A Grid-based Scientific Problem, Expressed in Haskell

D.B. Carpenter and H. Glaser

Introduction

Lazy functional languages have not, to date, made an enormous impact in scientific computing. Partly, this has to do with performance—codes written in these languages often run an order of magnitude slower than imperative codes, which limits their appeal for number-crunching production codes[1]. Apart from that, there is a suspicion that programming without assignments or side-effects is difficult, or restrictive. Evidently, this is not the opinion of functional programmers—but perhaps there are things in scientific computation which really cannot be *done* easily without assignments.

In a recent article [2] we have given detailed Haskell implementations of three "scientific" algorithms. Haskell is a relatively new language, put forward as an international standard for functional programming [5, 6, 7]. It is a lazy, pure functional language. In order of increasing complexity, the problems considered were: solution of a partial differential equation by the relaxation method; Monte Carlo simulation of a statistical mechanics system; and a similar simulation of a lattice gauge theory. This report describes a simplified version of the first of those programs. (Other examples of scientific functional programming can be found in [3, 4, 1]).

The 2-dimensional Poisson equation

A fundamental problem in electrostatics is calculation of the electric field produced by a given distribution of charges. The electrostatic potential is obtained by solving the associated Poisson equation. In two dimensions, the fixed charge distribution is a function $c(x, y)$ of the coordinates. The electrostatic potential $u(x, y)$ is the solution of the equation

$$(\boldsymbol{\nabla}^2 u)(x, y) = -c(x, y). \qquad (1)$$

The most simple-minded approach to solving (1), is to replace it by the finite difference approximation

$$\begin{aligned} &u(x+1, y) + u(x-1, y) + \\ &u(x, y+1) + u(x, y-1) - 4u(x, y) &= c(x, y). \end{aligned} \qquad (2)$$

The coordinates x, y are now integers. Rearranging (2) as

$$u(x, y) = \frac{1}{4}(\text{sum neighbouring } u \text{ values} + c(x, y)), \quad (3)$$

immediately suggests the simplest iterative algorithm for solving this system. In the *relaxation* algorithm, starting from some initial guess for u, each iteration replaces the value of u at a point by the average value of its neighbours (plus an offset determined by the local charge). Variants differ in whether the u values inserted in the RHS of (3) are those from the previous sweep of the grid (Jacobi relaxation) or whether they are the most recent values, which may include some values already updated in the current sweep (Gauss-Siedel relaxation). The last version has further variations depending on the order in which the grid is traversed—a popular scheme is "red-black relaxation" in which all even sites are visited first, then all odd sites.

[1] It seems that this caveat on functional programming nolonger holds for some non-lazy functional language—see [1].

Accessing neighbours

The two-dimensional geometry of the problem can be embodied in the following definitions.

```
type Coord   = (Int, Int)

neighbs :: Coord -> [Coord]
neighbs (x, y) = [(x, y + 1), (x, y - 1),
                  (x + 1, y), (x - 1, y)]
```

`neighbs` returns a list of sites neighbouring its argument. The remainder of the code will be independent of the lattice dimension or shape (in [2] we introduced some more general techniques for achieving the goal of "geometry independence"). We define a specialised `map`-like function for an array, which operates on a pair-wise, nearest-neighbour basis

```
nns :: (Coord -> Coord -> b -> c) ->
       Array Coord b -> Array Coord [c]
nns f a =
  array bds [i :=
    [f i j (a ! j) | j <- neighbs i]
           | i <- range bds]
  where bds = bounds a
```

(In Haskell, `array` is the array constructor function—it takes an "association list", in which index values are paired with data by the infix constructor `:=`, and returns an array. This can be subscripted by `!`. It has the same association between indices and data elements.)

`nns` is a function which takes a mapping function and an array, and returns an array in which each element is a mapped list of the nearest neighbour elements from the original array. If $j_1, j_2, \ldots$ are the neighbours of site i, and `b = nns f a`, then

```
b ! i ==
    [f i j1 (a ! j1), f i j2 (a ! j2), ...]
```

Another higher order function we will use is `azipWith`.

```
azipWith f a b =
  array bds [i := f (a ! i) (b ! i) |
             i <- range bds]
  where bds = bounds a
```

This is analogous to the standard `zipWith` function on lists.

Lazy implementation of Red-black relaxation

With the geometry sorted out, we can tackle the main program. Define a data type which captures the local state in a relaxation calculation:

```
type Obs    = Float

data State  = Interior Float Obs |
              Boundary Obs

siteObs :: State -> Obs
siteObs (Interior _  u) = u
siteObs (Boundary    u) = u

siteTrans :: State -> [Obs] -> State
siteTrans (Interior c _) nbs =
           Interior c ((sum nbs + c) /
                   fromIntegral (length nbs))
siteTrans  b              _   = b
```

There are two types of site: normal sites on the "interior" of the lattice, and those on some designated boundary (typically the sites on the edge of the array). Sites on the interior have a local charge density and a potential value. Those on the boundary just have a potential value, which will remain fixed throughout the iteration. The function `siteObs` extracts the "observable", i.e. the potential, from a `State` variable. `siteTrans` performs the update of equation (3) on the interior sites and leaves boundary sites unchanged. It is passed the old value of the site and a list of potential values at nearest neighbours.

The history of a single site can be regarded as a stream of `State` values[2]. The site evolves through application of a transition function which depends on the (changing) environment of the site. This pattern of evolution can be captured through the function[3]

```
process :: (s -> e -> s) -> s -> [e] -> [s]
process trans st ~(env : envs) =
    st : process trans (trans st env) envs
```

When the transition function is `siteTrans`, the "current environment" is a list of neighbour potential values, and the last argument to `process` will be a stream of such lists—the history of the environment.

The result of the computation will be an array of lists of `State` values, defining the history of the whole lattice. Suppose we are given this array and need to calculate the array of environments from it. This can be done by

```
env :: Array Coord [State] ->
         Array Coord [[Obs]]
env = amap transpose .
        nns (\_ _ -> map siteObs)
```

The application of `nns`, in which the mapping function extracts a stream of potential values from the stream of `State` values, gives an array containing a list of streams of neighbour histories. Mapping `transpose` over it turns these into streams of neighbour lists. These are just what `siteTrans` needs as its argument. So the implementation of relaxation is

```
evolve :: Array Coord State ->
            Array Coord [State]
evolve init = x
  where x = azipWith (process siteTrans)
              init e
        e = env x
```

The argument `init` is an array of initial states (`azipWith` is defined to pick up the lattice from its first array parameter).

As well as the lattice size, the array of initial states defines: which array elements are on the boundary of the physical system being described; the charge distribution; the initial guess for the potential on `Interior` sites; and the fixed values of the potential at `Boundary` sites. It's required that no `Interior` site should be on the edges of the array (only `Boundary` sites are allowed there). Because `siteTrans` doesn't evaluate its neighbours argument for `Boundary` sites, laziness ensures automatically that no neighbour access violates array bounds.

The code above implements Jacobi relaxation. More general schemes, like Gauss-Siedel, can easily be implemented in the same framework. Suppose a site A has site B as a neighbour. The scheme above calculates the new value at A in terms of the previous generation value at B. To calculate the value of A in terms of the *latest* value of B, we should look one place forward in the history of B—in other words the B stream should be tailed before feeding it into the environment of A. To organise this, it is convenient to attribute a colour to each site. If the colour of B, say, is less than the colour A, A is updated in terms of the latest value of B. Otherwise it is updated in terms of the previous generation value of B. The function `env` is changed to

```
env = amap transpose .
        nns (\i j -> map siteObs .
          if colour j < colour i
            then tail
            else id)
```

Red-black Relaxation in 2 dimensions can be implemented by setting

```
colour (x, y) = (x + y) `mod` 2
```

Discussion

The implementation creates an "array of streams". An alternative approach is to create a "stream of arrays", in which each update is performed on the whole lattice to create a new global state. It only involves minor changes to the implementation to create such a stream of arrays directly. Alternatively one can feed the results of the given implementation to a pipeline of `elems`, `transpose` and `map (listArray bds)` to produce a list of arrays. There are some advantages to the "array of streams"

[2] By definition, a "stream" is an infinite list

[3] **process** can also be implemented by the standard function **scanl**. Note the lazy pattern-match on the last parameter of **process**. This laziness is required for the implementations below.

approach (apart from the fact that it models process-parallelism). Some existing Haskell implementations do not handle array accesses very efficiently, and in this approach the array operations are confined to an initial stage where the streams are "tied together". In the "stream of arrays" approach array operations are needed in all updates. On the other hand it does seem more natural to have a list of arrays when it comes to processing the data (the set of configurations) generated by a simulation.

The algorithm described involves state transition in an essential way, but it is doubtful whether its description would have been any clearer in terms of variables and assignments. The explicit representation of sequences as infinite lists seems just as natural.

References

1 D. Cann, "Retire fortran? a debate rekindled," *Communications of the ACM*, vol. **35**, no. 8, p. 81, 1993.

2 D. B. Carpenter, "Some lattice-based scientific problems, expressed in haskell," Tech. Rep. CSTR 93-06, University of Southampton, Department of Electronics and Computer Science, March 1993.

3 P. Hudak and J. H. Fasel, "A gentle introduction to haskell," *ACM SIGPLAN Notices*, vol. **27**, no. 5, 1992.

4 P. Hudak, S. P. Jones, and P. Wadler, "Report on the programming language haskell, a non-strict purely functional language," *ACM SIGPLAN Notices*, vol. **27**, no. 5, 1992.

5 A. J. T. Davie, *An Introduction to Functional Programming Systems Using Haskell.* Cambridge University Press, 1992.

6 J. Liu, P. H. J. Kelly, S. M. Cox, and F. S. Taylor, "Functional programming for scientific computation," in *Functional Programming for Arrays of Transputers: The Collected Papers.*

7 W. G. Vree, "The grain size of parallel computations in a functional program," in *Parallel Processing and Applications* (E. Chiricozzi and A. D'Amico, eds.), Elsevier Science Publications, 1987.