RECEIVED
JUN 1 1 1996
OSTI

TITLE: Lightweight Computational Steering of Very Large
Scale Molecular Dynamics Simulations

AUTHOR(S): David M. Beazley
Peter S. Lomdahl

SUBMITTED TO: Supercomputing '96, November 1996
Pittsburgh, PA

MASTER

Los Alamos
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

DISCLAIMER

Portions of this document may be illegible
in electronic image products.   Images are
produced from the best available original
document.

Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations

David M. Beazley
Department of Computer Science
University of Utah
Salt Lake City, UT 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/T11_staff/Lomdahl.html

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 extremely easy to modify, works with existing C code, is memory efficient, and can be used from inexpensive workstations and networks. We demonstrate how we have used this system to manipulate 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 build systems that integrate common scripting languages (including Tcl/Tk, Perl, and Python), simulation code, user extensions, and commercial data analysis packages.

Keywords:

Molecular dynamics, data analysis, large-scale simulation, steering, scripting languages, visualization, CM-5, T3D, 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 large-scale 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 have grown to as many as 1 billion atoms in 3D [2,4,5]. This in turn has created a serious problem. Production simulations involving tens to hundreds of millions of atoms are now routinely possible, but

analyzing the resulting data has proven to be extremely difficult if not impossible. As a result, most MD simulations remain small even though most researchers agree that large-scale simulations are useful for studying many different types of material properties.

In this paper, we describe our efforts in addressing the practical problems of working with very large molecular dynamics simulations involving tens to hundreds of millions of atoms. Our primary goal has been to devise a system that addresses our scientific needs in basic materials science research. 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 designed a system that is extremely lightweight, portable, easily extensible, and not dependent on expensive special purpose hardware. (ie. graphics workstations or high-speed networking).

The SPaSM Code

We originally developed the SPaSM (Scalable Parallel Short-range Molecular dynamics) molecular dynamics code for use on the Connection Machine 5 at Los Alamos National Laboratory [1]. The code is written entirely in ANSI C with explicit message passing used for interprocessor communication. In 1992, SPaSM became the first code to simulate more than 100 million (10^8) particles in both 2D and 3D [1]. In 1993, SPaSM was one of the winners in the IEEE Gordon Bell Prize competition [2]. Since then, a memory optimization allowed us to increase simulation sizes to as many as 300 million particles in double precision (on a 1024 node CM-5 with 32 Gbytes of memory) and the code has been ported to a variety of other parallel machines. The code has also been used for a variety of production simulations involving as many as 104 million atoms.
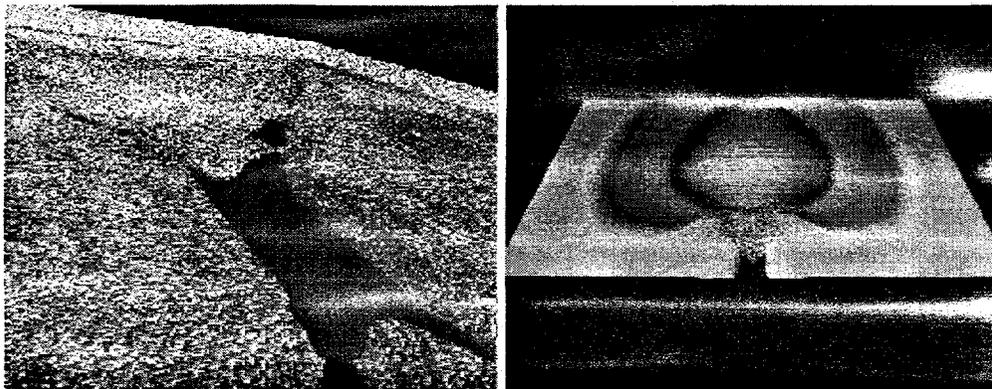

Figure 1 : Fracture experiments with 38 million particles (left)
and 104 million particles (right).

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 SPARCservers, and single processor workstations (for development purposes). Normally, the message passing wrappers are implemented in the native message passing environment available on each machine, but a general MPI version is also available.

The performance of the SPaSM code has been discussed extensively in the literature [1,2,13]. While early versions used CDPEAC assembler language for accessing the CM-5 vector units, the current

version no longer uses this approach and is optimized for RISC architectures instead. Table 1 shows some recent performance results of the SPaSM code on several different machines. This table is provided primarily to illustrate both the performance and simulation sizes that are currently possible.

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. 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)

Using SPaSM, we have been able to perform MD simulations with more than 100 million particles since 1992. However, running such a simulation in practice has proven to be almost impossible due to the enormous amount of data that must be analyzed. For example, the 104 million particle simulation in Figure 1 generated a collection of 40 1.6 Gbyte datafiles that only contained particle positions and kinetic energy in single precision. None of the users have a workstation capable of handling these datasets. Furthermore, even if we had a high-end graphics workstation fully loaded with memory, data analysis would still be difficult. Most networks are unable to ship 64 Gbytes of data around efficiently so it is difficult to get the data to the workstation in the first place. Once loaded into the workstation (assuming it has enough memory), we have found that systems such as AVS are completely ineffective for handling datasets with more than about a million atoms. 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 data analysis from a simulation running on a 512 processor Connection Machine or large 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 solve some of our data analysis problems in the past, we have sometimes used a high speed parallel rendering tool [15]. This is how the 104 million atom picture in Figure 1 was generated (on a Cray T3D). Unfortunately, this approach has not addressed our scientific needs. We can make pictures, but few of the SPaSM users know how to run the rendering code

and pictures alone don't provide that much information. The system also requires several minutes to make a picture and can barely be called interactive. We need a system where we can perform data-analysis and feature extraction coupled with visualization from our existing workstations. We also need this system to be of extremely high performance--the users simply do not want to sit around for minutes and hours on end waiting for results to appear.

Unfortunately, it seems that solutions to the large-dataset problem have become more and 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 true, our experience in the I-WAY at Supercomputing'95 was a complete disaster [17]. Even if it 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 a scientist who is trying to use SPaSM from a remote location and an ordinary UNIX workstation.

Computational Steering

To address our needs of large-scale simulation, data analysis, and visualization, we have adopted the approach of "computational steering" which aims to combine all of these aspects in an interactive environment. 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.

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 and must be able to run in the background without user intervention (just because you have steering doesn't mean you don't want to run batch jobs).
- Memory efficiency. Large-scale MD requires a significant amount of memory. The steering system should not.
- Network efficiency. There is too much data to be shipping across the network to a users workstation. We must be able to work with the data directly on the parallel machine running the simulation.
- Extensibility. The user must be able to extend the code with new features and new functionality with a minimal amount of work.
- Compatibility. The steering system should be able to work with existing code.
- Portability. The system must work on all parallel machines and workstations.
- Ease of use. Physicists must be able to both use and extend the system. This needs to be as simple and flexible as possible.

Previous efforts in computational steering have focused primarily on interactivity while ignoring many of the issues important for large-scale simulation [9,10,11,15]. As a result, most approaches continue to rely on high-end graphics workstations and high-bandwidth networks -- a solution that simply does not

work for very large-scale MD simulations at this time due to both the high cost and limited performance.

Our approach incorporates all of the features listed above, but with an emphasis on very large-simulations and simplicity. Instead of just worrying about interactivity, we focus on issues related to memory efficiency, long running jobs, portability, and flexibility. 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 lets us control and interact with our data, but is also sensitive to the needs of the users and the issues of working with very large datasets. 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 built our system around extensible scripting languages such as Tcl/Tk and Python [18,19]. This allows a user to both interact with the code (by issuing commands) or write scripts for controlling long-running jobs. The overall structure is shown in Figure 2.
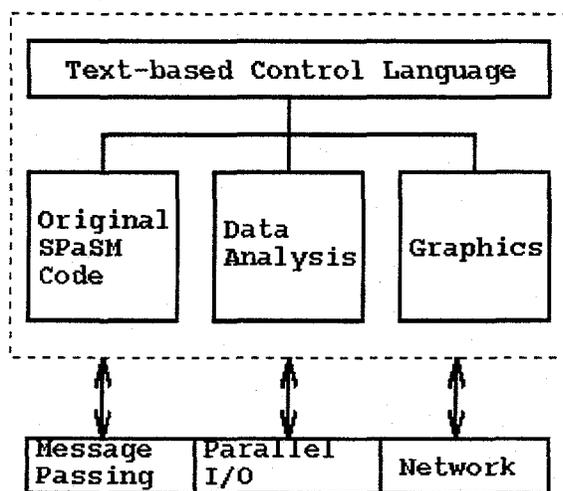

Figure 2 : SPaSM Organization

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 have 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 relies on 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. However, this must be usually be done with some care to avoid deadlock. As it turns out, the SPaSM code is independent of the

scripting language interface and can be compiled to run under Tcl, Perl, or Python as well (we return to this point later).

The code can now be used in a number of ways. First, to the skeptics who feel that a text-based interface is horrible, we point to the success of commercial packages such as MATLAB, Mathematica, IDL, and Maple--all of which have this type of interface. SPaSM can be used in a similar manner by interactively issuing commands to set up initial conditions, change boundary conditions, extract interesting data, or perform visualization. For long-running simulations, the user can write scripts for controlling an entire run. Secondly, we would like to emphasize the memory efficiency of this approach. Adding a scripting language requires very little memory. The parser really only manages a small stack for the LALR(1) parsing method used by YACC [20]. As a result, there is virtually no noticeable impact on memory usage. Finally, we feel that it is important to emphasize that while a text-based interface might not be as "flashy" as a sophisticated GUI, it is easily portable, doesn't require much network bandwidth, and can be just as powerful (if not more so).

Automated Language-Independent Interface Generation

In order for a steering system to be useful, we feel that the user must be able to extend it with new features. In our case, this involves extending the scripting-language interface with new commands and variables. The standard technique used by most scripting languages is to write special "wrapper" functions that provide the glue code between the scripting language interface and the C function. Unfortunately, writing these wrapper functions is usually quite technical, tedious, 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 builds the entire user interface from ANSI C function and variable declarations (Currently SWIG supports Tcl/Tk, Perl4, Perl5, Python, Guile, and our own scripting language) [21]. Using SWIG, a typical user-interface specification looks like the following :

```
%init User_Init
%{
#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(void);
extern void     set_boundary_free(void);
extern void     set_boundary_expand(void);
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 scripting language interface. In order to use the function, it's ANSI C prototype declaration is simply placed into the interface file. When SPaSM is compiled, that new function will automatically appear in the interface. This scheme makes it extremely easy to add new features. Since the interface file specification contain no code specific to any one scripting language, we are able to compile SPaSM under any number of scripting languages as needed (although most scripting languages do not work well in a massively parallel environment due to I/O problems).

Building Modules and Packages

SWIG allows users to easily build packages of interesting modules by allowing interface specification files to include other interface files. Thus, a more complex user interface could be built as follows :

```
%init User_Init
%{
#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 of the files can 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. 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

The SWIG interface builder can be used with a surprising variety of C code. In fact, it handles C pointers, structures, and objects in addition to simple functions as shown in Code 1 [21]. Within the scripting language, the user can create vectors, particles, arrays, and other objects. These in turn can be passed on to other C functions for later use. For example, suppose we want to make a module for culling some of the particle data based on the value of its individual potential energy contribution (a useful technique we have used for finding dislocations). We could write a special SWIG interface file containing a C function for searching the particle data. In the following example, we've written such 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 function 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.

Now, suppose we wanted to extract two sets of particles with different potential energy ranges and make an image. The user could type the following:

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

This example is interesting on several levels. First, we have used a C function to find and return pointers to interesting particles. These pointers have then been used to construct Python list objects. Once represented in this high level, we can form new lists with simple arithmetic operators. These lists, can in turn be passed to other functions for making images and analysis. Of course, there are many different ways of accomplishing the same task--an indication of the flexibility of this approach.

To further emphasize the flexibility of the system, SWIG has also been used to build interfaces out of commercial packages including MATLAB and the entire Open-GL library. In short, almost any type of C code can be integrated into our steering system (although not all codes will work on parallel machines).

A Scripting Example

Many 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 really isn't necessary. Thus, one

of the primary uses of our system is controlling long-running simulations. The example in 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, we see that our C function in the interface file are called as in C. It's also possible to include other scripts as shown in the `source("Examples/makemorse.script")` command. Scripts can also be executed when running interactively. This is often useful for simplifying common operations or setting up various parameters. In many cases, scripts offer a huge time savings as users can make changes to scripts and see the effects without ever having to recompile the SPaSM code.

An Interactive SPaSM Example

When working in an interactive mode, we can perform data analysis and visualization. We have developed a high-performance memory efficient graphics module that allows us to remotely visualize MD data with as many as 150 million atoms on a 512 processor CM-5 (this work is in progress and will be published elsewhere). The graphics module is integrated directly with the SPaSM code and can visualize data from running simulations. It can also be used for post-processing and analysis. The following example shows how this system is used on an 11 million particle impact simulation. Each data set is 180 Mbytes. Image is sent as GIF files (which use LZW compression) 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] >
```
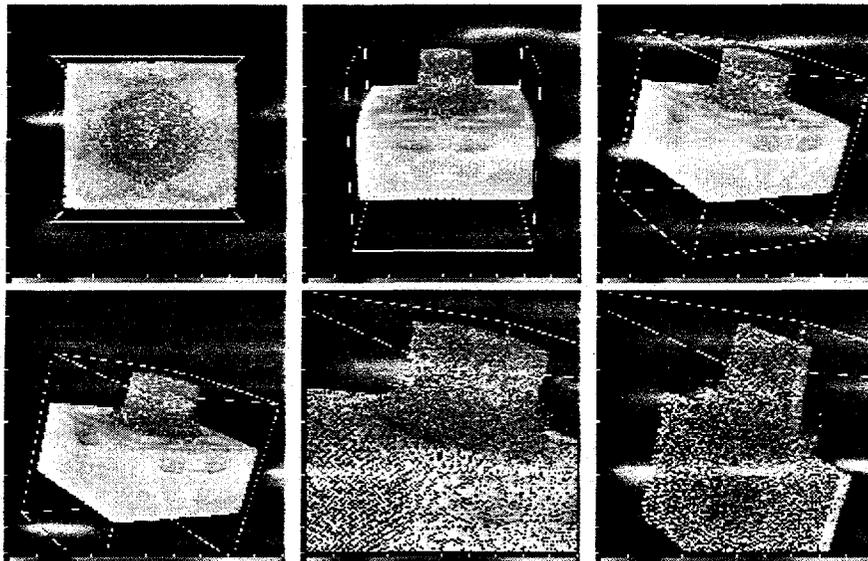

Fig 3 : Six images generated by the interactive example (in order from left to right).

To some, this approach may seem archaic or even insane, yet we have found the command based approach to work remarkably well in practice. 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. It is also important to keep in mind that when we tried to work with this 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 [15].

While this example has been simple, we can also use SPaSM 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 and without using a memory and network intensive steering "environment."

Data Exploration and Feature Extraction

One of the benefits of using large 3D materials simulations is that it allows us to study phenomena that require relatively large systems to be seen. For example, we may be interested in the formation of dislocations or damage effects. 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. Figure 4 shows two examples of this. 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 Gigabytes 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 local workstations. Images take about 10 seconds to generate, but this is a huge improvement from our prior experiences using graphics workstations. Interestingly enough, we can use the feature extraction capability to reduce the datasets down to a size that can be handled on an ordinary workstation. In Figure 4a, a single snapshot is 700 Mbytes, but by removing the bulk, this can reduced to only 10 or 20 Mbytes--a file that can be easily handled.
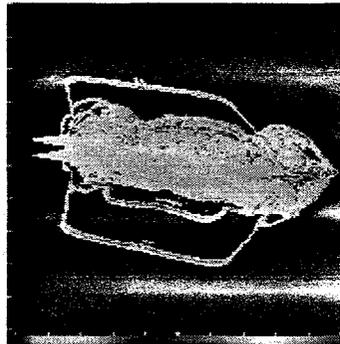

Fig 4a. Dislocation loops in 35 million atom fracture simulation. Generated from a 700 Mbyte datafile. (MPEG movies of these simulations are also available).

Fig 4b. Ion-implantation in 5 million atom silicon crystal. Generated from a 100 Mbyte datafile.

Debugging, Prototyping, and Development

Due to the portability of the SPaSM code, it is possible to do code development and debugging on ordinary workstations. Due to the language-independent interface specification, we can build SPaSM under a number of popular scripting languages including Tcl/Tk, Perl, and Python. It is also possible to integrate packages such as MATLAB directly into SPaSM for performing more sophisticated analysis. Figure 5 shows a screen shot of such a simulation. In this simulation, we are testing a small MD shock-wave problem on a single processor Unix workstation. The simulation itself is being controlled by the Tcl interpreter shown in the lower left corner. Any of the SPaSM commands (and arbitrary Tcl scripts) can be executed here in addition to MATLAB commands. In the upper left corner, we see a MATLAB figure showing particle velocities as a function of position. To the right of that we see the latest snapshot of particle kinetic energies produced by the SPaSM visualization module. These two screens are updated in real-time as the simulation runs. Two Tk windows are also shown. The toolbar on the right contains visualization shortcuts and the menu in the lower right allows us to load data files and change various settings.
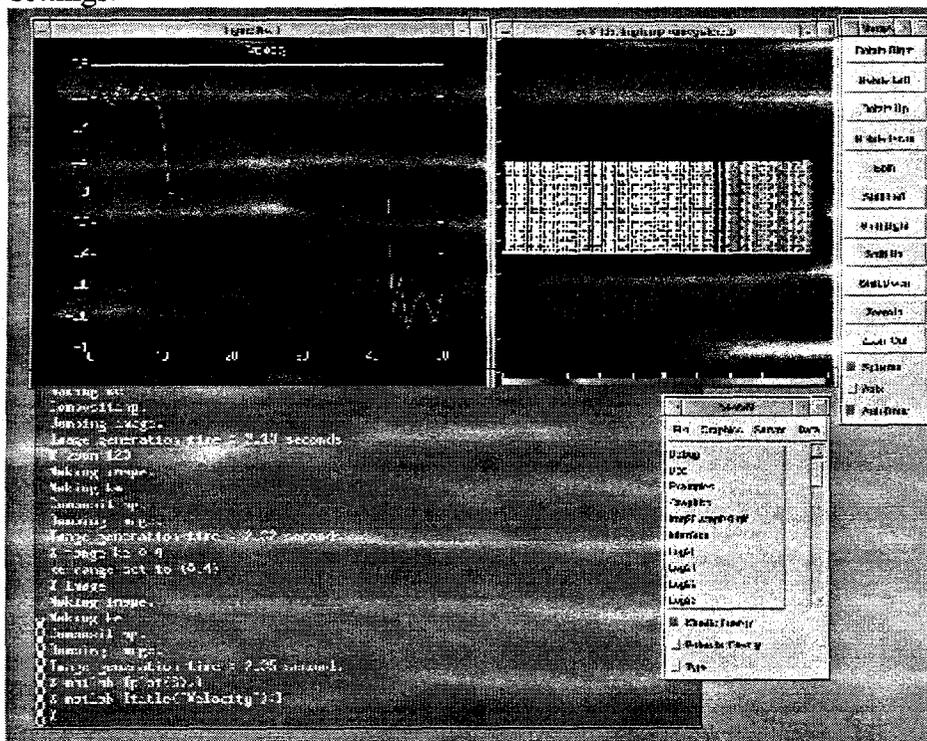

Fig 5: SPaSM/MATLAB running on a workstation under Tcl/Tk.
(Click on image for full-size view).

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 developed 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 parallel machines. By having a highly flexible architecture, we can combine a wide variety of analysis tools during the development phase and still move up to larger, less interactive, simulations when we are ready.

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 highly 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.

Future directions for our work include the development of better 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]. 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. As the SPaSM code becomes more widely used, we feel that management of data, run parameters, and output, will be more critical than simply providing more interactivity.

Acknowledgments

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.Tamyo, 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 4th USENIX Tcl/Tk workshop, (to appear 1996).

[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

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 on problems related to remote manipulation of very large-scale scientific datasets and parallel computation. He is also interested in extensible scripting languages, scientific databases, high-performance computer architecture, and large-scale materials science simulations. Since 1990, he has worked at Los Alamos National Laboratory in the Center for Nonlinear Studies and Theoretical Division on the development of the SPaSM molecular dynamics code for the CM-5 and Cray T3D. 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.
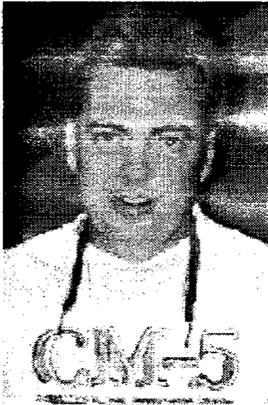
Peter S. Lomdahl
pxl@lanl.gov

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.