Natural Resources and Environmental Issues

Volume 8 *SwarmFest 2000* Article 3

2001

Writing fast models in Swarm

Marcus Daniels
Swarm Development Group, Santa Fe Institute, NM

WRITING FAST MODELS IN SWARM

MARCUS DANIELS

Swarm Development Group, 624 Agua Fria, Suite 2, Santa Fe, New Mexico 87501, Email: mgd@swarm.org

Abstract. Execution performance is an important practical issue for agent based modeling. This document traces the path of a Java Swarm model from performance profiling and evaluation through to redesign and reimplementation of a much faster sibling model. This document is available on the World Wide Web (http://www.santafe.edu/~mgd/swarmfest2000/performance.html). Additional example code is available here (ftp://ftp.swarm.org/pub/swarm/src/users-contrib/anarchy/java_performance-0.0.tar.gz).

GOAL

After a variety of internal optimizations to the Java layer of Swarm 2.1 and with identical parameters in batch mode, the Java implementation of Heatbugs runs four to five times slower than the Objective C version depending on the Java virtual machine used. Why is this and what can be done about it?

BACKGROUND

Typing: Theory vs. Practice

Objective C and dynamically-typed languages

In Objective C, and other dynamically-typed languages, the idea is that information about objects live in the objects themselves and that some appropriate level of typing emerges from experience in using those objects, evolving the taxonomy of objects.

Dynamically-typed languages can be useful for *finding types* during exploratory bottom-up programming because as components grow in complexity one component can be combined with another so long as the modeler is prepared to handle all the exception cases evident from looking at the types found at the objects at the interfaces (e.g. using conditionals). Since there is a natural drive to reduce the number of exception cases (in order to manage complexity), one seeks to evolve interfaces until everything fits together in a natural way.

One practical problem with Objective C is that the exception cases are typically handled in a violent way: by core dumps. Other languages, like Common Lisp have object-oriented condition systems that can be used to build taxonomies of the exception cases themselves in an interactive environment.

Java and statically typed languages

Java aims to put more typing information in the hands of the compiler, by encouraging the use of many types.

- The compiler can see more errors ahead of time.
- The compiler doesn't have to do things in a general way. When it knows lots of specific constraints of a situation, it can (in principle at least), write faster code.

It is one thing to say that A' is like A and AB is a superset of A, and to insist that B is not A, but it is quite another to do a better job with AB than with A. Unfortunately, what often happens is that compilers don't even end up *seeing* the conceptual level at all, with basic data structures being kicked out to libraries which don't benefit at all from *obvious* global optimizations.

To a considerable extent Java suffers from this. Very common data structures (e.g., java.util) that are used as standard equipment everywhere live it external libraries and the compiler has no special

knowledge about them. Of course, C punts completely on this, but at least the C programmers *know* nobody is looking out for them.

Functional programming languages

Languages like ML, Haskell, and Mercury have advanced type systems that really do use type information to improve the quality and correctness of code. Unfortunately these languages do not have the kind of mindshare that Java and C do.

Multilanguage Layer

Java Native Interface

The Java API of Swarm is provided via a standard Java interface called Java Native Interface (JNI). JNI is a way to take native-code libraries like Swarm and make them available in Java.

JNI provides the features below. Together they provide the low-level substrate for the Swarm Java layer. Each item has performance implications.

- A C interface for Java. This lets C programs interrogate and modify Java objects.
- Compared with compiled C or Java code, these requests are done through calls (not inline code) which can introduce costs from conservative argument checking that wouldn't be needed if the context was known.
- Type lookups are done by name or by encoding strings. These are extra conversions. (In practice, these costs are not hard to avoid.)
- A memory management model for giving C code control over what the Java garbage collector is allowed to move or destroy.

This is not a direct cost, however, with JNI is it very easy to get resource leaks on the object handles that JNI provides. Luckily, Swarm hides these details from modelers. This issue is related to multiplicity of objects, which will be mentioned in a moment.

- A C naming convention (or `mangling') that makes C functions appear as methods in Java classes.

This is a cost in the sense that there is an extra function, an otherwise unnecessary level of indirection, that only exists to make Swarm `appear' in Java land.

Java Virtual Machine

One of the main advantages of Java is its portability. There is a specification of what a `Java machine' looks like and different kinds of compilers can target that machine. Likewise, there are different ways to implement this virtual machine (VM):

- Interpreters
- To interpret Java virtual machine instructions (bytecode), a program is needed that reads instructions one by one and then maps them to fixed implementations of those instructions. Interpreters, although easy to implement and trace, are often less than half as fast as native code because so much time is spent just processing and calling instruction implementations.
- Just In Time Compilation (JIT). The basic idea with Just in Time compilation is to convert frequently used code patterns into native code that can be cached and glued together as needed.
- -Optimization benefits need to amortize costs
- + Potential for global optimizations from observed program dynamics
- - Complex and system dependent implementation
- + Speed without comprimising on portability
- Hybrid Ahead-of-time/Just-in-time

Static compilation with either of the above. There is now technology available (Kaffe/gcj) to compile Java classes to native code and load them as shared libraries as if those libraries were Java bytecode files. This can be convenient because when `turbocharging' a model is beneficial for big runs but not essential for development.

- + Benefits of traditional compilation technology (e.g. chess vs speed chess)
- - Statically compiled code becomes `opaque' to Java-level profiling and debugger

- - Statically compiled portions must be compiled on any target that the Java code is to run

Finally, there is the option of pure static compilation of Java code in cases where portability is not a concern.

Multiplicity of object types

Under the hood of a Java Swarm model is the Objective C Swarm kernel, and thus two coexisting language `runtimes'. A runtime includes the data structures and built-in functions that the language requires above and beyond the compiler.

In some cases a runtime will be fairly simple (e.g. Objective C), in other cases it can have many datatypes and features (e.g. Common Lisp).

Except in cases where there is a way to unify object representations (e.g. g++ and gcj do this), it is necessary to have a directory to associate one kind of object with another (to the extent there is a common denominator).

Consider the case of a Java Swarm model. An agent is implemented in Java and it has a step method that runs every time step. However, the Swarm activity library deals with Objective C classes like `ActionTo' that only know about Objective C objects and methods. In this situation it is necessary to introduce a proxy that looks like an Objective C object but can talk to the Java world when the time comes.

But here's the rub: when that time comes (quite frequently, actually), it's necessary to figure out what Java object a proxy actually represents and then make the method invocation. Even though this lookup and invocation can be implemented in a fairly efficient way, it can become a more and more expensive cost as a model becomes more fine-grained. (Swarm provides native-language callouts that can be used to avoid this cost -- it's the general case that must be used sparingly.)

PROFILING, ANALYSIS, AND RE-ENGINEERING

The first task in making a program go faster is to find out where it is slow. Both Java and [Objective] C provide a feature called *profiling* which accumulates a dataset on the time spent in each method or function as a given program runs. In the case of Java, the runtime simply has instrumentation in the virtual machine that tracks every call the machine makes. C tracks where time is spent by sampling the program counter and building a histogram.

Unmodified Java Heatbugs

Below we use a Swarm built for Kaffe with the "-prof" option, and then run some text processing on the output to see the most expensive methods:

- cost of individual methods in milliseconds
- cumulative cost of those methods in milliseconds
- full class name and argument types

(Java methods are polymorphic, so the argument types are necessary information for identification.)

$ time javaswarm -prof StartHeatbugs

You typed `heatbugs -b' or `heatbugs --batch' so we're running without graphics.

Heatbugs is running for 300 timesteps quitting at 300

real 0m37.410s

user 0m35.100s

sys 0m0.110s

$ awk '{ print $3, $2, $6 }' prof.out | sort -g | egrep -v 'e\+|clockTick:' | tail -8

1294.42 1294.42 swarm/space/Diffuse2dImpl.putValueatXY(III)Ljava/lang/Object;

1424.4 1424.4 swarm/random/UniformDoubleDistImpl.getDoubleWithMin$withMax(DD)D

1519.86 1519.86 swarm/random/UniformIntegerDistImpl.getIntegerWithMin$withMax(II)I

1575.28 1575.28 swarm/space/Grid2dImpl.getObjectAtX$Y(II)Ljava/lang/Object;

1676.89 19943.2 HeatSpace.findExtremeType$X$Y(ILHeatCell;)I

3839.54 3839.54 swarm/space/Grid2dImpl.putObjectatXY(Ljava/lang/Object;II)Ljava/lang/Object;

17163.3 17163.3 swarm/space/Diffuse2dImpl.getValueAtX$Y(II)I

34223.8 34223.8 swarm/activity/SwarmActivityImpl.run()Lswarm/defobj/Symbol;

The last line can be ignored; the discrete event simulator is native code that appears opaque to Java. However, there are two useful facts here:

- get and put operations are taking a lot of time. These are very simple methods that just read and write a fixed matrix. Their expense implies object directory lookup costs. Eliminating these costs would result in 50% speed up! (And is surely skewing the profile.)
- Significant time is being spent in findExtremeType$X$Y, which is expected since each heatbugs is motivated by the desire to find a more comfortable temperature.

Eliminating the object directory lookup costs

The get and put operations that are accumulating runtime are for checking diffusion heat space values and for updates to grid that holds the heatbugs themselves. These data structures are Objective C data structures accessed through the Java layer, so there is a lot of time being spent by the layer converting back and forth between Java wrapper objects for Objective C classes and the Objective C instances themselves. The direct way to get rid of this cost is to make a homogeneous implementation for these *hot spots*.

Since this is a Java implementation of Heatbugs, we want more Java not less Java; replacements for the diffusion space and for the grid that the heatbugs live on. Although these classes are not complicated, there's no need to implement them again since the RePast (http://repast.sourceforge.net) project already has implementations that very nearly mirror Swarm's in Java.

First cut: dropped in the RePast code

Changes.–

- HeatSpace from RePast's heatbugs
- Heatbug from RePast's heatbugs
- Modified Swarm's HeatbugModelSwarm to use them
- Modified Swarm's HeatbugObserverSwarm to use RePast's Object2DDisplay, Value2DDisplay, ColorMap, and DisplaySurface classes. (Not relevant to this profiling example, but as a simple way to confirm that the heatbugs are doing their thing.)

A step sideways.–

$ CLASSPATH=.:colt.jar:repast.jar javaswarm -prof StartHeatbugs -b

You typed 'heatbugs -b' or 'heatbugs --batch' so we're running without graphics.

Heatbugs is running for 300 timesteps

quitting at 300

real 0m36.263s

user 0m34.130s

sys 0m0.110s

$ awk '{ print $3, $2, $6 }' prof.out | sort -g | egrep -v 'e\+|clockTick:' | tail -4

519.98 519.98 java/lang/Math.floor(D)D

666.07 666.07 java/lang/Throwable.fillInStackTrace()Ljava/lang/Throwable;

838.183 920.211 java/util/Vector.<init>(II)V

1040.54 1040.54 swarm/SwarmEnvironment.initSwarm(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;[Ljava/lang/String;)V

2735.61 2735.61 uchicago/src/collection/DoubleMatrix.getDoubleAt(II)D

3782.95 4631.77
java/util/AbstractList.add(ILjava/lang/Object;)V

16101.5 24772.3
uchicago/src/sim/space/Diffuse2D.diffuse()V

33308.6 33308.6
swarm/activity/SwarmActivityImpl.run()Lswarm/defobj/Symbol;

However, the enemy has shown itself: notice the cumulative cost of the 'diffuse' method is more than half of total runtime!

Second cut: re-implemented HeatSpace, Diffuse2D, and Discrete2D.

Changes.–

- Changed Diffuse2D to use a Java array rather than functionally accessing a RePast DoubleMatrix.
- Eliminated use of `double' and `long' in diffusion arithmetic, instead doing prescaling and normalization. Unlike C (on a 32 bit box), `long' in Java is a eight byte integer for which arithmetic is slower, switch to using `int' which is like `long' in Objective C.
- Avoided redundant modulus computations on coordinates by explictly handling the boundary conditions outside of loops
- By re-sorting the profiling information it becomes evident findExtremeType is responsible for the AbstractList.add, thus:
- replaced use of Vector for tracking equal heatcells with preallocated x and y coordinate arrays
- changed HeatSpace to inherit the matrix for the cell search
- Do matrix loops in row major order in the hopes of getting better cache behavior.

YEAH, BABY!

The new model beats batch Objective C heatbugs. If the Objective C model is compiled with -pg (for profiling information provided by GCC) the Objective C model loses another .2 seconds. Java profiling is more costly still.

$ CLASSPATH=.:colt.jar javaswarm -prof StartHeatbugsBatch -b

You typed `heatbugs -b' or `heatbugs --batch' so we're running without graphics.

Heatbugs is running for 300 timesteps

quitting at 300

real 0m7.839s

user 0m5.760s

sys 0m0.060s

$ awk '{ print $3, $2, $6 }' prof.out | sort -g | egrep -v 'e\+|clockTick:' | tail -10

131.358 138.769
java/lang/Character.getCharProp(C)Ljava/lang/Character$CharacterProperties;

133.173 133.173
java/lang/Runtime.loadFileInternal(Ljava/lang/String;)Ljava/lang/String;

168.771 6531.21
StartHeatbugsBatch.main([Ljava/lang/String;)V

232.407 1957.56
space/Diffuse2D.computeRow()V

599.957 893.807
HeatSpace.findExtreme(III)Ljava/awt/Point;

1038.95 1038.95
swarm/SwarmEnvironment.initSwarm(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;[Ljava/lang/String;)V

1294.78 1294.78
space/Diffuse2D.computeVal()V

4906.42 4906.42
swarm/activity/SwarmActivityImpl.run()Lswarm/defobj/Symbol;

$ time ./heatbugs -b

You typed 'heatbugs -b' or 'heatbugs --batch', so we're running without graphics.

Heatbugs is running for 300 timesteps.

real 0m8.876s

user 0m7.840s

sys 0m0.020s

GIMME MORE OF THAT!

The results above are using a Kaffe snapshot from 2000-03-07 using the new JIT3 compiler on an i686 Debian 2.2 box. The same Kaffe build includes a feature to load in shared libraries compiled by the GNU Java Compiler (gcj) as if they were Java bytecode libraries. When we do this, we improve performance by another 36%, a factor of 9.6 speed-up from the original Java Heatbugs.

$ time CLASSPATH=.:jheatbugs.so:classes javaswarm -prof StartHeatbugsBatch -b

You typed 'heatbugs -b' or 'heatbugs --batch' so we're running without graphics.

Heatbugs is running for 300 timesteps

quitting at 300

real 0m5.788s

user 0m3.660s

sys 0m0.110s