# A Haskell Implementation of a Generalized Envelope Method for Sparse Matrix Factorization

Phil Grant
John Sharp
Mike Webster
Xiaoming Zhang

Department of Computer Science
University of Wales, Swansea
Swansea SA2 8PP, UK

**Abstract**

A sequential implementation in the functional programming language Haskell of a Taylor-Galerkin/pressure-correction numerical algorithm and related experimental results are reported. A generalized envelope method for Choleski factorization is used. The current implementation uses a trie structure for representing arrays to replace the ineffective built-in arrays. We demonstrate that this implementation can be used to solve practical numerical problems of limited size but a large heap space is needed to accommodate larger applications.

## 1 Introduction

Declarative functional programming has been shown to be a promising approach in overcoming the difficulties of writing large correct software systems and of exploiting the power of parallel computing. In the UK the FLARE (Functional Languages Applied to Realistic Exemplars) project is in progress with the dual aims of transferring effective functional programming technology and tools from the laboratory into the hands of serious practitioners, and demonstrating that functional languages are a suitable vehicle for effectively programming parallel machines.

As part of this project, we are using Haskell [1] to develop a computational fluid dynamics (CFD) exemplar. A sequential version of a Taylor-Galerkin/pressure-correction (TGP) finite element algorithm [2, 3], originally implemented in Fortran and used extensively by the CFD research group in the Department of Computer Science at Swansea, has been recoded into Haskell. Its parallel version is under development and will be tested on the GRIP platform [4]. The GRIP is a parallel system designed to perform graph reduction [5]. It consists of a number of processing elements (PEs), a number of intelligent memory units (IMUs) and an input-output unit. The PEs are responsible for performing reductions on the graph held in the IMUs. They are built from standard microprocessor technology, and consist of a processor, some local memory, and an interface to the bus. Generally speaking, it is a multiple instruction multiple data system (MIMD).

Haskell [1] is language designed by a committee to provide a common stable foundation for the development of real applications (including large systems), and a vehicle to encourage the use of functional languages.

As is natural for many CFD problems our implementation makes use of arrays. The exemplar involves solving large but *sparse* systems of linear equations. Currently we have access to two Haskell compilers. In both cases, the built-in arrays in these compilers are either too slow or too space demanding. In order to make progress we have been forced to build a more efficient array representation. Another reason for constructing alternative array representations was to handle sparse vectors. This has lead to the development of a replacement Array Prelude which will cater for such structures and only store essential non-default data.

## 2 The CFD Problem and the Taylor-Galerkin/Pressure-Correction Algorithm

The CFD problem we are dealing with is the flow of an incompressible fluid which can be described by the Navier-Stokes equations

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

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

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

where $\mathbf{u}$ refers to the *velocity* of particles in the fluid, $p$ the pressure, $\mathbf{f}$ an external force acting on the fluid, $\rho$ the density, and $\mu$ the viscosity.

In order to solve (1) and (2) numerically, we use the TGP algorithm [2, 3] which discretises the equations into four matrix subproblems, each calculating a time increment of a solution vector:

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

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

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

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

where $n$ is the index of the time-step.

In (3)–(6), matrices $M$, $S$ and $K$ are symmetric, sparse, banded and positive definite. The right-hand-sides of the equations are given in simplified forms; the actual computations involved are however significant especially in the case of $b_1$. For solving equations (3), (4) and (6), it is appropriate to use a Jacobi iteration [6, 7] approach which does not require the actual assembly of matrices $M$ and $S$, and can significantly reduce space resource requirements [8]. For solving (5), we employ a direct Choleski method [9, 10] which factorizes the fixed matrix $K$ into $LL^t$ form, where $L$ is a lower triangular matrix and $L^t$ its transpose. A generalized envelope method suggested by Liu [11] is an efficient storage scheme for sparse Choleski factorization combined with corresponding factorization and equation solving algorithms. The factorization operation needs only to be done once at a setup stage of the algorithm.

The equations (3)–(6) determine in general a sequential computation order of solution although the computation of $b_1^{(n+1)}$ in equations (3) and (4) can be partially overlapped with solving equation (6) because of the availability of $P^{(n)}$. However, for solving each individual equation, parallelism is possible. The discussion of parallelizing the computations of right-hand-sides and of the Jacobi iterations involves specific knowledge about the assembly processes of $b_1$, $b_2$, $b_3$ and system matrices $M$ and $S$. A guidance to parallelizing the solution of matrix problems in associations with the finite-element method can be found in [12]. Hawken *et al.* [13] have identified the following aspects of parallelism for this algorithm:

- parallel treatment of each solution component per fractional stage equation,
- parallelism afforded by the scanning over all finite elements to provide the necessary matrix-vector products at the element level, aside from the obvious fine-grain finite element level where contributions to each element must be evaluated separately,
- some limited parallelism embedded in the forward/backward component of the Choleski method [14],
- parallelism extracted from the iterative solution procedure over each nodal component unknown,
- possible division of the problem into subproblems, each defined on a separate spatial subdomain, and time-stepping with possibly different time-steps for each subproblem in parallel.

The parallelism in the factorization component of the Choleski method has been discussed in [15]. Other parallel oriented direct banded system solvers can been found in [16, 17].

In this paper, we place our main emphasis on the sequential implementation of this method and its experimental comparisons with other approaches in the context of functional programming.

# 3 Choleski Factorization

## 3.1 Choleski factorization and existing storage schemes

This direct method is employed to solve the linear equation (5). For any positive definite symmetric matrix $K$ there is a real non-singular matrix $L$ such that:

$$LL^t = K$$

The entries of the lower triangular matrix $L$ are given by

$$l_{ij} = (k_{ij} - \sum_{p=1}^{j-1} l_{ip} l_{jp})/l_{jj}, \quad j < i \tag{7}$$

$$l_{ii} = (k_{ii} - \sum_{p=1}^{i-1} l_{ip} l_{ip})^{1/2} \tag{8}$$

Construction of $L$ is only performed *once* at the outset of the time-stepping procedure. The solution of an equation $K\mathbf{x} = \mathbf{b}$ can then be determined in two steps:

$$L\mathbf{y} = \mathbf{b}, L^t\mathbf{x} = \mathbf{y}$$

As $L$ is a triangular matrix this greatly simplies the operations involved.

We now consider the storage of sparse matrices. First, we introduce some definitions. When considering the structure of a sparse symmetric matrix $K$, we may define its *semi-bandwidth* as

$$semi - bandwidth = \max_{1 \leq i \leq N} \{b_i\}$$

where $N$ is the number of rows in $K$ and $b_i$ is the difference between $i+1$ and the column index of the first non-zero entry in row $i$ of $K$. Later in this paper, for simplicity, we use the term *bandwidth* or *fixed bandwidth* to mean semi-bandwidth. Note that a single bandwidth is applied to all rows of $K$. The definition of variable bandwidth is usually taken to be

$$variable\ bandwidth = \{(i, b_i) | 1 \leq i \leq N\}$$

This means every row in the $K$ has its own band rather than sharing a maximum value. The *envelope*, or *profile*, is defined by

$$envelope = \sum_{i=1}^{N} b_i$$

In this paper, we also use the term *the envelope structure* to mean the collection of column indices of the first non-zero entry in each row of $K$.

From equations (7) and (8) it is not difficult to prove that $L$ has the same envelope structure as $K$. This property leads to two standard sparse storage schemes for $L$:

1. fixed bandwidth scheme: only entries in the bandwidth of $L$ are stored;
2. variable bandwidth scheme: only entries in the variable bandwidth of $L$ are stored. Obviously, the fixed bandwidth scheme is a special case here.

Because the solution of a system $K\mathbf{x} = \mathbf{b}$ will not be changed if we apply the same permutation to the rows of $K$, $\mathbf{x}$ and $\mathbf{b}$ and to the columns of $K$ (usually known as reordering), it is generally possible to reorder $K$ to obtain a new system matrix with smaller bandwidth or envelope and hence economise on storage [10]. A representative algorithm has been described by Sloan [18].

It is obvious that the above schemes exclude only the zeros outside the envelope structure or bandwidth of a matrix $L$. However only the generalized envelope method takes into account zeros that arise within the envelope itself [11].

By examining the equations (7) and (8), it is obvious that $l_{ij}$ is zero if $k_{ij} = 0$ and $l_{ip} \cdot l_{jp} = 0$, for $p = 1, 2, ..., j-1$. Such zeros are called predictable and require no numerical calculation to detect. Based on this observation, we classify the entries inside the envelope structure of a matrix $L$ into three categories:

1. non-zero $l_{ij}$ whose corresponding $k_{ij}$ is non-zero;
2. non-zero $l_{ij}$ whose corresponding $k_{ij}$ is zero (these entries are usually termed as *fill-ins*);
3. predictable zeros.

We know we cannot reduce the size of class 1. What the generalized envelope method does is to reduce the size of class 2 and excludes all entries in class 3 from storage. In this manner, the total storage requirement is reduced. In the context of functional programming, such a storage scheme is attractive because it generally results in less garbage collection activities and hence superior time efficiency.

The essence of the generalized envelope method is:

1. A minimum degree reordering [10] is first applied to the system matrix $K$ so that fill-ins that would result in $L$ are reduced.
2. The $L$ is then partitioned into block submatrices to exploit all predictable zeros.

This partition separates column $i$ and $i+1$ if and only if $l_{i+1,i} = 0$. Therefore, the partitioned matrix $L$ will exhibit the following structure

$$
\begin{array}{cc}
 & \begin{array}{ccccccccccc} \phantom{\bullet} & \phantom{\bullet} & \phantom{\bullet} & \phantom{\bullet} & i & i+1 & \phantom{\bullet} & \phantom{\bullet} & \phantom{\bullet} & \phantom{\bullet} & \phantom{\bullet} \end{array} \\
\begin{array}{c} \\ \\ \\ \\ i \\ i+1 \\ \\ \\ \\ \\ \\ \end{array} &
\left[\begin{array}{cc|ccc|cccccc}
\bullet & & & & & & & & & & \\
\bullet & \bullet & & & & & & & & & \\
 & 0 & \bullet & & & & & & & & \\
 & & \bullet & \bullet & & & & & & & \\
 & \bullet & & \bullet & \bullet & & & & & & \\
 & & & & 0 & \bullet & & & & & \\
 & & & & & \bullet & \bullet & & & & \\
 & & & & & & \bullet & \bullet & & & \\
 & & & & & & \bullet & \bullet & \bullet & & \\
\bullet & \bullet & & & \circ & & & \bullet & \bullet & \bullet & \\
\bullet & \bullet & & \bullet & \circ & & & & & \circ & \bullet
\end{array}\right]
\end{array}
$$

where $\bullet$ denotes an entry of class 1 and $\circ$ an entry of class 2. It can be proved that there are no predictable zeros within the envelope structure of each vertical block submatrix [11]. Consequently, only $\bullet$ and $\circ$ need be stored once the block envelope structure has been identified. The identification of the structure of $L$ can be achieved by establishing an elimination tree for $K$ [19], or performing a symbolic factorization on $K$.

## 4 An Implementation of Haskell Arrays

As has been mentioned earlier, we currently have access to two Haskell compilers. One is the Prototype Haskell compiler implemented by the GRASP group at Glasgow University. The other is the Chalmers Haskell-B compiler implemented by Lennart Augustsson at Chalmers University. At present, the Glasgow compiler implements arrays as lists [20]. This is not suitable for solving our large problems because of the access unefficiency. Although the Chalmers compiler arrays have constant access time, their updates are non-constant (whole arrays are duplicated when updated) [20]. Again this array implementation is not very suitable either. We have tried to do the same experiments using the built-in arrays. These experiments exhausted heap space on problems with moderately large sizes.

We have chosen radix tries as the underlying data structure. In such a scheme, data are stored only in leaves and their locations are decided by the bit-patterns of the keys. This is designed to cope with the need for efficient data access and update, and sparse data storage. Although some storage overhead is needed, an $O(log(n))$ access time can be guaranteed. To modify an array entry, only one trie branch has to be updated. Furthermore, sparsity can be achieved by pruning sub-tries. In our application, this is likely to reduce the overall storage for a sparse vector. The flexibility of the trie structure (being able to easily expand and contract) can also accommodate changes in the sparsity pattern (although this is unlikely to happen in our application).

The definition of the trie structure is given by

```
data Bin_Trie a = Null | Leaf a | Bin_Trie a :^: Bin_Trie a
```

This differs from the conventional definition in that a `Null` trie is introduced. This is designed to represent a trie of arbitrary size whose leaves all have a default value. Clearly, the default value only needs to be stored once.

The array data type is further constructed as

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

where `Maybe a` is defined as

```
data Maybe a = Nothing | Just a
```

An example of using a function, `s_def_array`, to create such an array is given below.

```
c = s_def_array (1,10) 0 [1:=2]
```

The index range of array `c` is 1 to 10. All entries in array `c` are set to `0` except the first entry, which has the value of `2`,

For convenience we have made the indices of our arrays `Int` only. The first argument of the constructor `Mk_t_Array` is designed for holding the array bounds as a pair. The second argument stores the default value. The `Maybe` type [21] is designed to handle the default. In this case, if the default is undefined then its value is `Nothing`, otherwise `Just v` where `v` is its value. The last argument stores the trie containing the array values.

As an example, we describe how the sparsity in our array data structure is utilized for storing a Choleski factor $L$ in the implementation of the generalized envelope scheme. The data type of a matrix $L$ is defined as

```
S_Array (S_Array Float, S_Array (Int,[Float]))
```

This means $L$ is stored as an array of block submatrices. Each submatrix is represented as a pair. The first component of the pair is an array for storing the diagonal entries in the block. The second component is an array of pairs each representing a row in this block. The first component of the pairs is an index indicating the envelope structure and the

| problem | geometry | mesh elements | $N$ | bandwidth | non-zeros in $K$ | envelope | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | | FB | VB | GE |
| problem-1 | cavity | 200 | 121 | 13 | 341 | 1573 | 1341 | 819 |
| problem-2 | glass oven | 352 | 209 | 16 | 605 | 3344 | 1930 | 1443 |
| problem-3 | cavity | 800 | 441 | 23 | 1281 | 10143 | 9281 | 4239 |

Table 1: Storage statistics on test matrices

second component is the list of actual non-zero entries. Note that we are likely to have lots of empty rows in a block; such rows are represented as the pair `(0,[])`, which is the default value for this array. Therefore, we create an empty matrix block by using function

```
blk = s_def_array (l,u) (0,[]) []
```

Further details of our array implementation can be found in [22].

## 5 Experimental results

To see the effectiveness of the generalized envelope (GE) method in the context of a functional programming environment, it has been tested using moderately large practical problems. The program tested contains a full implementation of the TGP algorithm. For comparison, the variable bandwidth (VB) scheme, fixed bandwidth (FB) scheme and generalized envelope scheme have been implemented in Haskell. Also, a Fortran implementation of FB scheme is used for the purpose of comparing Haskell and Fortran.

The platform for the following reported experiments is a Sun-Sparc-1 workstation.

Three reasonably large problems have been chosen (Table 1). It is seen that the generalized envelope scheme has saved about 40% storage for `problem-1` and `problem-3`, and 25% for `problem-2` when compared with the variable bandwidth scheme.

To obtain a deeper understanding of the program's time usage and heap space consumptions, evaluations have been forced at the end of fractional stage. For example, all inputs are forced before matrix assembly is started. This makes it possible for us to identify timing and heap profile information given in a profile plot. Figure 1 is such an example.

In Figure 1 two heap space consumption profiles are given obtained from experiments using `problem-3`. Since, here, we are mainly concerned with the Choleski factorization used in solving equation (5), all computation involved in solving equation (3), (4) and (6) has been omitted in the relevant experiments. The statistics shown in Figure 1 are all Choleski factorization related. Eleven repetitions of the forward/backward substitution

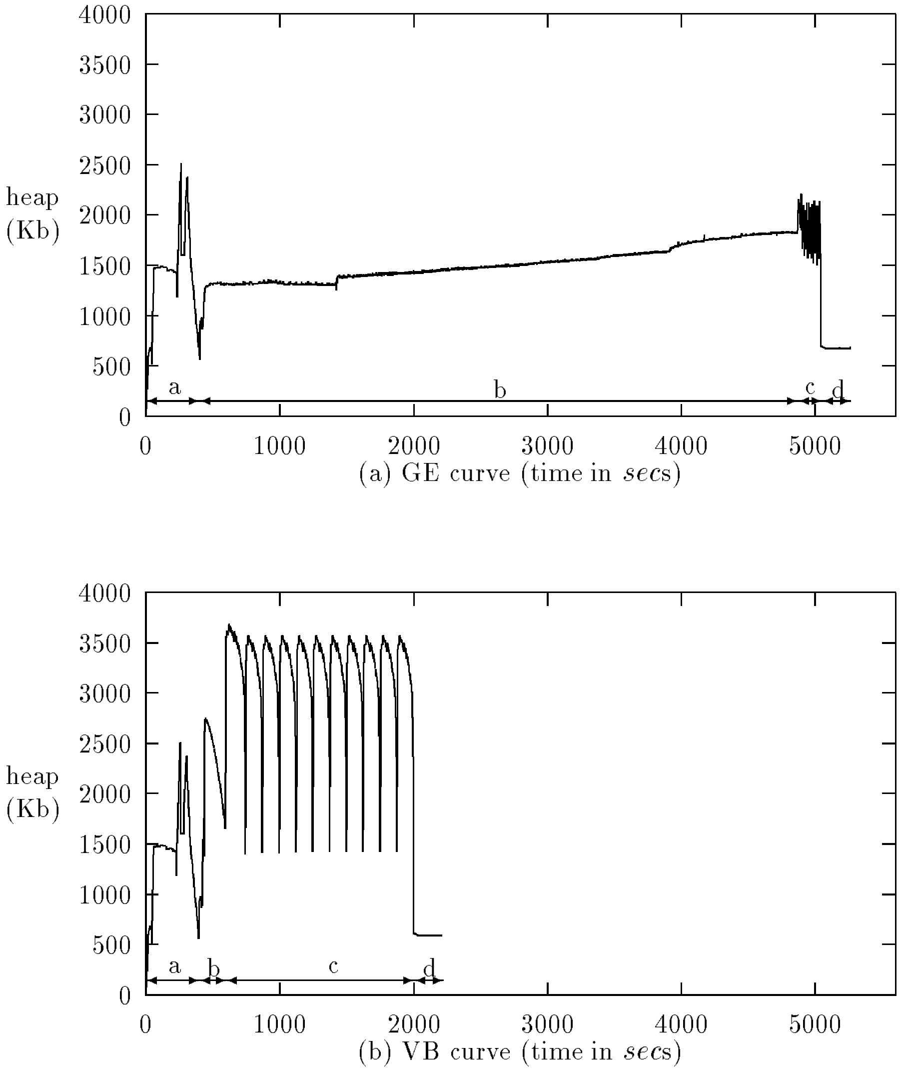


Figure 1: Heap space profiles for `problem-3` (system $K$ only)

| problem | factorization ($sec$) | | solution ($sec$) | | | max heap consumption ($Mb$) | | |
|---|---|---|---|---|---|---|---|---|
| | VB | GE | VB | GE | save (%) | VB | GE | save (%) |
| problem-1 | 19 | 160 | 4.4 | 2.3 | 47.7 | 0.61 | 0.52 | 14.8 |
| problem-2 | 36 | 543 | 10.3 | 4.2 | 59.2 | 1.06 | 0.81 | 23.6 |
| problem-3 | 196 | 4464 | 127 | 16.4 | 87.7 | 3.7 | 2.2 | 40.5 |

Table 2: Summary of resource consumptions (Choleski method)

component of the Choleski method have been performed in both generalized envelope and variable bandwidth schemes. In a real application it is not uncommon to encounter hundreds to thousands of time-steps. There are 11 time-steps involved here; this number being chosen somewhat arbitrarily to provide a clear picture.

Both curves in Figure 1 can been divided into four sections. The first sections, `a`, of these two curves have identical durations; both end at about 400 *secs*. This is actually the period when the program is reading a data file. The second section, `b`, ends roughly at about 600 *secs*) for the VB curve and at 4870 *secs* for the GE curve. In this period the program is factorizing the system matrix $K$. Because the generalized envelope scheme involves more operations, such as the minimum degree reordering and identifying the elimination tree [19], this period lasts much longer for the GE curve. The third period, `c`, ends at about 2000 *secs* and 5050 *secs* respectively. The reader can easily identify eleven equivalent ripples in the VB curve which correspond to the 11 repetitions of the forward/backward substitution process (it is believed that the sudden drop of heap space consumption at the end of each ripple is due to the forced evaluation at the end of each repetition). In this example, it is very obvious that the generalized envelope method has significantly reduced the time consumption in this period (the actual timings are given later). It is interesting to see that the generalized envelope scheme has almost halved the heap space consumption. As we have set the upper limit of the heap space to 4Mb (we can set it to a higher number, but this significantly reduces the efficiency of our Sun workstation), it is possible that the relatively long time spent in the VB case is partially due to excessive garbage collection activity. The last period, `d`, corresponds to the outputting of results.

As has been pointed out, for a large CFD problem, hundreds, or even thousands, of time-steps are common to reach say a steady-state solution. From the results illustrated in Figure 1 we can predict that each repetition of the forward/backward substitution process takes 127 *secs* using the VB scheme compared with 16 *secs* for the GB scheme. Thus assuming that we have similar setup times when the number of repetition is 40 or more, the overall time is longer for the VB scheme than for the GB scheme.

Table 2 is a summary of the statistics extracted from Figure 1 and other equivalent profile statistics (again, only the computation involved in the solution of equation (5) are relevant). The results shown are the durations of period 2 under the heading of `factorizaton`. The timings under the heading `solution` have been calculated by finding

| problem | Fortran (*sec*) | Haskell (*sec*) | | |
|---|---|---|---|---|
| | FB | FB | VB | GE |
| problem-1 | 4.3 | 91.3 | 92 | 89.6 |
| problem-3 | 16.8 | exhausted heap space | 654 | 468 |

Table 3: Timings for full TGP algorithm

the average of time difference between executing 1 time-step and 11 time-steps. The `max heap consumption` results are the maxima consumed in period 3. For the GE scheme, it is seen that as the size of the problem increases the time spent on the storage setup increases significantly. This is probably because the major operation, minimum degree ordering, has a high time complexity [10]. For the two smaller problems, the GE scheme has given us a speed-up of about 47-60%. The 127 *secs* spent by the VB scheme is seemingly too high and may not be a very meaningful result because the heap space has nearly been exhausted. For this reason no larger problem has been attempted yet.

Another set of results is presented in Table 3. In this table, average actual times spent to complete a full time-step calculation (i.e., all equation solution steps are involved) are listed. Results obtained by applying a fixed bandwidth scheme implemented in Fortran, and fixed bandwidth, variable bandwidth and generalized envelope schemes implemented in Haskell are compared. An obvious observation is that a full time-step computation takes much longer than solving equation (5) alone. In other words, much of the time is spent in solving equations (3), (4) and (6). Actually, the assembly of $b_1$ in equations (3) and (4) takes nearly half of the time. Note that, for `problem-1`, the time difference between the VB scheme and GE scheme (time saved) agrees with that in Table 2. The ratio between Fortran FB result and Haskell FB shows that Haskell is in this case about 25 times slower than the Fortran implementation. For `problem-3`, however, the time difference between the VB and GE cases does not agree with the corresponding result in Table 2. This probably points to the presence of excessive garbage collection activities in the VB scheme experiment.

Finally, a heap space profile plot for the full TGP algorithm for `problem-1` using the GE scheme is given in Figure 2. 125 full time-steps (solving (3)–(6)) have been processed and a steady-state solution has been reached. In this plot the setup time spent is clearly not excessive compared with the whole duration of the process. It also gives us the confidence that this Haskell implementation can process more time-steps if required; there is no sign of *space-leaks* [23], heap space exhaustion and computation performance degradation.

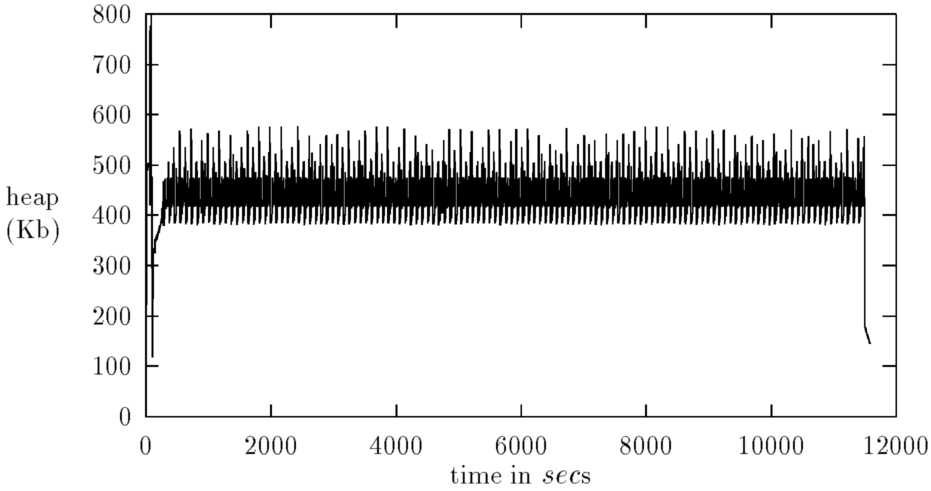


Figure 2: GE heap space profile for `problem-1` (125 full TGP time-steps)

# 6 Concluding Marks

We have concentrated on a sequential implementation in the functional programming language Haskell of a Taylor-Galerkin/pressure-correction algorithm for CFD problems using the generalized envelope method for Choleski factorization. We demonstrate experimentally that:

- It is possible to use a functional language like Haskell to solve practical numerical problems of limited size.
- Heap space resource seems to be a major difficulty which currently prevents us from attempting larger applications.
- As demonstrated by Liu in a procedural programming context [11], the generalized envelope method is also competitive in a functional environment.
- Our experience is that for the CFD exemplar considered, a Haskell implementation is about 25 times slower than a Fortran implementation. Better Haskell performance could be achieved by developing a better algorithm implementation and using an improved compiler. But to compete with procedural programming for time efficiency, the way forward is to introduce some parallelism as outlined in Section 2, which is commonly held to be easier in a functional setting than in an imperative one.

# 7 Acknowledgements

This research has been supported by grants from the UK Science and Engineering Research Council and the Department of Trade and Industry (SERC GR/F 99076 C2117).

# References

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

[2] P.Townsend and M.F. Webster. 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*, volume II, Swansea, 1987. NUMETA, Nijhoff, Dordrecht.

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

[4] S. P. Jones, K. Hammond, C. Clack, J. Salkild, and M. Hardie. Graph reduction in parallel: The GRIP parallel machine. Technical report, University of Glasgow and University College, London, Jan. 1991.

[5] S.L.P. Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall International, 1st edition, 1987.

[6] L.A. Hageman and D.M. Young. *Applied Iterative Methods.* Academic Press, London, 1981.

[7] D. Ding, P. Townsend, and M.F. Webster. Iterative solutions of Taylor-Galerkin mass matrix equations. *Int. Journal Num. Meth. Eng.*, 33:13 pages, 1992.

[8] P.W. Grant and M.F. Webster. Computational fluid dynamics exemplar. Technical Report FLARE/S/1, Department of Computer Science, University of Wales, Swansea, Feb. 1991.

[9] J.H. Wilkinson and C. Reinsch. *Handbook for Automatic Computation, Linear Algebra*, volume II. Springer-Verlag, New York, 1971.

[10] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Clarendon Press. Oxford, 1986.

[11] Joseph W.H. Liu. A generalized envelope method for sparse factorization by rows. *ACM Trans. on Math. Software*, 17:112–129, March 1991.

[12] Jagdish J. Modi. *Parallel Algorithms and Matrix Computation.* Clarendon Press. Oxford, New York, 1988.

[13] D.M. Hawken, P. Townsend, and M.F. Webster. Numerical simulation of viscous flows in channels with a step. *Computers Fluids*, 20(1):59–75, 1991.

[14] Paul Hudak. Para-Functional Programming. *IEEE Computer*, pages 60–69, 1986.

[15] Alan H. Karp. Programming for parallelism. *IEEE Computer*, pages 43–57, May 1987.

[16] J.J. Dongarra and A.H. Sameh. On some parallel banded syatems' solvers. *Parallel Comput*, 1:223–235, 1984.

[17] Ilan Bar-On. A practical parallel algorithm for solving band symmetric positive definite systems of linear equations. *ACM Transactions on Mathematical Software*, 13(4):323–332, Dec 1987.

[18] S. W. Sloan. An Algorithm for Profile and Wavefront Reduction of Sparse Matrices. *Int. Journal Num. Meth. Eng.*, 23:239–251, 1986.

[19] Joseph W.H. Liu. A compact row storage scheme for choleski factors using elimination trees. *ACM Trans. on Math. Software*, 12:127–148, June 1986.

[20] Kevin Hammond. personal communication. April 1992.

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

[22] FLARE Group. An implementation of the Haskell array prelude using pruned radix tries. Technical Report FLARE/S/7, Dept. of Computer Science, University of Wales, Swansea, March 92.

[23] W. Stoye. The implementation of functional languages using custom hardware. *PhD thesis, University of Cambridge Computer Laboratory*, Dec. 1985.