LA-UR-96-3386

CONF-970477--1

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36.



TITLE: Extensible Message Passing Application Development and Debugging with Python

Atuhors David M. Beazley
Peter S. Lomdahl



SUBMITTED TO: International Parallel Processing Symposium (IPPS'97)
April 1-5, 1997, Geneva Switzerland



Los Alamos

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO. 836 R4
ST. NO. 2629 5/81

# 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

September 19, 1996

*Submitted to the International Parallel Processing Symposium (IPPS'97)*

## Abstract

*We describe how we have parallelized Python, an interpreted object oriented scripting language, and used it to build an extensible message-passing C/C++ applications for the CM-5, Cray T3D, and Sun multiprocessor servers running MPI. Using a parallelized Python interpreter, it is possible to interact with large-scale parallel 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 applications of this approach.*

# 1 Introduction

Some 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. While much has been said about the "lack of tools" available for parallel computing, the situation seems to have improved little over the past few years–a fact which we feel is unfortunate, but perhaps indicative of the rapid growth (and demise) of parallel computing systems [1].

Given the complexity of working with parallel machines, there is tendency to develop parallel "problem solving environments" that attempt to hide the underlying complexity of running in parallel 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. Ideally, they should be reusable in a variety of situations. 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!).

If we turn to the workstation and PC world, a very different style of computing is emerging. Rather than building large monolithic systems, applications are being built from small modules and "applets" (for lack of a better word). Languages such as Tcl/Tk, Perl, Python, Visual Basic, and Java have been highly successful not because they are better languages, but because they allow extremely powerful applications to be built out of existing, often diverse, components. More often than not, a useful application can be built in only a matter of hours– not days or weeks. Unfortunately, this does not seem to be the case for parallel machines.

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 a parallel computing framework. It also provides us with a nice debugging, prototyping, and user environment for working with large parallel codes. We hope to illustrate the system with a large-scale molecular dynamics application we have been developing, but the methods are easily applicable to other kinds of applications.

# 2 The Python Language

Python is an interpreted object oriented scripting language developed by Guido van Rossum, at CWI, Amsterdam [2, 3]. It has been steadily increasing in popularity and is often compared to languages such as Tcl/Tk and Perl [4, 5]. 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 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 an exceptionally clean syntax that is easy to read and easy to learn.
- Python is interpreted and can run interactively.
- The language is dynamically typed and has a number of high-level constructs that are often only found in functional languages.
- 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 sophisticated and powerful scripts.
- The language has seen increased use in the scientific community and has a number of numerical extensions [6, 7].
- Python is free, but well supported by the Python Software Activity (PSA).
- We like it.

More information about Python can be found on the internet, or the forthcoming book "Programming Python" by Mark Lutz [3]. Fortunately, the syntax of the language is easily understood and shouldn't present a problem for understanding later examples. The remainder of this paper will focus primarily on the 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 extremely poor 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 clobbering itself 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 [8].

```
/* 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 [9, 10, 11]. Eventually, we would hope to implement the library using parallel I/O libraries such as MPI-IO [12].

## 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 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. For example, the C code and Python wrapper for a factorial function are shown below :

```
/* A factorial function */
int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

/* A Python wrapper function for it */
PyObject *wrap_fact(PyObject *self, PyObject *args) {
    int result;
    int arg;
    if (!PyArg_ParseTuple(args,"d",&arg)
        return NULL;
    result = fact(arg);
    return Py_BuildValue("d",result);
}
```

While writing a single wrapper function isn't too hard, it quickly becomes tedious if there are a large number of functions. Since we would like to use Python as a rapid prototyping and extension language, having to write all of these functions by hand is unacceptable. Thus, we have developed a tool, SWIG (Simplified Wrapper and Interface Generator), that automatically generates Python bindings from ANSI C/C++ specifications (in fact, it also produces Tcl and Perl bindings) [13]. Using SWIG, the user would extend Python by writing the following file :

```
// file : fact.i
%module fact
%{
/* Put headers and support code here */
%}

extern int fact(int n);
```

The module is then compiled and added to Python as follows (under Solaris):

```
unix > swig -python fact.i
unix > gcc -c fact_wrap.c -I/usr/local/include/Py
unix > ld -G fact_wrap.o -o factmodule.so
unix > python
Python 1.3 (May  2 1996)  [GCC 2.5.8]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import fact
>>> fact.fact(6)
720
>>>
```

While this is only a simple example, SWIG supports almost all C/C++ datatypes, C structures, and C++. More information about SWIG can be found in [13]. We now focus on how these tools can be used to build extensible parallel applications.

# 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 [14]. This code has been 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 [15, 16, 17]. Python serves as the glue of this system as shown in Figure 1.

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. These are integrated into Python using a collection of SWIG interface files. 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 completely independent of the Python interface (and can, in fact,

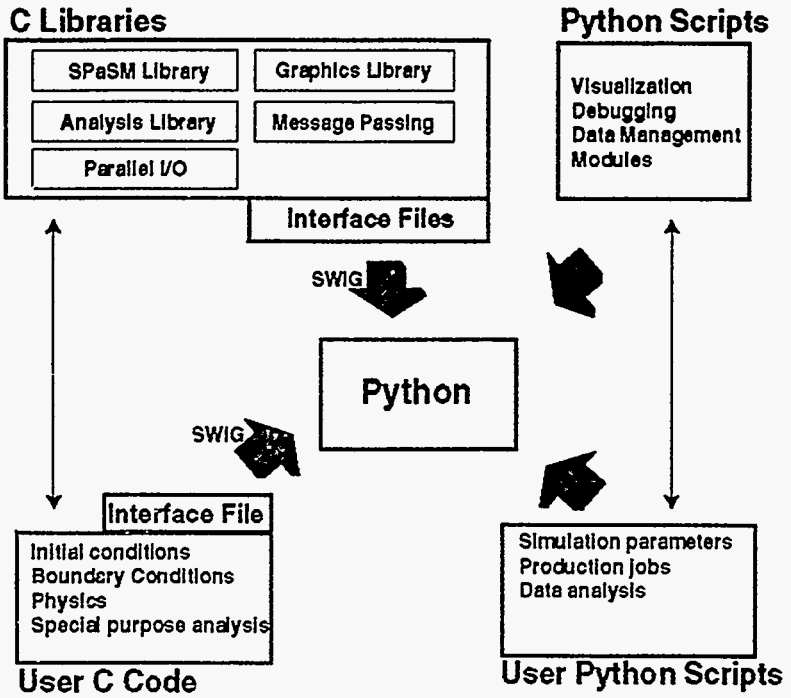



Figure 1: System organization.

be compiled without it). However, if the user would like to use Python, they simply write an interface file for their functions. Simulation scripts and new functionality can then be written in Python as needed.

## 5.1 Extending and Controlling the System

This approach provides an extremely straightforward and easy mechanism for extending the system and controlling large simulations. A user would provide an interface specification such as the following :

```
%module spasm
%{
#include "spasm.h"
%}
%include SPaSM.i            // Include the SPaSM library
%include graph.i            // Graphics library
%include analysis.i         // Visualization library
%include debug.i            // Debugger
%include methods.c          // Output methods

extern void ic_shock(int nx, int ny, int nz, double vel, double width, double gap,
                     double temp, double r0, double cutoff);
extern int  timesteps(int nsteps, int energy_n, int output_n, int checkp_n);
extern void set_boundary_periodic();
```

```
extern void set_boundary_free();
extern void energy();

Real     Dt,TotalTime;

// Different potential energy methods

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

Functions, variables, and constants defined in the interface specification correspond directly to their underlying C counterparts. When the code is compiled, all of the functions appear automatically as Python commands. They can then be used as shown in the following simulation script:

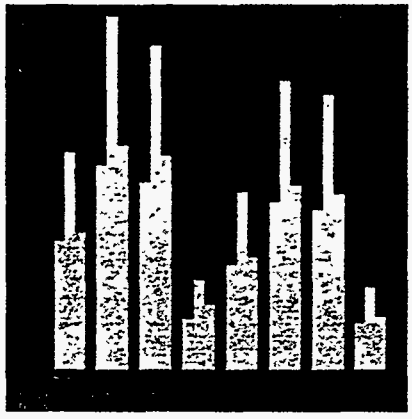
```
# Shock wave problem (Python script)
nx             = 15
ny             = 15
nz             = 50
shock_velocity = 8.5
temp           = 0.1
width          = 0.3333        # Shock width
r0             = 1.0901733     # Lattice spacing
gap            = 0.10          # Gap
cutoff         = 2.0           # Interaction cutoff
cvar.Dt        = 0.0025        # Timestep

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. Its prototype is placed into the interface file and that's it. 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 Sep 18 10:57:11 1996
```



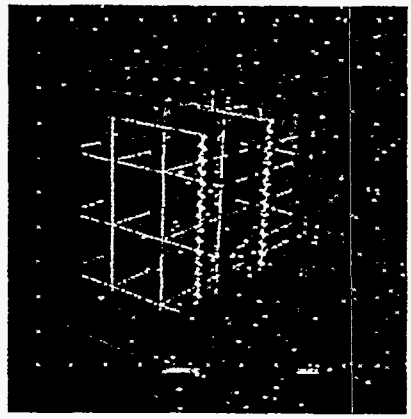


Figure 2: (a) Memory usage display (left), (b) Particles and processor assignments (right)

```
Using Python 1.3 (Sep  8 1996)  [GCC 2.6.3]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam

SPaSM [190] > ic_test()
Setting up test initial condition.
23776 particles created.
SPaSM [190] > from vis import *
Setting image server to  sleipner  port  35219
SPaSM [190] > m = MemoryUse()
SPaSM [190] > ke = Spheres(KE,0,20)
SPaSM [190] > ke.draw_processors=1
SPaSM [190] > ke.show()
...
SPaSM [190] > 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 memory use of the simulation can be displayed as shown in Figure 2a. From this graph, we see that 8 of the 32 processors contain no data. To find out where the load-imbalance is occurring, the particles and the regions assigned to each processor can be displayed as shown in Figure 2b. At this point, the user is free to change the geometry of the system or even the arrangement of processors. At any time, it is possible to start running the simulation and watch its progress.

# 6 Debugging with Python

Given the direct access to C functions and variables provided by SWIG, it is possible to interactively call C functions and query the values of system variables. It is also possible to access C data structures directly. For example, the algorithm used by SPaSM relies on creating a large collection of small subcells[14]. The data structure for each Cell is described by the following:

```
typedef struct {
     int          n;     /* Number of atoms          */
     vParticle *ptr;     /* Pointer to their location */
} Cell;
```

When added to Python, we can access this structure directly. In fact, with a little extra work it is even possible to manipulate arrays of Cells in an entirely natural manner. In the following example, we extract the first subcell in a simulation and loop over all of the subcells to find the maximum number of particles in any given subcell on each processor. When printing the value, output is from processor 0, but we can easily switch to another processor and print out its value as shown.

```
SPaSM [32] > c = first_cell()
SPaSM [32] > print c
Cell [ ptr = f3c78, n = 0 ]
SPaSM [32] > max = 0
SPaSM [32] > for i in range(0,cvar.Xcells*cvar.Ycells*cvar.Zcells):
       ...         if c[i].n > max : max = c[i].n
SPaSM [32] > print max
14
SPaSM [32] > pn(5)
(pn 5) SPaSM [32] > print max
16
(pn 5) SPaSM [32] >
```

Thus, it is possible to perform 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. For example, consider the following SWIG interface file :

```
%module message
%{

int reduce_int_max(int a) {
    int result;
    MPI_Allreduce(&a,&result, 1, MPI_INT, MPI_MAX, MPI_COMM_WORLD);
    return result;
}
%}

int reduce_int_max(int a);
```

When added to Python, we can modify our earlier debugging session in a natural way by performing a global reduction :

```
...
SPaSM [32] > max = 0
SPaSM [32] > for i in range(0,cvar.Xcells*cvar.Ycells*cvar.Zcells):
       ...         if c[i].n > max : max = c[i].n
SPaSM [32] > max = reduce_int_max(max)
SPaSM [32] > print max
18
SPaSM [32] >
```

With a little more work, it is possible to add direct counterparts to various MPI, PVM, or CMMD message passing operations to the Python interpreter [11, 18, 9]. As a result, Python scripts may send messages to each other just as can 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 the Cray T3D.

```
.t3d {118} > python
Starting Python on 32 processors...
Python 1.3 (Aug  6 1996) [C]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> 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]
pn 3 : [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 scriptable message passing environment is an interesting way to experiment with message-passing since it is unnecessary to write any C code (or to recompile after every code modification).

# 8 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. In fact, code size has dropped by more than 25%.
- 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 systems (in fact, with SWIG we were even able to turn it into a Perl5 module for making 3D graphs from http server logs).
- 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 including multiprocessor Sun and SGI workstations.

# 9 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.

# 10 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 `http://www.python.org`. SWIG is available at `http://www.cs.utah.edu/~beazley/SWIG`. The parallel modifications to Python can be obtained by contacting the authors.

# References

[1] C.R. Cook, C.M. Pancake, R. Walpole, Proceedings of Supercomputing '94, IEEE Computer Society, (1994). pg, 126-133.

[2] 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.

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

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

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

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

[7] 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).

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

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

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

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

[13] 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.

[14] 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.

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

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

[17] 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.

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