Building Flexible Large-Scale Scientific Computing Applications with Scripting Languages

David M. Beazley[*] Peter S. Lomdahl[†]

Abstract

We describe our use of scripting languages with a large-scale molecular dynamics code. We will show how one can build an interactive, highly modular, and easily extensible system without sacrificing performance, building a huge monolithic package, or complicating code development. We will also describe our use of the Python language and the SWIG automated interface generation tool that we have developed for easily creating scripting language interfaces to C/C++ programs.

1 Introduction

For the past five years, we have been developing a large-scale short-range molecular dynamics code (SPaSM), for use on massively parallel supercomputers, multiprocessor servers, and high performance workstations [7]. We are using this code to study dynamic cracks, dislocations, friction, ductile-brittle transition, and other related material properties. Our computational efforts have ranged from achieving maximum performance on RISC architectures, simulating large systems with as many as 10^8 particles, maintaining portability across a variety of parallel architectures, and developing more efficient algorithms [7, 8, 10]. Despite these efforts, effectively working with large-scale MD simulations has been a frustrating endeavor. Simulations may generate tens to hundreds of gigabytes of data that can not be effectively analyzed on normal workstations. Modifying and extending a parallel application is also difficult–especially as code size increases. To the solve the data glut and code management problems we feel that it is useful to develop flexible parallel applications that allow for interaction, simulation, data analysis, and visualization. However, how does one accomplish this while maintaining performance, portability, and coding simplicity?

In this paper, we describe how we have used extensible scripting languages such as Python, Tcl/Tk, and Perl to build a simple, yet extremely flexible molecular dynamics application. Our approach requires no modifications to our C/C++ code, makes it possible to glue different software components together, and has allowed us to interactively simulate, analyze, and visualize, huge amount of data from inexpensive workstations and standard network connections. As a result, we are now able to perform meaningful experiments with tens to hundreds of millions of particles.

2 Some Thoughts About Scientific Programming Languages

As computational scientists, we are "trained" to program in Fortran or C/C++ because these languages offer the performance needed to solve large numerical problems. However, few people would probably make the claim that programming in these languages is fun or

[*]Department of Computer Science, University of Utah, Salt Lake City, UT.

[†]Theoretical Division, Los Alamos National Laboratory, Los Alamos, NM.

easy. Using a compiled language is almost always difficult because code must be developed in a compile/debug cycle. Furthermore, applications are typically non-interactive (without a lot of work) and it becomes difficult to add or change features as software grows in size. While object oriented programming in C++ is sometimes sold as a “solution to all problems”, it too suffers from the lack of interactivity and difficulty of programming.

On the other hand, interpreted languages like Python, Tcl, and Lisp can be easy to use because they are interactive, you can debug a program without having to recompile it, high level data structures like lists, arrays, and classes can be created on the fly, and extremely powerful programs can often be written with a small amount of code. The downside is that these languages are slow–often as much as 50 times slower than a comparable Fortran or C program. As a result, they are usually considered to be too inefficient for “serious” scientific work.

However, a two-language approach can be applied successfully to large-scale scientific problems. Computationally intensive operations can be coded in C or Fortran, while an interpreted language can be used to provide the overall structure and organization. Such an approach is certainly not new–in fact, it has been used quite successfully in commercial packages such as MATLAB and IDL for years. The approach has also been used in large scientific “production codes” found at National Laboratories. Unfortunately, the use of interface languages is usually only found in large software packages. This is not that surprising considering the difficulty of writing a scripting language–a task that most scientists would prefer to avoid. However, in recent years, a number of freely available, high quality scripting languages such as Python, Tcl, and Perl have emerged [1, 3, 4]. These languages are easily integrated with C/C++ and have tens of thousands of users. As a result, they are well-documented, well tested, and well-supported. As it turns out, using these languages is easier than many scientists might imagine (although we will later show how we have made it even easier).

3 Parallel Python

In our molecular dynamics application, we have chosen Python as an extension and prototyping language [1, 2]. Python is a small, yet extremely flexible object oriented language that is being used increasingly in physics applications due to its clean syntax and variety of extensions including fast array manipulation and data visualization [5, 6].

While Python supports Unix, Win32, and MacOS based machines, we have modified it slightly to properly handle parallel I/O when used in a message-passing environment [11]. With these modifications it is possible to run Python on the CM-5, T3D, SP-2, and workstation clusters running MPI. When used, our original C program and a Python interpreter are running on each node of the system. Python runs in a pure SPMD style of programming and operates in the same manner as any C or Fortran message passing program. In other words, each interpreter tends to run the same set of commands, but may also participate in message passing operations, synchronize with other nodes, or perform independent operations. When running on a workstation, the ordinary Python interpreter can be used.

4 A C/Python Example

To illustrate how a scripting language can be used, consider the main loop of a typical MD simulation. Written in C, it looks something like this :

```
    void timesteps(int firststep, int laststep, double Dt, int examine_freq) {
     /* Set up */
     ...
     /* Run it */
     for (i = firststep; i < laststep; i++) {
             integrate_adv_coord(Dt);     /* Advance particle coordinates  */
             boundary_periodic();         /* Apply boundary conditions     */
             redistribute();              /* Move particles between nodes  */
             force_lj();                  /* Calculate Lennard Jones force */
             integrate_adv_velocity(Dt);  /* Update velocities             */

             /* Periodically examine the data */
             if (i % examine_freq) {
            ... Examine the data, dump datafiles, checkpoint, etc...
             }
     }
}
```

While simple, consider the difficulty of extending and generalizing this code. If we wanted to support multiple potential energy functions or different boundary conditions, we would need to add switches and options. If we wanted to add new data-analysis capabilities, we would need to make further modifications. As more options and features are added, we will get a a proliferation of special cases and run-time options. To manage this, we might further consider adding a simple graphical user interface or rewriting portions of the code in C++. At this point, our “simple” molecular dynamics program is starting to look like a bloated monolithic package that attempts to do everything, yet has become too complicated to use, modify, and maintain.

Rather than take this approach, a scripting language can be used to glue simple components together and to provide structure and organization. Basic functionality is still written in C, but applications are now controlled by scripts and procedures written in the interpreted language. For example, we could recode the main loop of our MD simulation in Python as follows :

```
    # Python implementation
def timesteps(firststep,laststep,force,boundary,Dt,examine_freq,examine_func):
     ... setup code ..
     for i in range(firststep,laststep):
          integrate_adv_coord(Dt)              # Update positions
          boundary()                           # Apply boundary conditions
          redistribute()                       # Move particles between nodes
          force()                              # Calculate force
          integrate_adv_velocity(Dt)           # Update velocities
          if (i % examine_freq) == 0:          # Perform various operations
                   for func in examine_func:
                         func(i)               # Call function for data-analysis
```

The Python version looks almost identical to its C counterpart. Functions such as `integrate_adv_coord()` and `redistribute()` are mapped directly onto the corresponding C functions. Unlike C, the force and boundary condition functions have been passed in as arguments so they can be easily changed. For data analysis purposes, we simply provide a list of functions in the variable `examine_func` and each one will be used during the simulation. A typical simulation script might look like the following :

```
    # Dump out a datafile containing particle data
def dump(n):
        filename = "Dat"+str(runno) + "." + str(n)
        output(filename)

# Calculate Lennard-Jones energy and save in a list
Times            = []
Total_Energy     = []
Kinetic_Energy   = []
Potential_Energy = []

def energylj(n):
        global Dt, Times, Total_Energy, Kinetic Energy, Potential Energy
        potential_lj()                       # Calculate energies
        totalk = total_kinetic()
        totalp = total_potential()
        totale = totalk + totalp
        Times.append(Dt*n)                   # Save energy values
        Total_Energy.append(totale)
        Kinetic_Energy.append(totalk)
        Potential_Energy.append(totalp)

# Debugging function to calculate maximum particles in a cell
# Loops over an internal C data-structure and extracts data from it.
def findmax(n):
        c = Cell(0,0,0)               # Grab first cell from C program
        max = 0
        for i in range(0,cvar.Xcells*cvar.Ycells*cvar.Zcells):
                if c[i].n > max:
                      max = c[i].n
        max = reduce_int(max)
        print "Step ",n, " : max = ", max

# Now run a simulation for 100 steps
timesteps(0,100,force_lj,boundary_periodic,Dt,10,[dump,energylj,findmax])
```

When run, our simulation will proceed normally and call all three of our user-defined functions each time data is examined. If we wanted to write more analysis functions or change the implementation, this is easily accomplished by simply changing the script. It is also possible to create or redefine functions on the fly without quitting a running simulation. Unlike C, no re-compilation is necessary. While this has only been a simple example, more complex scripts can coordinate simulation, data analysis, and visualization. In our production runs, it is not uncommon for a simulation script to produce data files, checkpoint files, log files of interesting quantities, and visualizations (saved as GIF images). Common functions can also be placed in a library of Python scripts and reused in any number of problems.

5 Automatic Interface Generation with SWIG

Virtually all of the computationally intensive functions in our Python interface are implemented in C for performance. To construct this interface, we have developed a tool called SWIG (Simplified Wrapper and Interface Generator) that automatically generates bindings from C/C++ to Python, Tcl, Perl, or Guile [9]. This is done by specifying ANSI C/C++ declarations in a special file like this :

```
    %module spasm
%{
#include "spasm.h"
%}
...
extern double total_kinetic();
extern double total_potential();
extern void force_lj();
extern void potential_lj();
extern void integrate_first_velocity(double Dt);
extern void integrate_adv_coord(double Dt);
extern void integrate_adv_velocity(double Dt);
extern void boundary_periodic(void);
extern void boundary_free(void);
...
```

SWIG requires no modifications to existing C code, supports almost all C datatypes, and provides direct access to C structures and C++ classes. The functions listed in this file automatically turn into Python commands when the C code is compiled. This process is hidden in the Makefile and is completely transparent. If we write a new C function, its declaration is placed in this file and it "magically" becomes a new Python command. As a result, adding new commands is a trivial exercise. This allows us to focus on physics, not interface building.

6 Object Oriented Programming and More

Using a high-level interpreted language provides our application with structure and organization. It also provides object-oriented programming without requiring everything to be rewritten in C++. We have created a data analysis and visualization system that can be used side-by-side with running simulations. The object-oriented interface is written entirely in Python, while computationally intensive operations such as graphics primitives are written in C. To use the system, a user can simply issue Python commands as follows :

```
    # Example of visualization system
from vis import *               # Load visualization module
ke = Spheres(KE,0,20)           # Create a plot of spheres
vp = Profilez(VZ,50,-2,10)      # Create a velocity profile
shear = PlotCells(SHEAR,-10,10) # Show shear stress
...
show(ke,vp,shear)               # Dump out all images
ke.rotr(90)                     # Rotate KE image right 90 degrees
shear.zoom(300)                 # Zoom in on shear stress plot
```

Each image type is defined by a Python class that inherits common image manipulation operations from a base class. The system can be used interactively or during batch processing. An unlimited number of images can be created and manipulated as needed. Furthermore, if a user wants to make an entirely new kind of plot, it is easily accomplished. For example :

```
    # Make a plot of kinetic, potential, and total energy
class EnergyPlot(Image2D):            # Inherits from Image2D class
      def __init__(self,times,ke,pe,te):
             self.times = times
             self.ke = ke
             self.pe = pe
```

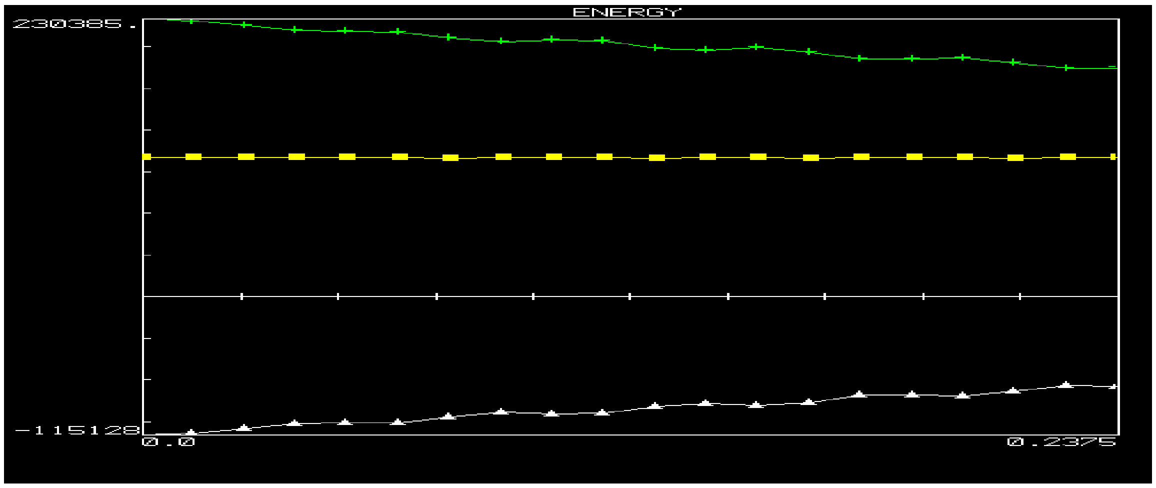

FIG. 1. *Sample energy plot*

```
                self.te = te
                Image2D.__init__(self,min(times),max(times),min(pe),max(ke)
                self.colors = [YELLOW,WHITE,GREEN]
                self.symbols = [SQUARE,TRIANGLE,CROSS]
        def draw(self):
                self.newplot()
                self.plotxy(self.times,self.te,self.colors[0],self.symbols[0])
                self.plotxy(self.times,self.pe,self.colors[1],self.symbols[1])
                self.plotxy(self.times,self.ke,self.colors[2],self.symbols[2])
...
# Run a simulation
timesteps(0,100,force_lj,boundary_periodic,Dt,10,[dump,energylj,findmax])

# Now make an energy plot
e = EnergyPlot(Times, Kinetic_Energy, Potential_Energy, Total_Energy)
e.title = "Energy"
e.show()
```

New classes and objects can be created or changed on-the-fly. In this case we have created a new type of image by inheriting from the class `Image2D`. Most of the functionality such as graph annotation, scaling, and tick marks is contained in this base class. Of course the object oriented interface is not limited to visualization. Entire simulations, initial conditions, and analytical techniques can also be encapsulated in Python classes. SWIG also allows Python to inherit from C++ classes.

Another powerful feature of Python and most other scripting languages is support for shared libraries and dynamic loading. Recently, we were trying to examine data from a new simulation, but we needed a different data analysis method written in C. Using dynamic loading, we were able to write the method in C, turn it into a Python module with SWIG, import it into a running simulation, and use it with our visualization system–all without recompiling the original MD code or quitting the running problem.

These capabilities are only scratching the surface of the power of this approach. It is also possible to write interpreted message passing scripts when SWIG is used to wrap parts of the PVM, CMMD, or MPI libraries. It is also interesting to note that SWIG supports a number of other languages that can be used with our MD code. Building a graphical user interface in Tcl/Tk or providing a Perl interface are easily possible although

Table 1
Simulation time of C vs. C with Python (seconds)

Atoms/processor	C	C with Python
13950	98.7	98.9
45000	314.1	314.8
180000	1317.1	1319.0

not all languages work properly on parallel machines. However, we feel that one should use whichever scripting language is appropriate for the task at hand. Scientific applications demand flexibility so why not allow any language to be used?

7 Design implications

The approach we have taken makes it easy to build interactive C/C++ programs with little or no modification to the underlying code. However, controlling a program in this manner encourages a different style of programming than many scientists might be accustomed to. Individual functions may now be called in any order at any time. In order to maintain correctness, it may be beneficial to add safety checks to many of the underlying functions. This also encourages the development of code that is highly modular and reusable. Rather than trying to create a huge monolithic package, our problems tend to be composed of collections of related functions and modules, but not everything at once. Finally, since we have automated the process of building language interfaces, our C code is clean and can be compiled on its own. Should we discover a better interface mechanism later, it will be relatively easy to use all of our existing C code.

With these considerations in mind, it is important to note that our MD code has actually decreased in size by more than 25%. Functions have been made more reliable and many of the problematic areas of the code have been eliminated entirely. Most of the common functionality has moved to C libraries and is now more maintainable.

8 Performance

"If you optimize everything, you will always be unhappy." – Don Knuth.

It is true that interpreted languages run much slower than equivalent C or C++ code. However, one would clearly not write a force calculation or matrix multiply in an interpreted language (although this is possible). In our MD code, all of the computationally intensive work is still implemented in C. The choice between writing the outer loop of an MD simulation in C or a scripting language is of no consequence as shown in Table 1. In the table, timings for a real MD simulation are shown for an all C implementation versus one being controlled by the Python function shown in an earlier example. A performance degradation of only 0.2% is not a concern.

9 Conclusions

The use of interpreted languages has revolutionized the way in which we work on large-scale molecular dynamics problems by making it easy to create new simulations, analyze data, visualize results, and to try different approaches to problems.

There seems to be a tendency in computer science to build highly sophisticated systems that attempt to hide all of the underlying details from users. As computational physicists however, this is exactly the opposite of what we want! It is critical that we understand the important aspects of our experiments including the numerical methods and physical models. Given the practical difficulties of working with huge simulations on parallel machines, we want our applications to be flexible, adaptable to new architectures, and maintainable. However, this does not imply that we want a huge computational problem solving environment with thousands of functions, widgets, knobs, sliders, and gizmos. The approach we have taken has allowed us to write extremely powerful applications, yet using normal C code and simple tools. At the same time, it encourages better programming while not forcing everyone into a rigid set of rules. We feel that this type of approach is particularly well suited for computational physics research.

10 Acknowledgments

We would like to thank the Advanced Computing Laboratory for their continued support, Paul Dubois and Brian Yang at Lawrence Livermore Laboratory, The Scientific Computing and Imaging group at the University of Utah, and the Cornell Theory Center. We would also like to acknowledge our collaborators, Shujia Zhou, Brad Holian, and Niels Jensen of Los Alamos National Laboratory, Tim Germann at UC Berkeley, and Bill Kerr at Wake Forest University. Development of the SPaSM code has been under the auspices of the United States Department of Energy.

11 Availability

The tools described in this paper are in the public domain. Python can be obtained at `www.python.org` and SWIG is available at `www.cs.utah.edu/∼beazley/SWIG`.

References

[1] Mark Lutz, *Programming Python*, O'Reilly and Associates, (1996).

[2] A. Watters, G. van Rossum, J. Ahlstrom, *Internet Programming with Python*, M&T Books, (1996).

[3] J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley (1994).

[4] R. Schwartz, L. Wall, *Programming Perl*, O'Reilly and Associates (1994).

[5] P. Dubois, K. Hinsen, and J. Hugunin, *Numerical Python*, Computers in Physics, Vol. 10, No. 3, (1996), pg. 262-267.

[6] T. Yang, P. Dubois, Z. Motteler, *Building a Programmable Interface for Physics Codes Using Numeric Python*, Proceedings of the 4th International Python Conference, Lawrence Livermore National Laboratory, June 3-6, (1996).

[7] D. M. Beazley and P. S. Lomdahl, *Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5*, Parallel Computing. 20 (1994) p. 173-195.

[8] D. M. Beazley and P. S. Lomdahl, *A Practical Approach to Portability and Performance Problems on Massively Parallel Supercomputers*, Proceedings of the Workshop on Debugging and Tuning for Parallel Computer Systems, Chatham, MA, 1994. IEEE Computer Society (1996). pg. 337- 351.

[9] D. M. Beazley, *SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++*, Proceedings of The Fourth Annual Tcl/Tk Workshop '96, Monterey, California, July 10-13, 1996. USENIX Association, p. 129-139.

[10] D. M. Beazley and P. S. Lomdahl, *Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations*, Proceedings of Supercomputing '96, IEEE Computer Society (1996).

[11] D. M. Beazley and P. S. Lomdahl, *Extensible Message Passing Application Development and Debugging with Python*, Proceedings of IPPS'97, IEEE Computer Society (1997). (To appear).