Petascale Tcl with NAMD, VMD, and Swift/T

James C. Phillips
Beckman Institute
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
jim@ks.uiuc.edu

John E. Stone
Beckman Institute
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
johns@ks.uiuc.edu

Kirby L. Vandivort
Beckman Institute
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
kvandivo@ks.uiuc.edu

Timothy G. Armstrong
Department of Computer Science
University of Chicago
Chicago, IL 60637, USA
tga@uchicago.edu

Justin M. Wozniak
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
wozniak@mcs.anl.gov

Michael Wilde
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
wilde@anl.gov

Klaus Schulten
Department of Physics
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
kschulte@ks.uiuc.edu

ABSTRACT

Tcl is the original embeddable dynamic language. Introduced in 1990, Tcl has been the foundation of the scripting interface of the popular biomolecular visualization and analysis program VMD since 1995 and was extended to the parallel molecular dynamics program NAMD in 1999. The two programs together have over 200,000 users who have enjoyed for nearly two decades the stability and flexibility provided by Tcl. VMD users can implement or extend parallel trajectory analysis and movie rendering on thousands of nodes of Blue Waters. NAMD users can implement or extend simulation protocols and multiple-copy algorithms that execute unmodified on any supercomputer without the need to recompile NAMD. We now demonstrate the integration of the Swift/T high-performance parallel scripting language to enable high-level data flow programming in NAMD and VMD. This integration is achieved without modifying or recompiling either program since the Turbine execution engine is itself based on Tcl and is dynamically loaded by the interpreter, as is the platform-specific MPI library on which it depends.

Categories and Subject Descriptors

I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Parallel*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism; D.1.3 [**Concurrent Programming**]: Parallel programming

General Terms

Algorithms, Design, Performance

HPTCDL November 16-21, 2014, New Orleans, Louisiana, USA

Keywords

Scripting, molecular simulation, molecular visualization, parallel rendering, GPU, many-core

1. INTRODUCTION

State-of-art molecular dynamics (MD) simulations provide researchers with access to atomic structural details and nanosecond timescales that are inaccessible to experimental methods alone. The combination of high-quality structure information, petascale computing, and MD simulation provides researchers with a powerful "computational microscope" that provides detailed views of the inner workings of large biomolecular complexes in the cell. The nature of MD simulation work leads researchers to develop customized tools that solve the unique computational problems arising in their scientific pursuits. The diversity of tasks involved in preparing, simulating, analyzing, and visualizing state-of-the-art molecular dynamics simulations poses a significant challenge to researchers, requiring software tools to be extensible, performant, and easy to use. Dynamic languages are ideal for the software tool development performed by many molecular scientists, because they are easy for scientists to learn; provide full access to filesystems, networking, and external programs; and can be linked directly with custom-written performance-critical subroutines written in conventional compiled languages such as C++ and GPU and heterogeneous computing languages such as CUDA [29, 17] and OpenCL [26]. The embedding of dynamic languages into parallel applications poses special challenges in terms of access to script source code, limited system call implementations on some supercomputer operating systems, and I/O scalability issues that can arise when using dynamic languages at scale, associated with runtime loading of new scripts or subroutines, dynamic link libraries, and so forth.

The molecular graphics program VMD [6] has incorporated a built-in Tcl interpreter since 1995 and added optional Python support in 2000. VMD originally used Tcl simply as an interactive text-based command and command scripting engine, but has since grown to depend much more on Tcl as a language particularly in the context of performing large-scale parallel trajectory analysis and visualization

tasks on petascale computers. VMD has recently been extended with Tcl bindings for parallel programming that enable scientists to easily adapt their existing analysis and visualization scripts and tools to throughput-oriented data-parallel execution on clusters and petascale computers [27, 32, 28].

NAMD [20] is a highly scalable parallel MD engine that is heavily used for simulations of large biomolecular complexes such as the HIV-1 capsid [42], running on state-of-the-art petascale computers with hundreds of thousands of CPU cores and tens of thousands of GPUs [22]. NAMD has incorporated a Tcl interpreter since 1999, enabling researchers to develop custom simulation protocols that involve user-defined force equations, replica exchange, protocol, and automation of other simulation tasks.

Swift/T is a dataflow language designed to enable easy composition of independent software tools and procedures into large-scale, throughput-oriented parallel workflows that can be executed on workstations, HPC clusters, and supercomputers [40]. Swift/T scales efficiently up to hundreds of thousands of CPU cores through a combination of runtime and compiler techniques [2]. NAMD and VMD have recently been successfully coupled to the Swift/T throughput computing system. Standard NAMD and VMD binaries can be launched across the nodes of a parallel computer and efficiently execute Swift/T dataflow programs with functions implemented in the embedded Tcl scripting language.

The NAMD integration with Swift/T has been used to demonstrate $n : m$ multiplexing of n replicas across a smaller arbitrary number m of NAMD processes, a capability that is complex to implement with normal NAMD scripting but that can be expressed naturally in under 100 lines of Swift/T code. Multiple-copy molecular dynamics sampling techniques that have dynamically varying breadth of work can be difficult to adapt to traditional supercomputing platforms and batch queuing systems. Adaptive Multilevel Splitting [1] and other NAMD replica exchange and multiple-copy algorithms can be enhanced to exploit unique features of cloud platforms, such as dynamic runtime resizing of parallel job node counts. When expressed as Swift/T dataflow programs, these algorithms can be executed on arbitrary node counts, with nodes joining or leaving the calculation based on both the internal demands of the sampling algorithm and the dynamic pricing of compute resources in the cloud.

Another contribution Swift/T provides for molecular modeling is the ability to incorporate a multitude of independent programs into large workflows. While NAMD and VMD can function independently and both have their own built-in mechanisms for parallel execution, Swift/T provides an alternative means for simulation protocols, such as molecular dynamics flexible fitting (MDFF) [33, 34], that currently require execution of both NAMD and VMD. The use of Swift/T can completely replace filesystem-based communication between disparate tools with direct access to in-memory data through existing Tcl bindings and inter-node communication performed by Swift/T.

2. BACKGROUND

Dynamic languages are well suited to act as the primary programming interface for molecular modeling software because of their inherent flexibility, ease of use, and not requiring complex and potentially time-consuming offline compilation phases. In exchange for their many virtues, dynamic languages often leave performance as a secondary concern, addressed in combination with traditional compiled languages, runtime code generation, just-in-time compilation (JIT), and similar approaches.

The performance-critical kernels involved in molecular modeling applications tend to be highly structured, heavily optimized, and rarely modified, lending themselves to traditional compiled languages such as C++, often with judicious use of hand-coded CPU vector intrinsics or assembly language, and in many cases CUDA or OpenCL kernels that target GPUs or other accelerators [11, 29, 21, 5, 30, 26, 28]. This observation leads naturally to programs that use traditional compiled languages for performance-critical data structures and kernels, but that use higher-level dynamic languages for the orchestration of complex molecular modeling tasks. Table 1 shows the decomposition of VMD and the VMD plugins into lines of code (LoC) associated with traditional compiled languages such as C/C++, CUDA, OpenCL, and assembly intrinsics and with dynamic languages such as Tcl and Python. While the details of this decomposition are unique to VMD, it shares much in common with other computational workbench environments that make heavy use of dynamic languages for plugins or other software components that users may want to modify, customize, or use as the basis for building derivative software.

3. VMD SCRIPTING INTERFACES

VMD [6] is a popular molecular visualization and analysis tool that incorporates user-extensible Tcl and Python scripting and many plugin modules that assist with common simulation preparation, visualization, and analysis tasks. Although VMD was designed from the outset to operate on large biomolecular complexes, the 10- to 100-million atom simulations that are performed on petascale computers represent a significant leap in size that exceeds the computing and visualization performance growth for a single workstation or compute node. The recent growth in simulation size has required many new algorithms and techniques to be developed, leading VMD to employ parallel algorithms on many-core processors and GPUs, thereby accelerating key visualization and analysis tasks [29, 27]. The VMD scripting interface has also been extended with parallel programming primitives to allow parallel molecular modeling tools to be developed and run by VMD users on petascale computers [27, 32, 28].

VMD was originally developed with a hard-coded command interpreter; but by 1995, it was replaced by an embedded Tcl interpreter that gave the program a far more powerful scripting engine and much greater user extensibility [6]. The embedded Tcl interpreter became the basis for development of many VMD scripts and plugins, including some particularly sophisticated tools for multiple-sequence and multiple-structure alignment and analysis [25] and for development of molecular dynamics force field parameters [15]. In 2000, VMD was refactored to incorporate support for Python and to support other languages through subclassing of the top-level `TextInterp` class. The addition of Python support has enabled a number of Python-based VMD plugins and analysis tools to be developed [16] and enabled VMD to be seamlessly embedded in HiMach, a Python-based MapReduce style trajectory analysis framework [35]. Recently, further subclassing of the `TextInterp` class allowed Matheny et al. to embed a Lua inter-

Table 1: Language breakdown of core VMD source code and VMD plugins in terms of lines of code (LoC).

Language	VMD Core LoC	VMD Plugins LoC
C/C++	206 K	211 K
CUDA	16 K	
OpenCL	2 K	
Assembly / intrinsics	2 K	
Tcl	38 K	252 K
Python	8 K	3 K

preter in VMD, enabling it to be used within the ExSciTecH distributed-computing client [14].

3.1 Parallel Scripting in VMD

VMD incorporates many plugins that automate complex or computationally demanding analysis and visualization tasks; particularly those that involve processing of simulation trajectories comprising tens of terabytes of data [31, 27]. VMD implements several easy-to-use commands that provide high level parallel programming abstractions that are well suited to development of visualization and analysis tools used by molecular scientists. VMD provides built-in commands for querying node counts and ranks, barrier synchronization (e.g. `parallel barrier`), collective operations (e.g. `parallel allgather`, `parallel allreduce`), and an easy to use work scheduler (`parallel for`).

The VMD parallel computing commands are provided in all VMD builds, thereby enabling visualization and analysis tools to use them irrespective of whether or not VMD was compiled with MPI support. Beyond being simple wrappers for lower-level shared memory multithreading and distributed memory message passing operations, the implementations in VMD also provide error checking to assist application scientists that are new to parallel programming and its associated challenges. The work scheduler in VMD uses fast atomic counter machine instructions, multithreading, and asynchronous communication to achieve good performance for a variety of throughput oriented analysis and visualization tasks with thousands of nodes, and enables the use of GPU accelerated kernels [27, 32, 28].

3.2 Parallel Analysis and Rendering in VMD

The VMD TimeLine trajectory analysis plugin provides an easy-to-use graphical interface for analyzing MD trajectories, calculating associated time-varying quantities, and highlighting rare or otherwise important events using both the VMD 3-D structure display and a heatmap-style 2-D matrix [27]. TimeLine is designed to help identify and assess trajectory events by performing analysis calculations for each component of a molecular system and for every frame of a simulation trajectory. The resulting TimeLine plot provides a "whole-trajectory" and a "whole-structure" view of the calculated property. The TimeLine plot is directly linked to the VMD 3-D structure display. When the user "scrubs" the mouse cursor on the TimeLine plot, the associated 3-D molecular structures are highlighted, and their configurations and motions are shown at the times of the corresponding events. Since TimeLine analyses typically involve very large datasets, the analyses it performs are tasks that are well suited to large-scale, batch-mode data-parallel computational approaches. All the graphical interfaces and high-level orchestration of TimeLine analyses are implemented with Tcl scripts that make use of both the VMD parallel scripting commands described above and fast data-parallel GPU algorithms, e.g. for tasks such as calculation of solvent-accessible surface area [27] and MDFF cross-correlation quality-of-fit metrics [28].

VMD includes various tools for rendering movies, and it supports a variety of high-quality ray tracing and advanced lighting techniques that can pose significant computational demands, leading to the use of parallel rendering [27, 32]. The existing parallel rendering tools are built using the parallel scripting interfaces in VMD; an extremely simplified pair of parallel rendering procedures is shown in Appendix A.

4. NAMD TCL SCRIPTING INTERFACE

Since we can never anticipate the needs of all NAMD users, it is imperative to enable ad hoc innovation by biomedical researchers. To allow user-driven innovations to transfer between platforms as smoothly as built-in NAMD features, users are encouraged to employ the Tcl scripting language to extend and modify NAMD. NAMD supports scripting for simulation option parsing, high-level methods such as annealing protocols, steering or restraints of small numbers of atoms (TclForces), and boundary-type forces applied to any atom in the system (TclBC). Since NAMD users are familiar with Tcl because of its incorporation in VMD, even non-programmers can use their basic skills to extend NAMD. A tutorial on "User-Defined Forces in NAMD" provides assistance.

4.1 Use of Tcl in NAMD

Scripts in NAMD run the same on all platforms and processor counts; the flexibility and power of Tcl scripting in NAMD are sufficient that the current replica exchange feature is written entirely in Tcl. Increased sizes of machines and simulations require continued innovation to ensure that scripts retain performance and scalability.

The NAMD simulation configuration is a human-readable text file, given as an argument on the NAMD command line, specifying simulation input files, output files, control parameters, and protocols. The configuration file was originally parsed by C++ code, but since most lines in the configuration file are of the form "*name value*", it was observed that this could be mapped readily to the Tcl syntax "*command argument*" to enable the use of Tcl in the configuration file while maintaining backwards compatibility.

A typical way of parsing a NAMD configuration file in Tcl would be to define a separate Tcl command for every NAMD parameter, possibly using a few generic C++ functions, with parameter-specific user data passed to Tcl when adding the command to the Tcl interpreter. However, NAMD configuration file syntax supported up to that time was case-insensitive and included both format variants such as *name=value* and end-of-line comments such as *name value #comment*, which are all violations of Tcl syntax. A solution was achieved via the Tcl `unknown` command, which is called by the Tcl interpreter whenever an unrecognized command is encountered. By replacing the

default Tcl `unknown` command with one that called the original C++ NAMD configuration file line-parsing function, complete backwards compatibility was achieved while providing the user with the complete power and flexibility of the Tcl interpreter. Other supported features include multiple configuration files on the command line and *`--name value`* command-line arguments.

Another valuable feature of Tcl configuration file parsing in NAMD is the ability to transparently subsume the configuration syntax of outside modules into the NAMD configuration file by simply encasing the module configuration text in braces or quotes, depending on the desired level of variable substitution by the Tcl interpreter. This syntax-embedding capability was first used for the NAMD 'free energy of conformational change' module and is currently used by the 'collective variables' module, which provides a comprehensive utility for biasing and monitoring the conformation of a biomolecule during a simulation.

While enabling Tcl parsing of the NAMD configuration file provided a significant usability benefit, it was then desirable to allow feedback from the full parallel NAMD simulation into Tcl. Only a single Tcl interpreter, on the rank-zero process, was desired, presenting a challenge because NAMD is written in message-driven Charm++. A solution was found in which the Charm++ parallel runtime is launched as usual across all nodes but, after initialization, the Charm++ scheduler (an event loop to process incoming messages and dispatch work) is terminated on rank zero and the Tcl interpreter is created and begins parsing the configuration file(s) and command-line arguments as input. The first `run` or `minimize` command encountered (or, failing that, at the end of input) sends a message to (its own) rank zero to trigger simulation startup and then re-launches the Charm++ scheduler to process that and future messages. After startup the simulation timesteps begin; and, on completing the final specified step, Charm++ *quiescence detection* is invoked with a callback function that exits the scheduler on rank zero, thus returning to the original Tcl interpreter command invocation for further processing.

Between `run` commands the NAMD configuration file Tcl code can modify a selection of simulation parameters (only those for which the reinitialization of related data structures and cached values is either implemented or unnecessary), write atomic positions/velocities to output files, and reinitialize atomic positions/velocities from files. The temperature, pressure, and various energies of the simulation are also available, enabling user-written annealing and other types of protocols.

Two Tcl-based mechanisms can be used to apply portable user-defined forces to the atoms in a NAMD simulation. The first, TclForces, is implemented as a user-written function callback on the master Tcl interpreter on rank zero during each timestep. The user must specify at the start of the simulation the (ideally small) set of atoms for which positions are required, but the callback function may apply forces to any atom in the simulation. In order to apply independent forces to potentially all atoms in the simulation, the TclBC (Tcl Boundary Conditions) interface is provided. A TclBC callback is invoked for each timestep simultaneously on Tcl interpreters on every core (i.e., thread) on which atoms are present, and may iterate exactly once through all atoms on the core, adding forces and energy based on the (variable) position and (fixed) charge/mass/species of the atom. It has been suggested to extend the TclBC interface with reduction and broadcast operations to allow the implementation of more general calculations but the need for such a capability has not yet arisen.

4.2 Multiple-Copy Simulation in NAMD

Although NAMD is capable of scaling only the largest biomolecular simulations to entire petascale machines [22], for the typical smaller simulations such machines may still be efficiently employed to achieve sampling equivalent to a much longer simulation by using multiple-copy algorithms (MCAs) [7]. MCAs link large numbers of otherwise independent simulations of a single biomolecular system (called *replicas*) by periodically exchanging temperature or other control parameters between pairs of replicas, typically between neighbors in a 1-D or 2-D grid. The exchanges are performed (or not) based on Metropolis criteria to ensure a proper sampling ensemble. This most common class of multiple-copy methods is referred to as *replica exchange*.

The first implementation of replica exchange in NAMD was done in 2006 and released in NAMD 2.6. This implementation employed a master-worker control structure and was implemented entirely in Tcl, with the master script requiring no NAMD-specific commands and hence able to execute in either tclsh or the NAMD Tcl interpreter (or, in theory, any other Tcl interpreter such as the one in VMD). The user was required to customize a spawn_namd.tcl file to launch NAMD runs for each of a list of NAMD configuration files that were generated by the `namd_replica_server` module. The configuration files differed only by the value the `replica_id` variable was set to, and otherwise contained the host name and port number on which the master Tcl server would listen for connections along with a generic script to open a socket to the master server and then enter a loop of listening for commands from the master, evaluating the received command, and sending the result back to the master.

The `namd_replica_server` module exported three functions: `start_replicas`, `replica_eval`, and `replica_push`. The `replica_eval` function would transmit a given Tcl script to all replicas and then wait for responses from all replicas through a socket handler function that stored the response in a Tcl array as `replica_data($replica_id.$field)`. The `replica_push` function would use the same mechanism to set a Tcl variable on each replica to the corresponding field value in the Tcl array `replica_data($replica_id.$field)`. These functions enabled a block-synchronous programming style that was sufficient to implement replica exchange.

The sockets-based master-worker implementation worked well on traditional clusters, but IBM Blue Gene and Cray XT machines originally lacked a sockets implementation on the compute-node operating system, and hence a similar communication pattern was implemented via the file system. A major drawback was the inefficiency of the fully block-synchronous programming model, requiring all replicas to reach the synchronization point when each individual exchange decision required energies from only a single neighboring replica. Trajectory and restart file output was similarly synchronized, a worst-case scenario for parallel filesystem performance. The greatest impediment from a user perspective was the need to adapt the spawn_namd.tcl scripts to each particular queueing system, partitioning the assigned node list into separate launches. Moreover, while sending data through TCP sockets or the filesystem was acceptable

for infrequent communication of a few control parameters, more intensive communication required access to the high-speed network of the machine.

Seeking to avoid the effort and complication of modifying NAMD to support multiple independent simulations internally within the Charm++ programming model, we looked with envy on the `MPI_Comm_split()` function that would allow any MPI program to be trivially run on a subset of `MPI_COMM_WORLD` via a local communicator, while also allowing communication between replicas by a set of cross-communicators between equivalent local ranks. It was fortunately realized that the MPI-based Charm++ machine layer could be easily modified in exactly this way, allowing unmodified Charm++ programs to operate independently within the local communicator of each partition, with stdout optionally redirected to a separate file per partition.

It was then a simple matter to add to the NAMD master Tcl interpreter on rank zero of each partition the simple commands `replicaSend`, `replicaRecv`, `replicaSendrecv`, `replicaBarrier`, `numReplicas`, and `myReplica`, each implemented via and mirroring the semantics of the corresponding MPI functions operating on the inter-partition rank-zero cross communicator. In addition to enabling use of the high-speed network, eliminating synchronization and bottlenecks of the master-worker model, and dramatically simplifying parallel job launching, this solution had the added advantage of exploiting the likely familiarity of advanced NAMD users with basic MPI programming concepts. Upon its initial availability in NAMD 2.9 the MPI-based replica implementation was extended from temperature exchange for parallel tempering to bias exchange for conformational free energy umbrella sampling and lambda exchange for alchemical free energy perturbation, with the latter two supporting arbitrary neighbor layouts in two or more dimensions.

The NAMD 2.9 MPI-based replica implementation had two main weaknesses. First, it was limited to the MPI-based Charm++ machine layer. Specialized machine layers have been developed to provide improved performance for Charm++ message-driven programs by bypassing MPI and instead accessing the underlying low-level communication interfaces of ethernet, InfiniBand, Cray Gemini, IBM Blue Gene, and other networks. These layers provide the greatest benefit for so-called SMP builds utilizing multiple threads and shared memory within a process and a dedicated communication thread for inter-process communication. SMP builds are essential for both reducing per-core memory requirements and for efficiently supporting accelerator offloading for GPUs and Xeon Phi.

The second weakness of MPI-based replicas is that the blocking MPI functions could be called only when NAMD was otherwise idle. Therefore inter-replica communication on every timestep would never be efficient. Such frequent communication could be used to couple collective variable or TclForces biases continuously between replicas. Implementing a Charm++ message-driven programming style inside NAMD via MPI calls was impossible, so the choice was made to add support for partitions and inter-partition communication to the Charm++ low-level run-time system (LRTS), an intermediate internal API on which the MPI and other recent machine layers such as Cray Gemini are based. As with MPI-based replicas, partitions are defined at the process level and threads within the same process cannot be shared between partitions.

The existing MPI-style `replicaSend/Recv` Tcl commands in NAMD were then re-implemented on the new Charm++ partitioning interfaces, allowing NAMD 2.9 replica-exchange scripts to run unmodified in NAMD 2.10 but more efficiently on more platforms. Also added to NAMD 2.10 were the commands `replicaAtomSend` and `replicaAtomRecv` that, rather than transferring arbitrary Tcl strings between replicas, send the full positions and velocities of all atoms directly between the distributed-memory data structures of NAMD, utilizing the full bandwidth available between all corresponding pairs of processors in the communicating partitions rather than funneling data through the rank-zero processes.

The remaining defect in the NAMD 2.10 replica implementation is the lack of load balancing across Charm++ partitions. This could be addressed with great effort and increased complexity by abandoning the partition concept and instead implementing replicas directly in NAMD; but since replicas remain a relatively rare mode of simulation this effort would be difficult to justify, and bugs would be likely introduced that did not manifest in single-copy runs. For nearly identical replicas the observed load imbalance is typically minor and results from simulation differences such as temperature, divergent load balancing due to minor timing differences, and hardware differences such as irregular toroidal network topologies [22]. The impact on performance is that the overall simulation rate is limited to that of the slowest replica/partition. The potential solution is to break the one-to-one replica-partition correspondence and instead multiplex replicas dynamically onto a smaller number of partitions, hoping that the decreased performance divergence losses compensate for the increased scaling losses due to each partition running on a larger number of processors. To manage the complexity of this multiplexing approach, we now turn to Swift/T.

5. SWIFT/T INTEGRATION

Swift/T is a new implementation of the Swift programming language [36] for high-performance computing. The new implementation operates on an MPI-based runtime and fully distributes the dataflow primitives that enable progress in the Swift model [39]. This enables extremely high task rates, running at 1.5 billion tasks/second on 512K cores of Blue Waters [2]. This rate makes it possible to consider using Swift to drive work to individual GPU warps over distributed memory [9]. Swift/T operates by translating (with optimization [2]) the user-written Swift script into a format that uses a carefully tuned C-language MPI-based runtime; this format is a Tcl script.

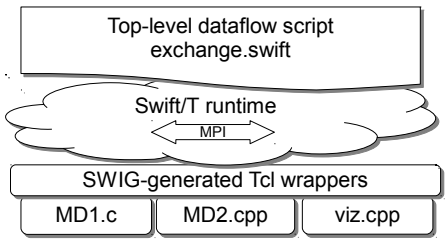

Figure 1: Typical Swift/T software integration pattern: Swift/T script coordinates user native code via Tcl interfaces over distributed infrastructure.

Swift/T normally interacts with external user code as shown in Figure 1. In a typical Swift/T use case, the user has existing codes written in C/C++/Fortran that must be composed into a parallel application. These native code libraries are exposed through bindings generated by SWIG [3] for Tcl or Python. Swift/T can then call this functionality and pass data in and out of the scripting language interface. As the Swift/T dataflow script progresses, these wrappers are called across the set of available processors and load balanced internally by ADLB [13, 40].

ADLB is designed as a minimal, high-performance master-worker system for MPI applications. Multiple ranks (∼1%) serve as masters, which distribute work to the remaining worker ranks. We extended ADLB for the Swift/T effort to support work-stealing among masters in the Scioto [4] model, in addition to adding data storage primitives to support the Swift dataflow model. A key extension, `ADLB_Dput()` (data-dependent put), extends the `ADLB_Put()` task submission call by making the task dependent on a data write; the task will not be released to the work queue until all its data dependencies are met.

Swift/T tasks may execute anywhere in the system unless constrained by the user. ADLB-level task properties are exposed at the Swift/T level, providing a rich feature set. Tasks may be assigned to a particular rank using hard or soft constraints, where soft-targeted tasks are prioritized by the target but allowed to be stolen by idle masters. Tasks may also be assigned a type, and workers can restrict work requests to given types, or any type. Tasks may be assigned a priority value relative to other tasks on the master (no attempt is made to achieve global prioritization).

Task-task communication in Swift/T is performed in a functional manner by connecting task outputs to task inputs using Swift data types including `int` (64-bit), `float` (64-bit), `string`, `blob`, and possibly nested arrays, structs and `typedef`s of these types. A blob is a binary byte array, represented in Tcl as a `[list` *pointer length*`]`. The Swift/T distribution comes with a Tcl library (`blobutils`) to facilitate blob management and transmission to and from SWIG-wrapped (strongly typed) native code functions.

Data movement is implemented by our extensions to the ADLB API, augmenting its task-oriented `Put`/`Get` calls, with `Store`/`Retrieve` and containers (for data structures). This essentially creates an MPI-based tuple space for dataflow processing (and can be used outside Swift). The ADLB data store is automatically garbage-collected by using reference counting.

The rich Tcl features of NAMD allow a powerful, new programming model that leverages the dynamic execution capabilities of Tcl. In this model, the Swift/T script is compiled to generate a Tcl program. This program is launched *by NAMD*, that is, using NAMD as a Tcl interpreter, as illustrated in Figure 2. NAMD thus launches the Swift/T program across its processes. As the Swift script progresses, it can access data and perform arbitrary operations in the NAMD context by *calling up* to the NAMD Tcl interpreter. This is performed with the Tcl `uplevel` command, which is like `eval` in many scripting languages (Perl, Python, Bash) but operates in a user-specified, calling stack frame, and returns a result. Thus, arbitrary NAMD features are available to Swift/T, creating a highly dynamic programming paradigm.

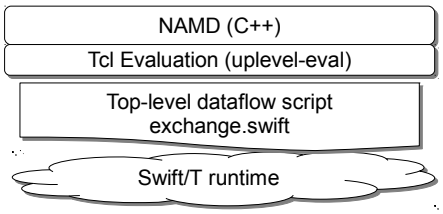

Figure 2: NAMD/Swift software integration: NAMD Tcl interpreter invokes Swift/T, exposing NAMD data via Tcl `uplevel`, `eval` features.

5.1 Swift/T Integration with NAMD

Swift/T programs may be executed by using NAMD built on the single-node-only shared-memory "multicore" Charm++ machine layer. This is achieved by virtue of the Swift/T Turbine engine Tcl module dynamically loading libtclturbine.so, which forces dynamic loading of the MPI library on which Turbine itself was build. Since NAMD multicore binaries do not use MPI, there is no potential for conflict, and standard downloaded NAMD binaries can be used so long as Turbine is built with a compatible underlying Tcl version.

The NAMD binary is launched with mpiexec, specifying via the `-n` argument one rank more than the desired number of NAMD worker ranks. The NAMD/Charm++ `+p` argument is used to specify the number of threads per NAMD process (e.g., the number of cores per node for a multi-node run with one worker per node). The last rank NAMD process will function as the Swift/T master, which is single-threaded, so a wrapper script may be used to limit this process to a single core, while a worker process running on the same node would run on one fewer cores. While the load imbalance due to this single slower worker would limit the overall performance of the Charm++ partition-based NAMD replica exchange implementation, its impact on the throughput-based and dynamically load-balanced Swift implementation should be negligible.

Execution of a NAMD/Swift program begins with uniform initialization, in which all NAMD ranks independently load the same NAMD configuration file specifying the molecular system and simulation parameters that cannot be altered in NAMD once the simulation has begun. Then the namdswift.tcl file is loaded, which sets Tcl variables for the various replica-exchange parameters, defines `min` and `max` math operations using definitions copied from the Tcl distribution init.tcl file (which NAMD does not otherwise require), sources the Turbine library pkgIndex.tcl file to make the dynamically loaded turbine package available to the interpreter, initializes the NAMD random number generator seed based on the environment variable (e.g., `PMI_RANK`) containing the MPI rank, defines NAMD Tcl callbacks used to access simulation energies, and defines the NAMD leaf functions that will be called by the Swift program.

The definition of NAMD/Swift leaf functions is aided by the Tcl wrapper commands shown in Appendix B. The `swift_proc` and `swift_return` Tcl commands take the place of the Tcl built-in `proc` and `return` commands that are normally used to define new functions. As shown by their use in defining the `run_t` leaf function in Appendix C, `swift_proc`

allows the definition of typed input and output arguments using the type nomenclature of the Swift programming language. Each invocation of the `swift_proc` command defines a pair of functions to interface with the Turbine engine. The first is a rule function, which is called by Swift to define when a particular function invocation is first runnable. The second is a body function, which is called on a worker rank when all input arguments are available and which may consume significant runtime. The Swift body function is generated by wrapping the Tcl function body passed to `swift_proc` with code to first access the input parameters and store them in local Tcl variables corresponding to the input argument names, and then, when either `swift_return` is called or the end of the function body is reached, to store the values of the output parameters to the Swift runtime.

Swift/T code for implementing replica exchange in NAMD is shown in Appendix D. Various system modules are imported, and trivial leaf functions are defined for evaluating Tcl commands to return string, integer, and floating point values to Swift. (Note that the Tcl set command, when called with a variable name but no value, returns the current value of the variable, and hence the Swift expression `tcl_eval("set` *`varname`*`")` returns the string value of the Tcl variable *varname* to the Swift program.) The run_t leaf function *defined* in Tcl in Appendix C is *declared* with matching argument types as corresponding to the `run_t` Tcl command in the namdswift package.

The actual Swift/T main function is more compact and transparent than the equivalent NAMD/Tcl replica exchange implementation shown in Appendix E, but note that the code in the body of the `run_t` Swift leaf function must also be considered. In addition, the leaf function body is in a different source file and the programmer must contend with both Tcl and Swift syntax and semantics. The contribution of Swift/T to the ease of parallel algorithm programming is that while the code is *written* in a easily understood block-synchronous style, it is *executed* in a fully dynamic and data-driven workflow manner. The price of this clarity is that in the Swift model any variable or array element may be written to only once; hence, the exchange frame index `f` is the first dimension index on many arrays, and, in theory, state data proportional to the length of the run must be maintained by the Swift runtime.

While the NAMD/Tcl replica exchange implementation was able to operate while exchanging only control parameters between replicas permanently attached to a partition, the Swift/T runtime can schedule any leaf function on any worker rank, and therefore the entire dynamic state of every replica simulation (i.e., atomic positions and velocities but not the common molecular topology and force field definitions) must be loaded at the beginning of the `run_t` leaf function and saved on its completion. The atomic state is saved to and read from the filesystem via the existing NAMD `output` and `reinitatoms` commands (see Appendix C) while only the file names are passed through the Swift runtime. Storing potentially large atomic data on the filesystem reduces the memory usage of the Swift runtime and works well for testing on a single machine with a fast local filesystem, in particular on a high-bandwidth, low-latency solid-state disk.

NAMD multicore binaries utilizing MPI only for Swift runtime communication as described above are limited to, at most, a process, and hence a single node, per replica. In order to utilize multiple nodes per replica for higher performance, a mechanism for NAMD inter-node communication must be devised that is compatible with the Swift runtime's use of MPI. We turn again to modification of the Charm++ MPI machine layer, now with the LRTS-based partition mechanism, using `MPI_Comm_split` to again create cross-communicators among corresponding ranks of Charm++ partitions. On each rank a pointer to the cross-communicator is stored and made available at startup in the NAMD Tcl interpreter as the 64-bit integer variable `TURBINE_ADLB_COMM`. The existence of this variable directs the Swift runtime to operate in library mode, using the existing MPI communicator rather than `MPI_COMM_WORLD` and not calling `MPI_Init/Finalize`. Library mode further enables Swift to be invoked repeatedly in a single NAMD parallel execution. Moreover, the NAMD myReplica Tcl command can be used to reliably initialize the random number generators without attempting to parse environment variables, or the default initialization can be safely used as the default wall-clock time seed is modulated based on the Charm++ partition index.

We tested the tightly-integrated MPI NAMD/Swift replica exchange implementation on up to 512 nodes of the Blue Waters supercomputer at the University of Illinois. Each Charm++ partition was one to four nodes of 32 ranks each, and one to four replicas were run per partition, with one partition used for the Swift runtime master. The atomic state files used to facilitate Swift task mobility were written to the Lustre-based scratch filesystem, using a separate directory for each replica to distribute load on and reduce contention for the Lustre metadata servers.

Significant performance variation was observed between runs for both the Swift and native NAMD/Tcl implementations. For fairness the Charm++ MPI machine layer was used for both, even though partitions are now available for the higher-performance Cray Gemini Charm++ machine layer. The best performance was typically from the native implementation, although the Swift implementation did occasionally outperform it. Our primary observation is that although Swift is by design quite *tolerant* of performance variation among workers, our current use of the filesystem for data exchange results in a massive and insurmountable *exposure* to storage server contention both from other replicas and from other jobs running on the machine. The Luster filesystems on Blue Waters are currently also the most failure-prone components of the system, so our pattern of intensive filesystem access would likely destabilize the machine for other users if employed for extended production runs.

The obvious next direction for NAMD/Swift development is to move from storing and accessing transient data on the filesystem to instead using the Swift `blob` raw binary datatype. In order to support extended runs, the Swift runtime will need to be capable of forgetting data elements that are no longer needed, likely under programmer control. With filesystem constraints removed we will be able to observe the performance advantages of the Swift programming model.

The second direction will be to make Swift compatible with the Charm++ native machine layers that will improve the performance of individual NAMD partitions. This can be accomplished either through the evolving capability of Charm++ to interoperate with MPI programs or by modify-

ing Swift to communicate between ranks using the Charm++ inter-partition communication functions rather than MPI.

Many Swift/T features, existing or planned, could be applied to enhance replica exchange algorithms in NAMD. The implicit concurrency of dataflow programming makes it particularly well-suited to dynamic infrastructures such as clouds, and fault-tolerant for next-generation HPC machines that may allow applications to manage fault recovery. Swift/T is built on MPI, which limits what we can currently do to support elasticity and fault tolerance, but MPI feature enhancements are expected in this area. Swift/T offers an automated checkpointing system that could be used for restart, as well as programming error diagnosis and possibly user-directed branching of ensemble progress. Combining checkpoint records with priorities could be used to develop a "catch-up" mechanism [37] to ameliorate delays due to unexpectedly slow replicas on heterogeneous computers. Most important, we intend that the Swift/T model enables and motivates novel algorithm development in highly asynchronous ensemble algorithms for molecular dynamics: these will make the best use of emerging exascale machines.

5.2 Swift/T Integration with VMD

Swift/T has been integrated with VMD in a manner directly analogous to the NAMD shared-memory "multicore" integration described above, with mpiexec used to launch a standard VMD across nodes. A wrapper script may be used to redirect input from /dev/null and output to a separate file per rank. If multiple VMD ranks are launched per node, then special environment variables must be set to force each VMD rank to limit its CPU usage to a specified number of CPU cores per rank and to prevent inadvertent sharing of GPUs by multiple VMD ranks.

We have adapted previously developed VMD parallel analysis scripts for solvent accessible surface area (SASA) [27] and cross-correlation quality-of-fit score calculation [28] to operate with Swift/T. The adaptations to Swift/T were trivial to implement since the original implementations based on the VMD `parallel for` construct each called a worker subroutine on a range of iterations, which could be directly wrapped as a Swift leaf function and called in a Swift `foreach` loop.

The Swift versions are, however, more complex than the existing VMD parallel commands since they require the use of the Swift language in addition to Tcl to implement even a trivial work distribution scheme. The comparative advantages of integrating VMD with Swift can be demonstrated only on more complex workflows that do not lend well to a bulk synchronous programming style and in cases that require execution of other programs where Swift could eliminate filesystem-based communication in favor of direct message passing.

Launching the existing MPI-based parallel version of VMD, modified as NAMD/Charm+++ above to set the TURBINE_ADLB_COMM Tcl variable to a pointer to the MPI communicator on startup, will cause Swift to operate in library mode, allowing repeated invocations of multiple Swift programs. The VMD `parallel for` control structure could then be implemented as a generic Swift `foreach` loop, providing an equally functional substitute and a path to implementing more complex control structures.

5.3 Swift/T Integration with Other Software

Swift/T has been integrated with many other applications in molecular dynamics including Rosetta [8], DOCK6 [10], and LAMMPS [23]. Additionally, many Swift/T applications exist in other domains, including materials science [38], power grid modeling [39], and visualization [41].

Swift/T can be integrated with applications in multiple ways. To integrate with Rosetta and DOCK6, we used a technique called *main-wrapping*. In this model, we simply rename the C or C++ `main()` function and recompile the application as a library (shared or static). We then use SWIG to generate the Tcl binding to this function, and call it in the normal way from Swift/T. This allows the Swift script to operate much like a shell script, passing an array of strings to each application invocation, but with Swift/T concurrency semantics. LAMMPS is naturally built as a C++ library; its `main()` routine is minimal. We simply applied SWIG to the LAMMPS header, exposing the full LAMMPS API to the Swift programmer.

In a materials science application based on DISCUS [24], our collaborator applied F2PY [19] to generate Python bindings for key DISCUS features. Swift/T provides an optional built-in Python 2.7 interpreter if configured to do so. Thus, Swift/T can easily call the DISCUS Python package. The goal of this application was to produce a crystal structure model based on X-ray scattering experimental data via inverse modeling. A genetic algorithm was developed in about 200 lines of Swift to run concurrent DISCUS simulations as population members, converging toward a good approximation of the crystal structure that fit the experimental data.

Our power grid application was an extension of previous work [12] on power grid modeling and scheduling for a "smart grid" sensitive to weather, renewable energy sources, and predicted load. The prior work produced a schedule for the grid. Our Swift-based application applied SWIG to the C++ header of the same codebase, but ran it in a different mode to check the schedule against a large quantity of scenarios for risk analysis—an ideal Swift problem.

Our visualization application was based on OSUFlow [18], a flow-line visualization package. This application task is a call to OSUFlow as an MPI library. Thus, we applied the Swift/T feature that can produce a variable-sized MPI subcommunicator via `MPI_Comm_create_group()`. The OSUFlow call of interest was wrapped for Tcl via SWIG, and a small amount of Tcl glue code obtains the subcommunicator from Swift/T and passes it to the library for task usage.

6. CONCLUSIONS

We have discussed the merits of dynamic languages in the context of petascale molecular modeling workloads, highlighting their ease of use by application scientists and describing the orchestration of large-scale NAMD replica simulations and VMD analysis workflows with Tcl and the Swift/T dataflow programming language. Several examples of parallel scripting have been provided in the appendices, and we have made the complete source code for these and other scripts available.[1] The Swift source code is also freely available.[2]

[1] `http://www.ks.uiuc.edu/Research/swift/`
[2] `http://swift-lang.org`

APPENDIX

A. VMD PARALLEL MOVIE EXAMPLE

```
proc render_one_frame { frameno userdata } {
  # retrieve user data rendering workers
  set formatstr  [lindex $userdata 0]
  set dir        [lindex $userdata 1]
  set renderer   [lindex $userdata 2]

  # Set frame, triggering user-defined movie
  # callbacks to update the molecular scene
  # prior to rendering of the frame
  set ::MovieMaker::userframe $frameno

  # Regenerate molecular geometry if not up to date
  display update

  # generate output filename, and render the frame
  set fname [format $formatstr $frameno]
  render $renderer $dir$fname
}

proc render_movie { dir formatstr framecount renderer } {
  set userdata {}
  lappend userdata $formatstr
  lappend userdata $dir
  lappend userdata $renderer

  set lastframe [expr $framecount - 1]
  parallel for 0 $lastframe render_one_frame $userdata
}
```

B. SWIFT/T LEAF FUNCTION WRAPPER

```
proc swift_proc { typed_outputs name typed_inputs body } {
  foreach i [lsearch -exact -all $typed_outputs int] {
    if { $i % 2 == 0 } { lset typed_outputs $i integer }
  }
  foreach i [lsearch -exact -all $typed_inputs int] {
    if { $i % 2 == 0 } { lset typed_inputs $i integer }
  }
  set body_args {}
  set output_code ""
  foreach { type arg } $typed_outputs {
    lappend body_args __swift_proc_output_$arg
    set output_code "${output_code}\
          store_$type \$__swift_proc_output_$arg \$$arg; "
  }
  set input_code \
     " set __swift_proc_typed_outputs [list $typed_outputs];"
  foreach { type arg } $typed_inputs {
    lappend body_args $arg
    set input_code \
      "${input_code} set $arg \[ retrieve_$type \$$arg \];"
  }

  set rule1 {rule $inputs [concat }
  set rule2 "[uplevel namespace current]::"
  set rule3 {_body $outputs $inputs] type $turbine::WORK}
  set rule $rule1$rule2$name$rule3
  set fullbody "$input_code$body\n$output_code"
  puts [list proc $name { outputs inputs } $rule]
  puts [list proc ${name}_body $body_args $fullbody]
  uplevel [list proc $name { outputs inputs } $rule]
  uplevel [list proc ${name}_body $body_args $fullbody]
}

proc swift_return { args } {
  upvar __swift_proc_typed_outputs typed_outputs
  foreach { type arg } $typed_outputs val $args {
    upvar __swift_proc_output_$arg __swift_proc_output_local
    store_$type $__swift_proc_output_local $val
  }
  return -code return
}
```

C. WRAPPED LEAF FUNCTION FOR NAMD

```
namespace eval namdswift {

swift_proc {string o float POTENTIAL} run_t \
    {string i int r int f int n float NEWTEMP float OLDTEMP} {
  global replica_index output_index output_root \
         saved_array steps_per_run
  set o $output_root.$r.$f
  stdout $o.log
  puts "Replica $replica_index running $i for $n steps to $o"
  firsttimestep [expr ($f-1)*$steps_per_run]
  reinitatoms $i
  rescalevels [expr sqrt(1.0*$NEWTEMP/$OLDTEMP)]
  langevinTemp $NEWTEMP
  ::run $n
  output $o
  save_array ;# stores energies in saved_array
  swift_return $o $saved_array(POTENTIAL)
  error "this should never happen"
}
}
```

D. REPLICA EXCHANGE IN SWIFT/T

```
import io;
import sys;
import math;
import random;

(string o) tcl_eval(string s) "turbine" "0.0" [
  "set <<o>> [ uplevel #0 <<s>> ]"
];

(int o) tcl_eval_int(string s) "turbine" "0.0" [
  "set <<o>> [ uplevel #0 <<s>> ]"
];

(float o) tcl_eval_float(string s) "turbine" "0.0" [
  "set <<o>> [ uplevel #0 <<s>> ]"
];

(string o, float POTENTIAL) run_t
  (string i, int r, int f, int n, float NEWTEMP, float OLDTEMP)
  "namdswift" "0.1" "run_t";

main
{
  string ifile = tcl_eval("set ifile");
  int num_replicas = tcl_eval_int("set num_replicas");
  int num_runs = tcl_eval_int("set num_runs");
  int steps_per_run = tcl_eval_int("set steps_per_run");
  float min_temp = tcl_eval_float("set min_temp");
  float max_temp = tcl_eval_float("set max_temp");
  printf("Running %d replicas from %f to %f for %d runs",
    num_replicas, min_temp, max_temp, num_runs);
  float TEMPERATURE[int];
  string states[int][int];
  int sources[int][int];
  float POTENTIAL[int][int];
  foreach i in [0:num_replicas-1] {
    TEMPERATURE[i] = min_temp * exp(
      log(max_temp/min_temp)*(itof(i)/itof(num_replicas-1)) );
    states[1][i], POTENTIAL[1][i] =
      run_t(ifile, i, 1, steps_per_run, TEMPERATURE[i], 300);
  }
  foreach f in [2:num_runs] {
    if ( f%%2 == 1 ) {
      sources[f][0] = 0;
    }
    if ( (num_replicas+f)%%2 == 1 ) {
      sources[f][num_replicas-1] = num_replicas-1;
    }
    foreach i in [f%%2+1:num_replicas-1:2] {
      BOLTZMAN = 0.001987191;
      dbeta =
        ((1.0/TEMPERATURE[i-1]) - (1.0/TEMPERATURE[i])) / BOLTZMAN;
      float delta = dbeta *
        (POTENTIAL[f-1][i] - POTENTIAL[f-1][i-1]);
      boolean doswap = (delta < 0.0) || (exp(-delta) > random());
      printf("frame %d reps %d %d swap %s\n", f, i-1, i, doswap);
      if ( doswap ) {
        sources[f][i] = i-1;
        sources[f][i-1] = i;
      } else {
        sources[f][i] = i;
```

```
        sources[f][i-1] = i-1;
      }
    }
    foreach i in [0:num_replicas-1] {
      int isrc = sources[f][i];
      states[f][i], POTENTIAL[f][i] =
        run_t(states[f-1][isrc], i, f, steps_per_run,
        TEMPERATURE[i], TEMPERATURE[isrc]);
    }
  }
}
```

E. REPLICA EXCHANGE IN TCL

The following Tcl-only NAMD replica exchange code corresponds to the "`foreach f in [2:num_runs]`" loop in the Swift/T version above and executes on all Charm++ partitions in parallel.

```
while {$i_run < $num_runs} {

  run $steps_per_run
  save_array
  incr i_step $steps_per_run
  set TEMP $saved_array(TEMP)
  set POTENTIAL $saved_array(POTENTIAL)
  puts $history_file \
          "$i_step $replica(index) $NEWTEMP $TEMP $POTENTIAL"

  if { $i_run % 2 == 0 } {
    set swap a; set other b
  } else {
    set swap b; set other a
  }

  set doswap 0
  if { $replica(index) < $replica(index.$swap) } {
    set temp $replica(temperature)
    set temp2 $replica(temperature.$swap)
    set BOLTZMAN 0.001987191
    set dbeta [expr ((1.0/$temp) - (1.0/$temp2)) / $BOLTZMAN]
    set pot $POTENTIAL
    set pot2 [replicaRecv $replica(loc.$swap)]
    set delta [expr $dbeta * ($pot2 - $pot)]
    set doswap [expr $delta < 0. || exp(-1.*$delta) > rand()]
    replicaSend $doswap $replica(loc.$swap)
    if { $doswap } {
      set rid $replica(index)
      set rid2 $replica(index.$swap)
      puts stderr \
        "EXCHANGE $rid ($temp) $rid2 ($temp2) RUN $i_run"
      incr replica(exchanges_accepted)
    }
    incr replica(exchanges_attempted)
  }
  if { $replica(index) > $replica(index.$swap) } {
    replicaSend $POTENTIAL $replica(loc.$swap)
    set doswap [replicaRecv $replica(loc.$swap)]
  }

  set newloc $r
  if { $doswap } {
    set newloc $replica(loc.$swap)
    set replica(loc.$swap) $r
  }
  set replica(loc.$other) [replicaSendrecv \
        $newloc $replica(loc.$other) $replica(loc.$other)]
  set oldidx $replica(index)
  if { $doswap } {
    set OLDTEMP $replica(temperature)
    array set replica [replicaSendrecv \
              [array get replica] $newloc $newloc]
    set NEWTEMP $replica(temperature)
    rescalevels [expr sqrt(1.0*$NEWTEMP/$OLDTEMP)]
    langevinTemp $NEWTEMP
  }

  incr i_run
}
```

ACKNOWLEDGMENTS

This and other NAMD and VMD development is supported by National Institutes of Health grants 9P41GM104601 and 5R01GM098243-02, directed by Klaus Schulten.

This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications. This work is also part of the Petascale Computational Resource (PRAC) grant "The Computational Microscope", which is supported by the National Science Foundation (awards OCI-0832673 and ACI-1440026).

The Swift parallel scripting language is supported in part by NSF award ACI 1148443 and the U.S. DOE Office of Science under contract DE-AC02-06CH11357.

REFERENCES

[1] D. Aristoff, T. Lelièvre, C. G. Mayne, and I. Teo. Adaptive multilevel splitting in molecular dynamics simulations. *ESAIM: Proc.*, 2014. In Press.

[2] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Proc. SC '14*, Nov. 2014.

[3] D. Beazley. Automated scientific software scripting with SWIG. *Future Generation Computer Systems*, 19(5):599–609, 2003.

[4] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A framework for global-view task parallelism. pages 586–593, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[5] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl. Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation. *J. Chem. Theor. Comp.*, 4:435–447, 2008.

[6] W. Humphrey, A. Dalke, and K. Schulten. VMD – Visual Molecular Dynamics. *J. Mol. Graphics*, 14:33–38, 1996.

[7] W. Jiang, J. Phillips, L. Huang, M. Fajer, Y. Meng, J. Gumbart, Y. Luo, K. Schulten, and B. Roux. Generalized scalable multiple copy algorithms for molecular dynamics simulations in NAMD. *Comput. Phys. Commun.*, 185:908–916, 2014.

[8] K. W. Kaufmann, G. H. Lemmon, S. L. Deluca, J. H. Sheehan, and J. Meiler. Practically useful: what the Rosetta protein modeling suite can do for you. *Biochemistry*, 49:2987–2998, 2010.

[9] S. J. Krieder, J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu. Design and evaluation of the GeMTC framework for GPU-enabled many task computing. In *Proc. HPDC*, 2014.

[10] P. T. Lang, S. R. Brozell, S. Mukherjee, E. F. Pettersen, E. C. Meng, V. Thomas, R. C. Rizzo, D. A. Case, T. L. James, and I. D. Kuntz. DOCK 6: Combining techniques to model RNA-small molecule complexes. *RNA*, 15(6):1219–1230, June 2009.

[11] E. Lindahl, B. Hess, and D. van der Spoel. Gromacs 3.0: A package for molecular simulation and trajectory analysis. *J. Mol. Mod.*, 7(8):306–317, 2001.

[12] M. Lubin, C. Petra, M. Anitescu, and V. Zavala. Scalable stochastic optimization of complex energy systems. In *Proc. SC*, 2011.

[13] E. L. Lusk, S. C. Pieper, and R. M. Butler. More scalability, less pain: a simple programming model and its implementation for extreme computing. *SciDAC Review*, 17:30–37, Jan. 2010.

[14] M. Matheny, S. Schlachter, L. M. Crouse, E. T. Kimmel, T. Estrada, M. Schumann, R. Armen, G. M. Zoppetti, and M. Taufer. ExSciTecH: expanding volunteer computing to explore science, technology, and health. In *eScience'12*, pages 1–8, 2012.

[15] C. G. Mayne, J. Saam, K. Schulten, E. Tajkhorshid, and J. C. Gumbart. Rapid parameterization of small molecules using the Force Field Toolkit. *J. Comp. Chem.*, 34:2757–2770, 2013.

[16] J. Mongan. Interactive essential dynamics. *J. Comp.-Aided Mol. Design*, 18:433–436, 2004.

[17] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proc. IEEE*, 96:879–899, 2008.

[18] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields. In *Proc. IPDPS*, Anchorage AK, 2011.

[19] P. Peterson. F2PY: a tool for connecting Fortran and Python programs. *Int. J. Comput. Sci. Eng.*, 4(4):296–305, Nov. 2009.

[20] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten. Scalable molecular dynamics with NAMD. *J. Comp. Chem.*, 26:1781–1802, 2005.

[21] J. C. Phillips, J. E. Stone, and K. Schulten. Adapting a message-driven parallel application to GPU-accelerated clusters. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Piscataway, NJ, USA, 2008. IEEE Press.

[22] J. C. Phillips, Y. Sun, N. Jain, E. J. Bohm, and L. V. Kalé. Mapping to irregular torus topologies and other techniques for petascale biomolecular simulation. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '14. IEEE Press, 2014.

[23] S. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J Comp Phys*, 117:1–19, 1995.

[24] T. Proffen and R. Neder. DISCUS: A program for diffuse scattering and defect-structure simulation. *J. Applied Crystallography*, 30(2):171–175, 1997.

[25] E. Roberts, J. Eargle, D. Wright, and Z. Luthey-Schulten. MultiSeq: Unifying sequence and structure data for evolutionary analysis. *BMC Bioinformatics*, 7:382, 2006.

[26] J. E. Stone, D. Gohara, and G. Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Comput. in Sci. and Eng.*, 12:66–73, 2010.

[27] J. E. Stone, B. Isralewitz, and K. Schulten. Early experiences scaling VMD molecular visualization and analysis jobs on Blue Waters. In *Extreme Scaling Workshop (XSW), 2013*, pages 43–50, Aug. 2013.

[28] J. E. Stone, R. McGreevy, B. Isralewitz, and K. Schulten. GPU-accelerated analysis and visualization of large structures solved by molecular dynamics flexible fitting. *Faraday Discuss.*, 2014. In press. doi:10.1039/C4FD00005F.

[29] J. E. Stone, J. C. Phillips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *J. Comp. Chem.*, 28:2618–2640, 2007.

[30] J. E. Stone, J. Saam, D. J. Hardy, K. L. Vandivort, W. W. Hwu, and K. Schulten. High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs. In *Proceedings of the 2nd Workshop on General-Purpose Processing on Graphics Processing Units, ACM International Conference Proceeding Series*, volume 383, pages 9–18, New York, NY, USA, 2009. ACM.

[31] J. E. Stone, K. L. Vandivort, and K. Schulten. Immersive out-of-core visualization of large-size and long-timescale molecular dynamics trajectories. *Lect. Notes in Comp. Sci.*, 6939:1–12, 2011.

[32] J. E. Stone, K. L. Vandivort, and K. Schulten. GPU-accelerated molecular visualization on petascale supercomputing platforms. In *Proceedings of the 8th International Workshop on Ultrascale Visualization*, UltraVis '13, pages 6:1–6:8, New York, NY, USA, 2013. ACM.

[33] L. G. Trabuco, E. Villa, K. Mitra, J. Frank, and K. Schulten. Flexible fitting of atomic structures into electron microscopy maps using molecular dynamics. *Structure*, 16:673–683, 2008.

[34] L. G. Trabuco, E. Villa, E. Schreiner, C. B. Harrison, and K. Schulten. Molecular Dynamics Flexible Fitting: A practical guide to combine cryo-electron microscopy and X-ray crystallography. *Methods*, 49:174–180, 2009.

[35] T. Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 56:1–56:12, Piscataway, NJ, USA, 2008. IEEE Press.

[36] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Par. Comp.*, 37:633–652, 2011.

[37] C. J. Woods, M. H. Ng, S. Johnston, S. E. Murdock, B. Wu, K. Tai, H. Fangohr, P. Jeffreys, S. Cox, J. G. Frey, M. S. P. Sansom, and J. W. Essex. Grid computing and biomolecular simulation. *Philosophical Transactions of the Royal Society A*, 363(1833), 2005.

[38] J. M. Wozniak, T. G. Armstrong, D. S. Katz, M. Wilde, and I. T. Foster. Toward computational experiment management via multi-language applications, 2014. DOE ASCR Workshop on Software Productivity for eXtreme scale Science (SWP4XS).

[39] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster. Turbine: A distributed-memory dataflow engine for high performance many-task applications. *Fundamenta Informaticae*, 28(3), 2013.

[40] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Large-scale application composition via distributed-memory data flow processing. In *Proc. CCGrid '13*, pages 95–102, May 2013.

[41] J. M. Wozniak, T. Peterka, T. G. Armstrong, J. Dinan, E. L. Lusk, M. Wilde, and I. T. Foster. Dataflow coordination of data-parallel tasks via MPI 3.0. In *Proc. EuroMPI*, 2013.

[42] G. Zhao, J. R. Perilla, E. L. Yufenyuy, X. Meng, B. Chen, J. Ning, J. Ahn, A. M. Gronenborn, K. Schulten, C. Aiken, and P. Zhang. Mature HIV-1 capsid structure by cryo-electron microscopy and all-atom molecular dynamics. *Nature*, 497:643–646, 2013.