Procedural versus Declarative Programming for Scientific and Engineering Applications: The Half-Time Score

Tony Hey, Bryan Carpenter and Hugh Glaser

1 Introduction

Declarative language enthusiasts claim that there are real benefits to be gained from programming in a declarative style:

> "We believe, as did McGraw in 1984, that increased productivity, generality, utility, portability, and performance are only possible if programmers avoid the constraints of imperative languages and adopt a higher level of abstraction." *David Cann, CACM 1992*

or

> "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." *P. Grant et al, FLARE Report 1992*

or

> "Hudak points out that functional languages are not oriented to 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." *P. Hudak, 1991*

These claims seem all-embracing—not restricted to any specific application area. So we can hope to adopt functional programming when coding operating systems, telecoms applications, safety critical embedded systems, and so on. Our particular interest here is in science and engineering applications. In this article we examine the evidence for the thesis that scientists and engineers should abandon Fortran and C or C++, and switch to a functional programming language.

Claimed benefits of declarative languages

It is claimed that compilers for imperative languages are handicapped by the *assignment statement.*

```
A = F (X)
B = G (Y)
```

Analysis of this program fragment is complicated by various issues. *Aliasing* can mean that any of the variables `A`, `X`, `B`, `Y` may represent the same memory cell. Common blocks or global variables allow functions to share data—`F` or `G` could update one of the four variables as side-effect. Potential parallelism is therefore not obvious. Extracting it requires detailed interprocedural analysis. This proves too difficult for all but the simplest programs.

Functional languages on the other hand express a problem in terms of well-defined, mathematically pure, functions. The elimination of side-effects improves the clarity of code and facilitates software engineering. The simplified semantics of the languages allow formal proofs and program transformations. The level of abstraction provided by features like higher order polymorphic functions, together with the liberation of the programmer from detailed concern about memory management issues, allow rapid prototyping of applications. Finally, since there is no notion of sequencing of operations in a functional program, there is implicit parallelism—and the expression of this parallelism is not tied to any architectural details.

Many of these advantages are indisputable, but one should also ask what scientists and engineers really care about. We suggest

- Transparent relation of mathematics to programming languages.
- Portability and standards, especially for parallel code.
- Efficient programs and high performance.

Functional languages are certainly more "mathematical" than imperative ones; but it remains to be established that a typical scientific applications needs much more mathematical sophistication than "formula translation". There is an emergent standard for functional programming, in the form of Haskell; but parallel extensions or annotations to functional languages are still the subject of research rather than standardisation. The efficiency issue will be reviewed in some detail in the remainder of the article.

Perhaps there is a mismatch of concerns here. *Formal* correctness is rarely an issue in a scientific program. Detractors from declarative programming would probably claim that C and Fortran are adequate vehicles for rapid prototyping of scientific and engineering software.

The remainder of this article

Section 2 collects together some evidence concerning the relative merits of functional and imperative languages in the domain of scientific programming.

Section 3 draws together our conclusions, based on this evidence.

2 Phenomenology

In this section we selectively review some results from two major projects, which have promoted the adoption of functional languages for scientific programming. We also describe a small case study of our own, in which we have applied some declarative languages to a representative kernel algorithm.

2.1 The LLNL Sisal project

Sisal (Streams and Iteration in a Single-Assignment Language) is a functional language developed at Lawrence Livermore National Laboratory. It differs from most other functional languages in taking support of scientific computation as a primary design goal. Consequently the early implementations omit many of the advanced abstractions of purist functional languages (higher order functions, normal order evaluation, Milner-Hindly type system, and so on). These are perceived as having limited usefulness in scientific programming. Some of them incur a significant penalty in implementation efficiency, or hinder compile-time analysis.

Sisal introduces a few extra ideas not found in conventional functional languages, including "parallel" and "sequential" loops. It retains those features of functional languages which facilitate program analysis and thus optimising transformations. Programs are *referentially transparent*. A function passed a given set of argument values always returns the same value, and (as a rule) within the range of definition of a variable, its value never changes (in Sisal, this constraint is relaxed in the eponymous iterative construct).

Dispensing with the idea of a modifiable state, the concept of "data" replaces that of "memory", and "creation" replaces "update". Where an imperative program would update a portion of an array, a Sisal program logically creates a new array with different values in the modified portion.

This presents an efficiency problem if the implementation genuinely allocates new space each time a modified array is created. But it is claimed that

> "In place operations can be automatically compiled into most functional programs without violating their computational semantics" [?]

This optimisation is called *copy elimination*. Table 1 illustrates the dramatic effects of copy elimination on two benchmarks (the Livermore Loops and a hydrodynamic code).

Program	no optimization	optimization
LOOPS	681.6	16.0
SIMPLE	177.7	9.4

Table 1: Effect of Copy Elimination. Cray Y-MP: one CPU. Times in seconds

The most audacious claim of Sisal is that, due to its simplified side-effect-free semantics, compiler analysis can achieve automatic parallelisation more effectively than for an imperative language. This claim is backed by data such as that in table 2 The comparison is between code compiled with the Sisal OSC compiler and Fortran compiled with the Cray CF77 compiler.

Program	1 CPU		4 CPUs		8 CPUs	
	OSC	CF77	OSC	CF77	OSC	CF77
LOOPS	30	37	39	42	42	43
SIMPLE	20	21	61	21	85	21

Table 2: Parallel execution. Cray Y-MP. Performances in Mflop/s

Cann concludes that functional programs

1. are easier to write and maintain and are more concise
2. are amenable to analysis
3. exhibit implicit parallelism and favour automatic parallelization
4. can run as fast as conventional languages.

2.2 The FLARE Project

The UK FLARE (Functional Languages Applied to Realistic Exemplars) project had dual aims:

- transferring of effective functional programming technology and tools from the laboratory to serious practitioners, and
- demonstrating that functional languages are a suitable vehicle for effectively programming parallel machines.

The project focussed on the Haskell language. Haskell is a very different language to Sisal. It is a full-blown, lazy functional language, with an elaborate type system and first-class, higher order functions. Sisal is restrictive—often more restrictive than an imperative language. Haskell is very free. It retains referential transparency, but has a powerful set of abstractions, and for many practical purposes feels more expressive than a typical imperative language.

Scientific Applications were investigated by a group in Department of Computer Science at Swansea. They have translated a sequential Fortran finite element code, used in Computational Fluid Dynamics research. In two reports they have described the results of benchmarking the sequential Haskell, and a parallel implementation for the GRIP machine [?].

Sequential results

Like many scientific applications, the exemplar developed made extensive use of arrays. In contrast to Sisal, neither of the major Haskell compilers [?] implements arrays particularly efficiently. The approaches currently adopted are either too slow (non-constant access time) or too demanding of space (naive copying at each update). The Swansea group built their own implementation of the Haskell array, better suited to their requirements. (The fact that this could be done easily is perhaps characteristic of the flexibility of Haskell.)

Various concrete representations of the sparse arrays in the problem were explored. Here we only record the results with the "fixed bandwidth" scheme, which is directly comparable with the original Fortran. Table 3 (derived from table 3 in [?]) displays timings for two problem sizes—one with 200 elements and one with 800 elements.

Program	Fortran	Haskell
Problem 1 (200)	4.3	91.3
Problem 3 (800)	16.8	exhausted heap space

Table 3: Performance on Sun workstation. Times in seconds

On the smaller problem there is a factor of about 20 between Fortran and Haskell timings. Probably more worrying is the fact that on the larger application, exhaustion of heap space prevented the Haskell code from running at all. More "Haskell-friendly" representations of the sparse arrays did allow the larger problem to be run, but again showed performance degradation by a factor 25 or more relative to the original Fortran.

The authors conclude that

> "The way forward is to introduce some parallelism ... which is commonly held to be easier in a functional setting than in an imperative one"

Parallel Results

GRIP (Graph Reduction In Parallel) is an experimental parallel architecture for executing functional programs [?]. It contains a number of processing elements (PEs) together with intelligent memory units and an input-output unit. To execute a program in parallel on GRIP it was necessary to insert explicit primitives for "sparking" threads. This was reported to be a rather easy modification to the code.

Table 4 is taken from section 4 of [?]. It compares times for the sequential and parallel versions of the CFD code. The authors state that

> "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"

	Fortran	Haskell	GRIP			
	Sparc 1	Sparc 1	1 PE	3 PEs	6 PEs	9 PEs
Iteration time	0.23	3.4	86.3	39.0	30.7	23.0

Table 4: Sequential and parallel times. Times in seconds

The authors conclude

> "Parallel performance fits into an expected trend"

> "There is a need to investigate larger problems to establish more clearly the advantages of parallelization"

> "A merit of functional programming includes relative simplicity of parallelization"

2.3 A Case Study: Jacobi Relaxation on the 2-D Laplace equation

As an further contribution to the emerging picture, we have taken a couple of very simple algorithms, and coded them in Fortran 77, Sisal and Haskell. So far as practical, we endeavoured to use exactly the same algorithms in the three versions. The two problems considered were relaxation for the Laplace equation and a matrix LU decomposition. We report results for the former, which are representative.

The two-dimensional Laplace equation,

$$\nabla^2 u(x, y) = 0,$$

can be solved numerically through the finite difference approximation

$$u(x, y) = \frac{1}{4}(u(x + 1, y) + u(x - 1, y) + u(x, y + 1) + u(x, y - 1)).$$

The values of u are defined on a two-dimensional grid. In the relaxation algorithm this system is solved iteratively through the scheme

$$\begin{array}{ccccc} & u(x,y) & = \frac{1}{4}(\sum & \text{neighbours}) & \\ \nearrow & & & \times & \nwarrow \\ \text{"new" value} & \times & & \times \circ \times & \text{"old" values} \\ & & & \times & \end{array}$$

The detailed codes are given in the appendices. The initial version was Fortran. The algorithm was transcribed to the other languages, trying to keep as close as possible to the style of the original program. This was not always practical, and small changes had to be made.

Remarks on programming style

The Fortran program is slightly longer than the Sisal and significantly longer than the Haskell, although the style was supposedly Fortran-like. To this extent the claims of declarative programs for conciseness are vindicated.

Subjectively, it was felt Sisal was more awkward to use than Fortran or Haskell. Sisal 1.2 does not have very powerful facilities for updating arrays (especially, for updating subsections of arrays). For example, the Fortran version of the Jacobi update, leaving boundaries unchanged, is just

```
          do i = 1, n - 1
            do j = 1, n - 1
              u (i, j) = (uold (i, j - 1) + uold (i, j + 1) +
     &                    uold (i - 1, j) + uold (i + 1, j)) / 4
            end do
          end do
```

whereas in Sisal 1.2 we have to make a new copy of the whole array, including the unchanged boundaries. This involves using a larger index range and a "masked" loop body:

```
u := for i in 0, n cross j in 0, n
     returns array of
       if i ~= 0 & i ~= n & j ~= 0 & j ~= n then
         (old u [i] [j - 1] + old u [i] [j + 1] +
          old u [i - 1] [j] + old u [i + 1] [j]) / 4.0
       else
         old u [i] [j]
       end if
     end for ;
```

As another example, the set of reduction operations is hard-wired into the language, and quite restrictive. So in the Sisal program, the convergence tests had to return integer 0 or 1 rather than a logical, because Sisal 1.2 has no *and* reduction operation. It was necessary to use integer product instead.

The Haskell transcription went smoothly. The *while* loop had to be converted to a tail recursive function call, but otherwise the program looks very much like the Fortran, underneath the unfamiliar syntax. (We could use the built-in list-reduction function *and*, which has no analogue in Fortran 77).

Some timings

The codes were run on a Sun Sparc I workstation. The compilers were: for Fortran, the Sun *f77* compiler; for Sisal, *OSC* version *V12.9.1*, the optimising Sisal 1.2 compiler from Lawrence Livermore; for Haskell, *ghc* version *0.16*, the Glasgow Haskell compiler. We also tried the Chalmers compiler. The Glasgow compiler was faster on this benchmark. Programs were timed with the UNIX *time* command.

grid size	iterations to convergence	f77	Sisal	Haskell
12	125	0.2	0.5	14.1
16	213	0.4	1.4	43.5
24	443	1.9	6.2	207.4
32	740	5.3	18.1	701.4

Table 5: Timings for Laplace Relaxation. Times in seconds.

3 Conclusions/Questions/Perspectives

Performance is a central concern in many scientific programs. Sisal does show a performance comparable to imperative languages on scientific, array-based problems. A suspicion lingers that the cost of this performance is reduced expressiveness of the language. Restricting the semantics of the language makes compiler analysis simpler; it is questionable whether it makes for more concise or maintainable code. Sisal 2.0 lifts some of the restrictions of Sisal 1.2, but implementations were not available for benchmarking.

It has been demonstrated that Sisal can be automatically parallelised—more effectively than sequential Fortran. A fairer comparision might be with High Performance Fortran (HPF), which retains all the facilities of conventional Fortran and adds parallel updates; there is every reason to assume that HPF-like imperative languages should speed up at least as effectively as Sisal on parallel machines.

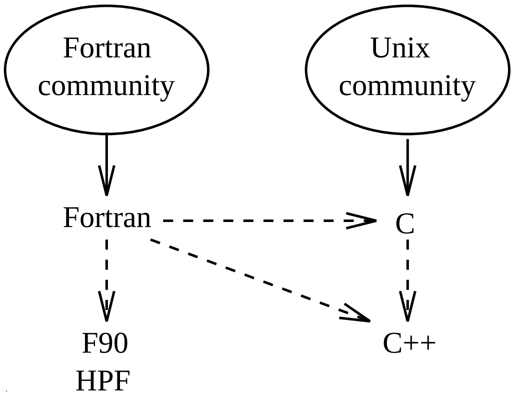
The lazy functional languages, of which Haskell is representative, are perhaps the most elegant and economical of all programming languages. They certainly deserve to be more widely known amongst scientific programmers, if only for specification and prototyping purposes. But on the available evidence it is difficult to claim that they are suitable vehicles for high-performance production codes. Even our simplest array-based programs executed a couple of orders of magnitude more slowly than the Fortran; their profligate use of memory is another major worry.

[TONY'S LAST WORDS...]

The case for appropriateness of functional languages for scientific and engineering applications is "not proven".

Imperative languages are well matched to mass-produced microprocessor architectures. The compilers and tools are pretty well understood. Functional languages should be used in applications where the are *evidently* useful and superior to other technologies.

Why no mass market for "Haskell"?



- Standards vital

  SQL, POSIX, HPF, MPI

- Best parallel programming tool

  FORGE90 from APR
  → sold < 100 licenses!