Referee's Report on "A Further Study of Sparse Matrix Representations in a Functional Language"

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

1 Summary

The paper is built around a case study, in which a series of different algorithms, coded in Haskell, are applied to a sparse matrix problem arising in fluid dynamics.

The basic formulae of Cholesky decomposition are introduced, and some issues relating to sparseness are summarised. The rest of the paper concentrates on implementation schemes for forward and backward substituion.

The central section outlines a series of implementations built around different concrete representations of the Cholesky factor. The level of detail given in the different cases is quite variable. In the case of the generalised envelope scheme virtually no detail is given. In the case of the quadtree scheme, implementations of all functions required are spelt out.

Section 4 evaluates six of these schemes applied to 3 sets of test data. Timings and space consumption are reported. The quadtree scheme is reported to outperform the others.

2 Suggestions to authors

Although the content of the paper seems worthy of publication, the presentation is a little careless. Fairly extensive rewriting is suggested. The use of English is sometimes poor. There are apparently some errors in the Haskell code fragments. The treatment of the different schemes could be made more uniform. Some extraneous detail could be removed. The following detailed points are not necessarily exhaustive.

Abstract. "a Cholesky direct solver rather than iterative ones is [...]". Bad grammar.

Introduction, paragraph 2. "There are several reasons which make this exemplar of interest [...]". Ugly phrasing ("reasons" should be "properties", or something?).

Introduction, paragraph 3. "[...] we do not consider the effects of lazy evaluation." But at least one of the program fragments given (`forwd_subs` in section 3.2) would diverge without laziness, and the choice of a lazy implementation language surely affects the performances quoted in section 4.

Introduction, paragraph 3. "[...] which contributes to large space consumption by [...]". Ugly phrasing ("contributes to the large space consumption of..."?).

Introduction, paragraph 4, point 2. "[...] is normally more space demanding [...]". Ugly phrasing ("normally demands more space"?).

Section 2, paragraph 2. "One of the system matrices does not vary across time steps.". Does this imply that some others do, and that the techniques discussed in the paper are inapplicable to them?

Section 2, paragraph 4. The classification of zeros and non-zeros (into predictable and non-predictable) seems irrelevant to the Haskell code given in section 3. Could it be omitted?

Section 2, paragraph 4. "[...] because no actual numerical computation is involved here [...]". It isn't entirely clear what "here" means (on closer reading, it means in the symbolic analysis, but it could be better stated).

Section 2, last paragraph. "In this investigation, we always apply a minimum degree reordering fill-in reduction algorithm [...]". No details are given of this algorithm, and it doesn't seem relevant except as a characteristic of the test data. It would probably be better to relegate the paragraph to section 4, which describes the results of the tests.

Section 3.1, paragraph 2. "It is mathematically guaranteed that the diagonals are always non-zero." "Mathematically guaranteed" is a bit vague. Guaranteed by what?

Section 3.1, paragraph 2. "It is questionable if one-dimensional schemes are suitable [...]". Make it clear that this is a major question investigated in the paper. The way it's stated makes it sound unlikely a priori, which makes the conclusion (that these schemes indeed aren't very effective) look weak.

Section 3.2, first paragraph. "[...] `S_array` is our concrete tree implementation of the Haskell abstract array data type [...]" This is slightly misleading. `S_array` isn't an implementation of the array data type in the normal Haskell sense. Also, it isn't used at all "abstractly" in some the algorithms presented, which exploit details of the concrete representation.

Section 3.2, paragraph 2. The implementation of `forwd_subs` is more obscure than it needs to be (to my eye). At the very least, replace the explicit use of `map` by a list comprehension. eg

```
forwd_subs (diag, off_diag) b = s_listArray bnds y
  where
  bnds = s_bounds b
  l = fst bnds
  y = (b!^l)/(diag!l) :
        [((b!^i)-inner_prod y (off_diag!i))/(diag!i) |
          i <- range (bounds off_diag)]
```

Section 3.2, paragraph 2. "The `inner_prod` is a vector inner-product function [...]". This is too imprecise. The "vectors" involved have different lengths!

Section 3.2, paragraph 2. The implementation of `backwd_subs` is even harder to read. It looks much clearer in a procedural language (with vector operations)

```
b := y
for i = n to 1 step -1
  new_x := b [i] / diag [i]
  x [i] := new_x
  b     := b - new_x * off_diag [i]
```

(if `off_diag [i]` is filled out with zeros on the right). Consider quoting this as a higher-level specification of the functional program. Also, I suspect the functional implementation is erroneous. Shouldn't it start

```
backwd_subs chl_factor y =
  fst (foldl gen_x (init_x, y) (reverse (range bnds)))
                                         ^^^^^
```

?

Section 3.2, paragraph 3. "For the list of row segments, we simply use [...]". I was thrown by the abrupt change of subject—from one representation to a different one. New paragraph, perhaps?

Section 3.3. This scheme is classified as one dimensional. Is this convincing? The matrix is divided horizontally into blocks then vertically into row segments.

Section 3.3, paragraph 5. "However, we can treat [...]" This one-sentence explanation of the strategy is too condensed to convey much. Sketch the implementation a bit more fully.

Section 3.3, paragraph 6. "the penalty of having to employ a more complex implementation of the forward/backward substitution steps"... this phrase is too long to go where it does in the surrounding sentence.

Section 3.4. I doubt whether it's really necessary to give the implementation of this scheme in so much more detail than the other schemes. For example the implementations of `multmv` and `addbv` are probably superfluous. They are more obvious than details omitted in earlier sections.

Section 3.4. According to its definition, `multmv` takes a second argument of type `Bin_Trie a`. But in the substitution functions it is passed an argument of type `Bin_Tree Float`. I assume this is a spelling mistake—I can't see where the type constructor `Bin_Trie` is defined.

Section 5. The discussion section needs reworking. It is too short, and not very inspiring. Some larger conclusions would be helpful. Perhaps make a stronger case for the virtues of the quadtree scheme—it's easy to understand, the recursive implementation fits naturally into a functional language, it outperforms the other schemes (at least amongst the Haskell implementations), etc. Perhaps one could compare the functional implementations with procedural implementations. Is the Haskell code simpler or more general? Could the Haskell quadree version ever compete in performance with procedural code? Even if it couldn't, did the investigation still teach us some useful lessons which could be carried over to optimised procedural implementations? (I hope the answer to at least one of these questions is "yes").