A study of sparse matrix representations for solving linear systems in a functional language

ROGER L. WAINWRIGHT
Department of Mathematical and Computer Sciences, The University of Tulsa, 600 South College Avenue, Tulsa, OK 74104-3189, USA

MARIAN E. SEXTON
Amoco Production Company, Research Center, P.O. Box 3385, Tulsa, OK 74102, USA

Abstract

This paper compares three different sparse matrix representations in Miranda for solving linear systems of equations: quadtrees, binary trees and run-length encoding. It compares the three data structures in each of two common linear system solvers, Conjugate Gradient and SOR. The test problems used in the paper arise from a simple reservoir model.

Capsule review

Scientific computing is one of the most important applications of computer technology in which both performance and correctness are of paramount concern. Functional languages are well-suited to writing provably correct scientific code, since their 'mathematical flavour' permits elegant derivations of programs from the mathematical specifications commonly found in the scientific computing literature. One of the most important design parameters in such derivations is the representation of the ubiquitous *array*. The choice of array representation can have a significant impact on the performance of the end product.

This short paper explores the effectiveness of several representations of *sparse matrices* in a non-strict functional language, in the context of solving linear systems of equations using two common algorithms from the scientific computing literature. The methods used are relatively straightforward, but the clear presentation and thorough analysis provides valuable insight into the solution of an important class of problems considered notoriously difficult for languages without efficient array operations. The paper should be useful to anyone interested in solving scientific computing problems in a purely functional language.

1 Introduction

Functional programming languages have many advantages. These languages emphasize describing the results of a computation and focus less on how to perform the computation. Consequently, functional programs tend to be easier to read, write, verify and modify than procedural programs. In a so-called declarative language, the

programmer approaches the task at a high level of abstraction. Modern functional languages support pattern matching, user-defined data types, polymorphic types, functions and other features.

This study used Miranda†, a modern functional language system developed by Turner (1985, 1986). Miranda is a non-strict language in that a computation can have a well-defined outcome even when it contains some undefined component computations. The language allows this so long as the outcome does not rely on the resolution of these undefined components. Miranda employs lazy evaluation, that is, the evaluation of functions and arguments takes place only when some portion of the computation requires its results. Miranda allows infinite data structures, higher-order functions and polymorphism for flexibility in its typing restrictions (for details see Fleck, 1990). This particular application did not make use of Miranda's non-strict semantics or higher-order functions. However, Miranda's algebraic types were found to be quite helpful and were used extensively.

In functional languages, lists and trees are the natural data structures to use. However, many applications require different data structures, such as arrays. In a functional setting, arrays may be harder to implement efficiently. Until recently, it has been common practice not to consider a functional language for such applications, even if the use of arrays constitutes only a small portion of the total computation. Usually, a functional language is not considered for scientific computations, especially large applications, because of performance issues. Some recent projects have dispelled this notion. Sisal is general-purpose applicative language. Feo *et al.* (1990) discuss Sisal's performance as comparable to that of FORTRAN for scientific applications. Arvind *et al.* (1988) and Ekanadham *et al.* (1987) demonstrate the success of functional programming for scientific computation using parallel hardware. Our intent in this paper is to provide greater insight into the abstract data type, array, and into various implementations for arrays, in Miranda, for an industrial scientific application.

2 The Problem

Many scientific problems require the solution of linear systems of equations where the coefficient matrix is large, sparse and banded. Implementation of sparse banded matrices in a non-functional language is typically accomplished by storing each band as a one-dimensional array (vector). Isolating rows of a matrix, one at a time, is an essential operation for applications using matrices. Therefore, vectors are 'offset' such that the ith element in each vector corresponds to the ith row in the sparse matrix. Storing bands of a matrix as vectors is extremely efficient in both time and space in a non-functional language. This representation, however, is very inefficient for expressing a banded sparse matrix in a functional language. Representing each band as a list of numbers makes it difficult to extract and perform row operations efficiently.

This paper compares three different sparse matrix representations in Miranda for solving linear systems of equations: quadtrees, binary trees and run-length encoding.

† Miranda is a trademark of Research Software Ltd.

```
>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' cnt')
>       where
>       cnt' = 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
```

Fig. 1a. Conjugate Gradient method in Miranda.

Step 0. Initially choose x_0 and let $r_0 = p_0 = b - Ax$. Then compute $\langle r_0, r_0 \rangle$. Then,
For $k = 0, 1, 2, \ldots$

1. form $q_k = \mathrm{A}p_k$
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) $\langle r_{k+1}, r_{k+1} \rangle = \beta_k \langle r_k, r_k \rangle$
4. (a) $r_{k+1} = r_k - \alpha_k q_k$
   (b) $x_{k+1} = x_k + \alpha_k p_k$
   (c) $p_{k+1} = r_{k+1} + \beta_k p_k$

Fig. 1b. Mathematical specification for the Conjugate Gradient method.

It compares the three data structures in each of two common linear system solvers: Conjugate Gradient and SOR. 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 partial differential equation is approximated in space using an irregular block centred 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 of size $N = R^2$ to solve for $K = 1, 2, \ldots$. The particular reservoir model used in this paper involves 38 time steps. Thus, for this reservoir simulation it was necessary to solve 38 different linear systems using either the SOR or the Conjugate Gradient method to determine the solution. The functional specification of this problem indicates its potential for parallel processing according to Page *et al.* (1990).

3 Conjugate Gradient and SOR in Miranda

Throughout this paper, the authors will use the notation of the Miranda functional language. The meaning of Miranda, for the most part, should be fairly clear to the reader even with no prior knowledge of Miranda. For example, consider the Miranda function, *conj_grad* in Fig. 1a. The function *conj_grad* solves the linear system $Ax = b$ using the Conjugate Gradient method. Notice how the Miranda implementation of the Conjugate Gradient Method shown in Fig. 1 a closely resembles the traditional mathematical specification found in Fig. 1b. In Fig. 1b, r_k is the residual vector associated with the approximate solution vector x_k at each iteration. The definition of r is $b - Ax_k$ and it must be null when x_k is the exact solution. p_k represents the direction vector of the gradient at the kth iteration, and $< >$ denotes the inner product of two vectors. This demonstrates one of the primary advantages of specifying a problem in functional form, that of its simplicity. Note further that the function *conj_grad* in Fig. 1a is written abstractly, and that it is independent of the actual implementation of matrices and vectors. Figure 2 illustrates the SOR method in Miranda for solving the same linear system $Ax = b$. The *sor* function, like the *conj_grad* function, is easy to read because it directly corresponds to the mathematical specification found in textbooks. Again, this function specifies what is to be done independent of the implementation of vectors and matrices. The abstract data types for vectors and matrices, which could be used in a non-functional language as well, are shown below along with a list of associated operations.

Abstract Type Signatures

```
> abstype vector, matrix
> with
>     apply       ::(num->num)->vector->num->vector
>     breakvect   ::vector->[num]
>     lower_band  ::num->[num]->matrix
>     makevect    ::[num]->vector
>     mmadd       ::matrix->matrix->matrix
>     mvmult      ::matrix->vector->vector
>     subscript   ::vector->num->num
```

```
>   svmult       ::num → vector → vector
>   update       ::vector → num → num → vector
>   upper_band   ::num → [num] → matrix
>   vfoldl       ::(* → num → *) → * → vector → *
>   vmap         ::(num → num) → (vector → vector)
>   vmap2        ::(num→num→num)→(vector→vector→vector)
>   vneg         ::vector → vector
>   vvadd        ::vector → vector → vector
>   vvdot        ::vector → vector → num
>   vvmult       ::vector → vector → vector
>   vvsub        ::vector → vector → vector
```

4 Array representations

4.1 Quadtree representation

The quad data structure shown below (in Miranda) emits an economical representation for matrices. Burton and Kollias (1989), Wise (1986) and Wise and Franco (1987) 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 number of rows in the matrix, and the second represents the matrix as a quad

vector = = (*num*, [*num*])

matrix = = (*num*, *quad*)

quad :: = *Quad quad quad quad quad* | *Diag num*

In this study, the size of the matrix, represented in quad form, must be 2^d by 2^d for $d = 0, 1, 2 \ldots$ (it is relatively easy, with some additional work, to represent matrices in the quad form that are not a power of two). If a matrix has a constant value along the main diagonal, such as three, and all off-diagonal values equal to zero, then a single value, Diag 3, represents the contents of the matrix. Note that Diag 3 can represent a quad tree of any allowed size. A value of Diag 0, 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 was arbitrarily set from left to right as northwest, northeast, southwest and southeast. In this way, each matrix has a unique representation. The quad representation exhibits a saving in memory, since the zero values can be compressed into Diag num's. Furthermore, it represents a saving 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 3

```
>sor
> :: matrix→iter_solution→iter_solution
>step_sor
> :: matrix→[num]→iter_solution→
>    iter_solution
>
>sor a (Iter_sol x r b main_diag cnt)
> = until converge
>         nextiter
>         (Iter_sol x r b main_diag cnt)
>    where
>    converge
>      (iter_sol x r b main_diag cnt)
>      = (rr < eps)
>        where
>        rr = vvdot r r
>        eps = ·000001
>    nextiter
>      (Iter_sol x r b main_diag cnt)
>      = incr_cnt a
>           (step_sor
>             a [1..lin_sys_size]
>             (Iter_sol x r b
>                   main_diag cnt))
>
> incr_cnt a (Iter_sol x r b main_diag cnt)
> = (Iter_sol x r' b main_diag (cnt+1))
>    where
>    r' = vvsub b (mvmult a x)
>
> step_sor a [ ]
>         (Iter_sol x r b main_diag cnt)
> = (Iter_sol x r b main_diag cnt)
>
> step_sor a (i:is)
>         (Iter_sol x r b main_diag cnt)
> = step_sor
>     a is
>     (Iter_sol x' r b main_diag cnt)
>    where
>    xi_new = xi_old + w/aii * ri
>    ri = bi − (vvdot' x ai)
>    ai = subscript_mat a i
>    bi = subscript b i
>    aii = subscript main_diag i
>    w = 1·66
>    xi_old = subscript x i
>    x' = update x i xi_new
```

Fig. 2. SOR method in Miranda.

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

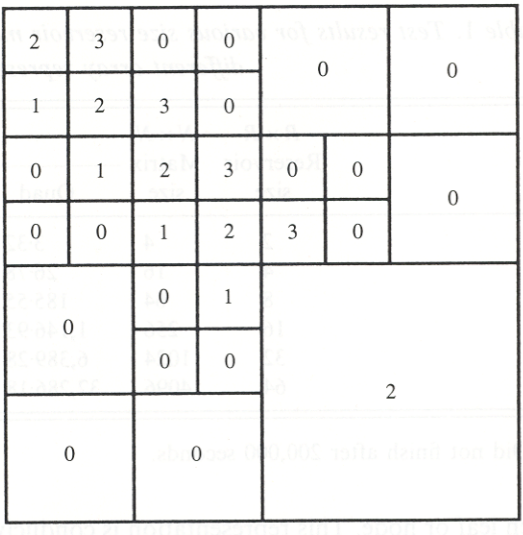

Fig. 3. (a) Sparse matrix; (b) Quad representation for Fig. 3a.

depicts an 8×8 banded matrix along with the corresponding unique quad representation

Examples:

(4, *Diag* 3) *represents* 3 0 0 0 *and*
0 3 0 0
0 0 3 0
0 0 0 3

(4, *Quad* (*Diag* 3) *represents* 3 0 0 1
(*Quad* (*Diag* 0) (*Diag* 1) (*Diag* 0) (*Diag* 0)) 0 3 0 0
(*Quad* (*Diag* 0) (*Diag* 0) (*Diag* 2) (*Diag* 0)) 0 0 3 0
(*Diag* 3)) 2 0 0 3

4.2 Binary tree representation

The study considered a binary tree structure for representing rows of a matrix. A matrix, then, is a list of binary trees. The *bin_tree* structure, in the following Miranda definition, represents a matrix row by a binary tree in which all of the valid numerical data are in the leaves of the tree

vector == (*num*, [*num*])
matrix == (*num*, [*bin_tree*])
bin_tree ::= *Bin_tree bin_tree bin_tree* | *Scalar num*

A Scalar num represents a single number or any group of numbers that are all the same. The compression of a group of like numbers into one Scalar num constitutes a saving in memory over, for example, a representation that has only one value in

Table 1. *Test results for various size reservoir models using Conjugate Gradient with different array representations.*

$R \times R$ Reservoir size	$N \times N$ Matrix size	*CPU* time (s) Quad	Binary tree	Run-length
2	4	3·32	3·68	3·67
4	16	26·78	37·53	31·72
8	64	185·55	332·58	205·08
16	256	1,146·93	2,760·45	1,259·38
32	1024	6,389·28	26,442·67	8,514·57
64	4096	32,286·18	*	74,941·83

* Did not finish after 200,000 seconds.

each leaf or node. This representation is conducive to sparse matrices. The definition for vectors, however, which are dense in this problem, remains lists of numbers

Examples:

(4, [(*Bin_tree* (*Bin_tree* (*Scalar* 3) (*Scalar* 0)) *represents* 3 0 0 0
(*Scalar* 0)), 0 3 0 0
(*Bin_tree* (*Bin_tree* (*Scalar* 0) (*Scalar* 3)) 0 0 3 0
(*Scalar* 0)), 0 0 0 3
(*Bin_tree* (*Scalar* 0)
(*Bin_tree* (*Scalar* 3) (*Scalar* 0))),
(*Bin_tree* (*Scalar* 0)
(*Bin_tree* (*Scalar* 0) (*Scalar* 3)))])

(4, [(*Bin_tree* (*Bin_tree* (*Scalar* 3) (*Scalar* 0)) *represents* 3 0 0 1
(*Bin_tree* (*Scalar* 0) (*Scalar* 1))), 0 3 0 0
(*Bin_tree* (*Bin_tree* (*Scalar* 0) (*Scalar* 3)) 0 0 3 0
(*Scalar* 0)), 2 0 0 3
(*Bin_tree* (*Scalar* 0)
(*Bin_tree* (*Scalar* 3) (*Scalar* 0))),
(*Bin_tree* (*Bin_tree* (*Scalar* 2) (*Scalar* 0))
(*Bin_tree* (*Scalar* 0) (*Scalar* 3)))])

4.3 Run-length encoding representation

The third data structure the study considered represents a sparse matrix using a run-length encoding scheme. A *Run_pair*, shown in the definition below, consists of the repetition factor followed by the numerical value. Thus, *run_len* is either a *Run_pair*

Table 2. Test results for various size reservoir models using SOR with different array representations.

$R \times R$ Reservoir size	$N \times N$ Matrix size	*CPU* time (s)		
		Quad	Binary tree	Run-length
2	4	34·40	53·32	53·78
4	16	265·42	448·02	385·73
8	64	2,255·72	3,097·02	2,081·47
16	256	26,379·45	22,822·42	13,595·83

or a list of nums. In this way, a list of nums depicts a dense vector, which is more efficient, and a run-length scheme depicts a sparse vector. A matrix is a collection of vectors made up of *Run_pairs*

vector = = (*num*, [*run_len*])
matrix = = (*num*, [[*run_len*]])
run_len : : = *Run_pair num num* | *List* [*num*]

Examples:

(4, [[*Run_pair* 1 3, *Run_pair* 3 0] *represents* 3 0 0 0
[*Run_pair* 1 0, *Run_pair* 1 3, *Run_pair* 2 0] 0 3 0 0
[*Run_pair* 2 0, *Run_pair* 1 3, *Run_pair* 1 0] 0 0 3 0
[*Run_pair* 3 0, *Run_pair* 1 3]]) 0 0 0 3

(4, [[*Run_pair* 1 3, *Run_pair* 2 0, *Run_pair* 1 1] *represents* 3 0 0 1
[*Run_pair* 1 0, *Run_pair* 1 3, *Run_pair* 2 0] 0 3 0 0
[*Run_pair* 2 0, *Run_pair* 1 3, *Run_pair* 1 0] 0 0 3 0
[*Run_pair* 1 2, *Run_pair* 2 0, *Run_pair* 1 3]]) 2 0 0 3

In summary, representing sparse banded matrices as quads or binary trees is possible in a non-functional language. However, these representations are more costly in time and space compared with representing each band as a vector. Run-length encoding, however, is an efficient representation for sparse matrices in non-functional languages. However, if the sparsity has a regular pattern, like bands, storing the bands as vectors is the preferred representation.

5 Vector representations

This investigation considered several different data structures for vectors other than lists of numbers. Wise (1986) and Wise and Franco (1987) suggest 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. Operations involving two dense vectors executed faster when the vectors were represented as lists of numbers compared to a quad-like representation.

6 Results

The current research investigated reservoirs of size $R = 2^K$, yielding matrices of size $N = R^2$, for $K = 1..6$. The study ran all of the simulations on a Sun4 workstation using release 2.009 (November 14 1989) of Miranda, and each simulation involved solving a series of 38 linear systems.

6.1 Conjugate Gradient results

Table 1 shows the results of solving various sized reservoirs using the Conjugate Gradient method with each array data structure. In each case, the quad representation for a matrix was clearly the best and the binary tree representation was the worst.

6.2 SOR results

Table 2 shows the results of solving various sized reservoirs using the *SOR* method with each array data structure. The nature of *SOR*, independent of the implemented data structure, is such that it requires more memory than the Conjugate Gradient method (problems larger than 256×256 ran out of memory). In the case of *SOR*, the best matrix representation was run-length encoding. The larger the model, the better the run-length data structure performed. The quad representation worked best for small models; however, these are of little interest. The quad representation actually performed worse as the matrix size increased. This suggests that when using *SOR* and dense matrices, the quad representation for matrices should be avoided. Recall that the *SOR* method, at each iterative step, involves the *i*th row of the matrix. Hence, it is necessary to isolate each row of the matrix, one at a time. This is very easy for matrices constructed as a list of binary trees or as a list of *Run_pairs*, since the *i*th item in the list can be located very easily. In the quad representation of a matrix, however, isolating a row of a matrix is not natural and requires a tremendous effort. The poor performance of the quad representation, when using *SOR*, reflects the problem of row isolation. The Conjugate Gradient method, however, does not require isolating any row of the matrix, thus the quad representation exhibited superior performance.

6.3 Cost analysis

If the number of non-zero elements in a vector of size n is $0(n)$, then the memory cost of representing the vector using quads, run-length encoding, binary trees, or lists of numbers is $0(n)$ in each case. Furthermore, the cost of representing a dense matrix of

Table 3. Analysis of mvmult using Conjugate Gradient with different array representations.

Timing analysis	Array representation	$N \times N$ matrix size 256	1024	4096
CPU time for one *mvmult* normalized for Quadtree	Quadtree	1·00	1·00	1·00
	Run-length	1·06	1·34	2·21
	Binary tree	2·64	3·85	9·38
CPU time for the reservoir model normalized for Quadtree	Quadtree	1·00	1·00	1·00
	Run-length	1·10	1·33	2·32
	Binary tree	2·41	4·14	> 10*
% *CPU* time for the reservoir model attributed to *mvmult*	Quadtree	88	93	95
	Run-length	85	94	91
	Binary tree	96	87	> 90*

* Estimated.

size n by n is $0(n^2)$ for all three representations. When the number of non-zero elements in a vector of size n is 0(1), sparse, the memory cost of representing the vector using quads, run-length encoding, or binary trees is 0(1). Furthermore, the cost of representing a sparse matrix of size n by n is $0(n)$ for all three array representations. The run-length and the binary tree representations are able to reduce memory due to sparsity only on a row by row basis. The quad tree representation, however, takes advantage of blocks of zeros across several consecutive rows or columns. This ability to compress blocks of zeros is the primary reason the quadtree representation is superior when using the Conjugate Gradient method. Notice in Fig. 1a that each Conjugate Gradient iteration uses the matrix a only once, in *mvmult* (matrix-vector multiply). The other operations involve vector-vector dot products, scalar-vector multiplies, and in general, only operations involving vectors. Independent of the matrix representation, these vectors are dense, and are represented as lists of nums.

6.4 mvmult analysis

The dominant calculation during each iteration is *mvmult*. The cost of the *mvmult* operation is independent of the data values, but dependent on the data structure. Therefore, the performance of a data structure within *mvmult* determines its success in the entire simulation. Since the quadtree representation takes greater advantage of the sparsity of the matrix, it was clearly superior.

Table 3 shows the time required for a single *mvmult*, normalized for quadtree, using all three array representations. In addition, Table 3 gives the total *CPU* times from Table 2, also normalized for quadtree, using all three array representations. Finally, Table 3 shows *mvmult*'s percentage of total CPU time when solving various sized reservoir models. As expected, the percentage of time spent in *mvmult* is quite high and it is independent of array implementation.

7 Summary, conclusions and further research

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 modelling, 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 (for details see Page *et al.*, 1990). This prototype application spent the majority of its execution time solving sparse linear systems. Further, it implemented and tested three different array representations, written in Miranda, using two common linear system solvers, Conjugate Gradient and *SOR*. The quad representation for sparse matrices out-performed the other representations when using the Conjugate Gradient method. However, run-length encoding proved to be the best sparse matrix representation when using *SOR*.

Acknowledgements

We would like to thank F. Warren Burton for his thorough reading of the manuscript, which resulted in numerous improvements and clarifications. We gratefully acknowledge the excellent suggestions made by the referees.

References

Arvind and Ekanadham, K. 1988. Future scientific programming on parallel machines. Laboratory for Computer Science, Massachusetts Institute of Technology, CSG-Memo-272.

Burton, F. W. and Kollias, J. G. 1989. Functional programming with quadtrees. *IEEE Software*, 90–97 (Jan).

Ekanadham, K. and Arvind. 1987. SIMPLE: Part I – An exercise in future scientific programming. Laboratory for Computer Science, Massachusetts Institute of Technology, CSG-Memo-273. (Simultaneously published as Technical Report RC 12686, IBM T. J. Watson Research Center, Hawthorne, NY.)

Feo, J. T., Cann, D. C. and Oldehoeft, R. R. 1990. A report on the Sisal language project. *J. Parallel and Distributed Computing*, **10**: 349–366.

Fleck, A. C. 1990. A case study comparison of four declarative programming languages. *Software – Practice and Experience*, **20** (1): 49–65 (Jan).

Page, R. L., Sexton, M. E. and Wainwright, R. L. 1990. A functional program describing a simple reservoir model and its potential for parallel computation. In *Proc. ACM/IEEE Symp. on Applied Computing*, pp. 85–91.

Turner, D. 1985. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architectures*, Volume 201 of *Lecture Notes in Computer Science*, Springer-Verlag.

Turner, D. 1986. An overview of Miranda. *SIGPLAN Notices*, 158–166 (Dec).

Wise, D. S. 1986. Parallel decomposition of matrix inversion using quadtrees. In *Proc. Int. Conf. on Parallel Processing*, pp. 92–99.

Wise, D. S. and Franco, J. 1987. Costs of quadtree representation of non-dense matrices. Computer Science Department, Indiana University, Bloomington, Indiana, Technical Report No. 229 (Oct).