JCTC Journal of Chemical Theory and Computation
Article
pubs.acs.org/JCTC

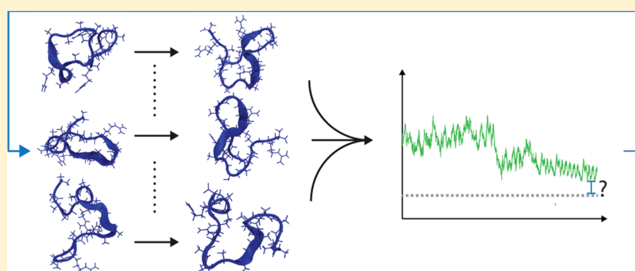
Molecular Simulation Workflows as Parallel Algorithms: The Execution Engine of Copernicus, a Distributed High-Performance Computing Platform

Sander Pronk,[†,§] Iman Pouya,[†,§] Magnus Lundborg,[‡] Grant Rotskoff,[†] Björn Wesén,[†] Peter M. Kasson,[¶] and Erik Lindahl*,[†,‡]

[†]Swedish eScience Research Center, Department of Theoretical Physics, KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden

[‡]Department of Biochemistry and Biophysics, Science for Life Laboratory, Stockholm University, SE-106 91 Stockholm, Sweden

[¶]Department of Molecular Physiology and Biological Physics, University of Virginia, Charlottesville, Virginia 22908-0376, United States



ABSTRACT: Computational chemistry and other simulation fields are critically dependent on computing resources, but few problems scale efficiently to the hundreds of thousands of processors available in current supercomputers—particularly for molecular dynamics. This has turned into a bottleneck as new hardware generations primarily provide more processing units rather than making individual units much faster, which simulation applications are addressing by increasingly focusing on sampling with algorithms such as free-energy perturbation, Markov state modeling, metadynamics, or milestoning. All these rely on combining results from multiple simulations into a single observation. They are potentially powerful approaches that aim to predict experimental observables directly, but this comes at the expense of added complexity in selecting sampling strategies and keeping track of dozens to thousands of simulations and their dependencies. Here, we describe how the distributed execution framework Copernicus allows the expression of such algorithms in generic *workflows*: dataflow programs. Because dataflow algorithms explicitly state dependencies of each constituent part, algorithms only need to be described on conceptual level, after which the execution is maximally parallel. The fully automated execution facilitates the optimization of these algorithms with *adaptive sampling*, where undersampled regions are automatically detected and targeted without user intervention. We show how several such algorithms can be formulated for computational chemistry problems, and how they are executed efficiently with many loosely coupled simulations using either distributed or parallel resources with Copernicus.

1. INTRODUCTION

The performance of statistical mechanics-based simulations in chemistry and many other fields has increased by several orders of magnitude with faster hardware and highly tuned simulation codes.[1−3] Conceptually, algorithms such as molecular dynamics are inherently parallelizable since particle interactions can be evaluated independently; however, in practice, it is a very challenging problem when the evaluation must be iterated for billions of dependent time steps that only take a fraction of a millisecond each. Large efforts have been invested in improving performance through simplified models, new algorithms, and better scaling of simulations,[4−7] not to mention special-purpose hardware.[8,9]

Most force fields employed in molecular dynamics are based on representations developed in the 1960s that only require a few dozen floating-point operations per interaction.[10] This provides high simulation performance, but it limits scaling for small problems that are common in biomolecular research. With a few thousand particles, there are not enough floating-point operations to spread over 100 000 cores within less than a millisecond, no matter what algorithm or code is used. This limit to strong scaling is typically expressed in a minimum number of atoms/core and is becoming an increasingly challenging barrier as computing resources increase in core numbers. Computational power is expected to continue to increase exponentially, but it will predominantly come from increased numbers of processing units rather than faster individual units, including the use of Graphics Processing Units (GPU) and similar accelerators.[11]

One potential solution to this problem derives from the higher-level analyses commonly used for simulations. In computational chemistry and related disciplines, a study almost never relies on a single simulation trajectory—multiple runs are used even in simple studies for uncertainty quantification and for comparison between conditions. Furthermore, sampling and ensemble techniques[12−17] are increasingly used to combine many simulation trajectories into a higher-level model that is

Received: March 12, 2015
Published: April 29, 2015

ACS Publications © 2015 American Chemical Society

DOI: 10.1021/acs.jctc.5b00234
J. Chem. Theory Comput. 2015, 11, 2600−2608

then compared to experimental data. This presents an opportunity for increased parallelism across simulation trajectories as well as within each trajectory. Simulation trajectories need not be completely independent, as some algorithms rely upon data exchange between simulations, but they should be loosely coupled compared to the tight coupling within simulations. This looser coupling permits efficient parallelization over much larger core counts and potentially higher latency interconnects than would be practical for a single simulation trajectory with a comparable number of atoms.

In this paper, we describe the execution engine of Copernicus:[18] a parallel computation platform for large-scale sampling applications. The execution is based on formulating high-level workflows in a dataflow algorithm. These workflows are then analyzed for dependencies, and all independent elements will automatically be executed in parallel. Copernicus has a fully modular structure that is independent of the simulation engine used to run individual trajectories. We have initially focused on writing plugins for the Gromacs molecular simulation toolkit, but this can easily be exchanged for any other implementation. Similarly, the core Copernicus framework is designed to permit easy implementation of a wide variety of sampling algorithms, which are implemented as plugins for the dataflow execution engine. As described below, the Copernicus formalism allows straightforward specification of any sampling or statistical-mechanics approach; once this has been done, the dataflow engine takes care of scheduling, executing, and processing the simulations required for the problem. The advantage of Copernicus compared to a completely general-purpose dataflow engine is that the structure of statistical-mechanics simulations is infused into the design of the engine, so it works much better "out of the box" for such applications.

2. FORMULATING A WORKFLOW AS A DATAFLOW

The key to parallelism in Copernicus is formulating problems as dataflow networks. This is illustrated in Figure 1 for a simple example: free-energy perturbation. In this calculation, the enthalpy and entropy changes associated with an event such as the binding of a molecule to a protein are calculated using a thermodynamic cycle composed of many individual simulations. Generally, a free-energy difference cannot be computed directly since the start and end conformations sample different

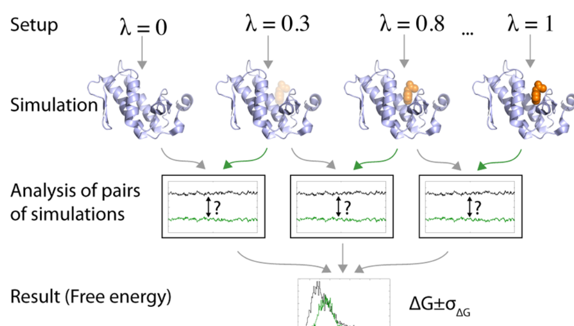


Figure 1. Free-energy calculation is an example of a sampling-based workflow. It is calculated by separating the binding into several stages (typically 10−20) with different coupling parameters λ, and running independent simulations. The work in each step is obtained by analyzing two (or more) simulations, and all parts contribute to the total free energy. Arrows denote the flow of data, and, thus, dependencies, which is equivalent to a workflow diagram.

parts of phase space. This problem is handled by artificially separating the change into many stages:[19,20] each of these requires an individual molecular dynamics simulation, so the difference between adjacent points is small enough for them to sample overlapping states. When simulations are finished, post-processing of the combined output yields the free energy. Clearly, the individual simulations can be run in parallel. This is apparent from the diagram of Figure 1, because the links between the nodes denote the *flow of data* and explicitly show dependencies. Therefore, the workflow is a *dataflow diagram* and thus can be executed by an algorithm that runs each individual component when its data dependencies are met.

In addition to parallelization, formalizing the free-energy perturbation process as a dataflow network permits easy reuse and automated repetition. For a single case, it is relatively straightforward to manually run 10−20 simulations and perform post-processing, but a screening study examining 1000 small molecules suddenly becomes much more onerous. The dataflow network formalism also enables more sophisticated approaches such as altering the simulation setup to achieve more efficient overlap with a different distribution of stages based on short initial runs (known as adaptive lambda spacing).

The basis of the execution mechanism in Copernicus is as follows: the highest-level description of a job is a *dataflow program*,[21−26] which the platform distributes among connected worker processes. If the individual work items are themselves parallel, this creates a hierarchy of parallelism where thousands of workers each parallelizing over thousands of cores can make efficient use of very large resources (Figure 2).

In addition to presenting an opportunity for parallel execution, the workflow description also allows for algorithms where the simulations to be executed are dependent on the analyzed results of completed simulations in an automated manner, enabling fully automated *adaptive sampling*. This class of algorithms attempts to determine whether regions of phase space have been oversampled or undersampled and uses this information to weight the priority of subsequent simulations. Such an approach can reduce the uncertainty associated with kinetic quantities by over 3 orders of magnitude using the same amount of compute time.[27] The sampling algorithm is implemented as a loop where simulations are started; results are processed and new iterations are started until some stop condition is reached, such as the error estimate of the end result falling below a threshold. This is where the real power of the dataflow program formulation becomes evident: the executable elements are no longer limited to the mere simulations, but it can also include complex analysis programs that analyze the output of previous simulations, or combine molecular simulations with docking or Monte Carlo approaches to sample different parts of phase space. The only limitation is that the algorithms must be formulated in terms of elements with well-defined input and output data.

2.1. The Copernicus Dataflow Network. To describe the Copernicus dataflow layout in detail, we reuse the free-energy example. Figure 3 shows an annotated graph of the components used for this application. The entire dataflow program itself is run inside a Copernicus *project*: this nomenclature was chosen to avoid confusion with the separate programs within the execution elements. This project consists of a network of active *function instances* and their *connections*. In this particular example, there are three types of function instances inside the project, corresponding to the preprocessing used to setup the

DOI: 10.1021/acs.jctc.5b00234
J. Chem. Theory Comput. 2015, 11, 2600−2608

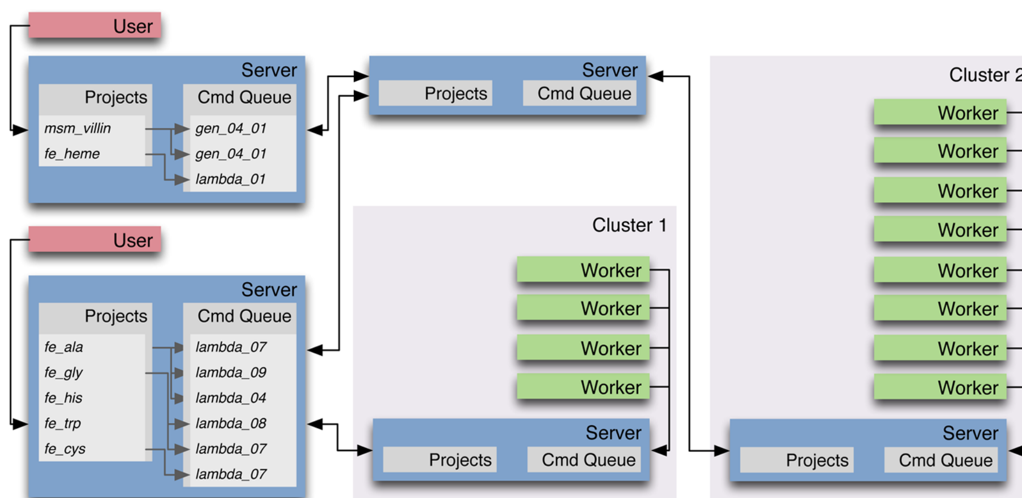


Figure 2. An example Copernicus network layout in computational chemistry. Two users each have a handful of different projects that they interact with on their workstation, and servers that schedule the project elements and worker execution. Here, Cluster 2 might be a larger resource where both users have allocations, and, because of a firewall, there is an additional Copernicus server acting as a gateway to propagate jobs and results.

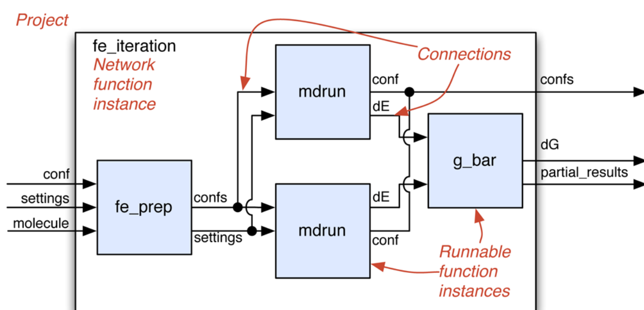


Figure 3. An annotated simplified Copernicus project. The project consists of runnable function instances corresponding to executable programs, and connections that describe the flow of data and dependencies. This assembly is called a network, and a complete such unit can itself be used as a network function instance with well-defined input and output as part of a larger project.

simulations (`fe_prep`), the program used to execute simulations (`mdrun`), and the final analysis to compute the free energy (`g_bar`). The arrows between them correspond to data such as conformations of molecules, settings, and energy differences. However, looking at the entire outer rectangle corresponding to the project, this network also can be seen as a function instance with input in the form of conformations, settings and a molecule description, and the free energy (`dG`) as well as final conformations as output. Copernicus fully embraces this hierarchical structure; any function instance either consists of an internal network, or it is directly *runnable*. Runnable function instances are the units of direct execution: once all their required inputs are present and there is hardware available, the instance will be run by executing the *function*. For the present example, this means the complete free energy project can be reused as a function instance to compute the free energy for a molecule as part of a more complex project screening different molecules, testing multiple protein conformations, or iteratively optimizing parameters.

The data in the dataflow program flows from output sockets to input sockets, both of which are strongly typed: the type of an input socket on a function instance must match the type of the output socket to which it is connected. Runnable functions are automatically executed every time an input changes, but only once all nonoptional inputs are available. There are four base datatypes that are tracked by Copernicus: integers (`int`), real numbers (`float`), booleans (`bool`), and strings and files (`string and file`). There are also two compound types: *array*, for items indexed by number, and *record*, for named lists of items. These types are intentionally kept simple to make it easy to reuse components: If data must be combined, we can easily create a function instance to achieve that, or when dealing with complex data such as a trajectory or molecule description in computational chemistry, we can simply send the entire file and let the runnable function instances deal with the contents of the file.

In order to enable dynamic execution (such as iterations and conditionals), two types of dynamism are supported in the dataflow network. First, a function can be both runnable and have a network; this type of function is described in section 2.4. The second type of dynamism is associated with arrays: *instance arrays* will instantiate as many copies of a function as there are inputs in its array of function inputs; the output is an array of function outputs (see Figure 4). This is a straightforward way to implement iterations and conditional execution. In the simplified annotation above, we would, in reality, have the `fe_prep` function instance output a (variable) number of simulation setups, and then use an instance array of the simulation execution `mdrun` to handle all those simulations. It is worth pointing out that Copernicus does *not* use explicit iteration constructs such as for-loops; at first sight, this might seem counterintuitive, but it can be extremely difficult or impossible to automatically analyze dependencies inside such loops. The fundamental idea of the dataflow networks is that we do not think in terms of single-threaded execution of functions, but instead focus on the flow of data and dependencies. This creates an obvious analogy between instance arrays and explicit parallelism, but Copernicus is not limited to only parallelize over a single instance array at a time—the network will always execute as many runnable function instances as possible in parallel, based on the full network.

2.2. Execution by Function Instances. When all required inputs are present, a runnable function instance is executed.

DOI: 10.1021/acs.jctc.5b00234
J. Chem. Theory Comput. 2015, 11, 2600−2608

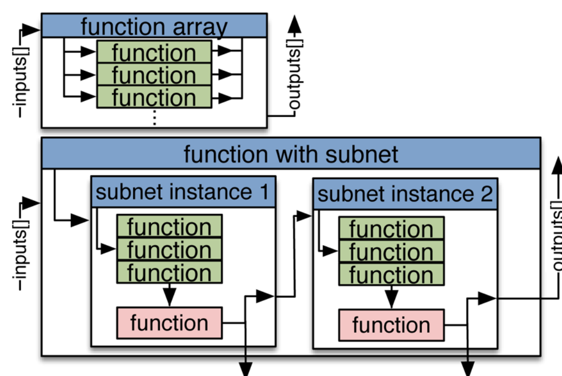


Figure 4. Example function instances. By using a function array, the actual work will be performed separately on each element in the variable size input, and produce an output array of the same size (top). A function instance will appear as a black box externally, but internally it can contain subnets, which, in turn, can be instantiated into arbitrary new copies on the fly (bottom).

This consists of collecting the input data, running the function, and propagating the output (see Figure 5). With powerful

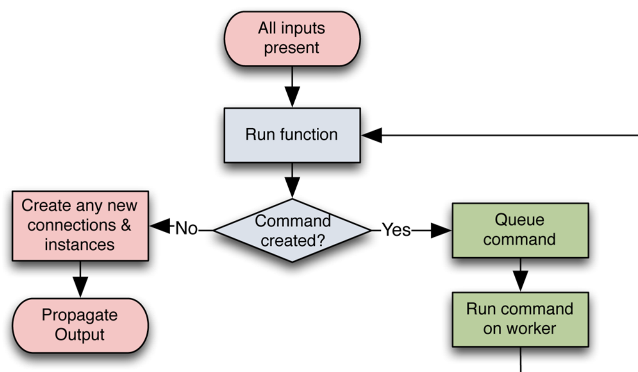


Figure 5. Function instance execution. Generally, function execution will correspond to large external computations such as simulations or analyses queued for running on worker hardware. When executions return output to the function, it will either propagate data from finished runs in the network or automatically reissue the command if a worker failed.

server hardware, short function instances can be run instantly on the server, but, generally, they will require significant computing resources, and in this case the execution step is more complex since it also requires suitable hardware, with function instances subject to prioritization to optimize dependencies in the entire network.

If execution on hardware separate from the server is appropriate, a function instance can issue a *command* to the Copernicus command queue, requesting a program to be run on a *worker*. The output is then passed back to the function instance, with the original inputs, allowing the function instance to check whether the command has completed successfully. The output data is propagated and updated atomically; updates to other runnable functions are done simultaneously in a single update per function instance.

2.3. Workers from Supercomputers to Desktops. Copernicus targets everything from high-end parallel jobs running through queues at supercomputers to small clusters, desktops, or throughput computing in the cloud. No matter what the hardware is, the computers running the commands are called *workers*. Workers announce their presence and availability to run commands by connecting to a server and presenting their *platform* and a list of available *executables*. This can be a client program running on a desktop, or a batch script submitted to a supercomputer queue—in this case, the worker will announce its availability when the supercomputer scheduler starts to execute it, and it ceases to be available when the batch job finishes. A worker's platform holds the capabilities and resources of the worker, such as the number of cores and whether there is a high-performance interconnect available for parallel MPI execution. The executable list describes the programs that can be executed on a worker (i.e., which are installed) given the platform.

On the basis of the platform and executable list, the server that the worker connects to either matches its available jobs and returns a list of jobs to run, or delegates the request to another server, e.g., when acting as a gateway. This allows for capability matching where large jobs are run on supercomputers or clusters available in the network while throughput-style jobs execute on desktops, clouds, or distributed computing resources. Copernicus is also capable of using, e.g., a 10 000-core worker allocation to execute 100 separate function instances, each needing 100 cores. This effectively makes the network of connected servers a global pool of commands that can share idle high-performance computing resources in a peer-to-peer fashion (Figure 2).

Worker progress is checked by monitoring regular heartbeat signals from workers. If no heartbeat has been received for a specified time, the worker is presumed dead, and the commands running on the worker are reissued. When executing in cluster environments, Copernicus makes use of checkpointing in programs that support it (such as Gromacs), which means the reissued command starts from the checkpointed state of the original command. This is particularly useful for preemptable cluster queues or spot-market cloud computing where large amounts of resources can be used freely or at very low cost on the condition that execution can be terminated at any time.

2.4. The Dataflow Network Is Dynamic. Representing networks as function instances provides encapsulation and abstraction, which facilitates reusing components similar to high-level programming languages. For this type of *subnet*, we define a separate set of standard input/output nodes that apply to the entire subnet, in contrast to the internal subnet inputs and outputs that are only accessible inside this particular subnet. Each subnet will also have a runnable function that is executed when all dependencies are available on the inputs. In many cases, this will be a trivial function that merely executes the first directly runnable functions inside the subnet, but we allow the subnet to be dynamically controlled by the subnet function. In other words, the subnet itself can establish new instances and connections inside its own dataflow network (see Figure 4).

This provides for a fully *dynamic dataflow network*. Based on intermediate results, a project can, e.g., introduce an extra clustering step or alter the algorithm. All these decisions are made based on availability, flow, and contents of data, and any time input data changes anywhere in the network it is subject to partial re-evaluation.

2.5. Transparent Client Access to Data. An advantage of using explicit dataflow descriptions is that program execution becomes transparent to the user; any value can be examined or set at any time. With Copernicus, this is possible through the

DOI: 10.1021/acs.jctc.5b00234
J. Chem. Theory Comput. 2015, 11, 2600−2608

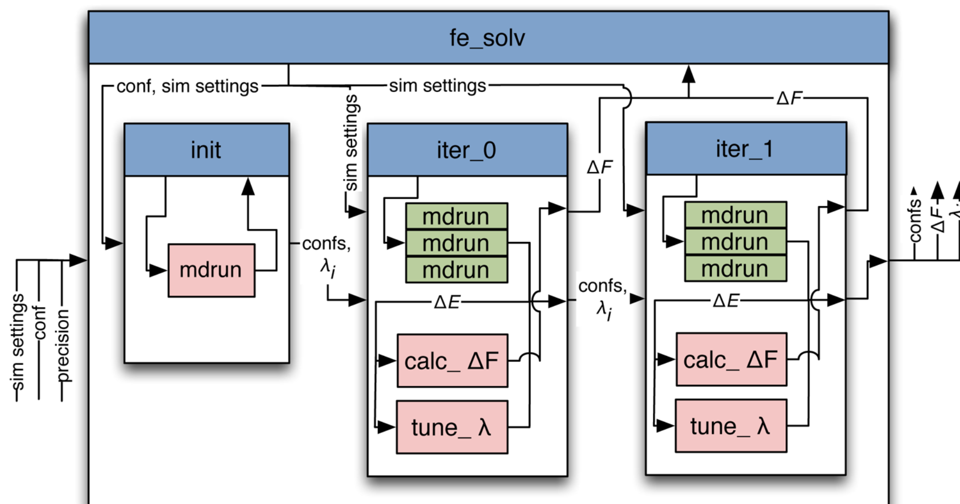


Figure 6. Free energy of solvation workflow after initialization, equilibration (iter_0) and the first iteration of production simulations. New iteration subnets to improve the precision are instantiated automatically. In most cases, only the final output is of interest, but it is trivial to manually query specific data from functions inside any subnet. Even this simple project uses function arrays, subnets, and dynamic instantiation for flow control.

command-line interface `cpcc` (copernicus-client) that is typically run on a client, such as a laptop, connecting to a server. The easiest way to illustrate this is to use an example:

```
> cpcc get fe.iter_lj_1.out.dg
```

Here, we use the top-level function `fe`, in which we access the instance called `iter_lj_1`, which is the first iteration of the Lennard-Jones decoupling. For that function instance, we fetch the output variable called `dG`. In this particular case, the value is an energy difference in a free-energy simulation. If the output data were not yet available, we would get a message saying so. Similarly, any nonconnected inputs can be set or reset, and we can even create new function instances through the client. When we alter input data, this will automatically cause re-evaluation of function instances that are dependent on this data—a good analogy is the Makefile, which is used for compiling large software projects. This allows the user to not only monitor a running project for intermediate results, but it allows real-time interaction with the algorithm and state of the dataflow network.

3. APPLICATIONS

Copernicus is meant for expressing high-level algorithms, which run on top of collections of individual parallel processes. In our work, this is typically focused of sampling algorithms analyzing collections of energies or conformations obtained from large numbers of molecular dynamics simulations. Many such projects are included in the Copernicus distribution. This section describes some of these as examples of how adaptive sampling algorithms can be implemented to provide automated optimal sampling.

3.1. Program Modules: Gromacs Molecular Simulation. Gromacs is one of the common high-performance molecular simulation packages.[7] In Copernicus, it is used as a molecular simulation engine for free-energy calculations, Markov state models, and string method calculations. The implementation consists of a fault-tolerant function for running simulations (relying on mdrun, the Gromacs program to run simulations), and a mapping of some of the Gromacs pre-processing and post-processing tools. If the user would prefer to use a different package, similar plugins can be written with very little work.

The Copernicus function controlling mdrun uses the mechanism shown in Figure 5, in conjunction with mdrun's capability to save and recover from checkpoints. Whenever a worker returns output from an mdrun command, it has either finished, saved a checkpoint file, or made no progress. In the latter case, the original command is reissued. If there is a checkpoint file, a new command is emitted to recover from the checkpoint. If the simulation has finished, all the output files are concatenated and sent as workflow outputs. This means the rest of the network can always count on complete and correct output data - any continuation or hardware failure will be handled by the program module. To support efficient parallelism, there is a run tuning function that checks which number of cores to use for a given set of simulations. This is used by the server's worker−workload matching to distribute the available cores over the available simulation commands in the most efficient way.

3.2. Free-Energy Perturbation Calculations. To describe a typical project, we expand the initial free-energy example. These are some of the most widely used throughput-focused molecular simulation methods, and aim to calculate a free-energy difference (ΔF) between two systems with Hamiltonians $\mathcal{H}_0$ and $\mathcal{H}_1$. The free-energy difference between the two states is calculated using an average exponential Hamiltonian difference in the ensembles of both end states.[20] Common applications include static quantities such as ligand−protein binding affinity in drug design,[28] solubility,[29] and phase stability.[30] In practice, this cannot be done in a single step without significant sampling error, so instead we introduce a Hamiltonian coupled to a parameter λ:

$$\mathcal{H}_\lambda = \lambda \mathcal{H}_0 + (1 - \lambda)\mathcal{H}_1 \quad (1)$$

The free-energy difference then becomes a sum of smaller free-energy difference calculations between a set of chosen points $\lambda_1, ..., \lambda_N$ with

DOI: 10.1021/acs.jctc.5b00234
J. Chem. Theory Comput. 2015, 11, 2600−2608

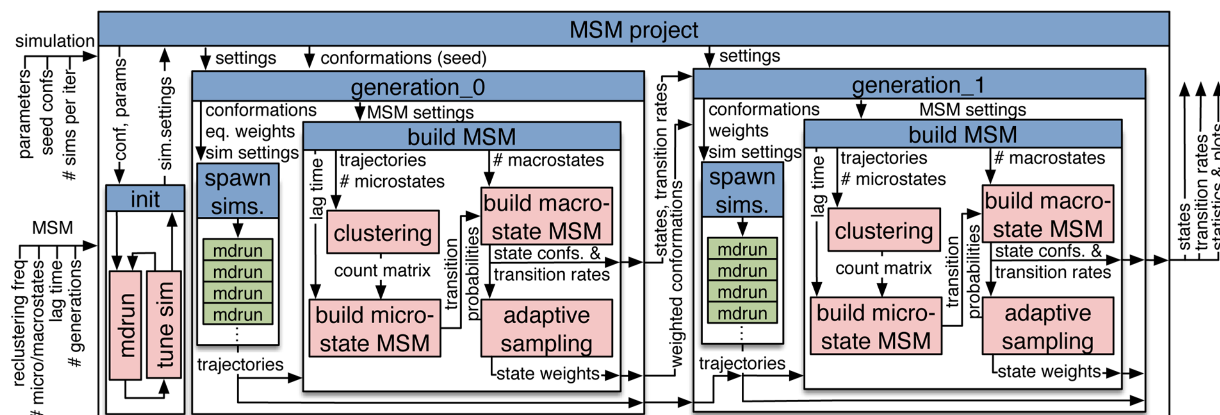


Figure 7. Markov state modeling (MSM) adaptive sampling project. Starting from seed conformations, a first generation subnet is instantiated which spawns a large number of simulations, and trajectories are provided to a subnet using MSMbuilder for the clustering and modeling of states. The output from the first generation is a set of weighted states, from which the next generation is instantiated. Only two generations are show; a typical MSM project uses many more, but the final output is always states and transition rates.

$$\Delta F = \sum_{i=1}^{N-1} \Delta F(\lambda_i, \lambda_{i+1}) \tag{2}$$

as illustrated in Figure 1. Using more points will lead to better phase space overlap, but also more simulations. The location of the set of points in λ provides an opportunity for adaptive sampling; the optimal location of these points is when the per-sample standard deviation[20] of all individual free-energy differences is equal. This distribution can be iterated toward, using very short individual free-energy calculations.

Copernicus has two different high-level free-energy calculation functions. The first handles the free energy of solvation calculations where the interactions of one or more molecules' with the rest of the system are gradually turned off as λ goes from 0 to 1. The second is one for the free energy of binding calculations, where the interactions are turned off while restraining the molecule to locations relative to other molecules, such as tethering a ligand to the protein for binding affinity calculations. This free energy is then compared to turning off the interactions in pure solvent, yielding the excess free energy of binding, or binding affinity.

The structure of the free energy of solvation function is outlined in Figure 6. Its inputs include the files necessary to run a simulation, the name of the molecule(s) to decouple from the rest of the system, and the precision threshold in the free-energy output. This precision serves as the stop condition for the calculation.

The solvation free-energy function first routes most of the inputs to an initialization function, where a short simulation is scheduled that gradually changes λ from 0 to 1 (separately for electrostatic and Lennard-Jones interactions), outputting configurations along the way. These configurations are routed to the first iteration of the lower-level function that actually performs the free-energy simulations and calculation. This function outputs its final configurations, the calculated free-energy difference, and the inputs for the next iteration, including new values for the set λ_i. The output free-energy differences are collected in the function and their weighted average is the output of this function. Copernicus was used for efficiently calculating solvation free energies of 50 molecules using three different force fields and four different water models.[31]

The free energy of binding function takes as inputs a bound configuration of ligand and target molecule (such as a protein), a set of restraints to keep the ligand close to the target as it is decoupled, in addition to the parameters needed for the free energy of solvation function. The function then consists of three different lower-level free-energy calculations: one to calculate a free-energy difference between the ligand bound to the target and the ligand decoupled from the system (ΔF_{target}), one to calculate the free energy of solvation of the ligand (ΔF_{solv}), and one to calculate the free-energy contribution of the restraints (ΔF_{restr}). The free energy of binding is then[28]

$$\Delta F_{\text{binding}} = \Delta F_{\text{target}} - \Delta F_{\text{solv}} - \Delta F_{\text{restr}} \tag{3}$$

where the structure of the individual free-energy contribution functions is similar to that shown in Figure 6. This also shows the power of reusing components in higher-level building blocks.

3.3. Markov State Modeling.

Perhaps no modern sampling technique is as closely associated with distributed computing approaches as Markov state modeling[16,32−34] (MSM). MSM attempts to give a picture of the dynamics of a system by combining clustering with a kinetic description of transitions.[35] The technique consists of several stages: first, the set of configurations of a large number of simulation trajectories is clustered. Second, an intercluster transition matrix is constructed, from which a rate matrix is calculated, assuming a characteristic "lag time". The highest-valued eigenvalues and eigenvectors of this rate matrix describe the slowest and largest-scale motions of the system in cluster space. This can then be used to further coarse-grain the clusters into a kinetic picture of the important transitions of the system.

MSMs work exceptionally well with adaptive sampling,[18,27,32] where new simulations are started from existing clusters, either by identifying undersampled clusters, or simply by evenly sampling clusters. Because it relies on ensembles of trajectories and stochastic processing, the progress of all individual runs is not necessary for the progress of adaptive sampling as a whole. This, combined with the high level of parallelism inherent in many hundreds of trajectories, makes MSM a very attractive sampling method for distributed computing.

The Copernicus MSM implementation is outlined in Figure 7 and has also been described in our prior work.[18] The MSM function inputs include a number of starting configurations,

DOI: 10.1021/acs.jctc.5b00234
J. Chem. Theory Comput. 2015, 11, 2600−2608

simulation settings, the number of simulations to initially start, the amount of trajectory data required for an MSM iteration, the lag time, and the number of microstates to generate. The MSM function will create a user-specified number of simulation runs, and collect enough data to create an MSM. After this, new runs will be spawned based on even sampling of the generated microstates. This will lead to subsequent generations of MSMs until a user-specified number of them has been reached.

With the dataflow network formulation, all the steps necessary to select conformations, execute simulations, cluster data, and build MSMs are fully automated. For a novice user, this means they can use very advanced sampling algorithms largely as black boxes (for better or worse). This is not necessarily a free lunch; an expert user might do better by manually analyzing clustered conformations and alter the settings. As described above, this is eminently possible—the user can both monitor convergence of the transition state matrix and inspect individual cluster conformations from each iteration as the project is running, or alter the settings on-the-fly to force a new clustering, which will result in new runs being generated.

An example that can be run in 24 h on a handful of computers is the folding of Fs-peptide, a 21-amino-acid peptide that folds into an α-helix on the 200 ns time scale.[36] The initial seeds were four extended conformations from a 500 ps simulation at 700 K. Each iteration spawns 10 new simulations and initiates clustering for every 10 ns of new trajectories; 1500 microstates were clustered using hybrid clustering and a lag time of 100 ps. These microstates were further clustered using PCCA+[37] to identify 10 macrostates. After 4 generations of MSM, a folded state was identified, and with 7 generations there was sufficient sampling to determine convergence and identify the folded state with no *a priori* knowledge (see Figure 8).

Simulations were run in Gromacs 4.6[7] using the Amber03 force field[38] and a rhombic dodecahedron box with 7000 TIP3P[39] waters. The MSM building steps utilize MSMBuilder,[40] but other packages, such as EMMA,[41] could be used to obtain similar results. The simulations were run on a total of 4 machines: two 32-core AMD Opteron machines equipped with dual Geforce GTX Titan GPUs, one 16-core Intel Xeon equipped with dual Geforce GTX Titan GPUs, and one 12-core Intel Xeon equipped with dual Tesla K20 GPUs.

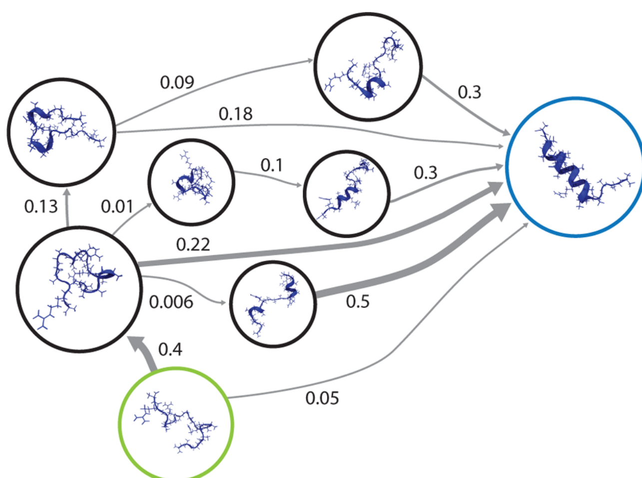


Figure 8. Macrostate MSM of Fs-peptide. Starting from extended conformation (green), each MSM generation captures new states and spawns simulations. After four generations, the folded state (blue) is sampled, and, with seven generations, it is identified as the lowest free energy without prior knowledge. Arrows indicate transition pathways, with thickness reflecting net flux probability (indicated); self-flux and low-probability edges are omitted.

3.4. String Method Using Swarms. In many cases, there are experimental structures available for different states of molecules, but computational methods are needed to understand the transition. It generally requires an extremely high-dimensional "reaction-coordinate" to describe the transitions between two states A and B. One powerful approach is the string method with swarms of trajectories, which finds transition paths between stable states by restricting sampling to refine a quasi-one-dimensional path embedded in a high-dimensional collective variable space.[42,43]

Briefly, the technique works by first interpolating a path between states A and B in the chosen collective variable space (for instance, some, or all, torsions) to get an initial set of points. Restrained minimization and relaxation simulations are used to produce several conformations for each point in collective variable space. A large number of short simulations from these conformations are then started without restraints. The average drift in collective variables space is computed for each "swarm", and used to move the path toward lower free energy, which will gradually converge toward the most probable transition pathway. To ensure that the sampling is optimally distributed along the path, a reparameterization scheme[42] is used to maintain equal spacing of points in collective variables space. Figure 9 illustrates the state of the swarms module after two iterations; for real projects, there can be hundreds.

Copernicus can easily execute the classical alanine dipeptide string example,[43] and the fully automated setup makes it trivial to, e.g., test different force fields. Using the ϕ and ψ dihedral angles of the peptide as collective variables, the path between the states C_{eq} = (−82,73) and C_{ax} = (70,−69) was optimized for the CHARMM27 force field with and without the CMAP corrections,[44] using 80 iterations with 18 string points. Relaxation simulations of 25 ps at 300 K with dihedral restraints (4000 kJ mol^{-1} rad^{-2}) were used to generate 20 structures from each trajectory, all of which were run for 30 fs without restraints. As seen in Figure 10, the application of CMAP correction leads to a slightly different transition path that converges to a local saddle point; the automated setup makes it possible to test a large number of initial seed paths. For large solvated protein complexes, the Copernicus swarms module can simultaneously execute over 10 000 short simulations if given a sufficient pool of workers.

4. DISCUSSION

Just as the number of transistors has doubled roughly every 18 months, according to the observation in Moore's law, the number of cores in large computers appear to have also followed this curve for the past decade. There is no sign of this trend reversing; in contrast, increased funding means supercomputers are currently outpacing it. With a rough extrapolation, this would imply the first machines with a billion processing elements a decade from now, and, at that point, even a small workgroup cluster will require millionfold parallelism. Hardly any current programs will exhibit strong scaling on such resources, but by combining parallelization of the core programs over tens of thousands of processors with higher-level sampling algorithms, it will become possible to harness this power for important applications. Several high-level

DOI: 10.1021/acs.jctc.5b00234
J. Chem. Theory Comput. 2015, 11, 2600−2608

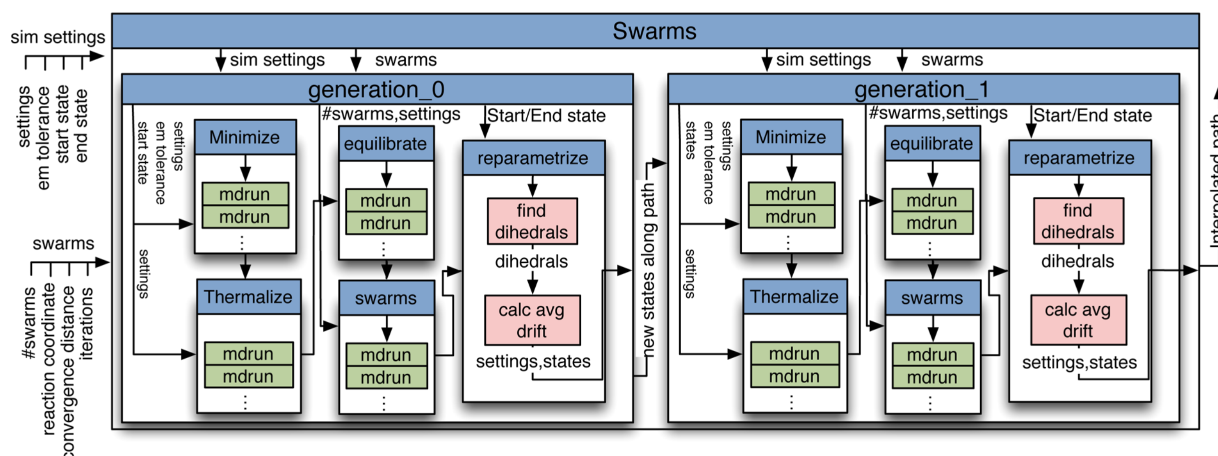


Figure 9. Swarms project after two generations. The module requires two states and a reaction coordinate, typically all or some of the torsions in the system. The first generation performs minimization, thermalization, and equilibration, followed by the large number of short swarm simulations. The final step updates the path and reparameterizes the spacing of points, after which the subnet function instantiates a similar function for the next generation.

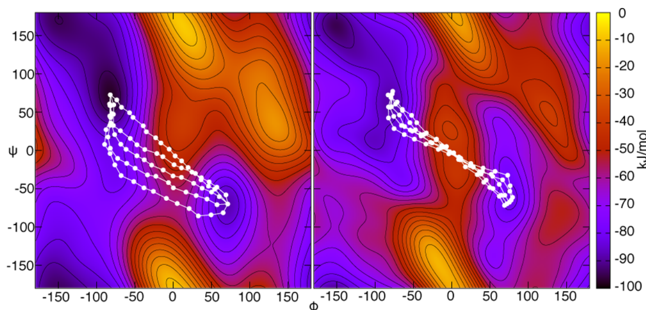


Figure 10. Transition paths in dialanine. Compared to the default CHARMM27 force field (left), the inclusion of the CMAP correction term leads to a slightly different transition path that converges in a local saddlepoint (right).

sampling algorithms—in particular, in molecular simulations—can be expressed in a straightforward way as dataflow networks for execution by a distributed execution platform. We have developed Copernicus as such an engine for the expression and execution of dataflow networks to solve problems in statistical mechanics and computational chemistry. Copernicus utilizes the explicit dependency structure of these networks to optimize parallel execution and also provides an interface for users to monitor and interact with the calculation while it is running, as is frequently desired in this discipline. The plug-in model allows straightforward implementation of many sampling or statistical-mechanics-driven algorithms and the use of many different simulation programs to compute trajectories, thus providing a powerful and flexible framework for parallel execution of complex and compute-intensive applications in computational chemistry. Copernicus is freely available from http://copernicus.gromacs.org.

AUTHOR INFORMATION

Corresponding Author

*E-mail: erik.lindahl@scilifelab.se.

Author Contributions

§These authors contributed equally to this work.

Notes

The authors declare no competing financial interest.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (No. 209825), the Swedish Research Council (No. 2013-5901), and the Swedish Foundation for International Cooperation in Research and Higher Education (STINT). Computational resources were provided by the Swedish National Infrastructure for Computing (No. 2014/11-33).

REFERENCES

(1) Brooks, B. R.; Bruccoleri, R. E.; Olafson, B. D.; States, D. J.; Swaminathan, S.; Karplus, M. *J. Comput. Chem.* **1983**, *4*, 187−217.

(2) Pearlman, D. A.; Case, D. A.; Caldwell, J. W.; Ross, W. S.; Cheatham, T. E., III; DeBolt, S.; Ferguson, D.; Seibel, G.; Kollman, P. *Comput. Phys. Commun.* **1995**, *91*, 1−41.

(3) Lindahl, E.; Hess, B.; Van Der Spoel, D. *J. Mol. Model.* **2001**, *7*, 306−317.

(4) Case, D. A.; Cheatham, T. E., III; Darden, T.; Gohlke, H.; Luo, R.; Merz, K. M., Jr.; Onufriev, A.; Simmerling, C.; Wang, B.; Woods, R. J. *J. Comput. Chem.* **2005**, *26*, 1668−1688.

(5) Phillips, J. C.; Braun, R.; Wang, W.; Gumbart, J.; Tajkhorshid, E.; Villa, E.; Chipot, C.; Skeel, R. D.; Kale, L.; Schulten, K. *J. Comput. Chem.* **2005**, *26*, 1781−1802.

(6) Brooks, B. R.; Brooks, C. L., III; Mackerell, A. D., Jr.; Nilsson, L.; Petrella, R. J.; Roux, B.; Won, Y.; Archontis, G.; Bartels, C.; Boresch, S.; Caflisch, A.; Caves, L.; Cui, Q.; Dinner, A. R.; Feig, M.; Fischer, S.; Gao, J.; Hodoscek, M.; Im, W.; Kuczera, K.; Lazaridis, T.; Ma, J.; Ovchinnikov, V.; Paci, E.; Pastor, R. W.; Post, C. B.; Pu, J. Z.; Schaefer, M.; Tidor, B.; Venable, R. M.; Woodcock, H. L.; Wu, X.; Yang, W.; York, D. M.; Karplus, M. *J. Comput. Chem.* **2009**, *30*, 1545−1614.

(7) Pronk, S.; Páll, S.; Schulz, R.; Larsson, P.; Bjelkmar, P.; Apostolov, R.; Shirts, M. R.; Smith, J. C.; Kasson, P. M.; van der Spoel, D.; Hess, B.; Lindahl, E. *Bioinformatics* **2013**, *29*, 845−854.

(8) Shaw, D. E.; Grossman, J. P.; Bank, J. A.; Batson, B.; Butts, J. A.; Chao, J. C.; Deneroff, M. M.; Dror, R. O.; Even, A.; Fenton, C. H.; Forte, A.; Gagliardo, J.; Gill, G.; Greskamp, B.; Ho, C. R.; Ierardi, D. J.; Iserovich, L.; Kuskin, J. S.; Larson, R. H.; Layman, T.; Lee, L.-S.; Lerer, A. K.; Li, C.; Killebrew, D.; Mackenzie, K. M.; Mok, S. Y.-H.; Moraes, M. A.; Mueller, R.; Nociolo, L. J.; Peticolas, J. L.; Quan, T.; Ramot, D.; Salmon, J. K.; Scarpazza, D. P.; Ben Schafer, U.; Siddique, N.; Snyder, C. W.; Spengler, J.; Tang, P. T. P.; Theobald, M.; Toma, H.; Towles, B.; Vitale, B.; Wang, S. C.; Young, C. Anton 2: Raising the Bar for Performance and Programmability in a Special-purpose Molecular Dynamics Supercomputer. In *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and*

DOI: 10.1021/acs.jctc.5b00234
J. Chem. Theory Comput. 2015, 11, 2600−2608

Analysis, New Orleans, LA, USA, Nov. 15−21, 2014; IEEE Press: Piscataway, NJ, USA, 2014; pp 41−53.
(9) Shaw, D. E.; Deneroff, M. M.; Dror, R. O.; Kuskin, J. S.; Larson, R. H.; Salmon, J. K.; Young, C.; Batson, B.; Bowers, K. J.; Chao, J. C.; Eastwood, M. P.; Gagliardo, J.; Grossman, J.; Ho, C. R.; Ierardi, D. J.; Kolossváry, I.; Klepeis, J. L.; Layman, T.; McLeavey, C.; Moraes, M. A.; Mueller, R.; Priest, E. C.; Shan, Y.; Spengler, J.; Theobald, M.; Towles, B.; Wang, S. C. *Commun. ACM* **2008**, *51*, 91−97.
(10) Lifson, S.; Warshel, A. *J. Chem. Phys.* **1968**, *49*, 5116−5129.
(11) Fuller, S. H. *Computer* **2011**, *44*, 31−38.
(12) Sugita, Y.; Okamoto, Y. *Chem. Phys. Lett.* **1999**, *314*, 141−151.
(13) Elmer, S. P.; Pande, V. S. *J. Chem. Phys.* **2004**, *121*, 12760−12771.
(14) Faradjian, A. K.; Elber, R. *J. Chem. Phys.* **2004**, *120*, 10880−10889.
(15) Noé, F.; Horenko, I.; Schütte, C.; Smith, J. C. *J. Chem. Phys.* **2007**, *126*, 155102.
(16) Pan, A. C.; Roux, B. *J. Chem. Phys.* **2008**, *129*, 064107.
(17) Bonomi, M.; Branduardi, D.; Bussi, G.; Camilloni, C.; Provasi, D.; Raiteri, P.; Donadio, D.; Marinelli, F.; Pietrucci, F.; Broglia, R. A.; Parrinello, M. *Comput. Phys. Commun.* **2009**, *180*, 1961−1972.
(18) Pronk, S.; Bowman, G.; Hess, B.; Larsson, P.; Haque, I.; Pande, V.; Pouya, I.; Beauchamp, K.; Kasson, P.; Lindahl, E. Copernicus: A new paradigm for parallel adaptive molecular dynamics. In *Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, Seattle, WA, Nov. 12, 2011; ACM: New York, 2011; p 60.
(19) Frenkel, D.; Smit, B. *Understanding Molecular Simulation*, 2nd Edition; Academic Press: London, 2002.
(20) Bennett, C. H. *J. Comput. Phys.* **1976**, *22*, 245−268.
(21) Chambers, C.; Raniwala, A.; Perry, F.; Adams, S.; Henry, R. R.; Bradshaw, R.; Weizenbaum, N. FlumeJava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*; ACM: New York, 2010; pp 363−375.
(22) Akidau, T.; Balikov, A.; Bekiroğlu, K.; Chernyak, S.; Haberman, J.; Lax, R.; McVeety, S.; Mills, D.; Nordstrom, P.; Whittle, S. MillWheel: Fault-tolerant stream processing at internet scale. In *Proceedings of the VLDB Endowment*, Riva del Garda, Trento, Italy, Aug. 28, 2013; VLDB Endowment: Secaucus, NJ, USA, 2013; pp 1033−1044.
(23) Murray, D. G.; McSherry, F.; Isaacs, R.; Isard, M.; Barham, P.; Abadi, M. Naiad: A timely dataflow system. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, Farmington, PA, USA, Nov. 3, 2013; ACM: New York, 2013; pp 439−455.
(24) Zhao, Y.; Hategan, M.; Clifford, B.; Foster, I.; Von Laszewski, G.; Nefedova, V.; Raicu, I.; Stef-Praun, T.; Wilde, M. Swift: Fast, reliable, loosely coupled parallel computation. In *Proceedings of the 2007 IEEE Congress on Services*, Salt Lake City, UT, USA, July 9−13, 2007; IEEE: Piscataway, NJ, USA, 2007; pp 199−206.
(25) Raicu, I.; Zhao, Y.; Dumitrescu, C.; Foster, I.; Wilde, M. Falkon: A Fast and Light-weight tasK executiON framework. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, Reno, NV, USA, Nov. 16, 2007; ACM: New York, 2007; p 43.
(26) Zaharia, M.; Das, T.; Li, H.; Shenker, S.; Stoica, I. Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, Boston, MA, USA, June 12, 2012; USENIX Association: Berkeley, CA, USA, 2012; p 10.
(27) Hinrichs, N. S.; Pande, V. S. *J. Chem. Phys.* **2007**, *126*, 244101.
(28) Mobley, D. L.; Graves, A. P.; Chodera, J. D.; McReynolds, A. C.; Shoichet, B. K.; Dill, K. A. *J. Mol. Biol.* **2007**, *371*, 1118−1134.
(29) Paliwal, H.; Shirts, M. R. *J. Chem. Theory Comput.* **2011**, *7*, 4115−4134.
(30) Polson, J. M.; Trizac, E.; Pronk, S.; Frenkel, D. *J. Chem. Phys.* **2000**, *112*, 5339−5342.
(31) Lundborg, M.; Lindahl, E. *J. Phys. Chem. B* **2015**, *119*, 810−823.
(32) Pande, V. S.; Beauchamp, K.; Bowman, G. R. *Methods* **2010**, *52*, 99−105.
(33) Swope, W. C.; Pitera, J. W.; Suits, F. *J. Phys. Chem. B* **2004**, *108*, 6571−6581.
(34) Singhal, N.; Pande, V. S. *J. Chem. Phys.* **2005**, *123*, 204909.
(35) Prinz, J.-H.; Wu, H.; Sarich, M.; Keller, B.; Senne, M.; Held, M.; Chodera, J. D.; Schütte, C.; Noé, F. *J. Chem. Phys.* **2011**, *134*, 174105.
(36) Gnanakaran, S.; Nymeyer, H.; Portman, J.; Sanbonmatsu, K. Y.; Garca, A. E. *Curr. Opin. Struct. Biol.* **2003**, *13*, 168−174.
(37) Deuflhard, P.; Weber, M. *Linear Algebra Appl.* **2005**, *398*, 161−184.
(38) Duan, Y.; Wu, C.; Chowdhury, S.; Lee, M. C.; Xiong, G.; Zhang, W.; Yang, R.; Cieplak, P.; Luo, R.; Lee, T.; Caldwell, J.; Wang, J.; Kollman, P. *J. Comput. Chem.* **2003**, *24*, 1999−2012.
(39) Jorgensen, W. L.; Chandrasekhar, J.; Madura, J. D.; Impey, R. W.; Klein, M. L. *J. Chem. Phys.* **1983**, *79*, 926−935.
(40) Beauchamp, K. A.; Bowman, G. R.; Lane, T. J.; Maibaum, L.; Haque, I. S.; Pande, V. S. *J. Chem. Theory Comput.* **2011**, *7*, 3412−3419.
(41) Senne, M.; Trendelkamp-Schroer, B.; Mey, A. S.; Schütte, C.; Noé, F. *J. Chem. Theory Comput.* **2012**, *8*, 2223−2238.
(42) Pan, A. C.; Roux, B. *J. Chem. Phys.* **2008**, *129*, 064107.
(43) Maragliano, L.; Fischer, A.; Vanden-Eijnden, E.; Ciccotti, G. *J. Chem. Phys.* **2006**, *125*, 024106.
(44) MacKerell, A. D.; Feig, M.; Brooks, C. L. *J. Comput. Chem.* **2004**, *25*, 1400−1415.

DOI: 10.1021/acs.jctc.5b00234
J. Chem. Theory Comput. 2015, 11, 2600−2608