The Conjugate Gradient Method for Solution of Large Sparse Systems of Linear Equations on Hypercubes Applied to Reservoir Modeling

Roger L. Wainwright

Computer Science Department
The University of Tulsa

Amoco Production Company
Tulsa, Oklahoma

Proceedings of the ACM South Central Regional Conference, November 16-18, 1989, Tulsa, Oklahoma.

ABSTRACT

Finite element discretization of a reservoir model yields linear equations of the form Ax=b, where A is a large, sparse, banded matrix. This paper describes two iterative methods for such equations, the conjugate gradient and successive overrelaxation (SOR) techniques. Each method was programmed for an NCUBE computing system with 64 processors and 32 megabytes of distributed memory, and for an IBM 3090 running the VM/CMS operating system. The conjugate gradient program performed substantially better than SOR on both computing systems, and the NCUBE proved superior to the IBM 3090 for this application both in terms of speed and the size of the problems it could solve.

The conjugate gradient program arrives at a solution about four times faster on the NCUBE than on the IBM 3090, giving the NCUBE a cost/performance rating about 100 times better than that of the IBM 3090 for this application. In addition the NCUBE accommodates systems of equations over three times as large as the IBM 3090 can handle because of memory restrictions imposed by VM/CMS.

1. INTRODUCTION

Many scientific problems require the solution of linear systems of equations where the coefficient matrix is large, sparse and banded. Consider, for example, sparse matrix problems resulting from approximations to certain partial differential equations. The test problems used in this paper arise from a parabolic partial differential equation which is the two dimensional, single phase diffusion equation from petroleum reservoir simulation. The reader is referred to References 4,7,14-16 for details of the mathematical development for reservoir modeling. The partial differential equation is discretized in space using an irregular block centered grid. If central differences are used to approximate the space derivatives and a backward difference is used to approximate the time derivative, then a five point difference equation is obtained. A similar approximation leads to a nine point discretization. The vast majority of reservoir simulations are two-phase, non-linear, non-symmetric models. In this paper the more basic single phase, linear problem is considered.

Consider Figure 1 which shows a 5 by 5 x-y grid space representing the physical layout of a small reservoir. The pressure, x, at each grid block at any time is given by the equation:

$$ax_{i,j-1} + bx_{i-1,j} + cx_{i,j} + dx_{i+1,j} + ex_{i,j+1} = f$$

where the values for a, b, c, d, e, and f come from the bands in the sparse matrix, and are constant over a given time step for grid block i,j. From the above formula, it is apparent the pressure in the reservoir at a given grid point is a linear function of the pressures at the surrounding grid points. The linear equations for all 25 grid points in Figure 1 can be arranged as a sparse system of linear equations, Ax = f as shown in Figure 2. In this paper reservoirs of size N by M are considered. This produces sparse linear systems of size NM by NM, where NM represents N times M. Figures 1 and 2 illustrate the simple case of N = 5 and M = 5.

Various parallel algorithms have been developed recently for solving sparse linear systems using iterative techniques. The Successive Over-Relaxation (SOR) iterative matrix solution technique is widely used in reservoir simulation as well as in many other disciplines [6,13,15,20]. Recent attention, however, has been given to the method of conjugate gradients (CG) for solving sparse linear systems on distributed memory multiprocessor systems [2,3]. This paper reports my findings of using the conjugate gradient method on a hypercube multiprocessor system for solving large sparse banded linear system of equations as applied to reservoir modeling.

The work done in this paper was performed on the NCUBE/10 hypercube located at the Amoco Research Center. This machine is currently a six-cube (64 nodes) system. In this paper the terms "node" and "processor" will be used interchangeably. The NCUBE/10 is a hypercube architecture parallel processing system. Each node is a 32-bit processor with 512 KBytes of local memory. Each processor is capable of register-to-register operations at the rate of one-half million floating point or two million fixed point operations per second [11].

Hypercubes are multiprocessor arrays (see Ref. 1,10,12,17-19) with elaborate interconnection features connecting the nodes together. In general, a hypercube parallel computer is a collection of sequential computing nodes joined to their neighbors in an n-dimensional cube. An n-cube consists of 2^n processor nodes numbered using n-bit binary numbers ranging from 0 to 2^{n-1}. Two processors (neighbors) are

directly linked by a bi-directional communication channel if their binary number representation differs by exactly one bit. Thus in an n-cube, each node is directly connected (neighbor) to n nodes. The nodes of a hypercube are completely independent machines with their own processor memory, I/O, and resident copy of the operating system.

The remainder of this paper is summarized as follows. In Section 2 the method of conjugate gradients is presented. Section 3 discusses the communication requirements for the conjugate gradient algorithm, and Section 4 discusses the implementation of the CG algorithm on a hypercube. Results are presented in Section 5, and summary and conclusions are given in Section 6.

2. THE METHOD OF CONJUGATE GRADIENTS

I have implemented the conjugate gradient method as described in Reference 3. This implementation is identical to the traditional conjugate gradient method except for the reorganization of some of the calculations. The reorganization of calculations has the benefit of requiring fewer communication steps among processors compared to the traditional implementation of the algorithm.

In general the problem is to solve the linear system, Ax=b. The conjugate gradient method is described as follows.

Step 0. Initially choose x_0 and let $r_0 = p_0 = b - Ax_0$. Then compute $\langle r_0,r_0\rangle$. Then

For k = 0,1,2,...

1. form $q_k = Ap_k$
2. form $\langle p_k,q_k\rangle$ and $\langle q_k,q_k\rangle$
3. a) $\alpha_k = \langle r_k,r_k\rangle/\langle p_k,q_k\rangle$
   b) $\beta_k = (\alpha_k\langle q_k,q_k\rangle/\langle p_k,q_k\rangle) - 1.$
   c) $\langle r_{k+1},r_{k+1}\rangle = \beta\langle r_k,r_k\rangle$
4. a) $r_{k+1} = r_k - \alpha_k q_k$
   b) $x_{k+1} = x_k + \alpha_k p_k$
   c) $p_{k+1} = r_{k+1} + \beta_k p_k.$

Here r_k is the residual vector associated with the approximate solution vector, x_k, at each iteration. r_k is defined as $b - Ax_k$ and must be null when x_k is the exact solution. Thus, a suitable criterion for halting the iterations is when $[\langle r_k,r_k\rangle/\langle b,b\rangle]^{1/2} < \varepsilon$, where ε is very small. In all my examples, I used a tolerance of $\varepsilon = 10^{-7}$ using 16 digit (double precision) arithmetic. A data flow diagram showing the data dependences in the CG algorithm is shown in Figure 3.

3. COMMUNICATION REQUIREMENTS OF THE CG ALGORITHM

Consider again the simple reservoir model in Figure 1. Suppose each grid row is assigned to a single processor. This means that each of the five processors is assigned five of the equations to solve in the sparse matrix shown in Figure 2. In general, the problem is to solve a reservoir model on an N by M grid space, which corresponds to solving an NM by NM sparse linear system. If p processors are available to process the M rows of the grid then each processor is responsible for M/p grid rows. Applying this strategy to the method of conjugate gradients implies that the matrix A and the vectors x,b,r,p and q are partitioned and distributed equally as possible among the available processors.

The general conjugate gradient method as described in Section 2 has three types of operations:

(1) matrix-vector operations in calculating Ax_0 in step 0, and Ap_k in step 1.

(2) vector-vector operations in calculating $\langle r_0,r_0\rangle$ in step 0, and $\langle p_k,q_k\rangle$, $\langle q_k,q_k\rangle$ in step 2, and

(3) scalar calculations as found in step 3.

The matrix-vector product involves the formation of a sparse dot product of each row of A with the dense vector x in step 0 and the dense vector p in step 1. In order to do this, interprocessor communication is required to obtain nonlocal components of the x and p vectors. The nodes are arranged in a ring topology where each processor exchanges exactly one row of vector x with each of its two neighbors. This is done once in step 0. Each processor exchanges exactly one row of vector p with each of its two neighbors in step 1 for each of the k iterations. The process of exchanging nonlocal components of vectors x and p with neighbors is called EXCHANGEx and EXCHANGEp, respectively.

This concept is illustrated in Figure 4. The 5 by 5 sample reservoir shown in Figure 1, which results in the 25 by 25 sparse linear system in Figure 2 is further illustrated in Figure 4. Here the vector to be exchanged is distributed evenly across each of the five processors. Directed lines show the required row exchanges between the processors. For example, processor two requires row 5 from processor one, and row 11 from processor three in order to perform its calculations. For simplicity the processors are numbered one through five in Figure 4, rather than the actual node numbering for a ring.

The vector-vector (inner product) calculations require global information and are inherently sequential. A series of communication steps is performed to calculate a global sum, and to broadcast the global sum to each of the processors. This process is called GS_GB (global sum and global broadcast). There are primarily two ways in which this can be accomplished. The traditional GS_GB algorithm is performed by viewing the hypercube as a tree of processors. This concept is illustrated in Figure 5 for a hypercube of dimension d = 4. Each processor calculates its local inner product, then a series of d communication steps is performed to obtain a global sum. In Figure 5, after d steps the global sum is accumulated in node zero. Next another series of d communication steps is performed to broadcast the inner product (global sum) back to each processor. The entire process requires a total of 2d communication steps. Notice most of the processors are idle most of the time. A more efficient mechanism is presented in Reference 3 using a process called Exchange-Add to accomplish the global sum and global broadcast. Details of the Exchange-Add version of GS_GB are shown in Figure 6 for a hypercube of dimension d = 4. In this algorithm a total of d communication steps are required, and every processor during each step is working by exchanging data with a neighbor. Each of the d communication steps involves a bidirectional exchange. Thus the same number of messages are used in both the Exchange-Add version and the traditional version of the GS_GB algorithm. The advantage of the Exchange-Add algorithm is that the bidirectional exchange messages can be performed at the same time. Generally, the Exchange-Add version of GS_GB performs better than the traditional tree structure GS_GB [3]. The Exchange-Add version of the GS_GB was used in all of the examples presented in this paper.

4. IMPLEMENTATION OF THE CG ALGORITHM ON A HYPERCUBE

The algorithm for implementing the conjugate gradient method is illustrated in Figures 7 and 8. Figure 7 depicts the role of the cube manager. The cube manager queries the user for the number of processors to be used to solve the problem. Input describing the reservoir model is read from an external file. This information is passed one time to each of the p processors. The cube manager then cycles through a loop capturing and saving results of the simulation from the processors until a termination notice is received.

Figure 8 shows the outline of the algorithm for each of the p processors. First, each processor determines its processor number. Using this information its previous and next neighbors are determined for use later with the EXCHANGEx and EXCHANGEp routines. Next, input describing the reservoir model is received from the cube manager. Each processor receives only that part of the x-y grid it is responsible for. Thus, the data is automatically distributed over the number of processors in use. During each time step in the reservoir simulation, a new linear system is solved. For each new linear system a one time EXCHANGEx and a one time GS_GB are performed. The algorithm as shown in Figure 8 corresponds to the CG algorithm presented in Section 2. During each iteration one EXCHANGEp and one GS_GB are required.

5. RESULTS

Table I depicts the CPU execution times for a fixed-size problem for a 68 by 68 grid, and the CPU execution times for various scaled problems. A fixed-sized problem is the largest problem that can be solved using one processor. The problem size is held fixed while the number of processors is increased [5,8]. Scaled problems are problems where the amount of computation per node is held constant. The total amount of computation is allowed to scale up as the number of processors increase. As the number of processors is increased, the size of the problem must necessarily increase (scale up). For example, the largest size problem that two nodes can accommodate is a 96 x 96 grid. The largest size problem that 16 nodes can accommodate is a 264 x 264 grid, and the largest size problem that all 64 nodes can accommodate is a 500 x 500 grid. This last problem represents solving a 250,000 by 250,000 sparse banded linear system. Each of these

scaled problems for a given size cube are in turn considered as fixed sized problems where additional nodes are used. The way in which the EXCHANGEx and EXCHANGEp algorithms are implemented makes it necessary for each processor to be assigned at least two rows of the grid to work on. Therefore, in Table I results for 64 processors are not included for the two smallest grid sizes.

It is important to be able to measure the speedup for scaled problems. There are several ways that have been suggested to determine scaled speedup [5,8,9]. The model I used in order to estimate the CPU time on a single processor for scaled problems is similar to the model described in Reference 5. Given an N x N square grid, the total computation time using one processor, T_1, is given by

$$T_1 = ciN^2$$

where i is the total number of iterations over the grid and c is the computation time per grid point. The number of iterations is independent of the number of processors used for a given problem, and c is the calculated constant for the particular method used. The value for i is determined by running the problem using multiple processors, then applying the value to one processor. Thus, the values for c and i can easily be determined without actually running the simulation on one processor. Furthermore, communication time, which is usually very difficult to estimate, is not a factor when using one processor. I used the fixed sized CG problem to determine the value for c. This value was used on each of the scaled problems to determine the estimated CPU time for one processor. The estimated values are indicated in Table I by an asterisk.

Table II depicts the scaled speedup corresponding to the processing times given in Table I. The values of interest are along the main diagonal which correspond to the scaled problems using various numbers of processors. The speedups range from 1.96 using two processors to a speedup of slightly more than 57 using 64 processors.

The results from the scaled problems for the conjugate gradient method corresponding to the main diagonal in Table II are summarized in Table III along with the efficiency. Notice the CG method maintains excellent speedup and efficiency over the entire spectrum of problem sizes. For comparison purposes Table III also summarizes the results for the Successive Over-Relaxation (SOR) method performed on the same problems [20]. The SOR method does not require any global communication during the iterative process, thus the method is implemented using a pipeline technique. The ratio of computation time to communication time is an important measurement to consider in the SOR pipeline model. This ratio decreases significantly as the scaled problem size increases and has a direct effect on the efficiency of the algorithm. This ratio is a limiting factor in speedup for the SOR algorithm [20]. The CG algorithm was from three to ten times faster than SOR in CPU time over the same problem set. The speedup and efficiency of the CG algorithm are also significantly superior. Clearly, CG is a superior algorithm in this case for solving the large banded sparse linear system arising from this reservoir model.

6. SUMMARY AND CONCLUSIONS

The conjugate gradient algorithm for solving large, sparse, banded linear equations arising from a reservoir model was implemented on a distributed-memory multiprocessor system. Results from a fixed size problem and various scaled problems for CG are presented. The CG algorithm had nearly linear speedup in all of the problems tested. For example, speedups of 1.96, 7.89, and 30.49 were obtained for scaled problems using 2, 8, and 32 nodes, respectively. The largest problem that could be solved with present hardware was a reservoir model discretized in space using a 500 x 500 grid. This represents a sparse linear system of size 250,000 x 250,000. On this problem a speedup of 57 was obtained using 64 processors. In all cases the efficiency of the CG algorithm was 89% or better.

The ratio of computation time to communication time is an important measurement to consider. Using one processor on a 68 x 68 grid, the processor is responsible for 4624 components. Of course there is no communication requirement. Using two processors, each processor is responsible for 4608 components (48 times 96). The number of components that each processor is responsible for is held relatively constant as the problems are scaled. This is the general concept for scaled problems: to keep each processor working at its capacity. Thus, as processors are added, the problem size necessarily increases. In practice, however, as the problem size increases so does the amount of memory required for support data structures for the grid. In the case of

64 processors each processor, was responsible for about 3900 components. Furthermore, as the number of processors increase the communication time increases proportional to the dimension of the cube. A four dimensional cube, for example, will require four communication steps to perform the GS_GB algorithm (see Figure 6). A six dimensional cube will require six communication steps. Thus as the problem is scaled up, the amount of computation performed by each processor slightly decreases, while the cost of communication slightly increases. Therefore, the ratio of computation time to communication time decreases very slowly as the problem is scaled up. Our experimental results verifies this in the slight decline in efficiency for the CG algorithm shown in Table III.

A comparison was made between the CG and SOR algorithms over the same problem set. The ratio of computation time to communication time is a critical measurement to consider in the pipeline model implementation of SOR. This ratio decreases rapidly as the scaled problem size increases, and has a direct effect on the efficiency of the algorithm. This ratio is a limiting factor in speedup for the pipeline model [20]. Notice the rapid decline in performance of the SOR algorithm compared to CG for the same set of problems shown in Table III. The CG algorithm out performed the SOR algorithm in every test case, and was as much as an order of magnitude faster when using all 64 processors.

As a comparison, the scaled problems were also executed on an IBM 3090 single processor with 16M of virtual memory. The vector facility of the IBM 3090 was not used. The 500 x 500 and 384 x 384 grid problems were too large to run on the IBM 3090 with the existing memory. The largest problem that could be run was the 264 x 264 grid. Running the SOR algorithm, the IBM 3090 executed this problem using 1688 CPU seconds, compared to 2183 CPU seconds for the SOR algorithm using all 64 nodes of the NCUBE hypercube. The same 264 x 264 grid problem was executed on the IBM 3090 using CG and took 1264 CPU seconds. This is summarized in Table IV. The wall clock time was approximately twice the CPU time. This implies the algorithm used about 50% of the available resources of the IBM 3090 and all 16M of memory. The same problem was executed by the CG algorithm in 306 seconds using all 64 nodes of the NCUBE hypercube. The CG algorithm ran four times faster on the hypercube compared to the IBM 3090. Furthermore, the hypercube is able to handle significantly larger problems. The IBM 3090 costs about 25 times that of the hypercube.

The conjugate gradient method proved to be an excellent algorithm to implement on a distributed-memory multiprocessor system for solving large banded sparse linear systems of equation. Its performance is nearly linear in speedup as the problem size increases. Furthermore, the hypercube has been shown to be extremely cost effective in solving these kinds of problems.

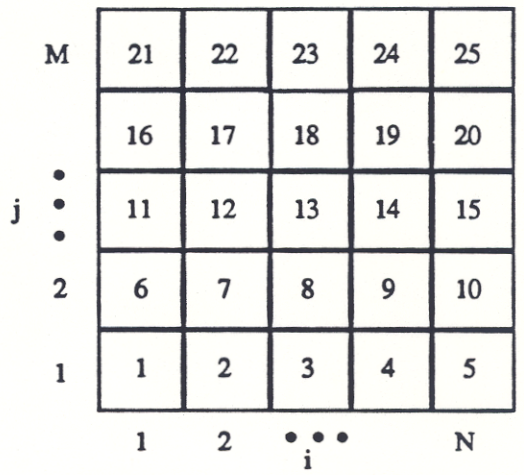

Fig. 1. Example x-y Grid Space for a Reservoir Simulation

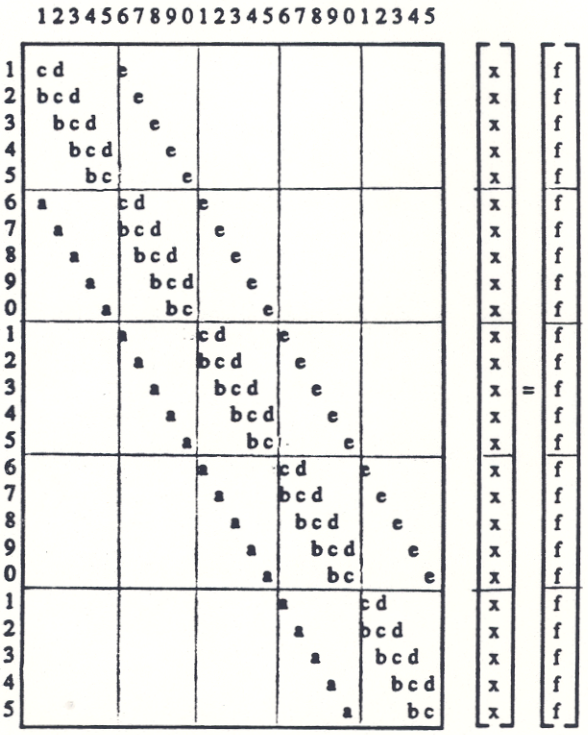

Fig. 2. Example Sparse Matrix Corresponding to Fig. 1.

EXTRA FIGURES 1 AND 2.

$$aP_{i,j-1} + bP_{i-1,j} + cP_{i,j} + dP_{i+1,j} + eP_{i,j+1} = f$$

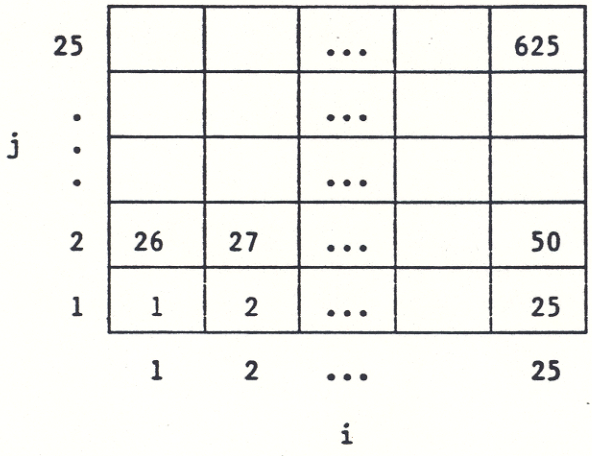

FIGURE 1. EXAMPLE x-y GRID SPACE FOR A RESERVOIR SIMULATION

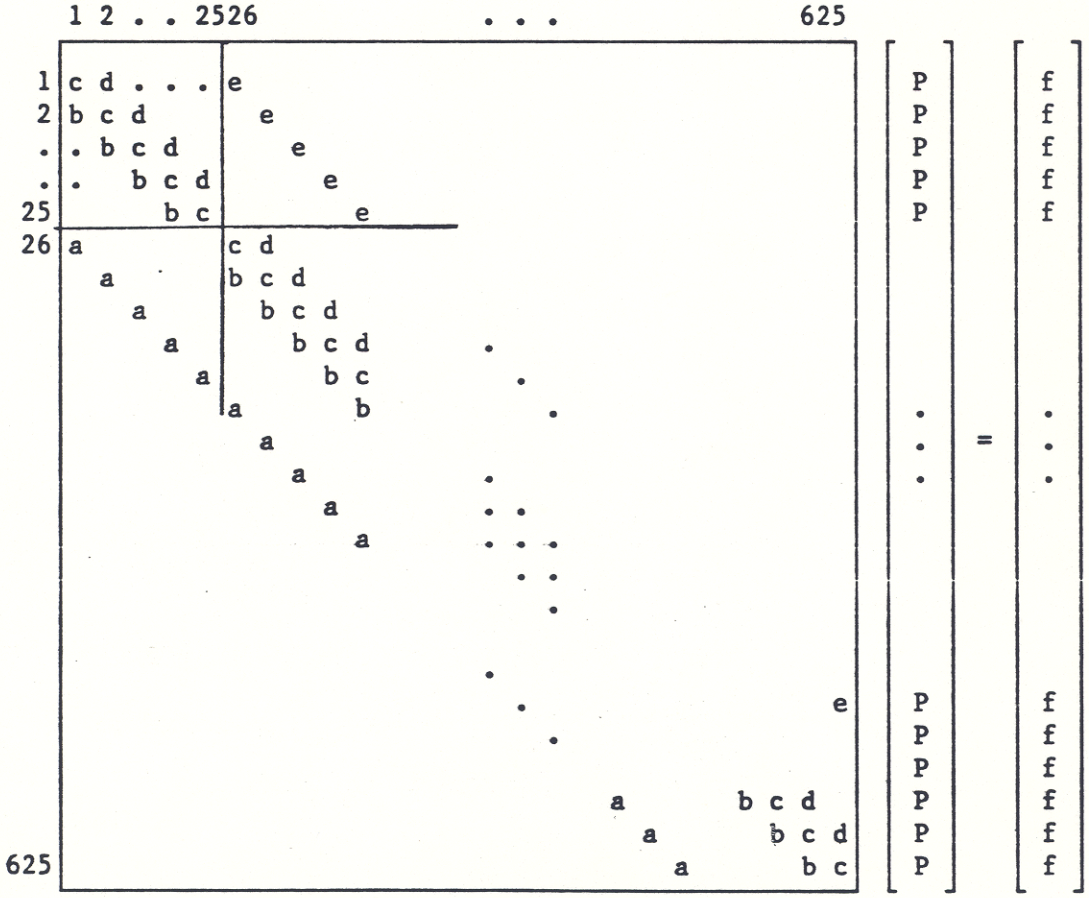

FIGURE 2. EXAMPLE SPARSE MATRIX CORRESPONDING TO FUGURE 1.

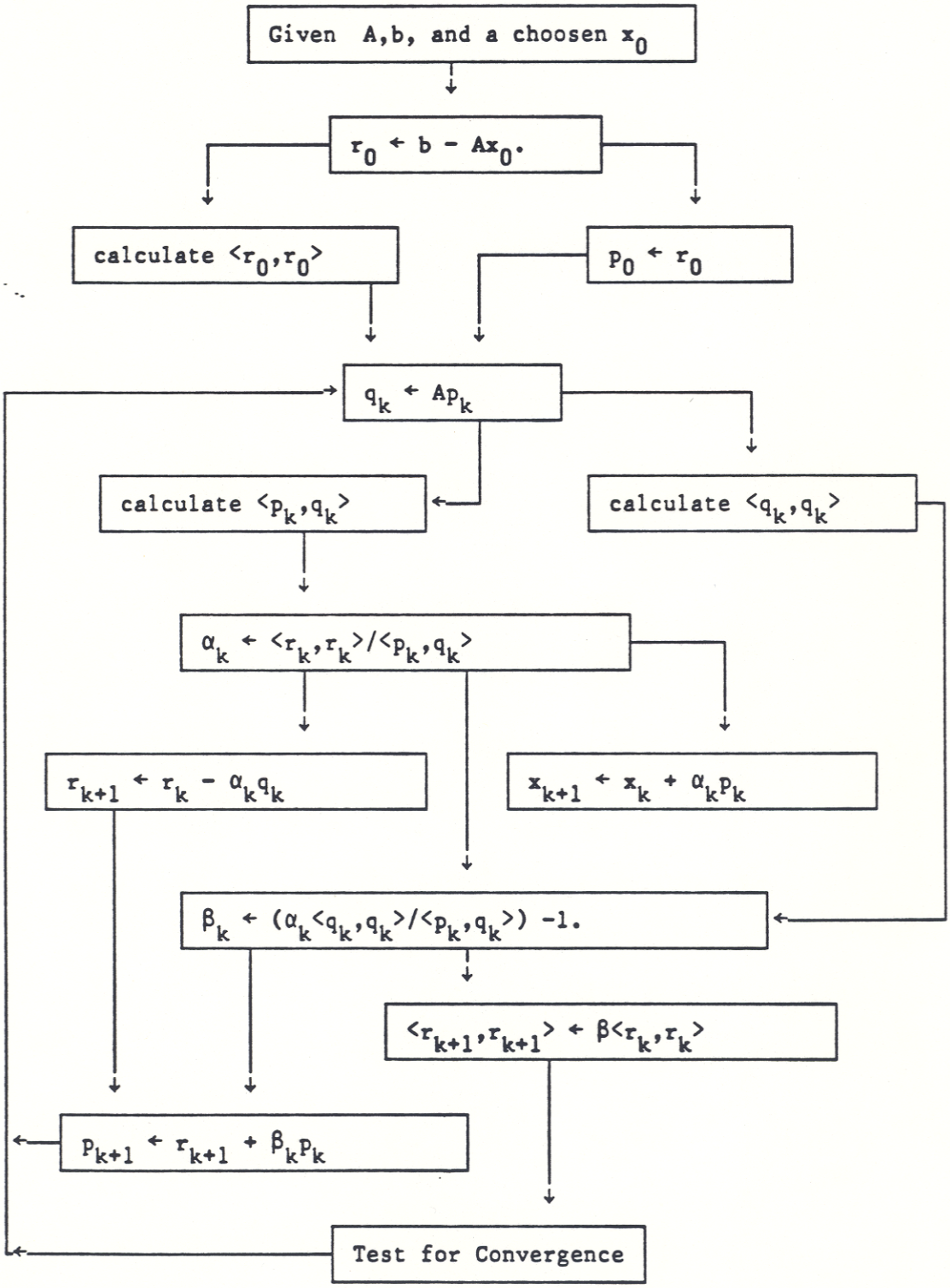

Fig. 3 Flow Diagram for the Conjugate Gradient Algorithm Solving Ax=b (The arrows carry along all the input as well as the results of the computation of the previous boxes)

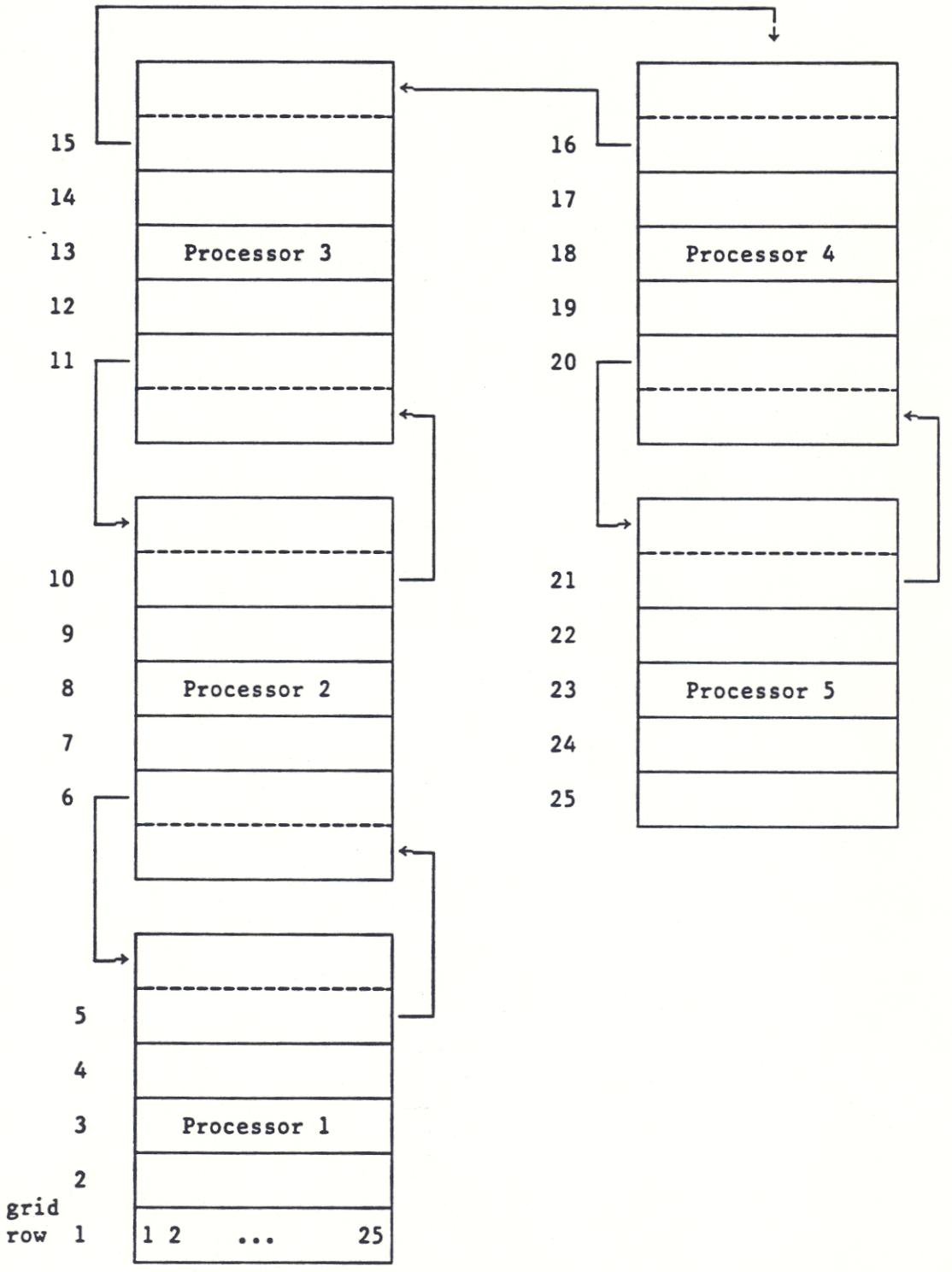

Fig. 4 EXCHANGE of nonlocal vector components with neighboring nodes in a ring topology (This figure corresponds to Figures 1 and 2).

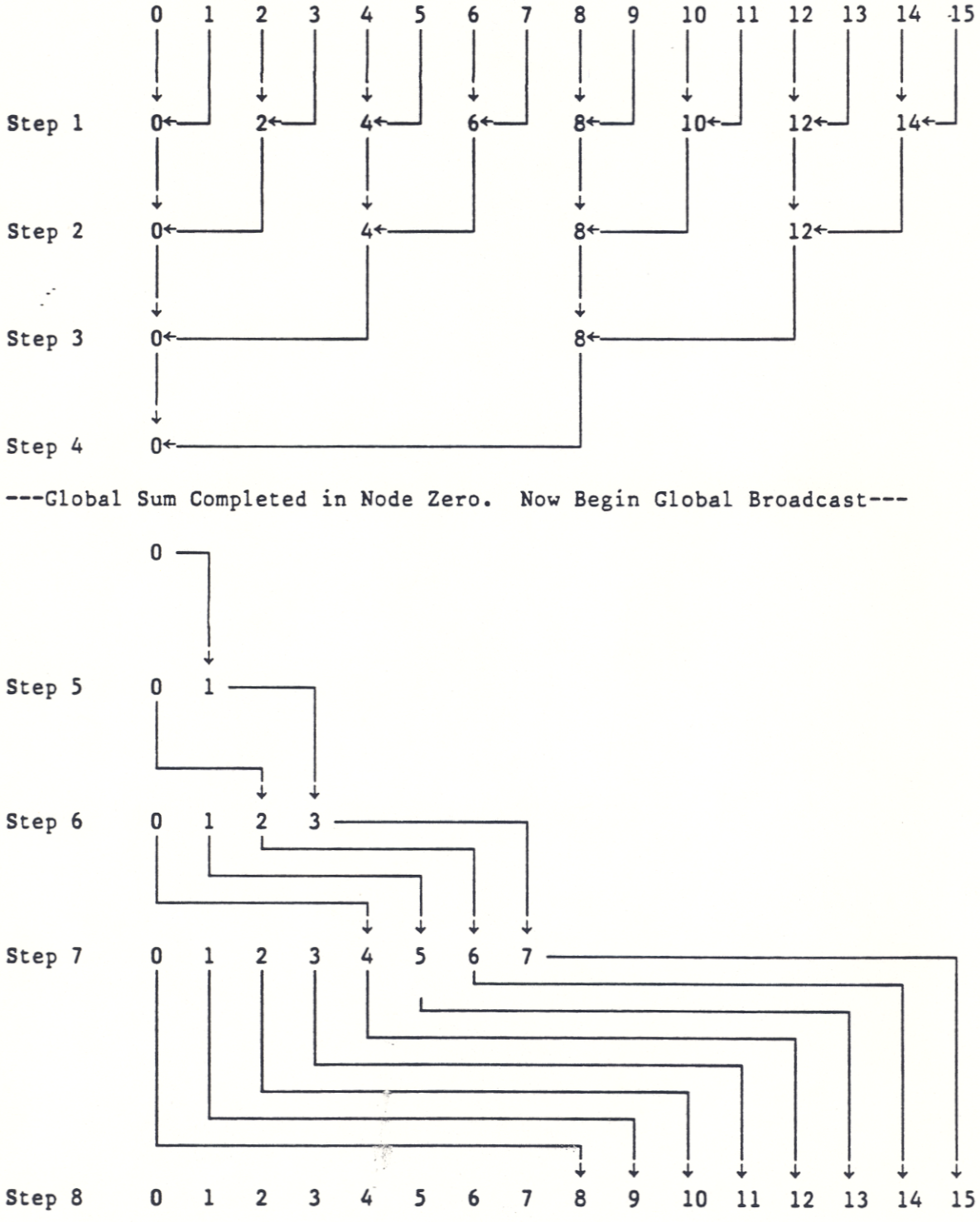

Fig. 5. Traditional Global Sum Global Broadcast (GS_GB) Algorithm Using a Tree Structure on a Hypercube of Dimension Four

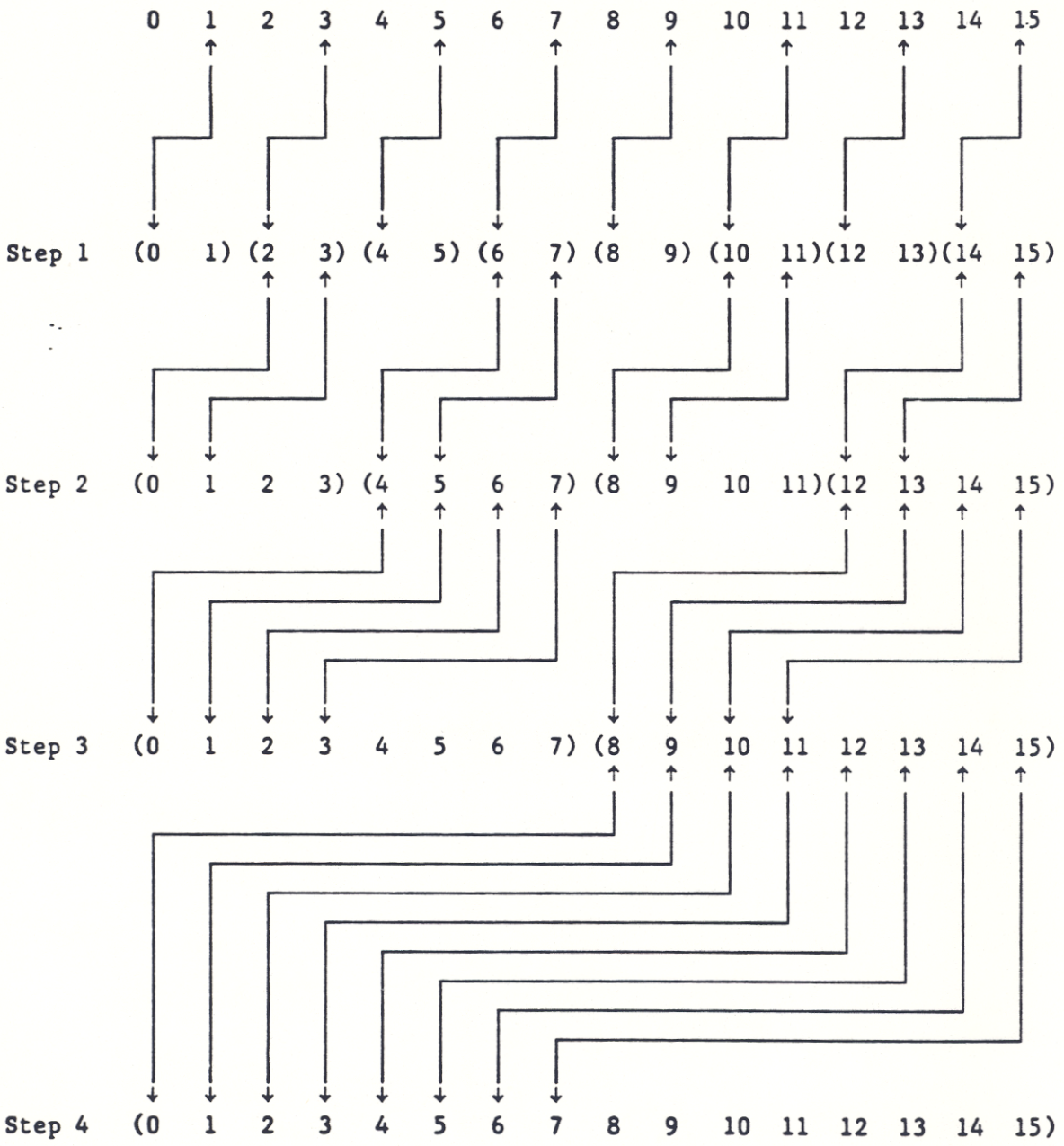

Fig. 6. Global Sum Global Broadcast (GS_GB) Algorithm Using a Sequence of Exchange-Adds on a Hypercube of Dimension Four. (Nodes shown in parenthesis have the same value after each step)

1. DETERMINE the number of processors to work on this problem, p.
2. INPUT data describing the reservoir model.
3. BROADCAST to each of the p processors the initial data describing the reservoir.
4. LOOP
   RECEIVE results of interest from a designated processor(s).
   PRINT (or save) results
   UNTIL termination message is received.

Fig. 7. Conjugate Gradient Algorithm (Cube Manager)

1. WHOAMI Processor determines which processor it is and its neighbors.
2. RECEIVE INPUT data (one time) from the cube manager describing the reservoir model. The processor determines which rows of the x-y grid (A and b) it is responsible for.
3. TIME LOOP (New Ax = b to solve)
   a. EXCHANGEx with both neighbors
   b. Calculate $r_0 = p_0 = b - Ax_0$.
      Calculate local $\langle r_0, r_0 \rangle$
   c. GS_GB of $\langle r_0, r_0 \rangle$

   For iterations k = 0,1,2,...
   1. EXCHANGEp with both neighbors
   2. form $q_k = Ap_k$
   3. local $\langle p_k, q_k \rangle$ and $\langle q_k, q_k \rangle$
   4. GS_GB of $\langle p_k, q_k \rangle$ and $\langle q_k, q_k \rangle$
   5. $\alpha_k = \langle r_k, r_k \rangle / \langle p_k, q_k \rangle$
   6. $\beta_k = (\alpha_k \langle q_k, q_k \rangle / \langle p_k, q_k \rangle) - 1.$
   7. $\langle r_{k+1}, r_{k+1} \rangle = \beta \langle r_k, r_k \rangle$
   8. $r_{k+1} = r_k - \alpha_k q_k$
   9. $x_{k+1} = x_k + \alpha_k p_k$
      send x_{k+1} to cube manager if designated node.
   10. $p_{k+1} = r_{k+1} + \beta_k p_k$.

   UNTIL CONVERGENCE

UNTIL end of time

Fig. 8. Conjugate Gradient Algorithm (Each Processor 1..p)

Table I.

CPU Results for Fixed and Scaled Problems
Using CG
All times are in Seconds

	GRID SIZE OF THE PROBLEM						
Nodes Used	68x68	96x96	132x132	192x192	264x264	384x384	500x500
1	512.3	981.1*	2627.6*	6357.8*	14205.8*	28199.5*	37340.3*
2	262.5	500.9	—	—	—	—	—
4	135.4	255.5	662.7	—	—	—	—
8	75.8	133.0	350.2	805.5	—	—	—
16	46.1	72.0	193.9	416.7	937.5	—	—
32	31.6	41.7	116.3	222.7	515.5	925.0	—
64	—	—	77.9	121.1	306.4	490.1	654.2

* Estimated CPU Times using $T_1 = ciN^2$

Table II.

Speedup for Fixed and Scaled Problems Using CG
(Corresponding to the CPU Times in Table I)

	GRID SIZE OF THE PROBLEM						
Nodes Used	68x68	96x96	132x132	192x192	264x264	384x384	500x500
1	1.00	—	—	—	—	—	—
2	1.93	1.96	—	—	—	—	—
4	3.78	3.84	3.97	—	—	—	—
8	6.76	7.38	7.50	7.89	—	—	—
16	11.10	13.64	13.55	15.26	15.15	—	—
32	16.19	23.55	22.60	28.55	27.56	30.49	—
64	—	—	33.74	52.50	46.42	57.40	57.08

Table III.

Performance Summary for Conjugate Gradient
Versus Successive Over-Relaxation

Nodes Used	Size Grid	Conjugate Gradient (CG)			Successive Over-Relaxation (SOR)		
		CPU Time	Scaled Speedup	Efficiency	CPU Time	Scaled Speedup	Efficiency
2	96x 96	500.9	1.96	97.93%	1865.3	1.81	90.31%
4	132x 132	662.7	3.97	99.13	2462.7	3.49	87.20
8	192x 192	805.5	7.89	98.66	3274.3	6.59	82.38
16	264x 264	937.5	15.15	94.70	4110.3	12.06	75.37
32	384x 384	925.0	30.49	95.27	5355.7	19.47	60.83
64	528x 528	654.2	57.08	89.19	6789.8	25.22	39.41

Table IV.

Performance Summary for Conjugate Gradient
and Successive Over-Relaxation on the
Hypercube and IBM 3090 for a 264 x 264 Grid
(CPU Times are Recorded in Seconds)

	Conjugate Gradient CPU Time	Successive Over-Relaxation CPU Time	Cost
IBM 3090	1264	1688	$ 6M
NCUBE/10 (64 Nodes)	306	2183	$ 250K

ACKNOWLEDGEMENTS

The work reported in this paper was supported by Amoco Production Company Research Center, Tulsa, Oklahoma.

REFERENCES

1. Agrawal, D., Jamakiram, V., and Pathak, G., "Evaluating the Performance of Multicomputer Configurations", Computer, Vol. 19, No. 5, May, 1986, pp. 23-37.

2. Aykanat, C. and Ozguner, F., "Large Grain Parallel Conjugate Gradient Algorithms on a Hypercube Multiprocessor", Proceedings of the 1987 International Conference on Parallel Processing, pp. 641-644, August, 1987.

3. Aykanat, C., Ozguner, F., Ercal, F. and Sadayappan, P., "Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes", IEEE Transactions on Computers, vol. 37, no. 12, Dec., 1988.

4. Aziz, K. and Settari, A., Petroleum Reservoir-Simulation, Elsevier Applied Sciences Publishers, London, 1980.

5. Denning, P. J., "The Science of Computing - Speeding up Parallel Processing", American Scientist, vol. 76, pp. 347-349, July-August, 1988.

6. Evans, D. J., "Parallel SOR Iterative Methods," Parallel Computing, vol. 1, pp. 3-18, 1984.

7. Ewing, R. E., editor, "The Mathematics of Reservoir Simulation" SIAM, 1983.

8. Gustafson, J. L., Monty, G. R., Brenner, R. E., "Development of Parallel Methods for a 1024-Processor Hypercube", SIAM Journal on Scientific and Statistical Computing, vol 9, no. 4, July, 1988.

9. Gustafson, J. L., "Reevaluating Amdahl's Law", Commun. ACM, vol. 31, no. 5, pp. 532-533, May, 1988.

10. Lakshivarahan, S. and Dhall, S. K., "A New Hierarchy of Hypercube Interconnection Scheme for Parallel Computers: Theory and Practice" Research report OU-PPI-TR-86-02, Parallel Processing Institute University of Oklahoma, Norman, Oklahoma, August 1986.

11. NCUBE Corporation, "NCUBE System Manual", 1985.

12. Ni, Lionel, M.; King, Chung-Ta, and Prins, Phillip, "Parallel Algorithm Design Considerations for Hypercube Multiprocessors," Proceedings of the 1987 International Conference on Parallel Processing, Pennsylvania State Univ. Press, Aug. 1987, pp. 717-720.

13. Patel, N. R. and Jordan, H. F., "A Parallelized Point Successive Over-Relaxation Method on a Multiprocessor," Parallel Computing, pp. 207-222, 1984.

14. Peaceman, W. D., "Survey Problems in Numerical Reservoir Simulation", in Mathematical and Computational Methods in Seismic Exploration and Reservoir Modeling", edited by W. E. Fitzgibbon, SIAM, 1986.

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

16. Scott, S. L., Wainwright, R. L., Raghavan, R. and Demuth, H. "Applications of Parallel (MIMD) Computers to Reservoir Simulation", Proceedings of the Ninth SPE Symposium on Reservoir Simulation, San Antonio, Texas, February, 1987.

17. Seitz, C. L., "The Cosmic Cube", Commun. ACM, vol. 28, no. 1, pp. 22-33, 1985.

18. Wainwright, R. L., "Deriving Parallel Computations from Functional Specifications: A Seismic Example on a Hypercube", International Journal of Parallel Programming, vol. 16, no. 3, June, 1987, (appeared May, 1988).

19. Wainwright, R. L., "Message Passing Considerations For Hypercube Multiprocessors", Proceedings of the Second Workshop on Applied Computing, The University of Tulsa, Tulsa, OK, March, 1988.

20. Wainwright, R. L., "A Software Kernel for a Pipeline Model for Solving Sparse Matrix Problems on a Hypercube Multiprocessor Applied to Reservoir Simulation, Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications, Monterey, CA, March 6-8, 1989.