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

Tony Hey, Bryan Carpenter and Hugh Glaser

Outline

1. Introduction
2. Phenomenology
   - LNLL Sisal project
   - FLARE project
   - Case study
3. Conclusions/Questions/Perspectives

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 and *not* restricted to any specific application area
  - ? Operating systems
  - ? Telecoms applications
  - ? Safety critical embedded systems
  - ? Science and Engineering applications

⇒ In this talk we examine the evidence for the thesis that scientists and engineers should abandon Fortran and C(++) and switch to a functional programming language.

Claimed benefits of declarative languages

Compilers for imperative languages handicapped *by the* assignment statement

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

⇒ Common blocks allow functions to share data

⇒ Aliasing can meant that `A`, `X`, `B`, `Y`, may represent same memory cell

⇒ Parallelism not obvious and requires detailed interprocedural analysis

⇒ *too difficult for all but simple programs*

Functional languages express problem in terms of well-defined functions

⇒ Referential transparency

- no side effects, better software engineering
- simpler semantics allows formal proofs and program transformation
- higher level of abstraction allows rapid prototyping
- implicit parallelism, no need to worry about architectural details

What do scientists and engineers care about?

— Transparent relation of mathematics to programming languages

— Portability and standards, especially for parallel code

— Efficient programs and high performance

— Support for data parallelism. (Only really useful type of parallelism for MPP's)

? Mismatch of concerns ?

— Formal correctness rarely an issue

— C, Fortran adequate for rapid prototyping of Sci/Eng problems

Data parallelism

— Common problem—solution of PDE

$$\nabla^2 u(x,y) = -c(x,y).$$

2-D Poisson's equation

— Numerical solution may be obtained by discretization

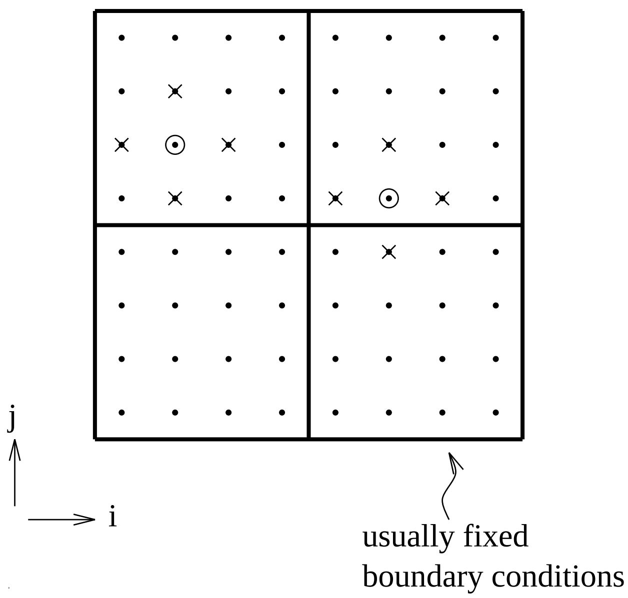
$$u(x,y) = \frac{1}{4}\big(\begin{array}{l} u(x+1,y) + u(x-1,y) + \\ u(x,y+1) + u(x,y-1) + c(x,y)\ \big) \end{array}$$

Finite difference approximation

⇒ 2-D grid of points

⇒ Relaxation algorithm

$$\begin{array}{ccccc} u(x,y) & = \frac{1}{4}(\sum & \text{neighbours} & + & c(x,y)) \\ \uparrow & & \times & \nwarrow & \\ \text{“new” value} & & \times \ \circ \ \times & \text{“old” value} & \\ & & \times & & \end{array}$$



Sequential version ⇒ Nested Do Loops

⇒ Parallel implementations "easy" on Vector, SIMD and MIMD

"SPMD style"
Single Program Multiple Data

2 Phenomenology

LNLL SISAL PROJECT

Concept of "data" replaces "memory"

Concept of "creation" replaces "update"

Problem: Without side-effects an array update must define an entirely new array

"In place operations can be automatically compiled into most functional programs without violating their computational semantics"
COPY ELIMINATION

PROGRAM	NO OPTIMIZATION	OPTIMIZATION
LOOPS	681.6	16.0
SIMPLE	177.7	9.4

Table 1: Cray Y-MP: One CPU. Seconds

PROGRAM	ONE CPU		4 CPUs		8 CPUs	
	SISAL	FORTRAN	SISAL	FORTRAN	SISAL	FORTRAN
LOOPS	30	37	39	42	42	43
SIMPLE	20	21	61	21	85	21

Table 2: PARALLEL EXECUTION. Mflop/s

Cann concludes: Function ...

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

$\Rightarrow$ But most of Sci/Eng community remain unconvinced

UK FLARE Project

Functional Languages Applied to Realistic Exemplars

Dual aims:

— transferring effective functional programming technology and tools from laboratory to serious practitioners

— demonstrating that functional languages are a suitable vehicle for effectively programming parallel machines

Scientific Applications being investigated by group in Swansea CS Department

⇒ Haskell for CFD application

1. Sequential: Fortran → Haskell
2. Parallel: version for GRIP

Sequential results

- Both Haskell compilers produce inadequate optimization of array operations

  — either too slow or too space demanding

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

Table 3: Timings on Sun (seconds)

⇒ "Heap space problems prevent us attempting larger applications"

⇒ Haskell/Fortran $\approx$ 25/1

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

2.0.1 Parallel Results

⇒ Need to insert explicit parallelism primitives for GRIP implementation

	Fortran	Haskell	GRIP			
	Sparc 1	Sparc 1	1	3	6	9
Speed(sec)	0.23	3.4	86.3	39.0	30.7	23.0

⇒ "Can only run our code on an 8-element trivial test problem"

Conclude?

— "Parallel performance fits expected trend"

— "Need to investigate larger problems to establish more clearly the advantages of parallelization"

— "Merit of functional programming includes relative simplicity of parallelization"

? ? ?

Case Study: Jacobi Relaxation on 2-D Laplace's equation

- Fortran version → SISAL, HASKELL versions
- Comments
  - — Length of code:

    Fortran > Sisal > Haskell
  - — Sisal (v1.2):

    lacks facilities for updating sections of arrays

    has a rather restrictive set of reduction operations

    is less easy to customize than Haskell
  - — Haskell implementations in terms of "array of streams" or "stream of arrays"—details in paper by Carpenter and Glaser

```
      program main
      implicit none

c     Simple Jacobi relaxation for Laplace equation.

        integer n
        real eps
        parameter (n = 64, eps = 1.0e-6)

        real u (0 : n, 0 : n), uold (0 : n, 0 : n)
        real bdry
        integer i, j, niter
        logical convgd

c       Zero interior and initialise boundaries.

        do i = 0, n
          do j = 0, n
            if (i .eq. 0 .or. i .eq. n .or. j .eq. 0 .or. j .eq. n) then
      u (i, j) = bdry (float (i) / n, float (j) / n)
            else
      u (i, j) = 0.0
            end if
          end do
        end do

        convgd = .false.
        niter  = 0
        do while (.not. convgd)

c         Save old value.

          do i = 0, n
    do j = 0, n
              uold (i, j) = u (i, j)
            end do
          end do

c         Update.

          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

c         Check convergence.

          convgd = .true.
          do i = 1, n - 1
    do j = 1, n - 1
              convgd = convgd .and.
     &                 (abs (u (i, j) - uold (i, j)) .lt. eps)
            end do
          end do

          niter = niter + 1
        end do

        print *, niter
c        print *, u
      end

      real function bdry (x, y)
      implicit none
      real x, y

        bdry = x**2 - y**2
      end
```

```
define main

% Simple Jacobi relaxation for Laplace equation.

function bdry (x, y : real returns real)
  x * x - y * y
end function

%function main (returns integer, array [array [real]])
function main (returns integer)
  let
    n     := 64 ;
    eps   := 1.0E-6
  in
    for initial

      % Zero interior and initialise boundaries.

      u      := for i in 0, n cross j in 0, n
returns array of
  if i = 0 | i = n | j = 0 | j = n then
    bdry (real (i) / real (n), real (j) / real (n))
  else
    0.0
  end if
        end for ;

      convgd := 0 ;
      niter  := 0

    while convgd = 0 repeat

      % Update.

      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 ;
```

```
      % Check convergence.

      convgd := for i in 1, n - 1 cross j in 1, n - 1
                returns value of product
                  if abs (u [i] [j] - old u [i] [j]) < eps then 1 else 0 end if
                end for ;

      niter  := old niter + 1
%     returns value of niter value of u
    returns value of niter
    end for
  end let
end function
```

```
module Main (main) where

-- Simple Jacobi relaxation for Laplace equation.

n   = 6
eps = 1.0e-6

main = print (let

                -- Zero interior and initialise boundaries.

                u = array ((0, 0), (n, n))
                          [(i, j) :=
                             if i == 0 || i == n || j == 0 || j == n then
                               bdry (fromIntegral i / fromIntegral n)
                                    (fromIntegral j / fromIntegral n)
                             else
                               0.0
                          | i <- [0..n], j <- [0..n]]

              in
                loop False 0 u)

loop convgd niter u =
  if convgd then
    --(niter, u)
    niter
  else
    let

      -- Update.

      u'      = u // [(i, j) :=
                          (u ! (i, j - 1) + u ! (i, j + 1) +
                           u ! (i - 1, j) + u ! (i + 1, j)) / 4.0
                     | i <- [1..n - 1], j <- [1..n - 1]]

      -- Check convergence.

      convgd' = and [abs (u' ! (i, j) - u ! (i, j)) < eps
                    | i <- [1..n - 1], j <- [1..n - 1]]

      niter'  = niter + 1
    in
```

```
      loop convgd' niter' u'

bdry x y = x * x - y * y
```

```
?- op(700, xfx, ->).

isN(16).
isEps(1.0E-6).

main(Res) :- gridDomain(UInit), initialVals(UInit), loop(0, UInit, Res).

% `gridDomain' asserts that a mapping (``array'') has an N x N grid as
% domain (``index space''). NB: it imposes a particular ordering---
% reverse lexicographic---on sites in the mapping list.

gridDomain(U) :- isN(N), enumDomain(N, N, U).

enumDomain(0, 0, [site(0, 0) -> _])     :- !.
enumDomain(I, 0, [site(I, 0) -> _ | U]) :-
!, IM is I - 1, isN(N), enumDomain(IM, N, U).
enumDomain(I, J, [site(I, J) -> _ | U]) :-
JM is J - 1, enumDomain(I, JM, U).

% `initialVals' asserts that a mapping defines a suitable initial state.

initialVals([]).
initialVals([site(I, J) -> B | UTail]) :-
    isN(N),
    (I =:= 0 ; I =:= N ; J =:= 0 ; J =:= N), !,
    X is float(I) / float(N),
    Y is float(J) / float(N),
    bdry(X, Y, B),
    initialVals(UTail).
initialVals([_ -> 0.0 | UTail]) :- initialVals(UTail).

% The main loop...

loop(Niter, U, Res) :- gridDomain(UNew),
                       update(U, UNew),
                       Miter is Niter + 1,
                       loopTest(Miter, U, UNew, Res).

% `update' asserts that all mappings in 2nd arg equal the average of
% values of neighbour mappings in 1st arg.

update(_, []).
update(Uold, [site(I, J) -> X | UTail]) :-
    isN(N),
```

```
    (I =:= 0 ; I =:= N ; J =:= 0 ; J =:= N), !,
    element(site(I, J) -> X, Uold),
    update(Uold, UTail).
update(Uold, [site(I, J) -> X | UTail]) :-
    JM is J - 1, element(site(I, JM) -> A, Uold),
    JP is J + 1, element(site(I, JP) -> B, Uold),
    IM is I - 1, element(site(IM, J) -> C, Uold),
    IP is I + 1, element(site(IP, J) -> D, Uold),
    X  is (A + B + C + D) / 4.0,
    update(Uold, UTail).

% Check for convergence.

%loopTest(Niter, OldU, U, res(Niter, U)) :- converged(OldU, U), !.
loopTest(Niter, OldU, U, Niter) :- converged(OldU, U), !.
loopTest(Niter, _,    U, Res)           :- loop(Niter, U, Res).

converged([], []).
converged([_ -> X | U], [_ -> Y | V]) :- Dif is abs(X - Y),
 isEps(Eps), Dif < Eps,
 converged(U, V).

bdry(X, Y, B) :- B is X * X - Y * Y.

element(X, [X | _]) :- !.
element(X, [_ | L]) :- element(X, L).
```

Some timings

n	iter	f77	Sisal	Haskell	Prolog
12	125	0.2	0.5	14.1	280
16	213	0.4	1.4	43.5	1080
24	443	1.9	6.2	207.4	
32	740	5.3	18.1	701.4	

Comments

- Difficult to optimize functional versions because of complex relation between execution speed and memory utilisation $\Rightarrow$ cf. Profiling tools of Runciman and Wakeling
- Functional versions capture "essence" of algorithm in a way that Fortran probably never can $\Rightarrow$ could have a role in specification or prototyping

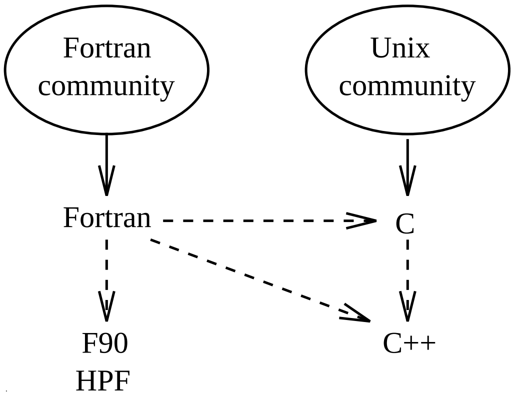
3 Conclusions/Questions/Perspectives

Conclude: Case for appropriateness of functional languages for scientific and engineering applications "not proven".

Imperative languages well matched to mass-produced microprocessor architectures $\Rightarrow$ compilers, tools pretty well understood

$\Rightarrow$ 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!