



MPI RUBY: SCRIPTING IN A PARALLEL ENVIRONMENT

By Emil Ong

SCRIPING LANGUAGES SUCH AS PERL AND PYTHON ARE EXTREMELY POPULAR IN MANY AREAS OF COMPUTING, AND THE SCIENTIFIC COMMUNITY IS TAKING NOTICE. ONE OF THE MOST FRUITFUL USES OF SCRIPTING LANGUAGES IS PROTOTYPING.

Programmers often use scripting languages in prototyping environments in which an algorithm is implemented, tested, and debugged. They can then recode this sample implementation in another faster, and possibly more complex language for actual deployment. This type of development has been available to the scientific community since Perl and other languages first appeared. However, these languages only offer prototyping for serial (or possibly multithreaded) programs. Most of the work in scientific computing today is done in parallel algorithms, often via message-passing architectures such as the message-passing interface, MPI.

While working in the MPICH (MPI CHameleon) group at Argonne National Laboratory, I investigated the possibility of using Perl and Python to write MPI programs. Both had interfaces to MPI available, but neither offered MPI's complete functionality. My colleagues who were involved in crafting the MPI standard considered this limitation quite severe. At the time, I had also become interested in a newly emerging language called Ruby, which maintains a strict adherence to object-oriented principles and a clean, intuitive syntax. The language offered many features that I had wanted over the course

of my programming career, and its consistency impressed me. So, putting the two together, I created MPI Ruby, a complete binding of MPI to Ruby.

In this article, I will introduce Ruby and MPI Ruby. You'll also see demonstrations of some applications and information on the project's current status and its availability. I assume that the reader has a reasonable familiarity with MPI.

Ruby: A clean, object-oriented scripting language

Yukihiro Matsumoto first developed Ruby in 1993. Over the last nine years, it has gained a significant audience in Japan and is becoming popular in the US. Ruby is freely available at www0.ruby-lang.org and runs on many platforms including Unix, DOS, Windows, Macintosh, and BeOS.

Ruby has gained a reputation for being extremely easy to write and, perhaps more importantly, easy to read. It is completely object oriented—everything is an object, even classes, methods, and operators:

```
"Hello, World!".length
returns 13

-2.abs
returns 2
```

You can probably guess that in the first example, we're getting the length of the string "Hello, World!", and in the second, we're getting the absolute value of -2. That's part of Ruby's easy readability. The OO part of Ruby comes when you see that what we're really doing in both lines is calling methods on objects. "Hello, World!" is a string object on which we're calling the `length()` method (parentheses are optional in Ruby in unambiguous cases). More surprising, though, might be that -2 is an object, an integer object, on which we're calling the `abs()` method.

Defining methods in Ruby is similar to other languages. For example, we can define a method for computing factorials as

```
def factorial(x)
  if x == 0 || x == 1
    return 1
  else
    return x * factorial
      (x - 1)
  end
end
factorial(5)
```

returns 120

Status and Availability

MPI Ruby in beta and version 0.3 is available at www-unix.mcs.anl.gov/mpi/mpi_ruby. Ruby is available at www.ruby-lang.org.

Dave's Sideshow

A brief introduction

It's a real pleasure to be joining Paul Dubois on the Scientific Programming column. I have known Paul for about seven years—in fact, his columns from the days of *Computers in Physics* were instrumental in introducing me to new technology and inspiring me to think about scientific programming in new ways. I certainly hope to work with Paul and continue that tradition.

The confessions of a bit twiddler

As they say, the first step to recovery is admitting that you have a problem—in my case that would probably have to be some kind of compulsive bit-twiddling disorder.

As a programmer, I've always been drawn to topics that pertain to the inner workings of software and systems. For instance, reverse-engineering copy protection on my old Apple II, figuring out the operation of CM-5 vector units, or trying to uncover obscure details of dynamic loading. I suppose some computer scientists would actively discourage such deviant activity. Besides, end users should never be bothered with such frivolity (not to mention the fact that one is never quite sure of its legality in our glorious post-DMCA era).

However, it also seems like one of the most important goals of any scientific experiment is to make sure that you fully understand the experiment and the environment in which it is conducted. How do you know that a result is correct? How do you know whether a bad result is due to bad physics or a side effect of the underlying system environment? How do you know that a clever parallel algorithm that appears to work for now won't fall apart when you port your code to a new platform? Such questions are never easy to answer. However, rather than shying away from seemingly unimportant details, I've always found a little knowl-

edge of bit-twiddling to be rather handy. In the future, I'm hoping to feature occasional articles that explore the deep, dark mysteries of scientific software and systems. Stay tuned.

Ruby

Scripting languages continue to have increased prominence in scientific software. Readers of this column are almost certainly aware of languages such as Python, Perl, and Tcl. However, new scripting languages continue to emerge. One of these languages is Ruby, described in more detail in this month's article by Emil Ong—a former University of Chicago student who started working with Ruby despite my best efforts to corrupt him with Python programming.

Last October, I had the pleasure of attending the First International Ruby Conference—an informal affair with about 40 attendees. Although predicting Ruby's future might be impossible, several scientific projects are underway, including MPI bindings and a matrix package.

Darwin (the operating system)

Having programmed extensively on Solaris and Linux, I recently decided to take the plunge and buy a Macintosh running OS-X. Underneath the covers of OS-X is Darwin, a Unix operating system based on FreeBSD and Mach. Of course, as with any new machine, the first thing you must do is figure out how to view DVDs and how to build shared libraries (although not necessarily in that order). The good news is that Darwin's developer tools are free and include all the standard GNU compiler tools. Furthermore, advanced programming techniques such as scripting language extensions, dynamic loading, and shared libraries all seem to work. The bad news is that all of these facilities sufficiently differ from other Unix variants that you



might have to do a little tweaking of makefiles and configure scripts to get them to work (more bit twiddling?).

If you're porting code, you probably want to take a look at Fink, a package installer, as well as some of the documents on the Fink homepage (<http://fink.sourceforge.net>). This information was enough to help me port some dynamically loadable Python extensions to Darwin with only a little effort. If you need to run X applications, XDarwin can run X11 applications on the same screen as Macintosh applications (www.xdarwin.org). Oh, and last but not least, playing DVDs was easy—especially considering the mystical voodoo ritual and IRQ chant I once had to perform to get a soundcard to work with Linux.

Of interest

The Fourth O'Reilly Open Source Conference is coming up this July in San Diego (<http://conferences.oreillynet.com/os2002>). Having attended several of these conferences, I can assure you that it's a great way to meet other open-source developers and explore several different software communities in a single venue. Paul Dubois has also asked me to mention the *Journal of Object Technology* (or *JOT*), a new Web publication debuting 1 May. It should come as no surprise that Paul is a member of its editorial board.

Notice that I called this defining a *method*. That’s because there are no “functions” in Ruby, only methods. (When we define `factorial()`, it is actually a method of the `Object` class.) This detail doesn’t have much effect on how we use top-level methods, but it makes everything consistent in that there are only method calls, not function calls and method calls.

However, because this is a method, doesn’t it make sense to have it be a call on an integer? Ruby will let us do just that—extend any class at any time:

```
class Integer
  def factorial
    if self == 0 || self == 1
      return 1
    else
      return self * (self-1).
        factorial
    end
  end
end

5.factorial

returns 120
```

We can use this technique to define new classes as well as extend old ones, as this example shows.

Ruby has a slew of other features, but to keep this discussion focused on MPI Ruby, we’ll only look at one more. Iterators are popular constructs in the OO world that several languages use, including C++, Java, and Ruby. Ruby differs from many other languages, however, in that it has a special syntax built around iterators:

```
x = 0
5.times{ |i|
  x += i
}
x = 0 + 1 + 2 + 3 + 4 = 10
```

In this code, first we set a local variable `x` to 0, then we call the iterator

`times()` on the integer object 5. The iterator is instantiated by the local variable `i` in the block (`x += i`). The result is that the value of `i` gets added to `x` on each pass of the loop. When the loop is done, `x = 10`.

Iterators are a powerful feature of Ruby that let you do all sorts of things you wouldn’t expect. For example, we can get a list of all the factorials from 1 to 8 using iterators:

```
(1..8).map{ |i| i.factorial }

[1, 2, 6, 24, 120, 720, 5040, 40320]
```

For those of you familiar with Python, notice how similar this technique is to list comprehensions.

This should be enough of Ruby to get us started. Now, let’s take a look at how we can write parallel programs using MPI Ruby.

A first look at MPI programs in Ruby

MPI Ruby consists of two parts: the interpreter (`mpi_ruby`) and the Ruby module (MPI). `mpi_ruby` is a simple MPI program that calls the Ruby interpreter. It initializes and finalizes MPI, but other than that, it simply runs Ruby.

Let’s look at a simple “Hello, World!”-type example:

Listing 1, `hello.rb`

```
1 printf("Hello, I am %d of
  %d\n",
  MPI::Comm::WORLD.rank,
  MPI::Comm::WORLD.size)
```

In Listing 1, we simply have a call to the Ruby version of the familiar `printf()` function in which we print out a message with the current process’s rank within the `WORLD` communicator as well as that communi-

tor’s size. The `WORLD` communicator is a constant, and like all constants in Ruby, begins with a capital letter. To run this program, we execute the MPI Ruby interpreter on the desired nodes, passing in the name of the script file as the argument. Using `MPICH`, the execution might resemble

```
% mpirun -np 5 mpi_ruby
  hello.rb
Hello, I am 1 of 5
Hello, I am 0 of 5
Hello, I am 4 of 5
Hello, I am 3 of 5
Hello, I am 2 of 5
```

Notice that writing and running this script took many fewer lines than C or Fortran would take. We didn’t have to call `MPI_Init()`, declare any variables, or finalize MPI. Already we’ve saved five lines of code.

This example program also lets us see more of Ruby’s OO flavor. I mentioned that part of MPI Ruby is the MPI module. The Ruby module system is similar to Python modules or Java packages. The MPI module contains the class `Comm`, which encapsulates MPI communicators. The `Comm` class contains the constant communicator `WORLD`, which corresponds to the `MPI_COMM_WORLD` communicator in the C version of MPI. Thus, we can refer to this communicator as `MPI::Comm::WORLD` in Ruby. Because the communicator is an object, we can call methods on it, as in this case where we call the `rank()` and `size()` methods to get the communicator’s rank and size.

Now let’s look at an example involving communication:

Listing 2, `basic.rb`

```
1 myrank = MPI::Comm::
  WORLD.rank
2 csize = MPI::Comm::
  WORLD.size
```

```

3 if myrank % 2 == 0 then
4 if myrank + 1 != csize then
5   hello = "Hello, I'm
      #{myrank}, you must
      be #{myrank+1}"
6 MPI::Comm::WORLD.send
  (hello, myrank + 1, 0)
7 end
8 else
9   msg, status = MPI::Comm::
  WORLD.recv(myrank - 1, 0)
10 puts "I'm #{myrank} and
     this message came from
11   #{status.source} with
     tag #{status.tag}:
     '#{msg}'"
12 end

```

In Listing 2, we save the `WORLD` communicator's rank and size in the first two lines. Next, all even-numbered processes send a message to the next-higher-ranked process (for example, process $2i$ sends a message to process $2i + 1$) while all odd-numbered processes receive the message from their predecessor processes and print it out. The arguments to the `send()` method (line 6) are the message object, the target process, and a tag, respectively. The arguments of the `recv()` method (line 9) are a source process and a tag, respectively.

This example has some Ruby-specific syntax, which deserves explanation. First, we see in line 5 that we have `#{myrank}` and `#{myrank+1}` in the string `hello`. When an expression is inside “`#{ }`” in a string, the expression is expanded as part of the string in place. Second, we see in line 9 that on the left-hand side of the assignment, we have two variables, `msg` and `status`, separated by a comma. The `recv()` method on `Comm` objects returns an array of size 2, with the first element being the message received and the second element being a `Status` object. This syntax lets us decompose the array and assign `msg` to the first element and

`status` to the second.

When we run the code, we might get the following output:

```

% mpirun -np 6 mpi_ruby
  basic.rb
I'm 1 and this message
  came from 0 with tag 0:
  'Hello, I'm 0, you must
  be 1'
I'm 3 and this message came
  from 2 with tag 0: 'Hello,
  I'm 2, you must be 3'
I'm 5 and this message came
  from 4 with tag 0: 'Hello,
  I'm 4, you must be 5'

```

When we called the `send()` and `recv()` methods, we didn't have to specify buffers, byte counts, or types. Again, this is because in Ruby, everything is an object. There are no native notions of buffers or bytes. All classes descend from the `Object` class. When I designed MPI Ruby, it was important to me that this aspect of programming in Ruby not be lost in the translation, despite the fact that the C and Fortran versions of MPI are very much centered around buffers, byte counts, and data types. Thus, we were simply able to pass in an object to `send()`. We received that object as the return value of `recv()`.

At the moment, the only objects that I have experimented with are data objects. Sending nondata objects such as file descriptors and sockets is possible, but the semantics are not yet well explored. Passing threads as objects for thread migration or checkpointing also offers an interesting area of research.

In general, most data objects that you send do not require manually writing serialization methods. You could overwrite the default serialization methods if you wish, though.

Now that we've seen what MPI Ruby code looks like, let's explore a real-world application that shows how easy it is to prototype communication.

Barnes-Hut particle simulation

The Barnes-Hut algorithm is a method for computing the interactions of particles under an arbitrary force. It is a solution to the classic N -body problem. The BH algorithm is particularly well suited to forces that weaken dramatically with distance, such as Coulombic or gravitational forces. By using trees whose leaves are the particles and whose interior nodes are approximations of forces exerted by the particles under them, BH avoids computations with little effect on the final computation. For example, if two clusters of particles are far from each other, we can approximate one cluster's forces on the other by simply assuming that there is one force vector at the center of the first cluster's mass. Placing a tree for a cluster on each node thus parallelizes the BH algorithm. Cluster size can vary depending on the number of nodes. After each node calculates its cluster's approximation tree, the tree can then be sent to other nodes that can compute that cluster's interactions with their own. The serial version of the algorithm computes the interactions by traversing the whole tree on one node.

I implemented this algorithm in MPI Ruby with David Garmire's help (he had previously worked on a serial version in C under Guy Blelloch at Carnegie Mellon University). I based a serial version of the Ruby code on this C code and ran it in the usual Ruby interpreter. The translation from C to Ruby was fairly straightforward. Of course, the Ruby code's performance was not close to the C code's performance, but it worked. Now, all that remained was to parallelize the code. Veteran parallel programmers will at this point say, “Ha!”—parallelizing code is usually nontrivial to say the least. However, in MPI Ruby, the task was simple:

Further Reading

J. Barnes and P. Hut, "A Hierarchical $O(N \log N)$ Force Calculation Algorithm," *Nature*, vol. 324, no. 4, Dec. 1986, pp. 446–449.

W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, Cambridge, Mass., 1999.

D. Thomas and A. Hunt, *Programming Ruby: The Pragmatic Programmer's Guide*, Addison Wesley Longman, New York, 2000.

PythonMPI, <http://sourceforge.net/projects/pympi>.

Scientific Python, <http://starship.python.net/crew/hinsen/scientific.html>.

Listing 3, `barnes-hut.rb` excerpt

```
1 computeBH(myTree, myTree)
2 sendTree = myTree
3 ($nprocs - 1).times { |i|
4   recvTree, status = MPI::Comm
      ::WORLD.sendrecv(sendTree,
      ($rank + 1) % $nprocs, 0,
      ($rank - 1) % $nprocs, 0)
5   computeBH(myTree, recvTree)
6   sendTree = recvTree
7 }
```

Listing 3 shows how we could parallelize the code in less than 10 lines. In the first line, we compute the forces of the particles in the local cluster on themselves. Lines 3 through 7 contain the loop in which we compute all the forces of the other processes' particles on the local cluster. In line 4, we use the classic send/receive communication loop; each node receives a tree from its left neighbor and sends a tree to its right neighbor. Following the communication, we compute the forces on the latest cluster to be received by this process on our cluster. The loop repeats until all particle interactions are computed.

So, why is this so easy in MPI Ruby and often so difficult in C? The basic communication algorithm is likely to be the same in both versions (it better be—this is supposed to be a prototype!), so there's no added convenience there. Instead, the ease of use comes from the fact that in Ruby, everything is an object and in MPI Ruby, messages are objects. If we

rewrote this code in C or Fortran, it would take us a long time to get to line 4, where we just say, "Send this tree and receive another tree." We would have to do recursive packing and make special data types. MPI Ruby handles this tedium for us, because most Ruby objects are easily marshaled (converted to a byte stream). I didn't have to write any explicit marshaling code for my tree class.

The end result is parallel code, which shows us the communication algorithm in a clear and understandable way without bogging us down in bytes and types. Moreover, we took serial code and parallelized it in five minutes. With minimal effort, we could test and verify that the algorithm worked on a small data set. Translation to a more optimized form can now be handled easily with the assurance that it will probably work the first time.


Exploiting Ruby's dynamic nature

Prototyping is likely to become the most frequent use of MPI Ruby, but many interesting applications are possible because Ruby is a scripting language. A colleague at Argonne suggested one of the cleverest examples to me. A major administrative issue with parallel programming and development is process startup. Setting up a job to run can have high overhead in terms of computing latency and human time. Because Ruby is interpreted, it can execute Ruby code from a string that is constructed dynamically or read in from a file. In fact, the interactive Ruby interpreter, `irb`, is itself written in Ruby.

We can apply this ability to the process startup problem by starting up an MPI Ruby process that simply reads in a program from a socket or a file and broadcasts it to other nodes that then run the program as a string. A quick prototype of such a program—called

RED, the Ruby Execution Daemon—is included with the MPI Ruby distribution. Execution times are no different when programs run under RED, and startup latency reduces to the time it takes MPI to transfer the program text.

Many interesting projects could come from exploiting Ruby's dynamic nature, such as an MPI Ruby parallel command line interpreter or perhaps even genetic programming. The possibilities are exciting and, as yet, unexplored.

Compiled languages have ruled the computing world, but interpreted and scripting languages have slowly made an impact. They are widely used in many areas of computing because of their fast development times and dynamicity. The scientific computing community now has an opportunity to exploit the usefulness of these languages in a parallel environment through MPI Ruby. Prototyped and even full applications can be written quickly and easily. With luck, MPI Ruby is only the beginning. Dynamic and interpreted languages offer an exciting new world to explore, and scientific computing can reap the benefits. 

Acknowledgments

Many thanks and much credit goes to the MPICH and support groups at Argonne National Laboratory. I also thank David Beazley, David Garmire, Guy Blelloch, and Katherine Yelick for their help and support.

Emil Ong is a graduate student at the University of California, Berkeley. His technical interests include systems programming, programming languages, and security. He received his BS in mathematics and computer science from the University of Chicago. Contact him at UC Berkeley, 464 Soda Hall, Berkeley, CA 94720-1776; emilong@eecs.berkeley.edu.