1/\mu(\bar{p}(x))$ $1/\mu(\bar{p}(y))$	$k_r(\bar{p}(x))\bar{k}(x)\rho(\bar{p}(x))=\bar{\ell}(x)$ $k_r(\bar{p}(y))\bar{k}(y)\rho(\bar{p}(y))=\bar{\ell}(y)$
*		
	"Idle"	$(\bar{\ell}(x)/(\mu(\bar{p}(x))(\Delta x)^2)$ $(\bar{\ell}(y)/(\mu(\bar{p}(y))(\Delta y)^2)$

Table 2

Time Sync.	C.P.U.	A.P.
	IIIA: $B=B(p_n^{m-1})$	IIIC: $b=b(p_n^{m-2})$
	IIIB: $A=A(p_n^{m-1})$	IIID: $H=H(p_n^{m-2})$
	IIIE(w^m): $H=H(p_n^{m-1})$	IIIE(v^m): $H=H(p_n^{m-2})$
		IIIF: $H=H(p_n^{m-2})$
*	IIIG	

Table 3

Grid Size	Stand alone time (minutes)	Parallel Time (minutes)	Ratio
10 x 20	.047	.014	3.4
20 x 20	.093	.028	3.3
20 x 30	.140	.042	3.3
20 x 40	.190	.057	3.3

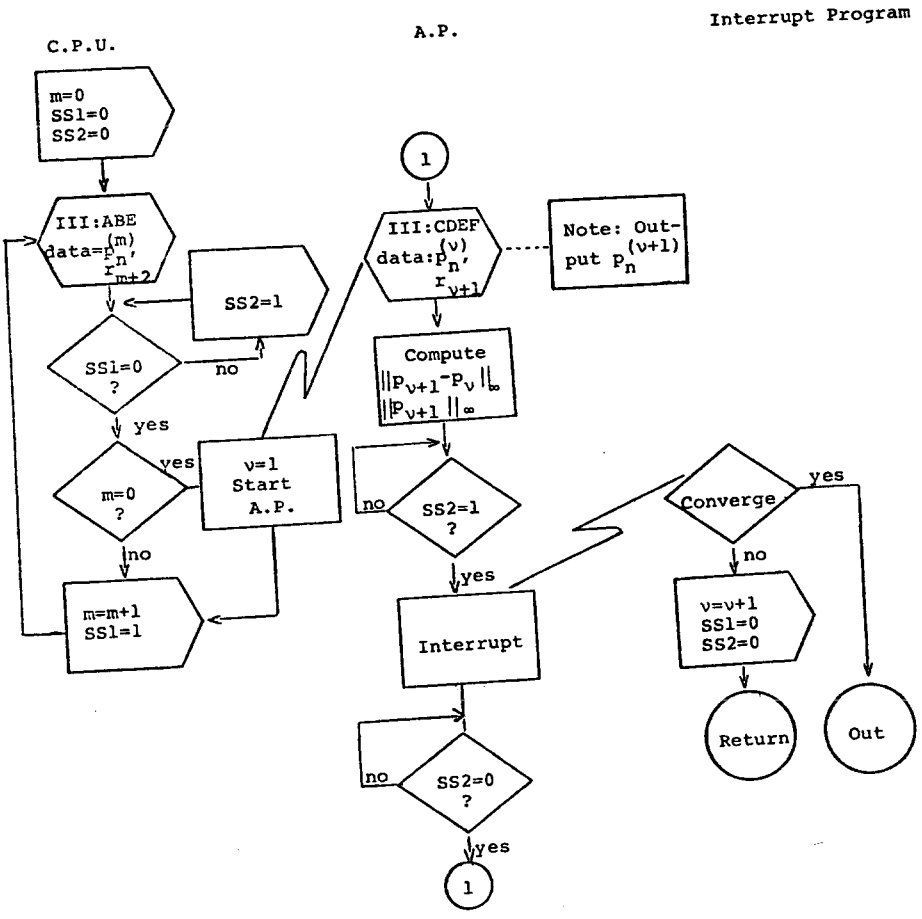

Fig. 1