JavaGenes and Condor: Cycle-Scavenging Genetic Algorithms

Al Globus, Veridian MRJ Technology Solutions, Inc. at NASA Ames Research Center
Eric Langhirt, Sterling Software, Inc. at NASA Ames Research Center
Miron Livny, University of Wisconsin
Ravishankar Ramamurthy, University of Wisconsin
Marvin Solomon, University of Wisconsin
Steve Traugott, Sterling Software, Inc. at NASA Ames Research Center

Abstract

A genetic algorithm code, JavaGenes, was written in Java and used to evolve pharmaceutical drug molecules and digital circuits. JavaGenes was run under the Condor cycle-scavenging batch system managing 100-170 desktop SGI workstations. Genetic algorithms mimic biological evolution by evolving solutions to problems using crossover and mutation. While most genetic algorithms evolve strings or trees, JavaGenes evolves graphs representing (currently) molecules and circuits. Java was chosen as the implementation language because the genetic algorithm requires random splitting and recombining of graphs, a complex data structure manipulation with ample opportunities for memory leaks, loose pointers, out-of-bound indices, and other hard to find bugs. Java garbage-collection memory management, lack of pointer arithmetic, and array-bounds index checking prevents these bugs from occurring, substantially reducing development time. While a run-time performance penalty must be paid, the only unacceptable performance we encountered was using standard Java serialization to checkpoint and restart the code. This was fixed by a two-day implementation of custom checkpointing. JavaGenes is minimally integrated with Condor; in other words, JavaGenes must do its own checkpointing and I/O redirection. A prototype Java-aware version of Condor was developed using standard Java serialization for checkpointing. For the prototype to be useful, standard Java serialization must be significantly optimized. JavaGenes is approximately 8700 lines of code and a few thousand JavaGenes jobs have been run. Most jobs ran for a few days. Results include proof that genetic algorithms can evolve directed and undirected graphs, development of a novel crossover operator for graphs, a paper in the journal *Nanotechnology* [Globus, et al. 1999], and another paper in preparation.

Introduction

This paper is a case study of running genetic algorithms written in Java under Condor. To understand the experience and results, it is necessary to have some understanding of both genetic algorithms and Condor. Since this paper is written for the Java Grande 2000 conference, we expect readers to understand the Java programming language and run-time environment.

Genetic Algorithms

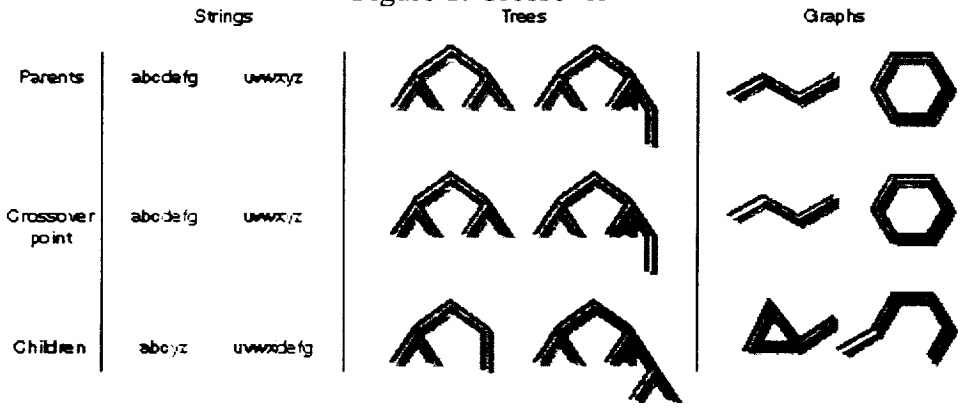
Genetic algorithms seek to mimic natural evolution's ability to produce highly functional objects. Natural evolution produces organisms. Genetic algorithms produce sets of parameters, programs, molecular designs, and many other structures. Genetic algorithms usually solve problems by:

1. Randomly generating a population of individual potential solutions.
2. For each new generation, repeatedly selecting parent individuals at random with a bias towards better individuals and applying transmission operators to produce children. Transmission operators include:
    1. Crossover: each of two parents is divided into two parts and one part from each parent is combined into a child.
    2. Mutation: a single "parent" is randomly modified to generate a child.
    3. Reproduction: a single "parent" is copied into the new generation.
3. Continuing until an acceptable solution is found or exhaustion sets in.

A key issue is what constitutes "better individuals." This is determined by a "fitness function." The fitness function takes individuals (strings, trees, or graphs) as input and returns a number representing the fitness of that individual.

Genetic algorithms differ in their representation of solutions. Bit string representations were used in the first genetic algorithms [Holland 1975], but arrays of floating point numbers, special symbols that generate circuits [Lohn and Colombano 1998], robot commands [Xiao, et al. 1997], and many other symbols may be found in the literature. Strings may be of fixed or variable length. Trees can also be evolved [Koza 1992]. This is usually called genetic programming, because trees are particularly useful for representing computer programs. Many molecules contain cycles, which chemists call rings, and strings and trees don't contain cycles. Therefore, we took the unusual approach of evolving graphs. Graphs are a set of vertices (for example, atoms) and a set of edges (for example, bonds), each of which connects two vertices. In this paper, the term graph does *not* refer to a two dimensional image used for data presentation. Figure 1 depicts crossover using strings, trees and graphs.

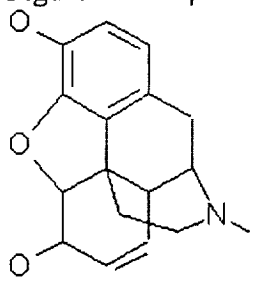
Figure 1: Crossover



Crossover can be applied to strings, trees, and graphs. Note that only graph crossover requires multiple break points and that recombination must work on fragments with different numbers of broken edges. Particularly with large multi-ring molecules, these complex data structure manipulations are more error prone in C/C++/FORTRAN than in Java.

Evolving graphs involves *randomly* splitting arbitrary graphs into two fragments and then recombining fragments. Splitting a complex graph, such as morphine (figure 2) involves complex data structure manipulations that can easily result in various bugs. Java was chosen as the implementation language to minimize these problems.

Figure 2: Morphine



JavaGenes has been used for pharmaceutical drug and digital circuit design. One approach to drug design is to find molecules similar to good drugs. Ideally, a candidate replacement drug is sufficiently similar to have the same beneficial effect but is different enough to avoid negative side effects.To use JavaGenes for similarity-based drug discovery we need a good similarity measure that can score any molecule. [Carhart, et al. 1985] defined such a similarity measure, all-atom-pairs-shortest-path, and searched a large database for molecules similar to diazepam. We use a closely related similarity technique to evolve a population of molecules towards a target drug molecule.

For an excellent review of genetic algorithms and related techniques as of Spring 1997, see [Baeck 1997].

Condor

Genetic algorithms have the fortunate property of being embarrassingly parallel, because fitness function evaluation is usually the most time-consuming step in the genetic algorithm and there is no dependency between fitness function evaluations. Not only can many fitness function evaluations be conducted in parallel, but since genetic algorithms are statistical, it is usually necessary to make many runs to support a hypothesis. In our work, we usually run 31 jobs with the same input parameters, each job differing only in the random number seed. This provides completely trivial 31 way parallelism. Furthermore, we are usually running several experiments at the same time. Thus, it is not uncommon for our project to run 100-200 jobs simultaneously. It's also quite easy to implement parallel fitness function evaluation within a single job, although we haven't found that to be necessary yet.

Embarrassingly parallel programs are a natural match for Condor [Litzkow, et al. 1988]. Condor is a software system that creates a High Throughput Computing environment by effectively harnessing the power of a cluster of UNIX workstations on a network. Although Condor can manage a dedicated cluster of workstations, a key appeal of Condor is its ability to effectively harness non-dedicated, preexisting resources in a distributed ownership setting such as machines sitting on people's desks in offices and labs. We ran JavaGenes on the NAS Condor pool. NAS is the primary NASA supercomputer center [NAS]. Approximately 200 workstations, purchased and used for software development, visualization, email, document preparation, etc., are available for batch processing during idle times. The Condor daemons watch these 200 workstations. When a workstation has been idle for 2 hours, a job from the batch queue is assigned to the workstation and will run until the workstation detects a keystroke, mouse motion, or relatively high non-Condor CPU usage. At that point, the job will be removed from the workstation and placed back on the batch queue. As mentioned before, it's not uncommon to have a few hundred JavaGenes jobs in the queue.

Because a JavaGenes job running under Condor may be killed at any time, the job must save state

(checkpoint) periodically. Condor provides a generic checkpoint/restart facility, but for reasons discussed below, we could not use this facility for JavaGenes. Checkpointing was initially implemented using standard Java serialization. Conceptually, this is relatively simple since the state of a genetic algorithm is simply the current population. However, standard Java serialization turned out to be a serious, but fixable, performance problem, as others have discovered [Wims and Xu 1999]. See the section below on serialization performance.

Approach

JavaGenes was written in 100% pure Java, version 1.1. There were approximately 8670 lines of source, not including the graph layout code (Jiggle) provided by Daniel Tunkelang [Tunkelang 1998]. The graph layout code is used to arrange graph vertices in three dimensions for viewing. We will now discuss the objects implemented in JavaGenes.

Objects

The nouns used to describe genetic algorithms all became classes, including the following:

- Population -- an array of Individuals.
- Individual -- an Evolvable and its fitness.
- Evolvable (nonstandard terminology) -- a data structure capable of being evolved by ChildMaker objects. Currently, only Graph plus subclasses Molecule and DigitalLogicGraph are implemented, but plans for arrays and trees exist.
- FitnessFunction -- these objects have a 'double evaluate(Evolvable)' method that implements the desired fitness function. General purpose FitnessFunction subclasses included weighted sum (of other FitnessFunctions) and MultiplyBy.
- Breeder (nonstandard terminology) -- a class with a 'Population breed(Population)' method that evolves one population into another.
- ChildMaker (nonstandard terminology) -- these objects tell a Breeder how many parents they want (two for crossover, one for mutation), then take an array of parents and produce an array of children.

Another set of classes are responsible for creating, manipulating, and managing graphs:

- Graph, along with subclasses Molecule and DigitalLogicGraph.
- Vertex, along with subclasses Atom, DigitalInput, DigitalOutput, and DigitalDevice (or, and, xor, etc.).
- Edge, along with subclasses Bond and DigitalWire.
- BrokenGraph -- responsible for a graph fragment after crossover splits a graph.
- BrokenEdge -- responsible for an edge broken during splitting.
- VertexProvider and EdgeProvider -- these classes are used during graph generation and mutation to provide random vertices and edges of various types.

In addition, there are several convenience classes:

- Parameters, along with subclasses MoleculeParameters and DigitalLogicParameters. These objects hold all the values that are typically varied from job to job; for example: population size,

maximum number of generations, fitness function, etc. Most of the JavaGenes class files are kept in a jar file, but the parameter files are compiled for each set of jobs and placed earlier in the CLASSPATH. This provides a flexible mechanism (taking advantage of Java dynamic loading) to set input parameters without writing an input file parser.

- InputTokenizer and OutputTokenizer -- these are used to save and restores state. They can read and write integers, doubles, etc.

Free Code

A certain amount of Java code is available for free on the Web. We took advantage of this in two cases. First, the Student T-Test code used in the statistics class was supplied by NWP Associates, Inc. This was a minor, but helpful, convenience. Second, and more important, one must examine the evolved graphs to understand the results. To examine a graph, it must be laid out in two or three dimensions for viewing. In other words, xyz locations for each vertex must be chosen. The graph layout problem is non-trivial. In fact, it is very difficult. Fortunately, Daniel Tunkelang made his Jiggle Java code [Tunkelang 1998] available, and Jiggle has done an excellent job of laying out graphs evolved by JavaGenes. Integration of the two packages was quick and easy. Only one bug was found in Jiggle (1615 lines of source). That bug was an infinite loop, which was found and fixed in a little over an hour.

Development Environment

JavaGenes was developed on a Compaq laptop running Windows 95. Windows was used because the main developer cannot type for significant lengths of time and uses a voice recognition system. The first development environment was Visual Cafe. This was abandoned because the debugger was quite buggy. The second development environment was Superseed. Various problems required periodic re-installation, which in turn caused serious problems with Windows 95. Finally, Borland JBuilder was tried and there have been relatively few problems. CodeWarrior was used occasionally for specific debugging problems. Although some compilers are a bit pickier than others about syntax, no significant problems were encountered moving the code from one development environment to another; or moving the source or class files to the SGI version of the JDK to run experiments.

Condor Support for Java

Condor runs each job in an environment called a "universe." The two most important universes are called "Standard" and "Vanilla." Standard jobs are programs that have been linked with a special Condor version of the C runtime library that mimics the effects of most Unix system calls and adds two kinds of enhanced functionality: remote system calls and checkpointing.

Remote system calls provide a uniform environment to a job running on any workstation on a network. The Condor runtime library replaces system calls with remote procedure calls to a shadow process running on the workstation that submitted the job. The shadow makes the system call on behalf of the remote process and returns the results. For example, the `open` system call sends the name of the desired file to the shadow, which searches for it on the submitter's home workstation. Subsequent `read` and `write` calls access that file over the network. The result is that the job sees the same file-system environment regardless of where it runs, and all file output is captured in files on the submitting workstation.

Checkpointing is also implemented by the Condor runtime library. Each Condor job is run under the

control of a starter process. When a workstation needs to be appropriated for another purpose, the starter sends a terminate signal to the application process. The Condor library catches this signal and sends a complete dump of the state of the process back to the submitting machine, where it awaits its turn to be restarted on another worker machine. This checkpoint file includes a binary dump of the entire virtual memory image of the process. It also includes a record (collected by the remote system calls) of the the current state of the process' interaction with the operating system kernel. For example, for each file opened by the job, the checkpoint file records the name of the file and the current offset within the file (the "seek pointer"). Condor can also be instructed to send a "checkpoint" signal to the starter at periodical intervals. The starter responds to this signal by suspending the application, checkpointing it, and then allowing it to continue. Under some circumstances, Condor may send other signals to the starter, asking it to suspend the application, resume it, or kill it without giving it a chance to checkpoint.

Neither remote system calls nor checkpointing require any source-level modifications to the application program, but they do require it to be re-linked with a special version of the system libraries. They also impose some restrictions on the set of system services available to the job. In particular, Condor does not currently support Standard jobs that use kernel-level threads. Programs that cannot be re-linked or that do not meet these requirements must be run in the Vanilla universe. An arbitrary executable program can be run as a Vanilla job, but any I/O operations will access the file system of whatever machine the job happens to be running on, and if the workstation is pre-empted (for example, by an interactive user), the job is simply killed and restarted from the beginning on another workstation. At NAS, in general, Condor jobs do not have permission to use the local disk on the worker workstation.

Most of the JavaGenes runs described in this paper used the Vanilla universe. The Java Virtual Machine (JVM) from Sun Microsystems was available to us only in binary (pre-linked) form. We tried the Kaffe open-source JVM, but found that it had bugs that prevented us from running JavaGenes correctly for more than a few generations. Moreover, both JVM's use certain facilities -- notably kernel-level threads -- that are not supported in Condor Standard jobs. Fortunately, the remote system call facility was not necessary in our environment, since we use the Network File System (NFS), which provides a uniform interface to files from all workstations. In this environment, the "job" submitted to Condor is a tcl script that calls the JVM, with the name of the application class supplied as a command-line argument. From the point of view of the JVM, the class files that comprise the application are simply data files that appear to be on the local disk of the worker machine through the magic of NFS. Similarly, NFS is used to create output files on the submitting workstation.

The lack of automatic checkpointing was a more serious problem. As mentioned earlier, we tried two different application-specific checkpointing strategies. Both involved periodically invoking a checkpoint method that saves the state of the computation into a file. JavaGenes invokes this method when it is relatively quiescent so that all relevant state is concentrated in a few objects. If a workstation is preempted before the program finishes, it is killed and restarted "from the beginning." However, at startup, jobs look for the checkpoint file (via NFS) so they can initialize state and continue from the last checkpoint. Thus, a job that is killed and restarted loses only the work it did between its most recent checkpoint an the time when it was killed.

To provide better support for JavaGenes and other Java Grande applications, the Condor project has been developing a Java universe. To run under this
universe, a Java program must implement the Checkpointable interface:

```
public interface Checkpointable extends Serializable {
```

```
    void start(String[] arguments);
    void restart();
    void beforeCheckpoint();
    void afterCheckpoint();
    void setCheckpointer(Checkpointer c);
}
```

Each universe has its own kind of starter. The starter for the Java universe is written in Java and extends `java.lang.ClassLoader`. A "job" in the Java universe is a class file. We assume that each worker machine has a JVM installed locally, but otherwise do not require any network file system or uniform file environment. The Java starter loads all required classes over the network by communicating with a Java version of the shadow using a Java RMI (Remote Method Invocation) interface. It creates an instance `prog` of the application program class and calls either its `prog.start` or `prog.restart`, depending on whether there is an existing checkpoint file from an earlier run of this job. The starter also creates an instance `cp` of a Checkpointer object, passing `prog` to its constructor, and calling `prog.setCheckpointer(cp)` so that `prog` and `cp` can refer to each other. When the starter receives a terminate or checkpoint signal, it calls `cp.checkpointWhenPossible`, which sets a flag indicating that a checkpoint has been requested. It does not force an immediate checkpoint because the application object may not be in a "quiescent" state in which checkpointing is convenient. The application itself is expected to call `cp.ok` periodically. If no checkpoint has been requested, this method simply returns without doing anything. However, if a checkpoint request is pending, the checkpointer uses Java serialization to save the state of the object by calling `writeObject(prog)`. It also calls `prog.beforeCheckpoint` before the checkpoint and `prog.afterCheckpoint` after.

The `writeObject` method saves all of the non-transient fields of the object, as well as all objects pointed to by those fields, all fields of those objects, etc. It does not, however, save anything from the runtime stack -- that is, the values of local variables. It is the responsibility of the application to update the non-transient fields of the application object to reflect the complete state of the application either before calling `cp.ok` or in the method `beforeCheckpoint`. If these updates are costly, they should be in `beforeCheckpoint` because this method is only called if the checkpoint is actually performed. The `beforeCheckpoint` and `afterCheckpoint` methods also provide hooks for application-specific performance monitoring, such as determining the amount of time spent checkpointing. A typical application might look like this:

```
class Application implements Checkpointable {
    private Checkpointer checkpoint;
    private GlobalState state;
    private int lastStepCompleted;
    private transient Vector intermediateResults;
    private int iterations;  // set by initializeState
    private PerformanceStatistics statistics;
    public void setCheckpointer(Checkpointer checkpoint) {
        this.checkpoint = checkpoint;
    }
    public void start(String[] args) {
        state.initialize(args);
        intermediateResults = new Vector();
        iterateFrom(0);
```

```
    }
    public void restart() {
        iterateFrom(lastStepCompleted);
    }
    private void iterateFrom(int start) {
        for (int i = start; i < iterations; i++) {
            intermediateResults.add(oneStepOfAlgorithm());
            lastStepCompleted = i;
            checkpoint.ok();
        }
        printResults();
    }
    public void beforeCheckpoint() {
        statistics.startTimer();
        state.update(intermediateResults);
        intermediateResults.clear();
    }
    public void afterCheckpoint() {
        statistics.stopTimer();
    }
    // Methods initializeState, oneStepOfAlgorithm,
    // startTimer, etc. omitted for brevity.
}
```

The Java universe provides several advantages over the Vanilla universe for Java applications:

- It supports remote system calls, so that it does not depend on the availability of a network file system (and uniform conventions for mount points).
- It automates more of the tasks necessary for checkpointing and recovery.
- It allows Condor to decide when to checkpoint a job, only requiring the application code to indicate when a checkpoint is safe.
- It allows a job to migrate among hardware platforms during its lifetime. A job in the Java universe is comprised of class files and a checkpoint file, all of which are platform-independent.

A prototype version of the Java universe successfully ran JavaGenes but it was not stable enough for production use at the time the experiments described in this paper were run. Also, the Java universe uses Java serialization to checkpoint jobs. Our experience indicates that standard Java serialization is too slow for JavaGenes to use.

Results

The bottom line for JavaGenes on Condor was to run many jobs to conduct the experiments necessary to understand the application of genetic algorithms to graphs. In this, we were successful. Thousands of jobs were successfully run, one genetic algoriths paper was published [Globus, et al. 1999], and another is in preparation [Globus, et al. 2000]. JavaGenes does a fairly good job of evolving pharmaceutical drug molecules, but can only evolve trivial circuits so far. We now examine the difficulties and benefits of using Java for our application.

Java Con

Java Serialization Performance

It was eventually discovered that creating serialization files and reading them to implement checkpoint/restart could take has much as three hours wall clock time. Although Java serialization is extremely general-purpose, any objects can be serialized and the format is CPU independent, it is difficult to understand why three hours is needed to serialize a few hundred graphs, some ancillary objects, and a few thousand real numbers (the data). In any case, jobs, under these conditions, made no progress. In addition, the serialization files were around 25 MB and there were often two files per job. Because jobs could be interrupted in the middle of serialization, JavaGenes wrote state information into a temporary file and, when finished, moved it to the permanent location. With long serialization times, most jobs were interrupted in the middle of writing the serialization file. Thus, the full checkpoint file and a partial checkpoint file were on disk most of the time. With hundreds of jobs, disk usage became substantial. To solve the performance problem, Java serialization was abandoned and new code written to save the state of the computation and read it back from disk. Development took approximately two days and one bug was found and fixed a few days after development was "complete." Most checkpoint files are now less than 1 MB. Checkpoint write and read usually take around 10 seconds, but can require up to about five minutes (including network delays). This was the only performance problem that required changes to the code.

Checkpointing with Serialization

Besides having performance problems, checkpointing with serialization must be handled with care. In particular, it is sometimes error prone. It is necessary to make sure that all of the necessary program state is saved. In the case of genetic algorithms, nearly all the state that must be saved is contained in the population. However, if checkpointing is allowed in the middle of constructing a new generation, then the loop index indicating how much of the next generation has been constructed must also be saved. Also, it is difficult to write code that can be interrupted at any time and saved to disk. Therefore, checkpointing is restricted to those points in the code where analysis can prove that saving and restoring will not cause problems.

One other fairly serious problem arose. In the initial implementation, JavaGenes started a new random number generator after restart. Because of this, evolution did not follow the same path taken after the last checkpoint. In other words, the genetic algorithm was controlled by different random numbers and therefore searched a different part of the search space. That made the algorithm appear more efficient than it was, because the number of generations to find a specific target was used as the efficiency measure. Consider the case where a job ran for 100 generations up to a checkpoint and 20 more generations before being killed. When the job was restarted, twenty different generations would be constructed and one of the new generations, say the 10th, might find the target. Therefore, it would appear as if the target was found in 110 generations, but actually 130 were necessary. It was therefore necessary, at job restart time, to restart the random number generator with the same seed and then execute the random number generator the number of times it was executed up to the last checkpoint. This insured that a job followed the same evolutionary path regardless of checkpoint history.

JIT Essential

Some very simple tests indicated that jobs run under the SGI just-in-time compiler were approximately

20 times faster than when the JIT was disabled.

Double Read/Write Java Libraries Bug

It was discovered that the standard Java libraries will write out certain double numbers in an ASCII form that is not tolerated as input by the same libraries. In particular, `System.out.write("" + aDouble)` will write out values such as `NaN` and `Infinity`, but `Double(String)` will throw an exception if these strings are passed as the argument.

Java Pro

Porting

We never had any problems porting the source or class files between Java environments.

Bugs

Compared to developing in C or C++, there were few bugs and they were easy to find and fix. Basically, almost all the bugs were logic problems, most of which were found and fixed while single stepping through the code with the graphical debuggers provided by the IDEs. The uncontrolled pointers, index out of range, memory management, and similar bugs extremely common in the primary programmer's 20 year C and C++ development experience almost never occurred. These bugs are also, usually, quite difficult to find. Reducing the number of bugs dramatically lowered development time, at least subjectively.

Memory Management

Memory management was trivial. Only one bug was encountered when we neglected to create a new xyz array in `class Vertex` during cloning. Before the bug was fixed, after graph layout, all the atoms ended out on a single point.

Performance

Other than the serialization problem, performance was not a major issue. This was, in part, because Condor supplies us with lots of nearly free CPU cycles, but also because Java performance has been reasonable (although certainly not exceptional).

In short, we were happy with Java for this application. While there is some run-time penalty, more rapid code development and more reliable end-product software is well worth the extra CPU cycles. Since cycles are always getting cheaper, and programmers and support staff seem to be getting more expensive, we expect Java to do well in the coming years.

Future Work

The main task before JavaGenes is to incorporate more chemical knowledge into the fitness functions, because JavaGenes usually evolves molecules that are not physiologically stable and that would be difficult, or perhaps impossible, to synthesize. To support more Java applications under Condor and

more diverse environments, the prototype Condor Java universe described in the section on Condor Support for Java needs to be brought up to production status. The performance problem with Java serialization needs to be solved more generally, or applications must reimplement serialization. An alternative approach would be to rebuild a Java Virtual Machine as a Standard universe job. This approach has the advantage of requiring less hand-modification of Java applications, but requires resolution of certain technical and licensing issues.

Acknowledgments

Many thanks to Daniel Tunkelang, formerly of Carnegie Mellon, for providing his graph layout code [Tunkelang 1998]. Thanks to NWP Associates, Inc. for providing their Student T-Test code. Thanks to Rich McClellan, University of California at Santa Cruz, for providing the mol file reading and atomic element code. Thanks to Gail Felchle for much of the graphics art work. Thanks to the Condor team at the University of Wisconsin for their support. This work was funded by NASA Ames contract NAS 2-14303 and NASA-Ames Cooperative Agreement No. NCC 2-5323.

References

[Baeck, et al. 1997] Thomas Baeck, Ulrich Hammel, and Hans-Paul Schwefel, "Evolutionary Computation: Comments on the History and Current State," *IEEE Transactions on Evolutionary Computation*, volume 1, number 1, pages 3-17, April 1997.

[Carhart, et al. 1985] Raymond Carhart, Dennis H. Smith, and R. Venkataraghavan, "Atom Pairs as Molecular Features in Structure-Activity Studies: Definition and Application," *Journal of Chemical Information and Computer Science*, volume 23, pages 64-73.

[Globus, et al. 1999] Al Globus, John Lawton, and Todd Wipke, "Automatic Molecular Design Using Evolutionary Techniques," *Sixth Foresight Conference on Molecular Nanotechnology*, Sunnyvale, California, November 1998 and *Nanotechnology*, volume 10, number 3, September 1999, pages 290-299.

[Globus, et al. 2000] Al Globus, Sean Atsatt, John Lawton, and Todd Wipke, "JavaGenes: Evolving Graphs with Crossover," in preparation.

[Holland 1975] John H. Holland, *Adaptation in Natural and Artificial Systems,* University of Michigan Press.

[Koza 1992] John R. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, MIT Press, Massachusetts.

[Litzkow, et al. 1988] M. Litzkow, M. Livny, and M. W. Mutka, "Condor - a Hunter of Idle Workstation," *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104-111, June1988. See http://www.cs.wisc.edu/condor/.

[Lohn and Colombano 1998] Jason D. Lohn and Silvano P. Colombano, "Automated Analog Circuit Synthesis Using a Linear Representation," *Second International Conference on Evolvable Systems: From Biology to Hardware*, Springer-Verlag, 23-25 September 1998.

[NAS] http://www.nas.nasa.gov/home.html.

[Tunkelang 1998] Daniel Tunkelang, A Numerical Optimization Approach to Graph Drawing, Dissertation, Carnegie Mellon University, School of Computer Science, December 1998.

[Wims and Xu 1999] Brian Wims and Cheng-Zhong Xu, "Traveler: A Mobile Agent Infrastructure for Wide Area Parallel Computing," ACM 1999 Java Grande Conference, San Francisco, California, 12-14 June 1999.

[Xiao, et al. 1997] Jiang Xiao, Zbigniew Michalewicz, Lixin Zhang, and Krzysztof Trojanowski, "Adaptive Evolutionary Planner/Navigator for Mobile Robots," *IEEE Transactions on Evolutionary Computation*, volume 1, number 1, pages 18-28, April 1997.