The Development of a Finite Element Algorithm in Haskell – Deliverable of the FLARE Project (Project 2117)

FLARE Group
Department of Computer Science
University of Wales, Swansea
Swansea SA2 8PP

February 9, 1993

Abstract

This is a report submitted by University College of Swansea to the SERC/DTI as part of the deliverables of the FLARE (project 2117). It describes the historical development of a numerical finite element algorithm for solving fluid flow problems.

1 Introduction to the Algorithm

The Swansea partner is involved in the development and application of a novel numerical finite element algorithm, namely Taylor-Galerkin/pressure-correction, for solving incompressible fluid flow problems. Such problems are described by a set of non-linear partial differential equations [1]. This algorithm adopts a time-stepping scheme and approximates the solution by solving the following set of equations:

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

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

$$\frac{\Delta t}{2\rho}K(P^{(n+1)} - P^{(n)}) = b_2(U^{(*)}) \tag{3}$$

$$\frac{2\rho}{\Delta t}M(U^{(n+1)} - U^{(*)}) = b_3(P^{(n+1)} - P^{(n)}) \tag{4}$$

where $M = [M_r]$, and $S = [S_r]$ are block diagonal matrices, $M_r = [m_{ij}] = [\int_\Omega \phi_i \phi_j d\Omega]$, $S_r = [s_{ij}] = [\int_\Omega \nabla\phi_i \cdot \nabla\phi_j d\Omega]$ and $K = [k_{ij}] = [\int_\Omega \nabla\psi_i \cdot \nabla\psi_j d\Omega]$. Jacobi iteration [2] is adopted for solving equations (1), (2) and (4), where explicit assembly of the system matrices is avoided and instead element matrix contributions only are dealt with. A Choleski direct method [3] is used for solving equation (3), where the complete banded system matrix K of size $m_p \times half\_bandwidth$ is used. See [4] for further details on the use of both iterative and direct solvers. The equation set has been derived by using a semi-implicit version of discrete schemes. The algorithm can also be implemented in an explicit manner in time, but then, however, it meets formidable stability restrictions for diffusion-dominated flows, as discussed in [4].

The major task was to find an efficient implementation in Haskell of the Choleski and Jacobi algorithms, and the assembly processes of the system matrices and equation right-hand-sides. More detail on the theoretical background can be found in [1]

2 Development Methodology

The Taylor-Galerkin/pressure-correction algorithm is used extensively by the Computational Fluid Dynamics research group in the Department of Computer Science at Swansea. This group uses a sequential implementation written in Fortran. To take advantage of this, it was decided to initially imitate this existing implementation in Haskell. Because the underlying algorithms originally used are not parallel algorithms, efficient parallel implementations had not been targeted initially. The development procedure was thus

1. establish a sequential prototype implementation,
2. optimise the sequential version, and
3. develop parallel algorithms.

The main programmer working on this exemplar had no previous experience of functional programming and only very limited knowledge of finite element algorithms. The above strategy of simulating the Fortran implementation eased the initial difficulties of Haskell implementation. Also, it facilitated the comparative study of Haskell and Fortran implementations, and allowed a deeper understanding of functional programming.

To date, the first two targets have been accomplished. Although some theoretical investigations into prospective parallel algorithms have been initiated, little has been done to incorporate them into the implementation. This is primarily because the GRIP parallel machine still cannot accommodate large input data sets to allow further investigation to be undertaken.

The following is a presentation of the historical development. It has been divided into four stages and three aspects are discussed for each stage:

1. issues relating to Haskell and its compilers,
2. algorithms and coding practice, and
3. use of GRIP.

3 Historical Development

3.1 Stage 1: 1/10/90 – 30/6/91

The development aims for this period were to become familiar with the Haskell language and to identify problems which might arise in implementing the Taylor-Galerkin/pressure-correction algorithm. These were achieved mainly by carrying out some simple implementations of equation solvers using trivial data structures.

Although the project started from Oct. 1990, the research assistant working on this project was not available until March 1991. Prior to this, work was conducted at a theoretical level.

3.1.1 Issues relating to Haskell and its compiler

In this period, much of the work involved the identification of problems and bugs in two available Haskell compilers. One was the prototype compiler provided by the University of Glasgow and the other was an interpreter provided by Yale University. The Yale interpreter was very inefficient and was therefore never put to serious use.

One bug found in the Glasgow compiler was related to accessing components of a 2D array. Another bug was related to arithmetic solutions on `Rational` numbers. Both bugs were fixed later by Glasgow.

Having expected heavy use of array data structures in future implementations, the time efficiency of Haskell arrays offered by the Glasgow compiler were investigated. It was found that the time complexity of accessing an array entry was not constant (proportional to the size of the array). It was later revealed by Glasgow that this was because the arrays were actually implemented by concrete list data structures. Such a performance was not acceptable for our exemplar.

Investigation into the possibility of utilizing the `Rational` data type in Haskell was also carried out. This was because such a data type can provide infinite numerical precision. However, the arithmetic operations on the `Rational` numbers were found to be very inefficient (about 17 times slower than that on Double numbers [5]). Thus their use was clearly not feasible and further investigation discontinued.

To facilitate the comparison of Haskell and Fortran implementations, it is desirable to be able to time functions inside a Haskell program. This problem was solved by calling C library timing functions from a Haskell program but the function to be timed had to be force-evaluated to make the results meaningful. Unfortunately, such an approach was not feasible for a more efficient Haskell compiler later obtained from Chalmers because interfacing C functions was not supported. The timing, for the new compiler, had to be carried out outside the program by using the UNIX `time` command.

3.1.2 Algorithms and coding practice

The implementation commenced by coding two prototype linear system equation solvers, namely, Choleski and Jacobi. The system matrices and right-hand-sides (RHSs) are assembled (explicitly or implicitly) by collecting contributions from mesh elements. This assembly process was not implemented at this stage. The implementations expect matrices and RHSs to be explicitly given in 1D and 2D arrays. There were no serious problems encounted at this stage.

3.1.3 Use of GRIP

During this period, some problems relating to GRIP were discovered. First, it was found that only single-precision floating-point computation was provided on GRIP. This could be a potential problem to our exemplar because it could make the finite element algorithm to be implemented become unstable. This was reported to Glasgow and there was no quick solution to this problem at that time. Investigation of the stability of the algorithm under the condition of single-precision calculation were instigated.

Attempts were also made to run the implemented simple equation solvers on GRIP. The programs were timed out before any result was delivered. This was later fixed by Glasgow. At the same time, Kent reported that they were also experiencing formidable problems with using GRIP. Further attempts to use GRIP were therefore postponed.

3.2 Stage 2: 1/7/91 – 31/10/91

During this period a complete implementation executable on a sequential platform was constructed. This implementation initially only performed 1 time-step of the computation but was later extended to accommodate multiple time-steps. The explicit implementation scheme mentioned in the introduction was adopted as it was simple compared to the semi-implicit one.

3.2.1 Algorithms and coding practice

The complete implementation can be divided into four major functional components: (i) I/O and control block, (ii) Choleski direct method, (iii) Jacobi iterative method, and (iv) system matrices and RHSs assembly.

This implementation should be able to accommodate large data sets. Hence it had to be both time and space efficient. Compared with the prototype implementation, two other main issues had to be considered.

1. **Choleski factors.** Because the Choleski method requires that the system matrices be assembled in advance, it is important to employ a space efficient storage scheme for these matrices. A generalized envelope storage scheme due to Liu [6] was thus implemented for storing Choleski factors. It is novel and space efficient. No zero entries in the matrices are stored. But it requires an initial application of a minimum degree ordering algorithm [7] to rearrange a given matrix. A row oriented data structure (array of arrays of lists) was used for the implementation. Such an implementation may have a weakness. The Choleski method applies both row and column oriented operations to the matrices. Column operations obviously do not fit in naturally with the way data is represented.

2. **System matrices and RHS assembly.** The RHSs in equations (1), (2) and (4) have to be assembled at run-time and the same assembly procedure is used to get all the system matrices. This process was not considered in the prototype implementation. This assembly process involves two phases:

   (a) computation of the local (element) contributions, and

   (b) accumulation of element contributions into a global vector (or matrix) although the specific operation in each phase varies according to the place it is applied.

   Because the same process is applied at various places, it is beneficial to express it as a high order function:

   ```
   assemble fn fe list_of_element_info
   ```

   where `fn` and `fe` are two functions performing phase (a) and (b) operations respectively. This strategy was adopted later.

The implementation was considered a success at that stage. It successfully produced solutions for a trivial 8-element problem and the solutions were consistent with those produced by a Fortran implementation. The Glasgow prototype compiler, which could only produce executable code for Sun3 workstations, was used. The major problem found in this implementation was that it was extremely inefficient both in time and space. This was caused by a naive implementation of the minimum degree ordering algorithm used in the generalized envelope method. By definition, the minimum degree ordering recursively applies the following process to remaining matrices:

1. find the row with minimum degree and label it as the ith row,
2. use the selected row as a pivot to perform a symbolic Gaussian elimination, and
3. remove the selected row (and column).

This was implemented in a straightforward way. As a result, it took about 8.4 hours to decompose a 25×25 matrix arising from a 32 element problem. Tests on larger problems resulted in heap overflow. These problems were solved later to a greater extent by optimising the implementation.

3.2.2 Use of GRIP

During this period, GRIP was still unreliable. Attempts at running a simple program to construct a trivial array on GRIP were unsuccessful.

3.3 Stage 3: 1/11/91 – 30/4/92

The objectives for this period were to optimize the existing implementation and carry out further code development if necessary. As a result, a much more efficient implementation was established. Hundreds of time-steps of simulation have been performed for an 800 element problem; this means, at least, that there were no significant space leaks. The execution of 5 time-steps of the 200 element problem was completed within 1000 seconds on a Sun Sparc1.

3.3.1 Issues relating to Haskell and its compilers

The Glasgow prototype compiler could only generate executable code for Sun3 workstations. Both the compilation and the execution of the resulting executables were slow. This hindered the development of our exemplar. At the beginning of this period, another Haskell compiler, hbc, became available from Chalmers University. It was much more efficient than the Glasgow compiler, and it produced Sun Sparc code. Experiments showed that the code produced by hbc was about 5 times as fast (on Sun Sparc) than their counterparts (on Sun3). The hbc was thus used for further code development.

The single-precision floating-point problem discovered in Stage 1 was investigated. The test of the algorithm stability was done by constructing a Fortran single-precision implementation. It was found that single-precision computation was enough for the convergence of solutions in most cases if the semi-implicit scheme was used [8].

3.3.2 Algorithms and coding practice

To make sure that the implementation gave numerically correct results, the results produced by the Haskell and Fortran implementations in double precision were compared. The average difference between the two was about 10^{-8} or less, which was acceptable [8].

The implementation was extended to enable multiple time-step executions. Some work on graphically displaying results using Haskell was also carried out.

Much effort was devoted to the optimization of the implementation. These efforts are listed below:

1. Make more use of higher order functions, such as `foldl` and `iteration`. This however did not improve the time efficiency of implementation in any obvious way. The reason could be that the implementations of the higher order function themselves were not optimised. No more time was spent with this investigation.

2. It was found that the original way of delivering an output `String` was not space efficient. The function which generates output was written in the following form:

   ```
   foo r =
     if condition
     then r
     else foo (r++result)
     where result = ...
   ```

   where `r` and `result` are output `String`s produced by previous iterations together with the current one. This approach caused significant space leaks [9] which prevented us from performing multiple time-step executions. This problem was solved simply by using a definition of the form:

   ```
   foo =
     if condition
     then []
     else result ++ foo
     where result = ...
   ```

   In this version, parameter passing is not involved in delivering results.

3. Space leakage was seen as a commonly occurring problem in various parts of the implementation. For example, forcing the evaluation of all input data before starting the major computation reduced the total execution time in one experiment by about 25%. A space-leak prevention scheme proposed by Runciman [10] was thus implemented.

4. An efficient algorithm for implementing minimum degree ordering described in [7] was adopted. It reduced the execution time for decomposing a 25×25 matrix on a Sun3 with 12M memory from 8.4 hours to 84 seconds. This was using Glasgow prototype compiler. Even if the time complexity of the new algorithm is one order of magnitude smaller than that of the older one, it is still difficult to explain why the improvement was so significant. Other possible reasons includes heap exhaustion and swapping.

5. As mentioned earlier, for the Glasgow prototype compiler, the built-in arrays are implemented by ordinary lists. This is definitely not usable for our exemplar. On the other hand, although the element access complexity for hbc arrays is constant, element update operation is not efficient. The inefficiency of built-in arrays was found to be the major problem which prevented us from running on larger test data sets. For example, the code produced by hbc running on the 200 element problem resulted in heap overflow. This forced us to build an implementation of arrays of our own using tree data structures [11]. A sparse scheme was used in the implementation because most of the data is of a sparse nature. The performance of the complete program was then significantly improved. It resulted in the successful program execution on the 200 element problem. By the end of the period, the time spent on the execution of 5 time-steps of the 200 element problem was reduced to less than 1000 seconds (on a Sun Sparc1). Details of the programming experience and experimental results appear in [12].

3.3.3 Use of GRIP

Having constructed an efficient and robust implementation of the algorithm, it was then tried on GRIP. Unfortunately it failed. The problem was that GRIP could not allocate enough space to run the program. Glasgow later solved this problem by tailoring the library so that only functions actually called were compiled. Although this temporary measure worked for this version of the software, another problem arose: the inability to read an input file. For some months, an extra Haskell module to define all input data had to be used. This enabled us to run a trivial 8 element problem on GRIP. Because this approach is only feasible for small data sets, serious experiments on GRIP were not possible. Glasgow promised that these problems would disappear after they ported a new Haskell compiler onto GRIP.

3.4 Stage 4: 1/5/92–31/12/92

Because of the problems encountered in using the GRIP, work had to be redirected from investigating parallel implementations, as original planned, to other issues in sequential functional implementations. The main emphasis was on efficient representations of Choleski factors and the assembly process mentioned earlier. A preliminary naive parallelism experiment was also carried out on GRIP using the 8 element problem [13]. On the other hand, after expanding the memory on our Sun Sparc1 workstation to 40M bytes, the program was successfully used to simulate an 800 element problem and the time needed to perform 1 time-step was about 5.5 minutes. This is less than 10 times slower than our Fortran implementation.

3.4.1 Issues relating to Haskell and its compilers

At the end of this period, Glasgow announced a new Haskell compiler. It produces C code and, hence, can produce executable for various platforms. In our case, this new compiler produced code which was about 30% faster than that produced by hbc 0.998.7 (sequential execution on a Sun Sparc).

3.4.2 Algorithms and coding practice

Two main issues relating to alternative implementations of the Choleski factor representation and the assembly process were investigated. The results are presented in [14] which has been submitted as a paper to an ACM SIGPLAN/SIGARCH Conference on Functional Programming Languages and Computer Architecture to be held at Copenhagen in June, 1993.

In [15], Wise suggests the use of Quadtrees for matrix representation in functional programming. Our main purpose was to contrast the Quadtrees and the generalized envelope method under various matrix ordering schemes. Quadtrees have been found to be generally more time efficient and less sensitive to the sparsity structures of Choleski factors.

As described earlier, the assembly process can be divided into two phases. In a procedural implementation, this is usually done by looping over all elements and, in each step, accumulating the obtained element contributions into the corresponding nodal entries in a global vector. An alternative approach is to loop over mesh nodes and collect all contributions to a specific node in one go. In our functional implementation, it was found that the first approach requires about four times as much space as the second one although it is slightly more efficient in time (with the presence of lazy evaluation).

3.4.3 Use of GRIP

It was found that the parallel implementation on GRIP produced significantly different numerical results in different runs. It is hoped that the new Glasgow Haskell compiler would solve this problem.

4 Summary

The history of developing a FLARE finite element exemplar is described. Generally speaking, the attempt has been successful. This is the first reported complete functional implementation of a numerical algorithm which can be used to solve sizable realistic problems. Furthermore, the essential part of the Haskell code is less than two thousand lines which compares favourably with the Fortran code of seven thousand lines. This contrasts with the results reported by Gelernter [16]:

> Professor Arvind's group conducted an experiment in which they translated a two-thousand line numerical Fortran code into a functional language. The result was three thousand lines long, and the experiment convinced them that the complexities of dealing with large data structures made functional languages unworkable for this kind of application.

Our experience confirms some commonly recognized advantages of functional programming including expressiveness and ease of maintenance. However we have not found any advantage in using lazy evaluation. Much effort had to be put into eliminating it to free space rather than utilizing it. A more detailed description of our experience can be found in [14].

References

[1] 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.

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

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

[4] 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.

[5] J.A. Sharp. Matrix multiplication in Haskell. Technical Report FLARE/S/2, Dept. Computer Sci., Univ. of Wales, Swansea, March 1991.

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

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

[8] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Some precision comparisons of Fortran and Haskell implementations. Technical Report FLARE/S/8, Dept. of Computer Science, University of Wales, Swansea, April 92.

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

[10] C. Runciman. TIP in Haskell – Another Exercise in Functional Programming. Technical Report FLARE/Y/3, University of York, 1991.

[11] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. 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.

[12] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. A Haskell implementation of a Generalized Envelope method for sparse matrix factorization. In *Proc. ATABLE-92*, Montreal, June 1992.

[13] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Functional programming for a computational fluid dynamics problem. In *Proc. Computational Mechanics in UK*, Swansea, UK, Jan. 1993. Association for Computational Mechanics in Engineering (UK).

[14] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Some issues in a functional implementation of a finite element algorithm. Technical Report FLARE/S/13, Dept. of Computer Science, University of Wales, Swansea, Dec. 92.

[15] David S. Wise. Matrix algorithms using quadtrees (invited talk). In *Proc. ATABLE-92*, Montreal, June 1992.

[16] D. Genernter. Domesticating parallelism. *Computer*, Aug. 1986.

[17] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. A naive solution of some matrix equations in Haskell. Technical Report FLARE/S/3, Dept. Computer Science, Univ. of Wales, Swansea, May 91.

[18] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Brief progress summary. Technical Report FLARE/S/4, Dept. of Computer Science, University of Wales, Swansea, June 91.

[19] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. A walkthrough of Jacobi iteration in Haskell. Technical Report FLARE/S/5, Dept. of Computer Science, University of Wales, Swansea, July 91.

[20] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Some timing tests and program modifications. Technical Report FLARE/S/6, Dept. of Computer Science, University of Wales, Swansea, Dec 91.

[21] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. An implementation of Quadtree for storing choleski factors. Technical Report FLARE/S/10, Dept. of Computer Science, University of Wales, Swansea, Aug. 92.

[22] P.W. Grant, J.A. Sharp, M.F. Webster, and X. Zhang. Analysis of an assembly process for implementating a finite element algorithm. Technical Report FLARE/S/12, Dept. of Computer Science, University of Wales, Swansea, Dec. 92.