

# A performance challenge

**Keith Piraino**

**W**HEN MY COMPANY began looking into Smalltalk we had quite a few skeptics. One challenged me to match the performance of the most time-consuming portion of an existing C-based application. At first, I resisted—I knew that Smalltalk was not as fast as C but felt that its' many other strengths greatly outweighed this weakness. I did my best to steer this effort toward a more complex problem that would better demonstrate the strong modeling capabilities of Smalltalk.

It became clear, however, that performance concerns would torpedo Smalltalk adoption unless these fears were met head on. I accepted the performance challenge and this article describes my experience in trying to meet it.

The problem was reading data from binary files (proprietary structure) and aggregating it into a multi-dimensional symmetrical matrix.\* I used VisualWorks 2.0 running under Microsoft Windows 3.11.

## DICTIONARY

I decided to use a Dictionary to represent my matrix. The keys are the cell coordinates and the values are the cells. I defined the matrix as follows:

```
Object subclass: #SymmetricalMatrix
  instance variables: 'dimensionSizes initialCellValue
                    cellDictionary'
```

dimensionSizes is a collection with the number of elements in each dimension. initialCellValue contains the value to initialize each cell to. cellDictionary is the actual Dictionary of cells. I added a class method to answer a new instance and an instance method to create the actual matrix:

```
SymmetricalMatrix class>>withSizes: aCollection
  initializeCellsTo: aValue
  ^(self new)
    dimensionSizes: aCollection copy;
    initialCellValue: aValue;
    build; yourself

SymmetricalMatrix>>build
  self cellDictionary: Dictionary new
```

For example, "SymmetricalMatrix withSizes: #(2 3 4) initialize

CellsTo: 0" would answer a 2x3x4 matrix where the default value of all 24 cells is zero.

To access the cells we implement #at:, #at:put:, and #at:incrementBy. The first two selectors should look familiar. The last is a convenience method to increment the value of a cell. In all three cases the at: parameter is a collection representing the coordinates of the cell. In our 2x3x4 example "at: #(1 1 1)" will answer the first cell and "at: #(2 3 4)" will answer the last cell. Here is the implementation of the #at: method:

```
at: aCollection
  ^self cellDictionary at: aCollection
  ifAbsent: (self initialCellValue)
```

Dictionary's #at:ifAbsent: method is used to answer the cell at the coordinates or return our initialCellValue if the cell doesn't exist yet. From this the #at:put: and #at:incrementBy methods are trivial. The only catch is to make sure we copy aCollection in #at:put: because if the caller modifies it our dictionary key will change. Here is the code:

```
at: aCollection put: aValue
  self cellDictionary at: (aCollection copy) put: aValue

at: aCollection incrementBy: aValue
  self at: aCollection put: (self at: aCollection) + aValue
```

That's it. Other than accessor methods the code shown is a complete implementation of the matrix. It works, but how fast is it?

The test consisted of creating a three-dimensional matrix and calling #at:incrementBy: over a million times. The coordinates to use were retrieved from binary files in a proprietary format.†

The cells were initialized to zero and always incremented by one. If the C program took X amount of time my first cut in Smalltalk took over 4X (i.e., 4 times slower). A profile of the code showed that most of the time was spent in Dictionary code—#at: and #at:put:. Leveraging the Dictionary class gave me a solution very quickly but there was no room for improvement so I turned to another approach.

## LINEAR

The C application uses a linear array to implement the matrix. A 2x3x4 matrix, for example, is represented by

\*The current C application can handle up to 6 dimensions. Symmetrical meaning that the number of elements in a dimension is constant across all combinations of the other dimensions. If, for example, Region and Brand were two of the dimensions in a matrix, the number of brands can't change between regions.

†Because I'm not presenting the code to "feed" the matrix, all times given are after that section of code was already optimized. In reality, both the matrix and supporting code were optimized simultaneously.

## A PERFORMANCE CHALLENGE

an Array with a size of 24. Given the coordinates of the cell we need to calculate the index in the linear array. One key optimization here is to pre-compute dimension ranges. The range is the number of cells we pass when the dimension index changes. It's calculated by taking the product of the inner dimensions. In a 2x3x4 matrix the range of the first dimension is 12 (3 \* 4), the range of the second is 4 and the range of the last is 1. For example, the index of #(1 1 1) is 1 and the index of #(2 1 1) is 13. The difference between them is the range for the first dimension: 12. The code following assumes that cellArray and dimension Ranges are instance variables containing the Array of cells and the Array of dimension ranges, respectively. Initializing these variables is left as an exercise for the reader.

```
at: aCollection incrementBy: aValue
  | index |
  index := self cellIndexFor: aCollection.
  self cellArray at: index
    put: (self cellArray at: index) + aValue

cellIndexFor: aCollection
  | index |
  index := 1.
  aCollection with: self dimensionRanges do:
    [:eachCoordinate :eachRange |
      index := index + ((eachCoordinate - 1) * eachRange)].
  ^index
```

At 2.5X, the linear matrix was an improvement over the dictionary approach but it was still pretty far off. As I would expect, the profiler showed almost all the time being spent in #cellIndexFor:. Apparently, this calculation was a lot slower in Smalltalk than in C. Other than caching dimension ranges, I don't know of any other optimizations to this algorithm. I still wanted to meet the challenge so I moved on yet again.

### TREE

To avoid the cost of calculating an index, I decided to implement a matrix as a tree structure where intermediate nodes are a kind of index but the actual cells are all in leaf nodes. Figure 1 shows an Object Explorer picture of a matrix created via the following code fragment:

```
matrix := TreeMatrix withSizes: #(2 3 4) initializeCellsTo: 0.
matrix at: #(1 1 2) put: 5.
matrix at: #(2 3 4) incrementBy: 1.
```

The code to create the initialized tree structure is left as an exercise for the reader. To access a particular cell, we must traverse the tree to get to the correct leaf node. The #leafNodeContaining: method answers the leaf node containing the cell defined by the given coordinates. From there it's a simple #at:put: to the Array. Here's the code:

```
at: aCollection incrementBy: aValue
  | leaf |
  leaf := self leafNodeContaining: aCollection.
  leaf at: aCollection last
```

```
put: (leaf at: aCollection last) + aValue.
```

```
leafNodeContaining: aCollection
  ^(aCollection copyFrom: 1 to: (self numDimensions - 1))
  inject: (self rootNode) into: [:node :each | node at: each]

numDimensions
  ^self dimensionSizes size
```

The TreeMatrix reduced the time to about 2.1X: only slightly better than the linear approach. Having run out of ideas, I did what I should have done earlier—looked closely at the profile results. Most of the time was spent in #leafNodeContaining:, but not performing the #at: that traverses the tree. There was some overhead in #copyFrom:to: but most of the time was spent in the #do: loop<sup>†</sup> called by #inject:into:. The overhead of looping was far greater than what I was actually doing in the loop!

I went back to LinearMatrix and, sure enough, most of the time was spent in #with:do:. The #with:do: method creates a Stream to traverse the second collection. That occupied some time, but again, most of the time was spent in the #do: loop. I was dead wrong when I assumed that calculating the cell index, the addition and multiplication necessary was the problem. With this revelation I could

have returned to optimizing LinearMatrix but I decided to stick with TreeMatrix because it was slightly faster even discounting the #do: impact.

### LOOP UNROLLING

If the problem is looping, I figured why not eliminate the loop? The #leafNodeContaining: method can easily be optimized for a particular number of dimensions. One way to do this would be to create a different subclass for each number of dimensions and override #leafNodeContaining: in each. A more manageable approach is to use blocks. When the matrix is instantiated, a block optimized for that number of dimensions is assigned to the instance variable leafAccessorBlock. Given the cell coordinates and the root node this block goes directly to the leaf node via a series of #at: messages. Listed next are the one-, two-, and three-dimensional blocks:

```
[:coord :root | root]
[:coord :root | root at: (coord at: 1)]
[:coord :root | (root at: (coord at: 1)) at: (coord at: 2)]
```

Modifying #at:incrementBy: to use the leafAccessorBlock dramatically reduced the time to about 1.1X. A fresh profile showed that doing "aCollection last" twice to access the index of the cell in the leaf node was now taking a noticeable amount of time. I was able to take advantage of the fact that the position of this last coordinate is known at the time the matrix is instantiated. After adding an instance variable to cache the number of dimensions in numDimensions, at:incrementBy: was changed to:

```
at: aCollection incrementBy: aValue
```

<sup>†</sup> I use the term "loop" somewhat loosely to also include enumerating the elements of a collection.

## A PERFORMANCE CHALLENGE

```
| leaf lastCoordinate |
leaf := leafAccessorBlock value: aCollection
      value: self rootNode.
lastCoordinate := aCollection at: self numDimensions.
leaf at: lastCoordinate
put: (leaf at: lastCoordinate) + aValue
```

This brought the time down to 0.9X. Actually, a little faster than the C code! Success!!

I'm not suggesting that Smalltalk is faster than C in a head-to-head comparison. There are mitigating factors<sup>§</sup> that prevent me from making that claim even in this case. The real point is that the performance gap can be narrowed to the point where it will only rarely be the deciding factor in technology selection.

To generalize, this idea of “unrolling loops via blocks” is useful in the following circumstances:

- Looping code takes more time than contents of loop
- Number of iterations varies but the maximum is relatively low
- Contents of loop relatively simple

It's fine for the problem at hand but the last two constraints limit the usefulness of this technique. If the problem involved 100 iterations with even 5 lines of code in the loop, this approach becomes completely unwieldy. At this point I released my findings and declared success. Although I wasn't completely satisfied with my solution, I didn't have any more time to spend on it.

### A BETTER LOOP?

Several months later I returned to this problem. I reviewed back issues of the *Smalltalk Report* looking for any performance information regarding looping. I found the answer I was seeking in an article on performance by Alan Knight.<sup>1</sup>

It turns out that certain kinds of blocks are inlined and others aren't. These optimizations are vendor specific but can have a major impact. In *LinearMatrix* I used `#with:do:` for looping and in *TreeMatrix* I used `#inject:into:`. Neither is inlined and both call `#do:` which is also not inlined. In *VisualWorks* both `#whileTrue:` and `#to:do:` are inlined. To take advantage of this, I rewrote `#leafNodeContaining:` as shown:

```
leafNodeContaining: aCollection
| node |
[node := self rootNode
1 to: self numDimensions - 1 do:
[:i | node := node at: (aCollection at: i)].
^node
```

I tested this and received results similar to unrolling the loop: 0.8X. The only cautionary note here is to avoid add-

ing parentheses to improve readability. If you change this code to read “(1 to: self numDimensions -1) do:” it will run much slower. Why? Because instead of the compiler recognizing `#to:do:` and inlining, it instantiates an *Interval* via `#to:` and then sends the *Interval* the `#do:` message. I was actually doing this in some of the code that feeds data to the matrix. When I removed the parentheses the time was further reduced to 0.5X (i.e., twice as fast as the C code!!)

Optimizing the loop via `#to:do:` is better than unrolling it for two reasons: (a) it is more generally applicable; (b) perhaps more important, the code is closer to the original intent and, hence, much more readable. In fact, although slightly less concise, the `#to:do:` version may be more easily understandable than my original version that used `#copyFrom:to:` and `#inject:into:`.

Letting go of code and ideas you've invested in isn't always easy. More than once I've seen developers refuse to do it. Sometimes they think they've gone too far to back off. This is understandable but usually misguided in light of long-term maintenance costs and the malleability possible with newer tools. A less defensible cause of this refusal is emotional attachment. People get excited about their first idea (which is good) but sometimes become blind to newer and better ones (which is bad). If you write something you think is really cool but you've since found a simpler more maintainable approach—file out the cool code and play with it in your spare time. Don't leave it in a production application.

Good systems come, in part, from a willingness to throw away some of your code. Vendors can provide the tools to rework code and enlightened management can provide the time but it simply doesn't work without people with the right attitudes.

### MISCELLANEOUS

In the course of optimizing your code be careful not to break it. An approach that works for me is to try out optimizations in subclasses. Once you finish optimizing you can decide whether or not to consolidate. Next is the hierarchy of *Matrix* classes I ended up creating:

```
AbstractSymmetricalMatrix
DictionaryMatrix
LinearMatrix
  LinearMatrixFastLoop
TreeMatrix
  TreeMatrixUnrollLoop
  TreeMatrixFastLoop
```

For the particular test I used, *TreeMatrix* gave the best performance. When I tried a test with more dimensions, however, *LinearMatrix* performed better than *TreeMatrix*. The best approach might be to instantiate the optimum *Matrix* using the *Bridge* pattern<sup>2</sup> in the same way that the *VisualAge* *Collection* classes are implemented.<sup>3</sup>

In the code that feeds data to the matrix I found blocks useful as a way to avoid reevaluating conditionals in tight

<sup>§</sup> My Smalltalk version has only a subset of the C functionality. This means that the C program must pass through a lot more conditionals and code even for my simple test. Compounding this is the fact that Smalltalk is running in 32 bits (via win32s) and the C code is a 16 bit app.

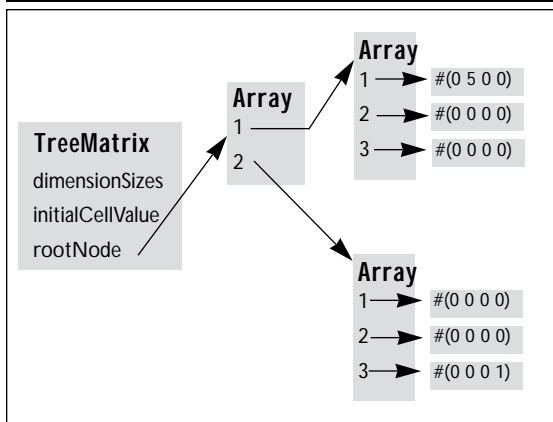



Figure 1. A 2x3x4 TreeMatrix.

loops. If there are more than two possibilities, this can help eliminate case-like statements.

One area where I didn't need to worry about performance was reading from the binary files. ExternalRead Stream seems to be plenty fast. My only problem was that methods like #nextLong assume a Little-Endian byte order and my data is Big-Endian. Correcting this wasn't a big deal but it's ironic since VisualWorks itself seems to store Integers in the image in Big-Endian order.

## CONCLUSION

Smalltalk's expressive nature, extensive class library, and interactive environment greatly improve the most important performance measure—*developer productivity*. You can quickly create a solution to the problem. The combination of hardware advances (faster, more memory) and Smalltalk vendor advances (dynamic compilation, faster library code) mean that you may never have to worry about performance. Don't assume Smalltalk isn't fast enough for your whole application.

If performance is an issue, use a profiler to locate the small percentage of code that needs work. It's easy to do and works a lot better than intuition. Always keep in mind the maintenance cost of tweaking code for performance. You might want to consider trying another algorithm/design approach instead. This will often give you greater performance leverage and more understandable code. It's also a lot easier to do in Smalltalk than in other environments. Again, don't resist throwing code away—it's an important part of the process. 

## References

1. Knight, A. More performance tips, THE SMALLTALK REPORT 4(2): 19-20, 1994.
2. Gamma, E. et al. DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE, Addison-Wesley, Reading, MA, 1995.
3. Lalonde, W. and Pugh, J. Communicating reusable designs via patterns, JOOP 7(8): 69-71, 1995.

---

Keith Piraino can be reached at [keith\\_piraino@npd.com](mailto:keith_piraino@npd.com).