Functional Programming for Finite Element Analysis

Junxian Liu Paul Kelly Stuart Cox

Department of Computing, Imperial College

Abstract

This report is a detailed study of the application of functional programming to a typical problem in engineering computing—finite element structural analysis. The report assumes no prior knowledge of structural analysis, but does presume some familiarity with the Miranda programming language. It begins by showing how a basic linear algebra (BLAS) library can be built in Miranda, to perform elementary operations on vectors and matrices of various kinds. This is then applied to the construction of a simple finite element analysis package. Performance results under various functional language implementations are presented, and several different formulations of the central computations are investigated. Finally opportunities for parallel computation are identified.

1 Introduction

This paper studies the application of functional programming in a fairly realistic and extended case study. We explore some of the claims, strengths and weaknesses of the functional approach, in terms of clarity of exposition, efficiency of execution, and in the search for efficient parallel formulations of the problem. Details of the numerical algorithms are included in an attempt to counter the widespread ignorance among the programming languages community of fundamental computational structures.

We begin by rehearsing some of the claims made for functional programming.

1.1 The functional approach

Although structured and object-oriented programming have helped control the complexity, programming has retained a word-at-a-time style: the program describes a sequence of state transitions to achieve the desired effect, rather than a description of what final result is required. This view of programming as scheduling a sequence of assignments was first described by Backus [1], and highlights the overspecification conventional languages require in the area of control flow and memory reallocation.

A functional language has no assignment operation. Whenever a new value is created, it is given a new name: the programmer need not, and cannot, express that a memory cell should be reused. Instead responsibility for storage reallocation is left entirely to the language implementation. This has the effect that the order of execution of the program is constrained in principle only by the data dependence between values.

Higher order functions and lazy evaluation, particularly lazy evaluation of lists ("streams"), are perhaps the most important features of functional programming, and compensate to some

extent for the lack of a general assignment operation. Together they allow control structures to be separated from the details of particular applications, providing very powerful support for modular programming and software reuse [7].

The value of lazy evaluation is demonstrated well by several examples later in this paper, where a data structure can be defined without specifying the order of evaluation of its components. This frees the programmer from specifying a particular evaluation order, and gives the implementation a free choice limited only by the data dependencies.

Functional languages are of interest in parallel computing because of the free choice of evaluation order—that is overspecification of control flow is avoided, and because higher-order functions capture easily-identified sources of parallelism.

Finally, we note that a modern functional language, like the ones used in this paper, have other important features which add to the clarity and convenience of the notation, but are not inherently confined to functional or declarative languages. Features found to be important here include user-defined algebraic data types, pattern matching, and strong type checking together with a notion of polymorphism which avoids unnecessary type restrictions on functions which act independently of their parameter's type.

1.2 Overview

In this report, we will study how to apply functional programming to finite element structural analysis, and investigate the advantages, disadvantages or problems found in applying functional programming in engineering computation with specific reference to the claims outlined above.

Section 2 gives a brief description of finite element method for structural analysis. Section 3 describes a basic linear algebra system developed in Miranda[2].[1] This BLAS system contains the abstract data types and linear equation system solvers involved in finite element analysis. In section 4, a finite element structural analysis program in Miranda is presented. The efficiency of the Miranda version of the program, a *Hope*$^+$ [11] version and a C version are discussed for several examples in section 5. Later, sections 6 and 7 give two different ways of parallelizing finite element structural analysis computation by using functional programming. We analyse our conclusions from the study in section 8.

2 Finite Element Structural Analysis

Structural analysis is concerned with the determination of stresses and displacements in a structure for a given loading so that the designer can provide sufficient resistance to resist the stresses developed. For example, when designing a bridge, the designer needs to know the stresses and displacements of the bridge while vehicles are passing through. If the displacement is too large and/or the stress is too big, the initial design has to be modified by increasing the amount of material used, restricting heavy vehicles, or changing the initial design completely.

The finite element method is a general numerical technique for solving problems in structural analysis and many other areas of physical science[5]. In the finite element method, a structure is discretised into a series of elements, each of which has its own characteristics. These characteristics are described by the element stiffness matrix which represents the

[1] Miranda is a trademark of Research Software Ltd.

relationship between its internal forces and displacements:

$$K^e \; u^e = f^e \tag{1}$$

Where, K^e is the element stiffness matrix; u^e is the displacement vector; and f^e is the internal force vector.

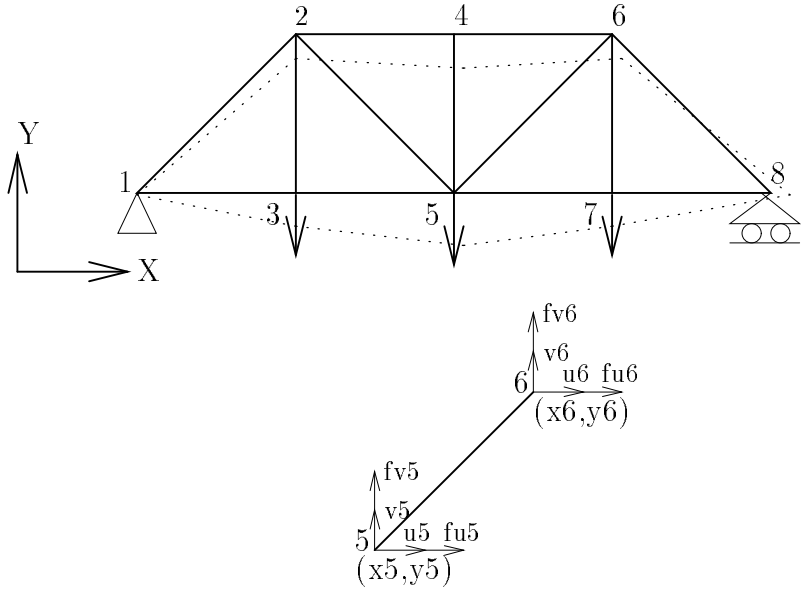


Figure 1: 2D pin jointed bar element

For a two dimensional pin jointed structure—that is, when the joints allow free rotation—the displacements of a node can only be in two directions. Equation (2) shows the case for the two dimensional pin jointed bar element as shown in Figure 1:

$$\begin{bmatrix} k^e_{u5\ u5} & k^e_{u5\ v5} & k^e_{u5\ u6} & k^e_{u5\ v6} \\ k^e_{v5\ u5} & k^e_{v5\ v5} & k^e_{v5\ u6} & k^e_{v5\ v6} \\ k^e_{u6\ u5} & k^e_{u6\ v5} & k^e_{u6\ u6} & k^e_{u6\ v6} \\ k^e_{v6\ u5} & k^e_{v6\ v5} & k^e_{v6\ u6} & k^e_{v6\ v6} \end{bmatrix} \begin{Bmatrix} u_5 \\ v_5 \\ u_6 \\ v_6 \end{Bmatrix} = \begin{Bmatrix} f^e_{u5} \\ f^e_{v5} \\ f^e_{u6} \\ f^e_{v6} \end{Bmatrix} \tag{2}$$

Where, u_5 denotes the displacement along X axis at node 5; f^e_{u5} denotes the internal force along X axis at node 5; $k^e_{u5\ v6}$ denotes the stiffness corresponding to displacements u_5 and v_6; and so on.

The stiffness matrix of an element depends only on the material used to build the structure and the geometrical shape and location of the element. So, to represent a 2D bar element one only needs to describe the coordinates of the two ends and the material used.

Assembling all these elements together, we will get following linear equation system:

$$K \; u = f \tag{3}$$

i.e.

$$\begin{bmatrix} k_{11} & k_{12} & \dots & k_{1n} \\ k_{21} & k_{22} & \dots & k_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ k_{n1} & k_{n2} & \dots & k_{nn} \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{Bmatrix} \tag{4}$$

Here, u is the global unknown displacement vector, K is the global stiffness matrix which is the contributions of accumulating all the element stiffness matrices; while f is the global load vector which in fact is the contributions of accumulating all the element internal forces. For each node in the structure, the accumulated element internal forces sum to zero, except where a load is applied they sum to the balance of the applied loads.

Solving equation (3), we will obtain the displacements of the structure. Applying these displacements back onto each element, together with loading, we can evaluate its internal forces and stresses.

In summary, the conventional computation procedure of finite element method structural analysis consists of the following steps:

- Computing the element stiffness matrix. The element stiffness matrix shown in equation (1) depends only on the material used to build the structure, the shape of the element and the location of the element in the structure. The computation of stiffness matrix for each element is independent of the others.

- Assembling the global stiffness matrix. The global stiffness matrix in equation (3) is the result of accumulating the contributions of all the element stiffness matrices. It is computed by adding each stiffness value associated with each of the elements to the appropriate location (i.e. matrix row number and column number) in the global stiffness matrix.

- Assembling the global load vector. The global load vector is the contributions of all the loads applied onto the structure. Assembly of the global load vector is similar to assembling the global stiffness matrix: each load value is accumulated to its computed location in the global load vector.

- Solving the global system equation to obtain displacements. Gauss elimination, Choleski decomposition or other methods can be used to solve the linear equation system (3) to obtain the unknown displacements vector u.

- Evaluating the element stresses. Using the calculated displacements of the structure, each element's internal forces and stresses can be computed from equation (1).

Finally, we note that in industrial applications it is common to find structural analysis problems in which the global system equation is described by a $10,000 \times 10,000$ matrix, which although sparse nonetheless contains hundreds of thousands of non-zero entries.

3 A Basic Linear Algebra System in Miranda

To support the many vector and matrix operations involved in structural analysis, a small BLAS (Basic Linear Algebra System) package in Miranda has been developed. The package has two groups of functions which corresponds to a two level system. The first level defines

the basic data types such as vector and matrix, and the corresponding primitive operations on them. The second level deals with solving simultaneous linear systems of equations [8][13][3], using both direct and iterative solution methods.

What follows is a brief description of these Miranda functions and data types.

3.1 Data types

The first level of the BLAS system provides vector and matrix data types, together with special implementations for common special and sparse cases. It is divided into following modules:

Module name	**Description**
vector	vector data type and primitive operations;
matrix	matrix data type and primitive operations;
tmatrix	triangular matrix data type and primitive operations;
vbmatrix	variable bandwidth matrix data type and primitive operations;
svector	sparse vector data type and primitive operations;
smatrix	sparse matrix data type and primitive operations.

3.1.1 Vector data type

We give here a definition of the basic vector data type. In Miranda it is implemented using lists, although this is of course not necessary and in fact should not be so in other languages[14] (e.g. *Haskell* [6] and *Hope*$^+$ as we shall see). We assume that vectors have integer bounds and are zero-based:

```
vec * ::= VEC num [*]
```

The most primitive operations on vectors are to build a vector, update a vector, reference a vector element and get the bound of a vector. To build a vector from nothing, there are two approaches: give all the values of the vector explicitly, or give a function which generates all the values of the vector elements. These two methods correspond to functions `defvec` and `makevec`. Function `vecsub` references a vector element. While function `boundvec` returns the bound of a vector.

```
makevec  :: num -> (num -> *) -> vec *
makevec bound generator
   = VEC bound elementlist
     where
     elementlist = [ generator i | i <- [0..bound-1] ]

defvec   :: num -> [ (num,*) ] -> vec *
defvec bound updates
   = updtvec replace undefvec updates
     where
     undefvec = makevec bound (const undef)
     replace old_value value = value

vecsub   :: vec * -> num -> *
```

```
vecsub (VEC bound elementlist) i
   = elementlist ! i,                          if i >= 0 & i < bound
   = error "out of vector bound(vecsub)",      otherwise

boundvec :: vec * -> num
boundvec (VEC bound elementlist) = bound
```

A vector may be updated by replacing the values of some elements, incrementing the values of some elements or decrementing the values of some elements. We can define a higher order function `updtvec` for all these operations:

```
updtvec   :: (* -> * -> *) -> vec * -> [ (num,*) ] -> vec *
updtvec f v updates
   = foldl (updatevec f) v updates
     where
     updatevec f v (i,x)
        = error "out of vector bound(updtvec)",   if i < 0 \/ i >= bound
        = makevec bound generator,                otherwise
          where
          bound = boundvec v
          generator j = f (vecsub v j) x , if i = j
                      = vecsub v j,        otherwise
```

The first argument of the function represents what operations is applied onto the vector. If the *update* is just to replace the old value, the argument function can be `replace`; if the *update* is to increment, the argument function can be `(+)`; and so on as shown below:

```
updvec   :: vec * -> [ (num,*) ] -> vec *
updvec v updates
   = updtvec replace v updates
     where
     replace old_value value = value

incrvec  :: vec num -> [ (num,num)] -> vec num
incrvec v updates
   = updtvec (+) v updates
```

3.1.2 Matrix data type

We can define the matrix data type to mirror the vector type:

```
mat * ::= MAT (num,num) [[*]]
```

Two integers (a tuple) give the bounds of the matrix, i.e. the numbers of rows and columns in the matrix. The elements of the matrix are stored as a list of lists in row by row order. For example, the matrix

$$\begin{bmatrix} 4 & 2 & 0 & 2 & 0 \\ 2 & 5 & 2 & -1 & 0 \\ 0 & 2 & 5 & -3 & 2 \\ 2 & -1 & -3 & 7 & 1 \\ 0 & 0 & 2 & 1 & 6 \end{bmatrix} \tag{5}$$

is represented as:

```
MAT (5,5) [[ 4, 2, 0, 2, 0],
           [ 2, 5, 2,-1, 0],
           [ 0, 2, 5,-3, 2],
           [ 2,-1,-3, 7, 1],
           [ 0, 0, 2, 1, 6]]
```

Similar to what we have got for vectors, we can have following primitive operations on the matrix data type:

```
makemat  :: (num,num) -> ( (num,num) -> * ) -> mat *
makemat (nr,nc) generator
   = MAT (nr,nc) rows
     where
     rows = map makerow [0..nr-1]
     makerow i = [ generator (i,j) | j <- [0..nc-1] ]

defmat   :: (num,num) -> [ ((num,num),*) ] -> mat *
defmat bounds updates
   = updtmat replace undefmat updates
     where
     undefmat = makemat bounds (const undef)
     replace old_value value = value

matsub   :: mat * -> (num,num) -> *
matsub (MAT (nr,nc) elementlist) (i,j)
   = elementlist ! i ! j,                 if i >= 0 & i < nr &
                                             j >= 0 & j < nc
   = error "out of matrix bound(matsub)", otherwise

boundmat :: mat * -> (num,num)
boundmat (MAT bounds elementlist) = bounds
```

where, `makemat` and `defmat` are the two functions to build a matrix. Function `matsub` subscripts a matrix element. While function `boundmat` returns the bounds of a matrix as a tuple of numbers.

To update a matrix, we can define a higher order function `updtmat` for various updating operations as we did for vector:

```
updtmat  :: (* -> * -> *) -> mat * -> [ ((num,num),*) ] -> mat *
updtmat f m updates
   = foldl (updatemat f) m updates
     where
     updatemat f m ( (i,j),x )
        = error "out of matrix bound(updmat)", if i < 0 \/ i >= nr \/
                                                  j < 0 \/ j >= nc
        = makemat (nr,nc) generator,           otherwise
             where
```

```
(nr,nc) = boundmat m
generator (id,jd)
        = f (matsub m (id,jd)) x, if (i,j)=(id,jd)
        = matsub m (id,jd),       otherwise
```

The first argument of the function is of function type, and it represents what operation to be applied on to the matrix. For example, if the operation is to increment, the argument is `(+)` as shown below:

```
incrmat  :: mat num -> [ ((num,num),num) ] -> mat num
incrmat m updates
   = updtmat (+) m updates
```

Here, we note that function `updtmat` repeatedly applies `updatemat`, which updates one element of a matrix each time it is applied. So, this kind of matrix operations are called of *incremental*.

3.1.3 Triangular matrix data type

A square matrix has the same number of rows as columns. In a symmetric matrix, the value of the element at row `i` and column `j` is equal to the value of the element at column `i` and row `j`, so we need store only the diagonal elements and the elements above or below the diagonals. In other words, we need store only little above half of the n^2 elements. In a lower triangular matrix, all the elements above the diagonal are zeros, and we need not to store these zero elements. While in an upper triangular matrix all elements below the diagonal are zeros, and we need not to store these zero elements either.

Essentially the same representation is possible for symmetric, upper and lower triangular matrices, and we present only the lower triangular case:

```
ltmat * ::= LTMAT num [*]
```

All the elements of the lower triangular matrix are stored in a list row by row. It is straightforward to define functions to build and decompose lower triangular matrices by rewriting the definitions using `ltmat *` instead of `mat *`.

As an example, `ltm51` is the representation using our lower triangular data type for the 5×5 symmetric matrix (5) shown in section 3.1.2:

```
ltm51 = LTMAT 5 [4,2,5,0,2,5,2,-1,-3,7,0,0,2,1,6]
```

3.1.4 Variable bandwidth matrix data type

While matrices occurring in engineering computation are often large, they often consist largely of zeroes. Such "sparse" matrices commonly, although not always, have some pattern in the distribution of non-zero elements. Banded matrices are very common—where the non-zero elements appear in the main diagonal and in a small number of diagonal subvectors parallel to the main diagonal. These off-diagonal elements generally correspond to interactions between neighbouring elements in a two- or three-dimensional lattice. Even in non-regular structures there is still a tendency for interactions to have relatively short range.

A good data structure for these matrices should not store or compute with zero elements. In this section, we define a variable bandwidth matrix type based on Miranda list for symmetric sparse matrices.

The idea of variable bandwidth is to store each diagonal element together with its row, from the first leading non-zero element up to the diagonal element in that particular row. This avoids storing many zeroes in the common case of a symmetric matrix with banded structure which is not entirely regular. It fails to avoid storing so-called "secondary" zeroes, which lie between the leading non-zero element and the diagonal in each row. In the matrix shown in (6), the zero at row 3 and column 1 is of first category, and the zero at row 4 and column 3 is of second category. So, only those elements in bold are to be stored.

$$\begin{bmatrix} \mathbf{4} & \mathit{2} & \mathit{0} & \mathit{2} & \mathit{0} \\ \mathbf{2} & \mathbf{5} & \mathit{2} & \mathit{-1} & \mathit{0} \\ \mathit{0} & \mathbf{2} & \mathbf{5} & \mathit{0} & \mathit{2} \\ \mathbf{2} & \mathbf{-1} & \mathbf{0} & \mathbf{7} & \mathit{1} \\ \mathit{0} & \mathit{0} & \mathbf{2} & \mathbf{1} & \mathbf{6} \end{bmatrix} \tag{6}$$

To represent a variable bandwidth symmetric matrix, we need a number for its index bound (i.e. dimension), and a list for its second category zero and non zero elements. We also need to know the semibandwidth of each row and the location where the row is stored in the list. In fact, the semibandwidth and the location of each row can be derived from each other. If we choose using the address of diagonal element in the list to describe the location of each row, then the semibandwidth of row i is equal to the address of diagonal element at row i subtract the address of diagonal element at row $i-1$. Suppose we use vector `addiag` to store the addresses of diagonal elements. Then the address of element at row i and column j is `addiag[`i`] + `j `-` i, if it is a non-zero or second category zero element.

The variable bandwidth matrix data type definition in Miranda is as follows:

```
vbmat * ::= VBMAT num (vec num) [*]
```

The variable bandwidth matrix representation of the matrix in (6) is `vbm52`:

```
vbm52 = VBMAT 5 (VEC 5 [0,2,4,8,11]) [4,2,5,2,5,2,-1,0,7,2,1,6]
```

3.1.5 Sparse vector and sparse matrix data types

When a matrix is sparse, but no structure in the disposition of zero elements is evident, we can choose a representation which lists the values and locations of just the non-zero elements. We give such an implementation here for completeness, although it will not be needed in the remainder of the paper:

```
svec * ::= SVEC num [(num,*)]
smat * ::= SMAT (num,num) [((num,num),*)]
```

For example, the sparse vector

$$\begin{Bmatrix} 0 & 0 & 0 & -1 & 0 \end{Bmatrix}^T$$

can be represented as:

```
sv5 = SVEC 5  [(3,-1)]
```

and the sparse matrix

$$\begin{bmatrix} 4 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -3 & 0 \\ 0 & 0 & -3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 6 \end{bmatrix}$$

can be represented by:

```
sm53 = SMAT (5,5)
       [((0,0), 4), ((0,1), 2),
        ((1,3),-1), ((2,3),-3),
        ((3,2),-3), ((4,4), 6) ]
```

3.1.6 Discussion of efficiency resulting from data structure copying

In section 3.1.2, an incremental version of array accumulating function `incrmat` was given. Following defines function `new_incrmat` which is a new version of `incrmat` and is usually called of *monolithic* version.

```
new_incrmat :: mat num -> [((num,num), num)] -> mat num
new_incrmat m updates
   = makemat bounds generator  || alist is the index list
     where                     || blist' returns the value list
     bounds = boundmat m       ||  of given index (i,j)
     generator (i,j)
         = matsub m (i,j),            if ~(member alist (i,j) )
         = matsub m (i,j) + sum (blist' (i,j)), otherwise
     alist = unzip2a updates
     blist' (i,j) = unzip2b (filter (isit (i,j)) updates)
     isit (i,j) ((l,m),x) = ((i,j) = (l,m))

unzip2a ((a,b):ls) = a: (unzip2a ls)
unzip2a []         = []

unzip2b ((a,b):ls) = b: (unzip2b ls)
unzip2b []         = []
```

In order to discuss the efficiency of the two functions `incrmat` and `new_incrmat`, we make following assumptions:

- the dimension of matrix M is $n \times n$;
- the length of index-value pair list `updates` is l; and
- the number of different indices appeared in list `updates` is k.

The first version repeatedly applies function `updatemat` to a list of index-value pairs. Each time the function `updatemat` is invoked, one and only one element value of the matrix is updated, and a new matrix is generated. Generating a new matrix is often referred as a matrix

copying, though in fact it might not be copying the whole matrix for different implementations of functional languages. So function `incrmat` needs to scan a l length list once, and do $n \times n$ matrix copying l times. The monolithic version uses function `makemat` directly instead of using `updatemat`. So it avoids unnecessary data structure copying, and needs only one $n \times n$ matrix copying. Instead, it scans the update list $n^2 + k$ times.

Surely, both of them are unacceptable. Ideally, an array accumulate operation should only need to scan the `updates` list once, and do one matrix copying. Although there exists better ways to define array accumulating operations, we believe that an array data type and corresponding primitive accumulate operators as in *Haskell* is necessary. In fact, destructive array operations are also very important for some cases.

Similar discussions can be raised for other vector and matrix update operations.

3.2 Solving linear equation systems

The second level of the BLAS system deals with solving systems of linear equations. It consists of the following modules:

Module name	**Description**
gausselimination	Gauss elimination method;
ppgausselimination	Gauss elimination method considering partial pivoting;
ludecomposition	LU decomposition method;
lltdecomp	LL^T decomposition method (Choleski decomposition method);
ldltdecomp	LDL^T decomposition method;
pcg	Preconditioned conjugate gradient method;
tlltdecomp	LL^T decomposition method (using triangular matrix data type);
tldltdecomp	LDL^T decomposition method (using triangular matrix data type);
vblltdecomp	LL^T decomposition method (using variable bandwidth matrix data type);
vbldltdecomp	LDL^T decomposition method (using variable bandwidth matrix data type).

3.2.1 Gauss elimination method

The problem to be solved is:

$$A\ x = b \tag{7}$$

It has the following expanded form:

$$\begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0\ n-1} \\ a_{10} & a_{11} & \cdots & a_{1\ n-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n-1\ 0} & a_{n-1\ 1} & \cdots & a_{n-1\ n-1} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{Bmatrix} = \begin{Bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{Bmatrix} \tag{8}$$

Here, A is a $n \times n$ matrix, b is a vector of n elements and x is the unknown vector to be computed.

The Gauss elimination method has two phases: transformation to upper-triangular form, and back-substitution.

In the first phase, transformations are applied simultaneously to the matrix A and the right-hand-side vector b, which eventually ensure that the A's lower triangle is all zero, while

preserving the same solution set as the original system of equations:

$$\begin{bmatrix} a'_{00} & a'_{01} & \cdots & a'_{0\ n-1} \\ 0 & a'_{11} & \cdots & a'_{1\ n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a'_{n-1\ n-1} \end{bmatrix} \begin{Bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{Bmatrix} = \begin{Bmatrix} b'_0 \\ b'_1 \\ \vdots \\ b'_{n-1} \end{Bmatrix} \tag{9}$$

The transformation to triangular form takes $n-1$ steps. At step k ($k = 0, 1, \ldots n-2$), the k-th column of matrix A is eliminated. The algorithm is given by definitions (10) to (12):

$$a_{ij}^{k+1} = a_{ij}^k - (a_{ik}^k / a_{kk}^k) a_{kj}^k \qquad (k < i, j < n) \tag{10}$$

$$a_{ik}^{k+1} = 0 \qquad (k < i < n) \tag{11}$$

$$b_i^{k+1} = b_i^k - (a_{ik}^k / a_{kk}^k) b_k^k \qquad (k < i < n) \tag{12}$$

In an imperative programming language this stage has the following form:

```
FOR k = 0 STEP 1 UNTIL (n-2) DO
        FOR i = k+1 STEP 1 UNTIL (n-1) DO
                FOR j = k+1 STEP 1 UNTIL (n-1) DO
                    A[i,j] = A[i,j] - A[i,k]/A[k,k] * A[k,j]
                ENDFOR
                B[i] = B[i] - A[i,k]/A[k,k] * B[k]
        ENDFOR
ENDFOR
```

The second phase, back-substitution, computes the values of `x[n-1]`, `x[n-2]`, ...and `x[0]` at last. The algorithm is given by definition (13):

$$x_i = (b'_i - \sum_{k=i+1}^{n-1} a'_{ik} x_k) / a'_{ii} \tag{13}$$

This is very straightforward, and can be coded as:

```
FOR i = (n-1) STEP -1 UNTIL 0 DO
        X[i] = B[i]
        FOR k = i+1 STEP 1 UNTIL n-1 DO
                X[i] = X[i] - A[i,k] * X[k]
        ENDFOR
        X[i] = X[i] / A[i,i]
ENDFOR
```

A Miranda version of the algorithm is as follows. Function `gausselimination` takes a tuple argument which has two components: matrix A and vector b as shown in (7). It returns a vector as the result of solving equation (7), i.e. the solution vector x:

```
gausselimination :: (mat num, vec num) -> vec num
gausselimination (a,b)
   = backwarding ( forwarding (a,b) )
```

where, functions `forwarding` and `backwarding` correspond to the two phases of the algorithm described above:

```
forwarding (a,b)
   = foldl eliminate (a,b) [0..n-2]
     where
     n = boundvec b

eliminate (a,b) k
   = error ("ill-condition at row "++(show k)),  if matsub a (k,k) = 0
   = (incrmat a updalist, incrvec b updblist),   otherwise
     where
     updalist = [ ((i,j),-(matsub a (i,k) / (matsub a (k,k)) )
                * (matsub a (k,j)) ) | i<-[k+1..n-1]; j <- [k+1..n-1] ]
     updblist = [ (i,-(matsub a (i,k) / (matsub a (k,k))) * vecsub b k)
                | i <- [k+1..n-1] ]
     n        = boundvec b

backwarding (aa,bb)
   = x
     where
     x = makevec n f
     n   = boundvec bb
     f i = (vecsub bb i - sum ([matsub aa (i,j) * (vecsub x j)
                               |j<-[i+1..n-1]]) ) / aii
           where
           aii = error ("ill-condition"++(show i)),
                                       if matsub aa (i,i) = 0
               = (matsub aa (i,i) ),   otherwise
```

3.2.2 Gauss elimination method with partial pivoting

The algorithm given so far is mathematically accurate but often results in very poor numerical results for some coefficient matrices. Pivoting is a technique for optimizing the numerical accuracy of the solver. It works by interchanging the rows of the matrix at each step in order to maximize the precision of the subtraction specified by definition (10) (we perform partial pivoting only for simplicity reason; full pivoting would involve column interchanging as well, but is generally unnecessary).

The algorithm is modified by adding a pivoting step before each elimination.

```
ppgausselimination :: (mat num, vec num) -> vec num
ppgausselimination (a,b) = backwarding ( forwarding (a,b) )

forwarding (a,b)
   = foldl eliminate' (a,b) [0..n-2]
     where
     n = boundvec b
```

```
eliminate' (a,b) k
   = eliminate (pivot k (a,b)) k

pivot k (a,b)
   = (intchrow k j a, interchvec k j b),  if k~=j
   = (a,b),                               otherwise
     where
     j = findpivot k a
```

Function `findpivot` returns the row number where the pivot is in. And function `pivot` interchanges the current row and the pivot row founded for both the coefficient matrix and the right hand side vector. While functions `backwarding` and `eliminate` are the same as before.

3.2.3 LU decomposition method

With the LU decomposition method, we first decompose the coefficient matrix A into following form:

$$A = LU \tag{14}$$

That is:

$$A = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{10} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n-1\ 0} & l_{n-1\ 1} & \cdots & 1 \end{bmatrix} \begin{bmatrix} u_{00} & u_{01} & \cdots & u_{0\ n-1} \\ 0 & u_{11} & \cdots & u_{1\ n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n-1\ n-1} \end{bmatrix} \tag{15}$$

The elements of L and U are derived from A according to the following formulae:

$$l_{ij} = (a_{ij} - \sum_{k=0}^{j-1} l_{ik}u_{kj})/u_{jj} \qquad (i > j) \tag{16}$$

$$u_{ij} = a_{ij} - \sum_{k=0}^{i-1} l_{ik}u_{kj} \qquad (i <= j) \tag{17}$$

Here, L is a lower triangular matrix with its diagonal values equal to unity, and U is an upper triangular matrix.

Based on the above decomposition, solving equation (7) is simplified to first solving equation

$$L\ y = b \tag{18}$$

and then solving equation

$$U\ x = y \tag{19}$$

Solving equation (18) is forward substitution, just as we had in Gauss elimination, and solving equation (19) is backward substitution. The algorithms are given by (20) and (21) respectively.

$$y_i = b_i - \sum_{j=0}^{i-1} l_{ij}y_j \tag{20}$$

$$x_i = (y_i - \sum_{j=i+1}^{n-1} u_{ij} x_j)/u_{ii} \tag{21}$$

The decomposition phase can be programmed in Miranda by following equations (16) and (17) very directly:

```
ludecomposition :: mat num -> (mat num, mat num)
ludecomposition mA
       = (mL, mU)
         where
         mL = makemat (boundmat mA) fL
         mU = makemat (boundmat mA) fU

         fU (i,j)
           = 0,                                    if i>j
           = matsub mA (i,j) -
             (sum [(matsub mL (i,k)) * ( matsub mU (k,j))
             | k <- [0..i-1] ] ) ,                 otherwise

         fL (i,j)
           = 1,                                    if i=j
           = 0,                                    if i<j
           = (matsub mA (i,j) -
             (sum [(matsub mL (i,k))*(matsub mU (k,j))
             | k <- [0..j-1] ] ) ) / mUjj ,        otherwise
             where
             mUjj = matsub mU (j,j),        if matsub mU (j,j) ~= 0
                  = error "ill-condition",  otherwise
```

The forward and backward substitution are just as before with Gauss elimination, except that we must take care to refer to the correct submatrix:

```
forwarding ( b, mL )
   = y
     where
     n = boundvec b
     y = makevec n f
     f i = (vecsub b i - sum ([matsub mL (i,j) * (vecsub y j)
                               |j<-[0..i-1]]) )

backwarding ( y, mU )
   = x
     where
     n = boundvec y
     x = makevec n f
     f i = (vecsub y i - sum ([matsub mU (i,j) * (vecsub x j)
                               |j<-[i+1..n-1]]) ) / mUii
           where
```

```
        mUii = error ("ill-condition"++(show i)),
                                    if matsub mU (i,i) = 0
             = matsub mU (i,i),     otherwise
```

We put this together by defining the function `lusolution` which takes two arguments: coefficient matrix A and right hand side vector b, and returns vector x as the result of solving (7).

```
lusolution      :: (mat num, vec num) -> vec num
lusolution (a,b)
   = backwarding ( forwarding (b, mL), mU )
     where
     (mL, mU) = ludecomposition a
```

It is obvious that the decomposition is an independent phase in the algorithm. One of the reasons LU decomposition is attractive compared to Gauss elimination because for a very similar computational effort the left-hand-side of the equation is transformed once and for all so that we can solve many linear equation systems with same coefficient matrix, but different right hand side vectors. In this case, we can decompose the coefficient matrix first, then apply the result to many right hand side vectors. This can be easily done by defining function `lusolutionLU`:

```
lusolutionLU    :: ( (mat num, mat num), vec num ) -> vec num
lusolutionLU ( (mL, mU), b)
   = backwarding ( forwarding (b, mL), mU )
```

where, matrices `mL` and `mU` are the result of decomposing a coefficient matrix.

3.2.4 LL^T decomposition method(Choleski decomposition method)

For a symmetric matrix A, we can use LU decomposition method just described in previous section. However, we can decompose A into following forms:

$$A = LL^T \tag{22}$$

where

$$L = \begin{bmatrix} l_{00} & 0 & \cdots & 0 \\ l_{10} & l_{11} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n-1\ 0} & l_{n-1\ 1} & \cdots & l_{n-1\ n-1} \end{bmatrix} \tag{23}$$

This allows us to use triangular matrix data type to store both coefficient matrix A and decomposed result L. By doing this, we benefit from both memory usage and computation as we need only to compute one triangular matrix.

As in LU decomposition method, solving equation (7) is simplified to solve equations (24) and (25).

$$L\ y = b \tag{24}$$

$$L^T\ x = y \tag{25}$$

Definitions (26) and (27) gives the LL^T decomposition algorithm, followed by the corresponding Miranda code:

$$l_{ii} = (a_{ii} - \sum_{k=0}^{i-1} l_{ik}^2)^{1/2} \tag{26}$$

$$l_{ij} = (a_{ij} - \sum_{k=0}^{j-1} l_{ik} l_{jk})/l_{jj} \qquad (i > j) \tag{27}$$

```
lldecomp      :: mat num -> mat num
lldecomp mA
    = mL
      where
      mL = makemat (boundmat mA) fL

      fL (i,j)
         = sqrt ( matsub mA (i,i) -
            sum [ (matsub mL (i,k) * matsub mL (i,k) )
            | k<-[0..i-1]] ) ,              if i=j
         = 0,                               if i<j
         = (matsub mA (i,j) -
            (sum [(matsub mL (i,k))*(matsub mL (j,k))
            | k <- [0..j-1] ] ) ) / mLjj,   otherwise
           where
           mLjj = matsub mL (j,j),          if matsub mL (j,j) ~= 0
                = error "ill-condition",    otherwise
```

3.2.5 LDL^T decomposition method

In the LL^T decomposition method described in section 3.2.4, square root operation is required for the computation of the diagonal elements of matrix L. To avoid this, alternatively we can decompose A into following form:

$$A = LDL^T \tag{28}$$

i.e.

$$A = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ l_{10} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n-1\ 0} & l_{n-1\ 1} & \cdots & 1 \end{bmatrix} \begin{bmatrix} d_{00} & 0 & \cdots & 0 \\ 0 & d_{11} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & d_{n-1\ n-1} \end{bmatrix} \begin{bmatrix} 1 & l_{10} & \cdots & l_{n-1\ 0} \\ 0 & 1 & \cdots & l_{n-1\ 1} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix} \tag{29}$$

where

$$d_i = a_{ii} - \sum_{k=0}^{i-1} l_{ik}^2 d_k \tag{30}$$

$$l_{ij} = (a_{ij} - \sum_{k=0}^{j-1} l_{ik} l_{jk} d_k)/d_j \qquad (i > j) \tag{31}$$

So, the original problem turns out to be solving three simpler equations (32–34):

$$L\ y = b \tag{32}$$

$$D\ z = y \tag{33}$$

$$L^T\ x = z \tag{34}$$

Because solving (32–34) is very straightforward and very similar with the algorithm described before, we only give the decomposition algorithm and the corresponding code. In the code, actually, the diagonal matrix D is stored in the diagonal of matrix L.

```
ldltdecomp      :: mat num -> mat num
ldltdecomp mA
  = mL
    where
    mL = makemat (boundmat mA) fL
    fL (i,j)
       = ( matsub mA (i,i) - sum [ (matsub mL (i,k) *
          matsub mL (i,k) * matsub mL (k,k) ) | k<-[0..i-1]] ),  if i=j
       = 0,                                                     if i<j
       = (matsub mA (i,j) -
          (sum [(matsub mL (i,k))*(matsub mL (j,k))*(matsub mL (k,k))
          | k <- [0..j-1] ] ) ) / mLjj,                  otherwise
         where
         mLjj = matsub mL (j,j),        if matsub mL (j,j) ~= 0
              = error "ill-condition",  otherwise
```

3.2.6 Preconditioned conjugate gradient method

The preconditioned conjugate gradient (PCG) method is an iterative method for solving large scale linear equation systems with positive definite coefficient matrices [8][3][4][10]. A typical PCG algorithm solving (7) has following form:

- Initialisation:
  - Choose x^0;
  - Compute $r^0 = b - Ax^0$;
  - Solve $C\ \hat{r}^0 = r^0$;
  - Set $p^0 = \hat{r}^0$.
- Iterations (k=0, 1, ... until convergence):
  - Compute $\alpha_k = \frac{<r^k, r^k>}{<p^k, Ap^k>}$;
  - Compute $x^{k+1} = x^k + \alpha_k p^k$;
  - Check convergence.
  - Compute $r^{k+1} = r^k - \alpha_k A p^k$;
  - Solve $C\ \hat{r}^{k+1} = r^{k+1}$;
  - Compute $\beta_k = \frac{<r^{k+1}, \hat{r}^{k+1}>}{<r^k, \hat{r}^k>}$;
  - Compute $p^{k+1} = \hat{r}^{k+1} + \beta_k p^k$.

In the above, $< x, y >$ denotes the inner product of vectors x and y and C is the preconditioning matrix which is introduced to speedup the convergence rate. If an unit matrix is chosen as the preconditioning matrix C, the above algorithm then become standard conjugate gradient method.

Following gives a Miranda version of the PCG method using diagonal preconditioning matrix, i.e. the diagonal matrix of the coefficient matrix A is used as the preconditioning matrix.

```
pcg :: num -> mat num -> vec num -> vec num -> num -> (num, vec num)
pcg n a b x0 eps
    = (k, x)
      where
      (k, x, r, p, gama) = until converge iterate state0

      converge (k, x, r, p, gama)
            = (k > n) \/ (t < eps)
              where
              t = vecprod r r

      iterate (k, x, r, p, gama)
            = (k+1, nx, nr, np, ngama)
              where
              u       = mmatvec a p
              alpha   = - gama / vecprod p u
              nx      = addvec' x p (-alpha)
              nr      = addvec' r u alpha
              rr      = mulvec nr diag'
              ngama   = vecprod nr rr
              beta    = ngama / gama
              np      = addvec' rr p beta

      diag' = makevec n f
              where
              f i = 1.0 / matsub a (i,i)

      state0
            = (0, x0, r0, p0, gama0)
              where
              r0 = subtvec b (mmatvec a x0)
              rr = mulvec r0 diag'
              p0 = rr
              gama0 = vecprod r0 rr
```

where, function `vecprod` computes the inner product of two vectors, `(addvec' v1 v2 a)` returns `(v1 + a*v2)`, `subtvec` and `mulvec` perform element wise substraction and multiplication on vectors respectively. While function `mmatvec` carries out matrix vector multiplication operation.

3.2.7 Using triangular and variable bandwidth matrix data types

As we have seen, triangular matrix data type can be used to represent symmetrical matrices, and variable bandwidth matrix data type can be used to represent sparse symmetrical matrices. The LL^T and LDL^T decomposition and PCG methods for symmetric coefficient matrix described in sections 3.2.4–3.2.6 can be easily modified to adapt triangular matrix data type and variable bandwidth matrix data type. The details of these codes are not listed here.

3.2.8 Discussion (Choosing the appropriate method)

Broadly, the direct solution methods discussed in sections (3.2.1–3.2.5) for solving linear equation system can be divided into two categories: Gauss elimination method and decomposition method.

Suppose the dimension of the coefficient matrix is n. In Gauss elimination methods, the elimination phase needs $n-1$ successive steps, with each step eliminates one column of the coefficient matrix and requires one matrix copying. In the back substitution phase, one vector copying is required before we can get the final result.

The decomposition methods fulfill the decomposition by using lazy evaluation, one of the most important properties of functional programming. They don't need more copyings of the coefficient matrix as in Gauss elimination methods. As shown in table 1, the LU decomposition method requires 2 triangular matrix copyings (or 2 variable bandwidth matrix copyings in the case of using variable bandwidth matrix data type) and 2 vector copyings, LL^T decomposition method requires 1 triangular matrix copying (or 1 variable bandwidth matrix copying when using variable bandwidth matrix data type) and 2 vector copyings, and so on.

Method	No. of Matrix Copyings	No. of Vector Copyings
Gauss elimination	N - 1	1
LU decomposition	2*	2
LL^T decomposition	1*	2
LDL^T decomposition	1*	3

Table 1: Comparison regarding to data structure copying. * denotes triangular matrix.

In addition, in decomposition methods, the decomposition phase is an independent module, and can be separated from the substitutions. We may first decompose a coefficient matrix once and for all, and later apply the decomposition result to many right hand side vectors. This is not so for the Gaussian elimination methods. So, generally speaking, decomposition methods rather than Gauss elimination methods should be employed.

Regarding data structure or data type, this very much depends on the problem to be solved. For the finite element structural analysis problems, the variable bandwidth matrix data structure is used in this paper. This is because the coefficient matrix (i.e. the global stiffness matrix) is usually highly banded.

4 Programming Finite Element Analysis Computation in Miranda

In section 2, we briefly described the finite element method for structural analysis. In this section, we discuss how to program finite element structural analysis computation in Miranda.

4.1 Element stiffness matrix computation

The stiffness matrix of a finite element is a kind of physical property of that element in the structure. Its value only depends on the material used to build the structure and the geometrical shape and location of that element.

Taking a two dimensional pin jointed bar element as example, we can design following functions to access the material property of an element, the nodal numbers of an element, and the coordinate values of a node:

```
getnxy  :: num -> (num,num)
getenlr :: num -> (num,num)
getemat :: num -> num
getmpro :: num -> (num,num)
```

where, given an element number, `getnxy` returns the coordinates of a node; `getenlr` returns the left-end node number and right-end node number of an element; `getemat` returns the material property number of an element; and `getmpro` returns the material coefficients given a material number.

The stiffness of a two dimensional pin jointed bar element is a 4×4 matrix. Using matrix data type defined in section 3.1.2, following gives the code for element stiffness matrix computation.

```
elemstiff :: num -> mat num
elemstiff element
  = makemat (4,4) f
    where
    f (i,j) = ... ea ei xl yl xr yr ... (detail not given)
    (ea,ei) = getmpro (getemat element)
    (nl,nr) = getenlr element
    (xl,yl) = getnxy nl
    (xr,yr) = getnxy nr
```

Here, function `elemstiff` takes an element number as its argument and returns the stiffness matrix of that element. In the code, the detail of calculating the value of element stiffness matrix is not given only for simplicity reason.

4.2 Assembling global stiffness matrix

To solve a structural analysis problem using finite element method, we need to build the global equation system, i.e. the global stiffness matrix and the global load vector. After having the global stiffness matrix and load vector, the unknown displacements can be obtained by solving the global system equations, and the element internal forces can then be evaluated.

To compute global stiffness matrix, we need a data type. We choose variable bandwidth matrix which was discussed in section 3.1.4 as our data type to store global stiffness matrix.

The global stiffness matrix, in fact, is the contribution of all the finite elements which compose the whole structure. Represented in a functional style, assembling global stiffness matrix has the following form:

```
global_stiffness_vbm :: vbmat num
global_stiffness_vbm
   = incrvbmat initial_value (concat (map assemble_stiffness [1..nelem]))
     where
     initial_value = makevbmat ndof diagadrv (const 0)

assemble_stiffness :: num -> [ ((num,num),num) ]
```

Where, `nelem` is the total number of elements; `ndof` is the total number of degrees of freedom, `diagadrv` is the vector which stores the addresses of the diagonal elements of the global stiffness matrix. While the function `assemble_stiffness` computes the contribution of a given element, and returns a list which specifies the value and location (row number and column number in the global stiffness matrix) of this element's contribution to the global stiffness matrix. And the function `incrvbmat` is the accumulate operation on variable bandwidth matrix. `map` and `concat` are two primitive functions in Miranda. `concat` joins a list of list together into a single list, `map` applies a function onto each element of a list.

4.3 Assembling global load vector

Very similarly to the discussion about assembling global stiffness matrix, the global load vector is the contribution of all loads. Assembling one load into the global load vector is to calculate the value of that load and find its location in the global load vector. The function to do this has following form:

```
global_load_vec :: vec num
global_load_vec
   = incrvec initial_value (concat (map assemble_load [1..nload]))
     where
     initial_value = makevec ndof (const 0)

assemble_load :: num -> [ (num,num) ]
```

Where, `nload` is the total number of loadings, and function `assemble_load` computes the contribution of a given loading and returns a list which indicates the locations and values of this loading's contribution to the global load vector.

4.4 Solving global system equation to obtain displacement

The global stiffness matrix is symmetric. So the linear equation system can be solved by using LL^T decomposition method, i.e. Choleski decomposition method which was discussed in section 3.2.4. In fact, we use the variable bandwidth matrix version as we have used variable bandwidth matrix data type to store the global stiffness matrix. The solution of solving the global system equation is the unknown displacement vector of the structure.

4.5 Evaluation of element stress

The stress of an element in a structure is a function of its stiffness, structure deformation and the loadings applied upon the element. So, evaluating the stress of a two dimensional pin jointed bar element can be designed as:

```
elem_stress :: num -> vec num
elem_stress element
   = makevec length f
     where
     f i     = ...ea ei xl yl xr yr ul vl ur vr ...
     (ea,ei) = getmpro (getemat element)
     (nl,nr) = getenlr element
     (xl,yl) = getnxy nl
     (xr,yr) = getnxy nr
     (ul,vl) = getnuv nl
     (ur,vr) = getnuv nr

stresses = map elem_stress [1..nelem]
```

where, function `getnuv` returns the displacements of a node. Based on function `elem_stress`, the stresses of all the elements in a structure can be easily computed by `stresses`.

5 Numerical Examples and Performance Result

In this section, we present the timing results of solving some small example problems using the functional programs we have developed, and discuss the related efficiency issue.

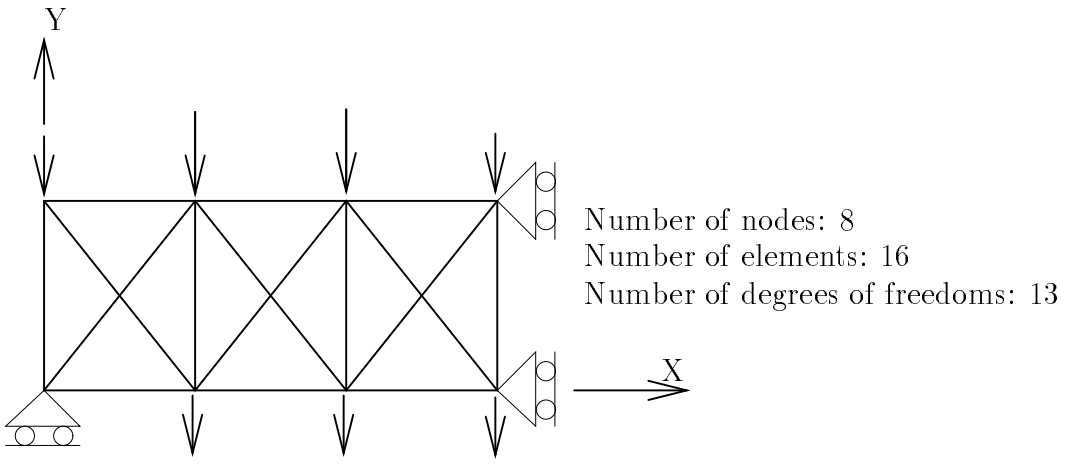


Figure 2: Structure of example 1

Using Miranda version 2.009, table 2 shows the statistics of solving the three example problems shown in Figures 2–4 using the Miranda program. The program runs on a Sun

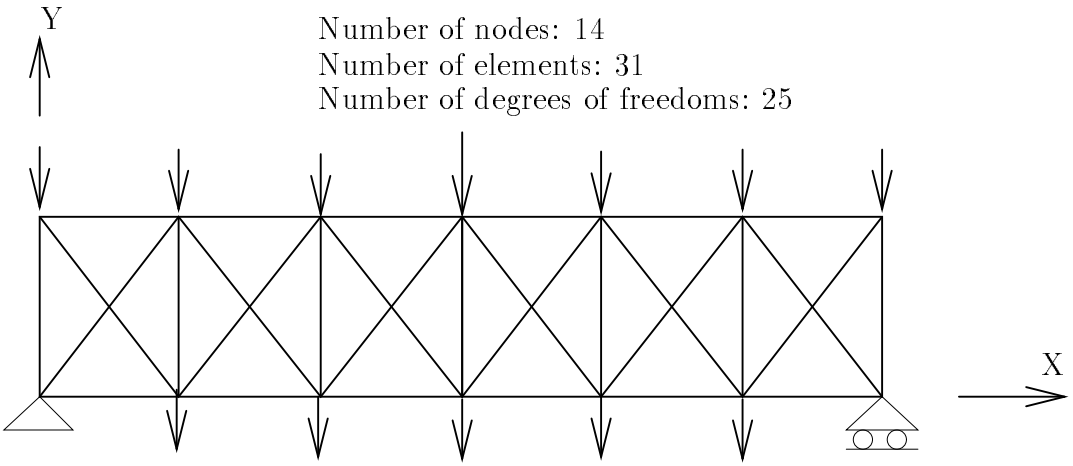


Figure 3: Structure of example 2

Example	No. of Nodes	No. of Elements	Matrix Size	Execution Time
1	8	16	13×13	36
2	14	31	25×25	100
3	36	72	62×62	624

Table 2: Execution times of Miranda program(in seconds)

workstation with Sun 3/280 CPU, 32MBytes memory, and SunOS 4.1.1 operating system[2]. We have set the Miranda heap size to be 800,000 cells, with each cell occupies 9 bytes memory.

Rewriting this Miranda program into *Hope*$^+$ [3], it derives into two versions: lazy version and unlazy version. Because *Hope*$^+$ does not support recursively defined data structures, we are unable to write a LL^T decomposition algorithm as in Miranda. To achieve this, we have to use constructors which are the only objects to be evaluated lazily in *Hope*$^+$. We construct an matrix with type "`mat (LAZY num)`", so that the elements of this matrix can be evaluated lazily and we can write a *Hope*$^+$ version of the decomposition algorithm. Of course, we can program LL^T decomposition algorithm just like we did in imperative programming language, not using the lazy evaluation property of functional programming. The algorithm can be viewed as a n step process, at each step one column of matrix L is generated. Here, n is the dimension of the matrix to be decomposed.

What are the differences between the *Hope*$^+$ lazy version and unlazy version of LL^T decomposition algorithm? We can see that the lazy version function takes the original coefficient matrix as argument, and generates a new matrix as the decomposed result. This means that theoretically it does only one matrix copying. While the unlazy version generates a new matrix at each step of decomposition which has n steps in total. So it does n matrix copyings.

[2] In order to get timing result under Unix system, we run our program during low load average period.

[3] *Hope*$^+$ vector data type is used, but it does not have Haskell like accumulate operator.

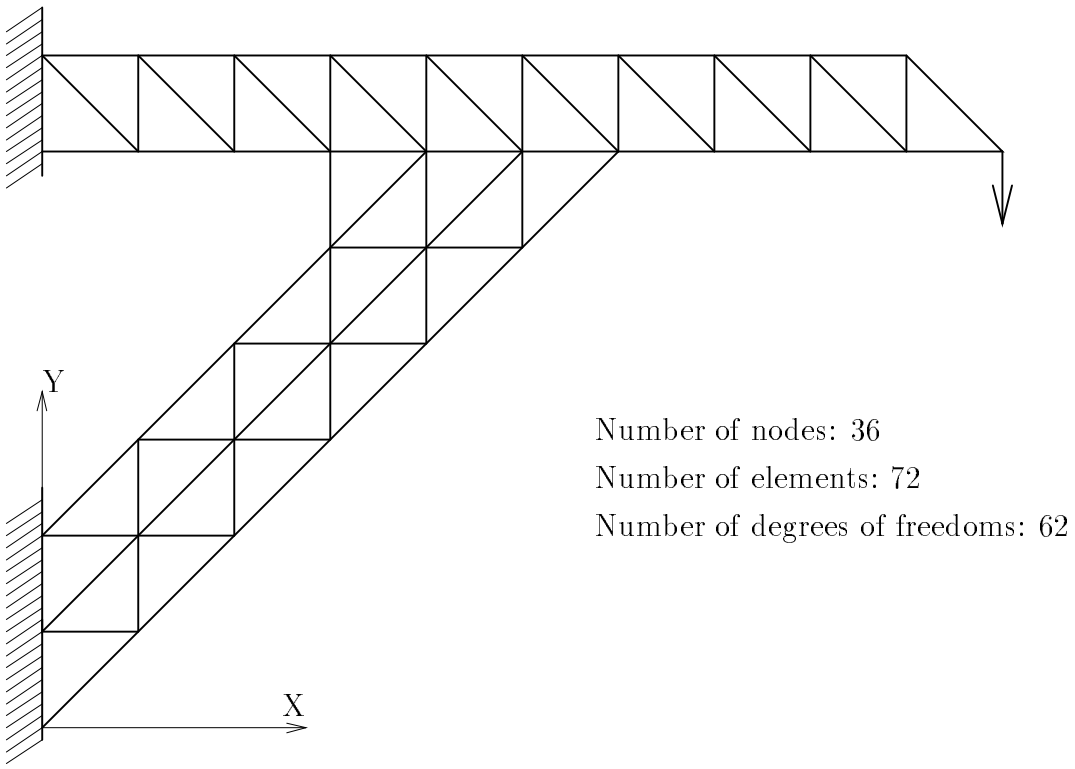


Figure 4: Structure of example 3

Table 3 shows the execution times of solving the three examples by using different versions of

Program version	Example 1	Example 2	Example 3
Lazy version	2.48	11.10	269.50
Unlazy version	2.06	8.46	176.90
Revised version	1.32	4.02	29.76

Table 3: Execution time of *Hope*$^+$ programs(in seconds)

the *Hope*$^+$ program. The programs were compiled by *Hope*$^+$ compiler version 3.2.1, and run on Sun 3/60 with 12MBytes memory under SunOS 4.1. The initial heap size of the compiled *Hope*$^+$ code was set to 2MBytes. The revised version corresponds to the result of a series of small changes in the code made after profiling the code using standard Unix tools. These changes include using improved versions of functions `incrmat` , `incrvbmat` and `makevbmat`, rewriting expression "$(i, j) = (i0, j0)$" as "$(i = i0)\ and\ (j = j0)$", and similar things to tune the code.

The lazy version LL^T decomposition algorithm does only one matrix copying, while the unlazy version does n^2 matrix copyings. We would expect that the lazy version took less

time to execute than that of the unlazy version. However, in fact, the result shows that the efficiency of the lazy version is much worse than the unlazy version. This might suggest that the *Hope*$^+$ system used was not properly implemented for problems with recursively defined data structures.

For completeness, table 4 gives the execution time of a C language version program running on the same machine as the *Hope*$^+$ programs did. Example 4, a new example problem which

Example 1	Example 2	Example 3	Example 4
0.22	0.60	1.50	704.67

Table 4: Execution time of C program(in seconds)

has 961 nodes, 1800 elements and 1919 independent d.o.f., is also solved by the C version program.

From the above results, it looks like that the performance of functional programming languages are very disappointing. However, it should be noticed that the revised *Hope*$^+$ program still does n copyings of the global stiffness matrix during LL^T decomposition and scans the `updates` list n^2 times while doing a single matrix accumulate operation. If we could eliminate the number of the global stiffness matrix copyings to 1 when doing LL^T decomposition and the number of times scanning the `updates` list to 1 when doing a matrix accumulation, the performance of the *Hope*$^+$ version program could be expected to improve very much.

In the process of pursuing efficient functional code, we have experienced the advantage of referential transparency property of functional programming. We can modify the code easily without changing the semantics provided only that the expressions replaced return the same values. However, the referential transparency property does not guarantee that it will improve the overall efficiency of the program. In fact, it has no relationship with computing resource required at all. Sometime, small changes in the code may end up with huge slow down of the program. So, it is quite important that programmers should have thoroughly understanding of functional programming and some detail of the system implementation as well.

6 Exploring Parallelism in Finite Element Computation by Using Functional Programming

In this section, we begin by giving a brief overview on parallelism paradigms, with emphases on programming distributed memory MIMD machine. Then we explore the large grain parallelism in finite element program in search for the way to parallelize it on arrays of transputers.

6.1 Parallelism paradigms

To achieve maximum parallelism on a distributed memory MIMD machine, there are two principle rules: to balance the workloads and to minimise the communication overhead in the processor network [12]. It is obvious that we should keep a balanced workload in the processor network, otherwise some of the processors may finish their work earlier than others and become idle. With present technology, the speed of communication is much slower than

the speed of computing, so too much communication overhead in a program certainly will reduce the overall efficiency of the program.

Furthermore, to reduce the communication overhead, it is often necessary to introduce some computation overlapping among processors, i.e. re-compute something rather than receive it from other processors. Obviously, the amount of the computation overlapping should be kept small so that it does not overweigh the reduced communication overhead. Finally, there are always some sequential segments in a computation process, such as input, output and so on. While executing these segments, it is clear that only one processor is active and all the others are idle. So the number and execution time of sequential segments should be controlled strictly.

To develop an efficient parallel program, it is important to identify all the opportunities for parallelism in the computation. As we will see, functional programming makes this task very easy. Broadly speaking, parallelism paradigms for distributed memory MIMD machines can be classified into three categories: **divide-and-conquer**, **processor farming**, and **pipeline** or **systolic array**.

Divide-and-conquer is an approach of first dividing a problem into a number of subproblems, solving these subproblems simultaneously and then combining the results to get the solution of the original problem. Divide-and-conquer parallelism often occurs in a recursive style which divides a subproblem into subsubproblems until it can be handled on a single processing element. Moreover, sometime the subproblems do need to communicate with each other in order to obtain the final solution of the original problem. Suppose we want to manipulate a list on a `p` processing element network. We can partition the list into `p` sub-lists, and manipulate each of them on one of the processors, and then combine the results of manipulating the sub-lists to get the solution of the original problem if each of the sublists can be handled by a single processor. This can be coded in functional style as:

```
f     :: [*] -> **
f ls  = combine [res_1, res_2, ..., res_p]
        where
        res_1 = f_1 sub_list_1
        res_2 = f_2 sub_list_2
        ... ...
        res_p = f_p sub_list_p
        [sub_list_1, sub_list_2, ... , sub_list_p]
              = partition p ls
```

Here, `res_i` (`i=1,2,...,p`) can be evaluated on different processors simultaneously, so parallelism is achieved.

Many scientific computation problems require repeated execution of the same code on different initial data. Later runs of the code don't need any knowledge of previous runs. Then the solutions of all these runs are combined to form the solution of the original problem. All these separate runs are independent with each other, and obviously can be executed in parallel. Using MIMD machines to solve this kind of problems, each processor can be assigned one or more of the runs and executes them in parallel with other processors. This is **processor farming**. Obviously, the expression `(map f [x1, x2, ..., xn ])` is a form of processor farming parallelism, as `(f xi)` and `(f xj)` can be executed on separate processors independently. However, as we will see later, not all `map` function applications are suitable for parallelising on distributed memory machines, because we have to control the program

granularity very strictly—if the task size is too small, the execution will be swamped by communication. Furthermore, when using processor farming, either the number of tasks should be far larger than the number of processors available if the tasks have different execution times, or the number of tasks should be dividable by the number of processors if the tasks have equal execution times in order to avoid the work load balance problem and keep the processor idle time small.

A **pipeline algorithm** is an ordered set of program segments in which the output of each segment is the input of its successor. As in a production line, all the segments must produce results at a same rate, so that all processors get a balanced workload, otherwise the slowest segment will become a bottleneck. Another point is that pipeline is always designed for large amount of 'production', otherwise it will not make much sense. In functional programming, function compositions are exactly pipelines, though most of them are not balanced. For example, when we apply `(f.g.h)` to `x`, the result of `(h x)` is the argument of function `g`, and the result of `(g (h x))` is the argument of function `f`. If functions `f`, `g` and `h` have different workloads, Figure 5(a) is not a good pipeline form. In this case, we can rearrange the pipeline to forms (b) and (c) as shown in Figure 5 very easily with the help of a higher order function, in the hope that one of the formulations might turn out to be a good pipeline form.

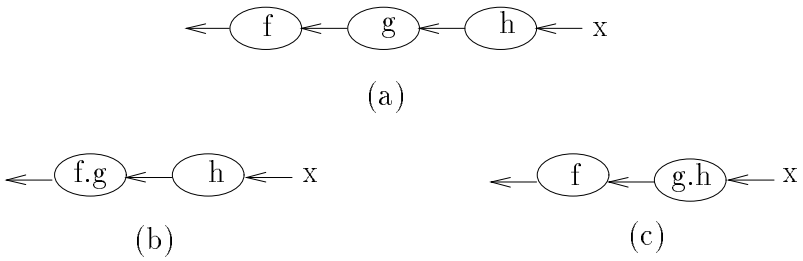
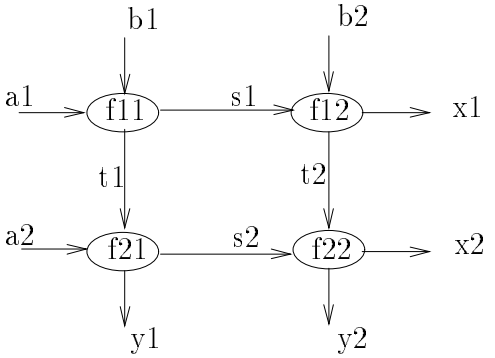


Figure 5: Function composition as pipeline

A **systolic array** is a more general form of pipeline. In a systolic array, the outputs of a processor are the inputs of several other processors. Figure 6 shows a simple form of systolic array. Just as in a pipeline all the processors should get a balanced workload, and it is also very important to keep the communication overhead low.

6.2 Parallel version of "map" function "farm"

In the previous section, we saw that there exists many opportunities for parallelism in functional programs. We can partition a complicated expression into smaller sub-expressions, and treat this as divide-and-conquer parallelism. We can treat function compositions as pipeline parallelism. And we can treat expressions with `map` function applications as processor farming parallelism. Also, as in the LL^T decomposition method, the decomposition was done by using lazy evaluation, one of the most important features of functional programming. In that case, the order of evaluation of all the elements of the matrix L was not specified. In fact, the order can be chosen arbitrarily provided that the data dependences among them were kept. This certainly creates opportunities for parallelism. However, in distributed memory MIMD machines, not all of this potential parallelism can be efficiently explored due to the issues of



(a) A systolic array

res = (x1, x2, y1, y2)
where
(s1,t1) = f11 (a1, b1)
(x1,t2) = f12 (s1, b2)
(s2,y1) = f21 (a2, t1)
(x2,y2) = f22 (s2, t2)

(b) Functional form

Figure 6: Systolic array and its functional form

workload balance and communication overhead.

In this paper, we investigate the use of a special version of the function `map`, called `farm` which implements self-scheduling processor farming parallelism, one of the most commonly used parallelisms in engineering computations.

To parallelise expression `(map f [x1,x2,...,xn])`, we would like to evaluate `(f x1)` on a free processor in the processor network, `(f x2)` on another free processor, and so on. By introducing a new function `farm`, we can rewrite the above expression as `(farm f [x1,x2,...,xn])`. So `farm` has type:

```
farm :: (* -> **) -> [*] -> [**]
```

Operationally, `farm` fires a number of expressions on to a processor network. Each expression is an application of the argument function `f` to a member of the argument list `xi`. The expressions `(f xi) (i=1,2,...,n)` are treated as tasks, and executed on the available processors. The result of evaluating an expression on a free processor in the processor network, say `(f xi)` is collected by the function `farm`. When all the results have been received, `farm` returns its result.

Regarding the communication issue which is essential in distributed memory MIMD machines, so far the `farm` function only sends the argument function (`f`) and the value of a member of the argument list (`xi`) to a free processor in the processor network when it fires expression `(f xi)`, and it receives the result when the expression is evaluated. This might be enough for small examples. However, we have not considered the free variables which might appear in the expression to be evaluated on the processor network. In this case, we can either evaluate the free variables on the processor network, or send it to the processor network, or even evaluate some of the free variables locally and the other globally. To support this, the function `farm` needs to carry an extra argument to indicate what are going to be sent along when it fires an expression onto the network to evaluate. So we redesign the function `farm` as `newfarm`, which has the type:

```
newfarm :: (* -> **) -> [*] -> *** -> [**]
```

In this paper, we will use `farm` rather than the more appropriate `newfarm` for simplicity. Finally, we have to note that the function `farm` (and `newfarm`) is a strict function, and it stricts on all its arguments.

The other parallelism paradigms, the divide-and-conquer and pipeline and systolic array, are very difficult to control in a self-scheduling implementation regarding to the issues on workload balance and communication overhead. So, they are not dealt further in this paper. They can be represented by using annotation language Caliban[9].

6.3 Parallelism in finite element computations

In the finite element structural analysis package presented in section 4, function `map` was heavily used. To parallelise the program, we can treat them as processor farming parallelism, and use the function `farm` introduced in section 6.2 to implement the parallelism. However, it is not necessarily beneficial to parallelise all the applications of `map` because of the issues on workload balance and communication overhead. To achieve maximum efficiency, we must select potential parallelism carefully, exploiting it only when the "grain-size" is large enough. We should re-examine the finite element analysis program code given earlier in search of likely candidates.

The functions for assembling global stiffness matrix in section 4.2, assembling global load vector in section 4.3 and evaluating element stress in section 4.5 have the required form:

```
map assemble_stiffness [1..nelem]
map assemble_load      [1..nload]
map element_force      [1..nelem]
```

where, the function `assemble_stiffness` takes an element number as argument, and computes the stiffness contribution of this element to the global stiffness matrix. The computation only involves the material property and coordinates of the element. So `(assemble_stiffness ith_element)` and `(assemble_stiffness jth_element)` are independent and can be evaluated in parallel. For assembling global load vector and evaluating element stress, we have the same conclusion. Clearly, this is processor farming.

Representing this processor farming parallelism, we can use the function `farm` to rewrite the above code into:

```
farm assemble_stiffness  [1..nelem]
farm assemble_load       [1..nload]
farm elem_stress         [1..nelem]
```

Here the tasks only deal with a single element or a single loading. It might seem that the grain size of the task is still too small. Instead, we can treat these processor farming parallelisms as divide-and-conquer parallelisms. For instance, if we have a `p` processor network, we can partition the lists `[1..nelem]` and `[1..nload]` into `p` smaller sub-lists, and the original problem turns out to be solving `p` subproblems and combining the results as the solution of the original problem. By introducing function `partition`, which divides a list into a number of smaller lists, we can get the following code:

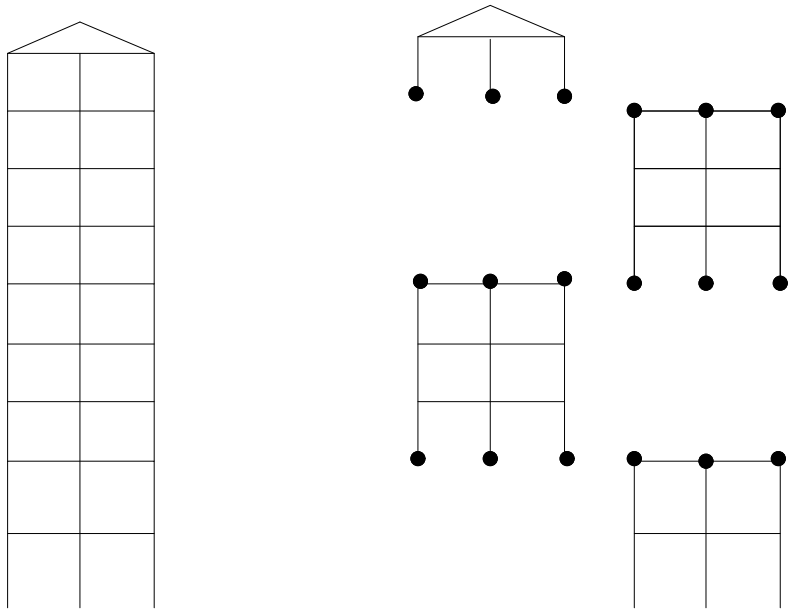
```
concat (farm (map assemble_stiffness) (partition p [1..nelem]))
concat (farm (map assemble_load)      (partition p [1..nload]))
concat (farm (map elem_stress)        (partition p [1..nelem]))
```

From above discussion, we can see that by using functional programming, it is very easy to spot the inherent parallelism in finite element computation. It is also very easy to express the parallelism, and to apply transformations to modify the grain size.

7 Parallelism in Finite Element Computation Using Substructure Technique

For large structures, solving system equations consume a great deal of computer time. As the size of the global equation system increases, solving global equation systems will soon dominate the whole system. If we leave it runs in sequential, the efficiency of the whole program running on a parallel computer would suffer a lot.

Substructure techniques have been applied extensively in the field of structural analysis



(a) Structure (b) Substructures

Figure 7: Substructure technique

for large scale problems. Using substructure technique, the finite element structural analysis can be viewed as a five stage process:

1. Divide structure into substructures as shown in Figure 7, each element belongs to one and only one substructure, and substructures are connected by the so called boundary nodes.

2. Assembling stiffness matrix and load vector for each substructure. As shown in (35), the stiffness matrix of a substructure has three parts: interior part K_{dd} , boundary part

K_{bb} and coupling part K_{db} and K_{bd} (i.e. K_{db}^T). And the load vector of a substructure has two parts: interior part f_d and boundary part f_b .

$$\begin{bmatrix} K_{dd} & K_{db} \\ K_{bd} & K_{bb} \end{bmatrix} \begin{Bmatrix} u_d \\ u_b \end{Bmatrix} = \begin{Bmatrix} f_d \\ f_b \end{Bmatrix} \tag{35}$$

3. Treat substructure as super-element and compute $K_{bb}^{'}$ and $f_b^{'}$, the contribution of each substructure to the global stiffness matrix and load vector. (38) gives the global equation system which has been assembled. Note, the global stiffness matrix and load vector here are corresponding to boundary nodes only.

$$K_{bb}^{'} = K_{bb} - K_{bd} K_{dd}^{-1} K_{db} \tag{36}$$

$$f_b^{'} = f_b - K_{bd} K_{dd}^{-1} f_d \tag{37}$$

$$K_{BB}\ u_B = f_B \tag{38}$$

4. Solve system equation (38) to get the unknown displacement vector of boundary nodes.

5. Solve system equation (39) for every substructure to get the unknown displacements of interior nodes of every substructure, and evaluate the internal forces of elements for every substructure.

$$K_{dd}\ u_d = f_d - K_{db} u_b \tag{39}$$

By using substructure technique, a large system is converted into a number of smaller ones. This will lead to efficiency improvement for large problems on sequential computers, if substructures are divided appropriately (i.e. with few boundary nodes) [4]. To solve the above problem, we can write following functional program code:

```
stiffness_ltmat
   = incrltmat initial_value (concat (map stiffness_assemble [1..processors]))
     where
     initial_value = makeltmat ndgrs_b (const 0)
```

Function `stiffness_assemble` takes a substructure number, generate the contribution of this substructure to the global stiffness matrix. The computations involved here include the computation of assembling stiffness matrix of this substructure, i.e. the stage 2 and 3 mentioned above. While function `stiffness_ltmat` returns the global stiffness matrix shown in (38). Here, lower triangular matrix is used for the data structure of global stiffness matrix, and `ndgrs_b` is the dimension of global system.

```
load_vec
   = incrvec initial_value (concat (map load_assemble [1..processors]))
     where
     initial_value = makevec ndgrs_b (const 0)
```

[4] Further efficiency improvement can be achieved if we make use of symmetry properties of a structure. For instance, some substructures may have the same stiffness, so we only need to calculate their stiffness once.

Very like `stiffness_assemble` and `stiffness_ltmat`, function `load_assemble` computes the contribution of this substructure to the global load vector. And function `load_vec` returns the global load vector as shown in (38).

After obtaining the global stiffness matrix and load vector, we can solve equation (38) to compute the displacements for those boundary nodes. The result is returned by function `t_Ub`. This fulfills the work of stage 4.

```
t_Ub = tllsolution ( stiffness_ltmat , load_vec )
```

The work at stage 5 is to compute the displacements for all the nodes and to compute the internal forces for all elements. This is achieved by function `uvw` and `element_internal_force`.

```
uvw  = incrvec initial_value (concat (map uvw_assemble [1..processors]))
       where
       initial_value = makevec (3*t_nodes) (const 0)

element_internal_force
     = incrvec initial_value (concat (map force_assemble [1..processors]))
       where
       initial_value = makevec t_elements (const 0)
```

Where, function `uvw_assemble` takes a substructure number and computes the displacements of the nodes in this substructure. And function `force_assemble` takes a substructure number and computes the element internal forces of this substructure.

On a processor network, if we take the number of substructures being equal to the number of processors, we can assign each processor responsible for one substructure, doing the work of stages 2, 3 and 5 which correspond to code:

```
map stiffness_assemble [1..processors]
map load_assemble      [1..processors]
map uvw_assemble       [1..processors]
map force_assemble     [1..processors]
```

Clearly, to represent this parallelism, we can use function `farm`(`newfarm`) mentioned in the previous section without any difficulty.

This still leave stage 4 to be done sequentially. Though the dimension of the global system equation (38) to be solved in stage 4 is much smaller (it only involves boundary nodes of the substructures), it is a full matrix and will still cost much computing resources. In the PCG method described in section 3.2.6, the main computation are matrix-vector multiplication, vector inner products and element-wise vector summations. Clearly, this can be easily parallelised. In particular, when applied into finite element computation, the global stiffness matrix is not needed to be explicitly assembled. Combining with substructure technique, K_{BB} in (38) does not required to be explicitly formed. A parallel PCG algorithm for solving the global system equations based on substructure technique is being implemented. However, this is beyond the scope of this paper.

8 Conclusion and Further Work

We have presented an example of how a functional language might be used in engineering computation. The program is not large but it does illustrate many concerns common to

much more sophisticated problems. In doing so we have attempted to present some of the general principles of the finite element method and its implementation, and to investigate the effectiveness of the functional notation in supporting such work.

The performance results under various implementations of functional languages presented in this paper are not convincing at all compared with the imperative counterpart. The main point is that functional programs often involve copying of structures, where an imperative formulation would use in-place update. Indeed were it not for this excess work being done, we suspect the performance would be more-or-less acceptable. The point is important here since the functional notation does not allow in-place update to be *specified*. A good example of this appeared in our implementation of Gauss elimination.

On the other hand, the implementation of Choleski decomposition which we presented very neatly supports the claim that the functional approach avoids the need to overspecify the computation's control flow, and the result is a formulation which matches the mathematical description of the algorithm very closely.

We experienced difficulties with debugging functional programs, and also suffered from the poor integration with graphics facilities. These should become available as the quality of available implementations improves.

The works discussed in section 6 and 7 have not been completely finished yet.

Acknowledgements

Thanks are due to our colleague S.Y. Huang and Frank Taylor, our collaborators Hugh Glaser, John Wild and Pieter Hartel on the FAST project at the University of Southampton and to Jim Cownie at Meiko Ltd. The work was funded by the UK Science and Engineering Research Council and the Department of Trade and Industry under grant numbers GR/F 35081 and GR/G 31079.

References

[1] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of ACM*, 21(8), August 1978.

[2] Richard Bird and Philip Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

[3] J. R. Bunch and D. J. Rose (ed.). *Sparse Matrix Computations*. Academic Press, Inc., 1976.

[4] G. H. Gloub and C. F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1983.

[5] E. Hinton and D. R. J. Owen. *An Introduction to Finite Element Computation*. Pineridge Press Ltd., 1979.

[6] P. Hudak, Simon-Peyton Jones, P. Wadler, and et al. Report on the programming language haskell — a nonstrict, purely functional language. *SIGPLAN Notices*, 27(5), March 1992.

[7] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2), 1989.

[8] Alan Jennings. *Matrix Computation for Enginneers and Scientists.* Addison-Wesley & Sons, 1977.

[9] Paul H. J. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors.* Pitman/MIT Press, 1989.

[10] J. M. Ortega and R. G. Voigt. Solution of partial differential equations on vector and parallel computers. *SIAM Review*, 27:149–240, 1985.

[11] N. Perry. Hope$^+$. Technical Report IC/FPR/LANG/2.5.1/7, Imperial College, 1989.

[12] Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill Book Company, 1987.

[13] G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, Inc., 1973.

[14] Philip Wadler. A new array operation. In *Proceedings of the Santa Fe Graph Reduction Workshop*, pages 328–335. Springer-Verlag LNCS 279, 1986.