SPE
Society of Petroleum Engineers

SPE 16020

Application of Parallel (MIMD) Computers to Reservoir Simulation

by S. L. Scott*, R. L. Wainwright, R. Raghavan*, U. of Tulsa
and H. Demuth, U. of Idaho

* SPE Members

This paper was prepared for presentation at the Ninth SPE Symposium on Reservoir Simulation, held in San Antonio, Texas, February 1-4, 1987.

ABSTRACT

The application of parallel computers to reservoir simulation is studied with the aid of two recently introduced Multiple Instruction, Multiple Data (MIMD) computers. The problems posed by reservoir simulators are shown to be particularly well suited for such parallel computers. Fundamental parallel programming techniques using Fortran are presented for the Hetrogeneous Element Processor (HEP) produced by Denelcor and the iPSC Hypercube produced by Intel. These techniques are then applied to two of the most time consuming tasks in reservoir simulation: forming the matrix coefficients and producing a sparse matrix solution. The problem of parallel formation of sparse matrix coefficients is addressed for the single and multiphase cases using both black oil and compositional fluid models. Parallel sparse matrix solution is addressed by considering D4 ordered Gaussian elimination, and the multigrid, conjugate gradient, and Successive Over-Relaxation (SOR) methods. Three parallel algorithms based on SOR are developed to illustrate the possible diversity of parallel algorithms. The red-black and multicolored SOR methods which are often used on vector machines are shown to be easily adaptable to parallel MIMD machine. Matrix partitioning and a new method of iteration pipelining are also presented as parallel SOR methods. The various SOR algorithms, in parallel form, are mapped onto the HEP to illustrate the speedups currently possible on an MIMD machine. Projections are made on the future importance of these types of machines to reservoir simulation.

INTRODUCTION

Over the past decade, significant advances have been made in the computing power available for reservoir simulation. Computer speeds have roughly doubled every two years for the past forty years. This increased computing power has made possible the practical implementation of more advanced reservoir simulator models. For example, use of the time consuming fully implicit method is now a practical alternative to the Implicit Pressure, Explicit Saturation (IMPES) method. Clearly, the advances in computing power are directly related to both the quality of the results obtained and the number of different scenarios that can be considered. In view of these changes, understanding and applying the new generation of supercomputers is of great importance to reservoir simulation.

References and illustrations at end of paper.

Recent advances in computing power have to a large degree been brought about by "supercomputer" vector machines such as the CRAY and the Control Data Corporation CYBER. There is increasing interest in producing even faster computers which will be highly parallel in nature. Even CRAY Inc. is following this trend by the introduction of it's XM/P computers which offer parallel central processing units. Highly parallel MIMD computers have been studied in research labs over the last dozen years and a few have been made available commercially. These parallel machines, while already in the supercomputer class, can be considered precursors to the next generation of supercomputers. An understanding of the basic capabilities and properties of these parallel machines is crucial to their appropriate application to reservoir simulation.

The intent of this paper is to show basic methods of applying parallel MIMD computers to reservoir simulator problems. Also, it is hoped that the paper will demonstrate the great potential these machines hold for reservoir simulation. First the basics of parallel computers and parallel programming will be introduced. Parallel programming concepts are then applied to two of the most time consuming tasks in reservoir simulation: forming matrix coefficients and sparse matrix solution. The forming of matrix coefficients is considered for the single and multiphase cases using black-oil and compositional fluid models. Parallel matrix solution is addressed using D4 ordered Gaussian elimination, multigrid, conjugate gradient, and SOR methods. The future of parallel computing in reservoir simulation is discussed based on the results

found on the currently available parallel machines.

PARALLEL COMPUTERS

A "parallel" computer is capable of executing independent parts of code simultaneously. A computer classification scheme that illustrates various kinds of parallelism was defined by Flynn [14] as:

1. SISD (single instruction stream, single data stream)

2. SIMD (single instruction stream, multiple data streams)

3. MISD (multiple instruction streams, single data stream)

4. MIMD (multiple instruction streams, multiple data streams)

The von Neumann or sequential type of computer is a SISD machine. Most computers available today are SISD machines in which instructions are executed sequentially one after another on one set of data. By contrast SIMD machines apply a single sequential set of instructions to multiple data sets. Typically, a single instruction, such as add, is performed on multiple data items simultaneously. For example, the CRAY has registers that allow simultaneously formation of 64 sums on a single vector add instruction. Most multiprocessor computer systems can be characterized as MIMD machines, or it can be thought of as a collection of independent SISD or SIMD systems connected by an extensive network or via shared memory that allows all of the processors to work together on a single problem.

Parallel Computer Technology

The realized and potential economic benefits of greater computing speeds (more calculations per dollar) have established a strong demand for faster computers. Such machines are being built with smaller, faster and less expensive circuitry. There is, in addition, a strong thrust to obtain faster computers through parallelism. Some designs, such as the C.mmp, MMP and S-1 [15, 16] and computers under development at Columbia [8] and The University of Illinois [15], focus on the use of many processors that can share the computations of a single problem. Other research designs suggest the use of many small identical circuits acting in a pulsing "systolic" way to perform, say, matrix calculations. Still other research and recent commercial designs focus on the parallelism of data flow through a problem. Here elements do a specific calculation and produce a result as soon as the input data is available. An assemblage of many elements of this type yields a high degree of parallelism and offers high calculation rates. Thus, research and commerical development of highly parallel computers is proceeding in many directions. Several commercial parallel computers of particular interest are discussed below.

The HEP Parallel Computer

The Hetrogeneous Element Processor (HEP) produced by Denelcor is an MIMD machine capable of executing instructions from more than one instruction stream simultaneously. The HEP performs as an ideal MIMD machine up to a limit of 8-12 processes in that there are essentially no message passing communication or synchronization overheads. The HEP achieves its parallel capabilities by use of an eight stage pipeline that is filled with instructions from different processes rather than from the same process. The parallel capabilities are enhanced by use of a switching network into a shared memory system. User control of the parallel execution of a program is easily performed through several additions to the FORTRAN language.

Parallel programming on the HEP is made possible by two additions to the Fortran language. The first is the CREATE statement. It is similar to a Fortran subroutine call, except the "created" subroutines execute in parallel along with the calling program. The second addition is the use of asynchronous variables that act as semaphores to allow control of the execution of the different processes.

Asynchronous variables on the HEP are differentiated from other variables by using a "$" as the first character in its name. Each asynchronous variable has an additional bit to indicate whether the variable is "full" or "empty". When the value of an asynchronous variable is requested and obtained then the variable is considered "empty". This indicates to other processes that the value associated with that variable may be in the process of being changed and the current value may not be valid. An asynchronous variable is termed full when its value is replaced (stored). A new value can not be stored in an asynchronous variable when that variable is "full" unless it is first emptied. Figure 1 shows an example of the use of asynchronous variables.

In this example two processors are trying to update the shared variable $TOTAL. Suppose process #1 reads the value of $TOTAL, thereby emptying it. When process #2 tries to access $TOTAL, it will be unable to do so because $TOTAL is empty. Therefore process #2 waits until $TOTAL is filled. Process #1 now has exclusive control of the variable $TOTAL. Process #1 now adds its sum to $TOTAL and stores the value into $TOTAL, making it full. Process #2, which has been trying all along to read $TOTAL, now finds that it can do so. It reads $TOTAL, setting it to empty, and the addition is performed correctly. Without the asynchronous nature of this variable, both process #1 and #2 could access the old value of $TOTAL, both could add their respective sums to it, and process #1 store the result first and then process #2 overwrite this value with its sum. Asynchronous variables provide a convenient mechanism to share variables between processes and to control the execution of processes.

Hypercube Parallel Computers

The hypercube processor connection method is employed by several different computer manufacturers. The iPSC computer produced by Intel will be the focus of this discussion. In general, a hypercube parallel computer is a collection of SISD machines joined to their nearest neighbors in a n-dimensional cube. The SISD machines are called nodes and are completely independent machines with their own memory and resident copy of the operating system. In fact each node of the iPSC is equivalent to the IBM PC-AT. The iPSC can be configured using from 32 to 128 nodes. A new version of the Intel hypercube is the iPSC-VX which uses up to 64 tightly coupled vector/array processor (SIMD) nodes to achieve a performance of 424 million floating-point

operations per second (MFLOPS) for 64 bit arithmetic. Each node is also connected to a host computer which serves to start and control the execution of the node processes. Figure 2 shows the communication network for a typical 3 dimensional cube.

The parallel capabilities of the iPSC are controlled by message passing over the cube network. FORTRAN subroutine calls are used to send and receive synchronous and asynchronous messages with message routing being performed automatically by the system. The cube architecture guarantees that a message from one node to any other arbitrary node will not have to be routed more than n times where n is the dimension of the cube. A message will on the average only have to be routed n/2 times. This means that how a problem is mapped onto the cube is not that important unless an extremely large amount of inter-process communication is required.

The use of the hypercube is somewhat similar to using an array processor. Subroutine calls are made to move information from the host to the cube and to retrieve the computed results. Two programs must be written to use the hypercube. One resides on the host computer and one or more programs reside on the cube nodes. Figure 3 shows an example of how information is passed between the host and the nodes. In this example processes 1 and 2 have calculated partial sums and are sending this information back to the host where it is to be totaled. Parameters such as buffer size, and various channel id's have been omitted in this example.

The CRAY X-MP Vector-Multiprocessor Computers

The CRAY X-MP machines combine both vector and multiprocessor capabilities. These machines can be configured with up to four processors. For example, the CRAY X-MP 48 is a collection of four CRAY-1 like vector processors with shared memory and I/O systems. The 8 represents eight million words of memory. Likewise, the CRAY X-MP 24 represents a configuration of two processors with four million words of memory. Use of the parallel capabilities of these machines on the same problem is possible, or they can be used for multitasking a number of different problems. The CRAY X-MP was not used in this study primarily because of its limited number of processors.

Multiprocessor Computers

Recently a number of computer vendors (Apollo, Harris, etc.) have introduced Multiprocessor computers. The Apollo DSP9000 allows parallel execution of simple vectorizable code segments through its optimizing compiler. It therefore can be considered to be a MIMD machine, however, when system loads become high it becomes a multiprocessing machine with each processor working exclusively on one process. The Harris MCX series, while having multiple processors, can not be considered a parallel MIMD machine since the processors can not simultaneously perform computations on the same process. These machines, while related to the MIMD machine, in reality should be classified as Multiprocessor SISD machines. They provide significant response time improvements in a timesharing environment but do little to speed the solution of large reservoir simulator problems.

PARALLEL PROGRAMMING TECHNIQUES

As with vector machines, some special programming is generally required to gain optimum performance from a MIMD machine. The dramatic effect of restructuring reservoir simulators for execution on vector machines has been shown in numerous papers [20,22,24,25]. Similarly, the successful speedup of a program on an MIMD machine is dependent on restructuring existing algorithms to take full advantage of architectural offerings of the machine. Fortunately, parallel processing is applicable to a much wider class of problems than is vector processing, and parallel programming is fairly natural for most applications.

Description of a Parallel(MIMD) Machine

Optimal program restructuring for parallelism may well depend on the architecture of the machine being used. Ordinarily, communication and synchronization costs must be considered with care. However, in order to illustrate some of the basic parallel programming techniques the concept of an "ideal" parallel machine is used. An ideal parallel machine is a machine that allows concurrent operations and communication between processes regardless of how it is physically implemented in the architecture. This type of machine can be simulated on both the HEP and iPSC hypercube. The HEP was chosen to be used in the following examples because its Fortran interface most closely followed the "ideal" parallel machine concept.

The objective of parallel programming is to find the best way to divide a program into independent parts, or processes, that can be executed in parallel. Ordinarily, each process should be of sufficient size to justify the startup costs associated with beginning a process. Ideally a problem can be split into parts of equal size which are allocated to all available processes or processors. Several of the basic methods for partitioning a program into independent parts include divide and conquer, pipelining, and reordering.

Divide and Conquer

The "Divide and Conquer" technique can best be described as the division of a task into a number of smaller tasks that are completely independent of each other. The DO LOOP of Figure 4 can be used to illustrate this technique. In this example 10 processors are available and the loop in divided into 10 equal parts and given to the processors to execute. All ten processes are independent except for the shared variable $TOTAL. When each process has finished, it's sum is added into $TOTAL. These additions can be performed without fear of conflict as explained previously.

Any algorithm that has been vectorized can directly utilize this technique. One would expect the parallel algorithm to be approximately ten times faster than the sequential algorithm. The definition of the Speedup of a parallel algorithm using p processes is:

$$S(p) = \frac{\text{Time to execute algorithm with one processor}}{\text{Time to execute algorithm with p processors}} \quad (1)$$

There are several limitations to the Divide and Conquer method. First, there is a finite amount of

work required to start each of the processes. Therefore, the processes must have a fairly large amount of useful work to perform in order to maintain the optimum linear speedup. This can be thought of as being analogous to the minimum vector start up length characteristic of vector type computers. For example, if 100 processes were available, the theoretical speedup possible for this algorithm on an MIMD machine would be 100, but in practice it is usually around (100)/LN (100) [16]. The definition of the efficiency of a parallel algorithm is given by:

$$E(p) = \frac{S(p)}{p} \tag{2}$$

Using the above general performance of a parallel algorithm as S(p)=p/LN (p), it is observed that the efficiency, E, approaches 0 as the number of processes, p, approaches infinity [26]. A direct result of (2) is that a machine with a small number of very fast processors may be more efficient than a machine with a large number of slower processors. However, since multitasking is also used on parallel machines, this conclusion is not always valid. Multitasking allows the efficiency to remain high in part because when the CPU is required to wait for one process it can continue to do useful work on another process.

A second limitation to the Divide and Conquer method is that it can not be universally applied due to recurrence relations. The next two methods consider how to solve recurrence type problems.

Pipelining

Pipelining is a method used to allow parallel execution of a sequential algorithm that is performed repeatedly. It exploits temporal parallelism in the execution of the algorithm. Pipelining is widely used at the instruction level in sequential CPU's where the execution of an instruction can be divided into several discrete steps such as: 1) fetch; 2) decode; and 3) execute. Since these steps are performed repeatedly, independently, and in order they can be pipelined in much the same way as an automobile assembly line where a number of people work in parallel on different steps of a sequential task.

For example, consider an implicit finite difference approximation to a one dimensional parabolic partial differential equation. The sequential algorithm can be visualized as in Figure 5. This algorithm can be pipelined using four parallel processors by calculating several interation levels simultaneously as shown in the example seen in Fig. 5. Figure 5 also shows pseudo code for this algorithm when executed on the HEP. The speedup that can be obtained by the pipelining method is dependent on p, the number of stages or processors, and k, the number of times the operation is performed. The theoretical maximum speedup of the pipelining method is given by:

$$S(p) = \frac{p*k}{p + (k-1)} \tag{3}$$

As can be seen from Eq. (3), S(p) approaches p for k>>p.

Reordering

Some algorithms must be completely reordered to execute in parallel. This will be illustrated in the matrix solution section of this paper. An example of this method is the red-black Successive Over-Relaxation (SOR) method of sparse matrix solution. Here the conventional sequential SOR algorithm is reordered to allow parallel execution. The resulting method is slightly different than the original in its formulation but is much better suited to parallel execution on an MIMD machine.

FORMING MATRIX COEFFICIENTS

The solution of matrices is commonly assumed to be the most time consuming task in reservoir simulation. But as shown by Kendall [20] and others, the forming of the matrix coefficients can often take as much or more CPU time as the actual matrix solution. Tasks such as PVT and saturation table look-up's and Newton-Raphson iteration require large amounts of time for the nonlinear, multiphase simulators. Therefore, consideration of methods to speedup these tasks by parallel methods is also important. First, the formation of coefficients for a simple single phase, black oil problem will be considered. Forming matrix coefficients for the multiphase case will then be discussed for both black oil and compositional simulator.

Single Phase Case

A single phase, five point discretized, two dimensional areal simulator is used to illustrate the types of operations performed to calculate matrix coefficients. The parallel methods discussed earlier are applied to these code segment and performance is estimated.

For the single phase liquid case a typical matrix equation is given below.

$$f_{i,j}P^{n+1}_{i,j-1} + b_{i,j}P^{n+1}_{i-1,j} + a_{i,j}P^{n+1}_{i,j} + c_{i,j}P^{n+1}_{i+1,j} + g_{i,j}P^{n+1}_{i,j+1} = d_{i,j} \tag{4}$$

The off-diagonal coefficients are functions of k, μ, Δx, and Δy; a(i,j) is also a function of c_t and d(i,j) is a function of μ, c_t, Δx, Δy, and Δt. For this case the matrix coefficients are usually treated as constant between times steps unless the size of the time step changes. If the time step size is changed, the elements in the main diagonal must be updated. The coefficients can be considered constants because for a single phase liquid the viscosity and compressibility will not change significantly over the simulation period.

However, the right hand side vector of the matrix equation $\underline{Ap} = \underline{d}$ does change after each time step due to its dependence on P^n. This calculation falls directly into the divide and conquer category of parallel computation, and can be equally divided among the available processors. Parallel code for this operation will significantly speed up the execution of this code segment, but for the single phase liquid case the formation of matrix coefficients is not a large part of the total execution time. A more

interesting case is that of single phase gas flow.

For single phase gas flow the effects of pressure on compressibility and viscosity can not be ignored. However, these are weak nonlinearities and can usually be handled by time lagging the compressibility and viscosity one time step behind the current pressure calculation. Thus, the matrix coefficients are not constant between time steps and must be recalculated. This recalculation represents a significant portion of the total simulator execution time and the benefits of parallel processing become important.

To calculate the matrix coefficients new values of μ and c_t must be obtained for each grid block after each time step. This involves two table lookups for each grid block. Since the table lookup for one grid block is independent of the lookup of every other grid block, the lookups can be done in parallel using the divide and conquer technique. Since the pressure at adjacent grid blocks can be very close to the same value, the use of one table would result in a large number of memory collisions and thus a large number of processor wait states. Therefore, each processor should be supplied with its own copy of the viscosity and Z-factor tables.

Since the nonlinearities caused by the pressure dependent viscosity and compressibility are fairly weak, they can be treated by using their value based on the pressure at the previous time level. That is, the viscosity and compressibility values used in the matrix to calculate P^{n+1} are based on P^n. With only slight modification of the model, additional parallel speedup can be obtained. If instead of basing the viscosity and compressibility on P^n they are based on P^{n-1} the calculation of matrix coefficients and the solving of the matrix can be done in parallel using the pipelining technique. Also, if an iterative matrix solution technique is used, the viscosity and compressibilities can be based on something closer to P^n than P^{n-1} by using the incompletely converged P^n values.

An example of the types of speedup this pipelining method can give is suggested by the following example. Figure 6 shows the two-stage pipeline. First the matrix coefficients are calculated and then the matrix is solved in the normal sequential manner. Now suppose that for this sequential algorithm the forming of the coefficients required 8 units of time and the matrix solution required 12. By parallel techniques using 10 processors the coefficients are formed in 1 unit and the matrix is solved in 2. This gives a speedup of about 7, but by pipelining these two steps using 20 processors, a speedup of 10 can be obtained.

Multiphase Case

To illustrate application of parallel techniques to multiphase flow problems the IMPES model and its adaptation to a semi-implicit method is used. Every part of the procedure except the matrix solution for pressures will be considered as part of forming the matrix coefficients. This then includes the explicit calculation of saturations.

In well coning models and in cases where rapid saturation changes occur, the IMPES formulation often experiences instabilities. This is caused by the evaluation of matrix coefficients based on the values of saturation at the beginning of the time step. Methods such as given by Letkeman and Ridings [21] and also by MacDonald and Coats [23] alleviate this problems to some degree by implicitly extrapolating saturations to the next time level. But, as noted by MacDonald and Coats, progamming complexity increases considerably for methods of increasing degrees of implicitness. The size of the matrix that must be solved also increases. By using an iterative matrix solution method such as SOR, this same type of treatment can be applied to the IMPES method through the use of parallel programming. As the implicit pressure values coverge toward the solution, other processors simultaneously update the saturation (and pressure) dependent matrix coefficients. This gives a semi-implicit evaluation of saturations. This procedure should provide an increase in solution stability similar to the fully implicit formulation while maintaining the simplicity and smaller matrix size of the IMPES method.

As noted by MacDonald and Coats [23] and in a recent paper by Thomas and Thurnau [31], fully implicit treatment is usually only necessary for the grid blocks near the wellbore (or wherever else saturations or pressures are changing rapidly). Thomas and Thurnau suggest nonlinear stability analysis or numerical experiments as methods of determining which grid blocks require implicit treatment. The above described parallel method would also lend itself to an adaptive type application of simultaneous updating of coefficients. The processors could be set to only update matrix coefficient for those grid blocks that continue to experience rapid saturation changes.

Another application of parallel computations is in the forming of matrix coefficients for composition reservoir simulators. Fully implicit compositional fluid models are highly non-linear and are often solved through use of a Newton-Raphson method [10,17,18]. Phase equilibrium and fluid property calculations are repeatedly applied in the updating of matrix coefficients. Since these calculations are independent for every grid block, this is an ideal application for the divide and conquer method. Pipelining of the formation of matrix coefficients and matrix solution steps could also be performed.

MATRIX SOLUTION

Three methods of matrix solution are investigated to illustrate the application of parallel programming techniques to reservoir simulation. A well known method to perform parallel Gaussian Elimination is presented. The application of this method to sparse matrices and matrices using the D4 ordering is considered. Also, parallel methods of preconditioning of the Conjugate Gradient method and multigrid methods are discussed. Next, the Successive Over-Relaxation (SOR) iterative matrix solution method is discussed and three methods to perform this in parallel are presented. All examples in this section use a five point discretization of an areal two-dimensional, single phase, single well problem to generate a 441 equation test matrix. This problem was chosen to simplify the calculations and consequentially better illustrate the matrix solution methods. In most cases extending these methods to other problems is straight forward.

Gaussian Elimination

Gaussian elimination with backward substitution is a popular method of matrix solution particularily when the D4 ordering given by Price and Coats [27] is used.

Unfortunately, parallel methods of Gaussian elimination are best applied to dense matrices, since the amount of work that can be done in parallel is proportional to the bandwidth of the matrix. Since the matrices solved in reservoir simulators are sparse, the amount of work that can be done is parallel is somewhat limited. In reservoir simulation our matrices are generally of the diagonally dominant type and do not require row or column interchanges during the elimination step. Therefore, the following algorithms do not consider any pivoting strategies.

Gaussian Elimination uses the a(1,1) value of the first row to eliminate the entry directly below it in the second row as seen in Figure 7. This algorithm can be made parallel by simultaneously eliminating all non-zero entries of the first column. In this case a processor would be assigned to each row or a group of rows to perform the necessary multiplications and subtractions. The algorithm would then proceed from column 2 to column N-1. For a dense matrix this method allows a large amount of work to be done in parallel, but for a sparse matrix of bandwidth, B, only $\lfloor B/2 \rfloor$ processors are used. This somewhat limits the application of this method to reservoir simulation. Ortega and Voigt [26] give a good review of variations of the Guassian elimination method, but as they note, the additional speedup over the basic column sweep method described here does not justify the increase in complexity.

For the sparse structure of the example matrix, Gaussian Elimination has the effect of turning zero entries within the band to non-zero values. The changing of zero entries to non-zero values is termed fill. Using the D4 ordering, minimizes the fill and thus reduced the number of elements that must be eliminated. A parallel method which simultaneously performs elimination on different rows can also be applied to a matrix using the D4 ordering.

Figure 8 shows a matrix exhibiting the D4 ordering. This ordering is very conducive to parallel computation. The single diagonal in the upper left quadrant can be used to simultaneously eliminate elements in the lower left quadrant. The fairly large bandwidth exhibited in the lower right quadrant allows a number of parallel processors to perform elimination as in the parallel by column elimination procedure given previously. In the parallel by column procedure no attempt was made to prevent the elimination of zeros contained within the band. This would not be appropriate when a D4 ordering is used. A better approach is to test for zero entries and assign processors on a demand basis.

The backward substitution step of Gaussian Elimination can also be performed in parallel in a column oriented fashion using the same number of processors as the elimination step. The idea is that once the value of P(i) is known, it can be used in the $\lfloor B/2 \rfloor$ equations above it in order to obtain their solution. The algorithm requires only minor increases in synchronization compared to the elimination step.

Conjugate Gradient and Multigrid Methods

The preconditioning for the conjugate gradient method is generally based on the LU decomposition of a similar matrix. LU decomposition is very sequential in nature and difficult to perform with much degree of parallelism. In a recent paper, Efrat and Tismenetsky [12] presented a Biconjugate Gradient algorithm which uses a variation of an incomplete block factorization technique. They show this algorithm to be well suited for parallel machines and competitive with other conjugate gradient algorithms on sequential machines. The multigrid method has recently been gaining popularity as a matrix solution technique. It is based on the performance of incomplete relaxations on a number of different size grids. This alternates the damping of high and low frequency errors and theoretically converges in O(NM) operations [4]. This method is very suitable for parallel computers in several ways. The large number of interpolations required to transfer solutions from one grid to another are easily performed in parallel. Also, the relaxation methods, such as Gauss-Seidel and SOR, that are used to calculate the solutions on each of the grids are also very adaptable to parallel computation as will be discussed in the following section.

Successive Over-Relaxation (SOR)

The Successive Over-Relaxation (SOR) iterative matrix solution technique is widely used in reservoir simulation and a number of other disciplines. SOR has properties that allow parallel implementations in a number of different ways. Some of the more common methods have been described in a number of recent publications [1,2,13,26,28,29]. The red-black, the iteration pipelining, and the matrix partitioning methods of parallel SOR matrix solution are considered after a brief description of the classical SOR algorithm.

The SOR method in matrix form is given by:

$$(D - wE)\,\underline{P}^{k+1} = \left[(1 - w)D + wF\right]\underline{P}^{k} + w\underline{d} \qquad (5)$$

where k indicates the iteration level and w is the iteration parameter. If w=1, then Eq. (5) reduces to the Gauss-Seidel method. A typical SOR equation for the single phase, 2-dimensional example problem is given by:

$$P^{k+1}_{i,j} = (1-w)P^{k}_{i,j} - \frac{w}{a}\left[fP^{k+1}_{i,j-1} + bP^{k+1}_{i-1,j} + cP^{k}_{i+1,j} + gP^{k}_{i,j+1} - d\right] \qquad (6)$$

where a, b, c, d, f, g are functions of (i,j). The classical point SOR sequential algorithm applies Eq. (6), with appropriate boundary conditions, left to right, top to bottom over the grid space as seen in Figure 9.

If w=1, Eq. (6) reduces the residual at each grid block to zero. If w>1, then each grid block is over-relaxed. Over-relaxation can be thought of as an attempt to account for the fact that the values at surrounding grid blocks will eventually be relaxed. Note that both SOR and Gauss-Seidel use the k+1 values of pressure where available. The Jacobi method ignores these values and uses the pressure at iteration level k exclusively.

Red-Black

The red-black ordering the of the matrix has been widely used to allow vectorization of the SOR method

when a 5-point discretization is used [20, 24, 25]. The red-black method decouples the sequential SOR algorithm by making two passes through the matrix. First a red pass is made using the values of the black blocks at iteration level k. Next, the black pass is performed using the values of the red blocks previously calculated. This is illustrated in Figure 10. While this is not strictly equivalent to the sequential natural ordered SOR method, under normal circumstances the convergence has been shown to be identical [2].

The sequential red-black algorithm is easily made parallel by the "divide and conquer" parallel technique. The calculations of the red black passes are divided among the available processes. Recently, a number of line and block type variations using the red-black concept have been proposed [13], but the divide and conquer method remains the most commonly used.

There are, however, some important limitations of the red-black method. First, the red-black ordering only decouples problems which use a 5-point discretization in space. If another discretation, say the 9-point is used, other colors are required to decouple the problem [1]. Secondly, the division into a red and a black pass essentially reduces the problem size in half. This reduces the amount of useful work performed by each process and thus increases the overhead especially when a large number of processors are used. This problem is magnified when additional colors are required to decouple more complex problems.

Iteration Pipelining

Multiple processors can concurrently calculate different iteration levels provided their execution is carefully orchestrated [28,29]. Figure 9 shows the dependence of the calculations for the example problem. The value of P^{k+1} is based on its value at iteration level k, and the value of its surrounding four grid blocks. The value at grid block j-1 is the "last" value required when a sequential algorithm using the natural ordering of the grid blocks is employed. This means that the calculations at iteration level k+1 are independent of the equations of the matrix N+1 equations behind it. Thus the latency, or distance between independent problems is N+1 for this example problem.

To perform the matrix solution in parallel, new iteration levels are started before the previous ones finish calculating all NM equations. This can be thought of as a pipelining of iterations. Two methods of processor assignments are possible. The first method starts processors N+1 grid blocks or equations after each other. This group of processors will then "march" through the matrix until convergence is achieved (see Fig. 11). The second method divides the matrix into submatrices with a processor assigned to each. The processors will proceed sequentially through their own submatrix once they have been signaled to start by the proceeding submatrix. This method is illustrated in Figure 12. The submatrix method requires less synchronization than the marching method and thus would be expected to achieve a higher speedup. Both of these methods can easily be extended into line SOR algorithms.

Matrix Partitioning

The matrix partitioning method is similar to the submatrix iteration pipelining method described above. The difference is that the partitioning method makes no effort to enforce the latency. This results in Jacobi type iterations being performed at the partition boundaries. Since our matrix is strictly diagonally dominant we would expect convergence for this method to be slower than if SOR iterations were performed at the partition boundaries. Figure 13 shows how a matrix might be divided into partitions. It is recommended that each partition contain at least as many equations as the bandwidth of the matrix being solved.

The partitioning method has the favorable characteristic of allowing partitions that have "converged" to cease execution. During periods of transient flow, the partitions containing the sink/source wellbore terms will require a large number of iterations to obtain local convergence. Some partitions containing grid blocks far out in the reservoir may require only 1 or 2 iterations to obtain local convergence. In spite of this, the sequential SOR method performed over-relaxations on all grid blocks until global convergence is obtained. This results in a large number of redundant calculations. The partitioning method alleviates this problem through local convergence checking. However, to insure that global convergence is obtained once local convergence has been obtained, several full matrix iterations should be performed and global convergence criteria applied. For the test problem we observed speedups of between 5 and 1.5 during periods of transient flow.

Comparison of Results

The above algorithms were tested on the HEP parallel computer. Figure 14 shows a comparison of the speedups on the HEP. The speedups all begin to level off to a maximum speedup at about 10-12 processes. This is machine dependent since the HEP is roughly equivalent to a 10 processor parallel computer. Therefore, the relative performance of these four methods can only be compared in a general fashion up to the 10 process limit. Overall, speedups of over 6 were found to be possible using 10 processes.

The Red-Black, Submatrix, and Partitioning algorithms are seen to be roughly equivalent in terms of speedups. The speedup for the Marching algorithm is much less than that found for the others. This is due to the large amount of process wait time built into the algorithm. In this algorithm all the processes were required to wait until every other processes had completed its computations before proceeding to the next grid block. This large amount of synchronization resulted in a large amount of waiting and seriously degraded the performance of the algorithm. The Submatrix algorithm, while based on the same idea of the pipelining of iterations, performed much better. This algorithm required only a little synchronization to start the processing of the submatrix. Once a submatrix processor was signaled to start it proceeded through the calculations of the entire submatrix without any further communication with the other processes. The Partitioning algorithm shows a sharp decrease in performance well before the machine limitation of 10 processes. This, therefore, is a general, not a machine dependent conclusion about this algorithm. This effect is believed to be caused by

the increase in Jacobi type iterations being performed at the partition boundaries as the number of partitions increases.

From this study the two best algorithms appear to be the Red-Black and the Submatrix. However, there are also limitations to both of these algorithms. The Red-Black algorithm is limited to the 5-point discretized problem, with additional colors having to be added to decouple more complex problems. Also, the Red-Black algorithm is not strictly equivalent to the sequential, natural ordered problem, and amount work given to each process is half of that for the Submatrix algorithm. This is likely to have resulted in the decreasing slope seen in Fig. 14 as the number of processes approaches the machine limit of 10. The Submatrix algorithm continues to exhibit straight line behavior at the 10 process limit. Based on these tests the Submatrix algorithm is the most general and the best performer but this algorithm has one important limitation. The maximum number of processes that can be used is limited to the average number of iterations required for convergence. For this example problem about 40 iterations were required to satisfy the convergence criteria. This is not that serious a limitation for the currently available machines which have from 4 to 128 processors. There also exists the possibility of combining algorithms such as the Red-Black and the Submatrix to extend the number of processors that can be used for matrix solution.

CONCLUSIONS

Parallel (MIMD) type computers are commonly believed to be the next generation in the evolution of supercomputers. The physical limits of the speed of a single processor are rapidly approaching. The use of multiple processors is now seen as possibly the only way to significantly increase the speed of future computers. Algorithm development for parallel computers is still in it infancy since most algorithms simply mimick those which were developed specifically for use on sequential machines. The development of parallel algorithms for reservoir simulation must be considered if the next generation of supercomputers is to be used effectively by the petroleum industry.

A number of obstacles exist for parallel algorithm development. The two general types of memory, shared and distributed, require somewhat different information to control the execution of processes and the sharing of information. This means that no standard has been developed, so the code written on one machine will not be directly transferable to any other machine. Automatic vectorization of code is very well developed for several type of vector computers and is also currently available for some parallel computers. At present any other type of parallel process must be done by the user and the compiler automation of this task is fairly distant.

Parallel computer have been shown to be applicable to reservoir simulation in a number of ways. In addition to being able to perform all the tasks of a vector (SIMD) type computer they also can perform a number of new tasks such as the simultaneous computation of matrix coefficients and matrix solutions. Three algorithms for the SOR matrix solution method have been developed in detail to show that a large number of approaches can be taken in the development of parallel algorithm. Maximum speedups of approximately 7 were shown possible for the 10 processor HEP parallel computer. Speedups of over one order of magnitude are currently possible on several hypercube type computers. These computers have already demonstrated that supercomputer speed is possible at much less than supercomputer prices when a number of cheap processes are used rather than a single very expensive processor.

NOMENCLATURE

A	pressure matrix
a,b,c,f,g	matrix coefficients
B	matrix bandwidth
c_t	total system compressibility, psi^{-1}
$\underline{d}$	right hand vector of pressure matrix equation
D	diagonal matrix of A
E	lower diagonal submatrix of A
E(p)	efficiency of an algorithm using p processors
F	upper diagonal submatrix of A
Δt	change in time, sec.
Δx	change in x-direction, ft.
Δy	change in y-direction, ft.
k	permeability, md.
M	number of y directional grid blocks
N	number of x directional grid blocks
p	number of processors
P	pressure, psia
S(p)	speedup of an algorithm using p processors
w	SOR iteration parameter
Z	real gas compressibility factor

Greek

μ	viscosity, cp

Subscripts

i	x-direction
j	y-direction
t	total

Superscripts

k	iteration level
n	time level

ACKNOWLEDGEMENTS

Computer time was provided by Los Alamos National Laboratory Computing Division and technical support was given by Ann Hayes. The example in Fig. 8 was provided by Al Reynolds. Stuart Scott is a Shell Doctoral Fellow at The University of Tulsa. Additional support for this work was provided by the Tulsa University Fluid Flow Projects and the SPE graduate fellowship program. The authors gratefully acknowledge this support.

REFERENCES

1. Adams, L., and J. Ortega, "A Multi-Color SOR Method for Parallel Computation," Proc. Int. Conf. on Par. Proc., pp. 53-56 (1982).

2. Adams, L., and H. F. Jordan, "Is SOR Color-Blind?," SIAM J. Sci. Stat. Comput., pp. 490-506 (April 1986).

3. Aziz, Khalid, and A. Settari, Petroleum Reservoir Simulations, London: Applied Science Publishers LTD (1979).

4. Behie, A. and P. A. Forsyth, "Multigrid Solution of the Pressure Equation in Reservoir Simulation," SPEJ, pp. 623-632 (August 1983).

5. Burden, R. L., J. D. Faires, and A. C. Reynolds, Numerical Analysis, Boston: Prindle, Weber, and Schmidt (1981).

6. Calahan, D. A., "Performance of Linear Algebra Codes on the CRAY-1," SPEJ, pp. 558-564 (October 1981).

7. Chen, S. S., J. J. Dongarra, and C. C. Hsiung, "Multiprocessing Linear Algebra Algorithms on the CRAY X-MP-2: Experiences with Small Granularity," Journal of Parallel and Distributed Computing, 1, pp. 22-31 (1984).

8. Christ, N. H. and A. E. Terrano, "A Micro-Base Supercomputer," BYTE, 4, pp. 145-160 (1986).

9. Coats, K. H., "Reservoir Simulation: State of the Art," SPE Distinguished Author Series (1982).

10. Coats, K. H., "An Equation of State Compositional Model," SPEJ, pp. 363-376 (October 1980).

11. Crichlow, H. B., Modern Reservoir Engineering - A Simulation Approach, Englewood Cliffs, New Jersey: Prentice-Hall, Inc. (1977).

12. Efrat, I. and M. Tismenetsky, "Parallel Iterative Linear Solvers for Oil Reservoir Models," IBM J. Res. Develop, pp. 184-192 (March 1986).

13. Evans, D. J., "Parallel SOR Iterative Methods," Parallel Computing, pp. 3-18 (August 1984).

14. Flynn, M. J., "Very High-Speed Computing Systems," Proc. IEEE, 54, pp. 1901-1909 (1966).

15. Hwang, K., Supercomputers: Design and Application, Silver Spring, MD: IEEE Computer Society (1984).

16. Hwang, K. and F. A. Briggs, Computer Architecture and Parallel Processing, New York: McGraw-Hill Book Company (1984).

17. Jones, J. R., Computation and Analysis of Single Well Responses for Gas Condensate Systems, Ph.D. Dissertation, University of Tulsa, Tulsa, Oklahoma (1986).

18. Jones, J. R. and R. Raghavan, "Interpretation of Flowing Well Response in Gas Condensate Wells," SPE 14204 presented at the 60th Annual Technical Conference and Exhibition of the SPE of AIME, Las Vegas, NV, September 22-25, 1985.

19. Jordan, H. F., "Experience with Pipelined Multiple Instruction Streams," Proc. IEEE, 72, pp. 113-123 (January 1984).

20. Kendall, R. P., J. S. Nolen, and P. L. Stanat, "The Impact of Vector Processors on Petroleum Reservoir Simulation," Proc. of the IEEE, pp. 85-89 (January 1985).

21. Letkeman, J. P. and R. L. Ridings, "A Numerical Coning Model," SPEJ, pp. 418-424 (December 1970).

22. Levesque, J. M., "Reservoir Simulation on Supercomputers," SPEJ, pp. 275-279 (April 1985).

23. MacDonald, R. C. and K. H. Coats, "Methods for Numerical Simulation of Water and Gas Coning," SPEJ, pp. 425-436 (December 1970).

24. Mrosovsky, I., J. Y. Wong, and H. W. Lampe, "Construction of a Large Field Simulator on a Vector Computer," JPT, pp. 2253-2264 (December 1980).

25. Nolen, J. S., D. W. Kuba, and M. J. Kascic, "Application of Vector Processors to Solve Finite Difference Equations," SPEJ, pp. 447-453 (August 1981).

26. Ortega, J. M., and R. G. Voigt, "Solution of Partial Differential Equations on Vector and Parallel Computers," SIAM Review, pp. 149-240 (June 1985).

27. Price, H. S. and K. H. Coats, "Direct Methods in Reservoir Simulation," SPEJ, pp. 369-383 (June 1974).

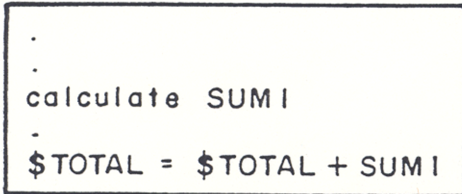
28. Scott, S. L., Computer Design to Optimize Matrix Operations and Solution of a Two-Dimensional Single Phase Reservoir Simulator, M. S. Thesis, University of Tulsa (1985).

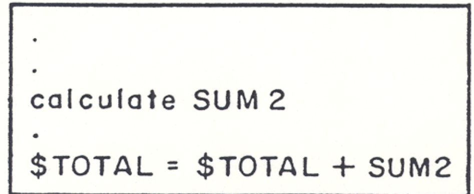
29. Scott, S. L., J. J. Haley, and H. Demuth, "Parallel SOR Algorithms for Solution of Sparse Matrix Problems," unpublished paper (1985).

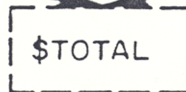
30. Thomas, G. W., Principles of Hydrocarbon Reservoir Simulation, Trondheim: TAPIR (1977).

31. Thomas, G. W. and D. H. Thurnau, "Reservoir Simulation Using an Adaptive Implicit Method," SPEJ, pp. 759-768 (October 1983).

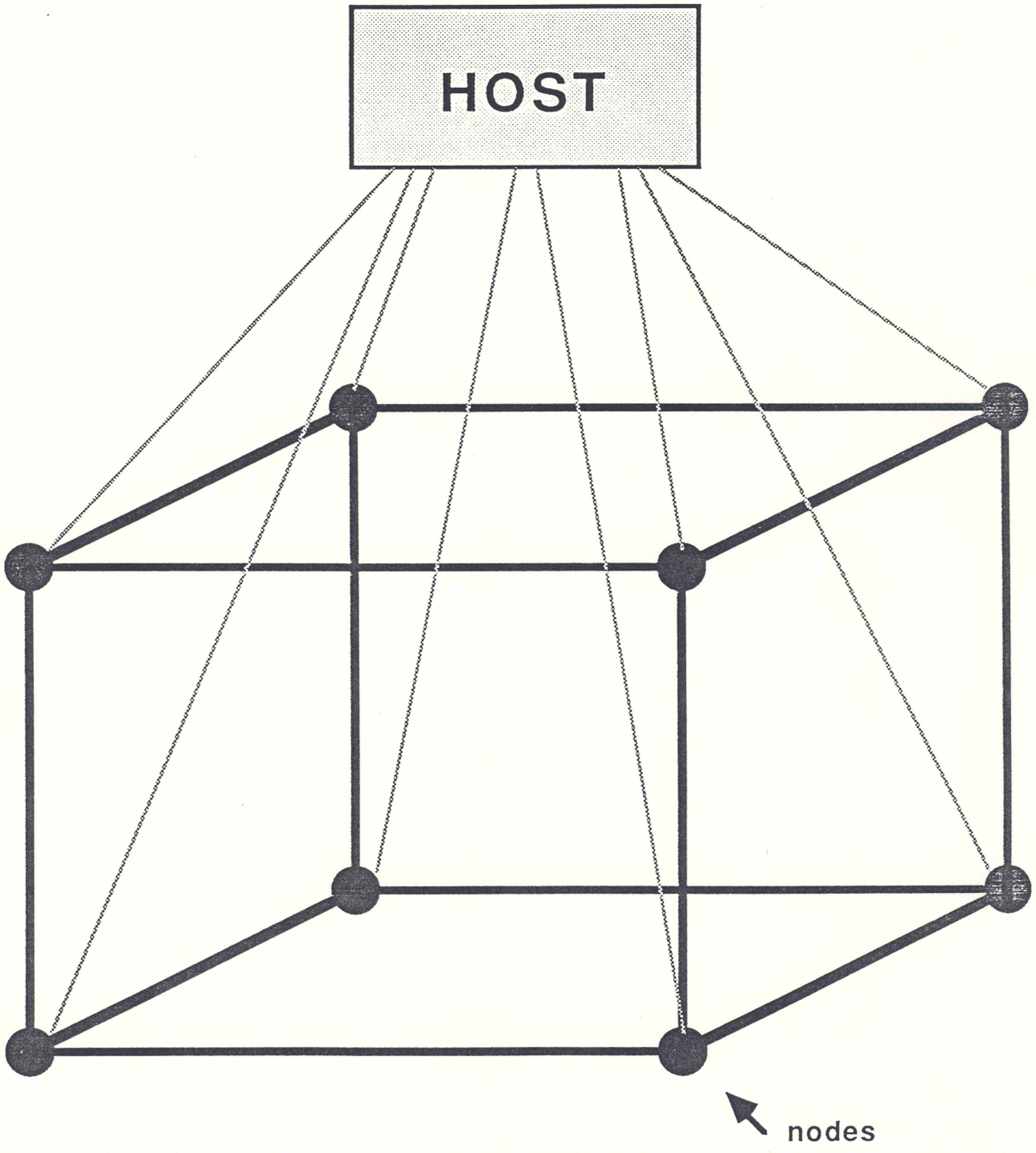
32. Woo, P. T. and J. M. Levesque, "Benchmarking a Sparse Elimination Routine on the Cyber 205 and the Cray 1-S," SPEJ, pp. 743-745 (October 1983).



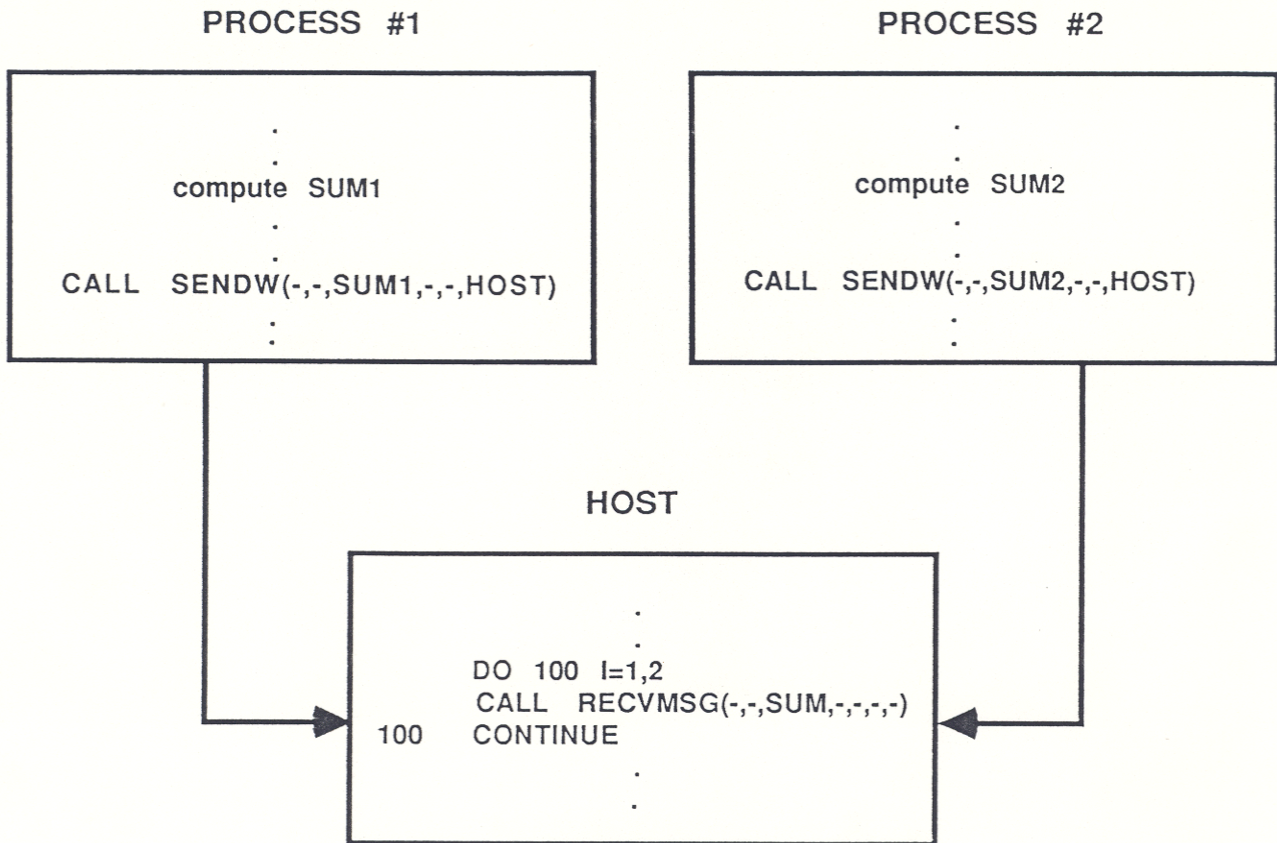




1. **Example of HEP Asynchronous Variables**



2. Hypercube Communication Network



3. Example of Message Passing on the iPSC Hypercube

SEQUENTIAL:

$$SUM = \underbrace{\boxed{\sum_{i=1}^{10,000} A_i B_i}}_{P_1}$$

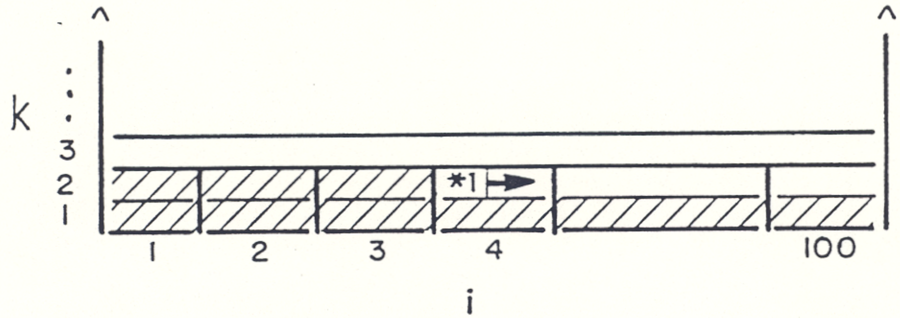
PARALLEL:

$$SUM = \underbrace{\boxed{\sum_{i=1}^{1,000} A_i B_i}}_{P_1} + \cdots \underbrace{\boxed{\sum_{i=9,001}^{10,000} A_i B_i}}_{P_{10}}$$

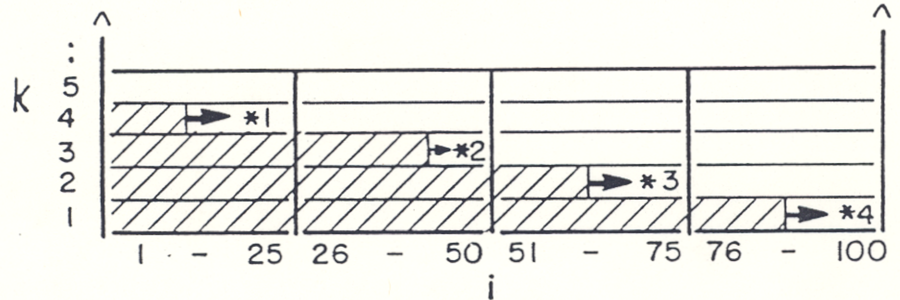
```
     $TOTAL = 0
     INC = 1000
     START = 1
     DO 100 I=1,10
     END = I*INC
     CREATE XSUM(START,END,$TOTAL)
100  START = END + 1

     SUBROUTINE XSUM(START,END,$TOTAL)
     SUM = 0
     DO 100 I=START,END
100  SUM = SUM + A(I)*B(I)
     $TOTAL = $TOTAL + SUM
     RETURN
```

4. Example of Divide and Conquer Parallel Programming

SEQUENTIAL:



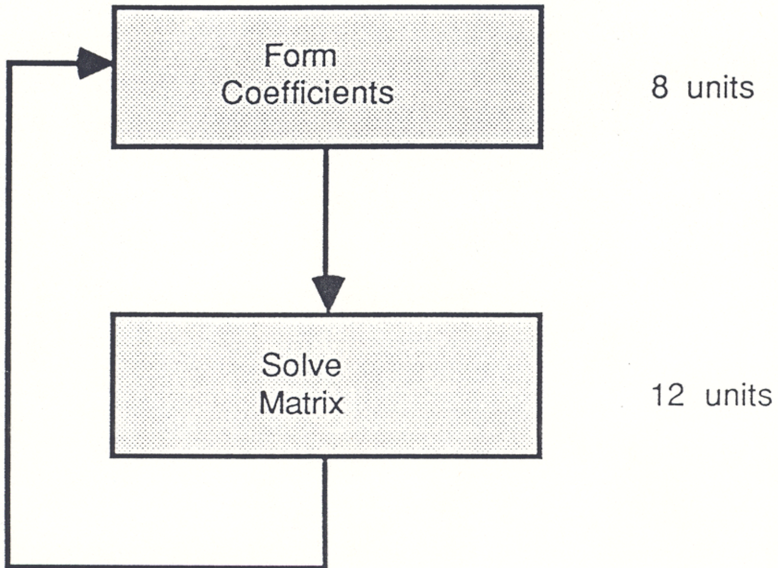
PARALLEL:



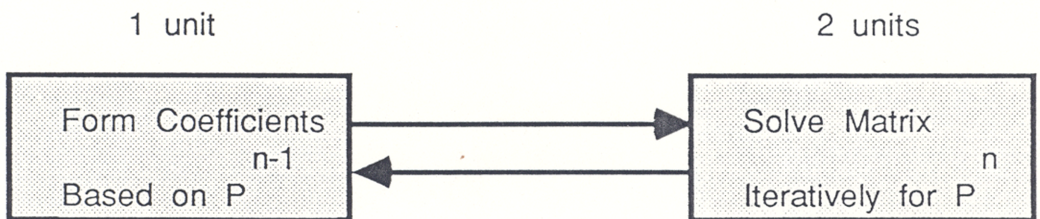

```
    BETA   = DELTA/ALPHA/DELTAX**2.0
    PNUM = 4
    INC = 25
    DO 100 I = 1,PNUM
    $NUM = I
100 CREATE STAGE($NUM,INC,BETA,P,$K)

    SUBROUTINE STAGE($NUM,INC,BETA,P,$K)
    NUM = $NUM
    ISTART = 1 + (NUM - 1)*INC
    IEND = ISTART + 24
    $K(NUM) = 1
10  IF(NUM.EQ.1) GO TO 30
20  KPREVIOUS = $K(NUM - 1)
    $K(NUM - 1) = KPREVIOUS
    IF(KPREVIOUS.GT.K) GO TO 30
    IF('not converged') GO TO 20
    GO TO 999
30  DO 40 I = ISTART,IEND
40  P(I) = P(I) + BETA*(P(I-1) - 2.0*P(I) + P(I+1))
    K = $K(NUM)
    K = K + 1
    $K(NUM) = K
    IF('not converged') GO TO 10
999 RETURN
    END
```

5. Example of Pipelined Parallel Programming

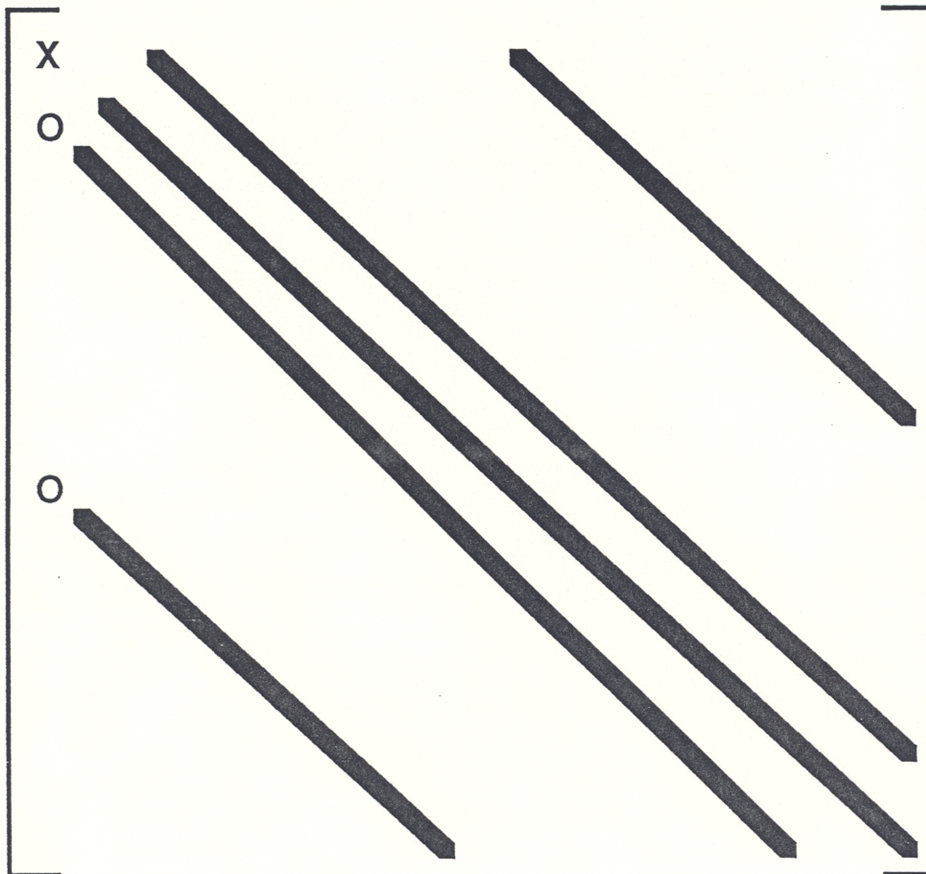
SEQUENTIAL



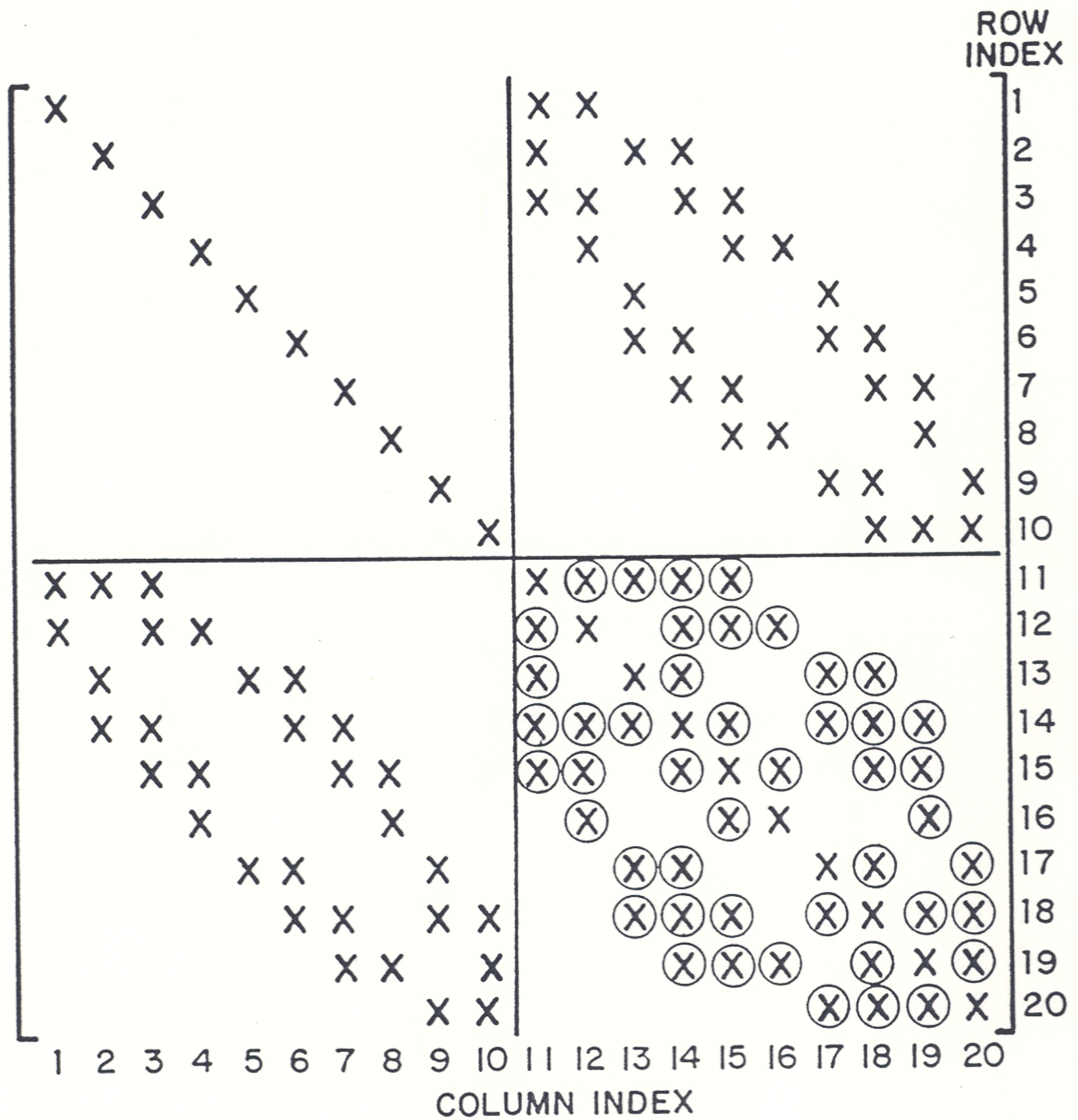
PARALLEL



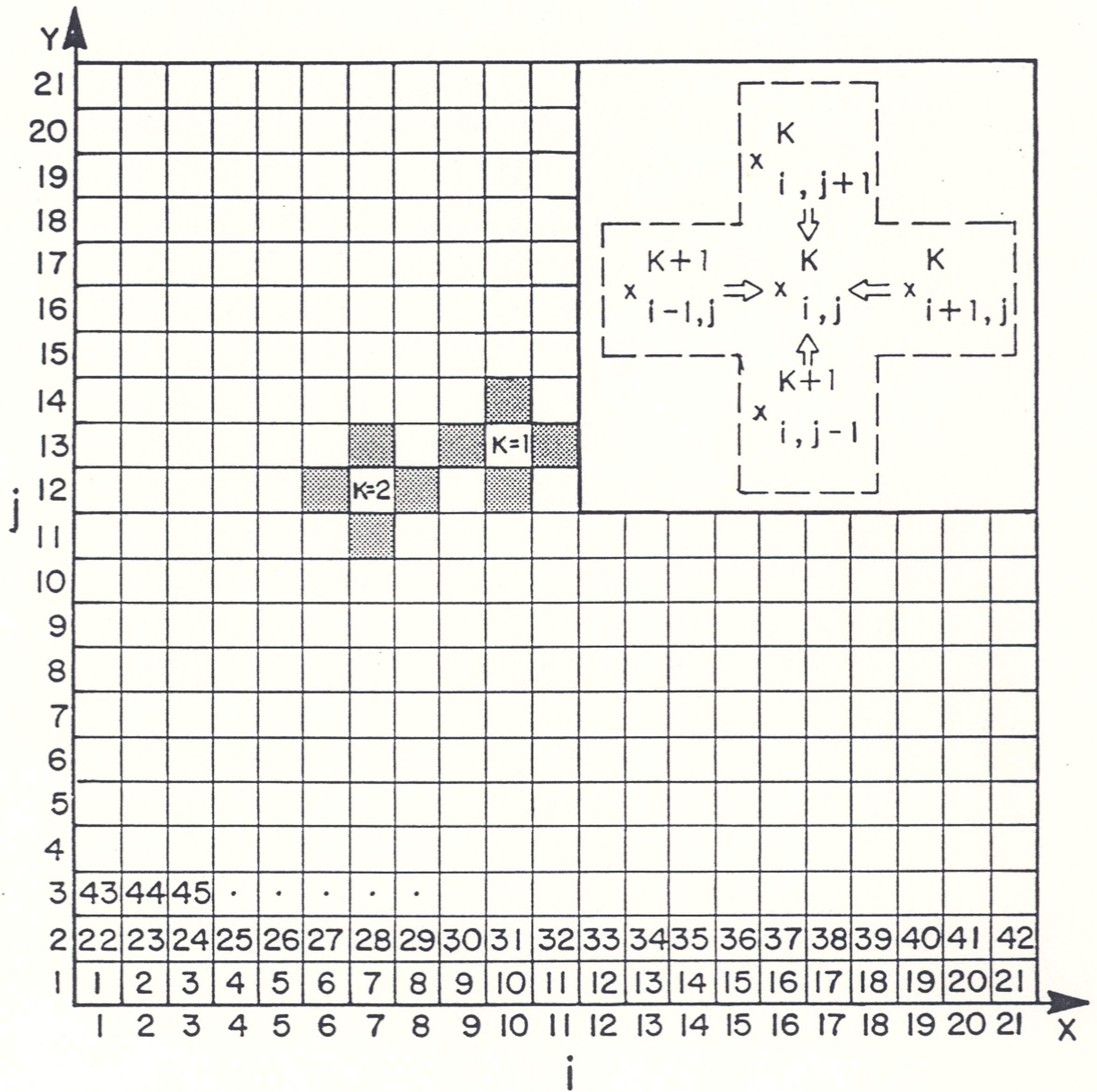
6. Pipelining the Formation of Matrix Coefficients and the Matrix Solution Steps



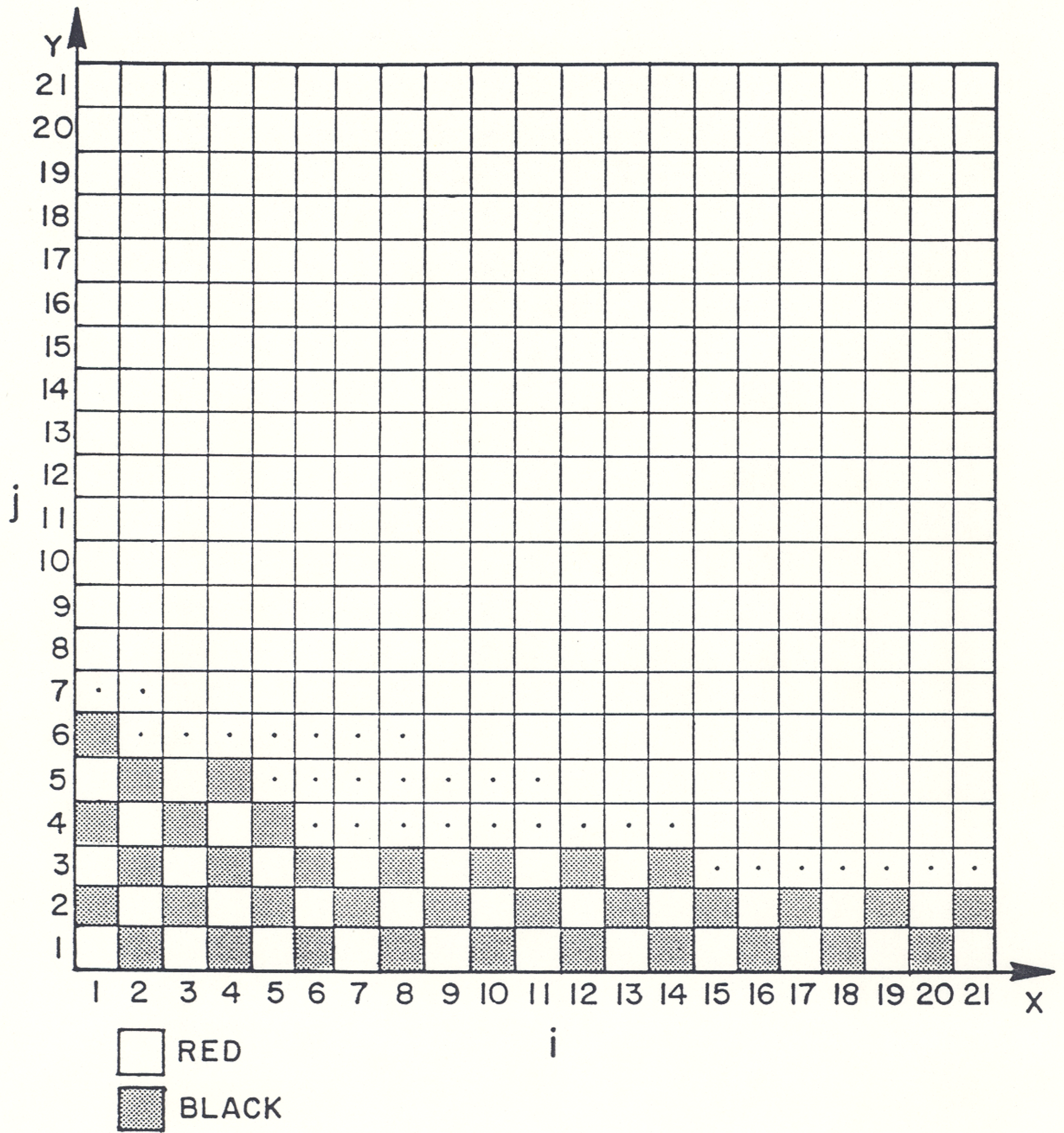
7. Example of Parallel Gaussian Elimination



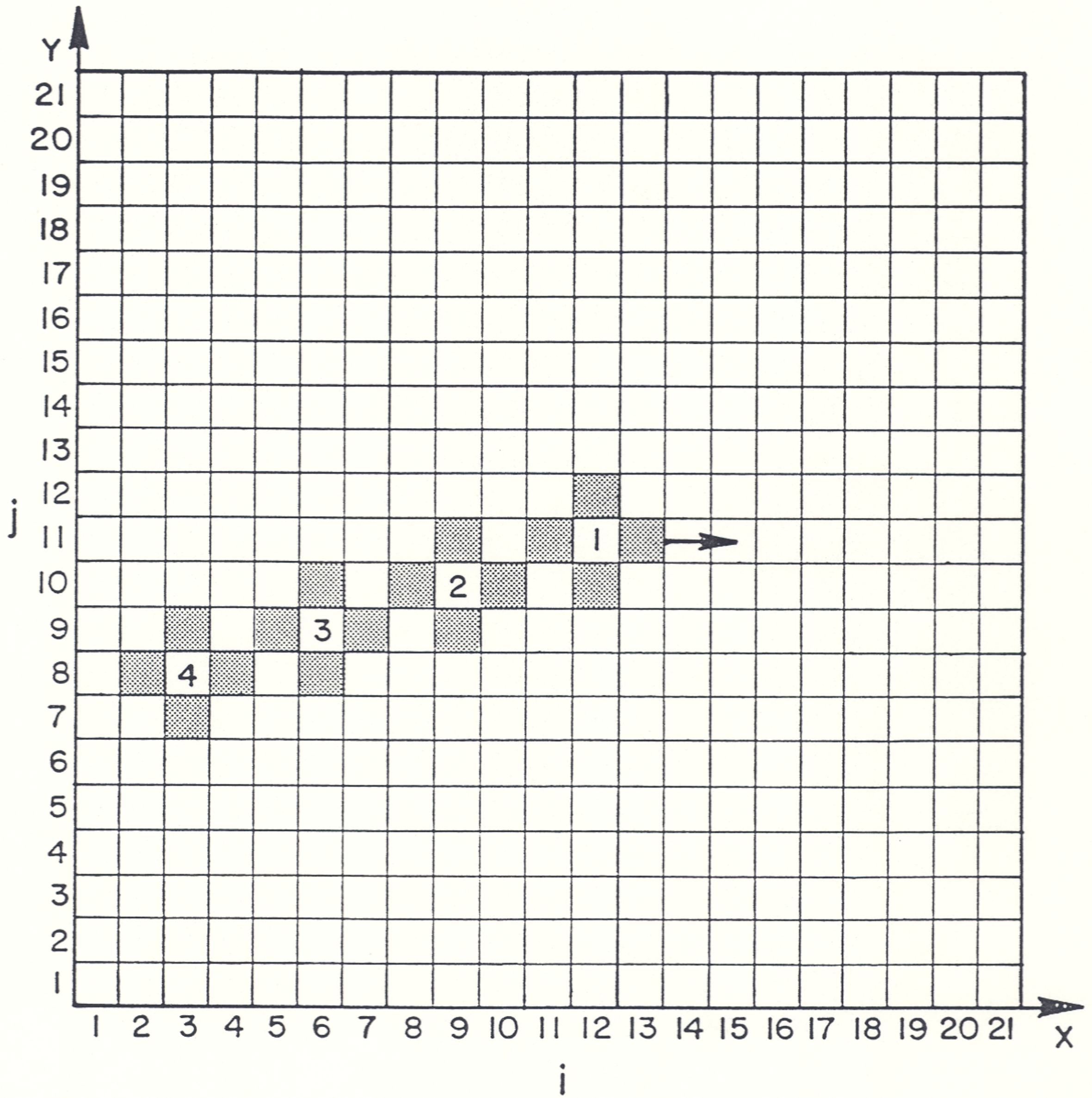
8. D4 Ordered Matrix



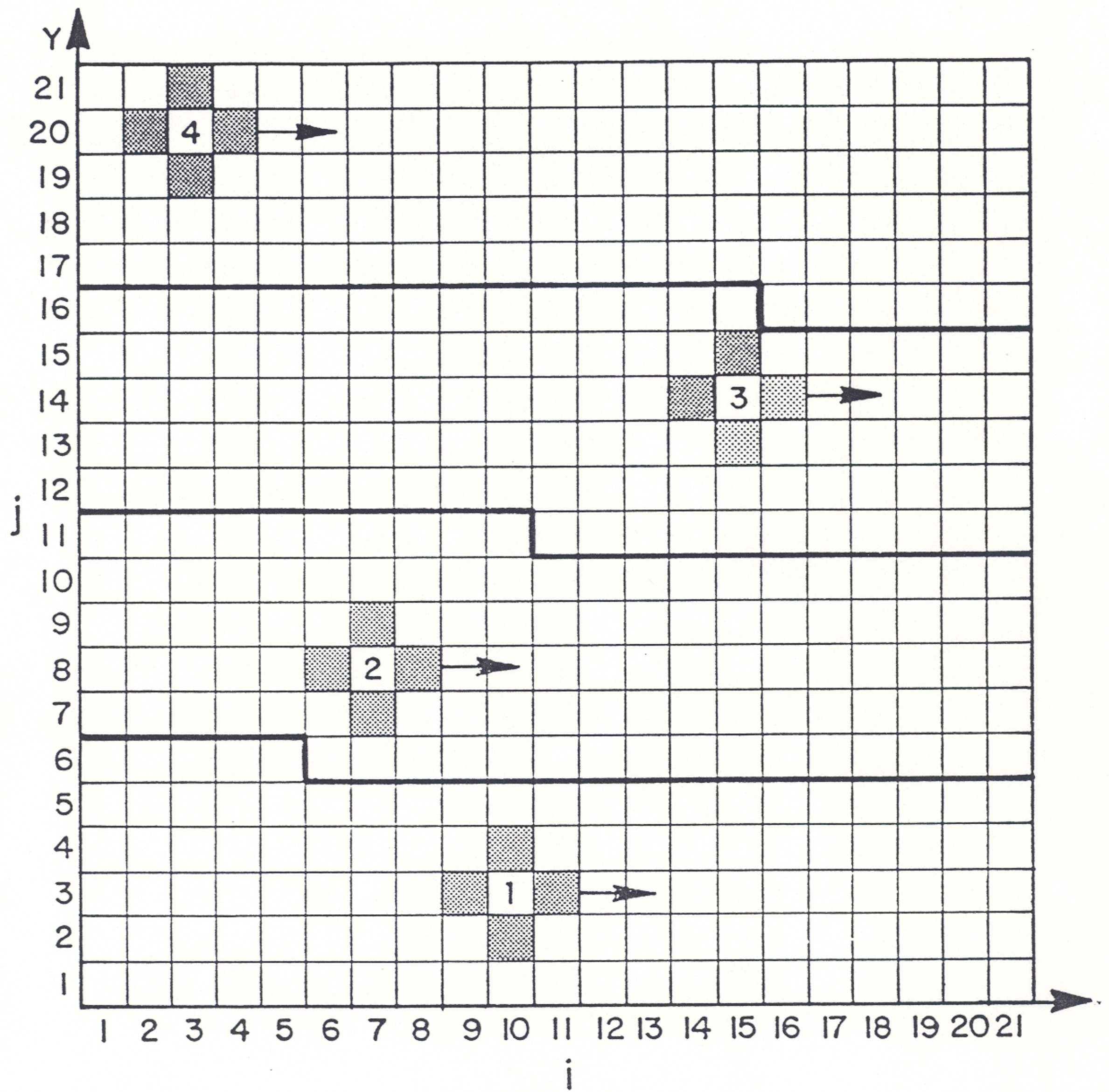
9. Sequential SOR Algorithm



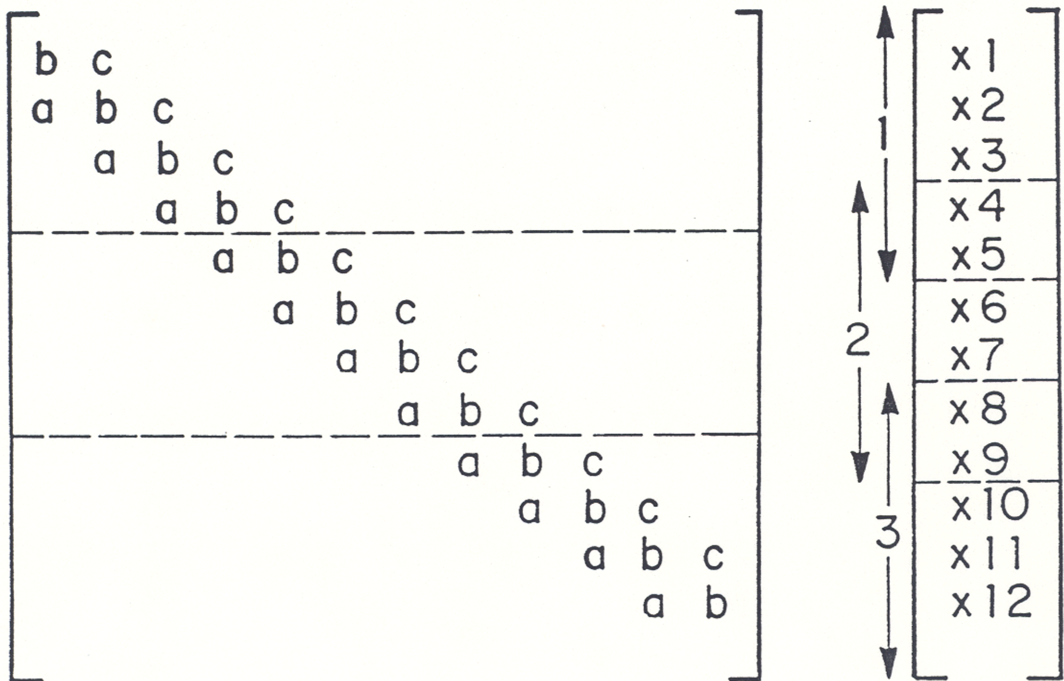
10. Red-Black Ordered Grid Space



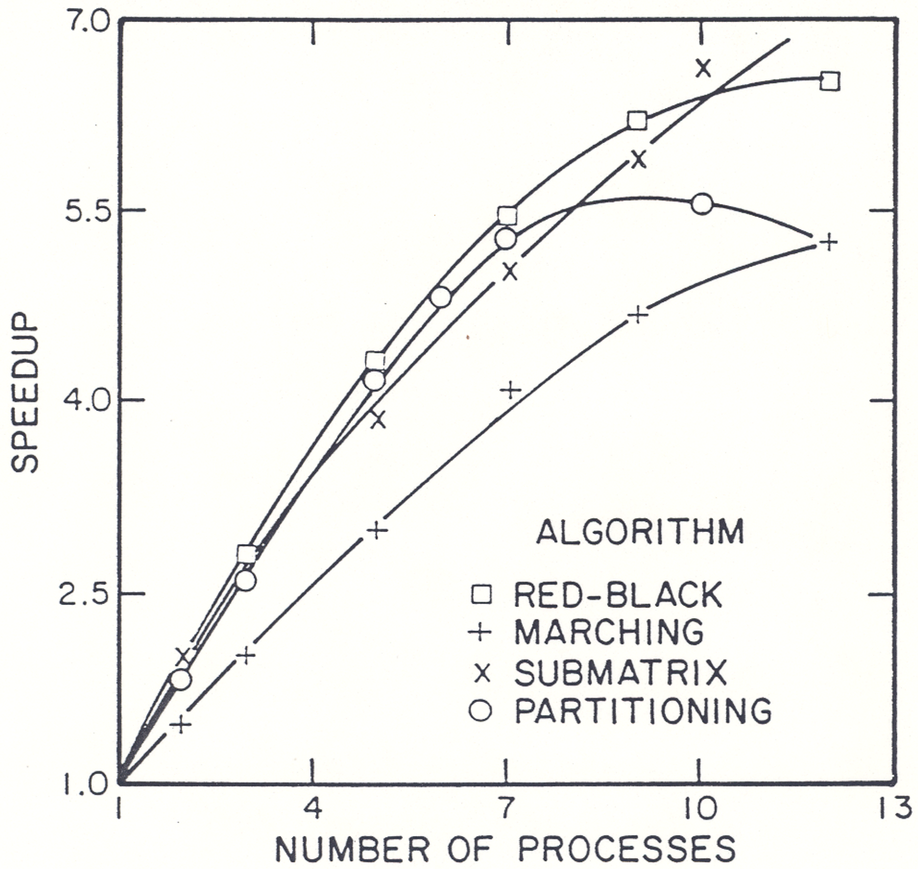
11. Marching Parallel SOR Algorithm



12. Submatrix Parallel SOR Algorithm



13. Example of Matrix Partitioning



14. Comparison of Parallel Speedup for SOR Algorithms