NIH Public Access
Author Manuscript
J Chem Theory Comput. Author manuscript; available in PMC 2014 January 08.

Published in final edited form as:
J Chem Theory Comput. 2013 January 8; 9(1): 461–469. doi:10.1021/ct300857j.

OpenMM 4: A Reusable, Extensible, Hardware Independent Library for High Performance Molecular Simulation

Peter Eastman[1], **Mark S. Friedrichs**[1], **John D. Chodera**[2], **Randall J. Radmer**[3], **Christopher M. Bruns**[4], **Joy P. Ku**[1], **Kyle A. Beauchamp**[5], **Thomas J. Lane**[6], **Lee-Ping Wang**[6], **Diwakar Shukla**[6], **Tony Tye**[7], **Mike Houston**[7], **Timo Stich**[8], **Christoph Klein**[9], **Michael R. Shirts**[9], and **Vijay S. Pande**[6,10]

[1]Department of Bioengineering, Stanford University, Stanford, CA 94035

[2]California Institute for Quantitative Biosciences (QB3), University of California, Berkeley, CA 94720

[3]SLAC National Accelerator Laboratory, Menlo Park, CA 94025

[4]Howard Hughes Medical Institute, Janelia Farm Research Campus, Ashburn, VA 20147

[5]Biophysics Program, Stanford University, Stanford, CA 94035

[6]Department of Chemistry, Stanford University, Stanford, CA 94035

[7]Advanced Micro Devices, Sunnyvale, CA 94088

[8]NVIDIA GmbH, Wuerselen, Germany

[9]Department of Chemical Engineering, University of Virginia, Charlottesville, VA 22904

[10]Department of Computer Science, Stanford University, Stanford, CA 94035

Abstract

OpenMM is a software toolkit for performing molecular simulations on a range of high performance computing architectures. It is based on a layered architecture: the lower layers function as a reusable library that can be invoked by any application, while the upper layers form a complete environment for running molecular simulations. The library API hides all hardware-specific dependencies and optimizations from the users and developers of simulation programs: they can be run without modification on any hardware on which the API has been implemented. The current implementations of OpenMM include support for graphics processing units using the OpenCL and CUDA frameworks. In addition, OpenMM was designed to be extensible, so new hardware architectures can be accommodated and new functionality (e.g., energy terms and integrators) can be easily added.

1 Introduction

Molecular mechanics simulation is a powerful and widely used technique for studying the behavior of biological macromolecules. Many software applications exist for performing these simulations. Although these applications are powerful and feature rich, nearly all of them share some important limitations:

1. They are not designed for reusability. They are intended to be self-contained applications, and the code in them can only be used as part of those applications. If someone wishes to add molecular simulation features to a new application, it is very difficult to adapt any of the existing codes to the purpose.
2. Many of them have very limited extensibility. Molecular simulation is an active field of research, and new methods are constantly being developed. Implementing those new methods in most of the widely used applications is very challenging. It requires

$watermark-text
$watermark-text
$watermark-text

$watermark-text

$watermark-text

$watermark-text

a deep understanding of the internals of the application code. The process is poorly documented, and is impeded by the fact that the code was not originally designed with extensibility in mind.

3. Many of the most widely used codes are not available under open source licenses. This further limits the ability of researchers to extend or reuse them. For example, AMBER[1], CHARMM[2], NAMD[3], and Desmond[4] are all covered by proprietary licenses that restrict users' ability to reuse the code for other purposes, or to modify it and share those modifications with others.

These limitations can create bottlenecks for the molecular simulation community. For a researcher developing new algorithms, prototyping their ideas inside an existing simulation code is typically unnecessarily difficult and time consuming. Once an algorithm is shown to work, it may take years to become available in all the major applications. Moreover, it is frequently impractical (and in many cases, not permitted by the license) for anyone but the core developers of an application to add the new feature and make it available to the community. Even after one application adds the feature, that does not bring any other application closer to having it; their architectures are different enough that most features must be implemented independently for every application.

In this paper, we describe OpenMM, a molecular simulation code designed to address these challenges. It is architected from the ground up to be extensible and reusable, while still permitting very high performance. At its core, it is not just an application; it is a library that can be easily used by any application. It is based around a modular plugin architecture allowing new features to be added without modifying the OpenMM library itself. Finally, it is available under a permissive open source license allowing it to be used for any purpose, including commercial applications.

1.1 Goals

OpenMM was designed with the following goals in mind:

1. Simplicity and ease of use—OpenMM aims to make it very easy for programmers to add molecular simulation features to their applications. This means having a simple application programming interface (API) that is easy to learn and understand, and that requires minimal code to use. It also means that OpenMM should be easy to use *correctly*. The API should minimize the opportunities for programmers to make mistakes and guide them toward making good choices.

2. Hardware independence—Molecular simulations are commonly run on many types of hardware, including conventional processors, supercomputing clusters, graphics processing units (GPUs)[5–8], and dedicated special purpose processors.[9] Technology continues to change quickly, and it is difficult to predict what hardware platforms will be important to support as little as five years in the future.

OpenMM acts as a hardware abstraction layer, similar to LAPACK for linear algebra or OpenGL for graphics. It provides a hardware independent API for users to call from their programs. Implementations of that API can be written for many different types of hardware, and any program that uses OpenMM can run efficiently and without modification on any of those hardware platforms—even ones that did not exist yet when the program was written.

3. Extensibility—OpenMM provides a large number of simulation features (molecular force fields, integration methods, etc.), but it cannot hope to include every possible feature any user might want. Instead, it is designed to be extensible so users can add new features without

$watermark-text

$watermark-text

$watermark-text

modifying the OpenMM library itself. This allows users to implement the features they need and, if they wish, distribute them to the molecular simulation community. Any program that uses OpenMM can easily add support for those features without needing to reimplement them.

4. Performance—Whatever other benefits a simulation code offers, it will not become widely used if its performance is not competitive with other applications. OpenMM is designed to permit efficient implementation on a wide variety of hardware platforms. This requires careful thought in the design of the API to avoid making assumptions about data locality, the latency of operations, or other features that vary between platforms. Otherwise, one could easily create an API that was impossible to implement efficiently on certain types of hardware. OpenMM strives to avoid these pitfalls. It includes very fast GPU implementations of all its standard features, giving users instant access to high performance simulation code.

1.2 Architecture and Features

OpenMM is based on a modular architecture. It consists of the following major components:

1. The core OpenMM library. This defines the API for working with OpenMM. This should be thought of as an interface: it defines what calculations may be done, but does not itself perform those calculations. It is written in C++, and also includes versions of the API for C, Fortran, and Python.
2. A set of “platforms” that implement the API on particular types of hardware. Three platforms are included with OpenMM: reference, OpenCL, and CUDA. Others may be added by user-supplied plugins.
3. An “application layer” that provides routines for loading and saving files, building molecular descriptions from force fields, etc. It allows OpenMM to be used as a full application, not just a library.
4. Plugins that implement additional features.

Each of these components is described in more detail below.

The most recent version of OpenMM at the time of this writing is 4.1.1, and the descriptions contained below correspond to this version. Many of these features were present in earlier versions, but some important features were added only recently. Most importantly, the application layer was first introduced in version 4.0, and custom integrators were added in 4.1. The AMOEBA force field was first added in version 3.0. Custom forces have existed since the very first stable release, but the set of custom force types has grown steadily over time.

2 Standard features and platforms

2.1 Core Features

The OpenMM core provides the standard covalent and noncovalent interactions used in most biomolecular force fields. Two implicit solvent models are also included in the OpenMM core. The covalent energy terms implemented in OpenMM include the standard harmonic bond and angle energies and the Ryckaert-Bellemans and periodic torsion energies. The CHARMM cross-term energy correction map, CMAP torsion, is also available.[10] The CMAP torsion term couples two dihederal angles using a tabulated potential energy surface.

The nonbonded interactions implemented in the OpenMM core include the Lennard-Jones and Coulomb energies. These interactions may be calculated with or without a distance cutoff; if a cutoff is applied, it is the same for both sets of interactions.

$watermark-text

$watermark-text

$watermark-text

Lennard-Jones—The Lorentz-Bertelot combining rule is used for Lennard-Jones interactions. If a distance cutoff and periodic boundary conditions are applied, an analytical, isotropic long-range dispersion correction may be included that approximately represents the Lennard-Jones energy contribution from all interactions beyond the cutoff distance.[11]

Coulomb interactions—The Coulomb interaction can be calculated with one of five options:

1. No distance cutoff
2. Reaction field without periodic boundary conditions
3. Reaction field with periodic boundary conditions
4. Ewald summation[12]
5. Particle-mesh Ewald summation[13]

If no distance cutoff is applied, the standard Coulomb energy is included for each particle pair. If a cutoff is applied, then the Coulomb energy is modeled using the reaction-field approximation.[14] This approximation is derived by assuming molecules beyond the cutoff distance are solvent molecules and the dielectric constant is uniform.

For the Ewald summation and particle-mesh Ewald (PME) options, the user specifies the direct space cutoff and an error tolerance. The program then determines the values of the internal parameters, including the reciprocal space cutoff for Ewald and the mesh size for PME, required to calculate the forces to the specified tolerance.

Implicit solvation models—Two implicit solvent forces are available in the OpenMM core: the OBC Generalized Born model[15], and the GB/VI model developed by Labute.[16] The OBC model is comprised of a polarization term to represent the electrostatic interaction between the solute and solvent, and a surface area term to represent the free energy of solvating a neutral molecule.

The GB/VI model includes a polarization term similar to the one in the OBC model, but with the replacement of the exponent in the solute volume integral with the value −6 instead of −4. In addition, the GB/VI model employs a volumetric dispersion term to account for the nonpolar contributions to the solvation free energy.

Integrators—Five integrators are available in the OpenMM core: three fixed time step integrators (Verlet, Langevin, and Brownian), and two variable time step integrators (Verlet and Langevin). The Verlet and Langevin integrators implement the leap-frog versions of their respective algorithms.

The variable time step integrators are variants of the respective fixed time step integrators that continuously adjust the step size to keep the integration error below a user-specified tolerance.[17] This is accomplished by comparing the positions generated by Verlet integration with those that would be generated by an explicit Euler integrator, and taking the difference between them as a conservative estimate of the integration error. The step size is then adjusted to make the estimated error equal the specified error tolerance.

Thermostats and Barostats—An Andersen thermostat and Monte Carlo barostat are also provided in the OpenMM core. The thermostat couples the system to a heat bath by randomly selecting a subset of particles at the start of each time step, then setting their velocities to new values chosen from a Maxwell-Boltzmann distribution. This represents the effect of random collisions between particles in the system and particles in the heat bath.[18] This is only needed

$watermark-text

$watermark-text

$watermark-text

when using a Verlet integrator, since the Langevin and Brownian integrators provide their own temperature coupling.

The Monte Carlo barostat models the effect of constant pressure by allowing the size of the periodic box to vary during the simulation.[19–20] It does this by periodically rescaling the box, computing the change in energy, then accepting or rejecting the change based on a Metropolis criterion.

2.2 Platforms

In OpenMM, a "platform" is an implementation of all the computations defined by the API. The architecture is modular so that new platforms may be added and the client code can select at run time which platform to use for a simulation. Three platforms are included with OpenMM:

1. The Reference platform—This platform is written in C++, and serves as a reference when writing other platforms. The emphasis is on clarity and correctness, not performance; it intentionally avoids any optimization that would reduce the clarity of the code. This platform is generally too slow to be useful for production simulations, but it serves two important functions. First, it can be used as a starting point for writing other platforms, since it contains complete implementations of all required algorithms. Second, it serves as a standard for "correct" behavior. Results produced by other platforms can be compared to the reference platform to verify their correctness.

2. The OpenCL platform—This platform uses the OpenCL framework[21] for its computational kernels. This allows it to run efficiently on Nvidia and AMD GPUs, as well as on multicore CPUs. It also can parallelize computations across multiple GPUs in a single computer. In most cases, this is the fastest platform, and it is the recommended platform for most production simulations.

3. The CUDA platform—This platform uses Nvidia's CUDA framework[22] for running on GPUs. Unlike the OpenCL platform, it only supports Nvidia GPUs. It also is more limited in other respects: it cannot parallelize computations across multiple GPUs, and it does not support as many types of custom forces (described below) as the OpenCL platform.

The CUDA platform support in OpenMM is derived from an older code base which makes it very difficult to support many of the newer features of OpenMM. We are currently in the process of writing a completely new CUDA-based platform modeled after the OpenCL platform. Once it is released in a future version of OpenMM, CUDA will support all the same features as OpenCL, as well as having significantly better performance on recent Nvidia GPUs.

2.3 The Application Layer

The OpenMM core library is focused on the essential computational tasks of running a molecular simulation: computing forces and energies, and integrating the equations of motion. To actually run a simulation, however, other features are also required: loading and saving files in various standard formats, building models based on force field descriptions, etc. That is what the application layer does. It consists of a set of Python libraries that perform these functions, allowing it to be used as a complete simulation application.

Strictly speaking, of course, a set of libraries does not constitute an application, and the user is responsible for writing a true application that calls them. In practice, they are complete and simple enough that they may collectively be thought of as an "application", in which the user provides a control script. This is illustrated by Listing 1. It is a complete Python script that loads a PDB file, builds a mathematical model from it using the AMBER99SB force field and

$watermark-text

$watermark-text

$watermark-text

TIP3P water model, does a local energy minimization, and simulates it for 1 million time steps, saving a frame in PDB format every 1000 steps. All of this is done in only a dozen lines of code. The script is no more complicated than the control scripts used by many other MD applications, and its logic can be easily understood even by users with no programming experience.

Listing 1

```
from simtk.pyopenmm.app import *
from simtk.openmm import *
from simtk.unit import *
pdb = PDBFile('5dfr_solv-cube_equil.pdb')
ff = ForceField('amber99sb.xml', 'tip3p.xml')
sys = ff.createSystem(pdb.topology, nonbondedMethod=PME,
nonbondedCutoff=1*nanometer, constraints=HBonds)
integ = LangevinIntegrator (300*kelvin, 1/picosecond, 0.002*picoseconds)
sim = Simulation (pdb.topology, sys, integ)
sim.context.setPositions (pdb.positions)
sim.minimizeEnergy ( )
sim.reporters.append (PDBReporter ('output.pdb', 1000))
sim.step (1000000)
```

The advantage of this approach is its extreme flexibility. A user who simply wants to run simulations can take an example script such as this and modify it in clearly documented ways to suit their needs. More advanced users have direct access to the entire OpenMM API, as well as the full power of the Python language and libraries. For example, Listing 2 shows a complete simulated annealing algorithm implemented in only three lines:

Listing 2

```
for i in range(100):
integ.setTemperature(3*(100-i)*kelvin)
sim.step(1000)
```

3 Extensibility

Extensibility is a fundamental feature of OpenMM. We cannot hope to provide every possible feature that any user might want, so it is important that other people can add those features themselves. OpenMM is also designed to be useful for researchers developing new simulation methods. By allowing them to easily prototype and test new ideas, it serves as a powerful tool for algorithm development.

Extensibility is provided through two mechanisms: plugins and custom forces/integrators.

3.1 Plugins

A plugin is a piece of code that adds new features to OpenMM. Typically, each plugin is packaged as a dynamically linked library. At run time, OpenMM can be told to load all the plugins found in a directory and make them available to the running program.

Plugins generally fall into two categories. First, there are plugins that add new features to OpenMM. This is illustrated by the AMOEBA plugin described below. It defines a set of new

$watermark-text

$watermark-text

$watermark-text

forces that users can include in their simulations. Plugins can also define new thermostats, barostats, and integrators. Programs can use features defined by plugins just as easily as those defined by the core library. In fact, the lower levels of the architecture do not distinguish at all between core features and those provided by plugins. All features are handled identically, wherever they are defined.

Another type of plugin is ones that implement new platforms. In fact, the OpenCL and CUDA platforms are actually implemented in plugins, not in the core library. Because plugins can add new platforms, and because the choice of what platform to use is made at run time, any program that uses OpenMM has a high level of hardware independence. If a new type of hardware becomes available, a plugin can be written to support it, and all existing programs can take advantage of it with no changes required.

Three plugins that provide additional features beyond those in the core library are bundled with OpenMM: the AMOEBA force field, the free energy plugin, and the Ring Polymer Molecular Dynamics (RPMD) plugin. An outline of each plugin is given below.

AMOEBA Plugin—AMOEBA (Atomic Multipole Optimized Energetic for Biomolecular Applications) is a polarizable force field[23–24] developed in Jay Ponder's lab. The AMOEBA force field was designed with the goal of obtaining chemical accuracy on the order of 2 kJ/mol for small molecule and protein-ligand interactions. To realize this level of accuracy, both the covalent and noncovalent interactions employed in AMOEBA differ from those employed in more common force fields.

The bond-stretch, angle-bending and bond-angle interactions are based on the corresponding terms in the MM3 force field[25]; anharmonicity is included in the bond-stretching term up to fourth order and in the angle-bending term up to sixth order. A Wilson-Decius-Cross interaction[26] is used at sp^2-hybridized trigonal centers to limit out-of-plane bending. A periodic torsion term is included and is based on the typical Fourier expansion. In addition a Bell torsion term[27] is applied to dihederal angles involving two trigonal centers. Finally a CMAP torsion interaction coupling two dihedral angles is also included.

The van der Waals interactions are modeled using a buffered 14-7 functional form.[28] One significant difference between the AMOEBA implementation and others is that the van der Waals interaction site for hydrogen atoms is not at the hydrogen nucleus, but at a position on the H-X bond, where X is the hydrogen's covalent partner. The interaction site is typically about 90% of the distance from the X nucleus along the H-X bond. This modification has been shown to improve the fit to water dimer structures based on quantum mechanical calculations.

The major difference between AMOEBA and traditional force fields is in AMOEBA's treatment of the electrostatic interaction. There are two major differences associated with the electrostatics: for each atom, AMOEBA includes a permanent charge, dipole and quadrupole, in contrast to traditional force fields which typically only include a permanent charge. The permanent electrostatic components are based on ab initio calculations performed on small molecules. The second major difference between conventional force fields and AMOEBA in the treatment of the electrostatic interactions is the inclusion of electronic polarization in AMOEBA. In traditional force fields polarization is only treated in an average way. In AMOEBA polarization is modeled as induced dipoles at each atom. The induced dipoles are determined by the atomic polarizability and the electric field at the site. The atomic polarizability in turn is based on Thole's damped interaction model.[29] Because the electric field at each site includes contributions from the induced dipoles of neighboring sites, the calculation of the induced dipoles must be iterated until the induced dipoles are self-consistent.

$watermark-text

$watermark-text

$watermark-text

The iterative process must be carried out at each time step and depending on the accuracy required can be the rate-limiting step in the calculation of the forces.

The AMOEBA plugin uses particle-mesh Ewald summation to calculate the electrostatic nonbonded interaction for simulations with explicit solvent. For implicit solvent simulations, the generalized Kirkwood algorithm is used.[30]

Free Energy Plugin—The free energy plugin allows "alchemical" free energy calculations to be performed. In these types of calculations, subsets of particles are inserted, removed, or transformed into other particle types in stages. The staging is performed using a parameter λ that modulates the interactions of the particles being transformed with the rest of the system: λ=0 results in no interactions with the rest of the system, while λ=1 recovers the standard interaction terms. At intermediate values of λ during particle insertion or deletion, a "soft-core" form of the interactions is employed that allows particles to overlap without generating numerical instabilities.[31] The soft-core form of the Lennard-Jones potential is given by[32]:

$$E=4\varepsilon\lambda\left(\left[\frac{(1-\lambda)^2}{2}+\left(\frac{r}{\sigma}\right)^6\right]^{-2}-\left[\frac{(1-\lambda)^2}{2}+\left(\frac{r}{\sigma}\right)^6\right]^{-1}\right)$$

However, other choices for free energy pathways[33] can be easily implemented using custom force as described below. The free energy plugin also includes modifications to the OBC and GB/VI implicit solvent models to allow alchemical free energy calculations. For the OBC model, the contributions of transformed particles to the cavity energy term are scaled by λ. In addition, for both implicit solvent models, the contribution of transformed particles to the calculation of the Born radii is scaled by λ. Free energies can be computed by evaluating the system energy at additional values of λ and applying the Bennett acceptance ratio (BAR)[34] or the multistate Bennett acceptance ratio (MBAR)[35]. Derivatives of the potential are not possible in this framework, so thermodynamic integration (TI) cannot be used. This is not a major obstacle for most applications, as BAR and MBAR are as efficient as or more efficient than TI for standard free energy calculations[36–37].

Ring Polymer Molecular Dynamics (RPMD) Plugin—This plugin supports running simulations with the ring-polymer molecular dynamics (RPMD) method.[38] This is a method for incorporating limited quantum mechanical effects, such as tunneling and zero-point energy, into a classical molecular simulation. It does this by simulating many copies of a system at once, which are connected by harmonic springs to form a "ring polymer". The plugin implements the path integral Langevin equation (PILE) thermostat algorithm to efficiently sample the static and dynamic properties of quantum systems.[39]

3.2 Custom Forces

Custom forces are a unique feature of OpenMM that allow a wide range of novel interactions to be implemented with very little effort, yet with high speed of execution. The user provides a mathematical description of the interaction in the form of a simple text string. OpenMM parses the expression, figures out how to compute it efficiently, and then uses it exactly as if it were a standard force. All of this is done at run time, so no compilation step is needed.

Listing 3 gives an example. It uses the CustomBondForce class to implement a Morse potential. This class implements pairwise bonded interactions whose energy is an arbitrary function of the inter-atomic distance *r*. This example specifies that the energy of each bond is given by

$$E(r)=D(1-e^{a(r_0-r)})^2$$

It also specifies that D, a, and r_0 are per-bond parameters, so a value of each one will be specified for each bond.

Listing 3

```
bonds = CustomBondForce("D*(1-exp(a*(r0-r)))^2")
bonds.addPerBondParameter("D")
bonds.addPerBondParameter("a")
bonds.addPerBondParameter("r0")
```

This code is very simple, but it hides a much more complicated computation. OpenMM parses the mathematical expression, analytically differentiates it to get an expression for the force, applies a series of optimizations to both expressions, generates OpenCL code for evaluating both force and energy, synthesizes a complete kernel to loop over bonds and evaluate each one, and sends this kernel to the OpenCL driver for compilation to the GPU's native instruction set. All of this is done at run time, and takes only a fraction of a second. The custom force can then be used exactly like any other force, and runs at the full speed of the GPU hardware. The details of this process are described elsewhere.[40]

OpenMM includes several custom force classes. In addition to the pairwise bonded force shown above, there are custom forces that depend on the angle between three atoms, the dihedral angle formed by four atoms, and the pairwise distance between nonbonded atoms. There also are more complicated custom forces. These include CustomHbondForce, which can implement a wide range of hydrogen bonding potentials, and CustomGBForce, which provides a flexible framework for implementing Generalized Born implicit solvent models. Listing 4 shows the essential code required to implement the OBC implicit solvent model using the CustomGBForce. This code may look complicated, but it would take roughly 1000 lines of code to implement the same force by more conventional means. Moreover, since it expresses the nature of the calculation directly in terms of equations (rather than OpenCL or CUDA code), it is generally much more transparent and easier to modify than traditional approaches. We have used CustomGBForce to implement several other solvation models, including the Hawkins-Cramer-Truhlar[41], GBn[42], and ABSINTH[43] models, thus demonstrating its great flexibility.

Listing 4

```
gb = CustomGBForce( )
gb.addPerParticleParameter("q")
gb.addPerParticleParameter("radius")
gb.addPerParticleParameter("scale")
gb.addGlobalParameter("solventDielectric", 78.3)
gb.addGlobalParameter("soluteDielectric", 1.0)
gb.addComputedValue("I",
"""step(r+sr2-or1)*0.5*(1/L-1/U+0.25*(1/U^2-1/L^2)*(r-
sr2*sr2/r)+0.5*log(L/U)/r+C);
U=r+sr2; C=2*(1/or1-1/L)*step(sr2-r-or1);
L=max(or1, D); D=abs(r-sr2); sr2 = scale2*or2;
```


$watermark-text $watermark-text $watermark-text

$watermark-text

$watermark-text

$watermark-text

```
or1 = radius1-0.009; or2 = radius2-0.009""",
CustomGBForce.ParticlePairNoExclusions)
gb.addComputedValue("B",
"""1/(1/or-tanh(1*psi-0.8*psi^2+4.85*psi^3)/radius);
psi=I*or; or=radius-0.009""", CustomGBForce.SingleParticle)
gb.addEnergyTerm("28.3919551*(radius+0.14)^2*(radius/B)^6-
0.5*138.935456*(1/soluteDielectric-1/solventDielectric)*q^2/B",
CustomGBForce.SingleParticle)
gb.addEnergyTerm(""" –138.935456*(1/soluteDielectric-
1/solventDielectric)*q1*q2/f;
f=sqrt(r^2+B1*B2*exp(-r^2/(4*B1*B2)))""",
CustomGBForce.ParticlePairNoExclusions)
```

3.3 Custom Integrators

Just as custom forces allow arbitrary new interactions to be defined, custom integrators can implement entirely new integration algorithms. An algorithm consists of a series of steps, each defined by a user supplied mathematical expression. A step may calculate a single value, or be executed separately for each degree of freedom. It also may involve arbitrary, user defined variables. This system is flexible enough to support a wide range of deterministic and stochastic algorithms including multiple time step integrators, Metropolized integrators, Generalized Langevin Equation based integrators, and even some simple Monte Carlo methods.

Listing 5 presents a simple example of using a custom integrator to implement the velocity Verlet algorithm. It creates a new CustomIntegrator, specifies a time step of 0.001 ps, and adds three steps to the algorithm to be calculated for each degree of freedom:

$$v \leftarrow v+0.5 \cdot dt \cdot F/m$$
$$x \leftarrow x+dt \cdot v$$
$$v \leftarrow v+0.5 \cdot dt \cdot F/m$$

Notice that the force F appearing in the first step is different from the force appearing in the third step, since the atom position x has changed in between. The user does not need to specify when the force should be recomputed. OpenMM recognizes when anything changes that might affect it and recomputes it automatically.

Listing 5

```
integrator = CustomIntegrator(0.001)
integrator.addComputePerDof("v", "v+0.5*dt*f/m")
integrator.addComputePerDof("x", "x+dt*v")
integrator.addComputePerDof("v", "v+0.5*dt*f/m")
```

4 Integration with Applications

OpenMM was designed to be easily integrated into molecular dynamics programs. Two examples are the embedding of OpenMM into Gromacs[44] and the OpenMM AMOEBA plugin into TINKER.[45]

In both cases the appropriate OpenMM routines are called within the time-step loop of each program to calculate the energies and forces and perform the time-step integration. The setup

$watermark-text

$watermark-text

$watermark-text

(reading of the input files, mapping the force-field parameters to the system particles, etc.) and any run-time analysis are executed by the molecular dynamics program.

The main steps involved in embedding OpenMM into these applications are identical:

1. Map and load the parameters into the OpenMM classes using the application's internal data structures.
2. Modify the main molecular dynamics loop to call the OpenMM method to perform the time-step integration; this includes the calculation of the forces.
3. Retrieve state information (energies, particle positions, velocities, and forces) as needed.
4. Free OpenMM resources when the simulation has completed.

In both cases only the source file containing the main time-step loop of the molecular dynamics programs required modification. The changes included one-line calls to execute the four steps outlined above. A new source file was included that implemented these four steps. The last three steps consist of only a few lines of code; for both cases, the setup step is the most involved. In the case of Gromacs, the code in the new, added file used the C++ OpenMM API, whereas the C OpenMM API was used for TINKER which is written in Fortran. By making minimal changes to the original molecular dynamics code and isolating the OpenMM-specific code, the risk of introducing errors into the molecular dynamics program is reduced and maintenance of the program is much more straightforward.

5 Testing and Validation

Three types of validation of OpenMM are performed with each major release: unit tests, system tests, and direct comparison with other applications if possible. In the following, a brief description of each type of validation is given.

Unit tests are provided for each major force and integrator class and other auxiliary functions (e.g., the random number generator) and can be run by users to check that the code is working properly for their particular hardware setup. This ability is critical since GPU software and hardware is rapidly evolving and incompatibilities or unexpected results on next generation hardware are not uncommon. The unit tests exercise the basic functionality of each class to probe for problems; a separate collection of unit tests for each force and integrator is available for each of the different platforms. Most of these tests use simple model systems comprised of a small number of particles. However for the nonbonded force and implicit solvent models, some of the unit tests include relatively large systems with 'extreme' parameter values to help isolate issues.

Whereas unit tests validate the operation of small pieces of code in isolation, system tests are designed to test the library as a whole. They are performed on biomolecules (proteins, RNA, and DNA) typical of what users actually simulate, and they use realistic force fields. Systems are included that use both implicit and explicit solvent. System sizes range from 75 up to 173,181 atoms. Detailed results for these tests are provided in the OpenMM Users Guide, which the reader should consult for more information. Here we present a summary of the most important results.

The types of tests included in this category are listed below:

1. Checks for consistency between the energies and forces for the different platforms on different hardware.

$watermark-text

$watermark-text

$watermark-text

The force on each atom is computed with two different platforms, and the relative difference is computed as $2|F_1-F_2|/(|F_1|+|F_2|)$. Table 1 shows the mean difference between the Reference and OpenCL platforms, averaged over all atoms in a collection of different systems. Because the Reference platform does all calculations in double precision while OpenCL does most calculations in single precision, these numbers mostly indicate the level of error in the OpenCL calculations.

2. Checks for energy-force consistency.

   This is done as follows:

   - Compute the potential energy V_0 and force $F_0 = -\nabla V$ for a given configuration.
   - Perturb the coordinates in the direction of the force F_0 by an amount ε: $\Delta r = \varepsilon F_0/|F_0|$, where $\varepsilon \sim 10^{-2} - 10^{-6}$ nm
   - Calculate the potential energy V_1 at the perturbed configuration. To leading order, this should satisfy $(V_1-V_0)/\varepsilon \cong |F_0|$.

   Table 2 shows the maximum relative difference $[(V_1-V_0)/\varepsilon-|F_0|]/|F_0|$ over a collection of different systems for the OpenCL platform.

3. Tests of energy conservation for the Verlet integrator and thermostability for the Langevin integrator.

A 1 ns simulation is performed. For Verlet integration, it verifies that the magnitude of fluctuations in the total energy are below a cutoff. For Langevin integration, it verifies that the average instantaneous temperature equals the expected value to within a tolerance. In both cases, it also monitors all distance constrains and verifies that none is violated at any point during the simulation.

Direct comparison between energies and forces computed using OpenMM and another application.

For the OpenMM core functions, the comparisons are made with Gromacs. For the OpenMM AMOEBA plugin, the comparisons are made with TINKER. Table 3 shows the average relative difference in forces between the OpenMM Reference platform and Gromacs 4.5.3 (compiled in single precision mode) over a collection of molecules. Numbers are not shown for PME, since differences between how OpenMM and Gromacs handle cutoffs on nonbonded interactions (charge groups, shifting functions, etc.) complicate the comparison and require a more sophisticated analysis.

6 Performance

To benchmark the performance of OpenMM, we used the dihydrofolate reductase (DHFR) models taken from the Joint Amber/Charmm benchmark. This is a 159 residue protein with 2489 atoms. The version used for explicit solvent simulations included 7023 TIP3P water molecules, giving a total of 23,558 atoms.

The first set of simulations used the AMBER99SB force field.[46] We used three different methods to model the effect of solvent:

1. Implicit: Solvent was represented with an OBC-GBSA implicit solvent model. Long range interactions were cut off at 2 nm.
2. Explicit-RF: Solvent was modeled explicitly. Long range interactions were cut off at 1 nm using the reaction field method.

$watermark-text

$watermark-text

$watermark-text

3. Explicit-PME: Solvent was modeled explicitly. Long range interactions were calculated using the Particle-Mesh Ewald method. The direct space cutoff was set to 0.8 nm, and the Ewald error tolerance was set to 0.0005.

We also used two different sets of constraints within the protein, allowing for different time steps:

1. H-bonds: The lengths of bonds involving hydrogen were constrained. The time step was set to 2 fs.
2. H-angles: All bond lengths were constrained, along with angles of the form H-X-H or H-O-X. The time step was set to 4 fs.

Water molecules were fully rigid in all cases, and the constraint error tolerance was set to 10^{-4}. (For H-bonds constraints, the error tolerance has a negligible effect on performance. For H-angles, reducing it to 10^{-5} causes a measurable but still quite small decrease in performance.) All simulations used an NVT ensemble, with a Langevin thermostat to maintain a temperature of 300K. The results for both the OpenCL and CUDA platforms are shown in Table 1. The CUDA simulations were run on a single Nvidia GTX 580 GPU. For OpenCL, simulations were run on both the GTX 580 and on an AMD Radeon HD 7970.

The OpenCL platform can parallelize an explicit solvent simulation across multiple GPUs. We therefore repeated each of the above benchmarks using 1 to 4 Nvidia C2070 GPUs in parallel. The results are shown in Table 2.

The scaling with number of GPUs is much less than linear. Using up to three GPUs produces a significant speedup, but there is little benefit beyond that. For PME, using four GPUs is actually slightly slower than using three. This is primarily due to the cost of transferring data over the high latency PCIe bus.

To demonstrate the performance of custom forces, we repeated the Explicit-RF, H-bonds benchmark for the OpenCL platform, but instead of using OpenMM's built-in implementation of Coulomb and Lennard-Jones forces, we reimplemented them using the CustomNonbondedForce class. Running on a single Nvidia C2070 it achieves 23.0 ns/day, compared to 25.9 ns/day for the built-in forces, a speed penalty of only 11%. This shows the great value of custom forces: completely arbitrary functional forms, defined with only a few lines of code, can nearly match the performance of a highly optimized, hand-written implementation.

Benchmarks for the AMOEBA plugin were carried out for villin, a small protein with 596 atoms, and DHFR in both explicit and implicit solvent on an Nvidia GTX 580 GPU. The error tolerance for the calculation of the induced dipoles was 0.01 and the time step was 1 femtosecond. The results are shown in Table 3.

To measure energy conservation, we performed a 10 ns simulation of ubiquitin[47], a 1231 atom protein in OBC implicit solvent. The simulation used the OpenCL Platform, a 1 fs time step, no constraints, and no cutoff on the nonbonded interactions, and had an average temperature of 298.6 K. A linear regression to the curve of energy versus time gives an overall drift rate of 2.5 kJ/mole/ns, or 0.00027 kT/ns/dof in the more commonly reported units. Note, however, that these units are not very meaningful since energy drift does not actually scale linearly with time, temperature, or system size. Furthermore, a simple linear regression is not sufficient to fully characterize energy conservation. A thorough discussion of these issues and a more detailed study of energy conservation in OpenMM is published elsewhere.[48]

$watermark-text

$watermark-text

$watermark-text

7 Conclusions

OpenMM is a software toolkit that addresses an unmet need in the molecular simulation community. It is designed to be modular, extensible, reusable, and hardware independent. This combination of traits makes it uniquely valuable to several different types of users. Application developers have instant access to a large library of robust, high performance algorithms. Researchers developing new methods can use OpenMM's plugin architecture and custom forces to implement and test new algorithms. Once written, those algorithms can quickly be incorporated into any program that uses OpenMM. For chemists and biologists interested in running simulations, the Python based application layer is a uniquely flexible and powerful environment offering complete control over nearly every aspect of the simulation protocol.

Acknowledgments

This work was supported by Simbios via the NIH Roadmap for Medical Research Grant U54 GM072970, and by NIH grant R01-GM062868. JDC acknowledges support from a QB3-Berkeley Distinguished Postdoctoral Fellowship.

References

1. Case DA, Darden TA, Cheatham TEI, Simmerling CL, Wang J, Duke RE, Luo R, Merz KM, Pearlman DA, Crowley M, Walker RC, Zhang W, Wang B, Hayik S, Roitberg A, Seabra G, Wong KF, Paesani F, Wu X, Brozell S, Tsui V, Gohlke H, Yang L, Tan C, Mongan J, Hornak V, Cui G, Beroza P, Mathews DH, Schafmeister C, Ross WS, Kollman PA. AMBER 9. 2006
2. Brooks BR, Brooks I, CL, Mackerell J, AD, Nilsson L, Petrella RJ, Roux B, Won Y, Archontis G, Bartels C, Boresch S, Caflisch A, Caves L, Cui Q, Dinner AR, Feig M, Fischer S, Gao J, Hodoscek M, Im W, Kuczera K, Lazaridis T, Ma J, Ovchinnikov V, Paci E, Pastor RW, Post CB, Pu JZ, Schaefer M, Tidor B, Venable RM, Woodcock HL, Wu X, Yang W, York DM, Karplus M. CHARMM: The Biomolecular Simulation Program. J Comput Chem. 2009; 30:1545–1614. [PubMed: 19444816]
3. Phillips JC, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel RD, Kale L, Schulten K. Scalable molecular dynamics with NAMD. J Comput Chem. 2005; 26(16):1781–1802. [PubMed: 16222654]
4. Bowers KJ, Chow E, Xu H, Dror RO, Eastwood MP, Gregersen BA, Klepeis JL, Kolossváry I, Moraes MA, Sacerdoti FD, Salmon JK, Shan Y, Shaw DE. Scalable Algorithms for Molecular Dynamics Simulations on Commodity Clusters. Proceedings of the ACM/IEEE Conference on Supercomputing (SC06). 2006
5. Anderson JA, Lorenz CD, Travesset A. General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units. J Comput Phys. 2008; 227:5342–5359.
6. Friedrichs MS, Eastman P, Vaidyanathan V, Houston M, LeGrand S, Beberg AL, Ensign DL, Bruns CM, Pande VS. Accelerating molecular dynamic simulation on graphics processing units. J Comput Chem. 2009; 30(6):864–872. [PubMed: 19191337]
7. Stone JE, Phillips JC, Freddolino PL, Hardy DJ, Trabuco LG, Schulten K. Accelerating molecular modeling applications with graphics processors. J Comput Chem. 2007; 28(16):2618–2640. [PubMed: 17894371]
8. van Meel JA, Arnold A, Frenkel D, Portegies Zwart SF, Belleman RG. Harvesting graphics power for MD simulations. Mol Simul. 2008; 34(3):259–266.
9. Shaw, DE.; Deneroff, MM.; Dror, RO.; Kuskin, JS.; Larson, RH.; Salmon, JK.; Young, C.; Batson, B.; Bowers, KJ.; Chao, JC.; Eastwood, MP.; Gagliardo, J.; Grossman, JP.; Ho, CR.; Ierardi, DJ.; Istv, #225.; Kolossvnry; Klepeis, JL.; Layman, T.; McLeavey, C.; Moraes, MA.; Mueller, R.; Priest, EC.; Shan, Y.; Spengler, J.; Theobald, M.; Towles, B.; Wang, SC. Anton a special-purpose machine for molecular dynamics simulationIn. Proceedings of the 34th annual international symposium on Computer architecture; San Diego, California, USA: ACM; 2007. p. 1-12.
10. Alexander D, MacKerell J. Empirical Force Fields for Biological Macromolecules: Overview and Issues. J Comput Chem. 2004; 25(13):1584–1604. [PubMed: 15264253]

$watermark-text
$watermark-text
$watermark-text

11. Shirts MR, Mobley DL, Chodera JD, Pande VS. Accurate and Efficient Corrections for Missing Dispersion Interactions in Molecular Simulations. J Phys Chem B. 2007; 111(45):13052–13063. [PubMed: 17949030]
12. Toukmaji AY, Board JA Jr. Ewald summation techniques in perspective: a survey. Comput Phys Commun. 1996; 95:73–92.
13. Essmann U, Perera L, Berkowitz ML, Darden T, Lee H, Pedersen LG. A smooth particle mesh Ewald method. J Chem Phys. 1995; 103(19):8577–8593.
14. Tironi IG, Sperb R, Smith PE, van Gunsteren WF. A generalized reaction field method for molecular dynamics simulations. J Chem Phys. 1995; 102(13):5451–5459.
15. Onufriev A, Bashford D, Case DA. Exploring protein native states and large-scale conformational changes with a modified generalized born model. Proteins. 2004; 55(22):383–394. [PubMed: 15048829]
16. Labute P. The generalized Born/volume integral implicit solvent model: Estimation of the free energy of hydration using London dispersion instead of atomic surface area. J Comput Chem. 2008; 29(10): 1693–1698. [PubMed: 18307169]
17. Press, WH.; Teukolsky, SA.; Vetterling, WT.; Flannery, BP. Numerical Recipes in C++. 2. Cambridge University Press; 2003.
18. Andersen HC. Molecular dynamics simulations at constant pressure and/or temperature. J Chem Phys. 1980; 72(4):2384–2393.
19. Chow KH, Ferguson DM. Isothermal-isobaric molecular dynamics simulations with Monte Carlo volume sampling. Comput Phys Commun. 1995; 91:283–289.
20. Åqvist J, Wennerström P, Nervall M, Bjelic S, Brandsdal BO. Molecular dynamics simulations of water and biomolecules with a Monte Carlo constant pressure algorithm. Chem Phys Let. 2004; 384:288–294.
21. Aaftab, M. The OpenCL Specification, version 1.0. The Khronos Group; Beaverton, OR: 2008.
22. CUDA Toolkit, version 4.1. NVIDIA; Santa Clara, CA: 2012.
23. Ren P, Ponder JW. A Consistent Treatment of Inter- and Intramolecular Polarization in Molecular Mechanics Calculations. J Comput Chem. 2002; 23:1497–1506. [PubMed: 12395419]
24. Ren P, Ponder JW. Polarizable Atomic Multipole Water Model for Molecular Mechanics Simulation. J Phys Chem B. 2003; 107:5933–5947.
25. Allinger NL, Yuh YH, Lii JH. Molecular mechanics. The MM3 force field for hydrocarbons. J Am Chem Soc. 1989; 111(23):8551–8566.
26. Wilson, EB., Jr; Decius, JC.; Cross, PC. Molecular Vibrations: The Theory of Infrared and Raman Vibrational Spectra. McGraw-Hill; New York: 1955.
27. Bell RP. Bond Torsion in the Vibrations of the Benzene Molecule. Trans Faraday Soc. 1945; 41:293–295.
28. Halgren TA. The representation of van der Waals (vdW) interactions in molecular mechanics force fields: potential form, combination rules, and vdW parameters. J Am Chem Soc. 1992; 114(20):7827–7843.
29. Thole BT. Molecular polarizabilities calculated with a modified dipole interaction. Chem Phys. 1981; 59(3):341–350.
30. Schnieders MJ, Ponder JW. Polarizable Atomic Multipole Solutes in a Generalized Kirkwood Continuum. J Chem Theory Comput. 2007; 3:2083–2097.
31. Beutlera TC, Marka AE, Schaikb RCv, Gerberc PR, Gunsteren WFv. Avoiding singularities and numerical instabilities in free energy calculations based on molecular simulations. Chem Phys Let. 1994; 222(6):529–539.
32. Shirts MR, Pande VS. Solvation free energies of amino acid side chain analogs for common molecular mechanics water models. J Chem Phys. 2005; 132:134508. [PubMed: 15847482]
33. Pham TT, Shirts MR. Identifying low variance pathways for free energy calculations of molecular transformations in solution phase. J Chem Phys. 2011; 135(3)
34. Bennett CH. Efficient estimation of free energy differences from Monte Carlo data. J Comput Phys. 1976; 22(2):245–268.

$watermark-text

$watermark-text

$watermark-text

35. Shirts MR, Chodera JD. Statistically optimal analysis of samples from multiple equilibrium states. J Chem Phys. 2008; 129(12)
36. Shirts MR, Pande VS. Comparison of efficiency and bias of free energies computed by exponential averaging, the Bennett acceptance ratio, and thermodynamic integration. J Chem Phys. 2005; 122 (14)
37. Paliwal H, Shirts MR. A Benchmark Test Set for Alchemical Free Energy Transformations and Its Use to Quantify Error in Common Free Energy Methods. J Chem Theory Comput. 2011; 7(12):4115–4134.
38. Craig IR, Manolopoulos DE. Quantum statistics and classical mechanics: Real time correlation functions from ring polymer molecular dynamics. J Chem Phys. 2004; 121(8):3368–3373. [PubMed: 15303899]
39. Ceriotti M, Parrinello M, Markland TE, Manolopoulos DE. Efficient stochastic thermostatting of path integral molecular dynamics. J Chem Phys. 2010; 133(12)
40. Eastman, P.; Pande, V. Accelerating Development and Execution Speed with Just-in-Time GPU Code Generation. In: Hwu, W-m, editor. GPU Computing Gems Jade Edition. Morgan Kaufmann; 2011. p. 399-407.
41. Hawkins GD, Cramer CJ, Truhlar DG. Pairwise solute descreening of solute charges from a dielectric medium. Chem Phys Let. 1995; 246(1–2):122–129.
42. Mongan J, Simmerling C, McCammon JA, Case DA, Onufriev A. Generalized Born model with a simple, robust molecular volume correction. J Chem Theory Comput. 2007; 3(1):156–169. [PubMed: 21072141]
43. Vitalis A, Pappu RV. ABSINTH: A New Continuum Solvation Model for Simulations of Polypeptides in Aqueous Solutions. J Comput Chem. 2008; 30(5):673–699. [PubMed: 18506808]
44. Hess B, Kutzner C, van der Spoel D, Lindahl E. GROMACS 4: Algorithms for Highly Efficient, Load-Balanced, and Scalable Molecular Simulation. J Chem Theory Comput. 2008; 4:435–447.
45. Ponder, JW. TINKER - Software Tools for Molecular Design, version 6.0. 2011.
46. Hornak V, Abel R, Okur A, Strockbine B, Roitberg A, Simmerling C. Comparison of multiple Amber force fields and development of improved protein backbone parameters. Proteins. 2006; 65:712–725. [PubMed: 16981200]
47. Vijay-Kumar S, Bugg CE, Cook WJ. Structure of ubiquitin refined at 1.8 A resolution. J Mol Biol. 1987; 194(3):531–544. [PubMed: 3041007]
48. Eastman P, Pande VS. Energy Conservation and Simulation Accuracy. Submitted for publication. 2012

$watermark-text

$watermark-text

$watermark-text

Table 1

Force	Mean Relative Difference
Harmonic bonds	$1.98 \cdot 10^{-5}$
Harmonic angles	$1.15 \cdot 10^{-5}$
Periodic torsions	$1.51 \cdot 10^{-5}$
Ryckaert-Bellemans torsions	$3.88 \cdot 10^{-6}$
Nonbonded (no cutoff)	$9.59 \cdot 10^{-7}$
Nonbonded (PME)	$3.68 \cdot 10^{-6}$
OBC implicit solvent	$3.04 \cdot 10^{-6}$

$watermark-text

$watermark-text

$watermark-text

Table 2

Force	Max Relative Difference
Harmonic bonds	$7.51 \cdot 10^{-3}$
Harmonic angles	$4.19 \cdot 10^{-3}$
Periodic torsions	$1.44 \cdot 10^{-2}$
Ryckaert-Bellemans torsions	$2.89 \cdot 10^{-3}$
Nonbonded (no cutoff)	$1.10 \cdot 10^{-3}$
Nonbonded (PME)	$3.89 \cdot 10^{-3}$
OBC implicit solvent	$6.18 \cdot 10^{-4}$

$watermark-text

$watermark-text

$watermark-text

Table 3

Force	Mean Relative Difference
Harmonic bonds	$1.66 \cdot 10^{-4}$
Harmonic angles	$6.35 \cdot 10^{-5}$
Periodic torsions	$3.70 \cdot 10^{-5}$
Nonbonded (no cutoff)	$6.13 \cdot 10^{-7}$
OBC implicit solvent	$3.82 \cdot 10^{-6}$

$watermark-text

$watermark-text

$watermark-text

Table 1

Performance of benchmark simulations run on a single Nvidia GTX 580 or AMD Radeon HD 7970 GPU. Results are in ns/day.

	CUDA	OpenCL (Nvidia)	OpenCL (AMD)
Implicit, H-bonds	127	127	134
Implicit, H-angles	213	176	133
Explicit-RF, H-bonds	28.8	39.5	33.2
Explicit-RF, H-angles	54.5	70.9	50.1
Explicit-PME, H-bonds	18.5	25.1	24.1
Explicit-PME, H-angles	35.5	46.0	37.0

$watermark-text

$watermark-text

$watermark-text

Table 2

Performance of benchmark simulations run on 1 to 4 Nvidia C2070 GPUs. Results are in ns/day.

	1 GPU	2 GPUs	3 GPUs	4 GPUs
Explicit-RF, H-bonds	25.9 (1.0)	40.2 (1.55)	48.5 (1.87)	52.3 (2.02)
Explicit-RF, H-angles	47.6 (1.0)	69.4 (1.46)	80.8 (1.70)	87.9 (1.85)
Explicit-PME, H-bonds	16.5 (1.0)	27.1 (1.64)	30.1 (1.83)	29.8 (1.81)
Explicit-PME, H-angles	30.9 (1.0)	49.7 (1.61)	55.5 (1.80)	54.9 (1.78)

The value in parentheses is the speedup relative to a single GPU.

$watermark-text

$watermark-text

$watermark-text

Table 3

Performance of AMOEBA benchmarks on an Nvidia GTX 580 GPU.

Explicit solvent:		
	Atoms	**ns/day**
villin	3182	2.68
DHFR	23558	0.493

Implicit solvent:		
	Atoms	**ns/day**
villin	596	7.12
DHFR	2489	1.53