

# MDL, A Domain-Specific Language for Molecular Dynamics

Trevor Cickovski, Chris Sweet and Jesús A. Izaguirre\*  
Department of Computer Science and Engineering  
University of Notre Dame  
Notre Dame, IN 46556

\* Author to whom correspondence should be addressed. E-mail: izaguirr@nd.edu.

## Abstract

*Molecular Dynamics (MD) involves solving Newton's equations of motion for a molecular system and propagating the system by time-dependent updates of atomic positions and velocities. As a severe limitation of molecular dynamics is the size of the timestep used for propagation, a key area of research is the development of efficient propagation algorithms which can maintain accuracy and stability with larger timesteps. We present MDL, an MD domain-specific language with the goals of allowing prototyping, testing and debugging of these algorithms. We illustrate the use of parallelism within MDL to implement the Finite Temperature String Method, and interfacing to visualization and graphical tools.*

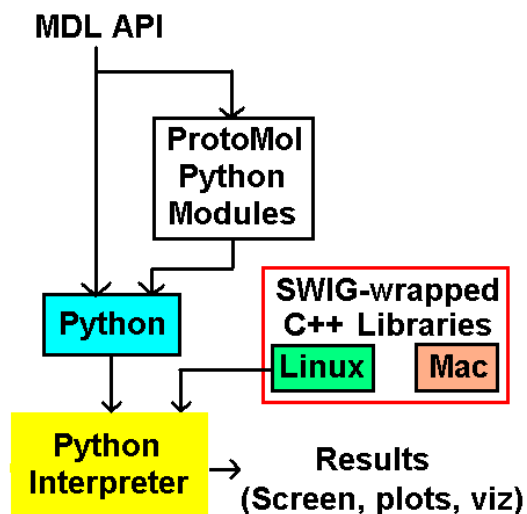
## 1. Introduction

In Molecular Dynamics (MD), a molecular system is propagated through time by solving Newton's equations of motion, or  $F = ma$ , for each atom in the system and subsequently updating atomic positions and velocities over a timestep  $\Delta t$  using the resulting acceleration  $a$ . Different types of forces can be accounted for, which occur at multiple frequencies. For example, bond fluctuations vary within a few femtoseconds, whereas forces between nonbonded pairs of atoms like van der Waals and electrostatic forces have more slowly varying components [28]. An unfortunate consequence of this is that the timestep for propagation is limited by the fastest motion, because a timestep which is too high fails to account for multiple fluctuations which occur at higher frequencies and introduces unacceptable inaccuracies in computation. This results in often impractically long simulations, and challenges in obtaining molecular data over periods of time of any biological relevance. A key performance bottleneck is associated with determining electrostatic forces between atom pairs, which can be very expensive for large molecular systems.

As a result of these challenges, software which runs MD simulations often tends to focus on performance. Examples of C++ MD frameworks include NAMD [16] and ProtoMol [19]. Each has been proven capable of running computationally intensive simulations. However, without a propagation scheme that can operate at larger timesteps, high performance computing only goes so far. For example, even with the massively parallel supercomputers we have today running highly optimized software, it is still extremely difficult to run MD simulations for a total time that exceeds the microsecond scale. This is the motivation behind the development of multiple time stepping integration schemes, where different propagation algorithms are invoked at various frequencies, calculating different types of forces [32]. This also motivates the design of propagation algorithms which make acceptable approximations and constraints [1, 34] of fast frequency forces and as a result can afford larger timesteps.

Thus, improving the overall simulation time for a molecule involves two goals: designing high-performance software and prototyping propagation schemes which can accept larger timesteps. We focus on the latter goal by providing the domain-specific language MDL (Molecular Dynamics Language) as an MD tool which allows for efficient and productive prototyping, testing, and debugging of new propagation schemes and simulation protocols. MDL provides a scripting interface to MD through a high level API resting upon Python [25] and SWIG-wrapped [3] C++ libraries from ProtoMol, as shown in Figure 1. The abstraction level of MDL represents MD-specific phenomena in a simpler fashion compared to a general purpose language, making it easier for domain experts to prototype simulations and test methods without requiring knowledge of a software framework. The scripting interface also provides better error-checking [29] and debugging, and Python has several advantages over other scripting languages, one of which is the abundance of general purpose Python tools which can be interfaced to MDL. Python is also dynamically typed, and is stable and cross platform. Performance

can be improved through byte compilation of scripts.



**Figure 1.** MDL control flow diagram. The MDL API sits on top of both Python and importable modules generated by SWIG-wrapping ProtoMol C++ source. Shared libraries are generated for each compatible platform, currently including 32 and 64-bit Linux, and Mac OS X.

Since our main goal is to facilitate numerical method and protocol development, in comparison to other MD languages MDL focuses more on ease of use and less on performance. For instance, MDX [31] can also be used to prototype MD simulations and methods and gains performance benefits over Python through the use of an API built upon C libraries. In contrast, MDL is interactive and scripted, saving the need to recompile simulation source code even after small changes, and also saving any requirement of user knowledge of compilation tools such as makefiles. In addition, MDL runtime errors are more understandable for a nonprogrammer, in comparison to C where for example nondescriptive segmentation faults could occur upon indexing an array out of bounds, making debugging more difficult. Our argument for emphasis on ease of use over performance is that once a numerical method or simulation protocol is obtained, it can always be rewritten using a faster implementation by a programmer. But creating the methods themselves is most of the battle. Arun *et al.* also provide an easy to use interface using XML in their MoDL language [2]; although XML was useful for their particular purposes of data description for visualizing data from pre-run simulations, it is much less practical for ours, which are mathematically oriented.

We first illustrate how MDL can be used to prototype

propagation schemes and simulations, followed by some examples. We then show some useful interfacing of MDL to other software such as visualization engines and graphing libraries, and useful optimizations such as parallelism. MDL is available open source on the Simulation Toolkit at <https://simtk.org/home/mdl/>.

## 2. Simulation Prototyping

An MDL simulation comprises four parts. The first specifies the *physical* system, or a specification of molecular structure including atomic coordinates, bond structure and spring constants, a description of the forces, etc. The second deals with *approximations* made in various calculations for the purposes of saving performance. For example, as computing electrostatic forces for all pairs of atoms in a large molecule can be computationally impractical, a common approach is to *cut off* electrostatic forces after a certain distance  $x$  where the force becomes small enough to approximate zero and maintain accuracy. To avoid the abrupt dropoff in electrostatic force after  $x$  which could result in instabilities, a *switching function* is often applied to smooth the function to zero at  $x$ . A third component is *data collection*. This could involve something as simple as screen output, or more complex file I/O such as energy files, DCD trajectory files to be played back in a visualization engine such as VMD, etc. The final component is the *task* to be performed, including the propagation scheme to be used, timestep, and total time.

An MDL program consists of four Python methods representing these four tasks: `physical()`, `approximations()`, `output()`, and `execute()`. We illustrate examples with a simulation of a 66-atom decalane molecule.

### 2.1. Physical System

The best way to input a physical system is through files. Some commonly used file types in MD are the PDB (Protein Data Bank) [4] or XYZ format for atomic coordinates, and CHARMM [6] PSF (structure) and parameter files (PAR). The physical system also specifies Kelvin temperature, boundary conditions, and force fields. Boundary conditions can be periodic or vacuum. The MDL framework includes several example physical systems, such as argon, solvated Bovine Pancreatic Trypsin Inhibitor (BPTI), alanine dipeptide, united atom butane, and water molecules. Below we show an MDL `physical()` method which loads PDB, PSF and PAR files for alanine, then sets the simulation boundary conditions to `Periodic` and temperature to 300 K. We also declare an MDL `ForceField` at the top of the file:

```
ff = forcefield("charmm")

def physical():
    coordinates("alanine.pdb")
    structure("alanine.psf")
    parameters("alanine.par")
    boundaryConditions("Periodic")
    temperature(300)
```

An MDL `ForceField` encompasses a set of forces for evaluation and several MDL methods can manipulate these structures. The MDL routine `forcefield()` returns a `ForceField` object. If one passes the string "charmm" as above, the `ForceField` will contain all bonded forces: due to two-atom bonds, three-atom angles and four-atom dihedrals and impropers, and the nonbonded LennardJones (or van der Waals) and Coulomb (electrostatic) forces. However, MDL `ForceFields` are customizable, and different force combinations can be specified through string parameters to MDL routines `addBondedForces()` and `addNonbondedForces()`. Within these strings, the character 'b' references bond, 'a' angle, 'd' dihedral, 'i' improper, 'l' LennardJones and 'c' Coulomb. So, the following statements are equivalent to the above:

```
ff = forcefield()
addBondedForces(ff, "badi")
addNonbondedForces(ff, "lc")
```

## 2.2. Approximations

We now show an example MDL `approximations()` routine which defines force calculation approximations for LennardJones and Coulomb forces (we compute them together as `LennardJonesCoulomb` to save performance). We use a cutoff algorithm (as opposed to a direct computation), alongside a continuous switching function `Cn`, cutting off the force computation at a distance of 12 Å and turning on switching at a distance of 10 Å. By specifying an order of 2, the second derivative of the switching function is continuous.

```
def approximations():
    algorithm(ff, "LennardJonesCoulomb",
              "Cutoff")
    switching(ff, "LennardJonesCoulomb",
              "Cn")
    cutoff(ff, "LennardJonesCoulomb",
           12)
    extraparameters(ff, "switchon=10.0",
                    "switchoff=12.0", order="2")
```

Coulomb forces can accept one of several algorithms for fast electrostatic evaluation, like Ewald [9], PME (Particle Mesh Ewald, [8]), and Multigrid [31].

## 2.3. Output

MDL allows a user to specify different types of output including data plots, files, and screen output, through the `output()` routine. For example, the following will plot total energy every 100 steps, and print timestep, total energy and volume to the screen every step:

```
def output():
    plotTotal(100)
    printScreen(1)
```

MDL can output several different types of simulation data, including temperature, energies (bond, angle, potential, kinetic, etc.), momentum, DCD trajectory files, and trajectory force, position and velocity files. All energies can be plotted with time, as well as pressure, volume and temperature. Position and velocity files can also be recorded in PDB, XYZ or binary XYZ formats, for example:

```
writePDBPos("positions.pdb")
writeXYZVel("velocities.xyz")
```

## 2.4. Execution

The `execute()` method contains a call or series of calls to `propagate()`:

```
def execute():
    # Run the Leapfrog method for 2000
    # steps at a timestep of 0.5 fs
    # with the passed force field ff.
    gamma = propagate("Leapfrog", 2000,
                      0.5, ff)
```

The Leapfrog method [12] is one of several methods that have been predefined and included with the MDL package as pre-compiled shared objects to improve performance. Other methods include: BBK [7], Langevin Impulse [30], Nose Hoover [13, 21], NPT and NVT Verlet [33], and various MOLLY methods [15, 17]. Other propagators allow inclusion of the Hybrid Monte Carlo [5] and Shadow Hybrid Monte Carlo [11] methods, as well as umbrella sampling [20]. A complete list of predefined propagation schemes can be found on the MDL website, along with a description of all available routines in the MDL API.

## 3. Prototyping methods

### 3.1. MDL Low Level Structures

MDL contains some core structures for performing operations, which we now describe.

1. **Vector3DBlock:** This Python class defines a vector of three-dimensional coordinates, and is the type returned by the MDL routines `position()`, `velocity()`, and `force()` which for an  $n$ -atom simulation represent  $n$ -element vectors of atomic positions, velocities and forces. The `Vector3DBlock` supports vector and scalar multiplication, addition, and subtraction. When constructing a propagator, it is often necessary to update atomic positions and velocities, which can be done through these vector operations along with the MDL routines `setVelocity()` and `setPosition()`. For example, a *kick* of the Leapfrog method [12] updates positions by a full timestep using the following equation:

$$X^{n+1} = X^n + \Delta t V^{n+1/2}, \quad (1)$$

which assuming `dt` has been defined to represent the propagation timestep, can be performed in MDL with the statement:

```
setPosition(position() +
             dt*velocity())
```

2. **DiagonalMatrix:** The MDL routine `mass()` returns an  $n \times n$  `DiagonalMatrix` with the element at position  $(i, i)$  corresponding to the mass  $m_i$  of atom  $i$ . The routine `invMass()` returns a similar structure but the element at  $(i, i)$  is equal to  $1/m_i$ . `DiagonalMatrix` supports the same operations as `Vector3DBlock`, optimized accordingly for a diagonal matrix. Performing a *halfkick* of the Leapfrog method uses the following equation to update velocities by half a timestep:

$$V^{n+1/2} = V^n + \frac{\Delta t}{2} M^{-1} F^n, \quad (2)$$

requiring the use of `invMass()`:

```
setVelocity(velocity() +
            0.5*dt*invMass()*force())
```

### 3.2. Propagation Schemes

We now show the use of the above structures to construct an MD propagator. An MD *propagator*  $\Psi$  maps a phase space point  $\Gamma$  to a new point  $\Gamma'$ :

$$\Gamma' = \Psi(\Gamma), \quad (3)$$

by calculating forces on each atom, and solving Newton's equations of motion to obtain new atomic positions and velocities after some time  $\Delta t$ . Propagators in general have a limited size of  $\Delta t$  due to stability and accuracy concerns. We illustrate an MDL implementation of the BBK [7] propagation scheme as a Python class.

The first step is to define a Python class `BBK`. `BBK` is a *single timestepping* (STS) propagator, meaning that all forces (bonded and nonbonded) are evaluated using the same timestep [28]:

```
class BBK(STS):
```

An MDL propagator defines methods for initializing (`init()`) and running (`run()`) the method. `init()` is invoked when the propagator object is created, and `run()` at every step of propagation. For purposes of this method, we simply calculate forces once for initialization. This is performed using the MDL `calculateForces` routine, calculating forces using the associated `ForceField` which is passed in the call to `propagate()`, and accumulating them into the `Vector3DBlock` returned by `force()`.

```
def init(self):
    calculateForces(position())
```

We will perform all phase space updates in a `run()` method:

```
def run(self):
```

We add a random force into the calculations:

$$F = F + F_R \sqrt{\frac{2kT\gamma}{\Delta t}}, \quad (4)$$

done through the MDL statements below, using an auxiliary `forceconstant`:

```
forceconstant = 2*boltzmann()*self.temp*
               self.gamma/timestep(self)
setForce(force()+randomForce(self.seed)*
         sqrt(forceconstant))
```

Assuming we propagate a system from step  $n$  to step  $n+1$  (where time corresponds to  $n\Delta t$  and  $(n+1)\Delta t$ ), we can now update the velocity vector by a half kick using Eq. (5), which provides some initial damping:

$$\begin{aligned} V^{n+1/2} &= V^n (1.0 - \frac{\Delta t}{2} \gamma) \\ V^{n+1/2} &= V^{n+1/2} + \frac{\Delta t}{2} M^{-1} F^n, \end{aligned} \quad (5)$$

```
setVelocity(velocity() *
            (1.0-0.5*timestep(self)*self.gamma))
setVelocity(velocity() +
            0.5*timestep(self)*invMass()*force())
```

We then perform a kick on positions:

```
setPosition(position() +
            velocity()*timestep(self))
```

And another calculation of forces followed by another random force:

```
calculateForces(position())
forceconstant = 2*boltzmann()*self.temp*
               self.gamma/timestep(self)
setForce(force()+randomForce(self.seed)*
         sqrt(forceconstant))
```

And then a final half kick on velocities, using Eq. (6),

$$\begin{aligned} V^{n+1} &= V^{n+1/2} + \frac{\Delta t}{2} M^{-1} F^{n+1} \\ V^{n+1} &= V^{n+1} (1 / (1.0 - \frac{\Delta t}{2} \gamma)), \end{aligned} \quad (6)$$

and implemented in MDL as follows:

```
setVelocity(velocity() +
            0.5*timestep(self)*invMass()*force())
setVelocity(velocity() *
            (1/(1.0+0.5*timestep(self)*self.gamma)))
```

The new propagator must be *registered* with the MDL back end so it is recognizable through a keyword when invoked. BBK requires some extra parameters to be defined by the user, which are specified in the following invocation. These are subsequently accessible in the propagator class.

```
object(name="BBK",
       parameters=("temp",
                  "gamma",
                  "seed"))
```

Once the propagator has been registered, the following call updates phase space using the new BBK propagator. Note that we must pass a timestep, number of steps, and a force field. Parameter and value pairs are passed as Python tuples.

```
# Run BBK for 1000 steps at a timestep
# of 0.1 fs
# with the passed force field ff and a
# temperature of 300 K, gamma of 0.3
# and seed of 1234
gamma = propagate("BBK", 1000, 0.1, ff,
                 ('temp', 300),
                 ('gamma', 0.3),
                 ('seed', 1234))
```

Alternatively, one can implement a propagator as a *procedure*, which accepts position and velocity vectors and updates them accordingly. The method can return updated position and velocity vectors as a pair of arrays. MDL supplies the method `toArray()` which converts a `Vector3DBlock` into a Numerical Python [22] array. This can be useful both from a computational perspective

and for data analysis, as Numerical Python provides several fast and useful matrix and vector operations.

Procedural propagators also accept a step count, timestep, and force field along with any other necessary parameters. Below, we show the same BBK integrator implemented as a procedure:

```
def bbk(position, velocity, steps,
         timestep, ff, temp, gamma, seed):
    force = Vector3DBlock()
    calculateForce(ff, position,
                  velocity, force)
    for (step in range(0, steps)):
        # assign random force
        forceconstant = 2*boltzmann()*temp*
                       gamma/timestep
        force = force+randomForce(seed)*
               sqrt(forceconstant)

        # first half kick
        velocity = velocity*
                  (1.0-0.5*timestep*gamma)
        velocity = velocity+
                  0.5*timestep*invMass()*force

        # drift
        position = position + velocity*timestep
        calculateForce(ff, position,
                      velocity, force)

        # assign random force
        forceconstant = 2*boltzmann()*temp*
                       gamma/timestep
        force = force+randomForce(seed)*
               sqrt(forceconstant)

        # second half kick
        velocity = velocity+
                  0.5*timestep*invMass()*force()
        velocity = velocity*
                  (1.0/(1.0+0.5*timestep*gamma))

    return [toArray(position),
            toArray(velocity)]
```

MDL procedural propagators are registered like propagator objects, but as a method instead of object. Parameters are defined in the same way.



```

S[conf] =
    FTSMaverage(conf, 1000,
                slopes[conf])
S, slopes = reparameterize(smooth(S),
                           42)

```

The first call to `initializeFTSM()` generates an initial diagonal string of 40 equally spaced points between the initial and final conformations, passed as PDB files. Next, `walk()` generates intermediate PDB files (labelled `conf1.pdb`, `conf2.pdb`, etc.) for these 40 intermediate points, by a walk through the conformations using a weak constraint on the backbone dihedral. Then `FTSMaverage()` runs 1000 steps of constrained dynamics at a timestep of 0.5 fs (for a total of 0.5 ps), starting from each of these PDBs using strong backbone dihedral constraints and storing average  $(\phi, \psi)$  in `S`. This is followed by a smoothing and reparameterization of `S`, the former of which uses the steepest decent method.

Averaging starting from a particular conformation can be performed in parallel, since it is independent of the averaging at other conformations. Thus, once we have our PDB files representing each conformation and constraints setup properly in the force field `ff`, we can run the averaging in parallel, dividing the intermediate conformations as evenly as possible among the nodes. Assuming that `numconf` holds the number of intermediate conformations, if we want to distribute the conformations sequentially (*i.e.* for four nodes and 40 conformations node 0 would get conformations 1-10; node 1 would get 11-20, node 2 would get 21-30 and node 3 would get 31-40), we can do the following, using `mpi.rank` as the node number (root is 0), and `mpi.size` as the total number of nodes.

```

start = mpi.rank*(numconf/mpi.size)+1
modVal = numconf % mpi.size
if (mpi.rank <= modVal and
    mpi.rank != 0):
    start += mpi.rank
else:
    start += modVal
end = start+(numconf/mpi.size)-1
if ((mpi.rank+1) <= modVal):
    end += 1

```

Now `start` and `end` respectively hold the starting and ending conformations for each individual processor. We can now run averaging in a loop on each processor:

```
currS = []
```

```

for conf in range(start, end+1):
    currS.append(FTSMaverage(conf, 1000,
                             slopes[conf]))

```

then collect results on the root, which we later will broadcast to all nodes after smoothing. We use `mpi.allgather` to concatenate the lists of updated  $(\phi, \psi)$  pairs, then append the starting and ending points of the string, followed by an `mpi.barrier()` which ensures no nodes pass this point until all have finished averaging:

```

tempS = mpi.allgather(currS)
S = S[0] + tempS + S[S.__len__()-1]
mpi.barrier()

```

We can leave smoothing and reparameterization as root node tasks. The updated string and slopes are returned into temporary arrays, which are subsequently broadcast to all nodes using `mpi.bcast()`. By placing an `mpi.barrier()` at the end, we ensure no nodes go back into running constrained dynamics before the root node finishes this process:

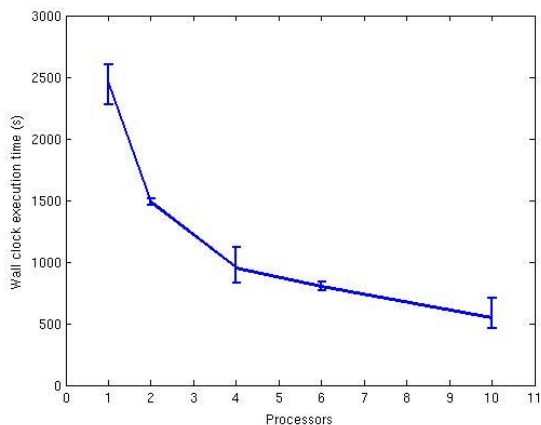
```

if (mpi.rank == 0):
    tempS, tempslopes =
        reparameterize(smooth(S), 42)
else:
    tempS = []; tempslopes = []
S = mpi.bcast(tempS)
slopes = mpi.bcast(tempslopes)
mpi.barrier()

```

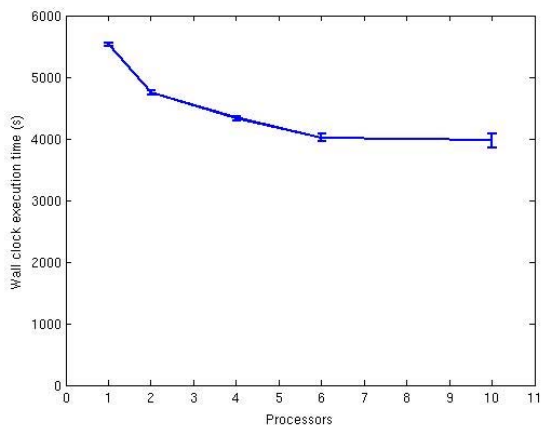
Tests were run on a 32-bit Atipa cluster running GNU Linux 2.6.13.1. Individual nodes contain dual Intel Xeon 2.4 GHz CPUs, with 1 GB RAM. Convergence was obtained in [27] after 60-70 updates. We show the wall clock execution times of 64 averagings of parallel FTSM for processor counts of 1, 2, 4, 6, and 10 in Figure 4, averaged over four runs.

Results illustrate significant savings between 1 and 4 processors, after which benefits level off slightly. With 10 processors, we were able to achieve about 45.3% parallel efficiency. Because averaging is the only FTSM routine currently parallelized, the overall savings is much less significant, shown in Figure 5, although we still were able to obtain roughly 30 minute savings on a 90-minute single processor simulation. This graph shows that the overhead of root tasks and subsequent communication with large amounts of processors is currently high, but we expect future improvement when we will attempt to parallelize other phases of the algorithm besides averaging. An initial attempt to parallelize the walking stage using PDB



**Figure 4. Wall clock execution times of the parallelized averaging stage of FTSM with various processor counts.**

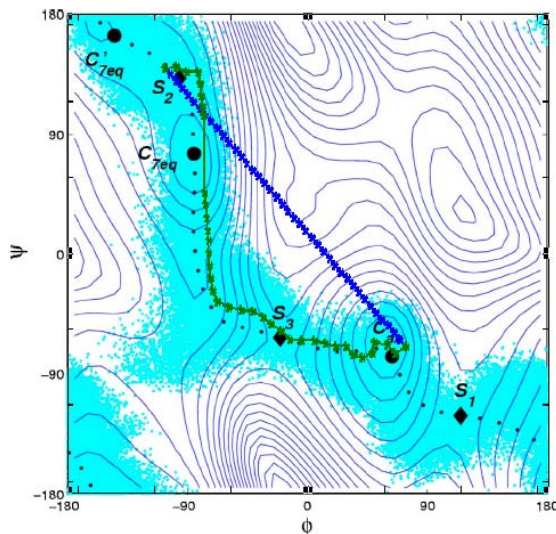
files from the previous iteration before smoothing indicated more painful dragging from an unsmoothed string to the smooth one as opposed to going point-by-point like we have been, which is non-parallelizable. However we will continue to explore other options, such as selective generation of PDBs while running constrained dynamics for use in subsequent walking stages. We will also look into piecewise smoothing of the string with each processor operating on its individual set of points (ideally, the same set as used in averaging), and study dependability of the resulting strings.



**Figure 5. Overall wall clock execution times of parallel FTSM.**

Figure 6 shows our initial diagonal and final string after 64 updates on alanine dipeptide, superimposed and con-

verging to the results of [27], shown as a dotted curve.



**Figure 6. Our initial diagonal string between two energetically favorable conformations of alanine dipeptide, and our finite temperature string after 64 updates. To illustrate convergence, we superimpose our results on those of [27] (dotted curve).**

## 5. Conclusions

We have presented a domain-specific language MDL as a propagation method development and simulation prototyping tool for molecular dynamics. MDL provides easy testing and debugging of various methods through a scripting interface and a domain-specific syntax understandable for molecular dynamics experts. In addition the scripting interface makes parameter sweeping easier. To improve its applicability as a development and prototyping tool, we have interfaced MDL to several Python visualization and plotting tools, along with Numerical Python libraries for fast vector and matrix operations, and PyMPI for parallelism. We have demonstrated its applicability as this type of tool through an example propagator definition and simulation prototype of a decalanine model.

Although performance is not a primary goal considering our intended application of the language, we have already taken some steps to reducing the penalty attributed to scripting by implementing most of the computationally intensive tasks such as force calculations and matrix-vector operations in either SWIG-wrapped precompiled shared binaries or providing the opportunity to execute them using

Numerical Python. Still, MDL in general is still on average roughly 2-3 times slower than leading MD frameworks such as ProtoMol and NAMD. An immediate goal is thus to carefully profile the framework to capture opportunities to optimize the code. Also, since we were able to take advantage of Python's cross-platform compatibility on Linux and Mac OS, we fully intend to extend this compatibility to Windows as well. We look forward to the continued growth of MDL for developing and testing novel MD algorithms.

## References

- [1] H. C. Andersen. Rattle: A 'velocity' version of the Shake algorithm for molecular dynamics calculations. *J. Comput. Phys.*, 52:24–34, 1983.
- [2] B. Arun, V. Chandru, A. D. Ganguly, and S. Manohar. Molecular dynamics visualization with XML and VRML. In *Proceedings of Computer Graphics International*. Geneva, Switzerland, 2000.
- [3] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceeding of The Forth Annual Tcl/Tk Workshop '96*, pages 129–139. USENIX Association, July 1996.
- [4] F. C. Bernstein, T. F. Koetzle, G. J. Williams, E. F. Meyer, M. D. Brice, J. R. Rogers, O. Kennard, T. Shimanouchi, and M. Tasumi. The Protein Data Bank: A computer-based archival file for macromolecular structures. *J. Mol. Biol.*, 112:535–542, 1977.
- [5] A. Brass, B. J. Pendleton, Y. Chen, and B. Robson. Hybrid Monte Carlo simulations theory and initial comparison with molecular dynamics. *Biopolymers*, 33:1307–1315, 1993.
- [6] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D. J. States, S. Swaminathan, and M. Karplus. CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *J. Comp. Chem.*, 4(2):187–217, 1983.
- [7] A. Brünger, C. B. Brooks, and M. Karplus. Stochastic boundary conditions for molecular dynamics simulations of ST2 water. *Chem. Phys. Lett.*, 105:495–500, 1982.
- [8] T. Darden, D. York, and L. Pedersen. Particle mesh Ewald: An N log(N) method for Ewald sums in large systems. *J. Chem. Phys.*, 98(12):10089–10092, 1993.
- [9] P. Ewald. Die Berechnung optischer und elektrostatischer Gitterpotentiale. *Ann. Phys.*, 64:253–287, 1921.
- [10] Gnuplot.py. Gnuplot.py on sourceforge, 2005. <http://gnuplot-py.sourceforge.net>.
- [11] S. S. Hampton and J. A. Izaguirre. Improved sampling for biological molecules using shadow hybrid Monte Carlo. In M. Bubak, G. D. von Albada, and P. M. A. S. J. J. Dongarra, editors, *4th International Conference on Computational Science, Kraków, Poland*, volume 3037 of *Lecture Notes in Computer Science*, pages 268–274. Springer-Verlag, New York, 2004.
- [12] R. W. Hockney and J. W. Eastwood. *Computer Simulation Using Particles*. McGraw-Hill, New York, 1981.
- [13] W. G. Hoover. Canonical dynamics: Equilibrium phase-space distribution. *Phys. Rev. A*, 31(3):1695–1697, 1985.
- [14] W. F. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *J. Mol. Graphics*, 14:33–38, 1996.
- [15] J. A. Izaguirre, S. Reich, and R. D. Skeel. Longer time steps for molecular dynamics. *J. Chem. Phys.*, 110(19):9853–9864, 1999.
- [16] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gurusoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten. NAMD2: Greater scalability for parallel molecular dynamics. *J. Comput. Phys.*, 151:283–312, 1999.
- [17] Q. Ma and J. A. Izaguirre. Targeted mollified impulse — a multiscale stochastic integrator for long molecular dynamics simulations. *Multiscale Model. Simul.*, 2(1):1–21, 2003.
- [18] Matplotlib. Matlab style python plotting. <http://matplotlib.sourceforge.net/>, 2005.
- [19] T. Matthey, T. Cickovski, S. S. Hampton, A. Ko, Q. Ma, M. Nyerges, T. Raeder, T. Slabach, and J. A. Izaguirre. PROTO-MOL: An object-oriented framework for prototyping novel algorithms for molecular dynamics. *ACM Trans. Math. Softw.*, 30(3):237–265, 2004.
- [20] M. Mezei. Evaluation of the adaptive umbrella sampling method. *Molecular Simulation*, 3:301–313, 1989.
- [21] S. Nosé. A unified formulation of the constant temperature molecular dynamics methods. *J. Chem. Phys.*, 81(1):511–519, 1984.
- [22] NumPy. Numerical python libraries. <http://numeric.scipy.org>, 2005.
- [23] PMV. Python molecular viewer. <http://www.scripps.edu/~sanner/python/pmv/webpmv.html>, 2001.
- [24] PyMPI. An introduction to parallel python using mpi. <http://sourceforge.net/projects/pympi/>, 2000.
- [25] Python. Python programming language, 2006. <http://www.python.org>.
- [26] W. Ren and E. Vanden-Eijnden. Finite temperature string method for the study of rare events. *J. Chem. Phys.*, 109(6688–6693), 2005.
- [27] W. Ren, E. Vanden-Eijnden, P. Maragakis, and W. E. Transition pathways in complex systems: Application of the finite-temperature string method to the alanine dipeptide. *J. Chem. Phys.*, 123(134109), 2005.
- [28] T. Schlick. *Molecular Modeling and Simulation - An Interdisciplinary Guide*. Springer-Verlag, New York, NY, 2002.
- [29] M. L. Scott. *Programming Language Pragmatics*. Academic Press, San Diego, 2000.
- [30] R. D. Skeel and J. A. Izaguirre. An impulse integrator for Langevin dynamics. *Mol. Phys.*, 100(24):3885–3891, 2002.
- [31] R. D. Skeel, I. Tezcan, and D. J. Hardy. Multiple grid methods for classical molecular dynamics. *J. Comp. Chem.*, 23(6):673–684, 2002.
- [32] W. B. Streett, D. J. Tildesley, and G. Saville. Multiple time-step methods in molecular dynamics. *Mol. Phys.*, 35(3):639–648, 1978.
- [33] W. Swope, H. Andersen, P. Berens, and K. Wilson. A computer simulation method for the calculation of equilibrium constants for the formation of physical clusters of molecules: application to small water clusters. *J. Chem. Phys.*, 76:637–649, 1982.
- [34] W. F. van Gunsteren and H. J. C. Berendsen. Algorithms for macromolecular dynamics and constraint dynamics. *Mol. Phys.*, 34(5):1311–1327, 1977.