

Technical report: An integrated system for executing and analyzing large-scale molecular dynamics simulations

John D. Chodera[‡], David E. Eramian[†], Larry Schweitzer[‡], and Ken A. Dill[‡]

[†] Graduate Group in Biophysics

[‡] Department of Pharmaceutical Chemistry
University of California, San Francisco
San Francisco, CA 94143

September 3, 2004

Abstract

This technical report outlines the requirements of a system for the execution and analysis of parallel molecular dynamics simulations that allows the easy implementation of efficient modern simulation algorithms for computing thermodynamic and kinetic quantities with classical molecular dynamics. Some notes on a possible design for such a system are also included as a first step toward the construction a generally useful system for the biomolecular simulation community.

1 Simulation algorithms

There are currently several kinds of simulation algorithms we currently employ (or would like to implement) to generate ensembles of molecular dynamics trajectories for estimation of thermodynamic or kinetic observables. All of these algorithms use short molecular dynamics trajectories (typically 1-100 ps in length) as a sort of fundamental unit of work, though Metropolis Monte Carlo simulations can fit into this framework as well. Here are some examples:

Replica-exchange. In a replica-exchange molecular dynamics simulation, N iterations are conducted in which each iteration consists of short (1-20 ps) molecular dynamics simulations of M independent replicas at different simulation conditions (temperature, field strengths, etc.) followed by a Metropolis Monte Carlo-like exchange trial using the potential energy [6]. The number of replicas M is typically in the range of 24-128, and the number of iterations N between 10^3 and 10^4 . This results in the generation of a large number of short trajectory segments $NM \approx 10^4 - 10^6$. If we have $L = 10^1 - 10^2$ snapshots per trajectory segment, we end up with $NML \approx 10^5 - 10^8$ snapshots per simulation.

Dynamical evolution from a prepared state. Often, the statistical dynamics of evolution from a prepared state (such as an initial configuration or phase space cell) or number of states (when estimating transition matrices) is of interest [7, 8]. In this case, a number of trajectories of moderate length (10-100 ps) are generated from an initial ensemble of starting states and simulated independently.

Transition path sampling. In transition path sampling, a Monte Carlo simulation in path space is conducted [3]. For each perturbation to the path, new trajectories are initiated in the forward and backward directions, and information from the produced trajectories is used to decide whether to accept or reject the move. In this way, a number of trajectories are generated in a serial manner.

Hybrid Monte Carlo. In Hybrid Monte Carlo (HMC) [?, 1], sampling from the canonical distribution is achieved by conducting a number of short molecular dynamics simulations in sequence. Each simulation begins with selection of a new momentum from the Maxwell-Boltzmann distribution, integration of the molecular dynamics trajectory, and then a Metropolis acceptance step using the change in total energy from initial and final steps of the trajectory. Because of the use of a Metropolis acceptance step, the timestep used to integrate dynamics can be increased to be rather large, generating uncorrelated snapshots with less computer time but leading to poor energy conservation. The effects of poor energy

* Author to whom correspondence should be addressed.

conservation are corrected by the acceptance step, but only the final snapshots of each trajectory are then distributed from the canonical ensemble.

Wang-Landau sampling. In Wang-Landau sampling [4, 5], trajectories are generated with random initial velocities as in hybrid Monte Carlo, but the acceptance of the trajectory is based on statistics accumulated in the form of a transition matrix containing counts of the generated transitions. The acceptance criterion attempts to produce a random walk in the coordinates of interest (e.g. potential energy, an intramolecular distance, etc.).

Each of these algorithms generates a large quantity of data which is then subjected to researcher-directed analyses in order to draw conclusions about the thermodynamics and kinetics of the molecular system being simulated. Our experience is that Perl scripts, manual job submission to local clusters and remote supercomputers, slow RAID storage systems, NFS-mounted filesystems, remote copying from supercomputers, and flat files are unable to easily deal with the scale these simulations are quickly reaching.

2 Task scheduling system

We propose a task scheduling system that will allow us to easily and efficiently make use of diverse computational resources in carrying out simulations of the sort described above. This system will be general enough to not just be able to carry out many simulations in parallel, but also potentially allow us to analyze the data in parallel.

Allow use of heterogeneous compute nodes. We would like to use compute nodes from our various local compute clusters as well as remote supercomputers. Most of these machines have separate environments and filesystems, and we wish to avoid extensive manual copying of files back and forth between the systems. The user will be expected to have set up the molecular dynamics codes and appropriate environments on each system. For now, we assume the user has manually executed client “Worker” processes on these systems (which connect to the master server), but we can eventually use tools like Globus to automatically start client “Worker” processes on available nodes based on the workload waiting in the queue.

Easy construction of new simulation algorithms. The code required to implement any of the aforementioned algorithms, as well as new advanced sampling methods, should be extremely simple and simple to write. This will greatly simplify the researcher’s ability to try out new sampling algorithms.

Tolerance to compute node failure. If a node fails or becomes disconnected from the network while executing a dynamics task, the task should be reassigned to another node. To avoid losing much data from the failure of a long dynamics task, long simulations should be split into shorter simulations.

Reasonably efficient utilization of resources. Since the client node pool available for work is rather heterogeneous, different clients may take different amounts of time to complete different tasks. However, as long as all the client “Worker” processes are always kept busy, there is no loss in efficiency due to idle CPUs, even though some algorithms like replica-exchange seem to be best matched to symmetric multiprocessing. (This common belief is actually fallacious, as the amount of time required to complete the same length trajectory segment for different replicas can vary greatly. Asynchronous execution could be much more efficient.) However, to prevent certain jobs from taking too much wall clock time to execute, and to be fair to users, an intelligent scheduling algorithm could try to prioritize execution of tasks to minimize the wall clock time of jobs that have been in the queue the longest.

Heterogeneous client node capability and tasking. It may be the case that certain nodes are only available for performing certain tasks. For example, some nodes may not have application software installed for performing other tasks. Another possibility is that some client nodes may be much less well suited for performing analysis tasks because of their slow network connection to the database server, and so we would want to limit or prioritize which task types different classes of nodes can execute.

Automatic deposition of simulation results into a database. Rather than a remote client node writing generated data to disk and then transporting it to our servers when the calculation is done (a process which can add significant overhead to the computation, sometimes up to half the compute time!) it would be optimal if the simulation was able to transfer output data back to our servers as the computation progresses, much like the Blue Matter code does.

Automatic checkpointing and disaster recovery. Large runs should be split into a sequence of shorter runs that get checked into the database in chunks, so if a node dies during execution, only a bit of work is lost, and the job automatically recovers. Similarly, “master jobs”, which generate the short simulation tasks, should be persistent.

Transactional security for all! A master job should take results of the simulations it spawned, update its internal state, and produce new tasks all under the umbrella of transactional security so that it will never be left in an inconsistent state. Similarly, a client worker tasks take tasks and submit results under a transaction, so failure anywhere in the process leaves the whole system in a consistent state.

Web-based monitoring and administration of jobs and resources. Eventually, we wish to have a nice web-based interface for managing our jobs and computational resources.

Easy interface to existing compute codes. We would like to be able to transform input for use with existing dynamics codes. A general intermediate data-interchange format is needed.

As an example of the heterogeneous computing environment we will be dealing with, here is a short list of some of our available computational resources:

davy. 30 dual-Xeon 2.8 GHz nodes located in the first-floor machine room at Mission Bay, running RedHat 7.3. The filespace is separate from the rest of the lab network.

kubo. 12 Athlon 750 MHz and 10 Athlon 1.4 GHz nodes running RedHat 9.0, located in the fourth-floor machine room.

CCPR cluster.

lemieux. Located at the Pittsburgh Supercomputing Center, lemieux consists of XXX quad-processor Alpha nodes.

morpheus and hypnos. Located at the University of Michigan...

Blue Horizon. Located at the San Diego Supercomputing Center.

seaborg. NERSC’s big machine, consisting of 400-something 32-way SP2’s.

3 Trajectory data analysis system

In order to allow researchers to be most productive, we need an efficient system to store, organize, and analyze the data generated by our simulations (or imported into the database from other sources).

3.1 Trajectory database

There are three types of data that need to be stored in a simulation database:

Raw snapshot data. Raw snapshot data that describes the simulation state consists of the coordinates, velocities, and possibly some other information such as the simulation box size. In order to restart a simulation (which is required in many advanced sampling techniques for launching new trajectories) the coordinate and velocity data needs to be stored to full machine precision, which causes these snapshots to be rather large (often 100KB). For the calculation of most mechanical observables (e.g. the radius of gyration, but not the potential energy), it is often possible to store coordinates to reduced precision.

Snapshot data is never used as a field in a user query – it does not make much sense for the researcher to ask a question “Show me all the snapshots where the x-coordinate of atom 5 is between 3.2 and 4.5 in absolute box coordinates.” The raw data is primarily accessed only when observables need to be computed all the snapshots in an ensemble. Here, it is often still useful to split the raw data up into subensembles for multiple CPUs to process in parallel. Another possibility is that the user performs a query on some computed observables and selects a subensemble and then requires the coordinates for some operation such as visualization.

Because of these standard usage expectations, storing the coordinates and velocities as columns in a relational database doesn’t make much sense and would require a huge amount of overhead. Storing this data as a binary string (or “blob”) in a single column is a possibility, and modern databases (such as PostgreSQL) can do this with relatively low overhead.

Another possibility would be to use a key-value database (such as BerkeleyDB) which allows the storage of TB of data, has transactional capability, and opens the possibility for redundant storage and making safe backups. The advantage of storing the data in a database is that the amount of data cached to system memory can be carefully tuned, with the possibility that all data can be cached in memory, making access to this data orders of magnitude faster than if it came off a filesystem.

Other possibilities for storage exist. The Storage Resource Broker, or SRB (<http://www.npaci.edu/DICE/SRB/>), can automatically distribute data to multiple sites with redundancy and staging to long-term storage, but without any sort of transactional security. Storing the data in flat files on a standard filesystem is also a possibility, though our experience with slow RAID's mounted over NFS has shown this to be slow and unreliable.

When storing data in binary formats, we may have to worry about platform-dependence of floating-point representations when storing numbers as binary strings – a standard platform-independent format will have to be adopted if portability is to be assured. Use of scientific dataset storage formats like HDF5 or netCDF may avoid this.

The snapshot data for experiments can grow to fairly large sizes. It is not uncommon for 50 GB of data to be generated for a single experiment, and we have been considering running experiments that may generate up to 1 TB of data.

Observables. Observables are quantities associated with each snapshot. These can be scalar-valued (e.g. the radius of gyration or instantaneous kinetic temperature), discrete-valued (e.g. number of native hydrogen bonds), boolean (e.g. whether or not a particular hydrogen bond is made), vector-valued (e.g. the current simulation box dimensions), or tensor-valued (e.g. the instantaneous pressure tensor). Each observable is much smaller than the raw full-precision coordinate snapshots (often just a scalar per snapshot). These observables are usually used in further processing, such as the generation of potentials of mean force or transition matrices. Queries will often need to be executed on observables – for example, the user might wish to select the ensemble of snapshots where particular observables fall within a certain range.

Because per-snapshot mechanical observables are able to fit nicely into tables and the user is often interested in running queries on these observables, putting these into a relational database makes a lot of sense. The problem is that we can't know beforehand what properties will be of interest for a given system, so designing a general fixed database schema is impossible. Instead, we can keep a database table for each system that is automatically augmented with more columns when new analyses are run; when the user asks a new observable to be computed from the snapshot data, another column needs to be added to the table. As long as this is automatically managed, this is fine. It would also be useful to try to keep the database schema as simple as possible, to avoid the spaghetti-like schema diagrams typical of large-scale database design.

There are other types of observables that are computed from simpler observables, such as time-correlation functions. These should still be stored in the database, but in a different table. [MORE HERE?]

Metadata. All the stuff left over that doesn't fit nicely into binary blobs or neat tables will be termed 'metadata'. This includes descriptions of what each of the observables in the database are, how they were calculated, and what they were computed from. This goes for the trajectory data too – we should store enough metadata to be able to reproduce the simulation.

The problem with metadata is that we cannot possibly know ahead of time what fields we are going to require to properly describe a simulation. Perhaps we want to run trajectories at a bunch of different temperatures in one case, or maybe different viscosities in another case. We have to avoid being trapped by a database schema. Because of this, it may be worthwhile to store the metadata in an XML-like format, where we can still search (possibly inefficiently) on the fields if required (using something like XPath), but the format is self-describing and extensible.

Requirements for a trajectory database:

A single standard format for data storage. One of the main goals is to simplify the coding and use of data analysis scripts. If coordinate snapshot data is stored in multiple native data formats (such as AMBER trajectory, CHARMM DCD, gromacs xtc, etc.) a different analysis program is required for each data format. Instead, all data is stored in a single standard format.

Import from multiple datasources. We can (as needed) write filters to import data from other sources, such as Vijay Pande's **Folding@Home** project.

Fast data storage and access. Because of the slow nature of storage systems (usually RAID's, often attached via slow NFS connections) storing data to or from disk can often be a bottleneck. To avoid this, the database should be able to exploit

large amounts of online memory, storing the entire dataset in memory or staging large portions in memory. Optimally, we would have a giant shared-memory multiprocessor machine where we could load the entire dataset into online memory and allow multiple processors direct access to the data.

Parallelization of analysis tasks. Many analyses have simple, natural parallel decompositions where a property is calculated for each snapshot or trajectory. A system that takes advantage of a shared-memory multiprocessor machine or computational cluster can divide up analysis tasks to achieve near perfect parallelization if applied to a large number of snapshots or trajectories, assuming the rate at which data can be provided to the CPUs is not a bottleneck. Parallelization of such analyses reduces the time needed for the user to ask scientific questions, accelerating the rate at which productive research can be carried out.

Transactional security. Failure of any operation should never leave the database in an unusable state. At worst, the last operation (such as depositing one trajectory segment worth of data) should need to be repeated.

Audit trails and records. Often, one dataset is analyzed in succession several times to produce results. If a problem is discovered in a simulation or analysis, all data derived from or with it needs to be recomputed. In order to do this, we need to keep track of the entire history of which datasets each piece of data was derived from and how it was computed. For reproducibility and documentation, we also need to have full records of the analysis code and input parameters used to produce each dataset. An analysis code shouldn't be allowed to be modified if data generated by that code is in the database – we must store and retain any version of the analysis code we used to generate data. (There may also be a nice way to propagate errors automatically if they are cleanly kept track of.)

User permissions. Eventually, we want to have a sophisticated user permission system where projects and datasets are accessible to users or groups of users. Datasets could be read-accessible to some, able to be added to by others, and changed or deleted by others.

Data deposition as the simulation progresses. Because it is slow to bring large amounts of data from the disk and push it over the network (such as when a large simulation completes at a remote supercomputing center and the data must be moved to our servers), it would be beneficial to be able to transfer generated data as a simulation progressed. Client nodes will be able to establish a data deposition session with the database, send snapshots as they are produced, and then signal when the trajectory segment is complete and it is safe to commit the data to the database. The user can also monitor progress of the simulation by looking at the current state of the database, and conduct analyses on partial datasets to get a better idea of how the simulation is going. This will allow for the early detection of problems or convergence, saving the researcher time.

Automatic analysis of incoming simulation data. Often, the researcher has some ideas before the simulation is run as to which observables will be of interest. These can be computed as data is sent to the database so that further analysis (such as WHAM) can be run on partially-completed simulations and convergence can easily be monitored as the simulation progresses.

Data replication and security. It should be convenient to automatically replicate the entirety or portions of the database to ensure data security. The raw data and records of all the analyses performed are in principle all that are needed to exactly reproduce the results of the calculation. If only one architecture is used to generate the data, only the initial restart files are needed in order to reproduce the raw data, but it may require a large amount of computer time to repeat the simulations. There may be some utility in just replicating periodic snapshots, where portions of the raw data can be regenerated (or even regenerated, saving data with a different time resolution) as needed.

Data sharing and export. It would be useful to make our data available to collaborators or to link database systems so that data can be shared among researchers or on the web. Export of the data both in general data interchange formats (HDF5?) and to common formats (PDB, AMBER, CHARMM, GROMACS, etc.) would be useful.

Data visualization. We need to make lots of plots and graphs of the data in the database, or look at what sorts of trajectories we have. Ideally, it would be easy to interface common data and molecular visualization programs (gnuplot, Pymol) or numerical analysis packages (Matlab) with the database.

The following is an example hierarchy for organization of data in the database.

Project. This is simply a way to organize a bunch of different simulations into a project for the research, such as “hydrophobic zippers”.

System. A system is a particular set of molecules in a box with a particular choice of settings as to how the energy is to be computed.

Simulation. A simulation is a particular way of sampling points in phase space for a particular system.

Trajectory segment. A simulation may contain many trajectory segments, which consist of a series of snapshots generated by Newton's equations of motion.

Snapshot. A snapshot is just a point in phase space, along with with computed values for some observables.

The actual organization will be flexible, so that there may be subprojects under projects, or different labgroups organized at the top-level. The goal would be to allow the users to design a desired arbitrary, hierarchical structure.

3.2 Common analysis tasks

There are number of common analysis tasks we would like to be able to perform. The basic idea is that, after the infrastructure is in place, we can write analysis scripts as needed, and these analysis scripts can be reused on other projects or even on different observables within the same project.

Snapshot operations. Snapshot analyses operate on the coordinates or some set of computed properties for a single snapshot in time. These are also sometimes referred to as *order parameters* or *mechanical observables* because they can be in principle be computed from the phase space coordinates of the system at a single instance in time.

Radius of gyration. This is a measure of the relative compactness of some set of atoms. *Parameters:* atom selection for which the radius of gyration is to be computed.

RMSD (root mean squared deviation) to a reference structure. A measure of the similarity of two structures. *Parameters:* target structure, atom selection, whether or not to LSQ align, whether or not we also want the transformation matrix that aligns the structures or the resulting aligned coordinates.

DME (distance-matrix error) to a reference structure. Another measure of structural dissimilarity based on the difference of distance matrices between two structures. *Parameters:* similar to RMSD.

Hydrogen bonds. The specific hydrogen bonds present in a particular structure. *Parameters:* hydrogen bond pairs to evaluate, hbond definition.

Contacts. The specific interresidue contacts present in a particular structure. *Parameters:* contact pairs to evaluate, contact definition.

Set operations. Some properties consist of sets, such as the set of hydrogen bonds or contacts present in a given structure. The full set of set operations should be available.

NMR observables. NOEs, chemical shifts, and coupling constants.

Fluorescence spectroscopy observables. FRET efficiency, Trp fluorescence (basically ASA).

SASA (solvent accessible surface area).

Circular dichroism (CD) spectra.

Infrared (IR) spectra.

Internal coordinates. Internal coordinates consist of bond lengths, angles, impropers, and torsions. *Parameters:* which internal coordinates to compute.

Discretization of an observable or set of observables. Often, further analysis requires that one or more observables be discretized in some fashion and the resulting bins indexed. *Parameters:* bin definitions.

Projection of Cartesian or torsion coordinates onto provided axes (like principal coordinates).

Trajectory operations. These analyses operate on an entire trajectory at a time.

Time-correlation functions.

Ensemble operations. These analyses operate on an entire ensemble of snapshots at a time. Usually, these operations take as input the result of another analysis (a scalar or a bin index) rather than the Cartesian coordinates directly.

Weighted histogram analysis (WHAM).

Statistical operations. (min, max, mean, variance)

Transition matrix.

Covariance matrix.

Conformational (or dynamical) clustering.

Calculation of potentials of mean force.

Auxiliary mathematical operations. Eigenvalues, eigenvectors. (for quasi-harmonic analysis, etc)

Requirements for analysis system:

Simple interfaces for rapid coding of analysis scripts. We wish coding of a new analysis function to be quick and easy. For reasons of portability, security, fault tolerance, and efficiency, analysis codes will be written in Java. (There is evidence Java code can be as efficient as C, if properly designed. It is also easy to design reusable libraries in Java.) There are three main types of analyses — analyses conducted by snapshot (such as computation of a structural observable like radius of gyration given a single coordinate snapshot), analyses conducted on whole trajectories (such as the calculation of time-correlation functions of an observable given a time-sequence of values of that observable), and analyses conducted on ensembles of snapshots (such as weights computed from weighted histogram analysis). We can provide base classes for each of these types of analysis operations, leaving the programmer only needing to implement the bits that are different for each individual type of analysis.

Reuse of analysis scripts. The analysis scripts can be housed in a database that the user can browse through to select which analysis code to use for analysis or to use as the starting point for writing a new analysis code.

Dependency awareness and (possibly) automatic resolution. Analysis codes should be aware of what kinds of data they can operate on. If there are dependencies that are unfulfilled, perhaps they can be automatically computed first. (For example, if we wish to compute the time-correlation function of a particular hydrogen bond, the hydrogen bond formation analysis must first compute the observable for all snapshots before the time-correlation function analysis code can process the observables.

Universal input format for analysis codes. Analysis codes will receive their parameters in XML form.

Parallel execution of analysis tasks. Just as simulation tasks are farmed out to remote compute nodes, we should be able to farm out analysis tasks to compute nodes. Several types of parallelism can be exploited: by-snapshot analyses (such as computation of structural observables like the radius of gyration) can be parallelized at the snapshot level and the entire snapshot pool can be divided in any way; by-trajectory analyses (such as the computation of time-correlation functions) require division by trajectory; finally, aggregate analyses (such as computation of free energy landscape projections with WHAM) require all of the data. The drawback here is that large amounts of data (raw simulation snapshots) usually need to be pushed out over the network to client machines, which may be a bottleneck.

Examples of common analyses our group might perform:

Potential of mean force of contact formation. A parallel-tempering simulation of a peptide fragment is conducted. Some interresidue distances are computed, and a multidimensional potential of mean force as a function of one or more of these distances is computed.

Transition matrix in two structural observables. A parallel tempering simulation of a peptide fragment is conducted. Two structural observables are computed (number of beta sheet hydrogen bonds and radius of gyration of the hydrophobic core). A set of bins in these two observables is defined and the transition probabilities among these bins for various lag times are computed, later used to estimate the transition rate. When compared to the potential of mean force, such an analysis can answer whether or not folding kinetics can be inferred from the thermodynamic landscape pictured by the potential of mean force in these two observables.

Hydrophobic zippers landscape projection. A parallel tempering simulation of a protein is conducted. The set of hydrophobic contacts present in each snapshot is computed. The relative free energy of each contact set is computed by WHAM, and these are plotted on a tree-like graph of the contact sets that represent zipping states.

4 Random notes

4.1 Bag-of-tasks applications for computational grids

The simplest method of implementing a job scheduling system for a heterogeneous computational grid seems to be the “bag-of-tasks” philosophy. Here, a number of servers can put tasks that need to be executed into a “bag”. Clients take tasks from the bag (checking first to see if they match certain criteria) and execute them, returning for more tasks when finished. Various implementations provide such features as transactional security, persistence, and notification services.

Sun Javaspaces. JavaSpaces is the standard tuple space implementation for Java, but seems to be rather complicated.

<http://java.sun.com/developer/technicalArticles/jini/javaspaces/>

IBM TSpaces. TSpaces appears to be easier to get running and to debug applications with, as a web interface to view tasks in the space is provided. TSpaces can also provide task persistence. Unfortunately, TSpaces is not open source, and may need to be licensed from IBM if we intend to use or redistribute it.

<http://www.almaden.ibm.com/cs/TSpaces/>

We should look into these two systems to see which would be best for us. Or maybe we want to write our own intelligent job scheduler.

4.2 Java class hierarchy

Here is an example class hierarchy for client-based processing tasks.

Task. An abstract class representing a task that will be executed by a remote machine. Includes a field describing what the task is (so that clients only need to take tasks they are capable of executing). Other class fields provide useful support data, such as the application responsible for creating the task, the user it belongs to, the creation date, etc.

AnalysisTask. An abstract class representing an analysis to be performed on data in the database. This class includes information on how to connect to the simulation database, how to obtain the data and deposit the results, etc. Methods for the application manager to query the script about what kinds of data it can operate on are defined here.

SnapshotAnalysisTask. An abstract class for analyses that operate on individual snapshots, and so can be parallelized by distributing sets of snapshots to individual compute nodes. The framework for executing an analysis function on data from a single snapshot, looping over the set of snapshots, is set up here, so the user only has to write an initialization method, a per-snapshot computation method, and a finalization method.

RadiusOfGyration. A concrete class for computing the radius of gyration of a selection of atoms.

RMSD. A concrete class for computing the RMSD to a reference structure for a subset of atoms, with the option of LSQ-aligning the structure first, or saving the LSQ-aligned structure.

TrajectoryAnalysisTask. An abstract class for analyses that operate on whole trajectories. Again, an initialization method, a per-trajectory method, and a finalization method are provided.

TimeCorrelationFunction. A concrete class for computing the time-correlation function for a pair of observables.

EnsembleAnalysisTask. An abstract class for analyses that operate on an entire ensemble of snapshots or trajectories.

WHAM. A concrete class that computes simulation free energies and per-snapshot (or -trajectory) weights. (Elaborate more on this later.)

DynamicsTask. An abstract class for generating a molecular dynamics trajectory from an initial point in phase space. Subclasses use a particular molecular dynamics program to generate the trajectory. Each subclass is basically an adaptor, transforming the universal system description and simulation input files into the program-specific input, running the program, and transforming the output back into the universal format for transmission to the server.

AmberDynamicsTask. Transforms input (if possible) into parameter and control files for AMBER7's sander package, and converts output to universal data storage format. Note that not every simulation algorithm is supported by AMBER, so some universal system specifications are not able to be represented in sander.

GromacsDynamicsTask. Uses the `gromacs` package to perform molecular dynamics.

CharmmDynamicsTask. Uses the `CHARMM` package to perform molecular dynamics.

LammpsDynamicsTask. Uses the `LAMMPS` package to perform molecular dynamics.

There will be a number of utility classes as well:

AtomSelection. Many analysis tasks operate on a user-specified subset of atoms, given in the XML document that specifies the parameters necessary to execute the analysis. We need a class able to parse these human-readable text-based selection strings and (given the system description) determine what subset of atoms the user has specified. We can probably use an established syntax for this, perhaps from something like MIDAS or carnal.

SnapshotData. We will need a class to encapsulate various properties for a single snapshot that an AnalysisTask may want access to. For example, if we need the atomic coordinates, or the radius of gyration and RMSD to a given structure, this class can encapsulate this data and provide access to it. The data will be stored by a unique key in something like a Map data structure.

4.3 Masters

A Master is responsible for creating new Tasks (such as molecular dynamics trajectories) to run on the distributed system. Users can easily write new Masters, debug them in the development environment, and add them into the running production system. Instances of the Master can then be started by any user. Masters are persistent, in that they will continue to execute until completion or receiving additional messages from the user. Masters may also spawn other Masters or receive communication from other Masters or the user during their execution.

User-created Master classes must be concrete subclasses of the Master abstract base class (or one of its subclasses). The Master class both specifies several methods that must be implemented by any concrete subclass and provides a number of methods for submitting new tasks to the space, retrieving results, sending and receiving messages to other masters, receiving notifications from the user or scheduler, or spawning new Masters. The Master base class also contains some private data fields for keeping track of the Master's unique ID, creation date, etc., though these can be read through public accessor methods.

- The Master class contains a number of abstract public methods that must have concrete implementations in any concrete base class:

`void initialize(Properties properties) throws MasterInitializationException;`

This method is called first by the MasterLauncher application, which creates new Master objects, initializes them, and deposits them in the space. The method should therefore be package scope, if MasterLauncher is to be in the same package.

A Properties object containing user-specified configuration information is passed to the Master, and if this does not contain sufficient information to properly configure the Master (or if there is some error in the specification of configuration information) the Master should throw a MasterInitializationException, indicating an error should be returned to the user and Master execution should not proceed.

* Note: The Properties object is a simple set of key-value pairs. It may be a better idea to use a more powerful data storage format for specifying configuration information, such as an XML document. However, we do want to keep the code required to process the configuration information as simple as possible. Would the extra code required to manipulate XML DOM trees be worth the added flexibility?

`String getDescription();`

Returns a human-readable description of what this Master is intended to do.

`void notify() throws MasterExecutionException;`

This method is called by the MasterController when some event (or events) that the Master may want to be aware has occurred. This includes the availability of results, arrival of messages, or requests from the user to modify the parameters of execution. `notify()` may be called at any time, and it is not necessarily the case that the Master will want to take results or generate new Tasks during any given call. Instead of processing results individually, for example, the Master may want to wait until all of its results are complete before removing them from the space and analyzing them.

* Note: Should a log of events that have transpired be passed to the Master in `notify()` to facilitate processing?

A MasterExecutionException should be thrown if a random error prevents execution from proceeding properly on this client, but there is some chance that the Task can execute properly on another client. The transaction should be aborted, returning the Task to the space. We may want to increment an ExecutionException counter in the Task to prevent the same Task from bouncing back and forth too many times, though.

If an unrecoverable error occurs during Task execution where no client would be able to properly execute the Task (due to incorrect input parameters, for example), a Result should be returned with appropriate error fields set and the transaction should be committed.

If the Master chooses to actually process the tasks, it can make use of several methods implemented in the Master base class (see below) to actually remove results from the space, retrieve messages, submit new tasks, etc.

A Master is guaranteed to only have one instance executing at a time. All operations performed (results taken, tasks generated, messages taken, placing the updated Master back in the space) are also guaranteed to be transactionally secure, so that the system will never be left in an incoherent state.

The Master object will likely be stored in a "sleeping" state between calls to notify() (likely by storing it in a JavaSpace Entry), so the class cannot have other threads executing after notify() has completed. All processing must occur within notify().

* Note: Is it best to have a single notify() method, requiring the Master to check what has happened, or to have separate notifyResult(), notifyMessage(), etc. methods to notify of different things?

String getBriefStatusReport(); String getVerboseStatusReport();

This method should return a human-readable status report containing information about the progress of the calculation the Master is performing. It is basically a way to return the internal state of the Master in human-readable format.

The brief status report is limited to a single line. The Verbose status report is a detailed multiline description.

- The Master base class also provides a number of protected methods that govern access to and creation of results, tasks, and messages. Only those that belong to or are sent to this particular Master instance are accessible to prevent unwanted interference between Master objects during execution.

* Note: These methods need the most work as far as defining and fleshing them out.

String submitTask(Task task)

Submit a Task. The globally unique Task ID is returned.

List getSubmittedTaskList()

Returns a list of the IDs of all the Tasks this Master has submitted.

List getPendingTaskList()

Returns a list of the IDs of all Tasks still waiting to begin execution.

terminateTask(String taskId)

Terminates a pending or executing task given the unique task ID.

* Note: Should we be able to terminate a task that is currently running on a client?

* Note: Masters that are terminated should have their Tasks automatically purged from the space.

String spawnMaster(String masterClassName, Properties properties)

Spawn a new Master given the class name and the properties object to configure it.

List getSpawnedMasterList() terminateSpawnedMaster(String masterId)

Methods for spawning new Masters and terminating them.

List getMessageList() Message getMessage(String messageId) sendMessage(String recipientId, Message message)

Methods for sending and receiving messages.

List getResultList()

Returns a list of completed Result IDs. These Result IDs match the Task IDs.

* Note: If a Master is destroyed, its Results should be purged. Suggests TaskEntry and ResultEntry should be descended from same base Entry class that contains a creatorId field so we can search for all Entries created by a Master when we terminate it.

Result takeResult(String resultId)

Removes a result from the space.

Result readResult(String resultId)

Reads the result from the space without taking it. The Master will eventually want to take the result from the space, but may chose to do this in batches.

* Note: What is the role of Results versus data in the simulation database?

* Note: Later, we must also provide methods for the Master to access the simulation database.

- There are also some general utility methods that are declared as package scope.

String getId();

This method returns the globally unique ID corresponding to this instance of the Master.

String getCreatorId();

Returns the globally unique ID of the human or Master that created this instance of the Master.

String getCreatorUsername();

Returns the username of the human that created this Master. If the Master was spawned by another master, this value is inherited.

Date `getCreationDate()`;

Returns the date this Master instance was created.

Date `getLastUpdateDate()`;

Returns the last time the internal state of the Master was updated.

RunningStatus `getRunningStatus()`;

Some sort of internal base class field will be able to be set to change whether the Master should be running or held (suspended).

* Note: Many of these are set in the class constructor, but we will need `set...()` functions for some of these. What should the scope be? Where will they be called?

* Note: Eventually, we will want to add some sort of permission system, and so we will need methods that query the permissions this Master has.

- As a matter of specific implementation, Master objects are encapsulated in a `MasterEntry` class (which implements the `Entry` interface) for persistent storage in a `JavaSpace`. Several fields are replicated in this wrapper class so that they may be matched when requesting Masters from the space for retrieval. These fields are:

`id` - The Master object's unique ID.

`status` - The status of the Master object, currently either "running" or "suspended".

`creationUser` - The name of the user who created the Master object.

`creatorId` - The ID of the user or Master who created the Master object.

* Note: Are there more fields we may want to search on?

The `MasterEntry` class provides a convenience constructor to populate its class fields (which must, of course, be public due to the constraints of `JavaSpaces`) given a Master object as an argument. The public accessor functions of the Master described above (such as `getId()`) are called to populate these class fields.

* Note: What happens to Masters when they terminate?

4.4 Tasks

All executable tasks are described by classes that extend the `Task` abstract base class.

- There are a number of abstract methods that must be implemented by subclasses:

Result `execute()` throws `ClientMisconfigurationException`;

The actual work is done in the `execute()` method. If execution is successful, the `Result` object returned is deposited in the same space the `Task` was retrieved from and the transaction is committed. If something goes wrong during execution but this is not the fault of misconfiguration of the client but rather a bad set of command arguments or something of the like, the `Result` will contain error information for the Master to process upon receipt. If something has been misconfigured on this particular client, but it is possible another client may successfully be able to execute the task, a `ClientMisconfigurationException` is thrown and the transaction is aborted, returning the `Task` to the space. If the client fails during execution or is disconnected from the network, the lease on the transaction will expire, also returning the task to the space.

All executable tasks are described by classes that extend the `Task` abstract base class.

Only one method must be implemented in a concrete subclass:

Result `execute()` throws `ClientMisconfigurationException`;

The actual work is done in the `execute()` method. If execution is successful, the `Result` object returned is deposited in the same space the `Task` was retrieved from and the transaction is committed. If something goes wrong during execution but this is not the fault of misconfiguration of the client but rather a bad set of command arguments or something of the like, the `Result` will contain error information for the Master to process upon receipt. If something has been misconfigured on this particular client, but it is possible another client may successfully be able to execute the task, a `ClientMisconfigurationException` is thrown and the transaction is aborted, returning the `Task` to the space. If the client fails during execution or is disconnected from the network, the lease on the transaction will expire, also returning the task to the space.

* Note: Should the client drop the capability or stop if a `ClientMisconfigurationException` is encountered?

* Note: Exception propagation back up to Master? Maybe in RMI-type execution...

* Note: How does a client know if it can execute this type of task or not? The client may only have certain software packages locally configured, such as AMBER but not CHARMM. Suppose the client did have a list of these capabilities (passed in through a configuration file, perhaps): Should there be a specific "requirements" field in the `TaskEntry` that the client must match on? Would the client have to try matching on each of its capabilities in turn? (e.g. first try to take a `TaskEntry` that

matches "AMBER", then try to take a TaskEntry that matched "CHARMM", repeating the cycle?) This seems inefficient. On the other hand, taking a TaskEntry at random and then checking this field also seems inefficient. Due to the space behavior, if we did this, we might take the same TaskEntry over and over again anyway.

The client will attempt to take Tasks matching a template with each capability the client provides, in turn. We just need to make sure that the client doesn't repeatedly poll the space quickly when no jobs are available – we need a delay to avoid hammering the space. Eventually, there can be a matching service that matches clients with tasks.

A few package scope (concrete) methods are provided to access data in the Task base class:

String getId();

This method returns the globally unique ID corresponding to this Task.

String getCreatorId();

Returns the globally unique ID of the human or Master that created this Task.

String getCreatorUsername();

Returns the username of the human that was indirectly responsible for the creation of this Task.

Date getCreationDate();

Returns the date this Task was created.

RunStatus getRunStatus();

Returns the run status of this Task. (* Note: Is it necessary for the Task class to have this field, or just the TaskEntry class?

Use an Enum?)

String getRequirements();

Returns the client requirements needed to execute this task. (* Note: How do we handle this?)

* Note: The Task will be stored in a TaskEntry class (which implements the Entry interface) for storage in the JavaSpace.

Some fields will be replicated, since we will need to match them:

id - The globally unique ID of this Task.

creatorId - The globally unique ID of the Master that created this Task.

status - The run status of this Task.

requirements - Some statement of the client-side requirements for executing this task.

–

We will also want the Worker (or compute client) to be able to communicate with the servers in some way.

To know how many compute clients are connected, for example, we could have all the compute clients deposit a ClientEntry in the JavaSpace, with a LeaseManager renewing the lease every 1-5 minutes. This ClientEntry can contain information about the client, such as:

String clientId;

A globally unique ID for this client. This ID can be used to direct messages to this particular client.

String hostname;

The hostname of the machine.

String cpuinfo;

Information about the CPU executing the code.

Date clientStartDate;

The Date the client code was started on this machine.

String taskId;

The globally unique ID of the Task currently executing on this client. (* Note: What if we have no Tasks or multiple Tasks executing on this client, as in a multithreaded environment?)

Date taskStartDate;

The Date which the Task began execution.

String status;

Some description of the status of the client.

String capabilities;

Some characterization of the capabilities (in terms of installed executables) of the client.

–

The Client will also register a handler to be run (in a separate thread) when messages destined for this client arrive in the JavaSpace. This will allow us to receive messages to shut down the client or delete a currently running Task. (* Note: What else might we want to communicate in a message? An update of client configuration or capabilities?)

–

4.5 Notes on available database packages

4.5.1 XML database

There are several database systems that provide XML database functionality.

dbXML. From the dbXML webpage:

dbXML is a Native XML Database. It is capable of storing and indexing collections of XML documents in both native and mapped forms for highly efficient querying, transformation, and retrieval. In addition to these capabilities, the server may also be extended to provide business logic in the form of scripts, classes and triggers.

<http://www.dbxml.com/product.html>

Berkeley DB XML. The Berkeley DB XML provides XML document storage capability along with the transactional security of Berkeley DB. Nice features include the integration of transactions with storage of binary blobs in the Berkeley DB. XPath searching for matching documents is available, but index generation needs to be requested manually. In order to update documents, they must be modified by the user and replaced in the store – no XUpdate or similar is provided. In recent versions, there is a new API that allows for something like updating documents in place.

From the Berkeley DB XML website:

Berkeley DB XML is an application-specific native XML data manager built on Berkeley DB, the world's most widely deployed data management engine. Berkeley DB XML provides fast, reliable, scalable and cost-effective storage and retrieval for native XML data and semi-structured data.

Berkeley DB XML is supplied as a library that links directly into the application's address space. This provides superior performance by eliminating bottlenecks that occur in client-server systems.

Berkeley DB XML stores XML documents in collections. A single application may operate on many collections at the same time. A single application may also combine data from different collections easily. Non-XML data may be included by creating standard Berkeley DB tables. Tables and collections may be used together, with full support for Berkeley DB transactions and recovery services, by multiple users simultaneously.

Berkeley DB XML enables fast look up by allowing individual collections to be indexed differently. This allows Berkeley DB XML to speed up the common queries over particular collections. Each collection supports multiple indexes. A wide variety of available indexing schemes support different XPath queries efficiently.

Berkeley DB XML's Query Processor implements XPath 1.0. A cost-based query optimizer considers the indices that exist, the data volume that a query is likely to produce and the cost of computation and disk I/O to select a query plan with the lowest run-time cost.

<http://www.sleepycat.com/products/xml.shtml>

4.5.2 Relational databases

PostgreSQL. PostgreSQL is a standard sort of relational database with transaction security and support for binary blob data with minimal overhead.

<http://www.postgresql.org/>

4.5.3 Binary storage databases

Berkeley DB. From the Berkeley DB website:

Berkeley DB Data Store is a high-performance scalable, embedded data management engine that links directly into the address space of the application that uses it. There is no separate server to install or administer. The application makes a simple function call, rather than sending a message to a remote server, to store, fetch, or modify records. All the work is done in a single address space for higher performance.

Berkeley DB Data Store is a library. It allows the application developer control over almost everything it does. The amount of memory dedicated to caching records, the on-disk storage structure used for individual tables, and more are configurable at runtime.

Berkeley DB Data Store can manage enormous amounts of data - up to 256 terabytes in any single table, with single records as long as two gigabytes. Applications can operate on one or more tables. Any Berkeley DB Data Store application can support one or more readers working with the database simultaneously. Any writer must have exclusive access to the database.

<http://www.sleepycat.com/products/data.shtml>

4.6 Data interchange notes

Some possibilities for data interchange formats:

HDF5. HDF5 is a hierarchical, extensible data storage format, much like XML, for numerical data. While intended for C, it has a nice F90 interface library and a reasonable Java interface. <http://hdf.nsa.uiuc.edu/HDF5/>

4.7 Database class

The Database class provides a connection to simulation database, isolating the various actual databases used (XML, RDBMS, and binary database) from the classes that need to access the database. This class is expected to interact with various other classes that act as servers for receiving and sending data. The Database class must be thread-safe, in that several instances can run simultaneously without causing trouble. (Do we intend Database to only be instantiated on the same machine that the actual databases are running?)

All Database operations catch exceptions from the various actual databases and throw a DatabaseException exception to isolate the class using Database from the actual database implementations.

Database(Properties config) This is the only allowed constructor. Information about how to connect to the various actual databases is provided in config – this information may be read from a configuration file or passed over the network.

create() Initializes the simulation database from scratch. Practically, this amounts to creating a new collection in the XML database and creating a projects.xml document to list the projects stored in the XML database:

```
<projects>
</projects>
```

A new empty RDBMS database is created, as well as a BerkeleyDB database.

destroy() Destroys the existing database.

connect() Connects to the actual databases. If a problem is encountered in connecting to any of the actual databases, the databases successfully connected to will have to be disconnected from gracefully.

disconnect() Disconnects from the actual databases.

createSubproject(String projectXML) Create a new project under

4.7.1 BinaryDatabaseConnection

4.7.2 RdbmsDatabaseConnection

4.7.3 XmlDatabaseConnection

4.8 Security and user access restriction

In order to prevent accidental (or malicious) data destruction, as well as to maintain security and privacy for the storage of simulation data, a UNIX filesystem-like system for handling read, write, deposition permissions will be added.

User database. A database of users will be stored in an XML file.

Group database. A number of groups (representing lab groups, lab subgroups, collaboratory networks) can be set up.

Access restrictions. Various aspects of reading and writing all data can be set.

Administration privileges. Some users can be given administration privileges.

4.9 Database transaction examples

4.9.1 Despoiting data from a trajectory

This is just an example of how a transaction between a client (a node generating a molecular dynamics trajectory, for example) and the database might go, just to give us an idea as to what problems we might run into.

1. The compute client makes a connection with the database manager.
2. The client transmits information about which project and simulation this trajectory is associated with.
3. New transactions are created for modifying this project in the XML, relational, and binary databases. (Note that we can't lock the databases if we are planning on transmitting data as it is generated – we either need the transactions to fail if the database has changed in a way that does not allow our transaction to be committed, or we need a more sophisticated intermediate storage form so that we only need to lock the database for a short period of time.)
4. Information about this trajectory segment is transmitted in XML form and stored in the XML database.
5. System state snapshots, coordinates, velocities, and forces that are transmitted are stored in the binary database, and the keys stored in the XML and relational database.
6. Other per-snapshot data is stored in the relational database.
7. The transactions for each database are committed. If any one fails, the transactions that have completed are rolled back, and an exception is transmitted.

We may have to be more clever to deal with the database system not changing state while we are accumulating data. We could store the data in a temporary database or separate region of the database until we have accumulated all the data, and then incorporate it all later.

Think about having 'gatekeeper' databases to collect data from a supercomputer and relay it back to us?

4.10 XML stuff

(THIS SECTION STILL UNDER CONSTRUCTION.)

Updated 2004-04-08. This is the new XML hierarchy.

There will be system files:

```
<!-- A system is the fundamental unit for organizing data from a single molecular mechanics sy
<system>
  <!-- A human-readable description. -->
  <description>C-terminal beta hairpin from protein G</description>

  <!-- A remote key telling us where to find the system description, which contains the force
  <systemDescription location='binaryDatabaseKey' remoteKey='GUID' />

  <!-- Information on a simulation. -->
  <simulation key='GUID'>
    <description>Replica-exchange simulation starting from the experimental structure</descrip

    <!-- The simulation can contain a number of subsimulations, subsubsimulations, etc. -->
    <subsimulation>
```

```

    <!-- A trajectory is the smallest unit in the XML database. Trajectory data is stored in
    <trajectory location='xmlDatabase' remoteKey='GUID' />
    <trajectory location='xmlDatabase' remoteKey='GUID' />
</subsimulation>
<subsimulation>
    <trajectory location='xmlDatabase' remoteKey='GUID' />
</subsimulation>
</simulation>

<!-- A property stores metadata about the different columns in the RDBMS. A property may re
<property key='GUID'>
    <!-- The class is what other analysis scripts use to verify whether or not they can take t
    <class>TensorProperty</class>

    <!-- A general descriptive term for the type of analysis. -->
    <name>PressureTensor</name>

    <!-- The actual datatype for the property -- how it is stored by the machine. -->
    <dataType>float[3][3]</dataType>

    <!-- A human-readable description. -->
    <description>The pressure tensor for the simulation box.</description>

    <!-- The name of the user who ran the analysis. -->
    <creator>jchodera</creator>

    <!-- The date the data was generated. -->
    <creationDate>2004-01-22 25:04:16 PST</creationDate>

    <!-- The thing that generated the stuff deposited in this column. 'simulation' means the
    <source type='simulation' />
</property>

<property key='GUID'>
    <class>ScalarProperty</class>
    <name>PotentialEnergy</name>
    <type>double</type>

    <!-- The units the property is stored in, to assist in automatic conversion between differ
    <units>kcal/mol</units>

    <description>The potential energy.</description>
    <creator>jchodera</creator>
    <creationDate>2004-01-22 25:04:16 PST</creationDate>
    <source type='simulation' />
</property>

<property key='GUID'>
    <!-- 'name' is a unique identifier for each type of analysis -->
    <class>ScalarProperty</class>
    <name>RadiusOfGyration</name>
    <type>float</type>
    <!-- 'description' is a human-readable description -->
    <description>Radius of gyration of they hydrophobic core</description>

```

```

<creator>jchodera</creator>
<creationDate>2004-01-22 25:04:16 PST</creationDate>
<source type='analysis'>
  <classname>RgAnalysis</classname>
  <version>0.2.4</version>
  <parameters>
    <atom-selection>SIDECHAIN and (RES 3 or RES 5 or RES 14 or RES 16)</atom-selection>
  </parameters>
</source>
</property>

<property key='GUID'>
  <name>binIndex</name>
  <type>integer</type>
  <description>The bin index for an Rg-RMSD free energy projection.</description>
  <creator>jchodera</creator>
  <creationDate>2004-01-22 25:04:16 PST</creationDate>
  <source type='analysis'>
    <classname>Bin</classname>
    <version>0.1.7</version>
    <parameters>
      <axis propertyKey='GUID' min='3.0' max='7.0' numberOfBins='10' />
      <axis propertyKey='GUID' min='0.0' max='15.0' numberOfBins='10' />
    </parameters>
  </source>
</property>

</system>

```

Then trajectory files:

```

<trajectory key='GUID'>
  <replica>7</replica>
  <iteration>13</iteration>
  <temperature units="Kelvin">300.0</temperature>
</trajectory>

```

4.11 Older notes

Since we will be receiving a trajectory at a time from the client machines, each trajectory will be represented by a separate XML document stored in a /trajectories part of the XML repository for this simulation or system. The server will build up the XML file as it receives snapshots from the clients performing the molecular dynamics simulations. (The clients transmit the data according to a defined protocol – this way, we can change the server side of things and not need to worry about changing the clients.)

The XML trajectory file will hold some information about the trajectory that does not easily fit into the relational database. This includes per-trajectory information on how the trajectory was generated, from which initial snapshot, by which client machine, at what time, how long the simulation took, etc.

The relational database will have a row for each snapshot in the simulation. The columns will hold different observables that are to be recorded for each snapshot. The meaning of each column (along with a record of how it was calculated from the raw data) will be described by another XML file in the repository.

System state snapshots will be stored in the binary database, and the keys will go into a column in the relational database. (We could then, for example, perform queries for snapshots that have a non-null entry in this column, meaning that there is a system state snapshot present in the binary database.)

A Appendix

AMBER7[2]

References

- [1] A. Brass, B. J. Pendleton, Y. Chen, and B. Robson. Hybrid Monte Carlo simulations theory and initial comparison with molecular dynamics. *Biopolymers*, 335:1307–1315, 1993.
- [2] D. A. Case, D. A. Pearlman, J. W. Caldwell, T. E. Cheatham III, J. Wang, W. S. Ross, C. L. Simmerling, T. A. Darden, K. M. Merz, R. V. Stanton, A. L. Cheng, J. J. Vincent, M. Crowley, V. Tsui, H. Gohlke, R. J. Radmer, Y. Duan, J. Pitera, I. Massova, G. L. Seibel, U. C. Singh, P. K. Weiner, and P. A. Kollman. Amber7, 2002.
- [3] Christoph Dellago, Peter G. Bolhuis, and Phillip L. Geissler. Transition path sampling. *Adv. Chem. Phys.*, 123:1–78, October 2002.
- [4] M. Scott Shell, Pablo G. Debenedetti, and Athanassios Z. Panagiotopoulos. Generalization of the wang-landau method for off-lattice simulations. *Phys. Rev. E*, 66:056703, 2002.
- [5] M. Scott Shell, Pablo G. Debenedetti, and Athanassios Z. Panagiotopoulos. An improved monte carlo method for direct calculation of the density of states. *J. Chem. Phys.*, 119(18):9406–9411, 2003.
- [6] Yugi Sugita and Yuko Okamoto. Replica-exchange molecular dynamics method for protein folding. *Chem. Phys. Lett.*, 314:141–151, November 1999.
- [7] William C. Swope, Jed W. Pitera, and Frank Suits. Describing protein folding kinetics by molecular dynamics simulations: 1. theory. *J. Phys. Chem. B*, 108:6571–6581, 2004.
- [8] William C. Swope, Jed W. Pitera, Frank Suits, Mike Pitman, Maria Eleftheriou, Blake G. Fitch, Robert S. Germain, Aleksandr Rayshubski, T. J. C. Ward, Yuriy Zhestkov, and Ruhong Zhou. Describing protein folding kinetics by molecular dynamics simulations: 2. example applications to alanine dipeptide and a beta-hairpin peptide. *J. Phys. Chem. B*, 108:6852–6594, 2004.