Functional Programming for a Computational Fluid Dynamics Problem

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

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

1 Introduction

In the recent years we have seen an explosion in the interest in, and availability of, parallel processors and a corresponding expansion in application programming activity. Many researchers feel that conventional imperative languages like Fortran and C are inadequate for expressing parallelism since these languages are intrinsically tied to the sequential von Neumann machine model and hence lay too large an emphasis on the underlying machine architecture. It makes more sense to use a language with a non-sequential semantic base. *Functional languages* are usually believed to be good candidates. Hudak [1] points out that functional languages are not oriented towards any particular machine model and they are easier to compile for parallel machine execution than conventional imperative languages because imperative languages are biased towards a sequential machine model. It is thus possible to write and debug a functional program on a sequential machine, and then execute it on a parallel machine for improved performance.

A functional program consists of a set of mutually recursive functional equations which mathematically define the named functions. The orderings of the equations does not alter the semantics. In a functional language no side effects are permitted. This means that a program is guaranteed to give the same result no matter what computation order is chosen in executing the program (assuming termination). It thus implicitly supports parallelism. There is generally no need for special message-passing constructs or other communication primitives, and no need for special "parallel" constructs such as "parbegin ... parend" [2]. Functional languages are believed to have the following elegant features.

- Expressiveness: They help to generate concise code. One particularly important example is the possibility of constructing and manipulating *infinite data structures* and *streams*. This is because implementing *lazy evaluation*, postponing evaluation until it becomes unavoidable, is much easier in functional languages than in imperative languages [3]. The price of lazy evaluation is, however, reduction in execution speed. Another important feature is that functional programming allows the use of higher-order functions, functions which take functions as arguments or deliver one as a result. For example, `map` is a higher-order function commonly provided in functional languages, which applies a function to each entry in a list and returns a new list of these values.
- Ease of maintenance: They are said to be more suitable for developing large systems than procedural languages as functional programs are believed to be easier to maintain. The lack of side-effects means that a change in any part of a program cannot have an obscure, unpredictable

effect elsewhere.

There are many researchers working on the parallel implementation of finite element methods using the procedural approach. Such work is illustrated in the following:

- Parallelism for distributed array processors: This work is mainly on the parallel implementation of the assembly process and the parallel solution of the associated equations [4].
- Parallelism in the adaptive composite grid method (FAC): Lemke and Quinlan [5] have developed a variation of FAC that employs a multigrid method, namely asynchronous FAC, designed for distributed architectures.
- Domain-parallelism (domain decomposition): This approach is usually employed as a method for solving large scale problems on Multiple Instruction Multiple Data machines (MIMD) [6, 7]. Bennighof and Wu [8] also suggest that subdomain computation for multiple time steps should be performed independently on each subdomain, to relax synchronization requirements, before unifying results on the complete domain at a set time.
- Time-parallelism: Solving time-dependent partial differential equations through time-stepping has a sequential nature to it. However, Horton [9] demonstrates that it is possible to generate simultaneous solutions of several time steps using a time-parallel multigrid method.

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 [10] to develop a computational fluid dynamics (CFD) exemplar [11, 12]. In this paper, we outline our preliminary work for parallelizing a Taylor-Galerkin/pressure-correction algorithm on a MIMD platform.

2 Taylor-Galerkin/Pressure-Correction Algorithm

A Taylor-Galerkin/pressure-correction finite element algorithm [13, 14] is used to solve incompressible fluid flow problems described by the Navier-Stokes equations. The equations are first semi-discretised in time, introducing a new set of fractional equations over each time interval $[t_n, t_{n+1}]$, and then subsequently solved in space for each such interval. The spatial domain over which the solution is desired is triangulated in 2D into a finite element mesh. External forces are ignored in the discussion below.

To semi-discretize equation (1), we introduce field variables $\mathbf{u}^{(n)}$ and $p^{(n)}$ at discrete time t_n where the time interval is Δt. We need to obtain a set of difference equations which we can use to solve for $\mathbf{u}^{(n+1)}$ and $p^{(n+1)}$ in terms of $\mathbf{u}^{(n)}$ and $p^{(n)}$. We present here the more robust semi-implicit version of these semi-discrete schemes with fluid density ρ and viscosity μ,

$$\frac{2\rho}{\Delta t}(\mathbf{u}^{(n+1/2)} - \mathbf{u}^{(n)}) = s(\frac{\mathbf{u}^{(n+1/2)} + \mathbf{u}^{(n)}}{2}) + c(\mathbf{u}^{(n)}) - \nabla p^{(n)} \tag{1}$$

$$\frac{\rho}{\Delta t}(\mathbf{u}^{(*)} - \mathbf{u}^{(n)}) = s(\frac{\mathbf{u}^{(*)} + \mathbf{u}^{(n)}}{2}) + c(\mathbf{u}^{(n+1/2)}) - \nabla p^{(n)} \tag{2}$$

$$\frac{2\rho}{\Delta t}\nabla \cdot \mathbf{u}^{(*)} = \nabla^2 p^{(n+1)} - \nabla^2 p^{(n)} \tag{3}$$

$$\frac{2\rho}{\Delta t}(\mathbf{u}^{(n+1)} - \mathbf{u}^{(*)}) = \nabla p^{(n)} - \nabla p^{(n+1)} \tag{4}$$

where $c(\mathbf{u}) = -\rho\mathbf{u} \cdot \nabla\mathbf{u}$ is a convection term and $s(\mathbf{u}) = \mu\nabla^2\mathbf{u}$ is a diffusion term.

In the second step, we use a Galerkin finite element method to obtain fully-discretised equations. The mesh may consist of triangular elements and for each node j we have functions $\phi_j(\mathbf{x})$ and functions $\psi_j(\mathbf{x})$ which are piecewise quadratic and linear in $\mathbf{x}$ respectively. The important property of these functions is that they are unity at the node j and zero at all the other nodes of the mesh, thus imbuing them with local compact support property in the element region comprising of those elements that share node j. We express the finite element approximation to our solution $\mathbf{u}^{(n)}(\mathbf{x})$ in terms of the ϕ_j's through $\mathbf{u}^{(n)}(\mathbf{x}) \approx \sum_{j=1}^{m_v} \phi_j(\mathbf{x})\mathbf{U}_j^{(n)}$ for m_v nodes of the mesh, and $p^{(n)}(\mathbf{x}) \approx \sum_{j=1}^{m_p} \psi_j(\mathbf{x})\mathbf{P}_j^{(n)}$, m_p vertex nodes. By substituting these interpolated solutions in equations (1)–(4) and integrating over the spatial domain Ω, we obtain the set of fully-discretized equations:

$$(\frac{2\rho}{\Delta t}M + \frac{\mu}{2}S)(U^{(n+1/2)} - U^{(n)}) = b_1(U^{(n)}, P^{(n)}) \tag{5}$$

$$(\frac{\rho}{\Delta t}M + \frac{\mu}{2}S)(U^{(*)} - U^{(n)}) = b_1(U^{(n+1/2)}, P^{(n)}) \tag{6}$$

$$\frac{\Delta t}{2\rho}K(P^{(n+1)} - P^{(n)}) = b_2(U^{(*)}) \tag{7}$$

$$\frac{2\rho}{\Delta t}M(U^{(n+1)} - U^{(*)}) = b_3(P^{(n+1)} - P^{(n)}) \tag{8}$$

where $M = [M_r]$, $S = [S_r]$ 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 $K = [k_{ij}] = [\int_\Omega \nabla\psi_i \cdot \nabla\psi_j \, d\Omega]$.

The above approach may be implemented in an explicit manner in time, but then however it meets formidable stability restrictions for diffusion-dominated flows, as discussed in [14]. Jacobi iteration is adopted for solving equations (5), (6) and (8), where we avoid the explicit assembly of the system matrices and instead deal only with element matrix contributions for m_e number of elements. A Choleski direct method is used for solving equation (7), where we deal with the complete banded system matrix K of size $m_p \times half\ bandwidth$. See [14] for further details on the use of both iterative and direct solvers.

3 Haskell and a GRIP Parallel Machine

Haskell is a committee-designed lazy pure functional language 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. Some preliminary tests of the functional implementation have been undertaken on the GRIP platform [15], which is an experimental parallel computer designed to perform graph reduction. Graph reduction is an implementation technique for functional languages [3]. The GRIP 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, and are built from standard microprocessor technology consisting of a 12MHz 68020 processor, 1M local memory, and an interface to the bus. The PEs share up to 20M-word global memory and the system is of MIMD type.

The machine manages a *thread pool*, a thread being a sequential computation whose purpose is to reduce a particular subgraph to normal form. A processor in search of work fetches a new thread from the thread pool and executes it. Initially there is only one thread in the pool whose job is to evaluate the whole program. A thread however can split its work by *sparking* child threads to the thread pool.

For the moment, the sparks can only be generated by embedding specific parallel function calls in a program, though this is usually rather easy. For example, let `l` be the list defined by `l=[1+2,3+4,5+6]`. Normally this declaration causes a sequential evaluation of the list entries on the GRIP. To evaluate the entries in parallel, we need only include one line of code in the program: `res=plist l`, where `plist` is a built-in primitive which causes `l` to be evaluated in parallel. Evaluating `res` generates three threads in the thread pool, where each points to a list entry.

4 Experimental Results and Comments

We have completed a Haskell implementation of the Taylor-Galerkin/pressure-correction algorithm. The essential part of the code is under 2000 lines (2D case). In this implementation, as yet there has been no consideration given to parallelism related issues such as granularity and adopting parallel algorithms. As a result, the parallel performance is still open to optimisation.

The parallelism on the GRIP is mainly achieved by using the built-in primitive `parlist`, a generalized form of the `plist` function, at various places in the program. As a result, the evaluations of nodal vectors, like equation right-hand-sides and iterates, are parallelized. Because the GRIP system as a whole is still under development, at present we can only run our code on an 8-element trivial test problem.

To contrast the functional approach and procedural approach, we experimentally compared the Haskell implementations (on the GRIP, a Sun Sparc-1, a Sun3/80) and a Fortran implementation (on a Sparc-1). The space resource used is not considered as an issue in this paper. It is almost certain that a functional implementation uses much more space in an accumulative sense than a procedural implementation. This is because in-place value updates are not allowed in functional programming and new values must be physically stored in another place. This matter is further complicated by the presence of lazy evaluation because unevaluating expressions usually take a lot of additional space. The following is a comparison of execution speed in terms of average time spent on computing just one complete time-step.

Language	Fortran	Haskell					
Platform	Sparc-1	Sparc-1	Sun3	GRIP			
# processors	1	1	1	1	3	6	9
Speed (`sec`)	0.23	3.4	15.5	86.3	39.0	30.7	23.0

We point out that the maximum number of PEs used is small, as at present there are only 9 PEs available on the GRIP (the design is scalable up to 80 PEs).

First of all, the Haskell implementation is seen to be about 15 times slower than a Fortran implementation on a Sun Sparc-1 sequential platform. The program has been coded in such a way that the solution at each stage, equivalent to equations (5)–(8), is not started until the solution of its predecessor is completed. Also, only 3 iterations are performed in each Jacobi solution. Therefore, the time taken to solve one complete time-step in parallel can be described by $f = \alpha\lceil\frac{m_v}{p}\rceil + \beta\lceil\frac{m_p}{p}\rceil + \gamma\lceil\frac{m_e}{p}\rceil$ and f reaches its minimum when $p = \max(m_v, m_p, m_e)$. In this case for $m_v = 25$, $m_p = 9$ and $m_e = 8$, the runtime performance observed fits into the trend described by f.

5 Conclusion

A functional parallel implementation of a Taylor-Galerkin/pressure-correction algorithm has been reported. It has been tested on a GRIP parallel machine and the performance is compared with that of a Fortran serial implementation using a trivial problem. It has been observed that the parallel performance fits into an expected trend, though there is a need to investigate larger problems to establish more clearly the advantage of parallelisation. A merit of functional programming includes the relative simplicity of parallelisation, achieved here using a simple strategy and then run on various machine configurations.

6 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] Paul Hudak. Para-Functional Programming in Haskell. In B. K. Szymanski, editor, *Parallel Functional Language and Compilers*, New York, 1991. ACM Press.

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

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

[4] C.H. Lai and H.M. Liddell. A review of parallel finite element methods on the DAP. *Applied Mathematical Modelling*, 11(5), Oct. 1987.

[5] M. Lemke and D. Quinlan. Fast adaptive composite grid methods on distributed parallel achitectures. *Communications in Applied Numerical Methods*, 8:609–619, 1992.

[6] G. Fen Paw and D.R. Owen. Domain decomposition procedures for finite element analysis on transputer based parallel computers. *Computing Systems in Engineering*, 2(5/6):451–460, 1991.

[7] I. St. Doltsinis and S. Nolting. Generation and decomposition fo finite element models for parallel computations. *Computing Systems in Engineering*, 2(5/6):427–449, 1991.

[8] J.K. Bennighof and J.-Y. Wu. An unconditionally stable parallel transient algorithm with multi-step independent subdomain computation. *Computing Systems in Engineering*, 2(2/3):217–230, 1991.

[9] G. Horton. The time-parallel multigrid method. *Communications in Applied Numerical Methods*, 8:585–595, 1992.

[10] P. Hudak, S.L. 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] S.L. Peyton 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.