alodar systems™

Macintosh Tech Tip

WHITE PAPER

Refactoring in Scientific Computing

Scientific computing addresses many large problems - large in execution time, disk usage, or memory bandwidth. Many of the seminal problems in this area have been addressed originally with FORTRAN, and ofttimes, the most popular algorithms are forward translations of that well tested FORTRAN code.

The wide prevalence of FORTRAN in the scientific community leads to certain habits of thought. For example, programs written in this style have a great many global variables, corresponding to FORTRAN common blocks. Very rarely are small methods or accessors used, since f77 does not really support the style of object oriented programming that encourages accessors.

This white paper discusses how one would approach refactoring an existing FORTRAN-style code base that is working and already efficient.

Establish the Baseline

For this white paper, we download the sequential numerical analysis benchmark from the Java Grande website. The specific benchmark we will work on is the Euler code module. It calculates airflow in a two dimensional hypersonic wind tunnel.

First, we ran a basic benchmark. On the machine used for this example the size B benchmark output is:

work@eidolon:section3$ java -cp .:.. JGFEulerBenchSizeB

Java Grande Forum Benchmark Suite - Version 2.0 - Section 3 - Size B

Section3:Euler:Init:SizeB 0.311 (s) 120080.38 (Gridpoints/s)

Section3:Euler:Run:SizeB 8.716 (s) 11.473153 (Timesteps/s)

Section3:Euler:Total:SizeB 9.042 (s) 0.110595 (Solutions/s)

Secondly, we examined the code in Tunnel.java and Statevector.java. These classes do the majority of the work, and so attempts to improve code clarity should be focussed here.

Look for Common idioms

Several common idioms exist in this code base. For example, a Statevector is a class with four double fields, named a, b, c, and d. They are public, and virtually all modifications are done through direct manipulation.

The Tunnel class has 55 package protected fields, including ones named ‘scratch’, ‘temp’ and ‘temp2’. This is not a problem a priori, but it does require one to intuit the lifetime of the fields and the conditions under which they might be accessed from the outside.

The majority of the storage is in several collections of two dimensional double or Statevector arrays, rather than an opaque collection class managing the data as an indexed array hidden by accessors. These arrays contain several hundred thousand Statevectors, and are not typically resized during a run.

The majority of the arrays are manipulated using direct field access via inlined code, which can obscure the meaning of that code.

For example, given that scrap4 and temp2 are both Statevector, the following is a standard vector add and multiply by constant operation:

```
temp2.a = 3.0*(localug[i-1][j].a-localug[i][j].a);
temp2.b = 3.0*(localug[i-1][j].b-localug[i][j].b);
temp2.c = 3.0*(localug[i-1][j].c-localug[i][j].c);
temp2.d = 3.0*(localug[i-1][j].d-localug[i][j].d);
```

while the following, very similar looking code is a constant add to vector and multiply by constant. Very different in effect, but very similar looking to the eye.

```
tempdouble = nu4*adt;
scrap4.a = tempdouble*(temp.a+temp2.a);
scrap4.b = tempdouble*(temp.a+temp2.b);
scrap4.c = tempdouble*(temp.a+temp2.c);
scrap4.d = tempdouble*(temp.a+temp2.d);
```

Create refactored methods

At many points, we broke out common blocks of code into a refactored method. For example, code like the "a times quantity x minus y" above is a perfect candidate - it exists at many points in the code base, and and it is a single idea in the mind of a computational physicist.

Other cases were different enough that they turned into three or four separate, but similar methods. For example, the 'calculateDamping' method was broken into four 'calculateDampingForFace' methods for the east, west, north, and south faces. This reduced the core iteration of the method to just ten lines, which is easier for a developer to understand than the original 169 lines. Further, each of those methods went from approximately 42 lines to 15 lines.

Taking out a hundred lines of code is good in its own right. It is likely that a bit more work would let us replace the four damping methods with just one, perhaps ten percent longer, to handle the differences between the four faces. This is reasonable with four fifteen line methods, but much harder with methods three times longer, let alone a single monolithic block of code.

The Statevector class ended up with a constructor, accessors, and the following methods:

```
public final void zero()
public final void set(Statevector that)
public void addResultOfAddMultiply(double m,
        Statevector left, Statevector right)
public void addResultOfAddMultiply(double m, double left,
        Statevector right)
public void addResultOfSubtractMultiply(double m,
        Statevector left, Statevector right)
```

```
public void setResultOfSubtractMultiply(double m,
        Statevector left, Statevector right)
public void set(double a, double b, double c, double d)
public void subtractResultOfMDeltaOverATimesRMinusD(double m,
        double delta, double a, Statevector r, Statevector d)
```

While complicated methods, such as addResultOfSubtractMultiply, could be expressed in terms of simpler methods, these fit the problem domain. Computational physicists think in terms of certain basic matrix and vector operations.

In addition to the replacement routines, we also made the fields private, and replaced access with accessors. This does not add to readability, but it make it easier to differentiate between read and write access. This can be important when choosing caching strategies for temporary variables. (Even a simple text editor can find all calls to any setA method, and a refactoring editor like Eclipse or IDEA can find usages of setA for a specific class.)

A Refactored Code Example

The code listed above became

```
temp2.setResultOfSubtractMultiply(3.0,
localug[i-1][j], localug[i 1][j]);
```

and

```
scrap4.setResultOfAdd(temp.getA(), temp2);
scrap4.multiply(nu4*adt);
```

or

```
scrap4.setResultOfAddMultiply(nu4*adt, temp.getA(), temp2);
```

Obviously, overloading the function setResultOfAddMultipy to take both vectors and doubles increases the possibility of confusion. For a math-literate audience, the names could be changed to fit conventional terms, such as setGAXPAY (generalized a x plus a y) instead of setResultOfAddMultiply.

Results

	Total Execution Time (sec)	Time steps/sec
Original 1.5 client vm	9.68	10.69
Original 1.5 server vm	10.09	10.27
Refactored 1.5 client vm	11.32	9.07
Refactored 1.5 server vm	8.78	11.83

Note that the original code performed much better under the standard VM, while the aggressively refactored code performed much better under the server VM. This is not a typo - the two virtual machines Sun provides in Java 1.5.0_06 are suited for different styles of code.

These were the average results from three runs of the same code. The VMs were not warmed up first, so these include compile time overhead. Warm-up runs designed to force JITC compilation showed no more than a five percent improvement over time. This makes some degree of sense, as commonly used methods will be inlined quite early in the compilation process.

Tests were carried out on a MacBook Pro 2.16 GHz. The code is single threaded, so only one CPU was used for actual calculation. Tests were also performed with a 1.6 beta 1 preview release VM. While they are not appropriate for benchmarks, they do show similar behaviors.

Dead ends

Conventional wisdom from the Java 1.2 era said that making your classes and accessors final would improve your speed. We changed the most often used accessors and classes to that, and measured again. It turns out, as one would guess from reading recent Java performance books (Shirazi, for example), that this does not help noticeably. While this was a good optimization hint in earlier releases of Java, modern JITC systems make this unnecessary.

Conclusion

Modern Java VMs remove much of the overhead of proper object oriented programming. When used with aggressive optimizations, like those afforded by the Hotspot Server VM, they can even be faster than older, inlined code.