RC25036 (W1008-051) August 17, 2010
Computer Science

IBM Research Report

Harnessing Emergence for Manycore Programming: Early Experience Integrating Ensembles, Adverbs, and Object-based Inheritance

David Ungar
IBM Research Division
davidungar@us.ibm.com

Sam S. Adams
IBM Research Division
ssadams@us.ibm.com

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

Harnessing Emergence for Manycore Programming: Early Experience Integrating Ensembles, Adverbs, and Object-based Inheritance

David Ungar
IBM Research
davidungar@us.ibm.com

Sam S. Adams
IBM Research
ssadams@us.ibm.com

Abstract

We believe that embracing nondeterminism and harnessing emergence have great potential to simplify the task of programming manycore processors. To that end, we have designed and implemented Ly, pronounced "Lee", a new parallel programming language built around two new concepts: (i) ensembles which provide for parallel execution and replace all collections and (ii) iterators, and adverbs, which modify the parallel behavior of messages sent to ensembles. The broad issues around programming in this fashion still need investigation, but, after our initial Ly programming experience, we have identified some specific issues that must be addressed in integrating these concepts into an object-based language, including empty ensembles, partial message understanding, non-local returns from ensemble members, and unintended ensembles.

Categories and Subject Descriptors D.3.2 [**Programming Languages**]: Language Classifications - Object-oriented languages, Nondeterministic languages: D.3.3 [**Programming Languages**]: Language Constructs and Features - Concurrent programming structures, Inheritance : D.1.3 [**Programming Techniques**]: Concurrent Programming - Parallel programming : D.1.5 [**Object-oriented Programming**]

General Terms Design, Human Factors, Languages.

Keywords *object-based inheritance; ensembles; adverb; multicore; manycore*

1. Introduction

Within the next decade, nearly every CPU will have dozens to hundreds of general-purpose cores. How can the vast majority of programmers, perhaps expert in applications, but not so well-versed in the arcane art of parallel programming, easily exploit such extreme parallelism?

Many in our field are exploring ways in which the programmer could continue to write deterministic parallel programs without over-specifying the order of events, or ways in which the runtime enforces determinism [1, 2]. We believe that even such a limited attempt to specify essential determinism will likely run into scaling problems. Others in our field follow a functional approach that frees the result from dependencies upon the order of execution. But if mutable state is needed, a monad is introduced, which makes the temporal dependency quite explicit [3]. We believe this approach to be fruitful, but limited in its ability to directly model systems that are naturally viewed as possessing manifold state.

For example, consider a simulation of a flock of birds, each independently choosing its own path, moment by moment.

At any given time, it seems natural to us that each bird has a state, for instance a definite location. Where is that bird, now? We search for a programming paradigm that can easily model such an interpretation of a massively parallel bird flock. Still others in our field turn to actors to order the chaos of rampant parallelism [4, 5]. However, actors seem to us to merely defer the problem by a constant factor: since an actor can contain mutable state, and since an actor may receive messages from other actors in various orders, we believe that actors do not solve the fundamental problem of taming nondeterminacy. Consider an actor A that holds a bank balance, an actor B that sends A a message to double the balance, and an actor C that sends A a message to add $5 to the balance. The final result is as indeterminate as the order of message passing. Since our field began we have struggled to impose determinism on complex computations, and our languages and programming models have co-evolved with that goal. Significantly easing the challenge of programming with massive parallelism requires a complete shift: instead of resisting, it is time to embrace nondeterminism.

Nature provides us with many examples of massively parallel systems. Without the benefit of any global synchronization at all, these systems manage to solve complex problems and achieve robust behavior. Consider a flock of birds (figure 1), a school of fish, an ant colony, a termite mound, a developing embryo, or even, as some

Figure 1. A starling flock

suspect, the phenomenon of consciousness in the human brain [6]. In each case, a large number of individuals, each following local rules, interact asynchronously to exhibit coordinated, robust, distributed behavior of a higher degree of complexity than that of the individuals. This phenomenon is known as *emergence.*

Could a new programming paradigm based on nondeterminism and emergence make it easy for ordinary application programmers to exploit the massive parallelism of future manycore processor chips? To answer that question, we have built a testbed comprised of:

- a Smalltalk virtual machine rewritten to support multithreading and object migration [7], hosting the Squeak IDE;
- a new programming language (Ly) featuring ensembles and adverbs, with object-based inheritance and a JavaScript-like syntax; and
- an integrated development environment including a source-management system and a source-level debugger.

Although still somewhat immature, our testbed, which runs on dual-core Mac laptops; 8-core, 16-hyperthread Mac Pros; and a 64-core Tilera manycore processor, has given us enough experience with these ideas to demonstrate an algorithm running 50 parallel threads with no application-level synchronization. More importantly, this effort has uncovered a number of interesting language issues. We intend to open-source our system so that others may experiment with manycore Smalltalk and our language. This short paper briefly describes our language, its implementation, and those thorny issues.

2. Ensembles and Adverbs

How do you go from a flock of birds to a programming language? Gazing at a flock, you see a constantly changing whole; blink, and you see a collection of individuals, each operating in parallel with its neighbors. Our language includes a concept that models this experience, called an *ensemble*. Our ensemble is a bit like an APL array; it can be referenced as one thing, yet performing an operation on it (i.e. sending it a message) causes each *member* to perform the operation in parallel. Unlike an APL array, which can only contain immutable values (numbers or characters), the members of an ensemble can be mutable objects, or even other ensembles.

Once *ensemble* is admitted to the pantheon of first-class computing concepts, questions arise: For example, in performing a computation over an ensemble, there are many options as to which members are involved: every member, the closest member geographically on the cores, some subset of members? Once the computation has occurred, how shall the results be returned? Shall they be bundled into a resultant ensemble? Discarded? Or reduced by some operation (e.g. averaging)? In order to separate out the specification of these and other questions of execution strategy, we add another concept to our model of computation: an *adverb.* In most object-oriented languages, the tuple required to perform a computation would be receiver-selector-argument(s). In our model, an adverb is added to that mix. Our current syntax denotes an adverb by appending a double-minus ('--') to the list of argument expressions, and following the double minus with an expression supplying the adverb. Perhaps even each argument could have its own adverb. We expect the most common case to be the parallel computation on every member, returning an ensemble of results. Therefore no adverb is needed in that case.

Finally, it is necessary to be able to perform a computation on an ensemble-as-a-whole. For example, sending *size* to a flock would normally return an ensemble of the sizes of each bird; asking the flock for its size (as a whole) would be a different kind of request. We add a second message-passing syntax to indicate this sort of computation: in `aFlock..size()` the double-dot indicates that the message is to be sent to the ensemble-as-a-whole, instead of to its members. This new variety of *reflection* completes the concepts we explore as a means to embrace nondeterminism.

3. Ly: An experimental programming language

What sort of language would naturally encourage a programmer of massively parallel hardware to embrace nondeterminism via the concepts of ensembles and adverbs? Such a language would have a familiar syntax, such as JavaScript's, and a simple yet powerful object-based inheritance model, such as Self's [8] (more below). Ensembles would be as easy to use as regular objects. Therefore, the concise and familiar message-passing syntax would also be used for ensemble message sends, for example `aFlock.turnLeft()`, would tell every bird in the flock (ensemble) to turn left simultaneously.

3.1 Object = Ensemble?

Coming from an object-oriented culture, we wanted to bring along all the benefits of that paradigm, so the next decision in the language-design process was how to integrate ensembles and objects. For the sake of uniformity, we wanted a singleton ensemble, for example a flock of one bird, to be indistinguishable from an object, the bird in our example. Following this train of thought, every object would be (in effect) an ensemble containing only itself as a member. Every message would burrow into its destination's member list and go one deep. For example, when sending *name* to a Parrot object named Polly, the message would be received by the parrot, and since she would also be an ensemble, the message would get redispatched to Polly's members, consisting solely of the bird herself. At that point, further redispatching would somehow be avoided, and Polly would answer "Polly", which would itself be an ensemble containing itself (i.e. the string "Polly"). Since Polly the parrot would be an ensemble, the result(s) of the *name* message would have to be returned as an ensemble, and so the one result (the string "Polly", itself an ensemble) would have to get wrapped into a singleton ensemble resulting in a doubly-nested ensemble containing the string "Polly". But this result would have an undesired level of nesting. The gymnastics required to deal with these potentially infinite member-regresses became challenging enough that we backed off: a singleton ensemble in Ly is not the same as the member object.

Still striving for unification, we next considered allowing every Ly object to be an ensemble. In this model, every object would contain slots, a single link to its parent (we hope to avoid multiple inheritance), and a one-to-many link to its members. In this scenario, we faced a choice between two equally plausible alternatives: searching an ensemble's slots before its members, or searching its slots after its members. For example, suppose our flock of birds includes a centerOfGravity method, which would compute the center of gravity of the flock, and furthermore that each bird also includes a centerOfGravity method, returning its own center of gravity. On one hand, the flock's slots could be searched first, so that centerOfGravity() would invoke the flock's center of gravity. On the other hand, the flock's members could be searched first, in order to obtain an ensemble of the individual birds' centers of gravity for a calculation of the polar moment of the flock. Because the choice of lookup order between slots and members seemed arbitrary and hard to remember, and because there might be useful cases for either choice, we decided to design the language so that ensembles could not have slots. Rather, using the power of object-based inheritance, ensembles could be parents of objects and search the slots first, or ensembles could be children of objects and search the members first.

In summary, a *reference* points to either an *object* or an *ensemble*. An *object* contains a *parent reference* and zero-or-more *slots.* A *slot* contains a *name* (i.e. a character string) and (a reference to) its *contents.* An *ensemble* contains a *parent reference* and zero-or-more (references to its) *members.* By arranging inheritance hierarchies, a programmer can achieve whatever look up behavior is desired.

3.2 Inheritance

Given the cache size and memory bandwidth issues in a manycore system, it seems practical to reify shared behavior by framing inheritance as a way to share parts of objects (just as in Self [8]). So, when a message is sent to an object, if the object has no matching slots, the lookup continues in its ancestors, but even if the match is found in an ancestor, the object that was the original destination of the message is the one bound to *self* or *this*. In contrast, when a message is sent to an *ensemble,* the result will be N parallel invocations, one per member, with each member bound to *self* or *this*. An ensemble may have a parent; if its members do not understand a message, the lookup then proceeds up the ensemble's parent chain, resulting in just one invocation, with the ensemble itself as the receiver. This scheme attempts to integrate object-based inheritance with ensembles while avoiding unintended consequences.

But the intended consequences of this model turned out to be non-trivial: when the lookup dives into an ensemble, it becomes a parallel lookup from each receiver. But if no matches are found, it must back out, reset the receiver to the original object, and continue up the ensemble's parent chain (figure 2).

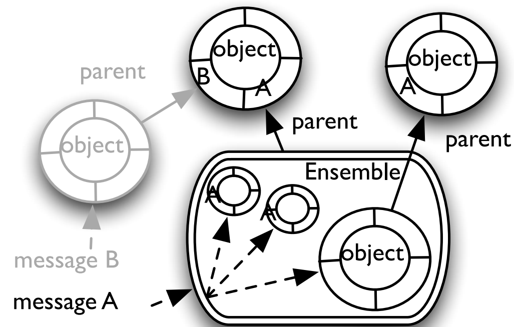

Figure 2. If members don't understand a message, try their parents first, then backtrack to the ensemble parent if necessary.

Figure 3 illustrates this complexity with an example: a message sent to the bottom-left object will first search its slots. If a match is found, there will be one invocation with the bottom-left object as receiver. Next, the members of the left-hand flock are searched, including their ancestors. If matches are found, there will be many parallel invocations, one per bird. If no matches are found, the flock defaults object will be searched. If a match is found, there will be one invocation, with the bottom-left object as receiver. Thus, the bottom-left object allows the turnLeft and turnRight messages to be caught before each bird turns left or right. The flock defaults object allows a default species method to be defined for the ensemble. If the birds understand species, then sending species to a flock will

return an ensemble of species. But if they do not, the default species method will invoke the error code.

Pseudocode Description of Ly Lookup in Ly

In order to examine the interesting language issues that arise from Ly's lookup semantics, it is necessary to illustrate those semantics, and this section does so with pseudocode. Ly syntax is used to help the reader gain a feel for the syntax as well as the semantics. The pseudocode ignores many of the features of our actual implementation, including:

- message sends to implicit self,
- message sends to super,
- message sends with an adverb,
- message sends with the double-dot (to the ensemble as-a-whole),
- compile-time lookups (an optimization),
- inheritance cycles (e.g. an object that is its own grandfather),
- membership cycles (i.e. an ensemble that transitively contains itself),
- the extra complexity of maps (an optimization),
- ensembles of ensembles,
- empty ensembles, as discussed below,
- message arguments (the argument count is also used to lookup a method), and
- a pluggable architecture that lets us experiment with different representations of Ly objects.

In the following code, the lookup operation returns an *ensemble* of results, which can then be used for method invocations. Each result will need fields to hold the method to be invoked and the receiver upon which to invoke it:

```
object LookupResult {
   var receiver, var method;
   // In Ly, the empty parentheses can be elided from a
   // method call with no arguments as in "super.new" below
   function new(r, m) {
      var x = super.new;  x.receiver = r;  x.method = m;
      return x; }
}
```

In order to represent a slot in a Ly object, our pseudocode defines an implementation object to hold a slot's name, its contents, and a function that returns a lookup result if the name matches the selector:

```
object Slot {
   var name, contents;
   function result_if_I_match(receiver, selector) {
      return name == selector
          ?  LookupResult.new(receiver, contents)  :  nil;
   }
}
```

Finally, here are the objects that implement a Ly object and a Ly ensemble. Common attributes are factored out into LyBase:

```
object LyBase {
   // state & code common to objects & ensembles
   var parent; // nil if no parent
   // Entry point for a lookup;
   // This object or ensemble is both the receiver and
   // the place to start looking.
   // Result is an ensemble of results
   function lookup(selector) {
      return lookup( this, selector);
   }
   // Flexible lookup method; look here,
   // but receiver may be somewhere else
   function lookup(receiver, selector) {
      // local_results is an ensemble
      var local_results = lookup_here(receiver, selector);
      // The dot-dot syntax below redirects the isNotEmpty
      // message to the ensemble-as-a-whole, rather than
      // to each of its members.
      if (local_results..isNotEmpty) return local_results;
      if (parent != nil)
         return parent.lookup(receiver, selector);
      return  Ensemble.new // empty ensemble
   }
}
// a Ly object with slots
object LyObject extends LyBase {
   var slots; // holds an ensemble of distinctly-named Slots
   function lookup_here(receiver, selector)  {
      // The double dash, '--', after the last argument
      // signifies the beginning of an adverb.
      // The 'ignoringNils' adverb filters out the nils
      // from the result ensemble.
      return slots.result_if_I_match(receiver,  selector
                                          -- ignoringNils);
   }
}
// a Ly ensemble with members
object LyEnsemble extends LyBase {
   // holds an ensemble implementing the one-to-many links
   var members;
   function lookup_here(selector, receiver)  {
      // In this case, the receivers will be the member objects.
      return members.lookup(selector);
   }
}
```

Lookup starts by sending the lookup message with one argument, the selector, to the receiver of the message. That method (inherited via LyBase), sends the two-argument lookup message to itself, in order to start searching locally. It also passes in the receiver of the eventual invocation, which will ultimately be bound to "this". If the receiver of the two-argument lookup message is an *object,* its *slots* are searched, and if a match is found, a singleton ensemble holding one result is returned. If no match is found, the search continues with the parent. But if the receiver of the two-argument lookup message is an *ensemble,* its *members* are searched, and the receiver(s) of the eventual invocation (s) is reset to correspond to each member. Only if no matches are found in the members does the lookup continue up the ensemble's parent link with the original receiver.

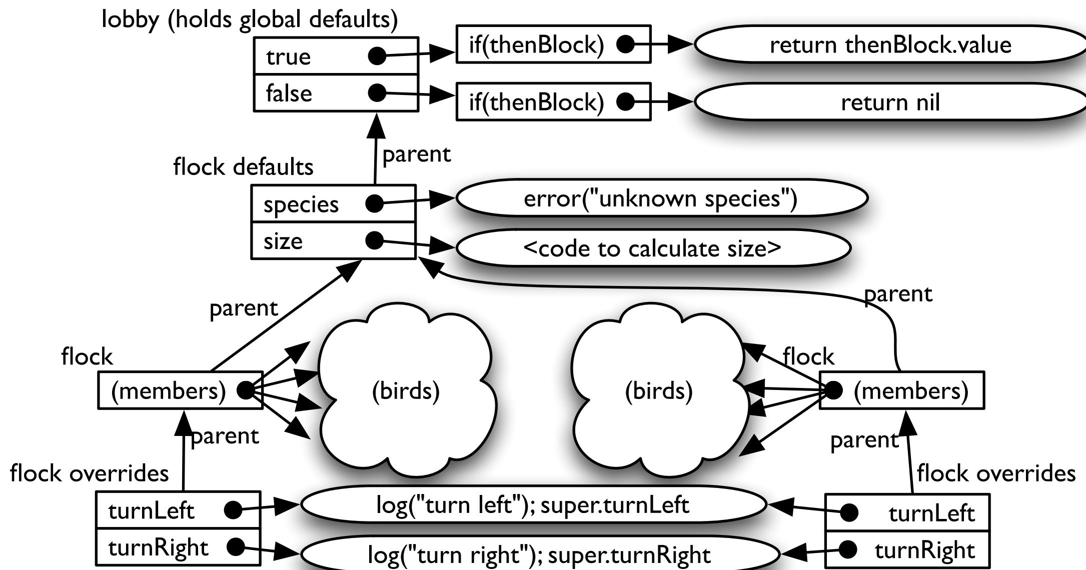

Figure 3. A detailed example of inheritance.

<table>
<tr><td>Smalltalk IDE</td><td>Ly IDE</td><td>Ly Source-to-Smalltalk Compiler</td><td>Ly Runtime</td><td>Visualization tools</td></tr>
<tr><td colspan="5">Squeak Smalltalk Virtual Image including class libraries</td></tr>
<tr><td colspan="5">Our Renaissance Smalltalk Virtual Machine: multithreading & object migration</td></tr>
<tr><td colspan="5">Operating system: Linux, or OS X</td></tr>
<tr><td colspan="5">Processor: 64-core Tilera, 16-hyperthread Intel, or 2-core Intel</td></tr>
</table>

Figure 4. The layers of our system

4. Implementation

Using our Smalltalk testbed, we implemented Ly with a hand-written parser designed to give good error messages, a compiler to transform parse nodes into Smalltalk objects, and an interpreter that executes the Ly program represented by the Smalltalk objects. The Ly interpreter, written in Smalltalk, runs atop the Smalltalk VM interpreter, written in C++, which in turn runs atop various platforms: the Tilera hardware, Intel Linux boxes, Mac Book Pros, and Mac Pros. The Mac Book Pro affords two-way parallelism, the other Intel platforms afford 16-way (with hyperthreading), and the Tilera system runs with 56 independent threads of execution. Figure 4 illustrates the layers.

The double-interpretation (i.e. many Smalltalk operations per Ly operation) resulted in intolerable performance, so we implemented several optimizations:

- the capability to use Smalltalk Point and Number objects instead of Ly-level points and numbers,
- compile-time lookups for slots in the current activation record, and
- maps and runtime-lookup caches to transform each run-time lookup from a sequence of hash table probes to an access into a linear array at a fixed offset.

These optimizations improved performance sufficiently to let us implement a simple version of the Boids flocking algorithm [9, 10] with 50 simulated birds ("boids") flying on 56 Tilera cores (figure 5). Boids is an interesting test application for manycore parallel programming in that there is an ever-changing mix of data parallelism and task parallelism that will stress our programming model and virtual machine implementation.

5. Environment

We have also implemented most of an IDE for Ly using Squeak's MVC framework (figure 6). It includes:

- a workspace, in which one can edit, parse, and execute Ly code;
- an object explorer, which allows one to examine and change the contents of Ly object slots and methods;
- a snippet browser, which allows one to store Ly code as text, leveraging the Smalltalk change-management facilities; and
- a source-level debugger, which shows the Ly invocation stack at the Ly source level, and has facilities for single-stepping, etc. The debugger relies on the underlying Smalltalk facilities and a runtime map from the Smalltalk-level execution state to the Ly-level execution state.

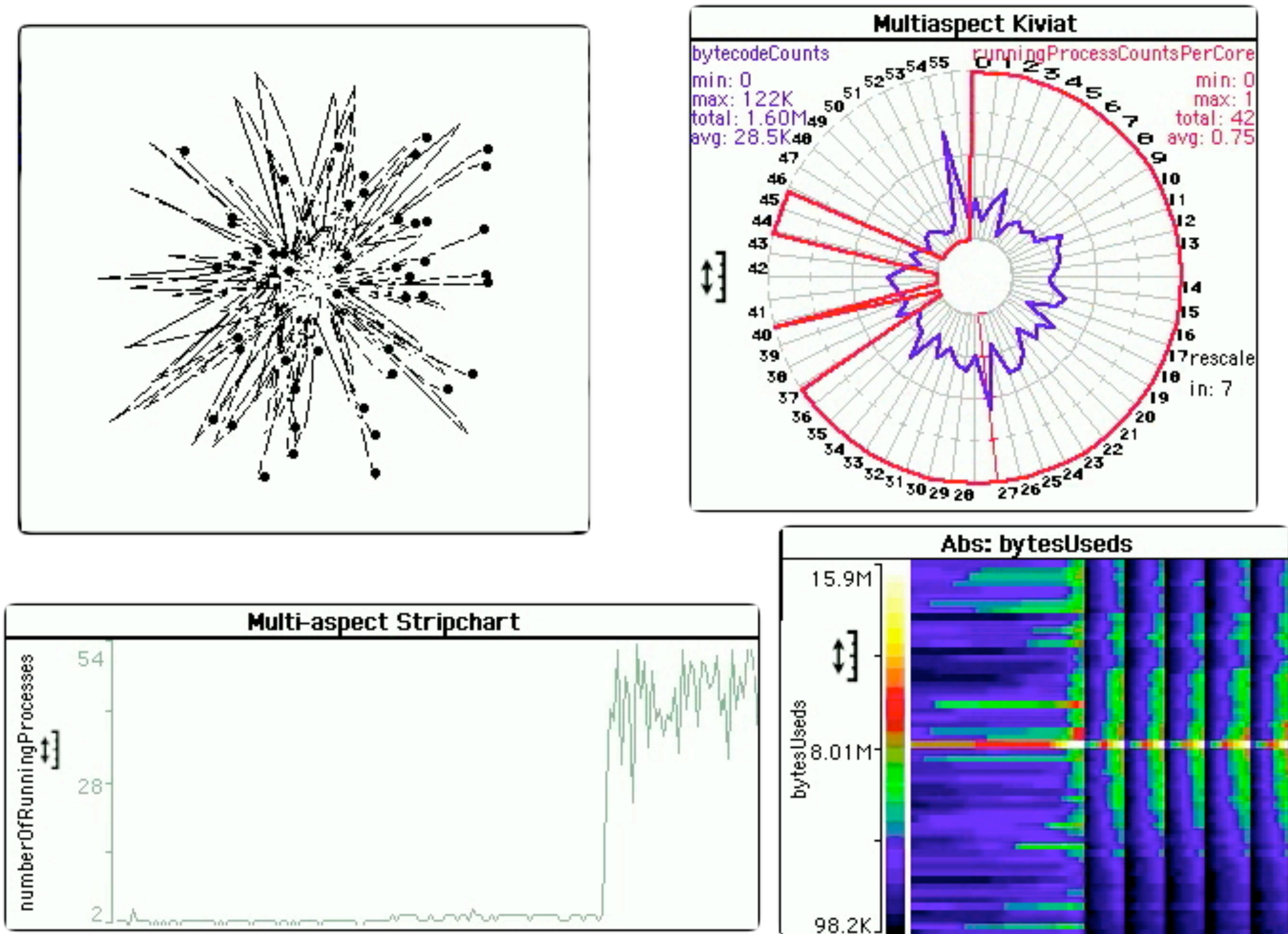

Figure 5. 50 Boids running on 56 cores. Clockwise from the top left: the graphical output of the Boids simulation, the instantaneous execution activity per core, the degree of parallelism vs time, and the occupancy of each core's heap vs time. (Time approximated by samples, so that GC appears to be instantaneous.)

6. Programming Experience in Ly

As of May, 2010, we have written approximately 1,500 lines of Ly code, including many regression tests, a few versions of Boids, and four adverbs: "totally," "pairwise," "serially," and a version of "serially" for ensembles of Smalltalk objects. Figure 5 shows 50 Boids running in 50 threads on 50 cores with several of our visualization tools. This program has no application-level synchronization, yet enjoys 50-way concurrency.

Despite the optimizations described above, Ly's performance on a manycore CPU with a relatively slow memory system led us to investigate other implementation techniques in the past few months, instead of writing many other programs in Ly. However our experience already has uncovered some interesting issues with this model of computation. We do not have all the answers as of yet, but uncovering the following questions may be the most interesting result of our efforts to date.

6.1 Empty Ensembles

For the sake of consistency, a programmer would expect the behavior of an ensemble to remain the same as its membership declined. For example, given a flock of birds, he would expect the `turnLeft` message to cause each bird to turn left, and, given a flock of no birds, he would expect the `turnLeft` message to just do nothing (assuming the ensemble's parent has no `turnLeft` slot). However, given the dynamically-typed nature of Ly, this expectation means that *any* message could be sent to an empty ensemble and the system would just silently do nothing. Later, if a member lacking the method in question were to join the empty ensemble, it would cause a previously running program to incur a "message not understood" error!

What if the ensemble's parent does have a `turnLeft` slot? In that case, the transition between an empty ensemble and a singleton ensemble would involve a switch from the method in the parent's `turnLeft` slot to the method in the member's. Such a change could be an unpleasant surprise. These scenarios suggest that Ly's current design is not satisfactory.

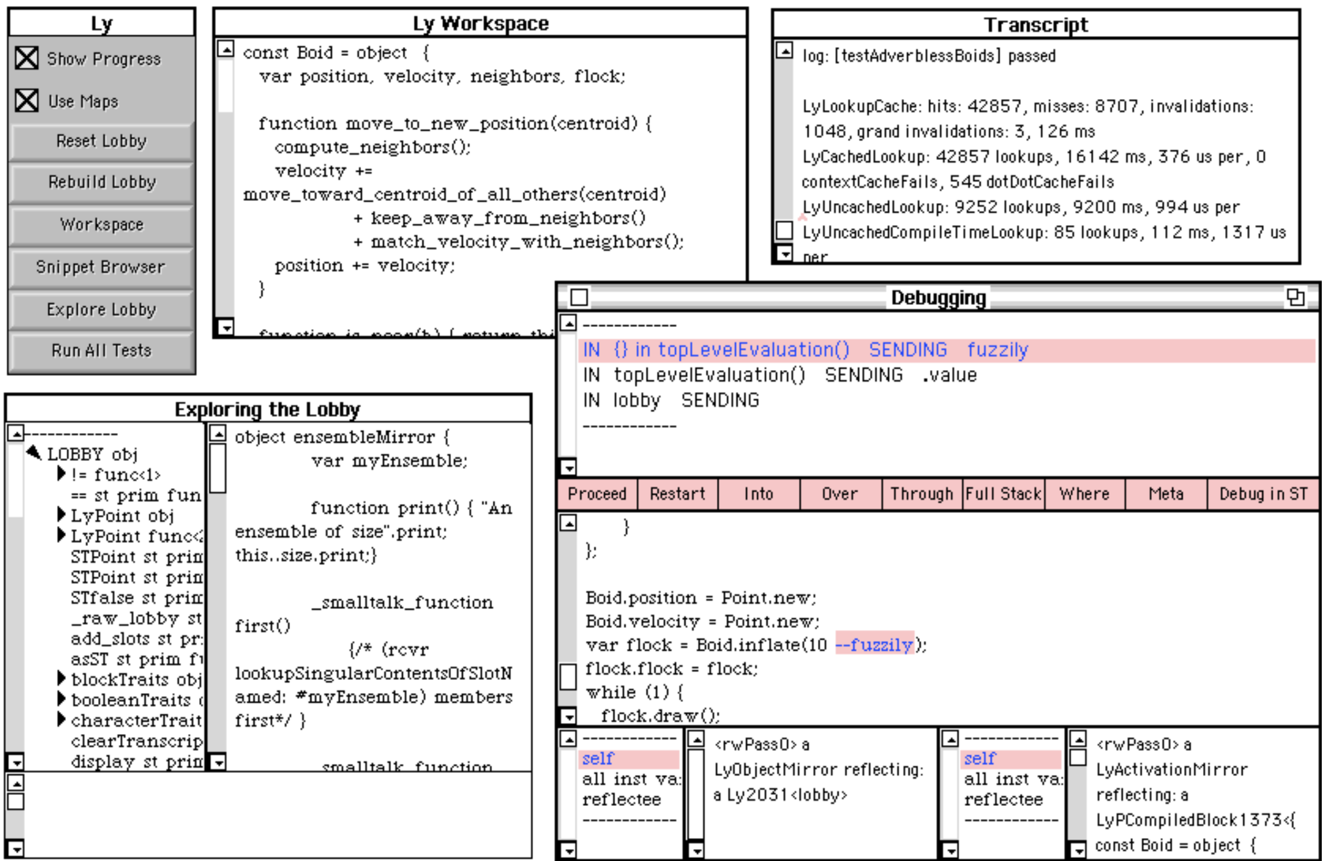

Figure 6. The Ly Environment

6.2 Partial Message Understanding

What if a message is sent to an ensemble and but understood by only some of its members? For example if the program sends "lay eggs" to a flock of birds, should the males silently ignore the request? Or should it result in a global error? Or should special return values come back in the result ensemble? We can see arguments for any of the three possibilities.

6.3 Partial Non-local Returns

Ly includes Smalltalk-like blocks, and as a consequence, a method may be passed a block that, when evaluated, causes the call stack to be cut back to a point above the invocation of said method. This scenario occurs when the block contains a *return* statement, as such statements cause the block to return from its enclosing lexical scope.

Suppose such a block were passed as an argument to an ensemble, so that each member ran a method which received the block as an argument. If some of the members were to invoke the block and it returned, there would be an attempt to cut back the call stack across a fork/join point! Should such non-local returns be an error? If not, what should they do?

6.4 Unintended Ensembles

Unlike the previous three situations, the following one took us by surprise; we were unaware of it until it actually happened to one of the authors. Ly is intended to make massively parallel programming easy, but it became too easy: parallel ensemble computation occurred where none was expected! Our system was executing a method that, like turnLeft in the diagram above, had an ensemble as receiver. Said method contained if (true) doSomething, but instead of invoking doSomething once, it invoked doSomething in parallel for every member of the ensemble! Single-stepping with the source-level debugger exposed the cause of the problem: Ly's compiler treats if(condition) statement as syntactic sugar for condition.if({statement}); that is, it turns the special form into a message sent to the condition, with a block argument. Rather than being built-in, true is merely a slot high up in the inheritance hierarchy. So, when the true message was sent to the current receiver, a match was found in *every* member (because every member inherited the true slot), and an ensemble containing a true for each member was returned. Then, when if was sent to that ensemble, there was a parallel invocation for each true in our ensemble of truth! The result was not what we expected at all: many "somethings" were done.

What can be done about this issue? Recasting true as a literal instead of a slot would merely defer the problem to other cases. Reducing an ensemble containing identical members into a singleton by default would destroy important frequency information.

6.5 Synchronization

On a broader level, Ly's attempt to eschew synchronization and embrace nondeterminism will not be compatible with many classic algorithms. For example, an exchange-based sort would require synchronization to serialize adjacent exchanges. We have started an exploration of alternative algorithms, but do not yet understand their effectiveness or efficiency. We will need to gain significant experience in implementing applications with this programming model before we can assess its efficacy.

7. Related Work

We are not the first to dream of harnessing emergence. Anthony has looked carefully at natural distributed systems and their application to distributed computer systems, and has devised an election algorithm that exploits emergence [11]. Agent-oriented computing also seeks to harness emergence. Varghese and McKee investigate swarms of agents as a means to achieve fault-tolerance [12]. Parunak and Brueckner have taken an information-theoretic approach to understand the conditions under which emergence can be effective [13]. Devescovi et al have devised a computational framework called SelfLets, and incorporated biologically-inspired self-organizing algorithms into it [14]. Finally, Fleissner and Baniassad investigated a programming paradigm based on information diffusion [15], in which there may be a duality relationship between information diffusing across a space with many points, and a system of active individuals in many points in space.

8. Conclusions

We hope to cut the Gordian knot that is manycore programming by embracing nondeterminism and harnessing emergence. To that end, we propose two new concepts: *ensembles,* which capture the notion of a flock or swarm, and *adverbs*, which specify how to perform an ensemble computation and how to treat the results. To test these concepts, we designed and implemented a new language, Ly, adding ensembles and adverbs to an object model loosely based on Self, in a syntax loosely based on JavaScript. Early experience with Ly has uncovered a number of issues which will point the way for our next iteration.

Acknowledgements

We would like to thank Erik Altman, Doug Kimelman, Kristen McIntyre, Leo Ungar for their help with this paper.

References

1. E.D. Berger, et al., “Grace: Safe Multithreaded Programming for C/C++,” OOPSLA, 2009.
2. R.L.B. Jr., et al., “A Type and Effect System for Deterministic Parallel Java,” OOPSLA, 2009.
3. S.P. Jones, et al., “Concurrent Haskell,” POPL, 1996.
4. C. Hewitt, et al., “A universal modular actor formalism for artificial intelligence,” IJCAI, 1973.
5. B. Bloom, et al., “Thorn—Robust, Concurrent, Extensible Scripting on the JVM,” OOPSLA, 2009.
6. D.R. Hofstadter, *Gödel, Escher, Bach: an Eternal Golden Braid*, Basic Books, Inc., 1979.
7. D. Ungar and S.S. Adams, “Hosting an Object Heap on Manycore Hardware: An Exploration,” Dynamic Language Symposium, 2009.
8. D. Ungar and R.B. Smith, “Self: The power of simplicity,” *SIGPLAN Not.*, vol. 22, no. 12, 1987, pp. 227-242; DOI http://doi.acm.org/10.1145/38807.38828.
9. C.W. Reynolds, “Flocks, Herds, and Schools: A Distributed Behavioral Model,” SIGGRAPH, 1987.
10. C. Reynolds, “Boids,” 1995; http://www.red3d.com/cwr/boids/.
11. R.J. Anthony, “Emergence: a Paradigm for Robust and Scalable Distributed Applications,” International Conference on Autonomic Computing, 2004.
12. C.A. Moritz, et al., “Exploring Optimal Cost-Performance Designs for Raw Microprocessors,” Field-Programmable Custom Computing Machines, 1998.
13. H.V.D. Parunak and S. Brueckner, “Entropy and Self-Organization in Multi-Agent Systems,” AGENTS'01, 2001.
14. D. Devescovi, et al., “Self-organization algorithms for autonomic systems in the SelfLet approach,” Autonomic Computing and Communication Systems, 2007.
15. S. Fleissner and E. Baniassad, “Harmony-oriented programming and software evolution,” OOPSLA, 2009.