Functional Programming applied to real applications in Science and Engineering

Case for Support

A submission to the Systems Architecture Committee of the SERC.

Purpose

Many of the problems in scientific computing are conveniently specified in a mathematical style; in addition, the computational requirements are such that they are a fruitful application area for the use of parallel processing. Functional languages have a distinctly mathematical flavour; they also claim a number of advantages in exploiting parallel machines.

This proposal is to investigate this relationship. Some work on the problems has already been completed. Further work is planned, and detailed work on parallelism will be undertaken.

The objectives are thus:

- To investigate further the expressive power of functional languages for solving scientific programming problems.
- To compare the merits of imperative and functional languages applied to scientific algorithms.
- To provide a demonstrator application.

Background

The FAST (Functional Programming for Arrays of Transputers) Project was funded by the SERC (GR/F 35081), and was a collaboration between the University of Southampton and Imperial College, London. Meiko Ltd of Bristol was the industrial uncle. It began in August 1989, and has just finished in February of this year.

The objectives of the project were to investigate the technology to exploit loosely-coupled multi-processors using functional languages. The model for exploitation was to use a process description language (Caliban [?]) to provide an abstraction that would allow coarse-grain parallel processes to be specified conveniently. The processes themselves were to be compiled from functional language source using a compiler that performed extensive analysis. The responsibility for the Caliban work was largely sited at Imperial College [?], and the sequential compiler largely at Southampton [?, ?].

At Southampton a compiler was constructed which took a number of possible functional language inputs (including interfacing to a Haskell subset front end written at Imperial), and generated C as the "machine code". The C object code is combined with a run-time system which thus provides an implementation that can be run on any Unix workstation [?].

The performance of both the compiler and the code compares well with other functional language implementations currently available [?].

Scientific application work in FAST

As part of the FAST project, a small number of applications were planned to test the technology. At Southampton it proved possible to employ a researcher (Dr Bryan Carpenter) for a short time to explore some applications. Dr Carpenter has longstanding experience of simulating models from high energy and condensed matter physics. He has been involved in many parallel machine projects for Prof Tony Hey, both in design and benchmarking workpackages.

Three problems of increasing complexity were investigated: solution of a partial differential equation by the relaxation method; Monte Carlo simulation of a statistical mechanics (condensed matter) system; and a similar simulation of a lattice gauge theory. Below we comment on the style of implementation that was developed.

The algorithms implemented were all formulated on grids. The model of computation in languages such as Fortran is to use an array of values to represent the state of the computation. An obvious approach in a functional language would be to use stream of arrays, representing the history of the computation. In fact it proved more convenient to use an equivalent representation, in which the elements of a single array were lazy streams representing the history of each individual grid site. To convert between these representations, for example for printing, it is easy to use the built-in higher order functions. Thus from the outset the algorithms began to exploit features such as laziness that are unique to the chosen implementation language.

Representation as an array of streams had various advantages.

Firstly it is easy to see that such streams can be viewed as interacting parallel processes, and this gives an excellent starting point from which to investigate the parallel execution behaviour of the program.

Secondly, arrays in functional languages are not yet implemented as efficiently as they might be. It turns out that the implementations used are better able to optimise the "array of streams" model than the "stream of arrays", because array-subscripting operations are required less frequently.

Thirdly many parts of the algorithms can be clearly specified. To try to give a flavour of this we consider a fragment of code from the Poisson equation relaxation program.

The history of a site depends on the history of its neighbours. For the simple case of Jacobi relaxation, the new value at a site is calculated from the values of the neighbouring sites in the previous generation. In more general schemes, such as Gauss-Seidel, the new value at a site is calculated as a combination of the previous and the latest generations. This can be implemented by colouring the sites to indicate which generation should be used. The function for updating site i is passed a stream representing the history of neighbouring site j, to which we apply the function

```
if colour j < colour i then tail else id
```

This expression returns either the function *tail* or *id*, depending on the relative colours of the sites. Its effect on the stream is to either look one place forward in the history, if necessary, by taking the tail of the stream, or to leave the stream unchanged by applying the identity function. This is the only change needed to go from the Jacobi update to the Gauss-Seidel update, with its more complex data dependencies.

The other major advantage observed was the level of abstraction facilitated by the language, and the opportunites for code reuse engendered. The problems under consideration have similar structures, but the geometries and local update vary. Careful consideration was given to the handling of the lattices to ensure the right level of abstraction. This care in separating the geometry was well rewarded. The update operations are not concerned with, and make almost no reference to, the geometry. Thus the first and second applications mentioned above shared geometry modules, but had different local update code, whereas the second and third applications had different geometry modules but shared the local update code.

Related work

Recently exciting claims have been made for performance "parity" between certain functional languages (specifically SISAL) and Fortran [?]. SISAL is an applicative language designed for scientific and numerical programming. It imports imperative-style constructs for expressing iteration and array construction, and apparently successfully exploits these features to generate code competitive with imperative languages. We have done some preliminary experimentation with SISAL, and the results have been promising.

Other work on scientific applications of functional language, including our FAST work and work in the Flare project, has concentrated on the advantages of high-level declarative languages, such as Haskell, rather then absolute performance.

SISAL (high performance, limited expressiveness) and Haskell (high expressiveness, limited performance) represent two extremes of the functional programming spectrum. Recently there have been some moves to bring these camps together in a "parallel Haskell" effort [?].

Research experience

Mr Hugh Glaser has been researching into functional language systems for more than 10 years. He is a Senior Lecturer in the Department of Electronics and Computer Science at the University of Southampton. Previously he was a Research Fellow at Imperial College, where he was Project Director of the Imperial team on the Alvey Flagship Project, under Prof John Darlington and Prof Mike Reeve. He has also been involved in other parallel programming research projects, including the programming environments workpackage of the Genesis ESPRIT project. He is currently funded under GR/H19439 (Distributed Prolog Execution.)

Prof Tony Hey is the Professor of Computation at the University of Southampton and Chairman of the Parallel Applications Centre (PAC) at Southampton. He has been a leading researcher in parallel computing for many years, being the initiator, with Dr David May of INMOS, of the ESPRIT Supernode project which delivered the T800 transputer. He has since been involved in number of projects, including the development of the parallel benchmark suite for the Genesis project.

Dr Bryan Carpenter is a theoretical physicist by training, and worked as a Research Fellow at Edinburgh and DESY in Hamburg. Since then he has worked in Computational Physics, and has been involved in the Supernode, Genesis and Puma projects in turn.

Programme

1. Further research

It is proposed to build on the FAST work on scientific applications described above, and extend it in a couple of directions.

Stream parallelism

The implementation model used has an obvious direct mapping to a parallel machine. However, to obtain useful acceleration a number of issues will need to be resolved, the most obvious of which is the question of granularity.

The "array of streams" model gives a very fine granularity, with probably unacceptable overheads. On the other hand the "stream of arrays" model gives very coarse granularity—*no* explicit process parallelism!

Compromise will be required, and some trade offs will need to be incorporated into the framework.

Grid geometry

In the earlier work some promising techniques were developed for handling grid geometries in an abstract way, based on generalised Haskell arrays incorporating "topological" information.

The problems studied so far had relatively simple geometries; they are regular and static. There are other, more general, algorithms that fit into the same general paradigm, but which are irregular or dynamic. A dynamic algorithm such as Multigrid, for example, will fit naturally into the framework already provided. The approach to lattice topology adopted in the earlier work may be generalised to incorporate "simplicial maps" for moving between grids.

Such algorithms will be investigated, and our geometric techniqes will be extended to encompass them.

2. Comparative Review of programming languages

In addition to these research avenues, we will undertake to produce a couple of more concrete outputs.

Firstly, we want to evaluate the merits and demerits of functional programming in the scientific domain, and do this in as systematic a way as possible. This will involve selecting a set of kernel algorithms, and a representative set of target languages.

The kernels will be short (less than 100 lines, say) and as varied as possible. For selection as a kernel, an algorithm should be either

a) representative—the kernels should include algorithms widely used in scientific programming—or

b) "interesting". We will try to include algorithms which, for example, involve non-trivial patterns of array update when coded imperatively.

Promising application domains from which to extract these kernels include

- Monte Carlo simulations (eg, from condensed matter physics).
- Advanced PDE solvers (eg, multigrid).
- Finite element methods.
- Computational fluid dynamics.

Each algorithm will be encoded in the target set of languages. This set will presumably include Fortran 90, SISAL, a more conventional eager functional language such as SML, and Haskell. The idea is to choose a spectrum of languages from the traditional imperative to the pure functional.

The objectives will be

1. To isolate any real problems that might exist in putting typical array-oriented problems into functional languages. Is there any important lack of expressiveness?

2. To explore the advantages of functional languages for this kind of programming. Some researchers have reported that laziness is not particularly useful in scientific programming. Our experience has been that it can be used effectively, but we need further evidence.

3. Although we do not wish to emphasize absolute performance unduly at this stage, the suite will be useful for benchmarking.

3. Full scale demonstrator application

Secondly we will produce one full-scale demonstrator application, taken from one of the domains mentioned above. This will be coded in a functional language, probably Haskell. The intention is that the program should be capable, in principle at least, of producing useful scientific data. It should have at least the same IO facilities as a typical scientific application coded in an imperative language. The demonstrator application will have an interactive graphical interface.

Resources

We have requested funding for a named individual, Dr Bryan Carpenter, whose research experience is detailed elsewhere in this proposal. His experience and abilities are ideal for the proposed research, and it is because of the quality of Dr Carpenter's work that this application is being made.

A high performance workstation has been requested which will provide the development platform for the software. This will enable Dr Carpenter to experiment with applications that require both significant compilation costs and are computationally intensive.

Some consumables associated with the project have been identified, such as magnetic media for archiving, as well as the costs of connecting the workstation to the network.

Applications, Collaboration and Exploitation

There are no collaboration or exploitation agreements in place, although the successful completion of the project will result in a system that can exploit state of the art parallel hardware to good effect. Since all the results will be in the public domain, however, direct commercial exploitation is inappropriate.