Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations

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

Peter S. Lomdahl
Theoretical Division
Los Alamos National Laboratory
Los Alamos, New Mexico 87545
pxl@lanl.gov
http://bifrost.lanl.gov/~pxl

Abstract:

We present a computational steering approach for controlling, analyzing, and visualizing very large scale molecular dynamics simulations involving tens to hundreds of millions of atoms. Our approach relies on extensible scripting languages and an easy to use tool for building extensions and modules. The system is easy to modify, works with existing C code, is memory efficient, and can be used from inexpensive workstations over standard Internet connections. We demonstrate how we have been able to explore data from production MD simulations involving as many as 104 million atoms running on the CM-5 and Cray T3D. We also show how this approach can be used to integrate common scripting languages (including Python, Tcl/Tk, and Perl), simulation code, user extensions, and commercial data analysis packages.

Keywords:
Molecular dynamics, data analysis, large-scale simulation, steering, scripting languages, visualization, parallel computing, SPaSM, SWIG

Introduction

With the development of massively parallel supercomputers, the materials science community has experienced an unprecedented explosion in both the size and complexity of short-range molecular dynamics simulations [1,2,3,4,5]. The method of molecular dynamics (MD) has been used since the 1950's for a variety of computational problems in physics, chemistry, and materials science [6]. The idea is really quite simple-- given a collection of atoms, we solve Newton's equation of motion $F=ma$ and track the atoms' trajectories. The physics of the simulation is incorporated into the force law in the form of a potential energy function. This function can range from a simple pair-potential to complicated many-body potentials [7] .

Prior to 1992, MD simulations were usually performed on relatively small systems (often in 2D) involving fewer than 1 million atoms [8]. However, in a span of only 3 years, MD simulation sizes grew to as many as 1 billion atoms in 3D [2,4,5]. Today, production simulations involving tens to hundreds of

millions of atoms are possible, but analyzing the resulting data has proven to be extraordinarily difficult. As a result, most MD simulations remain small even though many researchers agree that large-scale simulations are useful for studying certain types of material properties.

In this paper, we describe our efforts in addressing the practical problems of working with very large molecular dynamics simulations. We have taken a computational steering approach in which simulation, data analysis, and visualization are combined into a single package [9,10,11,12]. However, we have adopted an approach that is extremely lightweight--that is, it is memory efficient, simple to use, portable, easily extensible, and not dependent on expensive special purpose hardware. (ie. graphics workstations or high-speed networking).

The SPaSM Code

Since 1992, we have been developing a code, SPaSM (Scalable Parallel Short-range Molecular Dynamics), for performing short-range molecular dynamics simulations on massively parallel supercomputing systems at Los Alamos National Laboratory [1]. Our primary goal has been to study 3D fracture, dislocations, ductile-brittle transition, and other material properties. The code was originally developed for the Connection Machine 5 and is written entirely in ANSI C with message passing. In 1992, the SPaSM code was able to simulate more than 100 million (10^8) particles in both 2D and 3D [1]. In 1993, optimization work resulted in SPaSM receiving part of the 1993 IEEE Gordon Bell Prize [2]. Since then, the SPaSM code has been ported to a variety of other parallel machines. A memory optimization also allowed us to double our simulation sizes up to as many as 300 million atoms in double precision (on a 1024 node CM-5 with 32 Gbytes of memory). Currently, SPaSM is implemented on top of a collection of wrapper functions for both message-passing and parallel I/O [13]. This allows the code to run on a variety of machines including the CM-5, Cray T3D, Fujitsu VPP-550, SGI Power Challenge, Sun multiprocessor servers, and single processor workstations.

While the performance of the SPaSM code has been discussed extensively elsewhere, Table 1 shows some recent performance results of the SPaSM code on several different machines [1,2,13]. The table is provided primarily to illustrate the simulation sizes that are possible as well as the computational requirements for such simulations.

Number atoms	CM-5 (1024 nodes)	T3D (128 nodes)	Power Challenge (8 nodes)
1,000,000	0.39	0.728	8.68
5,000,000	1.60	3.86	40.43
10,000,000	2.98	6.93	80.96
32,000,000	-	-	275.60
50,000,000	14.20	33.09	-
75,000,000	-	46.95	-
150,000,000	41.26	-	-
300,800,000	90.59	-	-
600,000,000	241.73 (SP)	-	-

Table 1 : Time for a single MD timestep (in seconds). Atoms interact according to a Lennard-Jones potential and have been arranged in an an FCC lattice with a reduced temperature of 0.72 and density of 0.8442 [4]. The cutoff is 2.5 sigma. All runs performed in double precision except for (SP).

The Data Glut (and previous work)

One of our original goals was to perform large 3D molecular dynamics simulations of materials. Figure 1 shows snapshots from two such simulations.

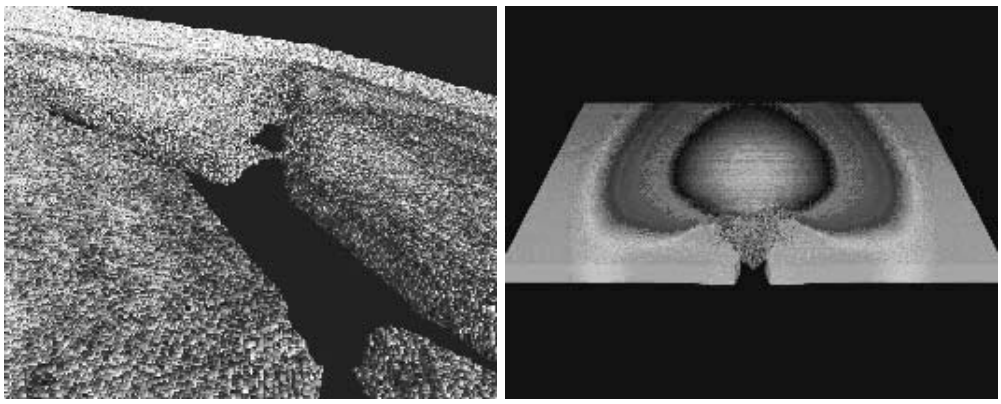

Figure 1 : Fracture experiments with 38 million particles (left) and 104 million particles (right).

The images are somewhat misleading. Performing such simulations in practice has proven to be very difficult due to the enormous amount of data that must be analyzed. For example, a single snapshot file

from the 38 million atom simulation in Figure 1 was larger than the total system memory available on the largest SGI Onyx workstation at LANL available at the time (in fact, the single image shown required more than 3.5 hours of rendering time). The 104 million particle simulation generated a collection of 40 1.6 Gbyte datafiles (containing only particle positions and kinetic energies stored in single precision). None of SPaSM's users have workstations capable of handing such datasets. In fact, most users don't even have enough local disk space to bring one of these datasets to their local machine. Even if this were possible, users at remote sites would be out of luck--shipping 64 Gbytes of data across the Internet would almost certainly be a nightmare. Unfortunately, the idea of simply offloading data from a supercomputer onto a high-end graphics workstation still seems to run rampant within the supercomputing community. However, in practice this only really works for relatively small simulations involving no more than a few million datapoints. Of course, this really should not come as a surprise, why would any reasonable person expect a workstation to be able to efficiently handle the analysis of data from a very large simulation performed on a 512 processor Connection Machine or T3D?

The large dataset problem has not gone unnoticed by the supercomputing community and some have come to call it the "Data Glut" problem [14]. To visualize some of our very large simulations, we have sometimes used a parallel rendering tool [15]. This is how the 104 million atom picture in Figure 1 was generated (on a 128 processor Cray T3D). Unfortunately, this approach has also proven to be ineffective. Few users know how to run the rendering code, and the system requires several minutes to make a single picture (making the system difficult to use for data exploration). More problematic is the fact that the tool was extremely limited in its data analysis capabilities--a tool which only makes millions of rendered spheres can hardly be called a data analysis package! We really needed a system that could perform visualization coupled with data analysis and feature extraction. We also needed a system that was easy to use and which could be used from our existing workstations.

Unfortunately, it seems that many solutions to the large-dataset problem have become more impractical in recent years. Some have even predicted the "end to batch-processing." [11] The truth of the matter is that large-scale computing is still difficult, still takes hundreds of hours of computing time on the fastest machines available, and still generates an overwhelming amount of data. Some seem to believe that the data-glut problem can be magically eliminated with very expensive special purpose hardware [10,11,12,16]. While this may be possible, our experience in the I-WAY at Supercomputing'95 was a complete disaster (unless you consider showing a pre-rendered MPEG movie of a MD simulation to a half dozen people a success) [17]. Even if the real demo had worked, most scientists we know do not have the resources to go buy a CAVE, a personal Power-Challenge Array, a wall-sized display, and a dedicated OC3 connection to their favorite supercomputing center just to look at their data [17]. Thus, while these efforts may be conceptually interesting, we feel they are of little practical value to scientists who are remotely accessing supercomputing facilities from an ordinary UNIX workstation.

Computational Steering

In reality, we feel that most of our data analysis problems are not the result of having large datasets, a lack of tools, or lack of high performance machines, but the entire methodology of the process. By decoupling the simulation and the analysis, we are left with the problem of figuring out how to move data around between tools and machines. Often times, this data is incomplete--and data analysis tools are unable to extract physically interesting quantities. It is also frustrating to run a large simulation only to find out during post-processing that it was flawed.

To address these problems, we have adopted the approach of "computational steering" which aims to

combine simulation, data analysis, and visualization into a single package. We feel that this combination is important because trying to understand very-large MD simulations is more than just a simulation problem, an analysis problem, a visualization problem, or a user-interface problem. It is a combination of all of these things---and the best solution will be achieved when these elements have been combined in a balanced manner.

In order to build an effective system for large simulations, we feel that it must support the following features.

- Interactivity. The user must be able to interact with the data and extract useful information.
- Scripting. The system must be able to support simulations that run for hundreds of hours without user intervention (just because you have steering doesn't mean you don't want to run batch jobs).
- Memory efficiency. Large-scale MD is still the primary goal, not user interfaces or graphics.
- Network efficiency. The system must be usable over standard Internet connections.
- Extensibility. The user must be able to extend the code with new features and new functionality.
- Compatibility. The steering system should be able to work with existing code.
- Portability. The system must work on all parallel machines and workstations.
- Performance. It is critical that the system be able to process massive amounts of data quickly. It must also be usable from any UNIX workstation.
- Ease of use. Keep it simple and the scientists will use it.

Previous efforts in computational steering seem to have focused primarily on interactivity and user interfaces while ignoring many of the issues important for large-scale simulation [9,10,11,15]. As a result, we end up with scientific computing "environments" that are unnecessarily complicated and which continue to rely on high-end graphics workstations and high-bandwidth networks--a solution that is simply not that attractive at this time due to both the high cost and limited performance on large-scale problems.

Our approach incorporates all of the features listed above, but with an emphasis on working with very large-simulations and simplicity at all levels. We're not interested in building a large monolithic steering system---in fact, we feel that the best steering "system" is one that you barely notice that you're using! Thus, we will primarily focus on issues related to memory efficiency, long running jobs, extensibility, and building steerable applications from existing code. We do not see our approach as a replacement for other efforts in computational steering. Rather, we see it as an "alternative world view" in which we have tried to build a balanced system that is well-suited for large-scale production simulation and basic scientific research. While most of our efforts have focused on MD, we feel that our approach is applicable to many other large-scale computing situations.

Lightweight Steering with Scripting Languages

Rather than relying on a sophisticated graphical user interface, we have taken a command-driven approach that has been used successfully in commercial packages such as MATLAB, IDL, or Mathematica. While this may not be "flashy", it is remarkably simple and expressive. A user can interactively issue commands to generate plots, run the simulation, or look for interesting features. It is also possible to write scripts for controlling long-running jobs or to write some functionality as procedures in the command language itself. To add this functionality to the SPaSM code, we can use extensible scripting languages such as Tcl/Tk and Python [18,19]. As a result, the overall code organization is as shown in Figure 2.

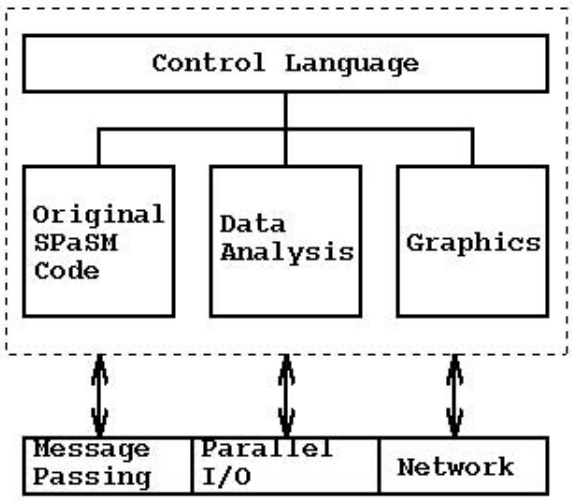

Figure 2 : SPaSM Organization

In the new system, the control language is used to glue together different modules for simulation, data analysis and visualization while the entire system is built on top of a message passing, parallel I/O and networking layer that hides hardware dependent implementation details.

On parallel machines, we originally developed our own scripting language based on a simple YACC parser [20]. This language allows the user to access C functions and variables, but also allows loops, conditionals, user-defined functions, and variables to be created on the fly. In reality, the scripting language is not unlike Tcl/Tk, except that we have written the system to work with parallel I/O and have cleaned up the syntax. Internally, the scripting language uses a SPMD style of programming. Each node executes the same sequences of commands, but on different sets of data. The nodes are only loosely synchronized and may participate in message passing operations. While our scripting language was primarily designed to run nicely with message-passing, it turns out that SPaSM can be compiled to use Tcl, Python, or Perl as well (see the next section).

We would like to emphasize the efficiency and simplicity of this approach. Adding a scripting language requires very little memory since parser really only needs a small stack for the LALR(1) parsing method used by YACC [20]. As a result, there is little impact on memory usage. Scripting languages are also easily portable and don't require much network bandwidth to operate. Finally, by having a carefully chosen set of commands and defaults, it is easy to perform complex analysis and visualization without making things too complicated.

Automated Language-Independent Interface Generation

While our command driven approach to steering is conceptually simple, in order to be useful, it is critical that a user be able to add new commands and features. In this case, one must extend the command-language interface with new C functions and variables. The standard technique for doing this is to write special "wrapper" functions that provide the glue code between the command-language and underlying C functions. Unfortunately, writing these wrapper functions is usually quite technical, tedious, poorly documented, and error-prone. Most users do not want to spend time figuring out how to

extend the user-interface with new functions. Therefore, we have built an automated interface generator called SWIG (Simplified Wrapper and Interface Generator)[1] that can be used to build the entire user interface from ANSI C function and variable declarations [21]. Using SWIG, a typical user-interface specification looks like the following :

```
%module user
%{
#include "SPaSM.h"
%}

extern void     ic_crack(int lx, int ly, int lz, int lc,
                         double gapx, double gapy, double gapz,
                         double alpha, double cutoff);

/* Boundary conditions */

extern void     set_boundary_periodic();
extern void     set_boundary_free();
extern void     set_boundary_expand();
extern void     apply_strain(double ex, double ey, double ez);
extern void     set_initial_strain(double ex, double ey, double ez);
extern void     set_strainrate(double exdot0, double eydot0, double ezdot0);
extern void     apply_strain_boundary(double ex, double ey, double ez);
```

Code 1 : A SPaSM user interface file. This file is automatically translated into C wrapper functions and can be combined with other modules at compile time.

In order to expand the system, the user writes a normal C function--exactly as would have been done without the command language interface. Its ANSI C prototype declaration is then placed into an interface file. When SPaSM is compiled, the interface file is automatically translated into C wrapper functions and a new command is created with the same usage as the underlying C function. This match between commands and C functions is important because typical users of SPaSM are writing C code to implement new physical models and initial conditions. Accessing these functions from the command interface is critical. Our automated tool provides a direct mapping between the command interface and the underlying C functions which is easily understandable. Since our interface file specifications are not specific to any one scripting language, SWIG has been designed to support multiple target languages and can currently build interfaces for Tcl, Python, Perl4, Perl5, Guile, and our own scripting language. Thus, SPaSM can be controlled by any of these languages (although obviously not all scripting languages will work properly on parallel machines).

Building Modules and Packages

To encourage code reuse and provide additional flexibility, SWIG allows SPaSM users to easily build packages of interesting modules with a special "include" directive. Thus, a more complex user interface could be built as follows :

```
%module user
%{
#include "SPaSM.h"
%}

%include initcond.i
%include graphics.i
```

```
%include dislocations.i
%include particle.i
%include debug.i
```

Code 2 : SPaSM interface file with modules.

Many files could be placed in a common repository of modules available to all users, but others can be written or customized by the user as needed for a particular simulation. Thus, instead of forcing every user to use the same system, this approach allows each user to customize SPaSM to their individual liking. We feel that this flexibility is critical--especially in an environment where the code is in a constant state of evolution as new physical models and simulations are being developed.

Pointers and Objects

While our interface builder is designed to be easy to use, it can be used with highly complex C and C++ code. Pointers to arrays, structures, and classes can also be manipulated [21]. Thus, a user can create vectors, particles, arrays, and other interesting objects all from within the scripting language interface. These in turn can be passed to other C functions for later use. For example, suppose we wanted to make a module for finding small subsets of atoms by culling the particle data based on the value of its individual potential energy contribution (a useful technique we have used for finding dislocations). We could do this entirely within a SWIG interface file, by including a special C function for searching the particle data. In this case, we've written a function that looks for the first particle matching the search range and returns a pointer to it. The function can be called repeatedly with the previously returned pointer value to find all of the matching particles.

```
// cull.i.  SPaSM interface file for particle culling
%{
Particle *cull_pe(Particle *ptr, double pmin, double pmax) {
    if (!ptr) ptr = Cells[0][0][0].ptr - 1;
    while ((++ptr)->type >= 0) {
        if ((ptr->pe >= pmin) && (ptr->pe <= pmax))
            return ptr;
    }
    return NULL;
}
%}

Particle *cull_pe(Particle *ptr, double pmin, double pmax);
```

Code 3 : Simple interface file for culling particles. Note that small C functions can be inlined directly into interface files.

Within a scripting language, we can now write some functions to build and manipulate lists of particles. In this case, we have built SPaSM under the Python scripting language [19].

```
# Return a list of all particles with pe in [min,max]
def get_pe(min,max):
      plist = [];
      p = spasm.cull_pe("NULL",min,max)
      while p != "NULL" :
         plist.append(p)
         p = spasm.cull_pe(p,min,max)
      return plist
```

```
# Make an image from particles in a list
def plot_particles(l):
      spasm.clearimage();
      for i in range(0,len(l)):
        spasm.sphere(l[i]);
      spasm.display();
```

Code 4: Python functions for extracting particle data and plotting.

One of the greatest strengths of extensible scripting languages is their ability to easily manipulate structures such as lists and associative arrays. If we wanted to extract two sets of particles with different potential energy ranges and make an image, the user could simply type the following Python commands:

```
>>> list1 = get_pe(-5.5,-5);
>>> list2 = get_pe(-3.5,-3.25);
>>> plot_particles(list1+list2);
```

The ability to work with complex objects has been critical for building more complex modules. While a user may only write a relatively simple interface file for their own C functions, other modules may be quite sophisticated--involving large libraries or C++ code. For example, we have used SWIG to build modules out of MATLAB and the entire Open-GL library--both of which can be imported into the SPaSM code if desired. In short, almost any type of C code can be integrated into our steering system (although clearly not all codes will work on all machines).

A Scripting Example

Some people might not think of scripting as a component of steering, but we feel that it is absolutely critical. Simulations can run for a very long time where user interaction isn't necessary. Code 5 shows a typical SPaSM script.

```
#
# Script for strain-rate experiment
#
printlog("Crack experiment.");

# Set up a morse potential

alpha = 7;
cutoff = 1.7;
init_table_pair();
source("Examples/morse.script");
makemorse(alpha,cutoff,1000);    # Create a morse lookup table

# Set up initial condition

if (Restart == 0)
   ic_crack(80,40,10,20,5,25.0,5.0, alpha, cutoff);
   set_initial_strain(0,0.017,0);
endif;

# Now set up the boundary conditions

set_strainrate(0,0,0.001);
set_boundary_expand();
```

```
output_addtype("pe");

# Run it
timesteps(1000,10,50,100);
```

Code 5 : A sample SPaSM script file. Commands can also be typed interactively.

In the script, the commands directly map onto the underlying C functions given in the interface file. Scripting also provides a rapid prototyping capability since users can make changes to simulation parameters without recompiling the SPaSM code after every change. Many operations can be first implemented as scripts and recoded in C after they have been sufficiently tested.

An Interactive SPaSM Example

We have developed a high-performance memory efficient graphics module that allows us to remotely visualize MD data with as many as 100 million atoms on a 512 processor CM-5 (this work is in progress and will be published elsewhere). The graphics system is implemented as a separate module and can be used to visualize data from running simulations or post-processing. The following example shows how this system is used on data from a 11 million particle impact simulation. Each data set is 180 Mbytes. Images are sent through a socket connection as GIF files to the user's workstation for display. The view is controlled by typing commands directly into the SPaSM code as shown in the following example. This example was run on a 64 processor node CM-5 (and has been edited slightly for clarity). Text typed by the user is shown in bold.

```
Run30 === cm5-4 === Sun Apr 28 10:22:23 1996
SPaSM [30] > open_socket("tjaze",34442);
Connecting...
Socket connection opened with host tjaze port 34442
SPaSM [30] > imagesize(512,512);
Image size set to 512 x 512
SPaSM [30] > colormap("cm15");
Colormap read from file cm15
SPaSM [30] > FilePath="/sda/sda1/beazley/backup/backup";
SPaSM [30] > readdat("Dat36.1");
Setting output buffer to 524288 bytes
Reading 11203040 particles.
11203040 particles { x y z ke } read from /sda/sda1/beazley/backup/backup/Dat36
SPaSM [30] > range("ke",0,15);
ke range set to (0,15)
SPaSM [30] > image();
Image generation time : 10.1531 seconds
SPaSM [30] > rotu(70);
Image generation time : 10.7456 seconds
SPaSM [30] > rotr(40);
Image generation time : 10.9436 seconds
SPaSM [30] > down(15);
Image generation time : 10.5469 seconds
SPaSM [30] > Spheres=1;
SPaSM [30] > zoom(400);
Image generation time : 19.8765 seconds
SPaSM [30] > clipx(48,52);
Image generation time : 7.29181 seconds
SPaSM [30] >
```

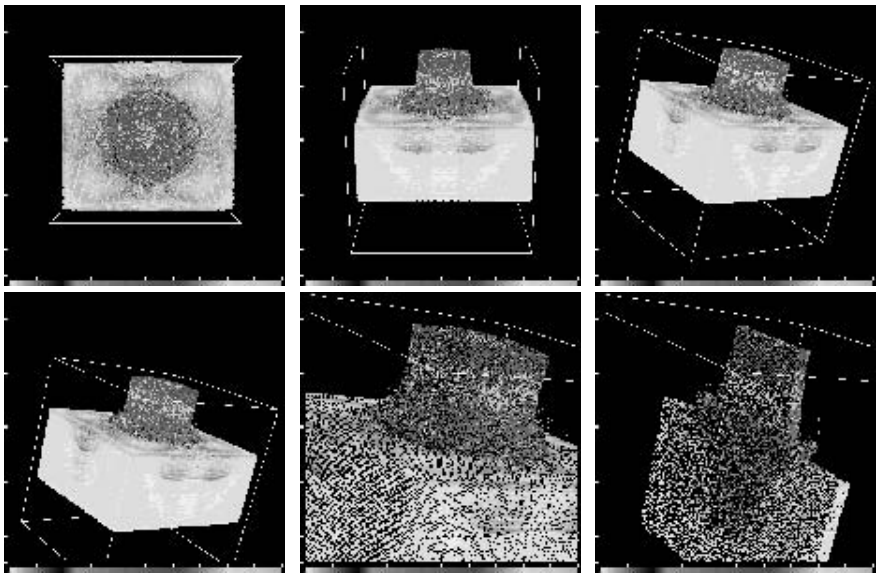

Figure 3 : Six images generated by the interactive example (in order from left to right).

With a careful choice of parameters, it is easy to move around in a data set and look at interesting features. Previously defined viewpoints can also be easily saved and recalled. When we tried to work with this same dataset on an SGI Onyx graphics workstation with 256 Mbytes of RAM, it was virtually impossible. Images required as many as 45 minutes to generate and the machine was simple incapable of dealing with a dataset of this size in an interactive manner [15]. However, by using our new system, it is possible to visualize large simulations in less time than that required to perform a single MD timestep (see Table 1).

While this example has been simple, this approach can also be used to interactively set up initial conditions, visualize the data, run the simulation, and perform analysis in real-time as the simulation runs. Periodically, the user can stop the simulation, look at the data in more detail, make changes to various parameters, and continue the simulation. All of this is possible without exiting the SPaSM code or loading a separate analysis tool.

Data Exploration and Feature Extraction

One of the problems of 3D materials simulations is that we may only be interested in a small subset of the data such as dislocations or defects. Often, a feature of interest is embedded within a large-bulk of uninteresting atoms. Using SPaSM, we are able to explore very large datasets and look for interesting features. In Figure 4a, we see dislocation loops generated inside a block of 35 million copper atoms (interacting via an embedded-atom potential). This simulation ran for 120 hours on a 128 processor Cray T3D and produced 35 Gbytes of output. In Figure 4b, we are looking at damage due to ion-implantation in a 5 million atom silicon crystal. In both cases we were able to explore and visualize the data by running the SPaSM code remotely and displaying images on a local workstation.

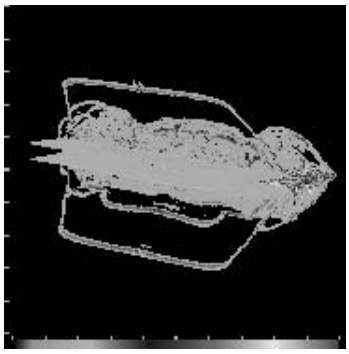

Figure 4a. Dislocation loops in 35 million atom fracture simulation. (700 Mbytes). (Click on each image for an MPEG movie)

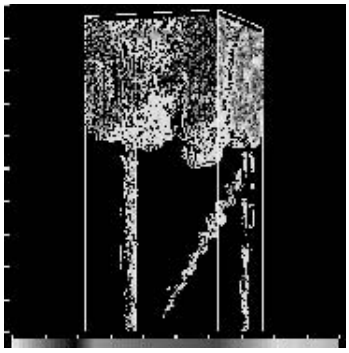

Figure 4b. Ion-implantation in 5 million atom silicon crystal. (100 Mbytes)

In practice, identifying interesting features in large-scale MD simulations is a hit and miss process that depends on a variety of simulation parameters. This is one of the primary reasons why it is so difficult to work with these datasets on a workstation--we must first work with all of the data to effectively identify interesting features. Interestingly enough, by being able to remotely explore large datasets, it is often possible to reduce the datasets to a size than can be later handled on a workstation. For example, in Figure 4a, a single snapshot file is approximately 700 Mbytes, but by removing the bulk, this can be reduced to only 10-20 Mbytes---a size that is more easily handled. The trick is figuring out which 20 Mbytes of data is interesting!

Debugging, Prototyping, and Development

Due to the portability of the SPaSM code, it is possible to do code development and debugging on ordinary workstations. With SWIG, we are able to build SPaSM under a number of popular scripting languages and to import packages such as MATLAB as a SPaSM module. Figure 5 shows a screen shot from such a simulation in which a small MD shock-wave problem is being run on a single processor Unix workstation. The simulation itself is being controlled by a Tcl interpreter, while visualization is being performed by MATLAB and our built-in graphics module. Both plots shown are updated in real-time as the simulation runs. We also see a simple GUI written in Tcl/Tk for performing various visualization shortcuts, changing settings, and loading datafiles for post-processing.

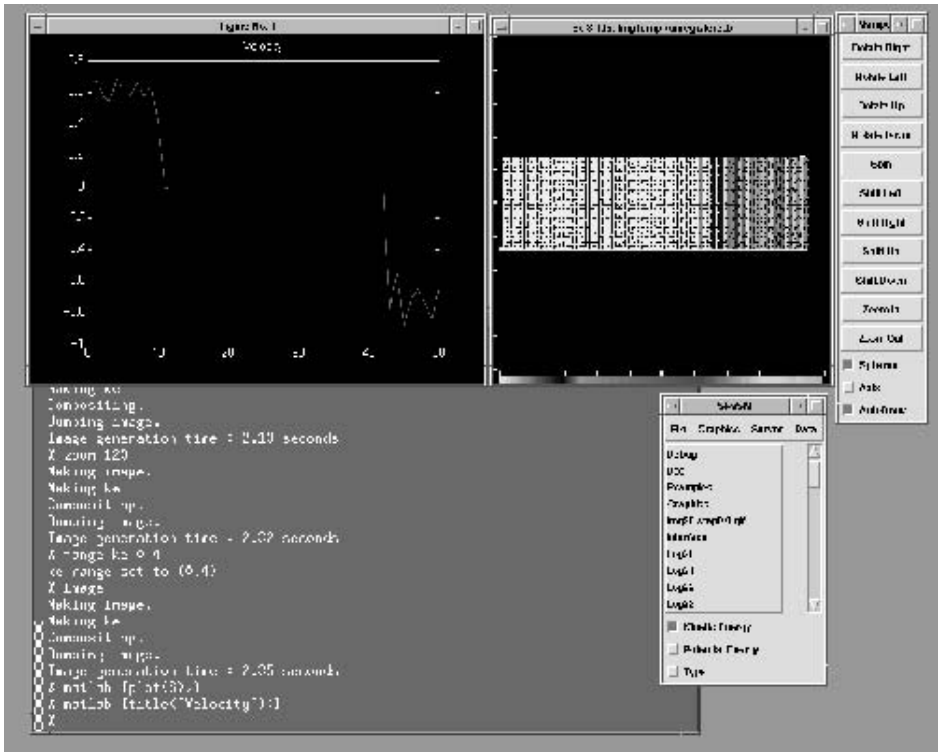

Figure 5: SPaSM/MATLAB running on a workstation under Tcl/Tk. (Click on image for full-size view).

It is important to emphasize that everything shown in the image has been combined into a single package using our automatic interface generator, yet the SPaSM code is unchanged from the version run on the CM-5 or Cray T3D. Thus, anything added to the SPaSM core in this environment can also be used on those machines. In practice, we have found that developing on a workstation is considerably easier, more reliable, and less frustrating than trying to develop everything on a parallel machine. By having a highly flexible system, we can utilize a wide variety of analysis tools during the development phase and move up to larger, less interactive, production simulations when we are ready.

Concerns about Steering and Supercomputing Usage

Our approach to steering has resulted in visualization and data analysis tasks being performed directly on the same parallel machines in which we perform simulation. We feel that this is absolutely necessary, because there simply aren't any other machines available that can effectively deal with the large datasets. Unfortunately, this is also troublesome. Most supercomputing systems are used for batch processing and number crunching, not interactive visualization and analysis. Furthermore, many supercomputing centers charge for time, in which case interactively typing commands into a job running on a 512 processor machine may, in fact, be quite expensive!

While it is true that we sometimes need to run interactively on large numbers of processors, a number of steps can be taken to reduce the amount of time required for analysis and visualization :

- Image and data analysis parameters can be saved for later use. If a good set of analysis procedures

are developed initially, they can be reused later with a great savings in time.

- Our code supports batch processing of data files. By loading a representative datafile, it is often possible to pick good visualization and analysis parameters. Once set, a single command can be used to process an entire sequence of datafiles without user intervention.
- If the analysis procedures are known in advance, visualization and analysis can be performed as the simulation runs. In a batch processing mode, such a simulation will not only produce a collection of datafiles, but a collection of GIF images, and other supporting output. This usually results in extremely high performance since the analysis is performed on the fly as opposed to having to reload the resulting datafiles and performing the analysis at a later time (although this is certainly still possible).

Conclusions and Future Work

We have presented a simple steering approach based on scripting languages and an automated interface building tool, SWIG, that we have developed for building extensions and modules. This approach has been used with the SPaSM code for about one year. We feel that this has been a step in the right direction, but want to emphasize the differences between our approach and others. First, rather than trying to build a sophisticated computational steering "environment" and integrating simulation codes into it, we have developed an extremely lightweight tool that works with our existing codes, but tries to stay out of the way. Secondly, our system has been designed primarily to support very-large scale simulation. While we can run interactively, the system works equally well in a batch processing mode for long-running simulations. The memory efficiency of the approach has allowed us to continue running very large simulations, even when performing analysis and visualization. Our approach also works well on inexpensive workstations and slow networks--making it useful to users who only have remote access to a large parallel supercomputer. Finally, we recognize that most scientists who would be using SPaSM are qualified professionals who demand both simplicity and functionality. Our approach works with existing C functions and is easy to extend. At the same, we impose no restrictions on the scripting language or types of C code that can be glued together. Thus, a scientist can easily build a simple implementation or one of almost arbitrary complexity.

Currently, we are working on the development of new graphics and data analysis modules for SPaSM. We are also interested in extending our work with the Python scripting language and exploring extensions such as Numerical Python which provide high-level mathematical operations on arrays and matrices [22]. We have no plans to build a sophisticated graphical user interface at this time. If this is desired, we feel that we could probably use any number of existing systems such as the SCIRun steering software developed at the University of Utah [9]. Finally, as data analysis and visualization become commonplace, we feel that data management and organization of results will be critical. Therefore we are quite interested in extending some of our work to scientific databases and data management systems. We feel that this management of data, run parameters, and output, will be more critical than simply providing more interactivity.

Acknowledgments

This work would not be possible without the support and input of many people. We would like to acknowledge our collaborators in MD work, Brad Holian, Shujia Zhou, Tim Germann, and Niels Jensen. Mike Krogh, Chuck Hansen, Jamie Painter, and Curt Canada of the Advanced Computing Laboratory have provided valuable assistance. We would also like to acknowledge Chris Johnson and the Scientific

Computing and Imaging group at the University of Utah. Finally, we would like to acknowledge the Advanced Computing Laboratory for their generous support. This work was performed under the auspices of the United States Department of Energy.

Footnotes

[1] The SWIG interface building tool is freely available for download at `http://www.cs.utah.edu/~beazley/SWIG/swig.html`.

References

[1] 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.

[2] P.S.Lomdahl, P.Tamayo, N.Gronbech-Jensen, and D.M.Beazley, "50 Gflops Molecular Dynamics on the CM-5," Proceedings of Supercomputing 93, IEEE Computer Society (1993), p.520-527.

[3] R.C.Giles and P.Tamayo, Proc of SHPCC'92, IEEE Computer Society (1992), p. 240.

[4] S.Plimpton, "Fast Parallel Algorithms for Short-range Molecular Dynamics," J Computational Physics, vol 117, (March 1995) p 1-19.

[5] Y.Deng, R. McCoy, R. Marr, R. Peierls, O. Yasar, "Molecular Dynamics on Distributed-Memory MIMD Computers with Load Balancing," Applied Math Letters 8, No. 3 (1995), p. 37-41.

[6] M.P.Allen and D.J. Tildesley. Computer Simulations of Liquids. Clarendon Press, Oxford (1987).

[7] MRS Bulletin, "Interatomic Potentials for Atomistic Simulations", Vol 21, No. 2 (1996). This volume provides several articles and an overview of atomic potentials.

[8] A.I.Melcuk, R.C.Giles, and H.Gould, Computers in Physics (May/June 1991). p. 311.

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

[10] G.Eisenhauer, W.Gu, K. Schwan, and N. Mallavarupu, "Falcon-Toward Interactive Parallel Programs : The On-line Steering of a Molecular Dynamics Application," Proc of the Third International Symposium on High Performance Distributed Computing (HPDC-3), IEEE Computer Society (1994), pg. 26-34.

[11] G. Eisenhauer, et al. "Opportunities and Tools for Highly Interactive Distributed and Parallel Computing," Proc of the Workshop on Debugging and Tuning for Parallel Computer Systems, Chatham, MA. 1994. (in print)

[12] J.A. Kohl, P. M. Papadopoulos, "A Library for Visualization and Steering of Distributed

Simulations using PVM and AVS", High Performance Computing Symposium ’95, Montreal, CA,(1995).

[13] D.M. Beazley and P.S. Lomdahl, "High Performance Molecular Dynamics Modeling with SPaSM : Performance and Portability Issues," Proceedings of the Workshop on Debugging and Tuning for Parallel Computer Systems, Chatham, MA, 1994 (in print).

[14] S.Bryson, "The Data Glut Revisited," Computers in Physics, Vol.9, No.5, (1995), p. 525-530.

[15] C.D. Hansen, M. Krogh, and W. White, "Massively Parallel Visualization : Parallel Rendering," Proc. of 7th SIAM Conference on Parallel Processing for Scientific Computing, (1994), p. 790-795.

[16] C. Cruz-Neira, et al. "Scientist in Wonderland: A Report on Visualization Applications in the CAVE Virtual Reality Environment," Proc. of IEEE Symposium on Research Frontiers in Virtual Reality (1993), p. 59-66.

[17] "Virtual Environments and Distributed Computing at SC’95 : GII Testbed and HPC Challenge Applications on the I-WAY", ed. H. Korab, M. Brown. ACM/IEEE. (1995).

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

[19] G. van Rossum, Python Reference Manual, (1995).

[20] J. Levine, T. Mason, and D. Brown, Lex and Yacc. O’Reilly & Associates, Inc. (1992)

[21] 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.

[22] P. Dubois, K. Hinsen, and J. Hugunin, "Numerical Python", Computers in Physics (to appear 1996).

Author Biographies

David M. Beazley
beazley@cs.utah.edu
http://www.cs.utah.edu/~beazley

Dave Beazley is a Ph.D. student in the Department of Computer Science at the University of Utah where he is working in the Scientific Computing and Imaging (SCI) group. Since 1990, he has worked at Los Alamos National Laboratory in the Center for Nonlinear Studies and the Condensed Matter and Statistical Physics Group in the Theoretical Division. Beazley received his M.S. in mathematics from the University of Oregon in 1993 and a B.A. in mathematics from Fort Lewis College in 1991. His research interests include parallel computing, high performance computing architecture, languages, and low-level software development tools for large-scale scientific computing.

Peter S. Lomdahl
pxl@lanl.gov
http://bifrost.lanl.gov/~pxl

Peter Lomdahl is a staff member in the Condensed Matter and Statistical Physics Group in the Theoretical Division at Los Alamos National Laboratory where he has worked on computational condensed-matter and materials-science research since 1985. From 1982 to 1985, he was a postdoctoral fellow in the Center for Nonlinear Studies. Lomdahl received his M.S. in electrical engineering and his Ph.D. in mathematical physics from the Technical University of Denmark in 1979 and 1982. His research interests include parallel computing and nonlinear phenomena in condensed-matter physics and materials science.

Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC'96)
0-89791-854-1/96 $ 10.00 ACM