Proceedings of the 1990 ACM/IEEE Symposium on Applied Computing
April, 5-6, 1990, pp. 85-91.

A Functional Program Describing a Simple Reservoir Model and Its Potential for Parallel Computation

Rex L. Page[1]
Marian E. Sexton[1]
Roger L. Wainwright[1,2]

[1]Amoco Production Company, Research Center
[2]Department of Mathematical and Computer Sciences
University of Tulsa

ABSTRACT

This paper presents results using a functional programming language, Miranda, to solve a simple reservoir modeling problem. The algorithm uses Miranda's functional form to determine a parallel decomposition of a reservoir modeling problem. There is discussion on discerning the parallel decomposition as well as the ease of specifying the problem in functional form. Finite element discretization of a reservoir model yields linear equations of the form $Ax=b$, where A is a large, sparse, banded matrix, and x and b are dense vectors. Each step of the simulation uses the Conjugate Gradient method to solve the sparse linear system. Matrices are represented as quads in Miranda to take advantage of their sparsity. Vectors are represented as lists of numbers. Other data structures yielded worse performance. The paper presents results of simulations for reservoirs which yield sparse matrices up to size 4096 x 4096 and estimates for matrices up to size 262,144 x 262,144.

Keywords:

Functional Programming, Miranda, Parallel Computation, Sparse Linear Systems, Reservoir Modeling

BRIEF OVERVIEW OF THE RESERVOIR MODEL

The test problems used in this paper arise from a parabolic differential equation which is the two-dimensional, single-phase diffusion equation from petroleum reservoir simulation. The partial differential equation is approximated in space using a discrete, irregular, block-centered grid. This approximation leads to a five-point difference equation using central differences to approximate the space derivatives and a backward difference to approximate the time derivative. A similar approximation leads to a nine-point discretization. The vast majority of reservoir simulations are two-phase, non-linear non-symmetric models. This paper considers the more basic single phase, linear problem. The study investigated square reservoirs of size $R = 2^K$, which yielded, in each case, a five-banded, sparse, symmetric, positive definite linear system to solve of size $N = R^2$, for K = 1,2,.... The particular reservoir model used in this paper involves 38 time steps. Thus, during the simulation, it was necessary to solve 38 different linear systems using, in particular, the Conjugate Gradient method to determine the solution. Wainwright [7] provides additional information on this particular reservoir model. A more detailed description of the Conjugate Gradient method as applied to parallel processing is given by Aykanat [1], and by Wainwright [8].

BRIEF OVERVIEW OF THE CONJUGATE GRADIENT METHOD

The simple reservoir simulation model used the Conjugate Gradient method, as described in Reference 1. 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 when using multiple 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, as implemented, follows:

Step 0. Initially choose x_0 and let $r_0 = p_0 = b - Ax$.
Then compute $< r_0, r_0 >$. Then,

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

1. form $q_k = Ap_k$
2. form $< p_k, q_k >$ and $< q_k, q_k >$
3. (a) $\alpha_k = < r_k, r_k > / < p_k, q_k >$

   (b) $\beta_k = (\alpha_k < q_k, q_k > / < p_k, q_k >) - 1$

   (c) $< r_{k+1}, r_{k+1} > = \beta_k < r_k, r_k >$
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. The definition of r_k is $b - Ax_k$ and it must be null when x_k is the exact solution. Thus, a suitable criterion for halting the iterations is when $[<r_k,r_k>/<b,b>]^{1/2} < \epsilon$, where ϵ is very small. In all cases, a tolerance of $\epsilon = 10^{-7}$ using 16 digit (double precision) arithmetic was used. p_k represents the direction vector of the gradient at the kth iteration, and $< >$ denotes the inner product of two vectors.

MIRANDA IMPLEMENTATION OF THE RESERVOIR MODEL

Throughout this paper, the authors assume the reader is familiar with both the functional style of programming similar to that detailed in Bird and Wadler's functional programming text [2], and, in particular, the functional programming language, Miranda, written by David Turner [5,6].

Data Structures Used in this Implementation

Each iteration of the Conjugate Gradient method requires the following operations. Note in this algorithm there are no matrix–matrix operations.

3 vector–vector dot products
2 vector–vector additions
1 vector–vector subtraction
3 scalar–vector multiplications
2 matrix–vector multiplications
2 scalar divisions
2 scalar multiplications
1 scalar subtraction

The sparse matrices arising from the difference equations in the problem range in size from approximately 100 x 100 to 100,000 x 100,000. The quad data structure emits an economical representation of these matrices. (Burton and Kollias and also Wise and Franco [3,9,10] elaborate on the quad data structure for sparse matrices.) The vector abstract data type shown below (in Miranda) is a tuple where the first value indicates the vector length, and the second represents an implementation of a vector as a list of numbers. The abstract data type for a matrix is also a tuple where the first value is the total number of entries in the matrix, and the second represents the matrix as a quad. For this problem, representing vectors as lists of numbers proved to be more effective than a quad–like representation.

```
vector == (num, [num])
matrix == (num, quad)
quad ::= Quad quad quad quad quad | Diag num
```

The size of a matrix, represented in quad form, must be 2^d by 2^d for d = 0,1,2... If a matrix has a constant value along the main diagonal and all off–diagonal values equal to zero, then a single value represents the matrix, that of the main diagonal. A value of zero, in this case, represents a zero matrix. If the matrix is not of this form, then the data structure divides the matrix into four subtrees of equal size, $2^{(d-1)}$ by $2^{(d-1)}$. The order of the quadrants is, left to right, northwest, northeast, southwest and southeast. In this way, each matrix has a unique representation. The quad representation exhibits a savings in memory since the zero values can be compressed into Diag 0's. Furthermore, it represents a savings in time because operations which use Diag num are extremely efficient. The size of the matrix that Diag num represents depends on the context in which it appears. An example quad expansion is given below. Figure 1 depicts an 8x8 banded matrix along with the corresponding unique quad representation.

Examples:

```
Diag 3  represents 3 0 0 0   and
                   0 3 0 0
                   0 0 3 0
                   0 0 0 3

Quad (Diag 3                                          )
     (Quad (Diag 0) (Diag 1) (Diag 0) (Diag 0))
     (Quad (Diag 0) (Diag 0) (Diag 2) (Diag 0))
     (Diag 3                                          )

     represents 3 0 0 1
                0 3 0 0
                0 0 3 0
                2 0 0 3
```

The study considered and tested other data structures to represent matrices. For example:

```
quad ::= Quad quad quad quad quad |
         Scalar num | Zero
```

In this data structure, Scalar is always a single digit, and Zero represents a block of any size of all zeros. Using this data structure, with the addition of a third construct, the CPU performance degraded 25 to 30%. This degradation in performance is probably attributable to the extra case check necessary when using both Scalar num and Zero instead of the single check required when using Diag num, the combination of the two. Furthermore, the particular reservoir model simulated yields diagonal bands of constant values which favors the Diag num version.

This investigation considered several different data structures for vectors other than lists of numbers. Wise [9,10] suggests the use of a sparse matrix to represent a vector. This sparse matrix contains all zero values on the off diagonals with the values of the vector itself placed on the main diagonal. Wise's method then converts the matrix representation of the vector to a quad form. In this way, Wise proposes the quadtree as a uniform data structure to represent every object: scalar, vector, and matrix. However, Wise was primarily dealing with problems which contain matrix–matrix and matrix–vector operations and for these operations this data structure works quite well. In the current simulation, which has virtually all vector–vector and scalar–vector operations, only two matrix–vector operations and no matrix–matrix operations, Wise's data structure is not practical.

Further, the study considered two different binary tree structures as the data structure for representing vectors (as an alternative to lists of numbers). A bintree, in the following Miranda definition, represents a vector with a binary tree in which all of the valid numerical data are in the leaves of the tree.

```
bintree ::= Bintree bintree bintree | Scalar num
```

The second binary tree representation of bintree contains valid numerical data in each node of the binary tree (both leaf and internal nodes alike). The Single construct in the following denotation represents a binary tree with only one child.

```
bintree ::= Bintree num bintree bintree |
            Scalar num | Single num bintree
```

The CPU times for the reservoir model simulation were about the same using either of the bintree representations for vectors. The first bintree definition requires about twice as much storage as the second definition, however, it requires less checking during processing. Thus, it is not surprising that the CPU times

were about the same. However, both of these binary tree representations are three to four times slower than the lists of numbers representation. Since most of the vector operations in this simulation are vector–vector or scalar–vector operations this is not unexpected. The binary tree definitions use more storage and require more execution time dividing and processing vectors than do simple lists of numbers. If most of the operations were matrix–vector operations, however, then perhaps one of the bintree representations would demonstrate better performance than the lists of numbers. This analysis assumes a sequential processor. At this time, Miranda is in use only on sequential processor machines. With a multiprocessor system, the binary tree representation may prove to be superior because of the divide and conquer strategy which is conducive to parallel processing.

The Conjugate Gradient Method in Miranda

The Miranda function CONJ_GRAD solves the linear system Ax = b using the Conjugate Gradient method. At the time the function CONJ_GRAD is called, the matrix, A, is already in quad form. The last parameter to CONJ_GRAD, CNT, gives the number of iterations required for convergence of the linear system. The Miranda implementation of the Conjugate Gradient method is given below. Notice how the Miranda implementation of the Conjugate Gradient method closely resembles the mathematical specification shown previously, thus demonstrating one of the primary advantages of specifying a problem in functional form, that of its simplicity.

```
>conj_grad :: matrix->iter_solution->
>             iter_solution

>conj_grad a (Iter_sol x r p cnt)
>  = until converge
>          nextiter (Iter_sol x r p cnt)
>    where
>    converge (Iter_sol x r p cnt)
>      = rr < eps
>        where
>        rr = vvdot r r
>        eps = .000001
>    nextiter(Iter_sol x r p cnt)
>      = (Iter_sol x' r' p' cnt1)
>        where
>        cnt1 = cnt + 1
>        rr = vvdot r r
>        x' = vvadd x (svmult alpha p)
>        r' = vvsub r (svmult alpha q)
>        p' = vvadd r' (svmult beta p)
>        alpha = rr / pq
>        beta = ((alpha * qq) / pq) - 1
>        pq = vvdot p q
>        qq = vvdot q q
>        q = mvmult a p
```

The Reservoir Simulation Driver Functions

To execute the reservoir simulation, the user assigns the desired value for LIN_SYS_SIZE and then types "restest". The function FORMAT displays the results of the simulation. The function SOLUTION strips off and keeps only the RESULT of RES. RES contains the accumulated results of the simulation after applying the function LIN_SYS_SOLV repeatedly through a sequence of time steps determined by TIME_SEQ_LIST. In the simulation, time begins at zero and continues through four different changes in time steps indicated by the variable TIME_SEQ_LIST until time = 10. Time unit measurement is in days. TIME_SEQ_LIST specifies that measurement of the reservoir begins at time 0 and increments by .001 of a day, the delta time step, until time = .009, time then increments by .01, the new delta time step, until time = .09, and so forth until the delta time step is 1 unit and time = 10.

```
>restest :: [char]
>restest =  header ++ "\n" ++
>           format (reverse
>                   (solution (res)))

>solution :: (vector,num,[res_tuple],num)
>            -> [res_tuple]
>solution (soln,cnt,result,ctime) = result

>res :: (vector,num,[res_tuple],num)
>res = foldl lin_sys_solv
>            (solnx, 0, [], 0)
>            time_seq_list

>time_seq_list :: [steps]
>time_seq_list = [Step_seq .001 .0099,
>                 Step_seq .01 .0999,
>                 Step_seq .1 .999,
>                 Step_seq 1 9.999]
```

Linear System Solver (LIN_SYS_SOLV)

The main diagonal of the matrix changes whenever the size of the time step changes, i.e., when the delta time step changes from .001 to .01, again when it changes from .01 to .1, and so forth.

The function LIN_SYS_SOLV computes the new matrix.

```
>lin_sys_solv
>  :: (vector,num,[res_tuple],num)->
>   steps ->
>   (vector,num,[res_tuple],num)

>lin_sys_solv (solnx,cnt,result,ctime)
>             (Step_seq dt tlimit)
>  = pickpart
>      (timesteps ((Prob a b),
>                  (Iter_sol x r p cnt),
>                  solnx,result,ctime,
>                  (Step_seq dt tlimit)))
>    where
>    a = mmadd (upper_band 0 main_diag)
>              res5band
>    main_diag
>      = map2 (+) const_alist
>                 (rep lin_sys_size
>                      (vvalue/dt))
>    x = solnx
>    r = vvsub b (mvmult a x)
>    p = r
>    b = apply f newb (lin_sys_size div 2)
>        where
>        f x = x - fvf * q
>        newb = svmult (vvalue/dt) solnx

>pickpart :: (problem,iter_solution,
>             vector,
>             [res_tuple],num,steps)->
>            (vector,num,[res_tuple],num)

>pickpart(lin_sys,(Iter_sol x r p cnt),
>         solnx,result,ctime,time_seq)
>  = (solnx,cnt,result,ctime)
>lin_sys_size :: num
>lin_sys_size = 1024
```

Timesteps

Each time the simulation increases in time but the delta time step is constant, the algorithm solves a new linear system. However, while the delta time step is constant, the matrix as generated by LIN_SYS_SOLV remains constant. The only change to each equation in the linear system is on the right hand side. TIMESTEPS generates the new b vector (for Ax = b) each time. Within the function, TIMESTEPS, is the call to the function CONJ_GRAD, the linear system solver using the Conjugate Gradient method described in the previous section.

```
> timesteps :: (problem,iter_solution,
>               vector,[res_tuple],
>               num,steps) - >
>              (problem,iter_solution,
>               vector,[res_tuple],
>               num,steps)

 > timesteps (lin_sys,sol_pieces,solnx,
>              result,ctime,time_seq)
>  = until timedone
>          timeunit (lin_sys,sol_pieces,
>                    solnx,result,ctime,
>                    time_seq)
>    where
>    timedone(lin_sys,sol_pieces,solnx,
>             result,ctime,
>             (Step_seq dt tlimit))
>      = ctime > = tlimit
>    timeunit ((Prob a b),
>             (Iter_sol x r p cnt),
>             solnx,result,ctime,
>             (Step_seq dt tlimit))
>      = ((Prob a b'),
>        (Iter_sol x' r' p' cnt),
>        solnx',result',ctime',
>        (Step_seq dt tlimit))
>        where
>        result'
>          = Result_tuple ctime count
>                         (middlex vect ) :
>                         result
>            where
>            (count, vect) = newx
>            middlex vect
>              = (subscript
>                   vect
>                   (lin_sys_size div 2)) -
>                dpp
>        newx
>          = pressure
>              (conj_grad
>                 a
>                 (Iter_sol x r p cnt))
>        solnx' = vect
>                 where
>                 (count, vect) = newx
>        pressure (Iter_sol x r p cnt)
>          = (cnt, x)
>        ctime' = ctime + dt
>        x' = solnx'
>        r' = vvsub b' (mvmult a x')
>        p' = r'
>        b' = apply f newb
>                   (lin_sys_size div 2)
>             where
>                   f x = x - fvf * q
>                   newb = svmult (vvalue/dt)
>                                 vect
>                          where
>                          (count, vect) = newx
```

RESULTS

The current research investigated reservoirs of size $R = 2^K$, yielding matrices of size $N = R^2$, for K = 1..6. All simulations were run on a Sun4 workstation using release 2.009 (November 14, 1989) of Miranda. Table I shows the test results for these problems, giving the CPU time in seconds, and the total number of Conjugate Gradient iterations required to solve all of the systems of equations. In addition, Table I reports the number of function reductions, claims, and garbage collections that Miranda performed for each reservoir problem. In all cases reported in Table I, the heap size used in Miranda was 5 million cells. This is the maximum heap size that will fit in main memory. Each cell in Miranda is 9 bytes of memory. The maximum allowable heap size for Miranda in our present system is 20 million cells (180 megabytes).

The reservoir for K = 7 was possible to model by utilizing the maximum heap size of 20 million cells. This did not run to completion due to the long execution time. However, partial execution of this model revealed only 16% CPU utilization. The low utilization was due to the large amount of time spent paging in and out of virtual memory. Reservoirs for K = 8 and larger were unable to complete due to insufficient heap space. Table II gives estimates of the CPU times required to model reservoirs for K = 7 through K = 9. The ratio of CPU times for successive K values in Table I is approximately a factor of 6. Based on this, Table II assumes a crude estimate of a factor of 6 in CPU times for each additional K. Furthermore, because of 16% CPU utilization for these large problems, the table employs an additional factor of 6 to estimate wall clock time from CPU time.

The limit on the size of the reservoir models that the current Miranda system can simulate is relatively modest, and the length of the execution times is rather large. This is even more evident in Table II. The current reservoir model is relatively simple. An "industrial size" reservoir simulation involving more complexity will increase the CPU time by a factor of 10. Furthermore, the model needs to be at least 512 x 512 grid blocks. Also, the current Miranda system is interpreted and not compiled. A compiled version of Miranda is expected within the next 12 months, and is conservatively estimated to be 30 times faster than the current interpreted system. Thus, using K = 9 in Table II as a basis, a large realistic model using a compiled version of Miranda will take approximately 30 days CPU time, and 180 days wall clock time using the current sub-megaflop Sun hardware. However, parallel processors running at 10 gigaflops are now commonplace. This means that a realistic model based on this approach with a modern parallel computing system could easily perform the computation for a realistic reservoir model in less than 5 minutes.

The following is an estimate for the memory requirements of a K = 9 reservoir model. The largest model that could run in the present memory system was K = 7, which represents a matrix of size N = 16,384. The matrix has five bands (each approximately size N) plus an additional four utility vectors used in the linear system solver (each, also, approximately size N). Assume further that this model used all of the available 180 Megabyes of memory. On this basis, the memory requirements for K = 9 (N is 262,144) would be approximately 2.9 gigabytes. Current parallel computers support ten to a hundred times more memory than this requirement.

PARALLEL SPECIFICATION OF THE RESERVOIR SIMULATION ALGORITHM

This reservoir simulation deals primarily with a sequence of linear systems to solve. Thus, the study focused its emphasis on the parallelization of the linear system solver, CONJ_GRAD. Each iteration of the Conjugate Gradient method requires various vector–vector, scalar–vector, and matrix–vector operations.

One of the major advantages of functional programs is their potential for parallelism. Furthermore, the evaluation of an expression cannot have side effects in a functional language. Thus, the evaluation of subexpressions can be in any order or it can be in parallel. In this application, quadtrees represent matrices, as described earlier. Vectors, in this application, are represented in Miranda as lists of numbers. Scalar–vector and vector–vector operations are straight forward and relatively inexpensive. The most time consuming portion of the Conjugate Gradient linear system solver is the matrix–vector multiply operation. During matrix–vector multiplication, the program decomposes the vector (list) into two equal parts corresponding to the natural matrix decomposition into quads. As the further decomposition of the matrix takes place, the program also performs the corresponding decomposition of the subvector. Each partition is a separate and independent process. Review the Miranda MVMULT function shown below which illustrates automatic decomposition depending on the sparsity of the matrix.

MVMULT. MATRIX VECTOR MULTIPLY PRODUCING A VECTOR (The matrix must already be in quad form)

```
> mvmult (sizem, m) (lenv, v)
>     = makevect (mvmult' m v),
>                          sizem = lenv
>     = error
>       "vector & matrix of different sizes",
>                          otherwise

> mvmult' (Diag 0) vect = rep (# vect) 0
> mvmult' (Diag a) [v] = [a*v]
> mvmult' (Quad nw ne sw se) vect = [a*y | y ← vect]
> mvmult' (Quad nw ne sw se) v
>     = v, v = [0| y ← v]
>     = firsthalf ++ secondhalf, otherwise
>       where
>       firsthalf
>         = map2 (+) (mvmult' nw v1)
>                    (mvmult' ne v2)
>       secondhalf
>         = map2 (+) (mvmult' sw v1)
>                    (mvmult' se v2)
>       v1 = take ((#v) div 2) v
>       v2 = drop ((#v) div 2) v

> makevect v = (#v, v)
```

The list representation of vectors and subsequent decomposition into two equal parts is logically identical to a binary tree representation. The study found no benefit, however, in representing vectors as a matrix type or as a binary tree type in Miranda. As discussed earlier, a higher CPU time resulted from each of these cases compared to the list representation of vectors.

The property of quadtrees is particularly valuable for matrices with regular patterns of non–zero entries, especially for sparse or banded matrices. Even if a banded matrix has pockets of non–zero entries elsewhere in the matrix, the quadtree representation will decompose automatically and efficiently, taking advantage of any sparse area in the matrix, wherever it may be. Matrices of this form are common place in reservoir modeling.

The quadtree decomposition of a dense matrix results in a distribution of the matrix into a block matrix format. Elster and Reeves [4] studied several matrix–matrix and matrix–vector operations for matrices decomposed in block format. In this block decomposition, the processors are arranged into a two dimensional grid, and the blocks of the matrix are distributed as evenly as possible across the processors. Elster and Reeves also studied these operations on matrices decomposed using the split–by–column format. The split–by–column format distributes columns of the matrix evenly over the available processors. They considered a two–dimensional grid topology as well as a two dimensional nearest neighbor grid topology for a hypercube for implementing both the block format and the split–by–column format. Their results for matrix–vector multiplication show that the block matrix format was superior to the split–by–column format in computation, as well as communication, for the given topologies. This makes the quadtree representation (a block format) a cost effective representation for matrices.

The decomposition of these basic operations for the Conjugate Gradient method is not the same as the implementation given by Aykanat and others [1]. Aykanat specifically targeted his implementation for a hypercube multiprocessor system. The decomposition presented here (using quads), did not take into consideration the nature of the matrix. Since the matrices are sparse, the quad representation which collapses blocks of zeroes into a single value will be very efficient. Moreover, the quad representation works on dense matrices and, in an adapted form, on non–square matrices, as well. Furthermore, the quad representation does not pertain to any specific target hardware; rather, it represents a general decomposition of the problem for refinement later on a target hardware. Aykanat, on the other hand, considered the nature of the matrix in his decomposition of the problem as well as the target hardware. Aykanat knew that he was dealing specifically with nine banded matrices. Knowing that, he stored the bands of the matrix into nine vectors with each band distributed evenly across all of the processors. Since the bands are not all consecutive in the matrix (i.e., these matrices have bands of zeroes intermixed with non–zero bands) it was necessary to establish the processors of the hypercube in a ring topology and pass part of the vectors of information between neighboring processors in the ring. Wainwright [8] established a similar process for a five banded system using the same reservoir model implemented in this paper. Adapting the quad representation to a target hardware is easily done. The authors maintain that the parallel decomposition of the problem, independent of target hardware, is more readily visible when specified in a functional form.

SUMMARY, CONCLUSIONS AND FURTHER RESEARCH DIRECTIONS

The work reported in this paper represents the first part of a two part research project. The ultimate goal of the overall project is to use a functional language to solve a large "industrial size" project; i.e., reservoir modeling, in this case. The authors contend that functional programming has enormous benefits not only in the ease of specifying a problem, but also in the potential for parallel decomposition of the problem. To date, the authors are not aware of any large scale problem being implemented using a functional language, such as Miranda.

The first part of the project reported here implemented and tested a simple reservoir model (prototype) using the functional programming language, Miranda. Due to memory limitations and CPU times required, the size of the reservoirs tested was relatively small. To overcome these limitations, a parallel processing system of at least 10 gigabytes of memory and a processing rate of at least 1 gigaflop is required. Such a system would

allow the investigation of even more detailed models; in addition to the simple model used here. The study uses the quad data structure to represent matrices, taking advantage of any sparsity in the matrices and it is easily parallelized. The research considered several data structures to represent vectors choosing lists of numbers because it is more efficient for a single processor; however, a binary tree is more conducive for parallel processing.

The authors have had experience implementing this model in both a functional language, Miranda, and in a procedural language, FORTRAN. Specification of the problem was much easier in Miranda than it was in a procedural language. An examination of the similarity of the Miranda specification of the Conjugate Gradient method to the mathematical specification illustrates this point. This specification is much more difficult to detail in FORTRAN. The abstract data type, MATRIX was implemented using quads and the abstract data type VECTOR was implemented as a list of numbers. These abstract data types allowed the problem to naturally decompose into parallel parts for execution by a parallel multiprocessor.

References

1. 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.

2. Bird, Richard and Wadler, Philip. "ntroduction to Functional Programming", Prentice Hall, Englewood Cliffs, N.J., 1988.

3. Burton, F. Warren and Kollias, J.G. "Functional Programming with Quadtrees", IEEE Software, Jan., 1988, pp. 90–97.

4. Elster, A.C. and Reeves, A.P. "Block-Matrix Operations Using Orthogonal Trees", Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Vol. II, Pasadena, CA., Jan., 1988, pp. 1555–1561.

5. Turner, David. "An overview of Miranda", SIGPLAN Notices, Dec., 1986, pp. 158–166.

6. Turner, David. "Miranda System Manual", 1987.

7. Wainwright, Roger. "A Software Kernel for a Pipeline Model for Solving Matrix Problems on a Hypercube Multiprocessor Applied to Reservoir Simulation", Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications, Monterey, CA., Mar. 6–8, 1989. Amoco Production Company Research Center Technical Report No. F89–C–2, Tulsa, OK., 1989.

8. Wainwright, Roger. "The Conjugate Gradient Method for Solution of Large Sparse Systems of Linear Equations on Hypercubes Applied to Reservoir Modeling", Proceedings of the ACM South Central Regional Conference, Nov. 16–18, 1989, Tulsa, OK. Amoco Production Company Research Center Technical Report No. F89–C–5, Tulsa, OK., 1989.

9. Wise, D.S. "Parallel Decomposition of Matrix Inversion Using Quadtrees", Proceedings 1986 International Conference on Parallel Processing (IEEE Cat. No. 86CH2355–6), pp. 92–99.

10. Wise, D.S. and Franco, John. "Costs of Quadtree Representation of Non-dense Matrices", Computer Science Dept., Indiana University, Bloomington, Indiana Technical Report No. 229, Oct., 1987.

2	3	0	0	0	0	0	0
1	2	3	0	0	0	0	0
0	1	2	3	0	0	0	0
0	0	1	2	3	0	0	0
0	0	0	1	2	0	0	0
0	0	0	0	0	2	0	0
0	0	0	0	0	0	2	0
0	0	0	0	0	0	0	2

Fig. 1a

Sparse Matrix

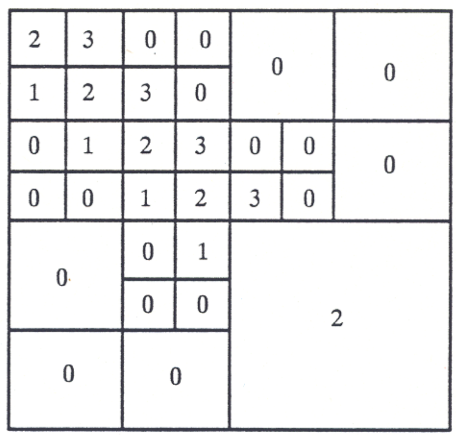

Fig. 1b

Quad Representation for Fig. 1a

TABLE I

Test Results for Various Size Reservoir Models

K	RxR Reservoir Size	NxN Matrix Size	CPU Time (Sec.)	Number of iter	Number of Reductions (millions)	Number of Claims (millions)	Number of GC
1	2	4	3	96	.2	.2	0
2	4	16	27	205	2	2	0
3	8	64	185	308	10	13	2
4	16	256	1,154 (19 min.)	430	59	76	15
5	32	1024	6,464 (1.8 hrs)	544	310	412	89
6	64	4096	37,994 (10.5 hrs)	592	1405	1914	654

TABLE II

Estimated Execution Times for Larger Reservoir Models

K	RxR Reservoir Size	NxN Matrix Size	CPU Time*	Wall Clock Time*
7	128	16,384	2.6 days	15.6 days
8	256	65,636	15.6 days	93.6 days
9	512	262,144	93.6 days	1.5 years

* estimate