Sparse matrix representations in a functional language

P. W. GRANT, J. A. SHARP, M. F. WEBSTER and X. ZHANG
Department of Computer Science, University of Wales, Swansea, Swansea SA2 8PP, UK

Abstract

This paper investigates several sparse matrix representation schemes and associated algorithms in Haskell for solving linear systems of equations arising from solving realistic computational fluid dynamics problems using a finite element algorithm. This work complements that of Wainwright and Sexton (1992) in that a Choleski direct solver (with an emphasis on its forward/backward substitution steps) is examined. Experimental evidence comparing time and space efficiency of these matrix representation schemes is reported, together with associated forward/backward substitution implementations. Our results are in general agreement with Wainwright and Sexton's.

Capsule Review

Computational fluid dynamics problems occur in many industrial applications of great commercial importance. If functional languages are to be successful in the arena of scientific computation, they must lead to software that performs fluid dynamics computations efficiently. The research that Grant *et al.* describe in this paper studies several implementation approaches to functional language software for a computationally intensive portion of fluid dynamics applications. It compares these approaches in terms of time and space, both analytically and experimentally, and it compares the effectiveness of functional languages in this domain to that of conventional programming languages. This research reveals information that is central to the solution of an important problem, and suggests some promising avenues of research to further iluminate the use of functional languages in scientific applications.

1 Introduction

Functional languages have many advantages over conventional procedural languages. However, the engineering community remains to be convinced that they are suitable for large scientific computations. As part of the UK FLARE (Functional Languages Applied to Realistic Exemplars) project, we have been using Haskell (Hudak *et al.*, 1992) to develop a Computational Fluid Dynamics (CFD) application. A sequential version of a Taylor–Galerkin/pressure-correction finite element algorithm (Townsend and Webster, 1987; Hawken *et al.*, 1990), originally implemented in Fortran and used extensively by the CFD research group in the Department of Computer Science at Swansea, has been recoded in Haskell. This algorithm involves solving large *sparse* systems of linear equations. The particular application area of

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

this algorithm is that of numerically simulating non-Newtonian flows, representing a field of some considerable industrial importance. Examples of such flows can be found within such diverse areas as polymer processing, the oil production industries, food processing and medicine.

There are several properties of this application which make it of interest to functional programmers. Firstly, complex fluid flow problems normally involve a large amount of data and solving such problems involves intensive numerical computation. As a result, a program which solves such problems must manipulate large data structures and perform numerical computation efficiently. In particular, efficient array operations are usually expected by engineers. Acceptance of functional programming technology by users solving engineering problems may largely depend on its computational efficiency. Secondly, extensive parallelism can be extracted from the algorithm under investigation. Finally, research into the advantages and problems of applying functional programming technology to the important field of numerical analysis is still in its infancy, and there is much scope for further work here.

Haskell (Hudak *et al.*, 1992; Davie, 1992) is a modern lazy pure functional language designed by a working group of leading researchers, supporting pattern matching, user-defined data types, polymorphic types, functions and other features. Some of our previous practical experience in the application of Haskell has been published in several papers (Grant *et al.*, 1992a, 1993a, 1993b; Zhang *et al.*, 1994a, 1994b). In this paper, we mainly concentrate on the performance of solving linear equations arising from a Taylor–Galerkin/pressure-correction algorithm using a Choleski direct method (Wilkinson and Reinsch, 1971) using several different sparse matrix representations for Choleski factors. However, this investigation is not solely a comparison of data structures as several different Choleski substitution implementations have been adopted to suit different situations. We are interested in the optimal performance functional programs can provide rather than issues associated purely with data structures.

In the above algorithm, the Choleski decomposition component of the Choleski direct method is only carried out once at the start of the time stepping procedure for each entire program execution. This means the Choleski decomposition is not a significant component in the entire iterative time-stepping algorithm and is therefore not under examination in this paper. Furthermore, a Choleski factor for a particular problem is never updated in the entire execution as this is static, and in this respect supporting efficient data update is not required here. However, some tests are also reported on the use of mutable arrays that have been made available only recently in a compiler provided by a group at Glasgow (Peyton Jones and Wadler, 1993).

There are two reasons why we are interested in sparse representations:

1. The practical systems we are dealing with can be huge but sparse which makes sparse representations *essential* even in programs written in procedural languages.
2. Functional programming normally demands more space than procedural programming. Any saving on space will generally result in less garbage collection activities and hence, in turn, improve time efficiency.

The patterns of sparse matrices that arise in our problems can be changed by reordering both matrix rows and columns (together). Although a minimum bandwidth pattern is normally adopted (for our Fortran program for example), in this investigation we always apply a minimum degree reordering scheme, which reduces the total number of non-zero entries in Choleski factors (Duff *et al.*, 1986), to make a generalised envelope sparse matrix representation scheme (described later) effective. This reordering scheme reduces the total number of non-zero entries in the resultant Choleski factors but may spread non-zero entries over the entire matrix. As the Choleski decomposition is not under investigation, further details are omitted.

The Haskell code under investigation also includes the implementation of a Jacobi iterative linear equation solver plus other components. The selection of data structures has been made with a view to the optimisation of run time of the overall code.

This work is complementary to Wainwright and Sexton's work (1992). The emphasis of their investigation was on sparse matrix representations in Miranda for solving linear systems of equations using two iterative system solvers: Conjugate Gradient and Successive Over-Relaxation (SOR) (Hageman and Young, 1981). They found that a quadtree representation out-performed other representations when the Conjugate Gradient method was used. However, the quadtree representation was less favourable compared with a run-length encoding scheme when using SOR. They suggested that this was because the SOR method needs to isolate each row of a matrix, one at a time, which is not a natural operation on quadtrees. Both direct solvers and iterative solvers are used extensively in practice. There are two main differences between the Choleski schemes and the schemes investigated by Wainwright and Sexton. The Choleski method first decomposes an original system matrix into a pair of Choleski factors. This operation introduces extra non-zero entries. In the solution stage, the Choleski method performs both row and column oriented operations on Choleski factors, while the Conjugate Gradient and SOR methods perform only row oriented (or only column oriented) operations. These two characteristics are not present in Wainwright and Sexton's iterative methods and warrant further investigation.

2 A Taylor–Galerkin/pressure-correction algorithm

For completeness, in this section we present a brief description of the numerical algorithm used. The flow of an incompressible fluid can be described by the Navier–Stokes equation (Cuvelier *et al.*, 1986)

$$\rho\frac{\partial \mathbf{u}}{\partial t} + \rho\mathbf{u}\cdot\nabla\mathbf{u} - \mu\nabla^2\mathbf{u} + \nabla p = \mathbf{f} \tag{1}$$

together with the equation that constrains the flow to be incompressible:

$$\nabla\cdot\mathbf{u} = 0. \tag{2}$$

In the above $\mathbf{u}$ refers to the *velocity* of particles in the fluid, p the *pressure*, ρ the *density*, and μ the *viscosity*. $\mathbf{f}$ is an external *force* acting on the fluid that is assumed

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

to vanish in the current implementation. These equations describe the velocity and pressure in a fluid flow as a function of time and location. Here we concentrate on Newtonian fluids with constant viscosity. These are highly viscous problems that display inertial effects and strong vortex structure. The numerical challenges posed by such systems lie in the discrete resolution of these transient convection-diffusion equations within an incompressible framework. By numerically solving equations (1) and (2), we can simulate the evolution of velocity and pressure in a transient flow. To achieve this goal, a Taylor–Galerkin/pressure-correction algorithm is employed.

The Taylor–Galerkin/pressure-correction algorithm incorporates four independent fractional stages per time-step for viscous incompressible flows. The merits of the scheme are its considerable computational accuracy and its space and time efficiency. In this investigation we consider only two-dimensional spatial domains, that are triangulated into a finite-element mesh.

This scheme discretises the Navier–Stokes equations and gives rise to equations of the following form:

$$\frac{2\rho}{\Delta t}\mathbf{M}(U^{(n+1/2)} - U^{(n)}) = b_1(U^{(n)}, P^{(n)}), \tag{3}$$

$$\frac{\rho}{\Delta t}\mathbf{M}(U^{(*)} - U^{(n)}) = b_1(U^{(n+1/2)}, P^{(n)}), \tag{4}$$

$$\frac{\Delta t}{2\rho}\mathbf{K}(P^{(n+1)} - P^{(n)}) = b_2(U^{(*)}), \tag{5}$$

$$\frac{2\rho}{\Delta t}\mathbf{M}(U^{(n+1)} - U^{(*)}) = b_3(P^{(n+1)} - P^{(n)}), \tag{6}$$

where $\mathbf{M}$ is an augmented mass matrix, $\mathbf{K}$ a pressure difference stiffness matrix, $U^{(n)}$ the velocity solution vectors, and $P^{(n)}$ the pressure solution vectors.

Precise details of these complex equations are given elsewhere (Townsend and Webster, 1987; Hawken *et al.*, 1990). However, the essential points to note are:

1. The right-hand-sides (RHS) and solutions in (3), (4) and (6) are *multiple* vectors corresponding to individual velocity components. The right-hand-sides and solutions in (5) are, however, *single* vectors.
2. The order of solution is to start with (3) and (4) to produce the intermediate result $U^{(*)}$ from $(U^{(n)}, P^{(n)})$, then (5) calculates $P^{(n+1)}$ from $(U^{(*)}, P^{(n)})$, and finally (6) computes $U^{(n+1)}$ from $(U^{(*)}, P^{(n)}, P^{(n+1)})$.
3. Matrices $\mathbf{M}$ and $\mathbf{K}$ are *sparse*, *symmetric*, *banded* and *positive definite* under appropriate boundary conditions.

A Jacobi iteration method (Hageman and Young, 1981), which does not necessitate the explicit assembly of system matrices arising from these finite element problems, is used for the solution of (3), (4) and (6). For solving (5) we employ a classical Choleski direct method (Wilkinson and Reinsch, 1971). There are many sparse matrix representation schemes which are suitable in this context, and it is the performance of these schemes that forms the main part of this study.

2.1 A Choleski direct method

System (5) does not vary across time-steps (the others do). This makes an explicit assembly of this particular system matrix and the employment of a classical Choleski direct method (Wilkinson and Reinsch, 1971) for its solution worthwhile. For readers not familiar with this method we give the details below.

The solution of the system, $\mathbf{Ax} = \mathbf{b}$, is determined by the forward and backward substitution steps:

$$\mathbf{Ly} = \mathbf{b}, \qquad \mathbf{L}^T\mathbf{x} = \mathbf{y} \tag{7}$$

where $\mathbf{L}$ is a lower triangular matrix satisfying $\mathbf{LL}^T = \mathbf{A}$. The elements of $\mathbf{L}$ are given by

$$l_{ij} = (a_{ij} - \sum_{p=1}^{j-1} l_{ip}l_{jp})/l_{jj}, \quad j < i \tag{8}$$

$$l_{ii} = (a_{ii} - \sum_{p=1}^{i-1} l_{ip}l_{ip})^{1/2}. \tag{9}$$

$\mathbf{L}$ is normally termed a Choleski factor and its construction is known as Choleski decomposition. The forward/backward substitution steps can be described by the following calculations:

$$y_i = (b_i - \sum_{p=1}^{i-1} l_{ip}y_p)/l_{ii}, \quad i = 1, 2, ..., n \tag{10}$$

$$x_i = (y_i - \sum_{p=i+1}^{n} l_{pi}x_p)/l_{ii}, \quad i = n, n-1, ..., 1 \tag{11}$$

where $n \times n$ is the size of $\mathbf{A}$. These two substitution steps compute $\mathbf{y} = \mathbf{L}^{-1}\mathbf{b}$ and $\mathbf{x} = (\mathbf{L}^T)^{-1}\mathbf{y}$ without explicitly evaluating $\mathbf{L}^{-1}$ and $(\mathbf{L}^T)^{-1}$.

Although the forward/backward substitution steps must be performed at each time-step, the Choleski decomposition of $\mathbf{A}$ needs to be performed only once. A typical simulation may require thousands of time-steps. We therefore limit our attention only to implementations of the forward/backward substitution steps in this paper.

Equations (10) and (11), although quite similar, actually imply different entry access order to the Choleski factor $\mathbf{L}$ using straightforward implementations. To calculate y_i, the ith *row* of $\mathbf{L}$, $\{l_{ip}, p = 1, 2, ..., i-1\}$, is required, but, to calculate x_i, the ith *column*, $\{l_{pi}, p = i+1, i+2, ..., n\}$, is required. In other words, the forward substitution accesses the matrix row by row, whilst the backward substitution performs access in a column by column manner. If matrices are stored row by row, the backward substitution may not be efficient for very large matrices under a straightforward implementation of equation (11) due to swapping and caching activities caused by accessing entries in different rows. We have implemented three possible solutions to this problem. The first is achieved by providing a separate copy of the Choleski factor for the backward substitution stored in a column by column fashion. The second is through storing the Choleski factor in a 'neutral' form. These

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

two solutions involve mainly data structure changes, examples of which are given in later sections. There is however a third solution, i.e. reformulating the backward substitution equation (11) to establish a row-oriented scheme. If we introduce a set of intermediate vectors $\mathbf{z}^{(j)} = \{z_i^{(j)} | i = 1, 2, ..., n\}$ and define them as

$$z_i^{(0)} = y_i, \qquad i = 1, 2, ..., n \tag{12}$$

$$z_i^{(j)} = z_i^{(j-1)} - l_{p,i} x_p, \qquad p = n + 1 - j,\ i = 1, 2, ..., n - j\ (for\ \ j > 0) \tag{13}$$

$$= y_i - \sum_{p=n+1-j}^{n} l_{pi} x_p$$

then equation (11) becomes

$$x_i = z_i^{(n-i)} / l_{ii}. \tag{14}$$

It can be easily verified that if we evaluate these equations in the order of $\mathbf{z}^{(0)}$, x_n, $\mathbf{z}^{(1)}$, x_{n-1}, ..., $\mathbf{z}^{(n-1)}$, x_1, the access to $\mathbf{L}$ will be row by row.

The extra cost of this scheme is the storage of the $\mathbf{z}^{(j)}$ vectors. In a procedural implementation, all the $\mathbf{z}^{(j)}$ vectors can share the same storage. In a functional language where in-place update is unavailable, this cost is at least as heavy as a two matrix copy scheme, where both a row-by-row version and a column-by-column version of the $\mathbf{L}$ factor are retained. The emergence of the Glasgow mutable arrays (Peyton Jones and Wadler, 1993), however, makes this scheme attractive in a functional implementation and calls for future investigation. Despite the necessity of updating RHSs, experiments show that the implementation of equations (12)–(14) has much better speed efficiency than a straightforward implementation of equation (11) in a one-copy scheme context. The implementation of equations (12)–(14) has been adopted for all the one-copy schemes cited below.

2.2 Comparison of Choleski and some other equation solvers

To facilitate comparisons between our strategy and that of Wainwright and Sexton's study (1992), a short comparison of the Choleski method and some other equation solvers is included in this sub-section.

Equation solvers can be generally classified as direct or iterative methods. Normally, direct methods are fast, and give high precision solutions, but require explicit system matrix assembly and may be difficult to parallelise. The Choleski method is one such method. In contrast, iterative methods may prove to be relatively slow. Their precision of solution normally increases with the number of iterations performed. Some of them are more suitable for parallel implementations and some do not require explicit system matrix assembly (the Jacobi method for example). The Conjugate Gradient and SOR methods investigated by Wainwright and Sexton are examples of efficient iterative solvers. The operations involved in these two solvers are mainly row-oriented vector-vector and matrix-vector multiplications.

There are two main operational differences when comparing the forward/backward substitution steps with the Conjugate Gradient and SOR. First, it is not difficult to appreciate from equations (10) and (11) that diagonal entries and non-diagonal

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

entries in a Choleski factor involve different operations. The property of the system matrix guarantees non-zero diagonal entries. This leads to differences in implementation compared with some sparse representations implemented by Wainwright and Sexton – segregating diagonal entries from non-diagonal entries and using sparse representations only for non-diagonals. Second, column-oriented operations are required in the Choleski method but not in the Conjugate Gradient and SOR methods. This adds extra complication to our investigation.

3 Data structures and Haskell implementations

3.1 Vector representations

In most of our tested schemes, the following data structure is used for representing system right-hand-side and solution vectors.

```
type S_Vector = S_array Double
```

where `S_array` is our implementation of the Haskell arrays (Zhang *et al.*, 1994a) using a concrete tree implementation.

There are a number of cases in our code where it is reasonable for us to use the same structure for all vectors. This can also simplify code maintenance and other tasks such as code parallelisation. There are two reasons why a tree structure has been chosen instead of the obvious Haskell standard built-in array implementation in the `hbc` (Augustsson, 1993) and Glasgow Haskell compilers which store array elements in contiguous memory locations. The first is the consideration of updates to vector entries which arise in the second Choleski substitution algorithm described earlier. Entry update supported by the standard Haskell arrays is simply not space (nor time) efficient ($O(n)$ where n is the length of the vector). The second is that the tree structure fits naturally into the recursive algorithm used in the quadtree scheme (see section 3.2.3). Therefore the tree structure is employed for all representation schemes.

The `S_array` has also been designed to support sparse vector representation, and is therefore also used for representing matrix rows (described later). The precise definition of `S_array` is

```
data S_array a = Mk_t_Array (Int,Int) (Maybe a) (Bin_tree a)

data Bin_tree a = Fork Int (Bin_tree a) (Bin_tree a) | Leaf a | Null

data Maybe a = Nothing | Just a
```

The first parameter `(Int,Int)` after the `Mk_t_Array` constructor is for the indication of array index bounds and the first parameter `Int` after the `Fork` is for keeping key information to facilitate the implementation of some array operations (although it is not strictly necessary, we found its presence improved execution speed). Figure 1 illustrates how this structure supports sparse representation of sparse vectors. Counterparts of all standard Haskell array operations have been defined on `S_array`. Table 1 lists the name correspondences of some of our operations and the standard Haskell operations. Operators `(!)` and `(//)` provide array entry accesses and

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

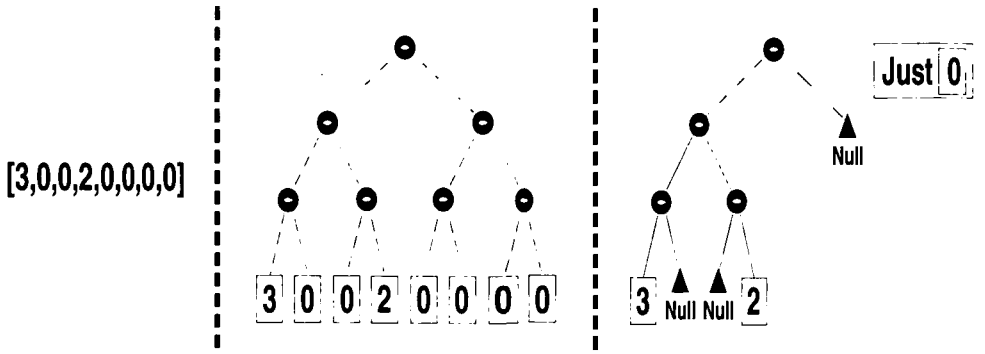

Fig. 1. An example of the tree representation.

Table 1. *Name correspondence of some array operations*

	S_array	(!^)	(//^)	s_bounds	s_array	s_listArray	s_accum
Standard Haskell		(!)	(//)	bounds	array	listArray	accum

updates respectively. Functions `array`, `listArray` and `accum` are various array constructors. Function `bounds` returns the bounds of an array.

Our scheme is slightly different from the implementation of Wainwright and Sexton (1992), whose corresponding binary tree data structure has a `Scalar num` constructor in place of our `Null`. In other words, our approach expects all subtrees represented by `Nulls` in the same tree to share the same numerical entry value while Wainwright and Sexton's approach does not. The actual shared numerical value (normally 0) in our case is stored in the `Maybe` parameter field (Spivey, 1990). Although this can be done in a more straightforward manner, for example using `Double` , `Maybe` allows the value to be undefined (being `Nothing`) and facilitates array entry access violation checks.

We have also conducted some experimentation using a monadic mutable array (see section 3.2.5) implemented in the Glasgow Haskell compiler (Peyton Jones and Wadler, 1993) for RHS and solution vector representation (a further study is planned when this implementation has reached a more mature state). A monadic mutable array contains a string of non-pointer bytes and stores unboxed (or untagged) array elements in consecutive memory locations. It allows constant time read/write access to array elements and also supports in-place update of array entries.

There are three basic operations defined on these arrays: `newDoubleArray`, `readDoubleArray` and `writeDoubleArray` (`Double` data type version). These operations need to be used in conjunction with monadic state operations.

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

3.2 Matrix representations

This paper compares four basic sparse matrix representations. These are a binary tree scheme, a list of row-segments scheme, a generalised envelope scheme (Liu, 1991) and a quadtree scheme. A non-sparse representation is also implemented for comparison. As has been mentioned earlier, our implementation of binary trees, which form the base of the Binary tree representation scheme discussed below, is slightly different from the corresponding scheme presented by Wainwright and Sexton (1992). A few non-crucial differences can also be found between our and their quadtree implementations due to our emphasis on representing triangular matrices for the Choleski method. Our list of row-segments scheme is actually a simple variation of their run-length encoding and in addition two two-matrix-copy schemes are also discussed.

In the following section, we describe the data structures and implementations of the four sparse representations together with the corresponding variations of the forward/backward substitution steps in Haskell. To test the quadtree scheme and as a reasonable basis for comparison with other schemes, we transform Choleski factors derived in the generalised envelope form to quadtrees.

3.2.1 Binary trees and lists of row-segments

The implementation of equation (7) is straightforward:

```
chl_method :: Chl_factor -> S_Vector -> S_Vector
chl_method chl_factor b = x
  where
  x = backwd_subs chl_factor y
  y = forwd_subs chl_factor b
```

where `S_Vector` is the data type defined in section 3.1. This function takes a Choleski factor and a system RHS vector as input and derives a solution vector. The vector **y** in equation (7) is implicit.

For both the binary tree scheme and the list of row-segments scheme, we have

```
type Chl_factor = (Diagonal, Array Int Sparse_vec)
```

where type `Diagonal` is defined as

```
type Diagonal = Array Int Double
```

This definition segregates the diagonal entries (the first component) from the remainder. The difference lies in how `Sparse_vec` is defined. The same structure is used for the implementation of the non-sparse representation except `[Double]` replaces `Sparse_vec`. There is an alternative here, that arrays instead of lists are used wherever appropriate. Assuming row-by-row storage, for the forward substitutions, a list data structure is adequate because no random access to list entries is necessary. For the backward substitutions, an array data structure is preferred but this still does not prevent possible swapping and caching activities. We have investigated a two-matrix-copy strategy to avoid the implications of both random accesses and

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

possible swapping and caching. Tests show that sequential access to all entries in a list is actually slightly faster than to all entries in an array for the hbc compiler.

For the list of row-segments scheme, the Sparse_vec is

```
type Sparse_vec = [(Int,[Double])]
```

The Int component is used to specify the starting point of a non-zero row-segment and the [Double] contains the row-segment. This is obviously a simple variation of the following Haskell translation of the run-length encoding scheme

```
type run_len = Run_pair num num | List [num]
```

used by Wainwright and Sexton (1992) and our version should be more efficient. This simplification is feasible because, in our case, the number to be excluded from storage is always zero.

For the binary tree scheme, the Sparse_vec is

```
type Sparse_vec = S_array Double
```

The following is the code for forward substitution:

```
forwd_subs :: Chl_factor -> S_Vector -> S_Vector
forwd_subs chl_factor b = s_listArray bnds y
  where
  diag = fst chl_factor      -- diagonal of Choleski factor
  off_diag = snd chl_factor  -- off diagonal of Choleski factor
  bnds = s_bounds b          -- vector bounds (1,n)
  l = fst bnds               -- l = 1
  y = (b!^l)/(diag!l) :      -- Y1
      [((b!^i)-inner_prod y (off_diag!i))/(diag!i) -- Yi
        | i<-range (bounds off_diag)               -- 2 <= i <= n
      ]
```

The expression for y is a straightforward implementation of equation (10). Notice that y is actually a list version of the solution vector. This extra copy of the solution vector allows us to avoid repeated updates to solution arrays (entry updates to the S_array are $O(\log n)$).

The inner_prod function multiplies two vectors of variable lengths (the longer vector is truncated) and has signature

```
inner_prod :: [Double] -> Sparse_vec -> Double
```

To avoid direct column accesses to Choleski factors, the function backwd_subs implements equations (12)–(14). The corresponding procedural operation can be described as

```
b := y                              -- equation (12)
for i := n to 1 step -1 do
begin
  new_x := b[i] / diag[i]           -- equation (14)
```

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

```
  x[i]  := new_x
  b     := b - new_x * off_diag[i] -- equation (13)
end
```

where * denotes a vector scaling operation and `off_diag[i]` represents a full Choleski factor row (expansion of row-segments filled out with zeros). The Haskell version, which is slightly less terse than the procedural description due to explicit initialisation of parameters and expression of recursion, is

```
backwd_subs :: Chl_factor -> S_Vector -> S_Vector
backwd_subs chl_factor y =
  fst  -- select only solution vector as the final result
  (foldl gen_x (init_x,y) [n,n-1..1])
                        -- successive updates of solution and RHS
  where
  init_x = s_array bnds []    -- initial (empty) solution vector
  diag = fst chl_factor       -- diagonal of Choleski factor
  off_diag = snd chl_factor   -- off diagonal of Choleski factor
  bnds = s_bounds y           -- vector bounds
  (1,n) = bnds                -- 1 = 1
  gen_x (x_vec,b) i =         -- one update step
    (
      x_vec//^[i:=new_x],     -- updated solution vector at i
      new_b                   -- updated RHS
    )
   where
   new_x = (b!^i) / (diag!i)
   new_b = s_accum (-) b (scale_vec new_x (off_diag!i))
```

Here, an extra solution list is not constructed because no vector-vector multiplication is involved. The `scale_vec` function basically scales a vector and has signature

```
scale_vec :: Double -> Sparse_vec -> [Assoc Int Double]
```

The implementations of the functions `inner_prod` and `scale_vec` are straightforward.

3.2.2 A generalised envelope scheme

A generalised envelope scheme due to Liu (1991) has also been implemented in Haskell. It involves the following four steps:

1. A minimum degree reordering is first applied to the initial system matrix to minimise the number of non-zero entries in Choleski factors.
2. A *postordering* of the system matrix follows to minimise storage overhead.
3. A Choleski decomposition is performed on the reordered matrix.
4. A generalised envelope partition is then applied to the Choleski factor so that it can be stored in a blockwise fashion to exploit all zero entries.

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

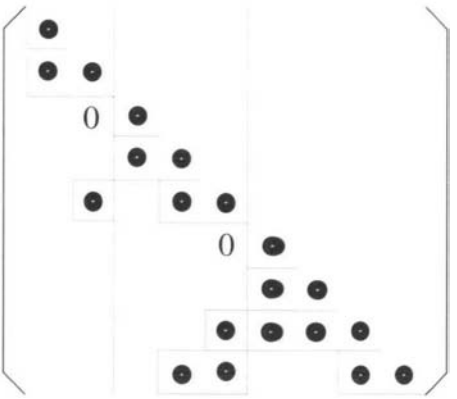

Fig. 2. An example of the generalised envelope partition.

This method divides a Choleski factor **L** into vertical blocks. A partition is made between columns i and $i+1$ if and only if $l_{i+1,i} = 0$, as illustrated in Figure 2. The • in Figure 2 denotes a non-zero entry. It can be shown that, within a vertical block, no zero entries will appear inside the block envelope. and each non-zero row-segment always ends at the last block column (Liu, 1991). Consequently, only • entries need be stored once the envelope structure of each block is identified. This is clearly ideal for a sparse representation. All we need to do is to identify the envelope structure of each block to eliminate zero entries from storage. The structure identification can be done by either establishing an elimination tree for **L** (Liu, 1986), or performing a symbolic factorisation.

In our Haskell program the scheme is implemented as

```
type GE_scheme = Array Int GE_block

type GE_block = (Diagonal, S_array Row_seg)

type Row_seg = (Int,[Double])
```

where `Diagonal` is as defined in the last section. In other words, a Choleski factor is represented as an array of blocks where each block is a tuple of a vector of matrix diagonal entries and a binary tree of matrix rows, which allows us to omit zero rows. Each non-zero row is then of type `Row_seg` where the `Int` component indicates the starting point of the row-segment. Figure 3 illustrates the data structure used to implement this scheme.

Again, tree structures are used for RHS vector and solution vector representation.

The Haskell implementation of this scheme has been based on that of the list of row-segments scheme. Because of the complication in representing a Choleski factor, the corresponding implementation of the forward/backward substitution steps is necessarily more complex than those for the binary tree and the list of row segments schemes. The actual implementation treats each block, which has more

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

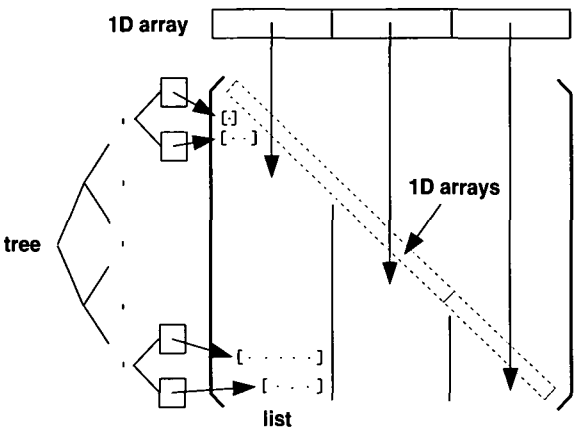

Fig. 3. Data structure for the generalised envelope method.

rows than columns, in the same way as a Choleski factor in the list of row-segments scheme applying the same forward/backward substitution strategy to each block. To accomplish the operation, a higher level recursion is introduced; each recursion sweep derives a segment of the result vector and an updated equation RHS.

In fact, the difference between this scheme and the list of row-segments scheme only lies in the way in which non-zero segments are organised (given that the same reordering schemes are applied). It carries a penalty of having to employ more complex implementations for the forward/backward substitution steps by the introduction of an extra level of data structures.

3.2.3 A quadtree scheme

The quadtree scheme (Wise, 1992) is a two-dimensional representation scheme. Wise (1992) discusses how the quadtree representation can be fitted into **LU** block-decomposition for a functional language, but as before here we are more interested in the performance of the forward/backward substitution steps on such a data structure in the current sparse matrix context.

A standard quadtree representation recursively partitions a matrix into four submatrices, dividing both matrix rows and columns. This scheme is thus not biased towards rows or columns. Quadruple trees are natural data structures for representing such partitions.

Because a Choleski factor **L** is lower triangular, our quadtree representation is different from that presented by Wise (1992) and Wainwright and Sexton (1992). Suppose we have

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_1 & \\ \mathbf{E} & \mathbf{L}_2. \end{bmatrix} \tag{15}$$

The following Haskell declaration is used to reflect the structure of a triangular matrix:

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

```
data TriMat a =
  TriM (TriMat a) (RectMat a) (TriMat a) -- L1 E L2
  | SingTM a
```

The first and third parameters following the constructor `TriM` accommodate the $\mathbf{L}_1$ and $\mathbf{L}_2$. The bottom-left rectangular submatrix is accommodated by the second parameter. The constructor `SingTM` stands for a one-entry matrix.

For the submatrix

$$\mathbf{E} = \begin{bmatrix} \mathbf{E}_{11} & \mathbf{E}_{12} \\ \mathbf{E}_{21} & \mathbf{E}_{22} \end{bmatrix},$$

a standard quadtree is used:

```
data RectMat a =
  RectM (RectMat a) (RectMat a) (RectMat a) (RectMat a)
                                              -- E11 E12 E21 E22
  | SingM a
  | ZeroM
```

The constructor `RectM` represents a matrix with four rectangular submatrices. `SingM` represents tree leaves (one entry matrices). An alternative definition for tree leaves is to use nodes representing four leaves instead of one, but this will not improve the complexity of the associated algorithms. `ZeroM` represents a zero matrix of any size, which can occur frequently in the quadtree splittings because we are dealing with matrices which normally have large blocks of zeros.

To show the implementation difference, the Haskell translation of the quadtree declaration used by Wainwright and Sexton is given below:

```
data QuadT = Quad QuadT QuadT QuadT QuadT | Diag num
```

They use the `Diag num` to represent a diagonal matrix with a unique diagonal value. Such a matrix structure is unlikely to appear in our current context where the `Diag` matrices would reduce to singletons and so our alternative approach using `SingM` has equivalent power. Our extra `ZeroM` constructor allows simpler pattern matching and expression of zero matrices than the equivalent expression `Diag 0`.

To make this scheme applicable to matrices of any size, a given matrix, $\mathbf{L}$, is first augmented to a size of $2^m \times 2^m$ by filling in 0's. The augmented matrix is than divided equally and recursively into quarters. The extra memory required for storing the augmented part is not significant because zero sub-matrices are automatically trimmed from quadtrees.

Figure 4 illustrates a quadtree matrix representation.

Wainwright and Sexton (1992) point out that operations explicitly isolating matrix rows represented by quadtrees should be avoided. To do this, we use a recursive forward/backward substitution implementation. To make the implementation feasible, we use binary trees to express vectors. In the following equations, the notation $\begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \end{bmatrix}$ is used to represent a vector with two subbranches and we explicitly use the `Bin_tree` representation for our `S_array` (defined in section 3.2.1).

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

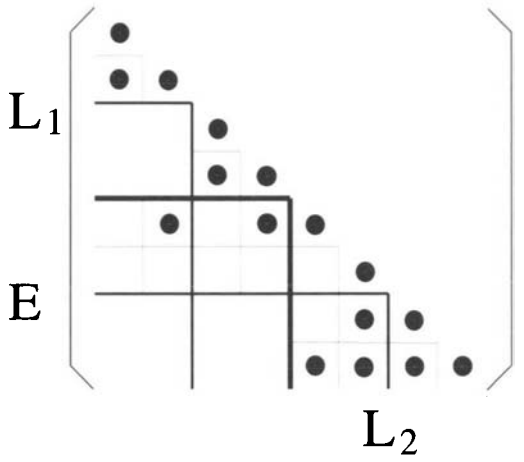

Fig. 4. Quadtree representation of a lower triangular sparse matrix.

Suppose the equation to be solved is $\mathbf{Ly} = \mathbf{b}$, or

$$\begin{bmatrix} \mathbf{L}_1 & \\ \mathbf{E} & \mathbf{L}_2 \end{bmatrix} \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix}.$$

A forward substitution calculating $\mathbf{y} = \mathbf{L}^{-1}\mathbf{b}$ can be achieved by first performing a forward substitution on $\mathbf{L}_1\mathbf{y}_1 = \mathbf{b}_1$, and then on $\mathbf{L}_2\mathbf{y}_2 = \mathbf{b}_2 - \mathbf{E}\mathbf{y}_1$ to get $\mathbf{y}_1 = \mathbf{L}_1^{-1}\mathbf{b}_1$, and $\mathbf{y}_2 = \mathbf{L}_2^{-1}(\mathbf{b}_2 - \mathbf{E}\mathbf{y}_1)$. This represents a recursive forward substitution algorithm:

```
forwd_subs :: (TriMat Double) -> (Bin_tree Double)
                                          -> (Bin_tree Double)
forwd_subs (TriM l1 e l2) (Fork s b1 b2) = -- forwd_subs L B
  Fork s y1 y2                            -- X
  where
  y1 = forwd_subs l1 b1
  y2 = forwd_subs l2 (subbv b2 (multmv e y1))
forwd_subs (SingTM v1) (Leaf v2) = Leaf (v2 / v1)
forwd_subs _ v@Null = v
```

The backward substitution of equation $\mathbf{L}^T\mathbf{x} = \mathbf{y}$, or

$$\begin{bmatrix} \mathbf{L}_1^T & \mathbf{E}^T \\ & \mathbf{L}_2^T \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix}$$

can be written in a similar manner using this quadtree recursive scheme. Notice that the quadtree data structure does not support a natural representation for the backward substitution algorithm outlined by equations (12)–(14) and therefore these equations are not adopted.

The essential operations involved in the above functions are clearly the multiplication $\mathbf{E}\mathbf{x}_1$, described by the function `multmv`, and the multiplication $\mathbf{E}^T\mathbf{x}_2$, described

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

by the function `multmtv`. The matrix-vector multiplication operation also appears in the Conjugate Gradient method. Wainwright and Sexton conclude that the quadtree representation is particularly suitable for this kind of operation (Wainwright and Sexton, 1992). Rewriting $\mathbf{E}\mathbf{x}_1$ as

$$\begin{bmatrix} \mathbf{E}_{11} & \mathbf{E}_{12} \\ \mathbf{E}_{21} & \mathbf{E}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{x}_{11} \\ \mathbf{x}_{12} \end{bmatrix} = \begin{bmatrix} \mathbf{E}_{11}\mathbf{x}_{11} + \mathbf{E}_{12}\mathbf{x}_{12} \\ \mathbf{E}_{21}\mathbf{x}_{11} + \mathbf{E}_{22}\mathbf{x}_{12} \end{bmatrix}$$

means $\mathbf{E}\mathbf{x}_1$ can also be recursively evaluated, and so too can $\mathbf{E}^T\mathbf{x}_2$. Unlike the other schemes, row-oriented operations and column-oriented operations (with respect to $\mathbf{E}$), appearing in the matrix-vector multiplications $\mathbf{E}\mathbf{x}_1$ and $\mathbf{E}^T\mathbf{x}_2$ respectively, can be expressed equally well using the same quadtree. The actual implementation of these operations in Haskell is straightforward.

Vector addition and subtraction are also involved here and they can be expressed in terms of sub-tree operations as

$$\begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \end{bmatrix} \pm \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1 \pm \mathbf{b}_1 \\ \mathbf{a}_2 \pm \mathbf{b}_2 \end{bmatrix}$$

The corresponding Haskell implementation is omitted.

3.2.4 *A Two-copy list of row-segments scheme*

Another possible way of achieving efficient backward substitutions is to use a column version of the factor matrix $\mathbf{L}$. From the one-copy representation schemes discussed above, the list of row-segments has been chosen as a basis for implementing such a two-copy scheme; storing the other copy of the $\mathbf{L}$ factor as an array of lists of column segments. The binary tree scheme is not seen to be appropriate for building a two-copy scheme. It is important for a two-copy scheme to support efficient vector-vector multiplications, optimal time complexity being $O(n)$. However, tree structures have an associated complexity of $O(n \log n)$ because of the $O(\log n)$ access time. The generalised envelope scheme uses the same fundamental data structure as in the list of row-segments scheme. There is no need to introduce a two-copy implementation for the quadtree scheme as this by nature is a two-dimensional approach.

The implementation of the two-copy scheme is straightforward. The forward substitution implementation is adopted without change. The backward substitution implementation is very similar to the forward substitution implementation except that the construction of the auxiliary solution list starts from its tail (calculating x_n first) rather than its head.

3.2.5 *Use of Glasgow mutable arrays*

As an extension to the two-copy list of row-segments scheme, another set of experiments has been conducted to test the use of Glasgow mutable arrays. Here, Glasgow mutable arrays instead of trees (`S_array`) have been used to represent RHS and solution vectors. The use of Glasgow mutable arrays for representing Choleski factors is not necessary because these are constant factors and do not therefore require

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

updates. The forward/backward substitution implementations have been changed to take full advantages of the mutable arrays.

One obvious advantage is that such an implementation can at least reuse the RHS vector space for storing the solution vector. A practical implementation has revealed two further advantages of employing mutable arrays that use in-place updates. First, the auxiliary solution lists used in the two-copy list of row-segments scheme are no longer necessary here, saving further space. Second, letting forward/backward substitutions share a common substitution implementation becomes very natural.

To use the Glasgow mutable arrays, it is necessary to include monads (Peyton Jones and Wadler, 1993) in the code to allow state-based computations. The basic functions used are `thenStrictlyST`, `seqStrictlyST` and `returnStrictlyST`. Functions `thenStrictlyST` and `seqStrictlyST` expect two arguments. Function `thenStrictlyST` first evaluates its first argument, yielding a result, and then applies its second function argument to the result. Function `seqStrictlyST` behaves similarly except the result of its first argument is always ignored. Function `returnStrictlyST` does nothing except packages its only argument with a state. These functions have their lazy counterparts, namely, `thenST`, `seqST` and `returnST`, but we prefer the eager ones to avoid space leakages (Peyton Jones *et al.*, 1994). The first two functions sequence the evaluations of their arguments. The third returns a result. An extra function `processListST` is defined on top of these functions to simplify the implementation; it sequences the evaluations of the entries of a list.

```
processListST [] = returnStrictlyST []
processListST (x:xs) =
  x `thenStrictlyST` (\r ->
  processListST xs `thenStrictlyST` (\rs ->
  returnStrictlyST (r:rs)))
```

The following is the substitution code implementing both equations (10) and (11):

```
gen_x list mat x =
  -- list: specifying the order of entry computation order
  -- mat:  copy of Choleski factor to be used
  -- x:    RHS
  processListST [
    readDoubleArray x i `thenStrictlyST` (\v ->
                                          -- read Xi (RHS) into v
    if i==first then  -- save the first solution entry
      writeDoubleArray x i (v/(diag!i))
    else              -- save other solution entries
      mult_x (off_diag!i) `thenStrictlyST` (\prods ->
      writeDoubleArray x i ((v-sum (concat prods))/(diag!i)))
      )
      | i <- list  -- go through all vector entries
  ]
  where
```

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

```
first = head list
        -- first=1 or n depends on the substitution required
diag = fst mat      -- diagonal of Choleski factor
off_diag = snd mat  -- off diagonal of Choleski factor
mult_x row =        -- vector-entry multiplication for a row
  processListST [
    processListST [
      readDoubleArray x i `thenStrictlyST` (\xv ->
                    -- read Xi into xv
      returnStrictlyST (xv * v))
                    -- return the product of vector entries
      | (i,v) <- zip [j..] as
                    -- go through all entries in a row segment
    ]
    | (j,as) <- row  -- go through all row segments
  ]
```

The actual forward/backward substitutions are simply

```
forwd_subs mat b = gen_x (range (boundsOfByteArray b)) mat b

backwd_subs mat b = gen_x [n,n-1..1] mat b
  where
  (1,n) = boundsOfByteArray b
```

specifying different computation sequences and they are called in

```
chl_method (chl_fac,chl_fac_t) b scalar =
  forwd_subs chl_fac b `seqStrictlyST`
  backwd_subs chl_fac_t b `seqStrictlyST`
  processListST [
    readDoubleArray b i `thenStrictlyST` (\v ->
    writeDoubleArray b i (v*scalar))
    | i <- range (boundsOfByteArray b)
  ]
```

where `chl_fac` and `chl_fac_t` represent the copies of the Choleski factor, and `boundsOfByteArray` returns the bounds of a Glasgow mutable array.

Compared with its predecessor, this implementation is less clear due to the presence of monads but less complex due to the omission of solution lists.

4 Results

4.1 Test problems and test schemes

Four test problems, listed in Table 2, have been adopted. These problems are small in engineering terms but realistic and their details are discussed in references cited in Table 2. In practice, it is not uncommon to see problems with more than 10^4

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

Table 2. *Four test problems*

Test problem	Geometry	Size of **A**	Non-zeros in **L**	sparsity (%)	Reference
1	cavity	121 × 121	819	11.1	(Hawken *et al.*, 1990)
2	glass oven	209 × 209	1443	6.6	(Ding *et al.*, 1993)
3	cavity	441 × 441	4239	4.3	(Hawken *et al.*, 1990)
4	L-shape	854 × 854	5946	1.6	(Al-Hussany *et al.*, 1992)

Table 3. *Summary of test schemes*

No.	Abbreviation	Scheme	Mat. structure	Mat. represented	Vec. structure
1	non-sp	non-sparse	array of lists	**L**	tree
2	bin-tree	binary tree	array of trees	**L**	tree
3	env	gen. envelope	array of trees of lists	**L**	tree
4	row-seg	row-segment	array of lists of lists	**L**	tree
5	q-tree	quadtree	quadtree	**L**	tree
6	row-seg2	row-segment	array of lists of lists	**L** and $\mathbf{L}^T$	tree
7	row-seg2-G	row-segment	array of lists of lists	**L** and $\mathbf{L}^T$	Glasgow mutable array

mesh elements, resulting in system matrices of equivalent sizes. In Table 2, sparsity is a percentage measure of non-zero entries in **L** factors.

The platform for the reported experiments was a Sun Sparc-1 workstation with 40Mb RAM. Except for the `row-seg2-G` scheme which has to employ the Glasgow 0.21 Haskell compiler to use mutable arrays, the `hbc 0.999.4` compiler supplied by Chalmers University has been adopted.

Two sets of tests have been performed to determine the space and time efficiencies of these test schemes. To get accurate estimates of the differences due to adopting the various schemes, the computation of system RHSs, system matrices, and Choleski decompositions are excluded from the statistics. In a realistic solution process, a system RHS must be assembled at run time and this may involve a lot of computation. Experiments show a typical RHS assembly for equation (5) takes twice as much time as the forward/backward substitutions. A typical system matrix assembly, which is performed only once, takes a hundred times as much. The time spent on a Choleski decomposition is equivalent to that on forward/backward substitutions (notice the equations for a Choleski decomposition are very similar to those for forward/backward substitutions).

Seven test schemes are examined in this paper, and are summarised in Table 3. Test scheme `non-sp` is non-sparse and is included for comparison.

The average complexities of the data structure traversals involved in the forward/backward substitution operations for all schemes are presented in Table 4. Numerical calculations are discounted as they are identical for the sparse schemes except for the `bin-tree` scheme. The `bin-tree` scheme does not explicitly exclude zero

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

Table 4. *Average data structure traversal complexities*

Scheme abbreviation	Forward substitution	Backward substitution
bin-tree	$(d_1 + n + 1 + \log n)n$	$(d_2 + ((s+1)n + 1)\log n)n$
env	$((s+1)n + 1 + (bn+1)\log n)n$	$(sn + ((s+b+1)n + 1)\log n)n$
row-seg	$((s+1)n + 1 + \log n)n$	$(sn + ((s+1)n + 1)\log n)n$
q-tree	n_q	n_q
row-seg2	$((s+1)n + 1 + \log n)n$	$((s+1)n + 1 + \log n)n$
row-seg2-G	$(sn+1)n$	$(sn+1)n$

entries. Consequently, the forward substitution computation in the `bin-tree` scheme has been implemented to perform numerical computations even if a zero entry is encountered. We exclude the `non-sp` scheme from the data structure traversal analysis as comparison with other sparse schemes is not meaningful; it performs the least amount of data structure traversals but performs the most numerical computations.

The forward and backward substitutions are segregated in the table since their implementations differ. s represents the matrix sparsity defined at the beginning of this section. It is assumed that the number of resultant block partitions in the generalised envelope scheme is proportional to the vector size n and characterised by the coefficient b (bn being the number of generalised envelope blocks). Experiments show b is of order 0.1. n_q denotes the number of terminal nodes in a truncated quadtree and should be viewed as a function of n^2.

A major difficulty involved in the complexity analysis is the parameterisation of traversing Choleski factor rows when they are represented by pruned trees, and there are sn non-zero entries in each row. This means at least $sn \log n$ traversals must be performed to go through all branches in a tree. Additionally, we also have the cost of locating the roots of removed subtrees. The number and the location of these roots are dependent on the sparsity pattern of the matrix rows. Generally, the pattern of row sparsity is dependent on how finite element mesh nodes are labelled, or equivalently how system matrices are ordered. Such orderings are normally used to produce certain optimal (or near optimal) matrix structures, resulting in different sparsity patterns.

Let n_r denote the total number of null subtree roots and m_i the location depth of the ith such root. Therefore, $2^{-m_i}n$ is the total number of zero leaves in the subtree removed at node i. In the implementation of the forward substitution, a subtree root i is visited $2^{-m_i}n$ times to retrieve the value for each subtree leaf. In the implementation of the backward substitution, due to the nature of the involved computation (vector scaling), the equivalent operation can be done more efficiently and each root is only visited once. Thus, the total numbers of traversals required to go through a sparse tree for the forward and backward substitutions, d_1 and d_2 respectively, are

$$d_1 = sn \log n + n \sum_{i=1}^{n_r} 2^{-m_i} m_i \tag{16}$$

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

Table 5. *Separated complexities for one-copy schemes 2–4*

Scheme abbreviation	Forward substitution			Backward substitution			
	b	$\sum l_{ip}y_p$	**y**	$\mathbf{z}^{(j-1)}$	$\mathbf{z}^{(j)}$	$\{l_{pi}x_p\}$	**x**
bin-tree	$n\log n$	$(d_1+n)n$	n	$n^2\log n$	$sn^2\log n$	$d_2 n$	$n\log n$
env	$n\log n$	$(s+1+b\log n)n^2$	n	$n^2\log n$	$sn^2\log n$	$(s+b\log n)n^2$	$n\log n$
row-seg	$n\log n$	$(s+1)n^2$	n	$n^2\log n$	$sn^2\log n$	sn^2	$n\log n$

$$d_2 = sn\log n + \sum_{i=1}^{n_r} m_i. \tag{17}$$

If we assume non-zero entries are evenly distributed and all subtrees have the same height

$$h = \log \frac{(1-s)n}{n_r},$$

$(1-s)n$ being the total number of non-zero entries in the tree, then, all

$$m_i = \log n - h = \log \frac{n_r}{1-s}.$$

In this case (16) and (17) simplify to

$$d_1 = sn\log n + (1-s)n\log \frac{n_r}{1-s} \tag{18}$$

$$d_2 = sn\log n + n_r \log \frac{n_r}{1-s}. \tag{19}$$

We now briefly explain our complexity analysis for each scheme. Schemes `bin-tree`, `env`, and `row-seg` use equation (10) for the forward substitution and equations (12)–(14) for the backward substitution. The main operations involved in equation (10) include accessing b_i, vector-vector multiplications $\sum l_{ip}y_p$, and the construction of **y** vector (the rest have constant complexities). The equations (12)–(14), for a specific j, have these main operations: access of $\mathbf{z}^{(j-1)}$ entries, update of $\mathbf{z}^{(j)}$ tree branches (excluding update of tree leaves), computation of $\{l_{p,i}x_p | i = 1, 2, ..., n-j\}$ (update of tree leaves), and construction of **x**. All schemes store **b**, **x** and $\mathbf{z}^{(j)}$ as trees. Schemes `bin-tree`, `env` and `row-seg` initially represent vector **y** as a list before it is converted into a tree at the final stage to reduce the overall complexity.

The individual operation complexities of these schemes are shown in Table 5 and explained below.

`bin-tree` : This scheme represents matrix rows by truncated trees. This affects operations $\sum l_{ip}y_p$, $\mathbf{z}^{(j)}$ and $\{l_{pi}x_p\}$. The complexity associated with $\sum l_{ip}y_p$ is $O((d_1+n)n)$ because the cost of traversing l_i and **y** are d_1 and n respectively. The complexity associated with $\{l_{pi}x_p\}$ is similar but **x** is not traversed. For each j, there are now only sn non-zero $\mathbf{z}^{(j)}$ entries that need to be updated and each update traverses $\log n$ tree branches.

`env` : In this case, matrix row storage is list based. Compared with the `bin-tree` case, only $\sum l_{ip}y_p$ and $\{l_{pi}x_p\}$ entries in table 5 need to be modified. For both

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

the forward and backward substitutions, accessing all non-zero entries in l_i for a specific i requires sn traversals plus an overhead of $bn \log n$ operations (bn being the assumed number of generalised envelope blocks) to locate all row-segment heads in different blocks (see figure 3). For the forward substitution, an extra n term must be added to account for the traversal of vector **y**.

`row-seg` : This can be viewed as a special case of the `env` scheme where the cost for locating the heads of row-segments is a constant.

The quadtree scheme uses a recursive substitution algorithm (see in section 3.2.3 for detail) which operates on trees for both forward and backward substitutions. The algorithm falls into a classical category whose associated complexity has the same order as the number of leaves in the trees, n_q in this case.

For `row-seg2` and `row-seg2-G`, because of the use of the column version of Choleski factors and the associated backward substitution algorithm, the complexities for forward and backward substitutions become identical, and hence we only analyse the forward substitution case here. For `row-seg2`, complexities are the same as that for `row-seg` forward substitution. For `row-seg2-G`, the complexities for accessing all b_i and constructing **y** are $O(n)$. The complexity for computing $\sum l_{ip} y_p$ is $O(sn^2)$. These are reflected in Table 4.

According to Table 4, theoretical comparisons of these scheme can be made. Among the schemes other than the `q-tree`, the `row-seg2-G` is obviously the best on time efficiency, followed by the `row-seg2` which used a better backward substitution algorithm. The `row-seg` scheme is better than the `env` scheme because of the non-existence of the bn entry in its complexities. Since from (16) and (17) d_1 and d_2 are always larger than sn, the `row-seg` is better than the `bin-tree` scheme.

It is, however, somewhat difficult to compare the `bin-tree` and `env` schemes at this stage, but the fact that the `bin-tree` scheme involves more numerical computation, as explained earlier, places it at a disadvantage. It is also difficult to compare the `q-tree` scheme with the others due to the complete difference in the adopted substitution algorithms. Furthermore, the parameter n_q, the number of quadtree leaves, not only takes into account all non-zero entries but also some zero entries in the system matrix.

One comment on the experiments is that results presented in Tables 6 and 7 for the `row-seg2-G` scheme are not directly comparable with the remainder as the compilers used are different (Glasgow compiler for `row-seg2-G` and `hbc` for the rest). This is reflected in both space and time profiling, but does not prevent us from drawing some general conclusions. Some simple tests show that the `hbc` immutable arrays are slower than Glasgow mutable arrays, but faster than Glasgow immutable arrays, as far as data retrieval is concerned. As for entry update, Glasgow mutable arrays are much more efficient than the others on both time and space accounts (one order of magnitude for time efficiency).

4.2 Space efficiency

Here statistics of maximum space consumption are gathered from ten executions of the algorithms by using two heap profilers that come with the two Haskell compilers

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

Table 6. *Comparison of maximum space consumption (Kb)*

Test scheme	non-sp	bin-tree	env	row-seg	q-tree	row-seg2	row-seg2-G
Test problem 1	331.1	156.5	136.0	109.3	114.5	51.3	60.5
Test problem 2	847.0	283.8	259.7	196.7	206.1	92.3	103.4
Test problem 3	3038.6	707.3	597.7	463.2	610.4	244.6	287.3
Test problem 4	-	-	1124	639.2	966.9	380.0	416.3

employed (Runciman and Wakeling, 1993; Sansom and Peyton Jones, 1992). The Glasgow profiler is only used for the `row-seg2-G` scheme. These results include both space used for storing static Choleski factors and that used for generating intermediate results. Space used by system modules is not taken into account. Table 6 summarises the gathered statistics, where blank entries occur due to heap space exhaustion during the assembly of system matrices.

Several observations can be drawn from the table:

- All sparse schemes are capable of saving more than 50% space resource over non-sparse implementations, and the larger the problem, the more these sparse schemes can save. The two-copy representations are better space saving schemes.
- It is somewhat surprising to see that the `row-seg` scheme uses about twice as much space as the `row-seg2` scheme, with similar findings for other one-copy schemes 2 and 3. Close examinations of some profile reports show that most space is used by the `S_arrays`, and accordingly it is the repeated lazy updates of RHSs in the backward substitution step that cause this overhead. Two-copy schemes can avoid this problem because the alternative backward substitution implementation, equation (11), can be used efficiently.
- The two-dimensional quadtree representation, `q-tree`, performs poorly on space consumption. The cause is attributed to the lazy evaluation of many submatrix-subvector multiplications (similar to the `row-seg` case).
- The `row-seg2` scheme is consistently better than the `row-seg2-G` scheme. However, a close study of relevant profiles reveals that the Glasgow Haskell compiler reports more space usage for the static copies of the Choleski factors. For example, for `problem 4`, the Glasgow compiler space usage by the matrix is $398.3Kb$ whilst the `hbc` usage is $255.7Kb$. This means the `row-seg2-G` actually uses less space, excluding the static Choleski factors, a saving attributable to in-place updates.

4.3 Time efficiency

The average timings over ten executions for the algorithms are presented in Table 7, using the UNIX `time` command. These tests have been performed to measure the relative speeds. Again the executable for the `row-seg2-G` has been generated by the the Glasgow compiler using mutable arrays and the rest have been that of `hbc` (using

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

Table 7. *Comparison of time efficiency (*sec *per solution)*

Test scheme	non-sp	bin-tree	env	row-seg	q-tree	row-seg2	row-seg2-G
Test problem 1	5.2	1.3	0.9	0.9	0.2	0.3	0.2
Test problem 2	16.9	2.9	1.7	1.7	0.4	0.6	0.4
Test problem 3	115.8	9.8	4.7	5.1	1.8	1.6	1.2
Test problem 4	-	-	8.1	8.4	5.3	3.5	1.7

immutable arrays). The Glasgow code is somewhat slower in general even though its mutable arrays provide faster array operations (especially for entry update which is one hundred times faster than that provided by `hbc` immutable arrays). However, it is difficult to estimate the overall influence on speed brought about by the Glasgow mutable arrays.

The following observations can be made:

- Most sparse representations reduce the execution time by at least 80% compared with the non-sparse representation. The `row-seg2-G` scheme, which reduces the execution time by up to 99%, is the most efficient scheme. Even the `bin-tree` scheme which involves much more numerical computation than the other sparse schemes is much faster than the non-sp scheme, indicating the importance of space saving.
- Two-copy schemes are better than the one-copy schemes 2–4, due to the differences in their adopted backward substitution implementations.
- The `q-tree` and `row-seg2` schemes have performance similar to that of the `row-seg2-G` scheme for the first three moderate sized test problems. However the `row-seg2-G` scheme displays a clear performance advantage, namely, less retardation as problem size increases further.
- The worse schemes are those using the slower backward substitution algorithm which updates RHSs (as in equation (12)–(14)) introducing a $\log n$ factor into the complexity. These include the `non-sp`, `bin-tree`, `env` and `row-seg` schemes.

5 Conclusions and remarks

In this paper, we have examined several sparse matrix representation schemes in relation to Choleski forward/backward substitution operations. Sparse matrix representation is very important for solving large numerical engineering problems, as it is simply impractical to store full matrices for very large problems. The advantage of using sparse representations has been clearly demonstrated by our experiments.

Although the Choleski forward/backward processes are mathematically straightforward, our investigation found many intricacies in their implementation. The main difficulties occur on account of the need for both row-oriented operations and column-oriented operations. This places certain representation schemes at a

https://doi.org/10.1017/S095679680000160X Published online by Cambridge University Press

disadvantage. Various strategies have been developed to avoid inefficient operations, such as random access to row entries and entry updates to immutable functional arrays. Starting from a straightforward implementation of the forward/backward substitution equations (10) and (11) for one-copy schemes 2–4, an auxiliary solution list was introduced to avoid repeated updates to the solution vector. This resulted in a time improvement. An alternative algorithm for backward substitution was then adopted to avoid random access to row entries but this led to repeated updates to system RHSs, introducing a $\log n$ factor instead of n as for its predecessor, into the data structure traversal complexity. The need to investigate two-dimensional and two-copy schemes became obvious. An original concern over the two-copy schemes was that they may be space demanding, but it turns out that they were better both in terms of time and space saving. The emergence of the Glasgow mutable arrays opens a new avenue for the solution of these problems and invites further study. Initial tests show that this facility eliminates the need for auxiliary solution lists and leads to better time and space efficiencies. Our main concern here is the practical difficulty of developing codes utilising monad states, where it is often difficult for us to comprehend error messages and to locate errors. The awkward syntax, that is somewhat procedural, is another disadvantage.

Although the quadtree scheme is time efficient over other one-copy schemes, possibly assisted by an advantageous recursive algorithm, it is space demanding (though eager evaluation may help here). Wainwright and Sexton (1992) state that the quadtree is suitable for expressing matrix-vector multiplications. We have shown that it is also appropriate for expressing both row-oriented and column-oriented operations on a common representation.

Based on our experience, we make the following general comments on functional programming:

Expressiveness: Functional programming is more expressive than procedural programming. It produces more concise code. The essential part of our complete implementation of the Taylor-Galerkin/pressure-correction algorithm is less than 2000 lines (for 2D problems only) which favourably compares with our Fortran implementation of about 7000 lines.

Debugging: It has been our experience that most errors (at least 80%) are detected at the compilation stage.

Coding effort: Becoming familiar with functional programming is not easy for a programmer only experienced in procedural programming. This increased the difficulty of constructing an implementation.

Modification: Once the implementation has been constructed, modifying a functional program is much easier than a procedural program. Particularly when the pattern matching technique is used, changing some complex data structures can be easily realised by changing the parameter patterns.

Parallelisation: We have had some experience in parallelising our functional implementation which has demonstrated that functional programming is a more appropriate vehicle for parallel programming than procedural programming (Grant *et al.*, 1993a; Grant *et al.*, 1994).

Arrays: With immutable arrays, the desirable operation of updating array entries in-place is unavailable. The Glasgow monadic mutable array does eliminate some problems in functional programming but also introduces extra programming difficulties.

Efficiency: A complete Haskell sequential implementation of the Taylor-Galerkin/pressure-correction algorithm (using the `hbc` compiler) is an order of magnitude slower than an equivalent Fortran implementation (Grant *et al.*, 1993b). Compared with procedural programs, functional programs generally need more space. The extra space is normally consumed by lazy evaluations, immutable array update and auxiliary data storage for efficient functional implementations. By using better implementations in the future, this extra cost may be reduced. In this investigation, storing two copies of a Choleski factor for `problem 4` and using the `hbc` compiler takes 256 Kb space. This is roughly 2.5 times as much as required by procedural implementations, and likely to be due to the fact that values are boxed. The unboxed Glasgow mutable arrays for vector representation are no more space demanding than procedural implementations.

Although currently the performance of functional programs is still falling behind that of equivalent procedural programs for numerical computation, the recent development in functional programming research has shortened the discrepancy a great deal and made the application of this technology to non-trivial engineering problems possible. It is particularly encouraging to see that, in this investigation, the use of mutable functional arrays makes our functional program space competitive as space consumption has been a major obstacle for many large applications.

Acknowledgements

This research has been supported by grants from the UK Science and Engineering Research Council and the Department of Trade and Industry, UK (SERC GR/F 99076 C2117 and SERC GR/J12321). The authors are grateful for the detailed comments and constructive advice from anonymous referees which has led to many improvements in this paper.

References

Al-Hussany, A. F. H., Townsend, P. and Webster, M. F. (1993) Computer simulation of temperature build-up in the radial filling of disc-shaped cavities. In *Proc. Computational Mechanics in UK*, pp. 49–54. Swansea, UK, January 1993. Association for Computational Mechanics in Engineering, UK.

Augustsson, L. (1993) *Haskell B user's manual.* Department of Computer Sciences, Chalmers University of Technology, Sweden, October.

Cuvelier, C., Segal, A. and van Steenhoven, A. A. (1986) *Finite Element Methods and Navier-Stokes Equations.* Reidel Publishing Co.

Davie, A. J. T. (1992) *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press.

Ding, D., Townsend, P. and Webster, M. F. (1993) Computer modelling of transient thermal flows of non-Newtonian fluids. *J. Non-Newtonian Fluid Mechanism*, **47**: 239–56.

Duff, I. S., Erisman, A. M. and Reid, J. K. (1986) *Direct Methods for Sparse Matrices.* Clarendon Press.

Grant, P. W., Sharp, J. A., Webster, M. F. and Zhang, X. (1992) A Haskell implementation of a Generalized Envelope method for sparse matrix factorization. In *Proc. ATABLE-92*, pp. 247–260. Montreal, Canada, June.

Grant, P. W., Sharp, J. A., Webster, M. F. and Zhang, X. (1993a) Functional programming for a computational fluid dynamics problem. In *Proc. Computational Mechanics in UK*, pp. 75–79. Swansea, UK, January. Association for Computational Mechanics in Engineering, UK.

Grant, P. W., Sharp, J. A., Webster, M. F. and Zhang, X. (1993b) Some issues in a functional implementation of a finite element algorithm. In *Proc. FPCA '93*, pp. 12–17. Copenhagen, Denmark, June. ACM SIGPLAN/SIGARCH.

Grant, P. W., Sharp, J. A., Webster, M. F. and Zhang, X. (1994) Experiences of parallelising finite element problems in a functional style. *Software – Practice and Experience*, to appear.

Hageman, L. A. and Young, D. M. (1981) *Applied Iterative Methods.* Academic Press.

Hawken, D. M., Tamaddon-Jahromi, H. R., Townsend, P. and Webster, M. F. (1990) A Taylor-Galerkin-based algorithm for viscous incompressible flow. *Int. J. Num. Meth. Fluids*, **10**: 327–351.

Hudak, P., Peyton Jones, S. L. and Wadler, P., editors (1992) Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *SIGPLAN Notices*, May.

Liu, J. W. H. (1986) A compact row storage scheme for Choleski factors using elimination trees. *ACM Trans. Math. Software*, **12**: 127–148, June.

Liu, J. W. H. (1991) A generalized envelope method for sparse factorization by rows. *ACM Trans. Math. Software*, **17**: 112–129, March.

Peyton Jones, S. L., Launchbury, J. and Partain, W. (1994) GHC prelude: Types and operations. *Technical report*, Computing Science Department, Glasgow University.

Peyton Jones, S. L. and Wadler, P. (1993) Imperative functional programming. In *ACM Symposium on Principles of Programming Languages*, pp. 71–84. Charleston, SC, January.

Runciman, C. and Wakeling, D. (1993) Heap profiling for lazy functional languages. *J. Functional Programming*, **3**(2): 217–245, April.

Sansom, P. M. and Peyton Jones, S. L. (1992) Profiling lazy functional programs. In *Proc. Functional Programming, Glasgow.* Workshops in Computing. Springer-Verlag.

Spivey, M. (1990) A functional theory of exceptions. *Science of Computer Programming*, **14**(1): 25–42, June.

Townsend, P. and Webster, M. F. (1987) An algorithm for the three-dimensional transient simulation of non-Newtonian fluid flows. In G. N. Pande and J. Middleton, editors, *Proc. Int. Conf. Num. Meth. Eng.: Theory and Applications*, **II**, pp. T12/1–11. Swansea, UK. NUMETA, Nijhoff, Dordrecht.

Wainwright, R. L. and Sexton, M. E. (1992) A study of sparse matrix representations for solving linear systems in a functional language. *J. Functional Programming*, **2**(1): 61–72, January.

Wilkinson, J. H. and Reinsch, C. (1971) *Handbook for Automatic Computation, Linear Algebra, vol. II.* Springer-Verlag.

Wise, D. S. (1992) Matrix algorithms using quadtrees (invited talk). In *Proc. ATABLE-92*, pp. 11–26. Montreal, Canada, June.

Zhang, X., Webster, M. F., Sharp, J. A. and Grant, P. W. (1994a) Computational fluid dynamics application. In C. Runciman and D. Wakeling, editors, *Applied Functional Programming*, chap. 9. UCL Press.

Zhang, X., Webster, M. F., Sharp, J. A. and Grant, P. W. (1994b) Parallelisation for computational fluid dynamics. In C. Runciman and D. Wakeling, editors, *Applied Functional Programming*, chap. 12. UCL Press.