Extensible Message Passing Application Development and Debugging with Python

David M. Beazley
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112
beazley@cs.utah.edu

Peter S. Lomdahl
Theoretical Division
Los Alamos National Laboratory
Los Alamos, New Mexico 87545
pxl@lanl.gov

Abstract

We describe how we have parallelized Python, an interpreted object oriented scripting language, and used it to build an extensible message-passing molecular dynamics application for the CM-5, Cray T3D, and Sun multiprocessor servers running MPI. This allows us to interact with large-scale message-passing applications, rapidly prototype new features, and perform application specific debugging. It is even possible to write message passing programs in Python itself. We describe some of the tools we have developed to extend Python and results of this approach.

1 Introduction

One of the greatest problems encountered when working with massively parallel machines is the complexity of software development, the difficulty of building flexible applications, parallel debugging, and dealing with the massive amounts of data that can be generated by large-scale parallel applications. Given the complexity of working with parallel machines, there is tendency to develop parallel "problem solving environments" that attempt to hide all of the underlying complexity by relying on sophisticated object oriented programming frameworks, software libraries, or language extensions. Unfortunately, we feel that this tends to result in large monolithic software systems that are too complicated to adapt to new uses, difficult to integrate with existing code, and almost impossible to debug (since the user is effectively isolated from all of the underlying implementation details). For scientific computing research applications, this is simply unacceptable. Research codes need to be simple to modify and use. It must be possible to understand exactly what is going on inside the code in order to verify correct operation (and to fully understand the experiment!).

In this paper, we describe how we have parallelized Python to serve as a "glue language" for building highly modular and component based parallel applications. The resulting system serves as the basis for developing extensible and flexible parallel codes without relying on a large software infrastructure or parallel computing framework. It also provides us with a nice debugging, prototyping, and user environment. While interpreted languages have often been considered to be too slow for "serious" work, we will illustrate how we have used the Python language with a large-scale molecular dynamics application (SPaSM) without suffering a performance penalty or significantly increasing software complexity.

2 The Python Language

Python is an interpreted object oriented scripting language developed by Guido van Rossum, at CWI, Amsterdam [1, 2]. It has been increasing in popularity and is often compared to languages such as Tcl/Tk and Perl [3, 4]. For controlling parallel applications, we wanted to provide a command driven model similar to that used in scientific packages such as Mathematica, MATLAB, or IDL. We chose Python to serve in this role for a variety of reasons :

- It is highly portable and runs under UNIX, MacOS, and Windows.
- The language is built around a small extensible core. This makes it easier to port to parallel machines.
- It has a clean syntax that is easy to read and easy to learn.
- Python can be run interactively.
- It is easy to build C/C++ extensions to Python.
- A large number of extension modules are already available.

- It is fully object oriented, making it possible to write powerful extensions.
- The language has seen increased use in the scientific community and has a number of numerical extensions [5, 6].
- Python is free and well supported.
- We like it.

More information about Python can be found on the internet, or the book "Programming Python" by Mark Lutz [2]. While a full treatment of the language is not possible here, the syntax is easily understood. The remainder of this paper will focus primarily on our use of Python rather than the language itself.

3 Parallelizing Python

Within a message passing environment, parallelizing the Python interpreter involves being able to safely running a copy of Python on every processor. Like C or Fortran, processors may only be loosely synchronized and will execute code independently unless message passing calls are involved. However, unlike C or Fortran, Python itself is written in C and uses the the C `stdio` library for many operations, including reading scripts from files, importing modules, getting input from the user, and writing byte-compiled versions of modules back to disk. Given the state of parallel I/O support on most machines, this presents a serious portability and usability problem. We need to make sure that Python can run properly on all processors without crashing during I/O operations. At the same time, we don't want to have to modify significant portions of the Python source.

In addressing the I/O problems, we assume that all I/O takes place on a common file system and that files may be shared between multiple processors simultaneously. This is the model most commonly found on large parallel machines and multi-processor servers. It may not be the model on distributed workstation clusters or heterogeneous systems, but the techniques we describe could still be applied (with modification) to those systems.

3.1 Remapping I/O Functions in Python

To remap the I/O operations used in Python, we have written a special C header file `pstdio.h`. This file is included into the Python header files prior to the inclusion of the C `stdio.h` header file. This remaps all of the stdio operations to a collection of "wrapper" functions that we will implement in a manner similar to that described in [7].

```
/* pstdio.h : Wrappers around stdio.h
   for parallel I/O */

#define  fopen      PIO_fopen
#define  fflush     PIO_fflush
#define  fclose     PIO_fclose
#define  rename     PIO_rename
#define  setvbuf    PIO_setvbuf
#define  fread      PIO_fread
#define  fwrite     PIO_fwrite
#define  fprintf    PIO_fprintf
#define  fgets      PIO_fgets
#define  fputc      PIO_fputc
#define  fputs      PIO_fputs
#define  printf     PIO_printf
#define  fseek      PIO_fseek
#define  ftell      PIO_ftell
#define  read       PIO_read
#define  write      PIO_write
#define  open       PIO_open
#define  close      PIO_close
```

3.2 Implementation of Wrapper Functions

The I/O wrapper functions are implemented using a combination of the C `stdio` library and message passing operations. File descriptors are managed in two different I/O modes :

- **BROADCAST**. In this mode, processor 0 reads data and broadcasts it to all of the other nodes. When writing, output is assumed to come from only one processor (usually processor 0, but this can be remapped). This mode is primarily used for handling interactive I/O using `stdin` and `stdout`.
- **BROADCAST_WRITE**. This mode allows all processors to read data independently, but only one processor can write data. This mode is used for most file operations in Python. For example, when reading a script, every node can simply open the file and process its contents independently. By restricting write access, we eliminate problems that occur when multiple copies of Python attempt to write to the same file (which would normally result in garbage). This mode is somewhat faster than the normal broadcast mode since it is not necessary for processor 0 to broadcast input data to the other nodes.

Currently, we have implemented the wrappers under CMMD on the CM-5, the shared memory library on the T3D, and MPI [8, 9, 10]. Eventually, we would hope to implement the library using parallel I/O libraries such as MPI-IO [11].

3.3 Other Changes to Python

Finally, three other changes were required to the Python core.

- A `putc()` call was changed to `fputc()` since it could not be remapped otherwise (since `putc()` is implemented as a C macro).

- A switch was installed to disable dynamic loading of modules. While supported on most workstations, this capability is not available on larger machines such as the CM-5 or Cray T3D.

- An initialization call was added to Python's main() program. This is sometimes needed to initialize MPI and other packages.

3.4 Compilation

The I/O remappings and minor fixes required less than 10 lines of modifications to the entire Python source (consisting of more than 50000 lines of C code). The I/O wrappers have been implemented in about 1000 lines of supporting ANSI C. Together with the Python source, everything is combined into new version of the Python interpreter and a C library for embedding a parallelized version of Python in other applications.

4 Using SWIG to Build Python Extensions

While Python is designed to be easily integrated with C/C++ code, doing so requires one to write special "wrapper" functions that provide the glue between the underlying C function and the Python interpreter. Since the process of writing these wrapper functions is tedious, we have developed a tool, SWIG, that automatically generates the Python bindings from a file containing ANSI C/C++ declarations[12]. Using SWIG, the user extends Python by writing an interface file such as the following :

```
%module spasm
%{
#include "spasm.h"
%}
...
void ic_shock(int nx, int ny, int nz, double vel,
              double width, double gap,
              double temp,  double r0,
              double cutoff);
int  timesteps(int nsteps, int energy_n,
               int output_n, int checkp_n);
void set_boundary_periodic();
void set_boundary_free();
void energy();

// Different potential energy methods

void init_lj(double epsilon, double sigma,
             double cutoff);
void init_table_pair();
...
```

All of the functions in this file turn into Python commands that can be used in scripts or typed interactively. SWIG supports almost all C/C++ datatypes, C structures, and a subset of C++. As a result, it is relatively easy to add new capabilities to the Python interpreter.

5 An Extensible Molecular Dynamics Code

Since 1992, we have been developing a short-range molecular dynamics code, SPaSM, for use on the Connection Machine 5 and Cray T3D systems at Los Alamos National Laboratory [13]. This code is capable of performing production simulations with more than 100 million atoms, yet managing such simulations in practice has proven to be nearly impossible—primarily due to the overwhelming amount of data generated, the difficulty of debugging and development, and the lack of analysis tools.

To address these problems, we have adopted the idea of "computational steering" and reorganized the code with a focus on modularity and integration of various components such as data analysis, visualization, and simulation [14, 15, 16]. Python serves as the glue of this system.

Rather than having a large monolithic application, the new organization features a collection of loosely organized modules. Most of the functionality is found in a collection of C library files for running simulations, performing data analysis, message passing and other things. A collection of Python scripts are also available. These scripts perform common tasks, and form the foundation of an object oriented visualization system we are developing.

The user provides C code for initial conditions, boundary conditions, numerical integration methods, and any problem specific features. While this code relies heavily on the base set of C libraries, it is completely independent of the Python interface (and can, in fact, be compiled without it). However, if the user would like to use Python, they simply write a SWIG interface file containing their functions. Simulation scripts and new functionality can also be written in Python as needed.

5.1 Extending and Controlling the System

The interface to the SPaSM code is built automatically using SWIG. As a result, one simply declares various C functions which automatically appear as Python commands when the code is compiled. After compilation, the code can be controlled with scripts such as the following :

```
# Shock wave problem (Python script)
nx             = 15
ny             = 15
nz             = 50
shock_velocity = 8.5
temp           = 0.1
```

```
width          = 0.3333
r0             = 1.0901733
gap            = 0.10
cutoff         = 2.0
Dt             = 0.0025

ic_shock(nx,ny,nz,shock_velocity,width,gap,temp,
         r0,cutoff)
init_lj(1,1,cutoff)
set_boundary_periodic()
set_path("/sda/sda1/beazley/shock2")
timesteps(10000,25,25,500)
```

When new functionality is needed, an ordinary C function can be written and its prototype placed into the interface file. Since no Python specific code is involved, any new functionality is easy to re-use in other kinds of C/C++ applications (even if they don't involve Python).

5.2 Interactive Simulation

Since Python is interpreted, it is possible to run SPaSM in an interactive mode. In this mode, the user is presented with a single prompt even though tens to hundreds of copies of the interpreter are running (our parallel I/O wrappers make this possible). Any commands typed by the user are executed in a pure SPMD mode with execution taking place on all processors. This environment is particularly useful for setting up problems and examining the state of a simulation. Here is a sample session :

```
.cm5-5 {106} > SPaSM -p4:4:2
SPaSM 3.0 (alpha) ==== Run 190 on cm5-5 ==== Wed!

Using Python 1.3 (Sep  8 1996)  [GCC 2.6.3]
Copyright 1991-1995 Stichting Mathematisch Centr!

SPaSM > ic_test()
Setting up test initial condition.
23776 particles created.
SPaSM > from vis import *
Setting image server to  sleipner  port  35219
SPaSM > ke = Spheres(KE,0,20)
SPaSM > ke.draw_processors=1
SPaSM > ke.show()
...
SPaSM > SPaSM_processors(2,4,4)
```

In the example, the user has set up an initial condition. A visualization module is then loaded (which attaches to a user's workstation). At this point the use can run simulations and analyze data. In this case, a plot showing the processor layout has been generated as shown in Figure 1. The user is free to change most aspects of the code at any time including the layout of processors and other simulation parameters.

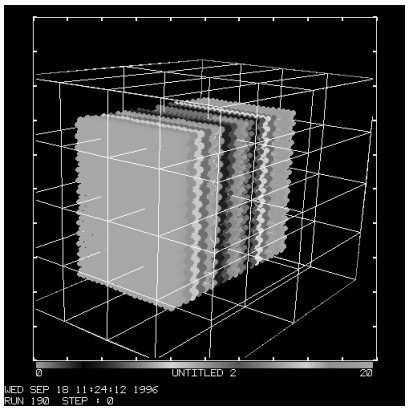

Figure 1. Particles and processor layout

6 Debugging with Python

Since SWIG also provides access to C data structures, it is possible to access the underlying data structures in our simulation directly. For example, the algorithm used by SPaSM relies on creating a large collection of small subcells[13]. We can examine these subcells on each processor as shown below. By default, output is from processor 0, but this is easy to change.

```
SPaSM > c = first_cell()
SPaSM > print c
Cell [ ptr = f3c78, n = 0 ]
SPaSM > max = 0
SPaSM > for i in range(0,Xcells*Ycells*Zcells):
    ...         if c[i].n > max : max = c[i].n
SPaSM > print max
14
SPaSM > pn(5)
(pn 5) SPaSM > print max
16
```

While this is only a simple example, it is possible to perform quite sophisticated debugging and diagnostic operations entirely within the Python interpreter. This can be done without recompiling the C code or quitting a running simulation. While this type of debugging certainly won't replace existing parallel debuggers, it provides an extremely powerful application specific debugging capability that can be used to explore data and examine the system in ways not commonly found in traditional debuggers.

7 Interpreted Message Passing

One of the most interesting features of this approach is that it is even possible to add message passing operations to the Python interpreter itself. Using SWIG, it is possible to build interfaces to CMMD, PVM, MPI, or other libraries

[10, 17, 8]. This allows Python interpreters to send messages to each other as would be done in C/C++. The following session shows a user interactively sending a Python list from processor 0 to all of the other processors using the PVM library on a Cray T3D.

```
.t3d {118} > python
Starting Python on 32 processors...
Python 1.3 (Aug  6 1996) [C]
Copyright 1991-1995 Stichting Mathematisch Centr!
>>> from pvm3 import *
>>> execfile("parallel.py")
>>> me = pvm_get_PE(pvm_mytid())
>>> nproc = pvm_gsize("")
>>> if me == 0:
...     a = [1,2,3,4]
... else:
...     a = [ ]
>>> print a
[1,2,3,4]
>>> if me == 0:
...     for i in range(1,nproc):
...             pvm_initsend(PvmDataRaw)
...             pack_list(a)
...             pvm_send(i,1)
... else:
...     pvm_recv(0,1)
...     a = unpack_list()
>>> pprint(a,range(0,nproc))
pn 0 : [1, 2, 3, 4]
pn 1 : [1, 2, 3, 4]
pn 2 : [1, 2, 3, 4]
...
>>>
```

As with C,C++, or Fortran, it is still possible to deadlock the machine and to experience all of the other problems associated with message passing. However, having an interpreted message passing environment is an interesting way to experiment since it is unnecessary to write any C code (or to recompile after every code modification).

8 Performance Concerns

Python provides our application with a high degree of flexibility, but performance is still of great concern. While it is true that Python runs significantly slower than C, most of the core functionality of our application is still written in C. Python is mainly used for control and writing the outer loop of a large calculation. Table 1 shows some performance measurements for a recent simulation in which the entire outer loop was implemented in Python vs. a simulation implemented entirely in C. Given that the outer loop takes much less than 1% of the overall CPU cycles, the fact that it is implemented in Python is of little concern (as confirmed by the table).

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

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

9 Conclusions

We have been using the techniques described in this paper with great success with our molecular dynamics application. While it is too early to provide any sort of formal "user study", we would like to outline some of the results of taking this approach :

- Emphasizing code modularity has resulted in a system that is more robust, reliable, and flexible.
- Scripting languages such as Python provide an extremely lightweight mechanism for building interactive parallel applications. The addition of Python to our code resulted in only a 10% memory overhead and still permits us to perform very large calculations. Currently, we are using Python scripts to control production simulations running on our 512 processor CM-5.
- We have recently built an object oriented data analysis and visualization system that is directly integrated with our simulation code. The high performance aspects of the system are implemented in C while the object oriented design is implemented entirely in Python. This system allows us to remotely visualize 100 million atom datasets from ordinary UNIX workstations and standard internet connections. Since visualization is performed on the parallel machine itself, we can make images in only a matter seconds—not minutes or hours (This work is still in progress and will be reported elsewhere.)
- Extending the system is now extremely easy. Users do not need to understand the details of the underlying Python implementation and can add new functions by simply declaring them in an interface file.
- This approach has resulted in the reuse of various software components. For example, the graphics library we developed for visualizing MD simulations can be used as a stand alone package in unrelated projects.
- By eliminating most of the problems of building highly modular and interactive applications, we have

been able to focus on the real problem at hand—performing large scale materials science simulations.

By providing a simple set of tools, we have been able to build an extremely powerful parallel application capable of dealing with 100 million particle data sets. Yet, we have we have been able to do this without relying on any sort of special purpose parallel computing environment, rewriting all of our C code, sacrificing performance, or making things unnecessarily complicated. We firmly believe that this is a model than can be successfully applied to other large-scale parallel computing applications that demand flexibility, portability, and high performance. In the future we hope to extend this system to provide better support for shared memory architectures.

10 Acknowledgments

We would like acknowledge Brad Holian, Shujia Zhou, and Niels Jensen of Los Alamos National Laboratory, Tim Germann at UC Berkeley, and Bill Kerr at Wake Forest University for their work on the SPaSM code. Paul Dubois and Tser-Yuan (Brian) Yang at Lawrence Livermore National Laboratory have also provided valuable feedback concerning the parallelization of Python and its use in physics applications. We would also like to acknowledge the Scientific Computing and Imaging group at the University of Utah for their continued support and Guido van Rossum for many discussions concerning the implementation of Python (and for making such an excellent tool). Finally, we would like to acknowledge the Advanced Computing Laboratory at Los Alamos National Laboratory. Development of the SPaSM code has been under the auspices of the United States Department of Energy.

11 Software Availability

All of the tools described in this paper are in the public domain and available. Python can be obtained from the Python homepage at `www.python.org`. SWIG is available at `www.cs.utah.edu/~beazley/SWIG`. The parallel modifications to Python can be obtained by contacting the authors.

References

[1] Guido van Rossum and Jelke de Boer, *Interactively Testing Remote Servers Using the Python Programming Language*, CWI Quarterly, Volume 4, Issue 4 (December 1991), Amsterdam, pp 283-303.

[2] Mark Lutz, *Programming Python*, O'Reilly and Associates, (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, *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.

[8] *CMMD User's Guide*, Thinking Machines Corporation (1995).

[9] *shmem Library Reference*, Cray Research Incorporated (1994).

[10] MPI: A Message-Passing Interface Standard, `http://www.mcs.anl.gov/mpi/index.html`.

[11] P. Corbett, et al. *MPI-IO : A Parallel File I/O Interface for MPI*, NAS Technical Report NAS-95-002. (1995)

[12] 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.

[13] 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.

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

[15] S.G. Parker and C.R. Johnson. *SCIRun: A Scientific Programming Environment for Computational Steering*, Supercomputing '95, IEEE Computer Society, (1995).

[16] G. Eisenhauer, et al. *Opportunities and Tools for Highly Interactive Distributed and Parallel Computing*, Proceedings of the Workshop on Debugging and Tuning for Parallel Computer Systems, Chatham, MA. 1994. IEEE Computer Society, (1996). pg. 245-278.

[17] A. Geist,et al. *PVM : Parallel Virtual Machine–A User's Guide and Tutorial for Network Parallel Computing*, MIT Press, (1994).