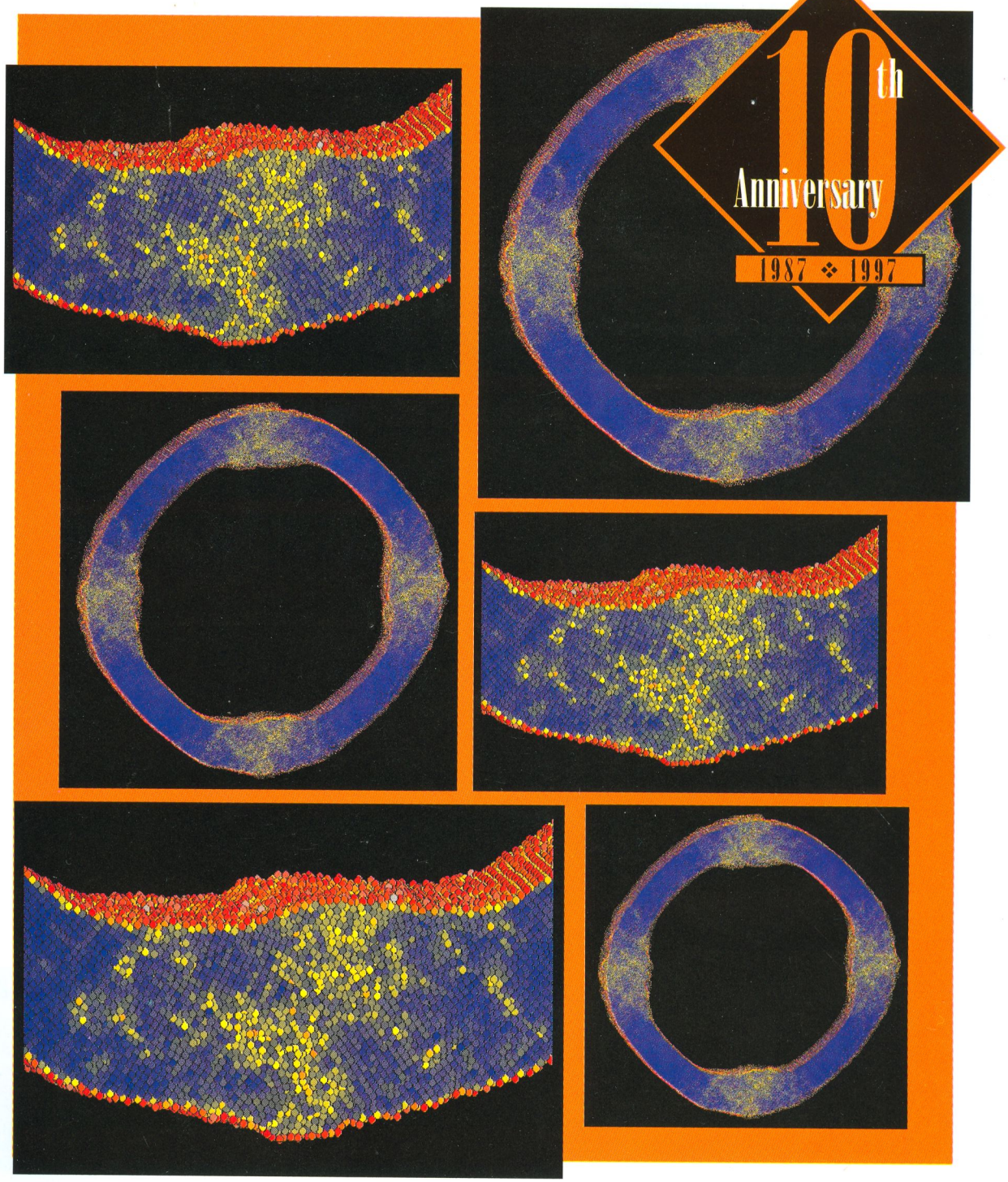
The Computer Magazine for Scientists and Engineers

COMPUTERS IN PHYSICS

MAY/JUN
1997



AMERICAN INSTITUTE OF PHYSICS

Controlling Data Glut

Anniversary Book Reviews

Controlling the Data Glut in Large-Scale Molecular-Dynamics Simulations

David M. Beazley and Peter S. Lomdahl

Scripting languages bring structure and convenience to large-scale scientific codes and permit simple analysis techniques to be deployed

With the growing availability of more powerful computing platforms, large-scale computer simulations of materials are becoming increasingly popular. In particular, the growth in recent years in the area of large-memory parallel multicomputers has allowed classical molecular-dynamics simulations to be performed in three dimensions with system sizes that are approaching experimentally relevant regimes, namely, the micrometer scale. Simulations at this length scale promise eventually to bridge the gap between the microscopic (atomistic) description and the macroscopic (continuum-mechanics) description of materials.

In 1992 we began work on a parallel molecular-dynamics code that would allow us to run classical molecular-dynamics calculations on systems with many millions of atoms. This work was in part inspired by the growing needs of the materials-science community to simulate systems large enough to be experimentally relevant and in part by the recent availability of the 32-Gbyte, 1024-processor Thinking Machines CM-5 parallel computer at the Advanced Computing Laboratory in Los Alamos. Our work resulted in a computer code called SPaSM, for Scalable Parallel Short-range Molecular dynamics, that has been described elsewhere in detail[1-3] (see "Cluster Impacts on Surfaces," p. 231). The SPaSM code has been ported to several multicomputer platforms supporting both message-passing (on distributed memory computers) and multithreading (on shared-memory systems). Although our work was at least partly successful, in terms of increasing the execution speed and accessible system size, it also opened a whole area of new problems.

David Beazley is a graduate student working toward his Ph.D. in the Department of Computer Science at the University of Utah, Salt Lake City, UT 84112. E-mail: beazley@cs.utah.edu

Peter Lomdahl is a staff member in the Condensed Matter and Statistical Physics Group in the Theoretical Division at Los Alamos National Laboratory, Los Alamos, NM 87545. E-mail: pxl@lanl.gov

Direct microscopic simulations at the micrometer scale require at least hundreds of millions, if not billions, of atoms. With such large systems, the physicists and materials scientists interested in using the tools we have developed are confronted with formidable interpretative challenges. How do you analyze and extract useful information out of the enormous data sets resulting from such simulations? Consider a molecular-dynamics simulation with 100 million atoms that runs for 10 ns. You might want to produce a "snapshot" of the system every 0.25 ns. A typical snapshot might consist of three coordinates and a couple of scalar values, for a total of 20 bytes/atom. This will result in forty 2-Gbyte files, for a total of 80 Gbytes, which typically exceeds the disk quota of most projects at large U.S. supercomputing installations.

Even a single 2-Gbyte snapshot presents a gigantic problem for today's high-end workstations. They simply do not have enough memory or CPU power to process this amount of data effectively. Even when a high-powered workstation is available for such a project, moving the data to the machine may be prohibitively difficult—especially when it is located at a remote location.

Faced by these daunting problems, we were forced to re-evaluate our approach to large-scale simulation, analysis, and visualization. In this article we describe some of the main ideas behind this process and the system that has evolved.

Physics, software, and the need for simple tools

As computational physicists, our primary goal has been to study physics, not computer science. However, computing

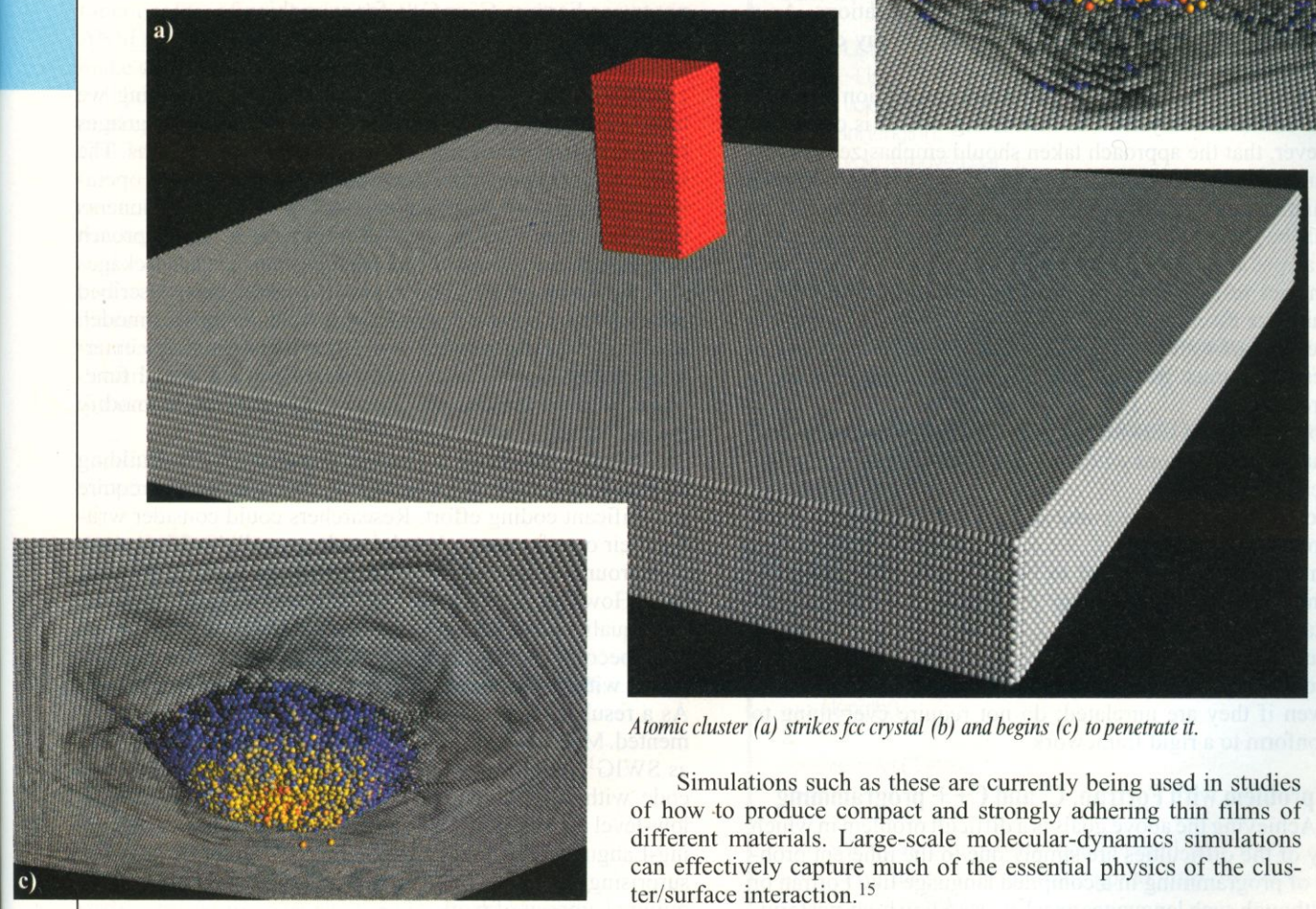
 S0894-1866(97)00603-2

Cluster Impacts on Surfaces

This modest-size simulation was performed in 1993 with a little more than 1 million atoms. A cluster of 1000 atoms impacts the {111} surface of an fcc crystal, with initial conditions as shown in part a of the figure. The interatomic forces were represented by a Lennard-Jones spline potential in this case. The initial kinetic energy of the small cluster was approximately 20 keV.

Part b shows a snapshot shortly after the impact. The anisotropy of the elastic constants in the fcc lattice is clearly seen in the hexagonal pattern generated by the impact of the square cluster.

Part c shows the underside of the thin plate a little further into the simulation. Careful analysis confirms that the velocity fields generated on the surface and inside the solid are in accordance with standard elastic theory.[15]



Atomic cluster (a) strikes fcc crystal (b) and begins (c) to penetrate it.

Simulations such as these are currently being used in studies of how to produce compact and strongly adhering thin films of different materials. Large-scale molecular-dynamics simulations can effectively capture much of the essential physics of the cluster/surface interaction.[15]

issues play a central role in two different aspects of our research. First, there is the development of new physical models, initial conditions, and numerical methods. These are typically implemented in C, make use of a variety of data structures, and involve parallel-programming methods. Second, there is the post-processing of simulation data. This traditionally involves the use of special tools or third-party packages for data analysis and visualization. Both tasks are important, and it is their separation that is at the root of our problem. Third-party or commercial tools are typically incapable of handling the data generated by simulation of large problems, and simply trying to move data around and trying to get different tools to work can waste an appalling amount of time. This situation is frustrating and unacceptable.

To alleviate these difficulties, considerable effort has been expended in the development of computational "problem-solving environments." These systems attempt to integrate various software components into a unified system and hide the underlying complexity from the user. Their ultimate goal is to tightly couple all aspects of a problem including simulation, data analysis, and visualization. Although we feel that this is an admirable goal, current implementations leave much to be desired. Most notably, such systems tend to provide very poor performance on large problems. Moreover, they are too complicated to extend and use, and they tend to rely too heavily on expensive graphics workstations. As a result, they are of limited usefulness in everyday computational-physics research.

Nevertheless, the idea of providing integration of simulation and data analysis is an interesting one. It is our view, however, that the approach taken should emphasize simplicity and efficiency rather than attempting to create a highly sophisticated package with thousands of features. Tools should be simple to use, offer high performance, be maintainable across a wide variety of machines, and not get in the way of physics research. In particular, we feel that the following features are essential:

- Performance and memory efficiency. Available computer resources must be used wisely when working with large problems.
- Extensibility. Users must be able to add new features and make modifications without having first to get a computer-science degree.
- Portability. Machines come and go; software tends to stick around for a while.
- Interactivity. Users need to be able to modify the data-analysis and visualization procedures at intermediate stages in the computation.
- Support of existing code.
- Loose integration. Allow varied components to be used, even if they are unrelated; do not require everything to conform to a rigid framework.

The problem with Fortran, C, and C++ programming

Achieving the above goals is a difficult problem in which many of the difficulties are simply due to the inherent problems of programming in a compiled language like Fortran or C. Although such languages excel at providing high-performance and low-level systems programming, they do a poor job at most other tasks. It is not easy to build an interactive program without a lot of work. Extending such systems with new capabilities requires more work and can result in a huge mess without a lot of attention. If our ultimate goal is to build a system that integrates simulation, data analysis, and visualization, the problem suddenly becomes very difficult. How do we get different components to talk to each other? How do we make it easy to add new features? Do we adopt object-oriented programming techniques and reimplement everything in C++? How do we avoid code-bloat and making the system so complicated that no one will want to use it anymore? When you add the difficulty of debugging and compiling such a system, it is no surprise that this approach seems to be plagued with so many pitfalls.

In our experience, it seems that most physics codes are relatively simple—it is all the supporting code that is complicated. Much of this complexity occurs, in large part, because compiled languages simply are not very good at managing the rapidly changing nature of research applications. When working on a problem, we are always trying different methods, changing parameters, adding new functions, and investigating new analytical methods. Writing a C program to handle all these possibilities is a nightmare. While some may see object-oriented programming in C++ as a solution to all of these problems, we suggest that a better approach might be not to use Fortran, C, or C++ for everything.

Controlling applications with scripting languages

Rather than using compiled languages for everything, we have been exploring the use of interpreted scripting languages for controlling and interacting with physics applications. The idea is really quite simple—write performance-critical operation in C and use a scripting language for gluing components together and providing a high-level interface. The approach is essentially identical to that used in commercial packages such as Matlab or IDL. This approach has also been described previously in *Computers in Physics*.[4] By using this model, users can control a physics application by interactively entering commands, writing scripts, and adding additional functionality in the interface language without having to modify the underlying C code.

Finding an interface language. Unfortunately, building an interactive physics application in this manner can require a significant coding effort. Researchers could consider writing their own language, but doing this usually requires some background in compilers and a significant amount of spare time. However, in recent years, a number of freely available, high-quality scripting languages such as Tcl, Python, and Perl have become available.[5-9] All these languages can be integrated with C/C++ code and have tens of thousands of users. As a result, they are stable, well-debugged, and well-documented. More recently, automated code-generation tools such as SWIG[10] have made it extremely easy to integrate C/C++ code with such languages without having to know many low-level details (see "SWIG Automatically Builds Scripting-Language Interfaces," p. 233). Consequently, it can be surprisingly easy to build a powerful physics code with a minimal amount of work.

The benefits. Scripting languages provide a simple yet

powerful mechanism for controlling physics codes. They are highly memory-efficient and easy to port between a variety of machines. They provide high-level data structures and make it possible to glue various software components together. They can also be run interactively in a manner similar to commercial packages. Given the widespread use of common languages, a huge number of third-party modules that can be used in an application are also available to a user. Most important, such languages provide a physicist with an easy-to-use, highly precise user interface that provides a tremendous amount of flexibility without getting in the way of the real problem at hand.

Extending the SPaSM molecular-dynamics code

For the past two years, we have applied the use of scripting languages to the SPaSM molecular-dynamics code. In doing so, we have provided a mechanism for integrating simulation, data analysis, and visualization. As a result, we are now able to manage effectively very large simulations involving tens to hundreds of millions of atoms. This can be done using ordinary workstations and network connections. Although parts of this system have been described elsewhere,[11-14] the next few sections illustrate various aspects of the system and how it is being used.

Automated interface generation with SWIG. The core functionality of our molecular-dynamics code is written in ANSI C. To build a scripting-language interface, we want to make certain C functions visible to the user. These functions will become new "commands" that can be typed interactively or placed in scripts. To do this, we are using SWIG (Simplified Wrapper and Interface Generator), which parses ANSI C/C++ declarations and automatically generates the code needed to extend a variety of scripting languages with new functions. SWIG supports Tcl, Python, Perl, and Guile, and so the choice of scripting language is mostly a matter of user preference (we have been using Python, but this is not a strict requirement). To create an interface, we simply create an interface file such as the following. It basically consists of the declarations for the functions we want to make available to the user of the code:

```
// spasm.i : SWIG Interface file
%module spasm
%{
#include "SPaSM.h"
%}

void memory(int size);
void geometry(double xmin, double ymin, double zmin,
   double xmax, double ymax, double xmax, double cutoff);
void redistribute();
void boundary_periodic();
void init_lj(double epsilon, double sigma);
void force_lj();
void integrate_adv_coord(double dt);
void integrate_adv_velocity(double dt);
void output_particles(char *filename);
void ic_crack(int nx, int ny, int nz, double vel, double width,
   double gap, double temp, double r0, double cutoff);
```

SWIG Automatically Builds Scripting-Language Interfaces

Simplified Wrapper and Interface Generator (SWIG) is an automated tool that we have developed for building scripting-language interfaces to C/C++ programs. The user supplies a list of functions and variables in the form of ANSI C/C++ prototypes. These are turned into the appropriate "wrapper code" needed to access the functions from a scripting-language interface. Here is a simple example:

```
// File : example.i
%module example
%{
#include "header.h"
%}

extern int fact(int);                          // A simple function
FILE *fopen(char *filename, char *mode); // Functions with pointers
void write_data(FILE *f);

void fclose(FILE *f);

class List {
public:
        List();
        ~List();
        void insert(char *);
        int length();
        void remove(char *);
};
```

These functions can be compiled and used by performing steps similar to the following:

```
unix > swig -python example.i
unix > cc -c example_wrap.c -I <includes>
unix > ld -shared example_wrap.c  <objs> -o examplemodule.so
unix > python
Python 1.4 (Oct 28 1996) [GCC 2.7.2.1]
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> from example import *
>>> fact(4)
24
>>> f = fopen("myfile","w")
>>> write_data(f)
>>> fclose(f)
>>> l = List()
>>> l.insert("Dave")
>>> l.insert("Peter")
>>> l.length()
2
>>> l.remove("Dave")
>>>
```

SWIG is extensively documented and freely available. For more information, go to http://www.cs.utah.edu/~beazley/SWIG.

At compile time, these specifications are automatically turned into scripting-language commands. This automation makes the interface-building process extremely easy and frees the user from worrying about the low-level details.

Using the scripting-language interface. Now to run a simulation, we can simply write a script that calls our various C functions. The following Python script shows a simple example of how this might be done.

```
from spasm import *
nx      = 15
ny      = 15
nz      = 50
vel     = 8.5
temp    = 0.1
width   = 0.3333
r0      = 1.0901733
gap     = 0.1
Dt      = 0.0025
cutoff  = 2.0
nsteps = 1000
freq    = 50

# Set up a problem

ic_crack(nx,ny,nz,vel,width,gap,temp,r0,cutoff)
init_lj(1,1,cutoff)

for step in range(0,nsteps):
    integrate_adv_coord(Dt)          # Update positions
    boundary_periodic()              # Apply boundary conditions
    redistribute()                   # Move particles between cells
    force_lj()                       # Calculate force
    integrate_adv_velocity(Dt)       # Update velocities
    if (step % freq) == 0:
        output_particles('Dat'+str(step))
```

Although simplistic, this illustrates the general idea. Rather than using C, we can specify the control of a simulation in a script. Unlike older versions of our code, there is no longer any main program written in C defining an exact sequence of operations; the user is given full freedom to call commands in any order and at any time. Most commands in the script are still C functions, but if we want to change parameters or modify the integration procedure in some manner, this is easily accomplished. In a sense, this is the real point: When an interpreted language handles the overall structure and control of a simulation, procedures become easy to change, and modification of the underlying C code is not really required.

In SPaSM, we have more than 200 C functions for running simulations, performing analysis, and visualization. Python is used to provide an environment for interacting with these functions and performing various high-level operations.

Interactive application-specific debugging

A common problem that we face almost daily is the debugging and testing of physical applications. Compared with the traditional task of resolving program crashes, this is more difficult. We need to be able to examine the state of a simulation and make sure we are getting the right answers. Often we need to compare output with other codes to verify correct operation. In other instances, subtle bugs can work their way into a simulation and become almost impossible to find later.

Since we are using an interpreted language interface, we can easily perform highly sophisticated diagnostics and debugging. With additional specifications, SWIG provides direct access to underlying data structures and functions, and so it is rarely necessary to write C code or to recompile the system. Suppose, for example, that we are implementing a new particle interaction. To verify its operation, we might need to to dump the data to a file as an ASCII-formatted table, possibly for visual inspection or in a format suitable for input into a different program. This is easily accomplished using a simple script such as the following:

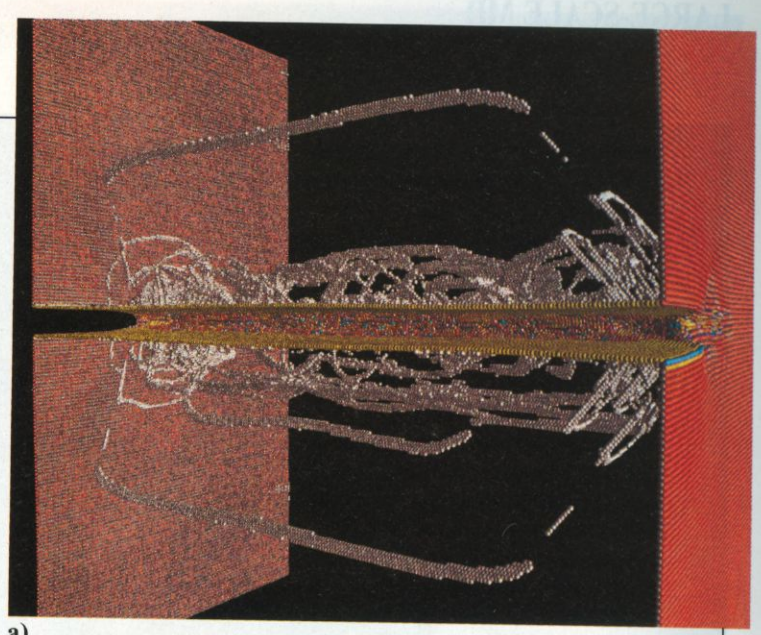
```
def dump(filename):
    f = open(filename,"w")
    p = first_particle()
    n = count_particles()
    for i in xrange(0,n):
        f.write("%d %f %f %f\n" % (p[i].type, p[i].r.x, p[i].r.y, p[i].r.z))
    f.close()
```

The script is simple to write and works exactly as you would expect. It loops over all the particles stored in memory and extracts individual members in a syntactically similar manner as in C. There is really no need (or desirability) to have this functionality in the C code. In many cases, debugging code might only be used once or twice. Scripting languages are extremely convenient for writing throwaway code and performing unusual operations.

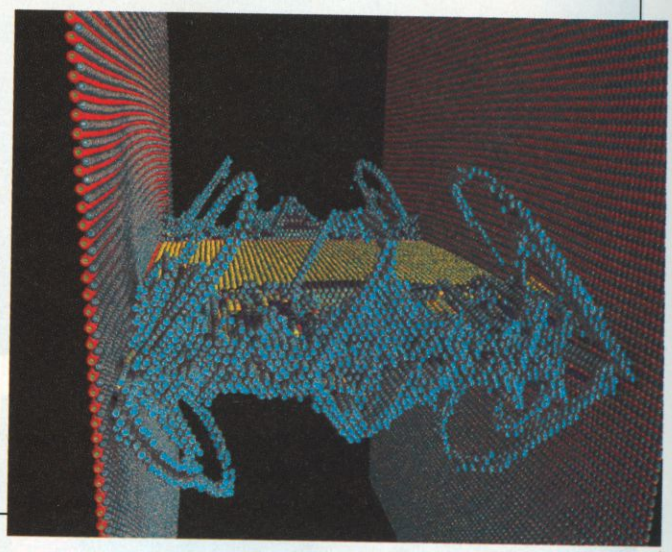
High-level data analysis and visualization

Using the scripting-language interface, we recently developed a powerful data-analysis and visualization system. The "engine" for this system consists of a small number of useful C functions and a simple graphics library for creating images on parallel machines. The library supports two- and three-dimensional images and creates GIF files as output. Python has been used to encapsulate the library into a fully object-oriented visualization system that is visible to the user. The associated Python code provides structure and organization to the underlying C code. Simple operations such as drawing tickmarks and axis labels are also implemented in Python and can be easily customized. Within this system, each "image" is an object that can be manipulated by the user as needed. There is no theoretical limit on the number of such images or how they can be used. To illustrate how this works, we consider the simple example of finding dislocations in a material.

Example: Visualizing dislocations in a solid. Visualizing dislocations in a material is useful during simulations of fracture. For this purpose, we can use a simple scheme in which the potential-energy contributions of individual atoms are considered. The displacement of atoms in the lattice affects their contribution to the potential energy, allowing us to distinguish them from the non-displaced atoms. A C function to perform this identification and make a plot would look roughly like this:



a)



b)

Selective plotting based on energy identifies atoms participating in dislocation phenomena. Parts (a) and (b) reveal sensitivity of the phenomena to details of the interatomic potential.

Ductile Failure and Dislocation Generation in Copper

This example shows how a large-scale simulation with 35 million atoms can be analyzed using the technique described in the text to show only those atoms with a potential energy slightly above the pristine bulk value. Part a of the figure shows a rich dislocation-loop structure as a result of a crack-blunting process in copper. The solid was pulled apart at a constant strain rate of approximately 10^9/s applied in the vertical direction. The crack blunts by emitting dislocation loops: blunting loops parallel to the crack front and oblique jogging loops at the interface with the free surface. Half the computational sample has been cut away so that it is possible to see the inside of the crack.

The importance of the specifics of the interatomic potential can be appreciated by inspecting part b of the figure, where a similar, but smaller simulation of approximately 1.8 million atoms has been performed with a Morse potential ($\alpha = 7$). In this case only jogging dislocation loops are seen. In general, the shorter the range of the potential, the more brittle the material and the harder it is to emit blunting dislocations.

Details of this work can be found in Ref. 16. See also the SPaSM home page: http://bifrost.lanl.gov/MD/MD.html.

```
void plot_dislocations(Plot3D *p3, double pemin, double pemax) {
  Particle *p = Particles;
  int  i, npart;
  int  color;
  npart = count_particles();
  for (i = 0; i  < npart; i++,p++) {
    if ((p->pe >= pemin) && (p->pe  <= pemax)) {
      ... compute color value ...
      /* Plot it using our C graphics library */
      Plot3D_plot(p3,p->r.x,p->r.y,p->r.z,color);
    }
  }
}
```

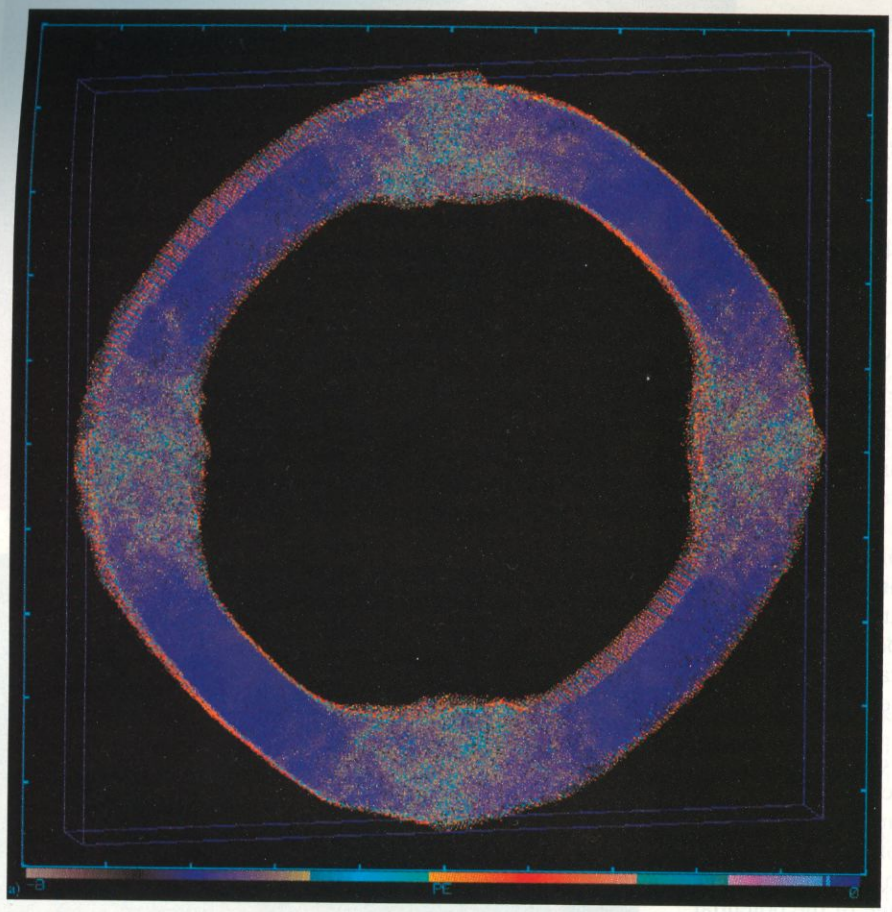
Given a data structure corresponding to a three-dimensional plot and some clipping values, this function simply scans through all the particles and displays only those of interest (see "Ductile Failure and Dislocation Generation in Copper," this page). By using SWIG, we can make the function available in Python. Now, to define a new type of "image" for plotting dislocations, we proceed as follows (in Python):

```
class PlotDisl(Image3D): # Inherits from a generic 3D image class
  def __init__(self,min,max):
    Image3D.__init__(self, ...
    self.pemin = min
    self.pemax = max
  def draw(self):
    self.newplot()
    plot_dislocations(self.p3,self.pemin,self.pemax)
```

Common operations such as rotation, translation, and scaling are found in the generic Image3D class, and so we get them for free through inheritance. Finally, to use our new image, a user can issue commands such as the following:

```
>>> p = PlotDisl(-7,-5)      # Create a new image
>>> p.rotd(45)               # Rotate it down
>>> p.zoom(200)              # Zoom in
>>> p.pemin = -6.5           # Change minimum value
>>> p.show()                 # Regenerate the image
```

Admittedly, this is only giving a hint of what is possible. The underlying C code for making a plot remains simple and uses built-in graphics-library functions. At a higher level,



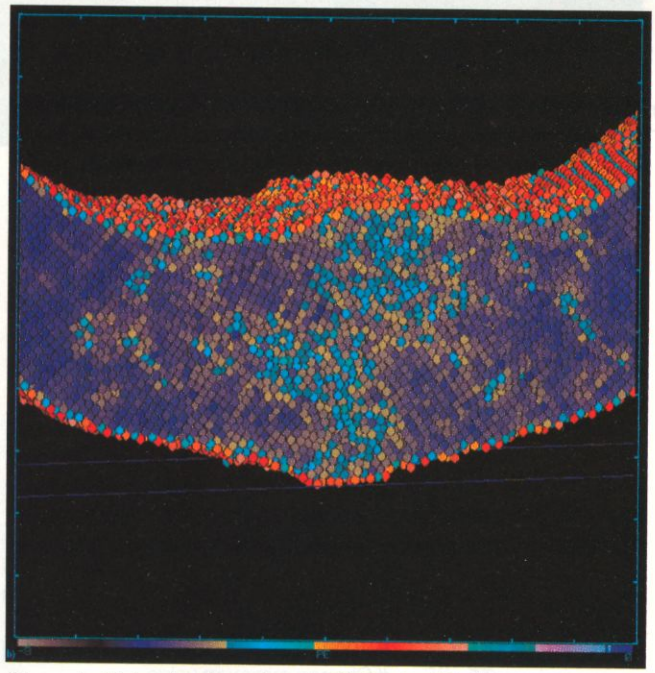


Figure 1. Molecular-dynamics simulation in SPaSM investigates an imploding cylinder of an fcc crystal. At the top is a full view in which atoms are colored according to the value of their potential energy; at the bottom is a detail of the region where the lattice has melted locally. Different views like these are easily generated with the graphics subsystem.

each "image" has several dozen methods for performing rotation, scaling, translation, and clipping. All these methods are inherited automatically when a new image is created; a user never has to worry about the implementation. Because various display modes are handled by the library, images can be displayed on a user's screen, dumped to disk, or transmitted over the Internet. In our current system there are about ten different types of images. These images can be generated in parallel with a running simulation or during postprocessing (see Fig. 1 for an example).

Currently, we are using this system for managing data from simulations involving millions to hundreds of millions of atoms. This approach has proven to be highly effective. For example, generating and displaying an image generated from a simulation of 100 million atoms on a 512-processor CM-5 takes approximately 20 s over a standard Internet connection. In contrast, generating such an image locally would require transferring the dataset, loading it into a high-performance workstation, and rendering it—a process that could take several hours to complete for a single dataset. Although the system does not attempt to be highly interactive, we have found it to be completely usable on virtually any size of dataset and from any kind of workstation.

Network programming and remote data analysis

The explosive growth of the Internet has brought significant interest in adding network capabilities to scientific applications. By using scripting languages, we can provide these capabilities for free. All the major scripting languages are used heavily in Internet-based applications and provide built-in support for a variety of network-programming protocols. To illustrate the networking capabilities, we can write a very simple Python loop to run a molecular-dynamics simulation and poll a socket for connections from a World Wide Web client. When a connection is received, we simply send a simple status message back to the client.

```
from socket import *
from select import *
sock = socket(AF_INET,SOCK_STREAM)
...
def run(nsteps):
    for i in range(0,nsteps):        # Do the MD step
        integrate_adv_coord(Dt)
        boundary_periodic()
        ...
        #Look for an incoming connection
        (r,w,e) = select([sock],[],[],0)
        for s in r:
```

```
        client = s.accept()[0]
        header = client.recv(2048)
        print 'Received a connection'
        ...
        client.send("HTTP/1.0 200 \n")
        client.send("Content-type:
text/html\n\n")
        client.send("I'm now at timestep " +
str(i) + "\n")
        ... send additional data ...
        client.close()
...
```

Now we can start a long-running simulation and check on its progress by simply running a Web browser and connecting to the appropriate port on the machine. When coupled with the visualization system, the physics code is able to serve images and other content on demand. Surprisingly, this is easily accomplished even if there is no conventional Web server running on the machine—the physics code itself is acting a simple Web server. While such a scheme could have been written in C, using a scripting language only required one half day of effort. Figure 2 shows the result of a typical Web-browser query for a running molecular-dynamics simulation.

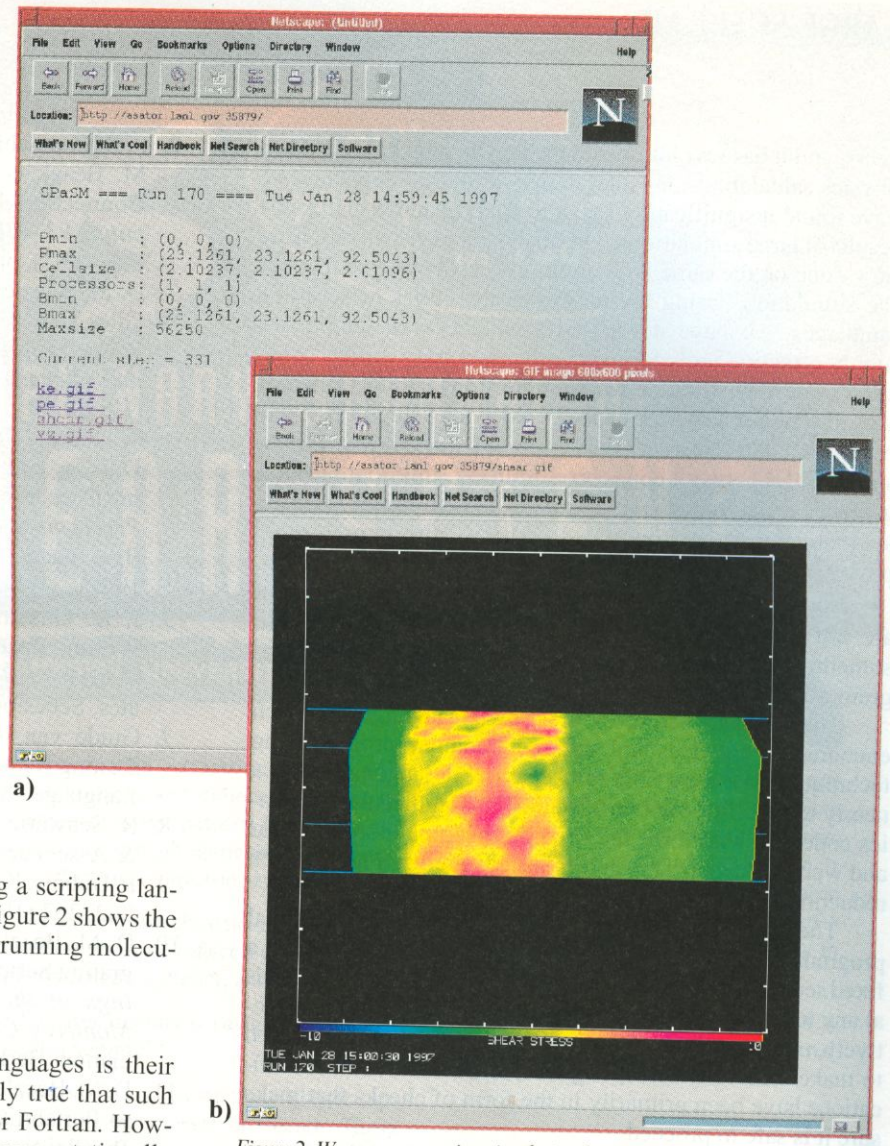


Figure 2. We query a running simulation from a Web browser. (a) The user is presented with a summary page indicating the status of the current simulation. (b) A detailed view shows the shear stress generated behind a propagating shock wave in an fcc crystal.

Performance concerns

One concern about using scripting languages is their impact on overall performance. It is certainly true that such languages are significantly slower than C or Fortran. However, it is important to emphasize that all the computationally demanding functionality in our system is still implemented in C. The scripting language provides structure and organization, and its use incurs little performance penalty, as confirmed by the timings shown in the table, which were taken from a recent simulation. The timings are for a simulation written entirely in C and for the same C simulation but with the outer loop written in Python instead. The observed 0.2% performance penalty is of little concern.

Despite the potential for some performance penalties, we feel that the scripting-language approach is sound and has allowed us to solve problems faster than before. No matter what computational improvements are made, time to solution is the most important measure, and high-level scripting languages have reduced this time significantly.

Table. Wrapping a C program in a Python outer loop adds convenience, organization, and structure at little performance cost.

	Execution time (s)	
Simulated atoms per processor	C	C with Python
13950	98.7	98.9
45000	314.1	314.8
180000	1317.1	1319.0

Grand Unification Theory?

The adoption of common scripting languages as the interface to physics codes brings a huge variety of benefits. The procedure provides a flexible user interface for controlling complex simulations. Programmers can access built-in modules and system services of a particular scripting language. These include sockets, file handling, and more. In addition, there are third-party modules and external packages for image processing and database access. Most scripting languages also provide toolkits such as Tk for graphical-user-interface development. In short, what starts out as a simple physics code can be transformed into a software package possessing a substantial level of functionality—most of which comes for free. Throughout this process, the integrity of the physics code is maintained; it remains essentially unchanged and simple to manage. We believe that this is a computational model for physics research that is highly effective and works.

A revolutionary improvement

We have been using this approach for approximately two

years, and it has revolutionized the way in which we perform physics calculations. By integrating various components, we have found it significantly easier to analyze and visualize the results of large simulations. The analysis and visualization are now done on the same large multiprocessor systems that run the simulation. Using the network capabilities of scripting languages, we have developed a collection of simple yet highly effective tools for examining data from ordinary machines and over slow network connections. (In fact, there is very little overall performance penalty for using the system over a standard Internet connection rather than over a high-speed FDDI network connection.) Finally, this approach has given us a flexibility: The system is easy to modify, allows sophisticated debugging, and provides the expressional power that computational physicists demand.

This approach has also provided a number of benefits that are not immediately obvious. Our code has evolved into something more modular, reliable, and smaller, and the program now functions in an event-driven fashion.

The modular nature of a scripting-language approach encourages, but does not enforce, modular programming techniques. We were able to use scripting languages with a nearly unmodified version of our original molecular-dynamics code, but with time, the code has become more modular and well-packaged. This in turn has resulted in a nearly 25% reduction in code size.

The use of a scripting language has turned our sequential program into an event-driven program. Rather than having a fixed sequence of operations, users now may issue commands at any time and in any order. To manage this behavior effectively, many portions of the code have been slowly modified to make it more reliable, bug-free, and stable; these modifications have been primarily in the form of checks that make sure it is safe to proceed.

Finally, we would like to emphasize the point that all these features have not come at the expense of added complexity. In fact, the code can still be compiled with a main program and no user interface, just as was the case before we added the scripting-language interface.

Acknowledgments

We would like to acknowledge our collaborators, Shujia Zhou, Brad Holian, and Niels Grønbech Jensen of Los Alamos National Laboratory, Tim Germann at the University of California at Berkeley, Pablo Tamayo from Thinking Machines Corp., and Bill Kerr at Wake Forest University. We would also like to acknowledge Paul Dubois and Brian Yang at Lawrence Livermore National Laboratory, the Scientific Computing and Imaging Group at the University of Utah, and the Cornell Theory Center for many interesting discussions and support. Development of the SPaSM code has been under the auspices of the United States Department of Energy. Many of the computations have been performed on the CM-5 and Cray T3D in the Advanced Computing Laboratory at Los Alamos National Laboratory.

References

1. D. M. Beazley and P. S. Lomdahl, Parallel Computing **20**, 173–195 (1994).
2. P. S. Lomdahl, P. Tamayo, N. Grønbech-Jensen, and D. M. Beazley, "50 Gflops Molecular Dynamics on the Connection Machine 5," *Proceedings of Supercomputing 93* (IEEE Computer Society Press, Los Alamitos, CA, 1993), pp. 520–527.
3. D. M. Beazley, P. S. Lomdahl, N. Grønbech-Jensen, R. Giles, and P. Tamayo, "Parallel Algorithms for Short-Range Molecular Dynamics," in World Scientific's *Annual Reviews in Computational Physics* **3**, 119–175 (1996).
4. P. Dubois, Comput. Phys. **10**, 359 (1996); T. Yang, P. Dubois, and 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).
5. J. K. Ousterhout, *Tcl and the Tk Toolkit* (Addison-Wesley, Reading, MA, 1994).
6. Mark Lutz, *Programming Python* (O'Reilly & Associates, Sebastopol, CA, 1996).
7. Guido van Rossum and Jelke de Boer, "Interactively Testing Remote Servers Using the Python Programming Language," CWI Quarterly **4**, 283–303 (1991).
8. R. Schwartz and L. Wall, *Programming Perl* (O'Reilly & Associates, Sebastopol, CA, 1994).
9. P. Dubois, K. Hinsen, and J. Hugunin, Comput. Phys. **10**, 262–267 (1996).
10. 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, Berkeley, CA 1996), pp. 129–139.
11. D. M. Beazley and P. S. Lomdahl, "A Practical Approach to Portability and Performance Problems on Massively Parallel Supercomputers," in *Proceedings of the Workshop on Debugging and Tuning for Parallel Computer Systems, Chatham, MA, 1994* (IEEE Computer Society Press, Los Alamitos, CA, 1996), pp. 337–351.
12. P. S. Lomdahl and D. M. Beazley, "Multi-Million Particle Molecular Dynamics on MPPs," Second International Workshop, PARA '95 (Lyngby, Denmark, August 1995), in *Lecture Notes in Computer Science* **1041**, edited by J. Dongarra *et al.* (Springer-Verlag, New York, 1996), pp. 391–407.
13. D. M. Beazley and P. S. Lomdahl, "Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations," *Proceedings of Supercomputing '96* (CD-ROM, IEEE Computer Society Press, Los Alamitos, CA, 1996).
14. D. M. Beazley and P. S. Lomdahl, "Extensible Message Passing Application Development and Debugging with Python," to appear in proceedings of IPPS'97, Geneva, Switzerland, LA-UR-96-3386.
15. J. Nordiek, M. Moseler, and H. Haberland, "Energetic Impacts of Cu Clusters on Cu Surfaces," in *Proceedings of COSIRES '96*, edited by R. Webb, to appear in J. Rad. Effect and Defects in Solids (1997).
16. S. J. Zhou, D. M. Beazley, P. S. Lomdahl, and B. L. Holian, Phys. Rev. Lett. **78**, 479 (1997).