MIST: A Simple and Efficient Molecular Dynamics Abstraction Library for Integrator Development

MIST: A Simple and Efficient Molecular Dynamics Abstraction Library for Integrator Development

Iain Bethune[a,f,*], Ralf Banisch[b], Elena Breitmoser[c], Antonia B. K. Collis[c], Gordon Gibb[c], Gianpaolo Gobbo[d], Charles Matthews[e], Graeme J. Ackland[f], Benedict J. Leimkuhler[g]

[a]*STFC Hartree Centre, Sci-Tech Daresbury, Warrington, WA7 6UE, UK*
[b]*Department of Mathematics and Computer Science, Freie Universität Berlin, Arnimallee 6, 14195 Berlin, Germany*
[c]*EPCC, The University of Edinburgh, James Clerk Maxwell Building, Peter Guthrie Tait Road, Edinburgh, EH9 3FD, UK*
[d]*Department of Chemical Engineering, Massachusetts Institute of Technology, 77 Massachusetts Avenue, Cambridge, MA 02139, USA*
[e]*Department of Statistics, University of Chicago, S. Ellis Avenue, Chicago, IL 60637, USA*
[f]*School of Physics and Astronomy, The University of Edinburgh, James Clerk Maxwell Building, Peter Guthrie Tait Road, Edinburgh, EH9 3FD*
[g]*School of Mathematics, The University of Edinburgh, James Clerk Maxwell Building, Peter Guthrie Tait Road, Edinburgh, EH9 3FD*

Abstract

We present MIST, the Molecular Integration Simulation Toolkit, a lightweight and efficient software library written in C++ which provides an abstract interface to common molecular dynamics codes, enabling rapid and portable development of new integration schemes for molecular dynamics. The initial release provides plug-in interfaces to NAMD-Lite, GROMACS and Amber, and includes several standard integration schemes, a constraint solver, temperature control using Langevin Dynamics, and two tempering schemes. We describe the architecture and functionality of the library and the C and Fortran APIs which can be used to interface additional MD codes to MIST.

We show, for a range of test systems, that MIST introduces negligible overheads for serial, shared-memory parallel, and GPU-accelerated cases, except for Amber where the native integrators run directly on the GPU itself.

*Corresponding author.
E-mail address: iain.bethune@stfc.ac.uk

As a demonstration of the capabilities of MIST, we describe a simulated tempering simulation used to study the free energy landscape of Alanine-12 in both vacuum and detailed solvent conditions.

Keywords: Molecular Dynamics Software; Enhanced Sampling; Simulated Tempering; Langevin Dynamics

---

PROGRAM SUMMARY

Program Title: MIST - Molecular Integration Simulation Toolkit
Licensing provisions: BSD 2-clause
Programming language: C++ (C and Fortran interfaces)
Nature of problem:
Production Molecular Dynamics codes have become increasingly complex, making it difficult to implement new functionality, especially algorithms that modify the core MD integration loop. This places a barrier in the way of new algorithms making their way from theory to implementation.
Solution method:
MIST provides a simplified abstract interface for integrator developers that may be interfaced via source-code patches and a library API to a variety of MD codes, with minimal loss of performance.
Restrictions and Unusual features:
MIST interfaces only to specific versions of MD codes: Amber 14, Gromacs 5.0.2 and NAMD-Lite 2.0.3
Comments:
MIST is freely available from `https://bitbucket.org/extasy-project/mist`.

1. Background

Molecular Dynamics with classical force-fields has proved to be an extraordinarily successful method for studying dynamical processes as well as sampling the conformational space of complex macromolecules (see [1] for a recent review). This success is largely due to advances in four directions; improving accuracy of force-fields, developing faster and more scalable force calculations, increasing computational power of high performance computing systems, and advanced sampling algorithms such as metadynamics [2], replica-exchange MD [3] and parallel tempering [4]. A number of highly-optimised MD packages such as NAMD [5] and GROMACS [6] have been developed which implement a range of different force-fields, are able to run on

a range of commodity (x86 CPU clusters and GPUs) and special-purpose [7] hardware and represent many hundreds of person-years of effort. All of this functionality and performance comes at a cost in terms of code complexity, and even if an MD code is open-source, in practice it is difficult for developers to add significant new features without close collaboration with the main developers of the code.

The result is that the core algorithms used for MD timestepping evolve slowly. Typical integration schemes are based on velocity Verlet or leapfrog integration, combined with one of several common thermo- or baro-stats [8, 9, 10, 11]. Recent innovation has centered on higher level methods for promoting space exploration [12] or modifying the potential energy surface to lower barriers between metastable states [13]. We argue that there is "room at the bottom"[1] for innovative methods which modify the core integration step to access larger time steps and/or improved sampling accuracy (e.g. [14, 15, 16, 17, 18, 19]) which have not yet been implemented in any 'production' MD codes.

The *status quo* is a catch-22 for applied mathematicians: if new algorithms cannot be easily incorporated into widely used MD packages, then it is impossible to demonstrate their benefits on complex systems of practical interest. If such demonstrations are not available, there will be little interest from the MD user community, and there is no incentive for MD package developers to implement the methods; in many cases algorithms are left 'on the shelf' for long periods.

Our solution to this conundrum is MIST–the Molecular Integration Simulation Toolkit. MIST is a software library (available from `https://bitbucket.org/extasy-project/mist`) which can be easily interfaced to a variety of MD codes (currently GROMACS, Amber [20], NAMD-Lite [21] and Forcite [22]) and which provides an abstract interface to the state of the system. This enables integration algorithms to be programmed without concern for the complexities of a typical MD code, with low performance overhead. We describe the architecture of the MIST library, the Application Programmer's Interface (API) which plugs in to existing codes, explain how to implement an integrator, and illustrate the performance on a range of computational benchmark tests.

Although we use the term "integrator" here to describe timestepping

[1] With apologies to Richard P. Feynman.

procedures used in MD, MIST is deliberately not restrictive regarding the types of equations that can be simulated; indeed the example we present at the end of this article implements simulated tempering, a complex enhanced sampling strategy, within MIST. While MIST currently supports a range of classical MD codes, the design of MIST is flexible enough that it could also be used for *ab initio* MD based on the Born-Oppenheimer approximation.

1.1. Related Work

PLUMED[23], is similar in design to MIST in that it is a software library that interfaces to a range of MD codes via API calls (which may be inserted using source-code patches). While PLUMED is widely used and at present supports more MD codes than MIST, it only allows the calculated MD forces to be modified and does not provide read/write access to the atomic positions and velocities. Thus, it is less flexible than MIST and facilitates a much smaller set of MD algorithms.

A simplified MD program (such as NAMD-Lite removes much of the complexity of a production MD code, making it easy to modify. However, this results in a loss of functionality (e.g. forcefield support, analysis tools, properties calculations) and performance–restricting the scale of problems which can be tackled.

OpenMM [24] is a toolbox for building MD applications which is designed to be extensible at the source-code level, while being portable to a range of CPU and GPU hardware. The `CustomIntegrator` interface is flexible and provides a Python API to allow declaration of (for example) variables which should be computed for each degree of freedom. However, we argue that this API approach results in code which is less clear and intuitive than the way an integrator is specified in MIST. Moreover, the idea in MIST is that the mathematical integration algorithm framework is independent of the MD engine (which encompasses force-field evaluation, boundary conditions, etc.); this allows comparison and cross-validation of results using alternative molecular software systems. Ultimately MIST offers improved interoperability with a broad range of existing codes and force-fields.

2. MIST Architecture

The design of MIST balances three main objectives: providing an abstraction of the state of a molecular dynamics simulation which is sufficiently general to allow it to be implemented in conjunction with any particular MD

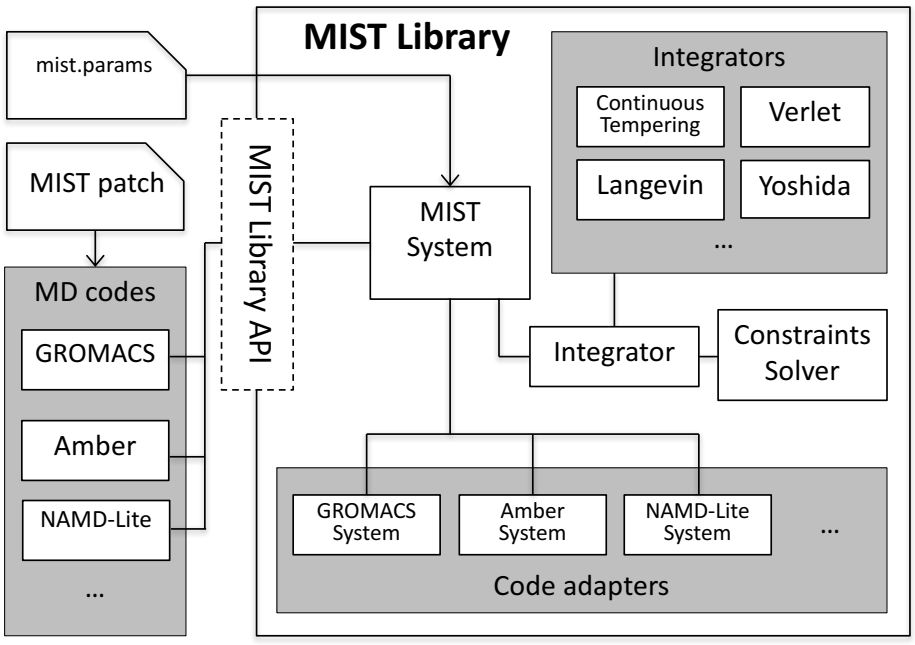


Figure 1: Schematic representation of the main components of the MIST library.

code; implementing this interface with a low performance overhead relative to standard MD codes; and providing an intuitive and expressive interface for developing integrators. Figure 1 shows how the components of the library are related. A 'host' MD code makes use of the MIST library to perform time integration of the state of the system. At the core of the MIST library is a representation of the state of the system (and a set of adaptors for specific MD codes), which can be used by developers to implement new integrator algorithms. Each of these components is discussed in detail in the following sections. Several integrators are included in the library, to serve as templates, as well as to provide new capabilities to users (see Table 1).

2.1. The MIST System

The conceptual model of the state of the system is very simple. We have a set of n point particles labeled $0..n-1$, where n is assumed to remain fixed for the duration of the simulation. Each particle has a set of properties: position, velocity, mass, kind (atomic species, typically) and force (the force acting on the particle). In addition, there are a number of global properties, such

as the cell lattice vectors (if periodic boundary conditions are employed), the total potential energy. This state is encapsulated as a C++ `System` class. For the dynamical variables (position and velocity), accessor (e.g. `GetPosition()`) and mutator (e.g. `SetPosition()` methods are provided. The other properties are read-only, and so only accessors are provided (e.g. `GetForce()`). Evaluation of forces is treated as a black-box, and a method `UpdateForces()` is provided to request the forces on each particle to be updated (usually the most expensive operation in an MD simulation). Access to a simple representation of the molecular topology is also provided: a set of b bonds labelled $0..b-1$, where each bond consists of a pair of particle indices and a fixed length, encapsulated as a lightweight `Bond` object. An important aspect of this model is that in this release of MIST we require that all of the data is stored within a single address space i.e. domain decomposition using MPI is not supported (although this is under development, see Section 6). Shared memory parallelisation such as OpenMP is permitted in the host code, and indeed is used within MIST.

The `System` provides everything that is needed to implement an integrator (see Section 2.2) but requires an adaptor to implement the MD-code-independent `System` methods using the data structures present in a particular MD code. The choice of MD code is made at compile-time via arguments to the `configure` script used to drive MIST's build process. For simplicity and performance, we provide MIST with access to the raw data structures in an MD code through pointers registered with MIST by the host MD code. This allows the library API (see Section 2.3) to be remain completely code-agnostic, and the details of how those pointers are interpreted to yield useful data is encapsulated with the code-specific `System` adaptor classes. For example, GROMACS (by default) stores data as arrays of single-precision floating point whereas Amber and NAMD-Lite use double-precision, and GROMACS stores the inverse masses of particles, rather than the masses themselves. These differences are hidden from the user by the `System` abstraction.

2.2. Integrators

In MIST, an `Integrator` is an abstract class which has a single method which must be implemented by any sub-class: `void Step(double dt)`, as the name suggests, a function which implements the time integration of the system state from t to $t + dt$, according to some algorithm. To add a new integration algorithm to the library, a developer need only create a new class

which inherits from `Integrator` and implements the `Step` method. A number of convenience functions are also included in the base class which simplify coding, for example a velocity Verlet integrator for the NVE ensemble is as simple as:

```
void VerletIntegrator::Step(double dt)
{
    // Velocity half-step
    VelocityStep(0.5 * dt);

    // Position full step
    PositionStep(dt);

    system->UpdateForces();

    // Velocity half-step
    VelocityStep(0.5 * dt);
}
```

More complex algorithms can be implemented by directly updating individual particle properties using methods of the `System` class. For example, the stochastic part of our Langevin dynamics integrator is implemented as (`c1` and `c3` are double precision floating-point constants, `v` is a variable of the lightweight `Vector3` type, and `rnd[tid]` is a (thread-local) instance of our random number generator):

```
for (int i = 0; i < system->GetNumParticles(); i++)
{
    v = system->GetVelocity(i);
    sqrtinvm = system->GetInverseSqrtMass(i); // 1/sqrt(m)
    v.x = c1 * v.x + sqrtinvm * c3 * rnd[tid]->random_gaussian();
    v.y = c1 * v.y + sqrtinvm * c3 * rnd[tid]->random_gaussian();
    v.z = c1 * v.z + sqrtinvm * c3 * rnd[tid]->random_gaussian();
    system->SetVelocity(i, v);
}
```

Most integrators advance the entire state of the system from time t to $t + dt$. However, a class of algorithms such as Verlet integration in the 'leapfrog'

formulation operate assuming the velocities to be offset by $dt/2$ from the positions at the start of each step. In MIST, such an integrator must be labeled with the 'feature flag' `MIST_FEATURE_POS_VEL_OFFSET_PLUS_HALF_DT`, to ensure that account of this is taken by the host MD code (for example, computing kinetic energy based on averages over two steps). The feature flags are used to signal any special requirements that the integrator might place on the host code, for example that access to individual components of the force-field is required (`MIST_FEATURE_FORCE_COMPONENTS`). This allows integrators to be coded quite generally and functionality be added incrementally to the MD code adaptors, with a check performed at startup to see if the features of the selected integrator are supported by the code.

Selecting an integrator and setting parameters are done through an input file `mist.params` which contains a list of (case-insensitive) key-value pairs. A separate file is used so that integrator settings are code-independent, whereas run control settings such as the number of time steps, when to write trajectory output etc. are managed by the usual input file(s) of the host code. A minimal input file to use velocity Verlet integration in the NVE ensemble would be:

```
integrator verlet
```

A slightly more complex example, to run Langevin NVT dynamics at 300K would be:

```
integrator langevin
langtemp 300 # Target temperature, in K
langfriction 1.0 # Friction parameter gamma, in ps^(-1)
```

A full list of integrators and possible parameters (including default values and units) is available online at `https://bitbucket.org/extasy-project/mist/wiki/MIST%20Integrators` and is summarised in Table 1. Since the lattice parameters are fixed in this release of MIST rather than treated as dynamical variables, at present it is not possible to implement constant-pressure MD schemes - this is under development for a future release (see Section 6).

2.3. MIST Library API

MD codes interact with the MIST library through a simple C or Fortran 90 API, which is designed to be general enough to interface to a wide range

Keyword	Supports Constraints	Ensemble	Description
`verlet`	Yes	NVE	Velocity Verlet
`leapfrog`	Yes	NVE	Verlet Leapfrog
`langevin`	Yes	NVT	Langevin Dynamics with BAOAB or ABOBA splitting method [14]
`yoshida4`	No	NVE	Symplectic 4th order integrator [25]
`yoshida8`	No	NVE	Symplectic 8th order integrator [25]
`tempering`	Yes	-	Continuous Tempering - enhanced sampling with continously varying temperature [15]
`tamd`	No	-	Temperature Accelerated MD - Langevin dynamics with an additional thermostat coupled to two backbone dihedral angles
`tempering_tamd`	No	-	TAMD where additional degrees of freedom are heated via the Continuous Tempering algorithm
`simulated_tempering`	Yes	-	Simulated Tempering with on-the-fly weight determination [26, 27] using Langevin thermostat

Table 1: List of current MIST integrators and algorithms

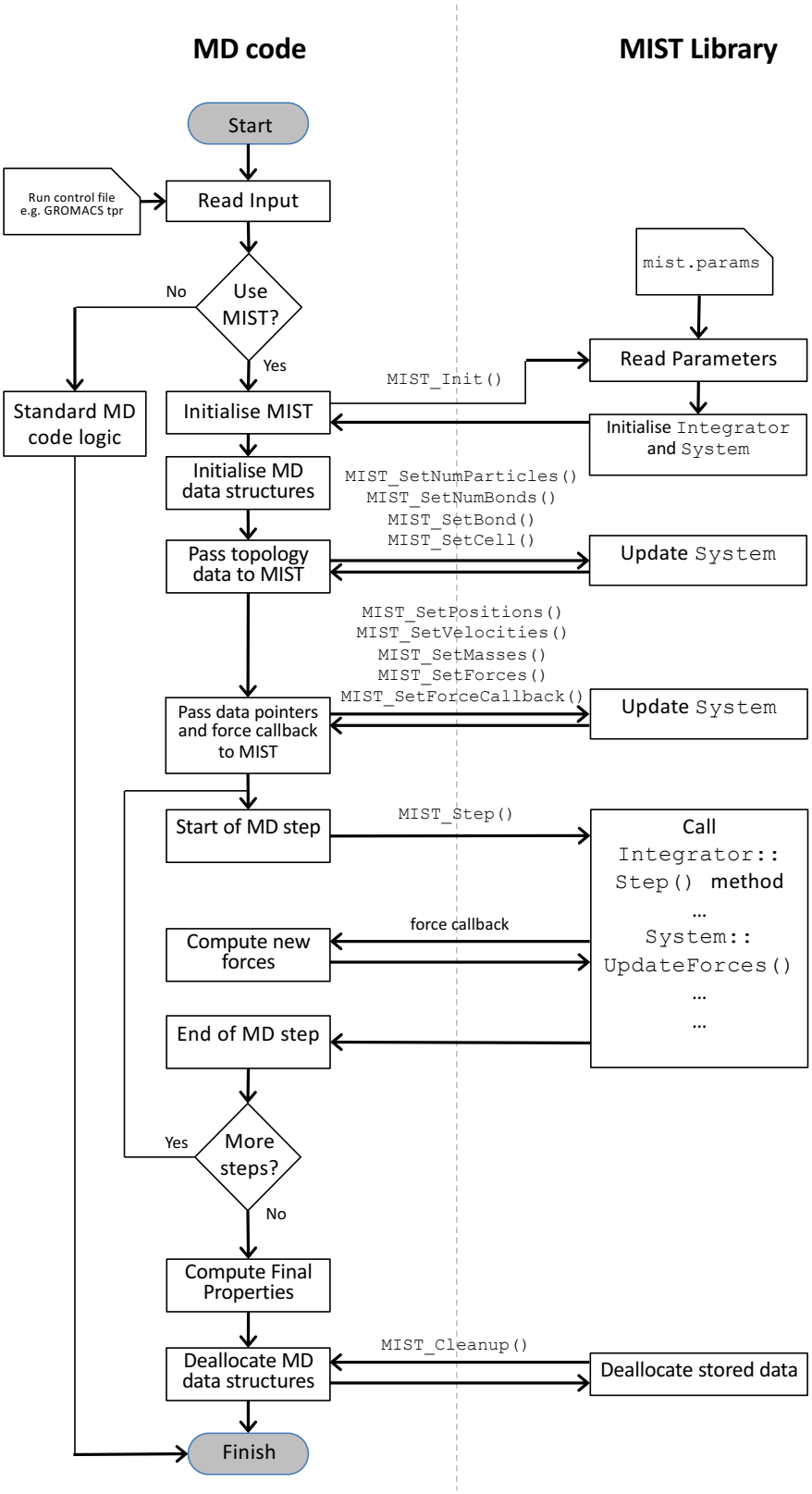


Figure 2: Control flow in an MD code using the MIST library

of possible MD codes. The C interface is declared in a header `mist.h` and the Fortran interface in a module `mist_f90`. The Fortran interface contains exactly the same functionality as its C counterpart, with the only difference being that the functions are name `MIST_F_*` rather than `MIST_*`. As well as function declarations, a range of predefined constants (the aforementioned feature flags, and error codes) are also part of the interface. MIST API calls may be added in to a host MD code either by hand, or for code versions that we support by automatically-applied source-code patches (the build process is explained in Section 3). MIST maintains an internal state and is responsible for its own memory management - a client calling the API always interacts with the same instance of the library, rather than for example having to pass an opaque handle back and forwards with each call. Full documentation of the API is provided in Doxygen format, but the key concepts and ordering of API calls are shown in Figure 2, which outlines the typical control flow between a host MD code and MIST.

It is important to note that by design, MIST extends the existing functionality of an MD code, and so if MIST is not selected as an option in the code's input configuration, it will behave exactly as normal. Assuming that MIST has been selected, the first step is to initialise the library by calling `MIST_Init()`. This triggers MIST to read its own `mist.params` input file and initialise the `System` and `Integrator` objects accordingly. If an error occurs at any stage (for example if the input file contained unrecognised keywords), the API returns an error code which the caller can check, and print an error message, or exit. If the call was successful, MIST returns `MIST_OK` (`0`). Once the host code has completed initialisation to the point where the molecular topology is built and initial coordinates and velocities for the particles are assigned, this information must be passed to MIST. A series of calls to (for example) `MIST_SetNumParticles()`, `MIST_SetNumBonds()` and `MIST_SetPositions()` are used to inform MIST of the number of atoms and bonds, and to pass a pointer to the location of the particle position data. In order to be completely flexible, data is passed across the API using void pointers, which are interpreted by MIST in the code-specific adaptor classes to yield data in the standard internal format provided by the `System` class for use by integrators. The design decision to store pointers to the host data, enabling integrators to directly modify the simulation state, is chosen because it is more efficient that making an MIST-internal copy of the data, modifying that and explicitly copying it back before force updates, or the end of the MD step .

In addition to passing data pointers into MIST, the host MD code must also register a callback function pointer and associated parameters by calling `MIST_SetForceCallback()`. This callback function may be called by MIST during an MD step as a black-box to compute updated forces given the current atomic positions (and in general velocities). Once again, we use a fully generic callback prototype, which accepts a single void pointer for any input data which may be required. Since the actual force computation routine typically does not conform to this interface, it is convenient to define a lightweight parameter data type to store all the arguments which should be passed to the force computation routine and a wrapper function which unpacks the type and calls the appropriate function to compute updated forces. For example, in GROMACS, we have:

```
typedef struct {
  FILE *log;
  t_commrec *cr;
  ...
  int flags;
  force_arrays_t *forces;
} force_params_t;

void do_force_wrapper(void *params){
    force_params_t *p = (force_params_t *)params;
    do_force(p->log,p->cr,p->inputrec,*(p->step),p->nrnb, \\
             *(p->wcycle),p->top,p->groups,*(p->box),p->x, \\
             p->hist,p->forces,*(p->vir_force),p->mdatoms, \\
             p->enerd,p->fcd,p->lambda,p->graph,p->fr \\
             p->vsite,*(p->mu_tot),p->t,p->field, \\
             p->ed,p->bBornRadii,p->flags);
}
...
// Declare a parameter type and store relevant local data in it
force_params_t p;
...
p.log = fplog;
p.cr = cr;
...
// Pass the function pointer and pointer to the parameter data to MIST
```

```
MIST_SetForceCallback(do_force_wrapper, &p)
```

At this point MIST has all the data required to carry out a single MD step. We note that, if for any reason the data pointers passed to MIST become out-of-date due to reallocation, or because the force parameters have changed, the `MIST_Set_*` functions must be called again as required.

In order to make using a MIST integrator as intuitive as possible for the user, we employ as much as possible of the unmodified code in the core MD time stepping loop. In particular, if trajectory output and computation of thermodynamic variables such as temperature are done at the start of the step before a 'native' integrator updates the system state, we do the same with MIST. If they are done at the end, we do likewise. However, in place of the native update code we insert a call to `MIST_Step()`. This hands over control to MIST to make whatever sequence of updates are implemented in the selected `Integrator`, including calls to the force callback routine as required during the step. When `MIST_Step()` returns, the system state has been advanced by a single time step `dt`, and any end-of-step actions which are required are taken such as incrementing step counters. In our model, MIST is responsible only for the integration step itself. This allows for a separation of concerns between configuring the integrator (via `mist.params`) and run control parameters e.g. number of steps, time step, output frequency and format, which are configured as usual for the host MD code.

Once the simulation has finished, the MIST library can be finalised by a call to `MIST_Cleanup()`, which simply deallocates any memory which has been allocated to allow a clean shutdown of the MD code.

2.4. Constraint Solver

In addition to the force-field i.e. the potential function of the atomic positions $U(\{\boldsymbol{r}\})$, from which the forces and therefore dynamics are derived, it is common in molecular simulations to apply *constraints* to the system. Bonds between heavy atoms and hydrogens have a high natural vibrational frequency and when using common integration algorithms such as the velocity Verlet method, which for a harmonic oscillator with frequency Ω has a stability threshold of $\delta t < 2/\Omega$, these vibrations severely limit the time step which may be used for stable MD (see [28, Chapter 4.2] for a more detailed discussion). For applications such as conformational sampling, constraints are typically used to remove such vibrational degrees of freedom from the simulation, for example replacing flexible covalent bonds which are modelled

as harmonic springs with rigid (fixed-length) 'rods', thus allowing a larger time step and longer overall simulated time scales to be accessed for the same computational cost. More complex constraints are also possible, including angular (fixing the internal angle between three atoms) and dihedral (fixing the torsional angle defined by four atoms), but these are not currently implemented in MIST. For integrators to be practically useful, they must be able to generate a series of positions which satisfy the constraints, and so to avoid complicated coordinate transformation, additional steps are needed after the standard time-propagation of the positions and velocities to correct these back onto the *constraint manifold* (the multidimensional surface made up of those points which satisfy the constraints). These functions are provided by the MIST `ConstraintSolver` class and may be called by `Integrator`s.

As described in Section 2.1, MIST has a representation of the molecular topology consisting of a set of bonds which link pairs of atoms (a, b), with an equilibrium bond length l (usually at the minimum of the bond-potential between the two atoms). MIST supports applying constraints to three different groupings of bonds: none (`constraints off`), only bonds involving hydrogen atoms (`constraints h-bonds-only`), and all bonds (`constraints all-bonds`). For the selected set of bonds, the `ConstraintSolver` sets up a list of k *holonomic* constraints (i.e. constraints depending only on the particle positions, and time), between atoms ka, kb of the form:

$$\sigma_k := ||\boldsymbol{r}_{ka} - \boldsymbol{r}_{kb}||^2 - l_k^2 = 0$$

Following the standard approach [29] of considering the force $\boldsymbol{G}_i$ due to each constraint involving a particle i, which is defined by method of Lagrange multipliers as:

$$\boldsymbol{G}_i = -\sum_k \lambda_k \nabla_i \sigma_k$$

Then the constraints can be resolved (up to a defined tolerance) by solving for the Lagrange multipliers σ_k and applying a correction to the unconstrained updated positions $\hat{\boldsymbol{r}}$:

$$\boldsymbol{r}_i(t + \delta t) = \hat{\boldsymbol{r}}_i(t + \delta t) + \sum_k \lambda_k \frac{\partial \sigma_k}{\partial \boldsymbol{r}_i}$$

By solving a second time for the set of Lagrange multipliers μ_k which satisfy the time derivative of the constraints:

$$\frac{d\sigma_k(t)}{dt} = (\boldsymbol{v}_{ka} - \boldsymbol{v}_{kb})(\boldsymbol{r}_{ka} - \boldsymbol{r}_{kb}) = 0,$$

where $\boldsymbol{v} = \dot{\boldsymbol{r}}$.

The unconstrained velocities $\hat{\boldsymbol{v}}$ may then be corrected by:

$$\boldsymbol{v}_i(t + \delta t) = \hat{\boldsymbol{v}}_i(t + \delta t) + \sum_k \mu_k \frac{\partial \sigma_k}{\partial \boldsymbol{r}_i}$$

Iterating through the constraints and adjusting the Lagrange multipliers, results in the RATTLE algorithm [30], and is selected with the keyword `constraints_method rattle`.

MIST also implements the adaptive Symmetric Newton Iteration (SNIP) scheme [31], where we construct a symmetric gradient matrix based on the configurations at the start of the timestep:

$$\hat{\boldsymbol{R}} \equiv \sigma'(\{\boldsymbol{r}\})\boldsymbol{M}^{-1}\sigma'(\{\boldsymbol{r}\})^t$$

Where $\sigma'(\{\boldsymbol{r}\})$ is the matrix of partial derivatives of the constraints with respect to the atomic coordinates and $\boldsymbol{M}$ is the diagonal matrix of particle masses. Since the definition of a bond constraint involves only on a pair of atomic positions, the gradient matrix is sparse, with entries on the diagonal:

$$\hat{\boldsymbol{R}}_{i,i} = ||\boldsymbol{r}_{ia} - \boldsymbol{r}_{ib}||^2 \left(\frac{1}{m_{ia}} + \frac{1}{m_{ib}}\right) = l_i^2 \left(\frac{1}{m_{ia}} + \frac{1}{m_{ib}}\right)$$

And off-diagonal for a pair of bonds i, j where $ia = ja$ (and equivalent expressions for other combinations):

$$\hat{\boldsymbol{R}}_{i,j} = \frac{(\boldsymbol{r}_{ia} - \boldsymbol{r}_{ib})(\boldsymbol{r}_{ia} - \boldsymbol{r}_{jb})}{m_{ia}}$$

We can then solve for the set of Lagrange multipliers:

$$\lambda_k(t) = \hat{\boldsymbol{R}}^{-1}\sigma_k(t)$$

Update the positions:

$$\boldsymbol{r}_i(t + \delta t) = \hat{\boldsymbol{r}}_i(t + \delta t) + \sum_k \lambda_k \frac{\partial \sigma_k}{\partial \boldsymbol{r}_i}$$

And iterate these two steps until convergence.

The velocity update is even simpler. As in RATTLE we directly solve for the Lagrange multipliers which satisfy the time derivatives of the constraints:

$$\mu_k(t) = \hat{\boldsymbol{R}}^{-1}\sigma'_k(t)$$

And finally, set the velocities as for RATTLE.

Importantly, for this method, the sparsity structure and the diagonal entries of the matrix $\hat{\boldsymbol{R}}$ are fixed for the duration of the simulation (since they depend only on the molecular topology), and the off-diagonal entries are fixed while the constraints are iterated. In MIST, we make use of the Eigen library [32] to store the sparse matrix, perform a Cholesky factorisation, and solve for the Lagrange multipliers. Eigen is particularly useful since we can perform a symbolic decomposition of the matrix once at the start of the simulation which makes the subsequent factorisation faster. SNIP is the default constraint solution method in MIST.

2.5. Units System

One of the objectives of MIST was to make development of new integrators easy, and to enable a single implementation to be reused with multiple MD codes. In addition to the abstractions discussed already, we must take account of the different units systems in use across different MD codes. To avoid having to include code-specific scaling factors in the `Step()` function of individual integrators, we assume that the integration takes place using the same units system as the host code, and where there are parameters, we choose a unit and scale this into the internal units system using a series of convenience functions. For example, in NAMD-Lite, energies are in kcal/mol, time is in femtoseconds, distances are in Angstroms. To obtain a consistent units system, the internal mass unit is 0.0004184 amu i.e. a hydrogen atom has a 'mass' of 2390.057... internal units. Conversely, Amber uses a units system where masses are in amu, distance is in Angstroms, energies are in kcal/mol, and time is in units of 1/20.455 ps!

The `System` class provides functions which return standardised lengths (1 Angstrom), masses (1 amu), times (1 picosecond), and Boltzmann's constant in the internal units system of the host code - values which are provided by the code adaptor classes. For example, in the Langevin dynamics integrator we require the constants $e^{-\gamma\delta t}$ and $\sqrt{k_B T(1 - e^{-2\gamma\delta t})}$. To convert the friction parameter γ from the specified units of ps^{-1} into internal time units we write:

```
double gdt = friction / system->Picosecond() * dt;
```

Similarly, to get the Boltzmann factor k_BT in internal energy units (T in Kelvin) and compute the two constants we can write:

```
double kbt = temp * system->Boltzmann();
...
double c1 = exp(-gdt);
double c3 = sqrt(kbt * (1 - c1 * c1));
```

3. Building and running MIST

In order to use MIST with a particular MD code requires inserting MIST API calls into the source code and implementing a `System` adaptor class. For codes which we provide adaptors for, source code patches are also given which can be applied during the build process. To a user, the process is very simple:

- Configure MIST for use with a particular MD code, providing the location of the source code as an argument to the MIST `configure` script. If configuration was successful, the script provides step-by-step instructions to complete the build for that specific MD code. In general, the steps proceed as follows
- Generate and apply the source code patch. For the versions that we support (NAMD-Lite 2.0.3, GROMACS 5.0.2, Amber 14), the code patches will apply seamlessly. It may be possible to apply the patches to other similar versions.
- Build the MIST library. Using a Makefile, MIST is built with support for the specified MD code compiled in.
- Build the host code. The host code is built (including the inserted MIST API calls) and linked with the MIST library. Appropriate modifications to the build options are automatically made by the patching process, so no manual configuration such as library search paths or other linker flags is required from the user.

Once the MD code is built, it can be used entirely as normal. To use a MIST integrator instead of the native ones provided by the host code requires adding a single parameter in the input file:

- NAMD-Lite: add `mist on` to the `.config` file.
- GROMACS: set `integrator = mist` in the `.mdp` file. Note that this change should be run through the GROMACS preprocessor `grompp` as usual, in order to have any effect.
- Amber: set `imist = 1` in the `&cntrl` namelist in the input file read by `pmemd`.

If MIST is enabled, execution will continue as described in Figure 2 and timestepping will be controlled according to the `mist.params` file which is read by MIST, see Section 2.2 for example parameters. Any trajectory or diagnostic output options specified in the host code's input will still produce output in the usual format (MIST only modifies the dynamics).

4. Performance Results

We have measured the performance of MD simulations using MIST by comparing with the native integrators in each of the host codes we plug into. CPU tests have been performed on ARCHER, a Cray XC30 with two Intel Xeon 12-core E5-2697v2 'Ivy Bridge' processors per node. We only use a single node due to the shared-memory parallelisation currently present in MIST. GPU tests have been performed on a Linux system with two Intel Xeon 8-core E5-2650v2 'Ivy Bridge' processors and eight NVIDIA Tesla K40m GPUs. GROMACS uses one GPU per MPI rank, so only a single GPU is used, even with multiple CPU threads.

All graphs show the average over 3 runs - error bars are not shown as the standard deviations were typically less than $< 1\%$, with the largest being 2.5%. We used different test systems for each supported MD code, in order to avoid making direct comparison between the performance of the MD codes themselves, and also to illustrate the versatility of MIST to be able to cope with differing force fields, periodic boundary conditions and constraints schemes.

4.1. NAMD-Lite

As an initial test, we simulated an isolated deca-alanine molecule using the input settings supplied in the `demo` directory of the NAMD-Lite distribution (also in the `examples/namd-lite/alanin` directory of MIST). A 12Å cut-off is applied for electrostatic forces, the system is initialised with random

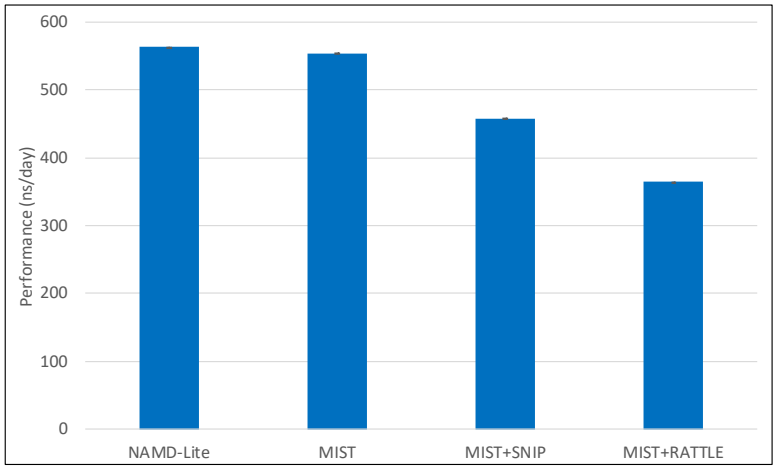


Figure 3: Performance of NAMD-Lite with and without MIST, for a deca-alanine molecule *in vacuo*.

velocities at 300K and we ran 1 ns of NVE dynamics using a 1 fs time step. A small system is a worst-case test for MIST, since the relatively cheap force evaluation and small number of atoms (66) means that the function call overheads of calling out to MIST to perform the integration are likely to be exposed.

As can be seen in Figure 3, we measured only a 2% slowdown when using the velocity Verlet integrator in MIST compared with the native integrator in NAMD-Lite, effectively running with MIST disabled. The figure shows the average over 3 runs for each setting - variability between runs was negligible (standard deviation $< 0.1\%$ in all cases). The performance when constraints are enabled in MIST, using both the RATTLE [30] and Symmetric Newton (SNIP) [31] methods, using the default constraint tolerance of 10^{-8} is shown. Resolving the constraints takes a significant amount of time, with RATTLE reducing the overall performance by 34%. We find SNIP to be significantly faster, with only a 17% performance drop. We are not able to compare directly with NAMD-Lite's constraints implementation since it only supports the limited capability of SETTLE [33] for rigid water models.

4.2. GROMACS

We illustrate the performance of MIST with GROMACS using a more realistic-sized system - a 50Å cubic periodic box containing 4069 water molecules. The input geometry and settings are supplied in the `examples/gromacs/water`

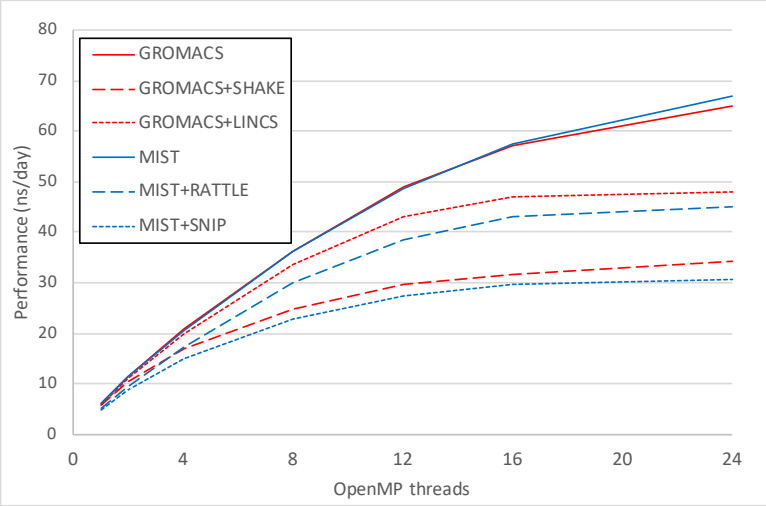


Figure 4: Performance of GROMACS on ARCHER with and without MIST, for the 12,207 atom water system.

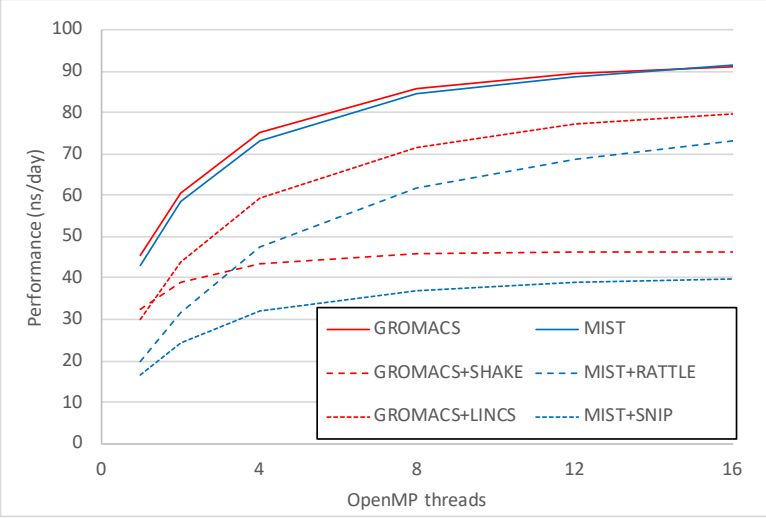


Figure 5: Performance of GROMACS on an NVIDIA K40m GPU with and without MIST, for the 12,207 atom water system.

directory of the MIST distribution. We use the TIP3P water model from the CHARMM27 force-field with a 12Å cut-off for the electrostatic and van der Waals forces. The system was initialized with random velocities at 300K and we ran 25 ps of NVE dynamics using a 1 fs time step. For this system (using a single CPU core) 96% of the run time is spent in force calculation and neighbor list search and less than 3% in the integration itself (the `Update` time reported by GROMACS). We tested both a fully flexible water model and one with bond constraints applied. GROMACS supports the SHAKE and LINCS[34] schemes for resolving constraints and we used the default settings for both. In MIST, we used a relatively loose constraint tolerance of 10^{-4}, matching the SHAKE tolerance used by GROMACS.

Figure 4 shows the performance achieved for each case using up to 24 OpenMP threads. For the unconstrained case, MIST is within 1% of native GROMACS performance and on 24 threads outperforms GROMACS by 3%. Comparing the constraint implementations, the LINCS [34] algorithm in GROMACS performs best although it is not possible to directly compare it with the SHAKE, RATTLE or SNIP solvers as it does not use a relative constraint tolerance, but rather a fixed (4th) order expansion and a fixed number of iterations (1). Interestingly, for the same tolerance, the RATTLE algorithm implemented in MIST is 12% slower than GROMACS' SHAKE implementation when running on a single thread, but when using 24 threads is 36% faster. Unlike for the small deca-alanine system, SNIP is the slowest algorithm due to the expensive inversion of the gradient matrix. It is important to recognize that SNIP is most advantageous when treating large macromolecules with high bond connectivity, whereas waters are actually better handled by RSHAKE [35] or SETTLE [33].

On the GPU (Figure 5), we see a slightly higher overhead between MIST and the native GROMACS Verlet integrator, of around 5.5% using a single core, becoming negligible on 16 cores. This reflects the fact that as the force evaluation using the GPU is much faster, with overall performance of 45 ns/day compared with 6 ns/day on the CPU only, the additional cost associate with calling out to MIST to do the integration is proportionally higher. However, the use of OpenMP within the MIST integrator offsets this at higher thread counts. Similarly to the CPU, the GROMACS LINCS implementation is fastest, but the difference in performance between the GROMACS SHAKE and MIST RATTLE implementations is much higher, with MIST being 39% slower on a single thread, but 57% faster using 16 threads.

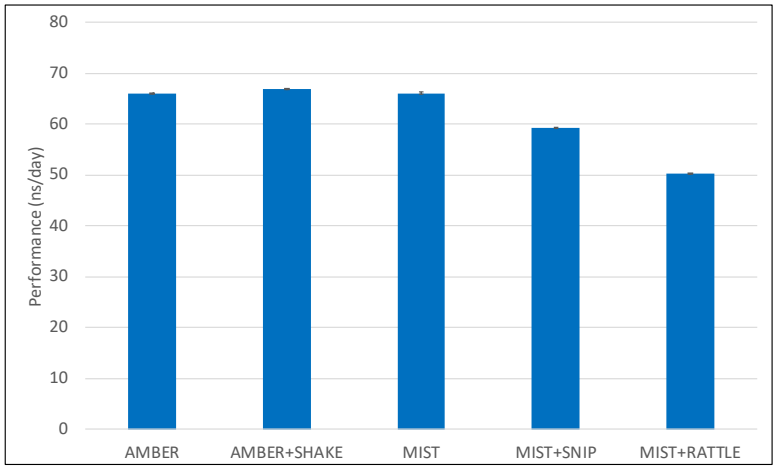


Figure 6: Performance of Amber on ARCHER with and without MIST, for the solvated NTL9 system.

4.3. Amber

To test the performance of MIST with Amber, we used the well-known [36, 37] NTL9(1-39) protein, which consists of 636 atoms and is solvated in 4488 water molecules, for a total of 14100 atoms in an approximately 54Å orthorhombic periodic unit cell. The CHARMM22 force field with a TIP3P water model are used, with a cut-off of 9Å for real-space part of the electrostatic forces and the long-ranged electrostatics computed on 54^3 PME grid. The system was initialised with random velocities at 300K, and we ran 25ps of NVE dynamics using a 1 fs time step with the `pmemd` or `pmemd.cuda` program. Input files are available in `examples/amber/ntl9` in the MIST distribution. Amber supports bond constraints for hydrogen atoms only on the GPU, so we used the corresponding `h-bonds-only` setting in MIST, and a relative tolerance of $1E-5$, matching the default Amber SHAKE tolerance.

Figure 6 shows the performance achieved running on a single CPU core on ARCHER (Amber 14 does not have thread parallelisation). We see that using MIST has negligible impact on the performance. For the constrained runs, we find that Amber is slightly faster (by 1%) as it is possible to skip the computation of the forces caused by the constrained bonds (`ntf=2` in Amber), offsetting the additional cost of the SHAKE algorithm. Similarly to NAMD-Lite and GROMACS, both the MIST constraint solver algorithms have an additional performance overhead. For this system, SNIP is faster with an 11% drop, compared to 24% for RATTLE.

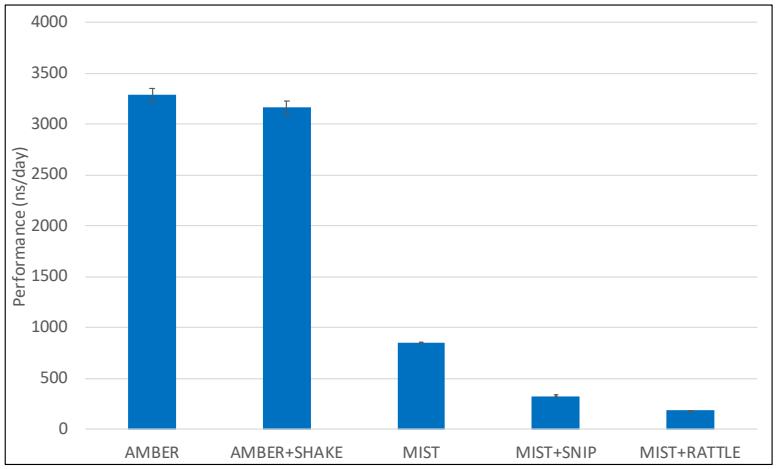


Figure 7: Performance of Amber on an NVIDIA K40m GPU with and without MIST, for the solvated NTL9 system.

For Amber, the overhead of using MIST with `pmemd.cuda` is much higher (see Figure 7). Whereas GROMACS achieves around 10× speedup with a K40m GPU compared to a single CPU core, Amber achieves a speedup of over 50× by doing the entire calculation (both the force evaluation and integration) on the GPU, thus avoiding relatively high-latency transfers between the GPU and CPU memory. In order to use MIST the updated coordinates must be transferred to the GPU and the resulting forces transferred back again *at each time step*, effectively throttling the GPU by memory transfers. As a result, running with MIST achieves only 844 ns/day, compared with 3282 ns/day using the native integrator running on the GPU. This is still over 10× higher than the 66 ns/day achieved by Amber and/or MIST running on the CPU. As expected, the constrained runs are slower still with a 62% overhead for SNIP and 78% overhead for RATTLE. These overheads are higher than observed in the CPU runs as the force evaluation is much faster so the time spent in the constraint solver is proportionally larger.

4.4. Discussion

The performance tests above show that, except in the case of Amber on GPU, performing integration within the MIST library has a negligible overhead compared to a native integrator. Performing constrained integration comes at an additional overhead, the cost depending on the system size, topology and the constraint solver method and accuracy chosen, but is com-

parable to common methods such as SHAKE implemented in Amber and GROMACS. As a result, MIST provides a suitable platform for developing new integrators with minimal loss of performance, but with much lower code complexity compared to developing an algorithm directly in one of the host codes. In addition, we have shown that the same algorithms–implemented once in MIST–can be used in the context of 3 different codes, enabling greater applicability of any newly developed algorithm.

5. Application: Simulated Tempering of Alanine-12

Alanine-12 is a classic example of an α-helical biomolecule. It is particularly interesting to disentangle the effects of solvation and temperature on the unfolding process. At room temperature, the unfolding process cannot be simulated directly with GROMACS because of slow kinetics. An advanced sampling method such as Simulated Tempering could overcome this, but is not currently implemented in GROMACS. To demonstrate the flexibility and capability of MIST, we implemented the Simulated Tempering algorithm of Nguyen *et al* [26, 27], which previously have only been made available as a set of shell scripts [38]. To set up a simulation using the scripts requires creating separate GROMACS input files for each temperature state, then running multiple short simulations, where the potential energy is parsed from the output file and a probabilistic change to another temperature state is made according to the algorithm. As a result, the scripts generate a set of trajectory data files, which must be concatenated for analysis, and running many short individual simulations makes it inefficient to operate through an HPC batch system. An additional novel element of our implementation is that the temperature of each state is controlled using Langevin Dynamics implemented using a BAOAB splitting scheme [14] to give more accurate configurational averages. Our simulations are not designed to test this assertion, but a thorough analysis by Fass *et al* [39] showed dramatic reduction in configuration space discretisation error compared with other schemes.

To run Simulated Tempering through MIST requires a single long MD run, enabling MIST as described in Section 3. The Simulated Tempering algorithm is selected and configured by a `mist.params` file as follows:

```
integrator simulated_tempering # Select Simulated Tempering
temperatures[0] 300            # Define a series of temperature states
temperatures[1] 310
```

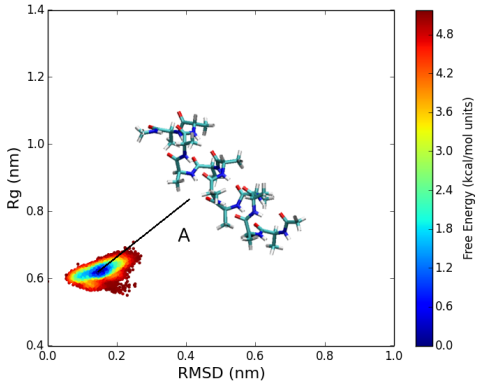
```
temperatures[2] 320
...
temperatures[14] 440
temperatures[15] 450
period 2500                    # Attempt to switch states every 2500 steps
constraints all-bonds          # Apply bond constraints

langtemp 300                   # Start system at 300K
langfriction 1.0               # 1/ps friction constant
```

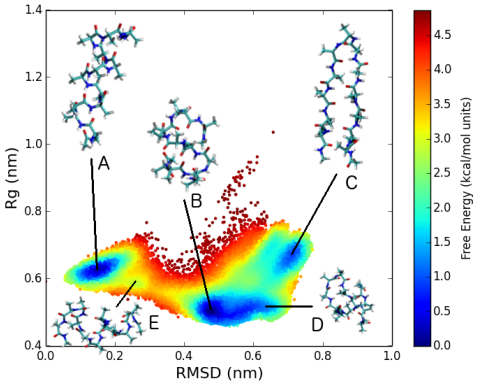
To test the implementation, we sampled the free energy landscape of the alanine-12 molecule, previously studied using the Diffusion-Map-directed-MD method [40]. Starting from the helical configuration *in vacuo*, we ran a total of 1 μs of MD using a 2 fs timestep. We used the Amber96 force-field with a 20Å cut-off for electrostatics, constraining all bonds using the SNIP method. Temperature was controlled using Langevin dynamics ($\gamma = 1.0ps^{-1}$), either set to 300K to sample an NVT ensemble or varied using Simulated Tempering with temperature states ranging from 300K to 450K at 10K intervals. The resulting free energy surfaces are plotted as a function of RMSD from the initial state and the radius of gyration R_g in Figure 8.

As expected, Figure 8a shows that at 300K, the system is trapped in a local minimum around the helical state (labelled A). Using Simulated Tempering, the elevated temperature is enough to allow the system to explore into a wider range of (partially uncoiled) configurations - comparable to those accessed by plain MD at 400K in Figure 7 of [40]. Figure 8b shows the complete set of configurations sampled, including those at temperatures greater than 300K. Restarting simulations from the configurations labelled B (a compact structure consisting of three hairpin turns) and C (where the two termini are aligned and a complex twisted structure forms in the backbone) and running a subsequent 1 μs of NVT dynamics at 300K shows that configuration B exists in a stable minima (Figure 8c), whereas the system is free to migrate between configurations C and D via a transition state F (Figure 8d), where the termini of the molecule have turned back on themselves.

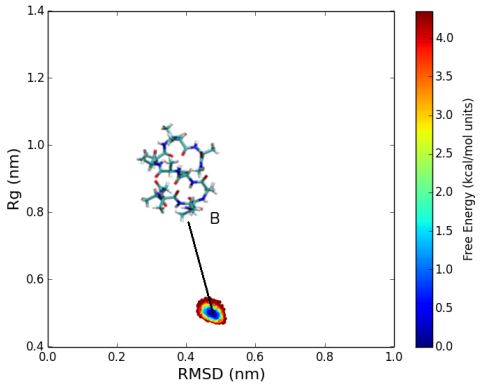
In contrast, even at 300K (Figure 9a) the solvated system is able to access a much wider range of states including the fully unfolded state (H) and a large basin (G) containing various extended structures. This is qualitatively similar to the behaviour of deca-alanine observed in [41], which has extended conformations of comparable free energy to the helical state. Physically, the



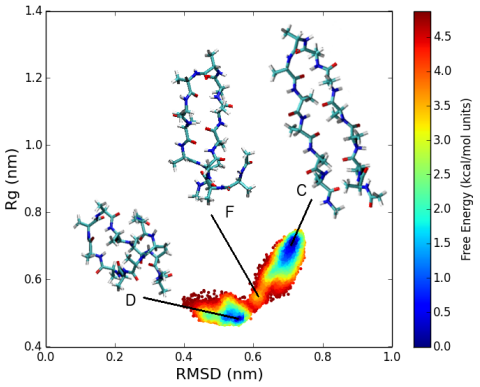
(a) NVT at 300K, starting from helical configuration



(b) Simulated Tempering 300-450K

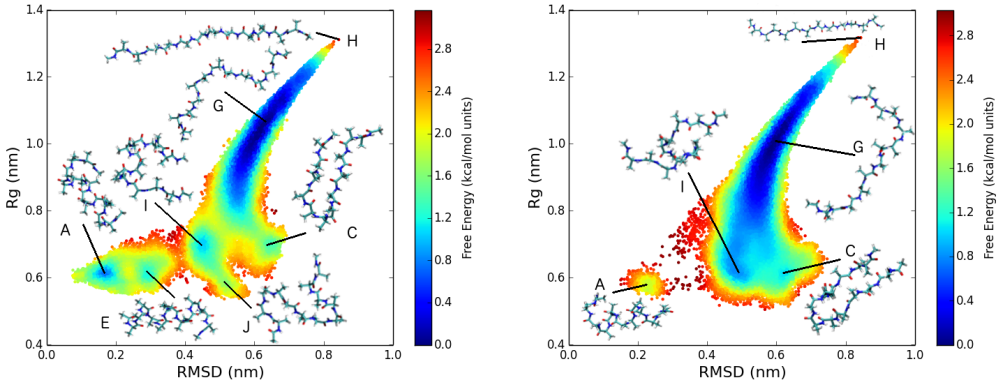


(c) NVT at 300K, starting from configuration B



(d) NVT at 300K, starting from configuration C

Figure 8: Free energy surfaces of Alanine-12 *in vacuo*



(a) NVT at 300K, starting from helical configuration

(b) Simulated Tempering 300-450K

Figure 9: Free energy surfaces of Alanine-12 solvated in TIP3P water.

addition of water molecules provides an alternative hydrogen bonding route than can effectively 'bridge' between -CO and -NH groups in the backbone, stabilising extended structures that are not observed in the *in vacuo* ensemble. Compared with the *in vacuo* simulations, the molecule does not sample the compact ($Rg \simeq 0.5$) states B and D, but instead states like I and J where both ends of the molecule are unbound and a hairpin turn or complete helix is present in the middle. Due to the relatively low energy barriers between states (< 2.8kcal/mol) compared with barriers of up to 4 kcal/mol in the vacuum case, simulated tempering does not provide any access to any qualitatively new states (Figure 9b).

6. Conclusion and Future Development

We have described the architecture and implementation of MIST, the Molecular Integration Simulation Toolkit, a C++ library which provides an abstraction layer over common MD codes to enable rapid development of new MD integration algorithms. The initial release of the library contains implementations of six different integrators, and is interfaced via a C or Fortran API to four MD codes: NAMD-Lite, GROMACS, Amber and Forcite. MIST provides a portable platform for the development of novel integrators, which can be implemented once in MIST and used with any of the MD codes interfaced to MIST. We have demonstrated the ease-of-use of MIST

by implementing the simulated tempering scheme of Nguyen *et al* [26, 27] in combination with Langevin Dynamics using a 'BAOAB' splitting [14] and applying it to study the free energy landscape of Alanine-12 using GPU-accelerated GROMACS. MIST is freely available under a BSD license from `https://bitbucket.org/extasy-project/mist`.

In serial, multi-threaded and GPU-accelerated configurations we have shown that MIST introduces negligible overhead compared to the native calculation, with the exception of Amber's GPU implementation. In that case, the additional data transfer of the system state off the GPU introduces latency which slows the calculation down to performance comparable to GROMACS, where the native integration step is computed on the CPU and only forces are evaluated on the GPU. It is possible to envisage a hybrid/multiple timestepping scheme where MIST is used to integrate the outer timestep for slow degrees of freedom such as a thermostat, and the inner timestep is integrated directly on the GPU using Amber's native integrator. This has not yet been implemented, however.

At present, MIST is restricted to use in a shared-memory environment. While it is possible to simulate quite large systems over reasonable timescales using multi-threading and/or GPU acceleration, to model very large macromolecules in complex environments requires the use of parallel computing using MPI. Work is currently underway to remove the underlying assumption that the complete state of the system is available within a single address space, allowing MIST integrators to be used where the host MD code employs a domain decomposition. The approach should be flexible enough to support particle-based, as well as space-partitioning strategies. In addition, MIST itself needs to be parallelized, in particular the constraint solver, which must be able to resolve constraints potentially involving particles located on multiple processes.

While our initial release of MIST focuses on biomolecular applications, typically solvated molecules in an NVT ensemble, the tempering methods in particular are of interest for materials applications. To represent a periodic solid, allowing for behavior such as expansion, contraction and phase transitions requires the lattice vectors of the simulation cell to become dynamical variables in the same sense as particle positions and velocities. An interface to a materials simulation code such as LAMMPS [42], DL_POLY [43], or GULP [44] combined with implementation in MIST of constant-pressure barostats such as Parrinello-Rahman [11] or Martyna-Tuckerman-Klein [10] would open up usage of MIST to a wider user community.

Acknowledgements

This work used the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk) and systems at the STFC Hartree Centre (http://www.hartree.stfc.ac.uk). Funding has been provided by the Engineering and Physical Sciences Research grants EP/K039512/1 and EP/P006175/1, the University of Edinburgh via a Staff Scholarship, the ERC grant HECATE and by the Hartree Centre.

We thank Charles Laughton for helpful suggestions on the interpretation of our Alanine-12 simulation results, and for the help of Phuong Nguyen on the implementation details of simulated tempering.

References

[1] R. Elber, Perspective: Computer simulations of long time dynamics, The Journal of Chemical Physics 144 (6) (2016) 060901. doi:10.1063/1.4940794.

[2] A. Laio, M. Parrinello, Escaping free-energy minima, Proceedings of the National Academy of Sciences 99 (20) (2002) 12562–12566. doi:10.1073/pnas.202427399.

[3] Y. Sugita, Y. Okamoto, Replica-exchange molecular dynamics method for protein folding, Chemical Physics Letters 314 (1) (1999) 141 – 151. doi:10.1016/S0009-2614(99)01123-9.

[4] U. H. Hansmann, Parallel tempering algorithm for conformational studies of biological molecules, Chemical Physics Letters 281 (1) (1997) 140 – 150. doi:10.1016/S0009-2614(97)01198-6.

[5] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, K. Schulten, Scalable molecular dynamics with NAMD, Journal of Computational Chemistry 26 (16) (2005) 1781–1802. doi:10.1002/jcc.20289.

[6] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, E. Lindahl, GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers, SoftwareX 12 (2015) 19 – 25. doi:10.1016/j.softx.2015.06.001.

[7] D. E. Shaw, M. M. Deneroff, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, K. J. Bowers, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Ierardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, S. C. Wang, Anton, a special-purpose machine for molecular dynamics simulation, Commun. ACM 51 (7) (2008) 91–97. doi:10.1145/1364782.1364802.

[8] H. J. C. Berendsen, J. P. M. Postma, W. F. van Gunsteren, A. DiNola, J. R. Haak, Molecular dynamics with coupling to an external bath, The Journal of Chemical Physics 81 (8) (1984) 3684–3690. doi:10.1063/1.448118.

[9] W. G. Hoover, Canonical dynamics: Equilibrium phase-space distributions, Phys. Rev. A 31 (1985) 1695–1697. doi:10.1103/PhysRevA.31.1695.

[10] G. J. Martyna, M. L. Klein, M. Tuckerman, NoséHoover chains: The canonical ensemble via continuous dynamics, The Journal of Chemical Physics 97 (4) (1992) 2635–2643. doi:10.1063/1.463940.

[11] M. Parrinello, A. Rahman, Crystal structure and pair potentials: A molecular-dynamics study, Phys. Rev. Lett. 45 (1980) 1196–1199. doi:10.1103/PhysRevLett.45.1196.

[12] W. Zheng, M. A. Rohrdanz, C. Clementi, Rapid exploration of configuration space with diffusion-map-directed molecular dynamics, The Journal of Physical Chemistry B 117 (42) (2013) 12769–12776, pMID: 23865517. doi:10.1021/jp401911h.

[13] D. Hamelberg, J. Mongan, J. A. McCammon, Accelerated molecular dynamics: A promising and efficient simulation method for biomolecules, The Journal of Chemical Physics 120 (24) (2004) 11919–11929. doi:10.1063/1.1755656.

[14] B. Leimkuhler, C. Matthews, Robust and efficient configurational molecular sampling via langevin dynamics, The Journal of Chemical Physics 138 (17) (2013) 174102. doi:10.1063/1.4802990.

[15] G. Gobbo, B. J. Leimkuhler, Extended hamiltonian approach to continuous tempering, Phys. Rev. E 91 (2015) 061301. doi:10.1103/PhysRevE.91.061301.

[16] A. Dullweber, B. Leimkuhler, R. McLachlan, Symplectic splitting methods for rigid body molecular dynamics, The Journal of Chemical Physics 107 (15) (1997) 5840–5851.

[17] B. Leimkuhler, C. Matthews, Efficient molecular dynamics using geodesic integration and solvent–solute splitting, Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 472 (2189). doi:10.1098/rspa.2016.0138.

[18] M. Ceriotti, G. Bussi, M. Parrinello, Colored-noise thermostats la carte, Journal of Chemical Theory and Computation 6 (4) (2010) 1170–1180. doi:10.1021/ct900563s.

[19] J. A. Morrone, T. E. Markland, M. Ceriotti, B. J. Berne, Efficient multiple time scale molecular dynamics: Using colored noise thermostats to stabilize resonances, The Journal of Chemical Physics 134 (1) (2011) 014103. doi:10.1063/1.3518369.

[20] D. A. Case, T. E. Cheatham, T. Darden, H. Gohlke, R. Luo, K. M. Merz, A. Onufriev, C. Simmerling, B. Wang, R. J. Woods, The Amber biomolecular simulation programs, Journal of Computational Chemistry 26 (16) (2005) 1668–1688. doi:10.1002/jcc.20290.

[21] D. J. Hardy, NAMD-Lite, http://www.ks.uiuc.edu/Development/MDTools/namdlite/, University of Illinois at Urbana-Champaign (2007).

[22] BIOVIA, Biovia materials studio forcite plus, http://accelrys.com/products/datasheets/forcite-plus.pdf.

[23] G. A. Tribello, M. Bonomi, D. Branduardi, C. Camilloni, G. Bussi, PLUMED 2: New feathers for an old bird, Computer Physics Communications 185 (2) (2014) 604 – 613.

[24] P. Eastman, J. Swails, J. D. Chodera, R. T. McGibbon, Y. Zhao, K. A. Beauchamp, L.-P. Wang, A. C. Simmonett, M. P. Harrigan, C. D. Stern, R. P. Wiewiora, B. R. Brooks, V. S. Pande, OpenMM

7: Rapid development of high performance algorithms for molecular dynamics, PLOS Computational Biology 13 (7) (2017) 1–17. doi:10.1371/journal.pcbi.1005659.

[25] H. Yoshida, Construction of higher order symplectic integrators, Physics Letters A 150 (5) (1990) 262 – 268. doi:10.1016/0375-9601(90)90092-3.

[26] P. H. Nguyen, Y. Okamoto, P. Derreumaux, Communication: Simulated tempering with fast on-the-fly weight determination, The Journal of Chemical Physics 138 (6) (2013) 061102. doi:10.1063/1.4792046.

[27] T. Zhang, P. H. Nguyen, J. Nasica-Labouze, Y. Mu, P. Derreumaux, Folding atomistic proteins in explicit solvent using simulated tempering, The Journal of Physical Chemistry B 119 (23) (2015) 6941–6951. doi:10.1021/acs.jpcb.5b03381.

[28] B. J. Leimkuhler, C. Matthews, Molecular Dynamics with Deterministic and Stochastic Numerical Methods, Springer International Publishing, 2015. doi:10.1007/978-3-319-16375-8.

[29] J.-P. Ryckaert, G. Ciccotti, H. J. Berendsen, Numerical integration of the cartesian equations of motion of a system with constraints: molecular dynamics of n-alkanes, Journal of Computational Physics 23 (3) (1977) 327 – 341. doi:10.1016/0021-9991(77)90098-5.

[30] H. C. Andersen, Rattle: A velocity version of the shake algorithm for molecular dynamics calculations, Journal of Computational Physics 52 (1) (1983) 24 – 34. doi:10.1016/0021-9991(83)90014-1.

[31] E. Barth, K. Kuczera, B. Leimkuhler, R. D. Skeel, Algorithms for constrained molecular dynamics, Journal of Computational Chemistry 16 (10) (1995) 1192–1209. doi:10.1002/jcc.540161003.

[32] G. Guennebaud, B. Jacob, et al., Eigen v3, http://eigen.tuxfamily.org (2010).

[33] S. Miyamoto, P. A. Kollman, Settle: An analytical version of the SHAKE and RATTLE algorithm for rigid water models, Journal of Computational Chemistry 13 (8) (1992) 952–962. doi:10.1002/jcc.540130805.

[34] B. Hess, H. Bekker, H. J. C. Berendsen, J. G. E. M. Fraaije, LINCS: A linear constraint solver for molecular simulations, Journal of Computational Chemistry 18 (12) (1997) 1463–1472. doi:10.1002/(SICI)1096-987X(199709)18:12<1463::AID-JCC4>3.0.CO;2-H.

[35] A. Kol, B. B. Laird, B. J. Leimkuhler, A symplectic method for rigid-body molecular simulation, The Journal of Chemical Physics 107 (7) (1997) 2580–2588. doi:10.1063/1.474596.

[36] K. Lindorff-Larsen, S. Piana, R. O. Dror, D. E. Shaw, How fast-folding proteins fold, Science 334 (6055) (2011) 517–520. doi:10.1126/science.1208351.

[37] V. A. Voelz, G. R. Bowman, K. Beauchamp, V. S. Pande, Molecular simulation of ab initio protein folding for a millisecond folder NTL9(139), Journal of the American Chemical Society 132 (5) (2010) 1526–1528, pMID: 20070076. doi:10.1021/ja9090353.

[38] T. Zhang, P. H. Nguyen, P. Derreumaux, Simulated Tempering for GROMACS.

[39] J. Fass, D. Sivak, G. E. Crooks, K. A. Beauchamp, B. Leimkuhler, J. Chodera, Quantifying configuration-sampling error in langevin simulations of complex molecular systems, bioRxivdoi:10.1101/266619.

[40] J. Preto, C. Clementi, Fast recovery of free energy landscapes via diffusion-map-directed molecular dynamics, Phys. Chem. Chem. Phys. 16 (2014) 19181–19191. doi:10.1039/C3CP54520B.

[41] A. Hazel, C. Chipot, J. C. Gumbart, Thermodynamics of deca-alanine folding in water, Journal of Chemical Theory and Computation 10 (7) (2014) 2836–2844, pMID: 25061447. doi:10.1021/ct5002076.

[42] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, Journal of Computational Physics 117 (1) (1995) 1 – 19. doi:10.1006/jcph.1995.1039.

[43] I. T. Todorov, W. Smith, K. Trachenko, M. T. Dove, DL_POLY_3: new dimensions in molecular dynamics simulations via massive parallelism, J. Mater. Chem. 16 (2006) 1911–1918. doi:10.1039/B517931A.

[44] J. D. Gale, GULP: A computer program for the symmetry-adapted simulation of solids, J. Chem. Soc., Faraday Trans. 93 (1997) 629–637. doi:10.1039/A606455H.