5 NAMD: Scalable Molecular Dynamics Based on the Charm++ Parallel Runtime System

Bilge Acun, Ronak Buch, Laxmikant Kale, and James C. Phillips

CONTENTS

5.1 INTRODUCTION

Nanoscale Molecular Dynamics (NAMD) is a scalable molecular dynamics (MD) application designed for high-performance atomic-level simulation of large biomolecular systems at a *femtosecond* time step resolution [1]. Used by tens of thousands of scientists—85,000 users since 2000—on everything from laptops and desktops to supercomputers and graphical processing units (GPUs) (even iPads!), NAMD has enabled significant breakthroughs in understanding the structure and the behavior of viruses and cellular organelles.

The fixed size nature of the molecular systems requires fine-grained parallelization techniques and *strong scaling* to achieve efficient simulation of long timescales. NAMD is built on top of the parallel framework Charm++ [2,3], which provides a robust infrastructure that enables simulations of molecular systems that has billions to scale up to hundreds of thousands of cores, while providing portability with performance across different supercomputing architectures. The NAMD and Charm++ collaboration was started in 1992 by principal investigators Klaus Schulten, Laxmikant V. Kale, and Robert Skeel. NAMD's performance needs have motivated Charm++ to develop new methods and abstractions resulting in successful codevelopment and interdisciplinary collaboration. In 2012, the IEEE Computer Society's Sidney Fernbach Award was jointly awarded to Kale and Schulten "for outstanding contributions to the development of widely used parallel software for large biomolecular systems simulation." One example of this is the petascale simulation of the HIV capsid with 64 million atoms on the Blue Waters Cray XE6 System at NCSA, enabling the precise determination of the chemical structure of the HIV capsid for the first time. This groundbreaking study was published and featured in *Nature* in 2013 and "petascale simulations are now being used to explore the interactions of the full capsid with drugs and with host cell factors critical to the infective cycle" [4]. Meanwhile, our efforts continue to prepare NAMD for exascale simulations not only in terms of performance and portability, but also in terms of reliability and energy efficiency as well.

This chapter discusses the scientific methodology behind NAMD (Section 5.2), algorithmic details (Section 5.3), NAMD's Charm++ based programming approach (Section 5.4), software practices used in maintaining NAMD and Charm++ (Section 5.5), scaling results (Section 5.6), and finally reliability and energy considerations for future architectures (Section 5.7).

5.2 SCIENTIFIC METHODOLOGY

NAMD is one of many programs enabling what are generally called MD simulations, distinguished from molecular mechanics (MM) calculations. MM provides energies and forces associated with a particular molecular configuration (three-dimensional [3D] coordinates for all atoms), which can combine with a minimization algorithm to obtain an optimized low-energy configuration. MD simulation, in contrast, calculates a trajectory of atomic coordinates evolved from initial atomic positions and velocities via classical dynamics as expressed by Newton's second law, $F = ma$.

It is important to distinguish the properties of dynamics simulations as applied to molecular systems from other *n-body problems* such as planetary dynamics under gravitational interactions. In planetary dynamics, the initial positions and velocities of the bodies being simulated are known with high accuracy, allowing the simulation to be run for millennia not only forwards, but also backwards to reproduce historical observations. The numerical integrators employed for such calculations must be of comparable high accuracy.

In molecular simulations, the initial positions are either of low accuracy, as from X-ray crystallography of proteins, or are randomly generated representative configurations of bulk materials such

as water and lipid membranes, and hence only the statistical behavior of the simulated system is of interest rather than the specific trajectory. The choice of an appropriate integrator for MD is governed not by short-time accuracy but by long-time properties such as conservation of energy, and therefore the symplectic and reversible Verlet integrator is most often employed, with the trajectory generated at fixed time steps and stored at fixed intervals for later analysis.

Atomic velocities are generated randomly based on classical thermodynamics for the target temperature of the simulation, and the simulation must be run for an equilibration period in order to establish an equipartition between position and velocity degrees of freedom. During the equilibration period, and often for the duration of the simulation, the temperature of the system is steered toward a target by either random or deterministic adjustments to the atomic velocities.

Simulations of biomolecular systems in particular are often performed under periodic boundary conditions, in which the simulated particles interact not only directly with each other, but also with an infinite regular 3D lattice of images. This lattice does not typically correspond to an actual crystal structure, but is arbitrarily sized so as to contain the simulated structure plus a suitable quantity of solvent (water and ions). Such boundary conditions eliminate edge effects such as surface tension and evaporation and thus provide an illusion of infinite bulk solvent. Despite the infinite domain available, the volume per image is that of a single periodic cell. If the volume is too large bubbles will form during the simulation, whereas if the volume is small and incompressible liquid-phase simulation will experience high pressure. For this reason, the pressure of the system must be equilibrated as well by adjustments to the cell volume. The individual cell dimensions may be adjusted independently for anisotropic systems such as a bilayer membrane, or uniformly for bulk solvent. Again, both random and deterministic methods are available.

Temperature and pressure control enable simulations to be conducted in three common thermodynamic ensembles: NVE (constant particle count, volume, and energy), NVT (constant particle count, volume, and temperature), and NPT (constant particle count, pressure, and temperature). Advanced methods may enable simulations in a constant chemical potential ensemble where the number of particles varies, the most common being a constant pH ensemble in which protons (hydrogen ions) are added and removed from specific titratable locations in the molecule.

The forces (negated energy gradients) employed in an MD simulation may derive from quantum mechanical (QM) calculations, in which the electronic forces on the atomic nuclei are classically integrated to propagate nuclear coordinates. QM–MD calculations can accurately model electronic polarization and excitation effects, and the breaking of chemical bonds, but the expense of such calculations precludes their use for long-timescale calculations. Fortunately, life as we know it is limited to moderate temperatures and pressures where chemical reactions only occur at the catalytic sites of enzyme proteins and most processes consist of low-energy conformational transformations and molecular migrations. Therefore, biomolecular applications can generally substitute low-cost classical potential functions for quantum calculations, although hybrid QM–MM methods may be employed to model chemical reactions, modeling only the few atoms near the region via QM.

Classical MM potential functions employ additive terms for bonded and nonbonded interactions. Bonded terms represent the covalent bonding structure of the simulated molecules, with directly bonded atoms linked by simple springs of the form $U(r) = k\,(r - r_0)^2$ where r is the interatomic distance and the constants k and r_0 are parameterized for each pair of atoms based on QM calculations of either the actual molecule or appropriate analogues. This parameterization is greatly simplified by the fact that all proteins employ the same 20 amino acids, and likewise all DNA the same four bases, and hence a single set of parameters can be employed for all biomolecular simulations. New parameters need be developed only for small novel ligands such as drugs. A similar energy term $k(\theta - \theta_0)^2$ is applied to the angles between all pairs of bonds to a common atom, and also to planar sets of three bonds to a common atom. Finally, one or more torsion terms of the form $k(1 - \cos(n(\varphi - \varphi_0)))$ are applied to all dihedral angles between pairs of angles with an overlapping bond.

Nonbonded terms apply to all pairs of atoms in the simulation, including images from periodic boundary conditions, with the exception of pairs of atoms that are bonded to each other or to a

common atom because the interactions of these closely bonded pairs are fully represented by the bonded terms in the potential function. Nonbonded terms represent electrostatic forces between partial charges on the atoms (due to uneven distribution of the molecular electron cloud relative to the nuclear charge), van der Waals attractive interactions, and short-range repulsive interactions. Electrostatic interactions follow Coulomb's law $U(r) = Cq_1q_2/r$, whereas the van der Waals r^{-6} attractive potential is traditionally coupled with r^{-12} repulsion in the Lennard–Jones form $U(r) = 4\epsilon((\sigma/r)^{12} - (\sigma/r)^6)$. As with the bonded terms above, partial charges q and Lennard–Jones parameters ϵ and σ are fitted based on QM calculations. For simplicity and efficiency, several atom types are defined for each element present in the simulation to represent the variety of chemical bonding environments, and these types are used to assign bonded and Lennard–Jones parameters. Lennard–Jones parameters for pairs of atoms with different types are approximated by geometric averaging of the individual ϵ energy terms and (most commonly) algebraic averaging of the atomic radii σ. The formalism for deriving parameters specified by the force-field developer must be adhered to and combining parameters developed via different methods is not advised.

The simplified form of the potential functions necessarily limits the accuracy and generality of the simulation. Parameters are typically developed to reproduce structures at standard temperature and pressure. Water models, in particular, are designed as solvents to model proteins, lipids, and DNA, and to maintain correct structure at 1 atm and 300 K. Water is a highly complex fluid that maintains a network of hydrogen bonds both with hydrophilic solutes and between molecules of water itself. It is unreasonable to expect a simple three-atom water model (or even more complex five-point models employing additional charge centers) to serve the needs of biomolecular simulations while also correctly reproducing all properties of water, for example, the various phases of ice.

It is common for the great majority of atoms in a biomolecular simulation to be solvent (i.e., water and a few ions), in particular when a single protein or protein complex is being studied. The explicit representation of water molecules is then sometimes replaced with an implicit solvent model, in which more complex and expensive multibody interaction terms are used to approximate the averaged effect of the possible solvent configurations on the solvent. The solvent thus experiences a smooth mean-force potential that enables greater conformational exploration, useful in the study of large domain motions and protein folding. Implicit solvent cannot, however, reproduce specific interactions with small numbers of water molecules, such as inside channels. The reduction in atom count and resulting performance increase is also reduced or eliminated for larger simulations.

The nonbonded electrostatic and Lennard–Jones interactions are nominally $O(N^2)$, which would make large-scale simulations prohibitively expensive. This is addressed by imposing a short-range cutoff of 8–12 A on the Lennard–Jones terms, reducing them to $O(N)$, whereas the long-range $1/r$ electrostatic interactions are most commonly calculated in an efficient manner by the particle-mesh Ewald (PME) method for periodic simulations with explicit solvent. PME smoothly separates the Coulomb interaction into a $1/r$-like but exponentially decaying short-range potential that is obtained by direct pairwise calculation, and a long-range potential that is smooth and finite at short ranges, allowing its efficient representation via a 1A grid. Atomic charges are spread with $O(N)$ complexity onto small regions of the grid ($4 \times 4 \times 4$ to $8 \times 8 \times 8$), convolved with the long-range potential via $O(N \log N)$ 3D fast Fourier transform (FFTs), and atomic forces interpolated from the grid again with $O(N)$ complexity. Other methods such as fast multipole and multilevel summation can achieve $O(N)$ complexity for the full electrostatics calculation, but in practice the FFTs are a small fraction of the PME calculation and the grid can be coarsened to 2A by employing higher-order interpolation. The actual performance limitation of the PME FFT is the global communication required for its parallelization.

Simulations must be long enough to provide sufficient sampling to observe the molecular process of interest to the scientist. A standard all-atom MD simulation can be integrated stably with a 1 fs time step via the Verlet *leap-frog* integrator (equivalent to Velocity verlet). Constraining the lengths of all bonds to hydrogen atoms, and the H–O–H angle of water molecules, allows a 2 fs time step (or longer with new *geodesic* integration methods that calculate constraint forces at several subtime steps). In

addition, multiple time-stepping allows the slowly changing long-range electrostatics calculation to be done only every three steps. Beyond this point, longer time steps require coarse-grained models, in which groups of atoms are represented by larger particles with smoothed interactions that foreclose the study of many processes that rely on atomic detail.

A billion 1 fs time steps must be executed in sequence to achieve a microsecond of simulated time, reasonable by modern standards, but requiring 12 days even with excellent performance of a millisecond per step. Multiple independent simulations can and should be executed simultaneously with different initial velocities to increase sampling and statistical confidence, but other methods can be used to enhance sampling. The simplest enhanced sampling method is simulated annealing—raising the temperature of the simulation to increase barrier hopping followed by a slow cooling. More sophisticated methods such as accelerated MD directly lower barrier heights by scaling calculated forces when total system energies are above the thermally accessible level. When the exact mechanism under study is known but too slow to be observed during the simulation, time-dependent biases or steering forces can be used to induce the required transitions. Biased calculations can be used to efficiently extract the conformational free energy profile along a transition path. It is even possible to calculate the free energy difference between chemical states of the system by gradually creating, destroying, or transmuting atoms.

Multicopy algorithms enhance and direct sampling by loosely coupling otherwise independent simulations. The most common multicopy algorithms employ replica-exchange protocols, in which simulations along a range of temperatures or bias parameters periodically attempt to exchange configurations with their neighbors based on the Metropolis criterion, with overlapping bias windows analyzed to produce free energy profiles. Transition paths can be found via the string method, which tracks swarms of trajectories released along a candidate path in order to reach a lower energy path. Milestoning uses large numbers of independent trajectories to measure transition rates and first passage times between conformational hypersurface *milestones* in order to calculate both free energy profiles and transition rates. Finally, independent trajectories can be launched from known states to construct and extend Markov state models of biomolecular dynamics, including protein folding.

The types of scientific questions that can be asked and answered about biomolecular systems varies with the scale and complexity of the system. Alanine dipeptide has but two nontrivial degrees of freedom (rotatable dihedral angles) and can be sampled completely, thus making it a necessary but insufficient test for method development. Single protein systems, fully solvated, comprise around 100,000 atoms, and all but their slowest processes are studied with modern enhanced sampling methods. The ribosome, a complex aggregate of multiple proteins and nucleic acids responsible for protein synthesis in the cell, requires three million atoms to be simulated and remains a topic of cutting-edge research. Finally, the HIV capsid, at 64 million atoms, was assembled by MD simulations combining individual capsid protein crystal structures with cryoelectron microscopy density profiles, yielding an atomic-resolution capsid structure that is being simulated binding to both drugs and naturally occurring molecules in the cell to better understand the function of the capsid in the viral infection process. Simulations of the HIV capsid are run on NCSA Blue Waters, one of the most powerful supercomputers in the world.

5.3 ALGORITHMIC DETAILS

The basic design of NAMD's parallel algorithm dates to the early 1990s. The fact that it has survived this long, with only relatively modest changes to its basic parallel architecture, is due to the following factors:

1. The original design was done with a careful *isoefficiency analysis*, that is, analysis of asymptotic scalability.

2. Use of message driven execution to facilitate communication/computation overlap and enable flexible runtime scheduling.
3. Separation of concerns between what to do in parallel and where/when to execute computations, thus enabling powerful adaptive runtime system (RTS) support that could improve and change over the years in response to new architectural and application-related challenges without changing the basic structure of NAMD itself.

We explain these factors below.

Simply put, isoefficiency analysis deals with the question: for a given parallel algorithm (and, in the original formulation, characteristics of a parallel machine), at what rate must the problem size grow in order to maintain the same parallel efficiency when the number of processors is increased. Here, parallel efficiency is simply the speedup obtained divided by the number of processors. Suppose an algorithm is running on 10 processors with a speedup of 8. Running the same problem on 20 processors, the speedup should ideally be 16, but due to increased costs (typically, and for our purposes here, mainly, communication costs) it falls to a lower value. If you run a large size problem, for a properly designed algorithm, the communication-to-computation ratio *may* decrease and then it becomes possible to get a speedup of 16. Such algorithms are said to be scalable, and the rate at which the problem size must increase with respect to the number of processors is the isoefficiency of the algorithm. Thus, an algorithm with quadratic isoefficiency is one for which the problem size (defined as sequential execution time) must increase four times when the number of processors doubles to maintain the same parallel efficiency.

In the 1990s, many parallel MD programs were developed by converting an existing sequential program. Methods such as atom replication (where the array of all atoms was available on all processors) or simple static atom partitioning combined with force-decomposition [5] naturally arose in this scenario. However, our early analysis showed that these techniques were not scalable in the sense of isoefficiency analysis, that is, no amount of increase in problem size could bring the communication/computation ratio down, when increasing the number of processors. Along with a few peers [6], NAMD pioneered the idea of *spatial decomposition*, which was tedious but scalable. Atoms were decomposed into cubes, and they had to be migrated every K steps to the correct cubes based on their coordinates. Cut-off based electrostatic force calculation, the dominant part of the MD time step, becomes scalable with this technique.

The second technique was somewhat more radical at that time. Which processor should do the task of calculating forces between the atoms of two neighboring cubes? Prevailing processor-centric thinking would typically answer this by making the processor that housed one of the cubes do the computations. NAMD stipulated that the work will be done by another object, called the *compute* object denoted by a diamond in Figure 5.1, and that the placement of this object to a processor be left to the RTS. In particular, the placement could be (and often was) on a third processor which housed neither of the two cubes involved. This creates a degree of freedom for load balancing and allows utilization of a large number of processors (e.g., many more processors than the number of cubes). A similar idea, with more specific choices for the *third processor* has since been independently invented with names such as mid-point method, by Shaw et al., [7], Snir et al., [8], and Blue Matter team from IBM [9]. One can think of this technique as a hybrid between spatial decomposition and force decomposition.

Until the time of NAMD design, Charm++ (and Charm, its precursor) was an experimental programming model used mainly for divide-and-conquer and state-space search problems, which were representative of the problems addressed in the era of the so-called Fifth-Generation computing systems. Its main feature was the idea that the computation could be broken down into logical objects by the programmer but the assignment of those objects to processors was to be automated by a *RTS*. The related idea was that of message-driven execution: given that there are many objects on a processor, the object to be selected for execution should be the one for which a message (or method invocation) is already available. This allowed automatic, adaptive overlap of computation and communication (see the description in the next section).

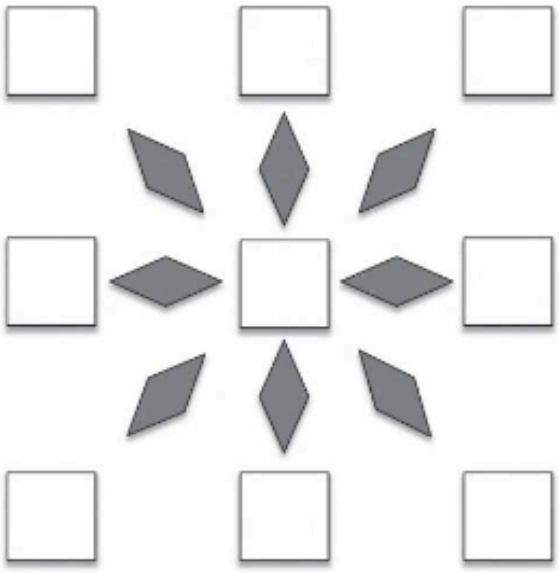

FIGURE 5.1 NAMD hybrid decomposition of patches (squares) and nonbonded computes (diamonds).

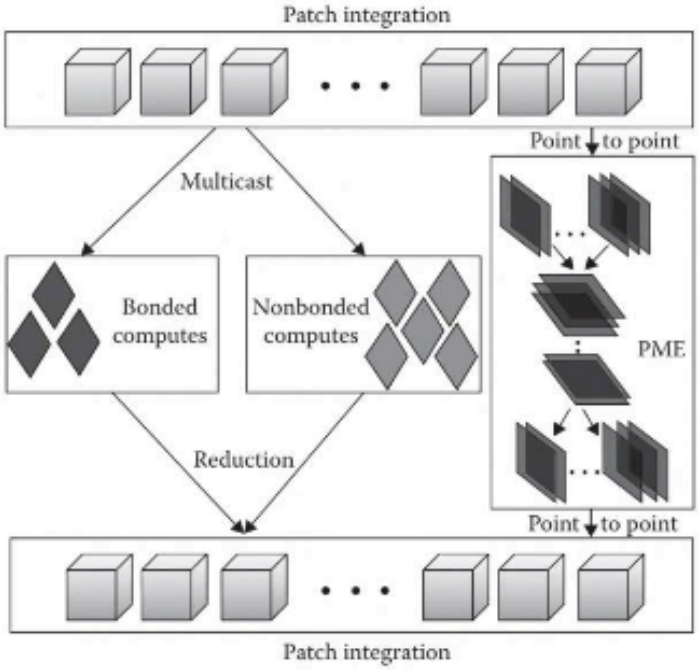

FIGURE 5.2 Compute flow of NAMD.

The above idea of hybrid decomposition might be novel in the context of a processor-centric programming model, but was natural in Charm++. The computation was logically decomposed into multiple collections of objects (see Figure 5.2). The PME summation, or the fast-multipole based algorithm for aperiodic systems, was used to calculate the long-range contributions of electrostatic forces. Because Charm++ was in the early stages of development, NAMD was initially implemented in Parallel Virtual Machine (PVM) as well as Charm++. However, the message-driven execution was a natural fit for the NAMD algorithm. For example, it could easily handle interleaving of PME messages, coordinate messages, and force messages, which is tricky to express modularly in PVM (or for that matter, in message passing interface [MPI]).

Thus, Charm++ and NAMD coevolved synergistically, in a process that we may call codesign today. Charm++ development was further helped by codevelopment of applications in astronomy and quantum chemistry as well as several engineering applications that arose at the rocket simulation center at Illinois.

5.4 PROGRAMMING APPROACH

NAMD is built upon the Charm++ parallel programming framework, which is a C++ based model with data-driven objects [2]. The Charm++ adaptive RTS has three main properties:

- **Overdecomposition:** This property consists in dividing the application data into objects, where the number of objects is typically much more than the number of processors. In Charm++ these are C++ objects called *chares*. Programmers make this division logically without having to worry about the underlying low-level components, that is, cores, and the RTS controls mapping of the objects to the processors. Overdecomposition carries many benefits; the logical division of data both increases programmer productivity and also application performance. It enables the overlap of computation and communication by increasing the likelihood that there are objects ready to execute in a processor while others are waiting for messages to be received over the network. Furthermore, division into small units may result in better cache performance as well. Charm++ also provides indexed collections of *chares,* which are called *chare arrays*. The programmer has the flexibility to create multiple types of *chares* and *chare arrays*.
- **Asynchronous message-driven execution:** This attribute means that execution of work units is driven by availability of messages, instead of a preprogrammed sequence. Messages are simply function calls on objects that can reside on local or remote processors. When the RTS detects the arrival of a message, it puts it on the message queue of the corresponding object and schedules it when possible. *Asynchronous* implies that the sender does not block the execution for any immediate response from the receiver side. The function calls, also called *entry* methods, look like standard C++ methods; the only addition is that the programmer specifies these functions in an interface file, called the *ci* file. Charm++ uses a simple translator to read the functions in the interface file and generate the necessary code in the background to be able to make these function calls execute on remote objects. The programmer does not need to worry about object locations, the RTS takes care of transferring the function call and the argument data to remote or local objects transparently.

Combined with the overdecomposition property, asynchronous execution can help reduce the communication related delays that can significantly hurt performance of applications. After the send operation, the sender can continue execution and does not have to wait for a response from the receiver. Also, the receiver can work on executing other messages as long as there are messages in its queue and does not need to be blocked waiting for a specific message. This scheme hides the latency of communication and can reduce the network pressure by spreading it over time.

- **Migratability:** This property is the ability to move objects from processor to processor. It allows the RTS to map and redistribute objects dynamically during execution to improve performance. For example, the RTS measures the load of the processors during application execution and moves objects around to keep the amount of work on every processor balanced. Charm++ provides a *packing* and *unpacking* (PUP) framework to enable migration of objects. The programmer specifies which data fields of the *chare* object need to

be migrated and the RTS handles serialization and deserialization. Dynamic migratability enables many features in Charm++:

- Dynamic load balancing
- Recovery from hard and soft failures
- Temperature and power aware work distribution
- Malleable job scheduling with shrink and expand number of processors

These are discussed later in more detail in Sections 5.4.1 and 5.7.

NAMD benefits from these properties of Charm++. Other features of Charm++ that improve performance of NAMD are described further in this section.

5.4.1 Performance and Scalability

Charm++ provides multiple methods to improve performance and scalability for both intranode and internode contexts. These methods are dynamic load balancing, TopoManager application programming interface (API) for topology-aware mapping, symmetric multi-processing (SMP)-specific mode, communication optimization methods such as prioritized messages and persistent communication, GPU Manager API for heterogeneous architectures, and Parallel I/O support.

5.4.1.1 Dynamic Load Balancing

The Charm++ runtime has a rich dynamic load-balancing framework that is based on *overdecomposition*. Each processing element (PE) typically has multiple chares that can be migrated from one PE to another to achieve better balance. The load balancing framework stands on the *principle of persistence*, which means that the load of objects in the past is often a good indicator of their load in the future. During execution, the runtime collects load information in terms of execution time of chares and the communication graph between them. This information is used to make load balancing decisions and decide where to migrate chares. Further, applications can add custom metrics to implement application-specific load balancing strategies to achieve better load balance.

NAMD has multiple different kinds of chares: patches, bonded computes, nonbonded computes, and three types of PME computes. All of these chares are instrumented in the runtime to collect their load information and the total load of each processor is calculated by summation of the object loads on that processor. Although all object loads are considered for load balancing, only nonbonded computes can migrate from one processor to another. This is partly because the nonbonded computes occupy a significant portion of the total execution time, and partly to minimize migrations and to simplify load balancing decision-making. The other object locations are statically assigned at startup.

The load balancing strategy described is centralized, requiring load and communication information of all objects to be collected on one processor. However, when running at large scale, this can create a serialization bottleneck and the data may become too large to fit in memory. To prevent this bottleneck, support for hierarchical load balancing was implemented. This way, processors are organized into a tree with customizable levels and number of children per tree node. First, load information is exchanged in a bottom up fashion and load balancing is done top down. At every level, each node and its children form a group where the previously explained refinement-based load balancing strategy is invoked within each group. This effectively restricts collection of information inside the group. Hierarchical load balancing has been shown to perform 100 times better than centralized load balancing in terms of the time it takes to load balance on petascale systems [10].

5.4.1.2 Topology Aware Mapping

Up to a few thousands of nodes, NAMD performs well on toroidal network topologies that are commonly found in supercomputing systems without worrying about the underlying network topology. However, scaling up to tens of thousands of nodes requires topology-aware placement of the work

units to avoid network contention. Charm++ provides an API, TopoManager that provides physical topology information to the application to be able to map the PEs to the underlying topology. Topology aware placement schemes can deal with noncontiguous allocations (e.g., a random subset of processors within a 3D torus, or a denser torus with *holes,* i.e., unavailable nodes, in the middle).

The *replicas* in multicopy NAMD algorithms [11] do not exchange much information except some control parameters and aggregate information such as the total energy of the simulation or the potential energy of an applied bias; the majority of the communication happens within the replicas, or partitions. However, the overall performance of the simulation is limited by the slowest replica, therefore, it is important to have the same simulation rates for the replicas. There can be performance variations between partitions for various reasons; data-dependent algorithmic reasons, or network/system noise. The network noise can be solved by having equal, uniform, and compact partitioning of the physical nodes as much as possible. NAMD uses a recursive bisection strategy to do topology-aware partition mapping and topology-aware placement of compute objects within each partition [4].

5.4.1.3 SMP Optimizations

Multicore compute nodes in parallel architectures have driven the need for a multithreaded SMP runtime mode of Charm++. Charm++ offers two different build modes: SMP and non-SMP. In the non-SMP mode, each Charm++ PE runs as an OS process. Each process can send and receive messages from other processes. In SMP mode, by contrast, PEs run as threads, and all threads that belong to the same OS process form a *logical node,* or a *Charm++ node.* Each logical node has a communication thread with its own dedicated core that is responsible for sending and receiving internode communication messages. Other threads, whose count per logical node is customizable, simply act as workers. This mode, where the PEs on the same node share a single memory address space, provides multiple benefits on multicore nodes. These are improved memory footprint, more efficient intranode communication, fine-grained parallelism, and reduced launch time.

- **Reduced memory footprint:** In SMP mode, Charm++ PEs on the same node can now share common read-only and write-once data structures. This not only reduces the memory footprint, but it also improves the cache performance as well, compared to the non-SMP mode. Optimizations are done in NAMD to share certain information such as objects that contains static physical attributes of atoms. This has shown 3–10 times reduction in the memory usage in NAMD and better cache performance that is directly attributed to reduction in the memory footprint [3,12].
- **Improved intra- and internode communication**: Usage of the shared address space among the PEs in the same node improves intranode performance. When sending messages within a node, the runtime only needs to send a pointer to the message, instead of making a copy of it as in the non-SMP mode. In addition to intranode benefits, this optimization provides better communication performance in internode communication as well, such as in broadcasts and multicasts within the Charm++ runtime and in NAMD's multicast operations [12]. In such cases, when a remote PE sends a message to multiple PEs on the same node, only one internode message is sufficient to be sent, then the message can be forwarded to the intranode PEs locally. This removes the need to send multiple expensive internode messages, replacing them with cheaper intranode communication.
- **Reduction in launch time:** Because the SMP mode creates fewer OS processes than the non-SMP mode, it uses less system resources and significantly reduces the launch time of the applications at large scales. For example; in the non-SMP mode, it takes 6 minutes to launch NAMD with *mpirun*, when running on 224,076 cores on Jaguar supercomputer

with a Charm++ build that uses MPI for communication. Whereas in the SMP mode with 1 process and 12 threads per physical node, *mpirun* takes around 1 minute to launch NAMD [3]. This effect is all the more important on cluster-like machines, such as Texas Advanced Computing Center (TACC)'s Stampede.

- **Enabling fine-grained parallelism:** During the PME phase of NAMD, it has been observed that whereas some cores are busy with computation, the neighbor cores are idle. To exploit more fine grained parallelism, the *CkLoop* library was developed in Charm++. CkLoop offers OpenMP-like shared memory multithreading where Charm++ PEs are reused to execute the spawned tasks, and it has been shown to improve performance [13]. Using OpenMP for this purpose with Charm++ on the same cores is not easy because the two runtimes are not aware of each other. However, we have an ongoing effort to realize this because it is more programmer friendly to use existing well-known libraries like OpenMP instead of a custom one to achieve the same task.

5.4.1.4 Optimizing Communication

Communication constitutes a major part of the PME phase in NAMD, therefore, optimizing PME is critical to achieve scaling to a large number of nodes. Prioritized messages and persistent communication are two ways to optimize PME communication.

- **Prioritized messages:** Charm++ runtime provides the ability to assign priority to messages. Commonly, applications send different types of messages because they have different phases or functions. Some of those messages might be on the critical path and their delay might affect overall performance. For example, in NAMD arrival of PME messages drives the FFT computation. Another type of message in NAMD triggers nonbonded computation when two messages from the patches arrive at the compute object. Because PME messages are more critical, we assign a higher priority to them. The runtime queues honor priorities and use first-in-first-out (FIFO) order among messages of identical priority.
- **Persistent communication:** In the default mode of Charm++, when messages arrive at the receiver, the corresponding memory is allocated and the message is placed there. The memory destination is not known beforehand. On the other hand, most scientific applications are iterative and have *persistent* communication patterns that repeat over the execution of the application. Exploiting this knowledge gives the opportunity to save the time to allocate and free memory on the receiver side, and utilize remote direct memory access (RDMA) more efficiently. Implementing PME with persistent messages, NAMD's performance improved by 10% when running a 100-million-atom simulation on the Titan machine at Oak Ridge National Laboratory (ORNL).

5.4.1.5 GPU Manager and Heterogeneous Load Balancing

Charm++ supports GPUs on a basic level via an extension called GPU Manager. GPU Manager handles the delegation and execution of CUDA kernels in the context of the asynchronous message-driven runtime of Charm++. GPU Manager operates by registering GPU kernels to be managed by the RTS. The runtime asynchronously invokes kernels when data is available on the device, automating the overlap of data movement and execution. GPU Manager also automatically copies data to and from the device before and after kernel execution.

To use GPU Manager, the user supplies an explicit CUDA kernel and denotes buffers to be moved to and from the device. A callback must also be registered with the runtime, which is called after the kernel has finished and data has been copied back. This step is necessary because the call to GPU Manager returns once the runtime has copied the CUDA buffers; it does not block until the

kernel has finished. GPU Manager coordinates data movement and kernel invocations through a FIFO queue. When a processor on the node is idle, the runtime invokes a function to issue new requests to the GPU.

Although GPU Manager allows GPUs to be utilized by the Charm++ runtime, it does not provide much control over execution. Namely, load balancing, a hallmark of Charm++, cannot be done because GPU load information is not used and there is no mechanism to move or retarget work. In order to solve this issue, Charm++ offers some experimental constructs on top of GPU Manager. The first of these is the Accel Framework [14], which can dynamically generate CUDA kernels from host code and dynamically decide where particular methods should be executed. The Accel Framework allows programmers to annotate methods with the "accel" keyword to indicate that a method can be targeted at multiple hardware platforms. Methods can contain multiple implementations for different hardware targets separated by keywords. The Accel Framework has a variety of strategies to determine where to execute particular entry methods. The strategy can be specified as a runtime argument. These strategies can specify a static division between the various available hardware targets (e.g., 30% of invocations go to CPU, 70% go to GPU) or use dynamic measurement techniques to automatically balance work. In order to maximize GPU utilization and avoid serialization, the Accel Framework tries to batch multiple device method calls into a single kernel launch. Object data used inside *accel* methods must be annotated to mark it for transfer in, out, or both. Additional annotations can specify the lifecycle of a variable, such as indicating that it should remain on the device for later use. After execution, a callback is called just as in GPU Manager. One deficiency of the Accel Framework is that load information is only used to balance work across the different hardware resources of a single node. It can run in a distributed job, but each node is considered separately.

Building upon within-node balancing capabilities of the Accel framework, we are experimenting with new strategies for heterogeneous load balancing across nodes. A *vector load balancer* adds additional fields to the load database used in regular Charm++ load balancers. These additional fields can be used to store different load metrics and data to inform load balancer decisions. In particular, it can be used to store both CPU and GPU load for every object in the application. When combined with multidimensional load balancing algorithms, both CPU and GPU load can be balanced simultaneously. As the diversity and speed of high-performance computing (HPC) machines increase, this will likely be a very important technique in load balancing because it allows more data to be used when making load balancing decisions.

NAMD utilizes GPUs without using the Charm++ extensions because GPU support was added to NAMD before it existed in Charm++. The GPU related Charm++ additions have been driven by the needs of NAMD, however. For example, NAMD performance analysis concluded that CPUs were sitting idle when GPU enabled methods were being used and that GPU load balance was a significant disparity for certain types of problems and jobs. When control of GPU work is ceded to the RTS, it should be able to address these issues and better prepare NAMD and other applications to scale to future molecular systems and future HPC systems.

5.4.1.6 Parallel I/O

Charm++ provides a library for parallel I/O called CkIO. CkIO abstracts the parallel file system away from the application I/O code. By doing so, it is able to agglomerate I/O operations and decouple application file reading and writing operations from system configuration. Additionally, it simplifies and optimizes I/O operations made from migratable objects. CkIO allows the number of involved processors, arrangement of involved processors, and file stripe parameters to be specified to customize agglomeration. These parameters are used to construct and distribute objects that perform I/O. One key feature of allowing this customization is that it allows developers to properly size and align I/O blocks without having to change the access pattern of the application. Using the correct alignment is critical on parallel file systems, such as Lustre, because operations that are not well aligned can dramatically increase network and disk interference and load.

Because CkIO is implemented in Charm++, it utilizes asynchrony and overdecomposition to improve the effective performance of I/O. For a properly structured application with sufficient work, these features of the Charm++ RTS can hide the latency of messages used to agglomerate work and perform I/O operations to disk. Although other parallel I/O systems can accomplish similar feats, it is notable that the standard structure for Charm++ applications should fulfill these criteria without any changes.

NAMD writes out the positions of the particles in its molecular system periodically over the course of the simulation, and increasing molecular system size and a number of processors were creating performance woes. To address these, NAMD uses parallel I/O routines with rudimentary agglomeration. However, these do not align writes to file system or storage hardware boundaries and the implementation is somewhat NAMD specific. Thus, CkIO was written to solve a real application-driven problem while also being generic enough to be used in other Charm++ applications.

5.4.2 Portability

NAMD development has been specifically geared toward addressing both high-end and low-end platforms, with and without NVIDIA GPUs or Intel Xeon Phi coprocessors, and with x86_64, POWER, ARM, or self-hosted Xeon Phi CPUs. We do not see any future alternative for programming NVIDIA GPUs that will be as well supported as CUDA, but OpenMP 4.0 SIMD directives provide at last a standard cross-platform means of reliably accessing vector instructions. These single instruction multiple data (SIMD) directives will be most critical on non-GPU-accelerated machines, which will predominantly have x86_64 and/or Xeon Phi processors, both targets of the Intel compiler. Hence, we first write OpenMP 4.0 SIMD kernels vectorizable by the Intel compiler, and only later address ARM and POWER performance for those kernels that are not offloaded to the GPU via CUDA. We use identical data structures between Xeon Phi CPUs and offload coprocessors, and also other CPUs and GPUs whenever reasonable, using modern C++ features to keep as much code in common as possible.

Charm++ provides NAMD portability across many architectures and compilers. Charm++'s portable network communication layer is optimized for each vendor's network API. Unlike many other task-based parallel programming models, or partitioned global address space (PGAS) languages, Charm++ implements native communication API's of the vendors such as uGNI for Cray [15], PAMI for BlueGene [16], IB-Verbs for Infiniband [17], and TCP, UDP for Ethernet. Charm++ also has an MPI-based communication layer; however, native communication layers usually outperform MPI. For example, with a Gemini-based Charm++, NAMD performs up to 54% better than MPI-based version due to fine grained communication optimizations [13]. We are also working on an Open Fabrics Interfaces (OFI)-based communication layer for OmniPath-based architectures [17].

Charm++ supports many compilers, including Intel, IBM XL, GNU, Clang C/C++, Cray, Portland Group, and Microsoft compilers. Some of the supported operating systems are Linux, Mac OS X, Microsoft Windows (native and Cygwin), AIX, and various compute-node kernels.

5.4.3 External Libraries

5.4.3.1 FFTW

NAMD uses FFT calculations to approximate the long-range forces over the 3D grid. For this purpose, NAMD uses the popular FFT library Fastest Fourier Transform in the West (FFTW) for serial 1D or 2D FFTs, which are aggregated into 3D FFTs via NAMD-specific communication code written in Charm++. The MPI-parallelized 3D FFTs provided by the FFTW library are not used in production NAMD simulations, although they have been used as an example of how MPI and Charm++, two distinct approaches for parallelization, can interoperate and be used together [18]. MPI and Charm++ can share both resources and the data. In this case, at every iteration of NAMD, while

Charm++ tasks calculate the short-range forces, MPI tasks simultaneously calculate the long-range forces. Data are also shared between Charm++ objects and the MPI ranks through a data transfer repository. When Charm++ objects produce and deposit their data into the repository, the repository triggers the execution of a parallel FFT in MPI. Use of FFTW compared with a custom Charm++ based implementation provides similar performance, however, it has many advantages in terms of productivity [18]. It reduces the total source lines of code (SLOC), code development and maintenance. Moreover, vendors may provide highly optimized implementations in certain architectures that can be utilized without additional effort.

5.4.3.2 Tcl

NAMD leverages the Tcl scripting language to parse its configuration file, providing a complete and powerful programming language. Tcl is a mature, portable, and stable language; the interfaces have not changed during the history of NAMD over the past 20 years. The Tcl interpreter is self-contained as linked to the NAMD binary and does not require the presence of any external libraries. This lack of dependencies has enabled the NAMD Tcl interpreter to "just work" without issue on generations of supercomputers. In addition, most NAMD users are familiar with Tcl from its use in the popular codeveloped visualization program Visual Molecular Dynamic (VMD) [19]. It is a simple and easy to use scripting language that is ideal for nonprogrammer NAMD users.

5.4.3.3 Python

In response to user requests, it is possible to build NAMD with an optional Python interpreter, which can be called from and call back to the main Tcl interpreter. Unlike Tcl, the Python interpreter requires an extensive library, which complicates distribution, installation, and execution. It is, however, the extensive Python library that advanced users and method developers find appealing. The initial application of Python was in the development of constant pH simulation in NAMD, but with a small amount of work the dependencies on the Python library were replaced with Tcl equivalents such that the Python interpreter is no longer required.

5.5 SOFTWARE PRACTICES

5.5.1 NAMD

NAMD's software development practices have been in place for two decades and are largely shared with the codeveloped visualization program VMD. NAMD development is done primarily by a few key technical staff, with contributions of new features from scientists both within the National Institutes of Health (NIH) Center and externally. In order to retain full continuity of the source code revision history, NAMD uses Git version control system and uses Gerrit for code review and testing in same way that Charm++ is hosted, which will be described more later in this section. Branches are not used. NAMD is maintained as production-quality software at all times, with source code and read-only repository access publicly available and recent commits tracked publicly online.

NAMD Linux and Linux-CUDA binaries are automatically built nightly and posted for public download, whereas semiautomated NAMD builds across all supported local and remote supercomputer platforms are launched manually after major commits and installed as *latest* binaries available to all users of the machines, ensuring that features are regularly exercised by the researchers of the NIH Center prior to final release. New NAMD features are developed in collaboration with driving projects that provide eager test users, and the initial input sets are preserved for regression testing during refactoring. All NAMD executions include extensive internal checksums and stability tests that raise fatal errors on unexpected conditions.

A complete NAMD User's Guide is maintained, documenting all nonexperimental simulation features. Instructions for building and running the software are distributed as release notes both online and with each binary and source code download. More detailed information on running Charm++ programs, such as NAMD, is linked from the Charm++ website (http://charm.cs.illinois.edu). The NIH Center has since 2003 taught 46 hands-on computational biophysics workshops at locations across the US and the world. The hands-on components of the workshops are selected from 40 tutorials that teach both basic and high-end NAMD and VMD skills. The tutorials are publicly available via the Center/NAMD website at all times (http://www.ks.uiuc.edu/Research/namd/).

Although we enable and encourage a growing NAMD *user* community, with many independent projects requiring zero interaction with the development team, prospective outside *developers* are strongly encouraged to contact the development team, either privately or via the public e-mail list, before beginning any significant alteration or extension. In most cases, the desired functionality is either already available, inadvisable for some reason, or readily implemented via the NAMD Tcl scripting interfaces (documented in both the User's Guide and the tutorial "User-Defined Forces in NAMD"). For the remaining cases, the development team is often able to point to specific small modifications necessary, or to suggest analogous features that may be used as an exemplar without needing to understand the full parallel execution of the program. The NAMD source code is maintained in a legible state, with multiple-word variable and function names and brief explanatory comments as necessary to clarify algorithms and warn of potential hazards, and online NAMD source code documentation is generated nightly. We have experienced many times that graduate students will ask for guidance on adding some capability and, when asked several weeks later, report that they were successful without further assistance.

The primary barrier for new NAMD developers is learning Charm++ message-driven programming, which is not widely taught as is MPI, but is equally intuitive for beginners and presents a smaller set of concepts necessary for advanced usage. Complete documentation is available, and the Charm++ source distribution includes many tests and examples. The Charm++ performance visualization tool *Projections* allows a complete and interactive view into the message-driven execution of a program and is used extensively for parallel performance tuning by NAMD developers. Further, the Parallel Programming Laboratory has since 2002 hosted an annual "Workshop on Charm++ and its Applications" with archived presentation slides and videos, as well as a hands-on Charm++ tutorial.

The main NAMD and VMD software components are distributed under a custom open source license agreement developed by the NIH Center in 1999. The license is designed to minimize restrictions on scientific users of the software while ensuring that the center is able to collect use and impact statistics to justify continued funding. Thus, the primary restriction in the license is against redistribution of more than 10% of the software source code, or binaries created from it. The licensee has an obligation to acknowledge use of the software and to cite a specific reference paper in resulting publications. The license specifically grants the user the right to create and distribute complementary works such as scripts and patches and to redistribute without restriction works with up to half of their noncomment source code derived from at most a tenth of the noncomment source code of the software packages. The later clause allows unrestricted reuse of functionality, such as the core NAMD computational kernels. In contrast, under the GPU Public License a single line of copied code subjects the entire work to the license.

NAMD and VMD are distributed by web download. Registration for a username and password is required to enable tracking of download statistics. Charm++ and NAMD releases often coincide to support new platforms or NAMD features, and because NAMD beta releases provide the best test of a new Charm++ release. Matching Charm++ source code is included with the NAMD source distribution. The primary determinants of NAMD releases are the availability of new features of value to the wider user base and the observed stability of the code for internal users. We aim for one major release per year, and several beta releases over multiple months precede the final stable release.

5.5.2 Charm++

Charm++ uses a variety of software development practices to facilitate development. Git is used for version control. Git provides an ability to easily make lightweight branches, work in a decentralized manner, and encourages quick, iterative development, all of which are helpful for allowing the team to provide experimental research and stable software simultaneously. The Charm++ team makes use of precommit code review, requiring every commit to be reviewed by hand and automatically tested before being integrated into the mainline repository. This prevents breaking changes and regressions from being committed to the repository and helps the developers stay abreast of what changes are being made to the codebase because multiple people must look at every change by necessity. When code review was first adopted, there were some concerns that it would unnecessarily slow down the development process and cause patches to stagnate in code review limbo. These worries were not unfounded, as the development velocity did decrease in some cases, but the benefits of code review have vastly outweighed any of the negatives. Bugs are caught early, code style is maintained, implementations are well designed, and so on. Gerrit is used for code review and code hosting (https://charm.cs.illinois.edu/gerrit). Gerrit was chosen because it is easy and free to self-host, it offers more robust code review scoring than most competitors, it integrates well with testing systems, and it is fairly customizable to suit particular needs.

Charm++ uses two different services for testing and validation. The first is a home grown nightly build system called autobuild. Autobuild tests the bleeding edge version of Charm++ every night on various hardware and software platforms. Some of these tests are run on local machines with different configuration options (e.g., Windows vs. Linux vs. Darwin, GCC vs. Clang vs. other compilers), whereas others are run on remote clusters and HPC systems that cannot be tested locally due to lack of hardware. Because Charm++ supports a wide variety of different PC, cluster, and HPC configurations, a system like this is necessary to ensure that changes work well on all supported platforms. Autobuild is implemented as a Cron job, series of shell scripts, and an e-mail and web interface to view results. The second testing system is Jenkins. Jenkins is a widely used automation and build server. The Charm++ testing system integrates Jenkins with Gerrit, triggering a new Jenkins test with every commit made in Gerrit. When the test completes, the results are automatically posted to Gerrit so that code reviewers can use the results in making their reviews. A successful Jenkins result is required before a commit can be merged in Gerrit. Testing every platform for every commit would take too long and cause too much computational work, so commit-triggered Jenkins tests are only run on the default Linux configuration of Charm++. This is sufficient to detect most issues, and manual code review can identify situations when automated testing does not fully test a change, which can be resolved by manual testing. Occasionally, breaking changes make it through this process, but nearly all of these are caught by autobuild the following night.

Charm++ uses a custom build script that wraps configure and make. The custom build script parses various user specified options to customize the build for a particular system, selecting things like the desired network layer, compiler, and optional Charm++ modules to use. Many of the options have an associated shell script and header that are read when the option is selected; these files define the specific environment variables and compiler or linker flags for the selected option. These files are undocumented and are not intended for modification, but they allow advanced users to easily modify the configuration and even add new options.

Charm++ is implemented in C++. Charm++ programs are usually written in C++, but Fortran bindings also exist, as well as interoperation with any language that supports C++ bindings. Charm++ defines some language structures on top of C++. The most significant of these is called Structured Dagger (SDAG). SDAG can be used to express control flow dealing with sending, receiving, and ordering remote messages in Charm++. Charm++ also defines new keywords to deal with its object model, the most significant of which are *chare,* which indicates a class definition for an object exposed to the RTS, and *entry,* which indicates a method that can be remotely invoked with a message sent from a chare.

Charm++ is freely available for noncommercial usage under the Charm++/Converse license. The code is distributed from the website of the University of Illinois Parallel Programming Laboratory (http://charm.cs.illinois.edu). Commercial usage is allowed under a different licensing structure, as well.

Community contributions are allowed to Charm++. In order to contribute, a contributor should register with the code review system described earlier and submit a patch there. From there, the normal review and testing procedure occurs, eventually merging the patch into the mainline repository. Most community contributions come from collaborators of the Parallel Programming Laboratory and are shepherded along by a Charm++ developer, who often takes care of the logistical details and assists the submitter with any issues that may arise.

5.6 BENCHMARKING RESULTS

In order to benchmark NAMD performance for large simulations on petascale platforms we have developed freely distributable synthetic benchmarks by replicating the 1.06 M atom STMV benchmark (satellite tobacco mosaic virus—the first full virus simulation performed with NAMD in 2006). Two benchmarks were formed by tiling the 216.832 A cubic cell: a $5 \times 2 \times 2$ system of 21 million atoms and a $7 \times 6 \times 5$ system of 224 million atoms. The benchmarks use rigid water and bonds to hydrogen to enable a 2 fs time step and use a 12 A cutoff distance with PME every three steps using a 2A grid and eighth-order interpolation. Pressure control with a relaxed reduction-broadcast barrier is enabled to avoid global synchronization.

Below we compare existing petascale architectures in the United States (Figure 5.3). We note that the largest machines, Blue Waters and Titan, date from 2013 and use a Cray toroidal network and AMD Opteron CPUs. Edison and Theta both employ Intel processors, Edison having two older 8-core processors per node, whereas Theta uses a single newly released 64-core Xeon Phi KNL processor with AVX-512 vector units.

A clear performance advantage is provided by the NCSA Kepler GPUS on Titan, outperforming even the latest CPU on a per-node basis. However, as the non-Knights Landing (KNL) systems use two-socket nodes (for Titan one CPU is replaced with a GPU), whereas KNL is single-socket-only,

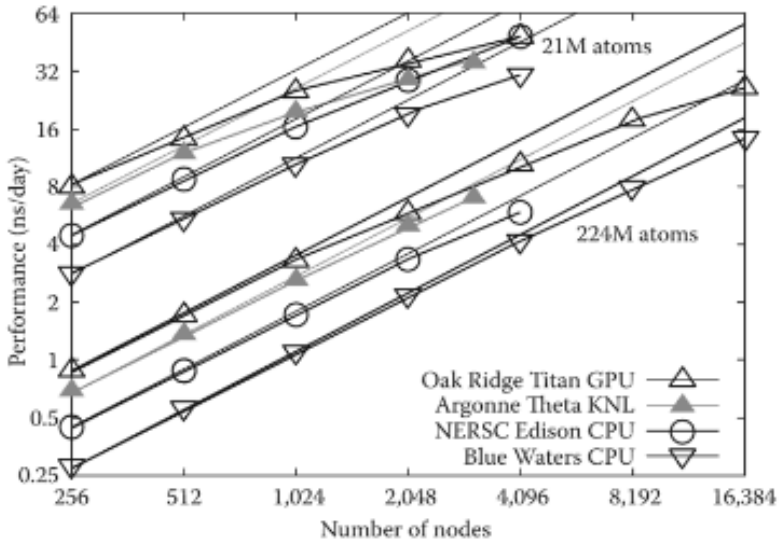

FIGURE 5.3 NAMD strong scaling performance on four systems with 21 and 224 million atoms up to 16 K nodes.

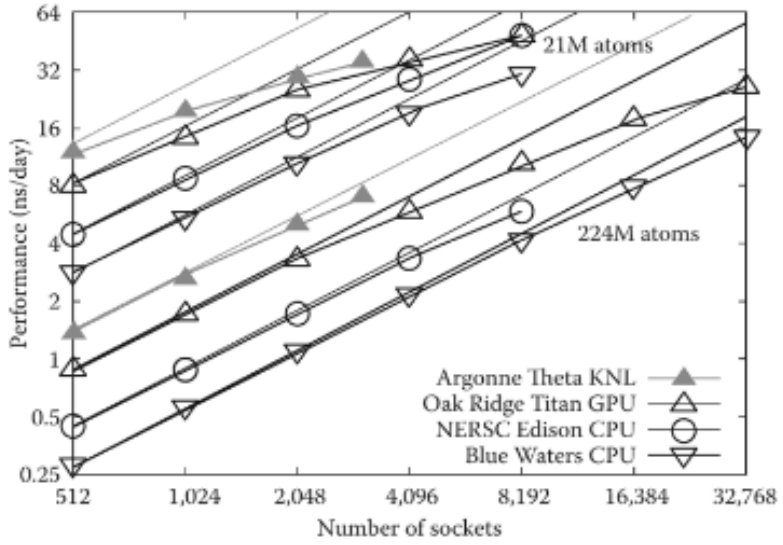

FIGURE 5.4 NAMD strong scaling performance on four systems with 21 and 224 million atoms up to 32 K sockets.

a fairer comparison may be on a per-socket basis (Figure 5.4), for which KNL is dominant (although over much older processors and GPUs).

A significant observation with regards to upcoming 200 petaflop and exaflop machines is that scaling is worse for systems using larger numbers of low-power cores (GPUs and Xeon Phi) than for traditional CPUs. In the case of Theta, this can be ascribed to serial bottlenecks, particularly in the Charm++ communication thread, even though seven processes are being used per-node. For GPU-accelerated Titan serial bottlenecks and communication are handled by the powerful CPU threads, and scalability is limited more by parallel decomposition and offloading inefficiencies in mapping work to the many cores of the GPU.

NAMD is part of the application readiness program for the Oak Ridge Summit machine based on IBM Power9 CPUs and NVIDIA Volta GPUs. We are able to make some performance predictions based on publicly available information. We assume that on the 3400 nodes of Summit, we will obtain a similar efficiency as we do currently on 4096 nodes of Titan, on which our 224 million atom benchmarks run at 9.0 ns/day, 19 ms/step. Scaling this performance by a factor of 4.5 to the full Titan machine, and then by another factor of five for the anticipated relative total performance of Summit, we arrive at an expected performance of roughly 200 ns/day, 0.9 ms/step. This corresponds to 150 16.4 A cubic domains of 440 atoms each per node, sending and receiving at each step positions and forces of 102 domains, a per-node network injection of 2.1 MB/step for short-range forces. PME on a 2 A grid would require $48 \times 48 \times 56$ grid points per node, plus transposes of $40 \times 40 \times 48$ grid points, yielding an additional 2.3 MB on PME steps, for a worst case of 4.4 MB/step. This is well within the 23 GB/s injection bandwidth of the internode network, which at 0.9 ms/step is 20.7 MB/step. Scaling smaller simulations across the full machine would require even higher injection rates and limited contention, so we are hesitant to predict beyond 200 ns/day for a multinode simulation. Multiple-copy algorithms with one or more replicas per node, however, should scale almost perfectly due to their limited interreplica communication.

Collective variables: Collective variables [20,21] (colvars in short) are reduced representations of degrees of freedom of the simulation, which can be either analyzed individually or manipulated in order to alter the dynamics in a controlled manner. Colvars can be sampled extensively to calculate certain statistical quantities accurately, unlike the far more numerous atomic positions. A single colvar can combine different functions of Cartesian coordinates, herein termed colvar components, which can be used to describe macroscopic phenomena.

Serial colvar calculations become a bottleneck as the complexity of user-defined variables increases (e.g., multiple root mean square displacement [RMSDs] [22]). The calculation of individual components of colvars are parallelized using CkLoop/OpenMP on multiple cores of a node, thus giving an even distribution of work across cores (on the same node) in the case of multiple colvars, each defined with many components. An individual component of a colvar is computed serially, thus the parallelization is beneficial only when there are multiple colvars or complex colvars which have multiple components. CkLoop parallelization of colvar in shared memory, as shown as SMP in Figure 5.5, improves colvar performance approximately by 2×.

5.6.1 Extrapolation to Exascale

5.6.1.1 Science Goals

Looking forward to the next generation of 200 petaflop machines, in particular Oak Ridge Summit and Argonne Aurora, we foresee two classes of simulations. First, more routine and a greater variety of simulations in the 200-million-atom range, which encompasses a variety of small organelles, membrane budding processes, and even complete viruses. Second, highly accurate and detailed multicopy simulations of millisecond-timescale processes in 300,000-atom systems such as complex membrane transporters. Future exaflop machines will enable basic simulations of multibillion-atom assemblies including models of recently experimentally constructed *minimal cells,* as well as enhanced sampling of the full range of smaller simulations. What is yet to be seen is whether future technologies enhance internode communication latencies sufficiently to enable the further reduction of time per step rather than only extending simulation scale and sampling (Figure 5.6).

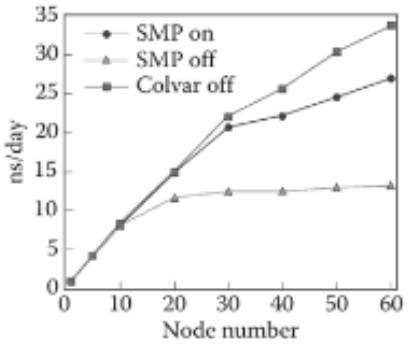

FIGURE 5.5 Colvar performance improvements.

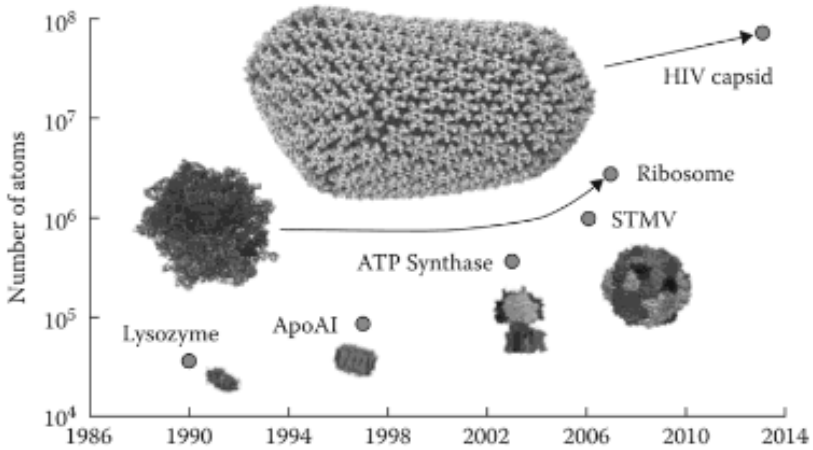

FIGURE 5.6 NAMD simulation sizes over the years.

5.6.1.2 Runtime System Enhancements Needed

What makes biomolecular simulation challenging to scale is the fact that scientists want to study relatively modest-size molecular systems but with huge number of time steps. Even a system with 100 million atoms, when running on tens of millions of cores, has only a few tens of classically represented atoms per core (with only about 24 bytes for each). Yet, because each time step simulates only a femtosecond, billions of time steps are needed for useful science. The challenge at exascale is to be able to complete a single time step in tens of microseconds! To continue to execute NAMD efficiently at exascale, the RTS of Charm++ will need to be improved in several essential ways.

5.6.1.3 Supporting Fine-Grain Computations

Grain size can be defined as the amount of computation per asynchronous (potentially remote) interaction. With current Charm++, average grain sizes as low as tens of microseconds can be handled without significant overhead. We need to reduce the overhead further in order to support the short time steps needed for NAMD. Other applications, including graph algorithms, are also driving this requirement. We have initiated a review of the Charm++ RTS to audit the overhead and to reduce it mainly by specializing implementations of individual chare collections so that only the necessary overheads are paid, and creating leaner representations of chare object handles.

5.6.1.4 Optimizations Related to Wide Nodes

One of the known characteristics of future machines is the large number of cores on each node. Several optimization challenges arise from such wide nodes, which we plan to address in the near future. Serialization due to locks is one such growing challenge, especially for many core chips such as Xeon Phi, but also on regular multicore chips. C11 atomics provide a potential way for avoiding or reducing serialization. One of the steps we are taking is to utilize atomics-based queues for within-node communication. For a widely used code, such as NAMD and Charm++, it is important to ensure that all machines and compilers support primitives that we use, which has made this transition challenging, in addition to the challenges arising from complex race conditions.

High bandwidth memory (HBM) is increasingly being used in next generation supercomputers, especially with accelerators such as general-purpose graphical processing units (GPGPUs) and Xeon Phi. Utilizing this effectively, via adaptive runtime support, is another planned objective. The ability to peak at the scheduler's queue gives the RTS an opportunity to effect prefetch and scheduling policies. At a lower level, scratchpad memories also provide potential for runtime optimization via prefetching.

Nonuniform memory access (NUMA) effects on large multicore chips present another challenge, which needs to be considered along with the issue of overload on communication threads. In the Charm++ RTS, in each process, a thread is typically dedicated to communication processing. Splitting a node's cores into multiple processes increases the ratio of communication threads to worker threads, reducing the potential bottleneck on communication threads. However, this leads to reduced opportunities for sharing resources such as HBM and caches. A potential solution we plan to explore is supporting multiple communication threads within a single process. The success and utility of this approach will depend on the ability of the hardware and the operating system to provide multiple concurrently usable ramps to the network FIFOs. Automatically controlling the number of SMT threads to use at runtime, based on runtime instrumentation, is another optimization we are considering in Charm++. Some of these optimizations will automatically benefit NAMD, whereas others will require some localized restructuring of NAMD code.

One insidious challenge we see to fine-grained performance is the increase in static and dynamic speed variability across cores within a chip, as well as across chips. We have shown the existence and magnitude of such variation in the presence of turbo-boost on supercomputers in use currently. As this variation increases in magnitude, stronger techniques that combine dynamic and static load balancing and reduce overhead without losing locality will be needed. We plan to develop such tech-

niques for use in Charm++ and NAMD, but also to develop flexible mechanisms (such as integration of OpenMP or similar constructs) so that runtime research from other groups, such as the SOL-LVE project, OMPSS project from Barcelona and several others can be used to benefit Charm++ applications.

5.7 RELIABILITY AND ENERGY-RELATED CONCERNS

In the face of power, energy, and reliability challenges in exascale architectures, Charm++ offers multiple solutions to address these concerns for its applications. These features are implemented within the runtime and require minimal code change to be enabled in the applications. As specific needs for these features arise for NAMD users, it will be modified to utilize these features.

5.7.1 Fault Tolerance

Charm++ provides various fault tolerance strategies to handle hard and soft errors: checkpoint/restart [23], message logging [24], proactive evacuation [25], and targeted protection for silent data corruption [26]. The checkpoint/restart mechanism is the most popular technique among them and is the one used in production Charm++ applications. Two schemes are provided:

1. On a shared file system with split execution where the application execution is interrupted and can be resumed later
2. With double local storage (in memory, local disk, or solid state drive) for online automatic fault tolerance

LeanMD is the mini-app version of NAMD and it is commonly used to prototype and test new features. Figure 5.7 shows in-memory checkpoint and restart time on a BlueGene/Q system with simulated failures with LeanMD application of two settings: 1.6- and 2.8-million-atom systems. The time it takes for both checkpoint and restart are in the few tens of milliseconds range. When the number of processors increases from 2 to 32 K, the checkpoint time for the 2.8-million-atom system decreases from 43 to 33 ms. During restart phase, there are several barriers to ensure consistency until full recovery of the crashed node. Because of these barriers, the restart time for the 2.8-million-atom system increases slightly from 66 to 139 ms when the number of processors is increased from 4 to 32 K.

With runtime support, failure detection and recovery is made automatic. Even though the application execution can take longer time to complete, the user does not need to be aware of the failure because the RTS will automatically put the application back on track and make forward

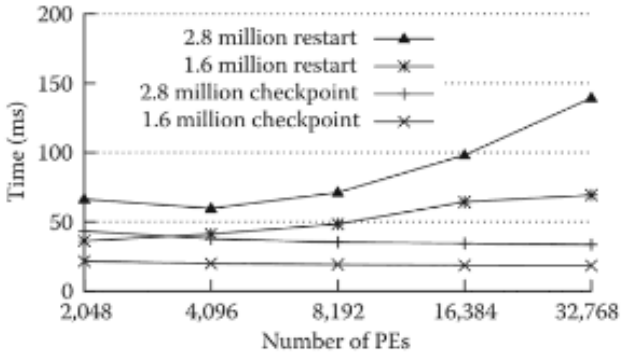

FIGURE 5.7 Checkpoint and restart time of LeanMD.

progress. NAMD uses its own disk-based checkpoint-restart mechanism. However, given this high-performance demonstration of online fault tolerance, we expect that NAMD will use the Charm++ mechanism in the future, especially when job schedulers start allowing applications to handle failures and continue in the presence of failures.

5.7.2 Energy, Power, and Variation

Not only the increasing power and energy consumption with scale, but also power and thermal variations among processors in the supercomputing platforms have become important concerns for HPC applications and data centers. Charm++ applications have a unique advantage: the object/task-based model and overdecomposition property enable applications to adapt to the variations in the hardware via dynamic load balancing and reconfigure the hardware resources transparently from the application while minimizing the performance overhead. Some of the strategies Charm++ offers are thermal-aware load balancing [27], variation-aware load balancing [28], power-aware job scheduling with malleable applications, and hardware reconfiguration [29,30].

5.7.2.1 Thermal-Aware Load Balancing

Thermal variations across components are common in large-scale data centers and may further increase in exascale. A thermal-aware load balancing strategy enables the runtime to remove the thermal hotspots among the processors by applying dynamic voltage and frequency scaling (DVFS) followed by load balancing. This technique can reduce the overhead of frequency reduction compared to naïve DVFS by 20%, while providing up to 57% cooling energy savings [27].

5.7.2.2 Speed-Aware Load Balancing

Future architectures may have heterogeneous components operating at different frequency levels. Even some of today's architectures having dynamic overclocking exhibit frequency variations across identical hardware components due to manufacturing-related reasons [28]. Charm++ offers speed-aware load balancing to obtain better performance under these variations. Any Charm++ load balancing strategies can be speed aware by taking into account the speed of the processors while moving the load from one to another. This way Charm++ can improve performance when the source of load imbalance is not the application but the hardware.

5.7.2.3 Power-Aware Job Scheduling with Malleable Applications

Future architectures may use power-aware resource managers where each application might have a limited power allocation. Charm++ applications can be malleable, that is, they can shrink or expand the number of processors that they are running on [31]. Malleability increases the data center utilization, reduces the mean response time, and achieve high job throughput under a strict power budget [32]. Demonstration of malleability with LeanMD application is shown in Figure 5.8.

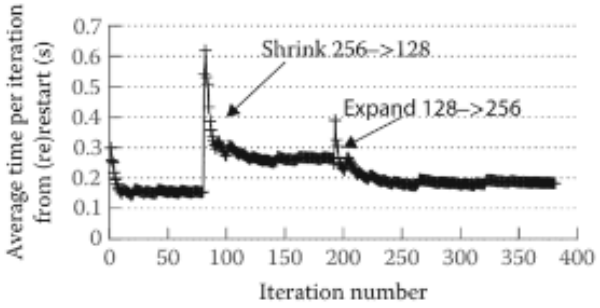

FIGURE 5.8 Runtime system (RTS) adapts the load as the number of processors shrinks and expand.

5.7.2.4 Hardware Reconfiguration

The RTS can adaptively turn on and off or reconfigure hardware components such as caches and network links to reduce power and save energy with minimal performance overhead [29,30]. Some applications, including NAMD, do not use the full caches. For example, with a configuration of 1000 atoms per processor and 80 B of data, per processor uses only 78 KB of memory, which is only a small fraction of a typical last level cache (LLC). Charm++'s runtime can predict the application's future usage and find the best cache hierarchy configuration. With this method 67% of the cache energy can be saved with only 2.4% performance degradation. A similar technique can be applied with network links if the application is not using the network links fully [30].

Many of the current HPC platforms do not give users the necessary rights to control frequency or power, or even to measure power and temperature data. This may change in future exascale architectures with the pressing needs for energy optimizations, and hence allowing the described optimizations to be put in production for applications including NAMD.

5.8 SUMMARY

NAMD is one of the best scaling, most used, and impactful parallel MD applications. Simulations of influenza virus (A/H1N1) and the HIV capsid are excellent demonstrations of NAMD's impact in the field of biomolecular simulations. NAMD is installed on many major supercomputers in the United States and supported on many architectures by means of its underlying robust Charm++ infrastructure. NAMD and Charm++ collaboration started years ago and has generated mutual benefits via its synergistic codesign process, illustrated in multiple features discussed in this paper. NAMD motivated implementation of new features, optimizations, and abstractions in Charm++, and these in turn have enabled implementation of new features and better performance for NAMD. NAMD was one of the first science applications using Charm++. Since then, many other Charm++ science/engineering applications have been developed, including OpenAtom [33], ChaNGA [34], and EpiSimdemics [35], with similar codesign and synergy. Implementation of the described features and optimizations within the runtime has not only benefited NAMD but also other Charm++ applications as well.

ACKNOWLEDGMENTS

First and foremost, we acknowledge the guidance and leadership of Prof. Klaus Schulten, who led the NIH center, the home of NAMD, from 1992 until his passing in late 2016. Many contributors to the development of the NAMD software have made this project a success, including: Bilge Acun, Ilya Balabin, Rafael Bernardi, Milind Bhandarkar, Abhinav Bhatele, Eric Bohm, Robert Brunner, Floris Buelens, Christophe Chipot, Jordi Cohen, Jeffrey Comer, Andrew Dalke, Surjit B. Dixit, Giacomo Fiorin, Peter Freddolino, Paul Grayson, Justin Gullingsrud, Attila Gürsoy, David Hardy, Chris Harrison, Jerome Henin, Bill Humphrey, David Hurwitz, Antti-Pekka Hynninen, Barry Isralewitz, Sergei Izrailev, Nikhil Jain, Neal Krawetz, Sameer Kumar, David Kunzman, Jonathan Lai, Chee Wai Lee, Charles Matthews, Ryan McGreevy, Chao Mei, Marcelo Melo, Esteban Meneses, Mark Nelson, Ferenc Ötvös, Jim Phillips, Brian Radak, Till Rudack, Osman Sarood, Ari Shinozaki, John Stone, Johan Strumpfer, Yanhua Sun, David Tanner, Kirby Vandivort, Krishnan Varadarajan, Yi Wang, David Wells, Gengbin Zheng, Fangqiang Zhu.

This work was supported in part by a grant from the National Institutes of Health NIH 9P41GM104601 "Center for Macromolecular Modeling and Bioinformatics." This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) 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 research used resources of the Argonne Leadership Computing Facility at

Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This research also used resources of the Oak Ridge Leadership Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the US Department of Energy under Contract No. DE-AC05-00OR22725. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575 (project allocations TG-ASC050039N and TG-ASC050040N).

REFERENCES

1. L. Kale and K. Schulten. Scalable molecular dynamics with NAMD. *J. Comput. Chem.*, 26, 1781–1802, 2003.
2. B. Acun et al. Parallel programming with migratable objects: Charm++ in practice. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, New Orleans, Louisana. IEEE, 2014, 647–658.
3. L. V. Kale and A. Bhatele, Eds. *Parallel Science and Engineering Applications: The Charm++ Approach*. Boca Raton, FL: CRC Press, 2011.
4. 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. *SC Conf. Proc.*, 2014, 81–91, 2012.
5. R. Das and J. Saltz. Parallelizing molecular dynamics codes using the parti software primitives. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, 1993, Norfolk, Virginia, March 22–24, 1993.
6. S. Plimpton. Fast parallel algorithms for shortrange molecular dynamics. *J. Comput. Phys.*, 117, 1–19, March 1995.
7. D. E. Shaw et al. Anton, a special-purpose machine for molecular dynamics simulation, *Commun. ACM*, 51, 91–97, July 2008.
8. G. S. Almasi et al. Demonstrating the scalability of a molecular dynamics application on a Petaflop computer. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*. New York, NY: ACM Press, 1998.
9. B. Fitch et al. Blue matter: Approaching the limits of concurrency for classical molecular dynamics. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, Tampa, FL, 2006, 44–44.
10. G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale. Periodic hierarchical load balancing for large supercomputers. *Int. J. High Perform. Comput. Appl.*, 25, 371–385, March 2011.
11. J. Comer, J. C. Phillips, K. Schulten, and C. Chipot. Multiple-replica strategies for free-energy calculations in NAMD: Multiple-walker adaptive biasing force and walker selection rules. *J. Chem. Theory Comput.*, 10, 5276–5285, December. 2014.
12. C. Mei et al. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis—SC '11*, Seattle, WA. IEEE, 2011, 1–11.
13. Y. Sun et al. Optimizing fine-grained communication in a biomolecular simulation application on Cray XK6. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*, Salt Lake City, Utah. IEEE, 2012.
14. D. Kunzman. Runtime support for object-based message-driven parallel applications on heterogeneous clusters. Department of Computer Science, University of Illinois, 2012. http://hdl.handle.net/2142/34256.
15. Y. Sun, G. Zheng, L. V. Kale, T. R. Jones, and R. Olson. A uGNI-based asynchronous message-driven runtime system for cray supercomputers with Gemini Interconnect. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, Shanghai, China. IEEE, 2012, 751–762.
16. S. Kumar, Y. Sun, and L. V. Kale. Acceleration of an asynchronous message driven programming paradigm on IBM Blue Gene/Q. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, Boston, MA. IEEE, 2013, 689–699.
17. Libfabric OpenFabrics. *Libfabric Programmer's Manual*. https://ofiwg.github.io/libfabric/.

18. N. Jain et al. Charm++ and MPI: Combining the best of both worlds. In *2015 IEEE International Parallel and Distributed Processing Symposium*, Hyderabad, India, 2015, 655–664.
19. Humphrey, W., Dalke, A., and Schulten, K. VMD: visual molecular dynamics. *Journal of molecular graphics*, 1996, 14.1, 33–38.
20. G. Fiorin, M. L. Klein, and J. Hénin. Using collective variables to drive molecular dynamics simulations. *Mol. Phys.*, 111, 3345–3362, December 2013.
21. NAMD User's Guide. Collective variable-based calculations. 2017. http://www.ks.uiuc.edu/Research/namd/2.12/ug/node53.html. Accessed: July 23, 2017.
22. NAMD User's Guide. Component rmsd: Root mean square displacement (RMSD) with respect to a reference structure. 2017. http://www.ks.uiuc.edu/Research/namd/2.9/ug/node55.html. Accessed: July 23, 2017.
23. L. V. Kale, G. Zheng, and X. Ni. A scalable double in-memory checkpoint and restart scheme towards exascale. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*, Boston, MA, 2012, 1–6.
24. J. Lifflander, E. Meneses, H. Menon, P. Miller, S. Krishnamoorthy, and L. V. Kale, Scalable replay with partial-order dependencies for message-logging fault tolerance, In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, Madrid, Spain, 2014, 19–28.
25. S. Chakravorty, C. L. Mendes, and L. V. Kalé. Proactive fault tolerance in MPI applications via task migration. *HiPC*, 4297, 485–496, Madrid, Spain, 2014, 19–28.
26. X. Ni and L. V. Kale. FlipBack: Automatic targeted protection against silent data corruption. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*, Salt Lake City, Utah. IEEE, 2016.
27. O. Sarood and L. V. Kale. A 'cool' load balancer for parallel applications. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis—SC '11*, Seattle, Washington: ACM, 2011.
28. B. Acun, P. Miller, and L. V. Kale. Variation among processors under turbo boost in HPC systems. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS'16, Istanbul, Turkey: ACM, 2016.
29. E. Totoni, J. Torrellas, and L. V. Kale. Using an adaptive HPC runtime system to reconfigure the cache hierarchy. In *SC'14: International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, Louisana. IEEE, 2014, 1047–1058.
30. E. Totoni, N. Jain, and L. V. Kale. Toward runtime power management of exascale networks by on/off control of links. In *2013 IEEE International Symposium on Parallel and Distributed Processing, Workshops and Phd Forum*, Cambridge, MA, 2013, 915–922.
31. A. Gupta, B. Acun, O. Sarood, and L. V. Kale, Towards realizing the potential of malleable jobs. In *2014 21st International Conference on High Performance Computing (HiPC)*, Dona Paula, India. IEEE, 2014, 1–10.
32. O. Sarood, A. Langer, A. Gupta, and L. Kale, Maximizing throughput of overprovisioned HPC data centers under a strict power budget. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, Louisana. IEEE, 2014.
33. N. Jain et al. OpenAtom: Scalable ab-initio molecular dynamics with diverse capabilities. In *High Performance Computing: 31st International Conference, ISC High Performance 2016*, Frankfurt, Germany, June 19–23, 2016, Proceedings. Vol. 9697. Springer, 2016.
34. H. Menon et al. Adaptive techniques for clustered N-body cosmological simulations. *Comput. Astrophys. Cosmol.*, 2, 1, March 2015.
35. M. V. Marathe, C. L. Barrett, K. R. Bisset, S. G. Eubank, and Xizhou Feng, EpiSimdemics: An efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing (SC '08)*, Austin, Texas. IEEE, 2008.