

High-Performance Molecular Dynamics Simulation for Biological and Materials Sciences: Challenges of Performance Portability

Ada Sedova

National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN,
sedovaaa@ornl.gov

John D. Eblen

Center for Molecular Biophysics
University of Tennessee
Knoxville, TN,
eblenj@ornl.gov

Reuben Budiardja

National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN
reubendb@ornl.gov

Arnold Tharrington

National Center for Computational Sciences
Oak Ridge National Laboratory
Oak Ridge, TN
arnoldt@ornl.gov

Jeremy C. Smith

Center for Molecular Biophysics,
University of Tennessee/Oak Ridge National Laboratory,
Oak Ridge, TN, and
Department of Biochemistry and Cellular and Molecular Biology
University of Tennessee, Knoxville, TN
smithjc@ornl.gov

Abstract—Highly-optimized parallel molecular dynamics programs have allowed researchers to achieve ground-breaking results in biological and materials sciences. This type of performance has come at the expense of portability: a significant effort is required for performance optimization on each new architecture. Using a metric that emphasizes speedup, we assess key accelerating programming components of four different best-performing molecular dynamics programs— GROMACS, NAMD, LAMMPS and CP2K— each having a particular scope of application, for contribution to performance and for portability. We use builds with and without these components, tested on HPC systems. We also analyze the code-bases to determine compliance with portability recommendations. We find that for all four programs, the contributions of the non-portable components to speed are essential to the programs’ performances; without them we see a reduction in time-to-solution of a magnitude that is insufferable to domain scientists. This characterizes the performance efficiency that must be approached for good performance portability on a programmatic level, suggesting solutions to this difficult problem, which should come from developers, industry and funding institutions, and possibly new research in programming languages.

- **Index Terms**—performance portability, molecular dynamics, heterogeneous architectures, SIMD instructions, CUDA-C

I. INTRODUCTION

Molecular dynamics (MD) is a ubiquitously-used computational tool for a number of disciplines, from biology and

biochemistry, to geochemistry, materials science and polymer physics [1], [2]. Many of MD’s algorithmic elements can be parallelized, and due to intense efforts from developers over several decades, certain MD programs have been highly successful in the high performance computing (HPC) setting [3]–[15]. However, as can be seen from the wealth of recent literature documenting porting efforts onto various HPC architectures and environments, significant amounts of development time and worker-hours were expended for each re-targeting of a particular application. These efforts are impressive and valuable, not only in the end-product they provide, but also in the knowledge gained in the porting process. However, it is also important to consider if it is possible to avoid such re-development expenses with each novel high-performance computing (HPC) hardware solution and to create optimally-performing scientific HPC applications such as MD that are more portable. Performance portability has become an important consideration in scientific computing [16], [17], and has recently been the subject of initiatives from both the U.S. Department of Energy (DOE) [16]–[18] and the National Science Foundation (NSF) [19]. If the code-base of large software applications has to be repeatedly, significantly altered for each new HPC architecture with low-level, machine-specific programming elements, an unproductive development model results. This is not simply because of the extra expenses in terms of worker-hours, but also, because the resulting code becomes more error-prone, due to shortened lifetimes, multiple authors, and therefore, fewer chances to debug, test, and improve the code over time [20]–[24]. This concern is often shared by commercial software companies.

Domain scientists may not have this background in software design; in addition, the goals of academic research and

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (<http://energy.gov/downloads/doe-public-access-plan>).

its needs for maximal performance when faced with tight, unstable budgets may not match that of a large commercial software business, and therefore design and work-management decisions diverge from those of a commercial project. Many algorithms used in scientific programs, which are required, to a certain extent, by the theoretical description, frequently do not parallelize as easily as some of the benchmark algorithms which are the focus of vendor-supported, optimized scientific libraries [25]. Thus domain scientists may often have to write application-specific code themselves. With these challenges in mind, we analyze the contributions to performance, and the achievement of performance portability, for four top-performing MD programs which are used for different scientific applications, using recommendations from the computer science field, and a metric that emphasizes the importance of time-to-solution. In each case, to provide a large dataset we have tested the programs on Oak Ridge Leadership Computing (OLCF) and National Energy Research Scientific Computing Center (NERSC) resources and have also supplemented these tests with some published reports from the developers. We aim to provide both an assessment, insights, and a foundation for future solutions to this difficult problem in the context of scientific computing.

II. BACKGROUND

A. Performance Portability

An application can be said to be portable if the cost of porting is less than the cost of re-writing [20]–[22]. Here, costs can include development time and personnel compensations, as well as error production, reductions in efficiency or functionality, and even less tangible costs such as worker stress or loss of resources for other projects. To quantify portability, an index has been proposed, the degree of portability (DP):

$$DP = 1 - (C_P/C_R) \quad (1)$$

where C_P is the cost to port and C_R is the cost to rewrite the program [20]. Thus, a completely portable application has an index of one, and a positive index indicates that porting is more profitable. There are several types of portability; binary portability is the ability of the compiled code to run on a different machine, and source portability is the ability of the source code to be compiled on a different machine and then executed [20]–[22].

Performance portability (PP), a type of portability wherein the application is not only source portable to a number of computing architectures, but its performance remains within an acceptable range for these highly efficient programs, has been discussed since the 1990s [26], [27]. The focus of these and later studies has varied from the ability to change and tune the algorithms themselves for specific architectures, to the use of a certain APIs or even languages that can provide a more performance portable program [28], [29]. In the HPC context, a performance-portable scientific application could be described as one being *source-portable* to a variety of HPC architectures using the Linux operating system and commonly-provided compilers, while *its performance remains within an*

acceptable range so that the program is used by domain scientists while performing competitive research in their fields. To avoid the ambiguity in the phrase “acceptable range,” Pennycook proposed the following metric for PP [28], [30]:

$$PP(a, p, H) = \begin{cases} \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a, p)}} & \text{if } a \text{ is supported } \forall i \in H \\ 0 & \text{otherwise,} \end{cases} \quad (2)$$

where $|H|$ is the cardinality of the set H of all systems used to test the application a , p are the parameters used in a , and e_i is the efficiency of the application on each system $i \in H$. Efficiency, here, is the ratio of performance of the given application to either the best-observed performance (BOP), or the peak theoretical hardware performance [30].

B. Molecular Dynamics

We focus on molecular dynamics within the Born-Oppenheimer (BO) regime: motion occurs according to classical mechanics, while forces may be calculated using any type of physics. A system, represented by atomistic units, is acted on by these forces and propagated in time using numerical integration of Newton’s equations of motion. Therefore, the time-per-step is a highly important quantity: the simulation cannot proceed with the next step until the last one is completed. A very small time-step is required (1-2 femtoseconds) for keeping the simulation from sustaining unacceptable drifts in energy [1], [31]. For adequate statistics, correlation functions used to estimate transitions often require times much longer than the de-correlation times of the physical property analyzed, [32], [33]; in addition, for analyzing relative stabilities of macromolecular conformations, a sufficient time must be simulated for the associated Boltzmann statistical distributions to be adequately sampled. The timescales required for such convergence can vary from picoseconds to seconds or above [34], while classical MD has only recently begun to approach the millisecond timescale using HPC programs. As long timescales and large systems are simulated, errors in the forces are being found which have been previously undiscovered. For instance, the adoption of highly stable but unphysical conformations over the course of several microseconds [35], and inaccurate dynamical properties unless a large amount of bulk water is modeled explicitly rather than with periodic images provided by the particle mesh Ewald (PME) method [36].

We analyze *open-source, highly parallel, high-performance software that can utilize over a thousand nodes of an HPC platform to achieve state-of-the-art results in their respective areas.* The four programs studied here each fulfill a specific function within MD, and are among the highest performing and most used codes in their particular areas. Fig. 1 illustrates the system size differences, and the differences in application, for NAMD, GROMACS, LAMMPS, and CP2K. NAMD is a program that is used for biomolecular simulation of large solution-state biomolecular systems (> 6 M atoms) for up to

1 microsecond [37], [38]. GROMACS is optimal for biomolecular simulation of medium-sized (100K-6 M atom) solution-state biomolecular systems for multiple microseconds [12]. LAMMPS [39], [40] is a molecular mechanics (MM) engine that allows the application of numerous force field types. We also examine ab initio molecular dynamics (AIMD) in the BO regime, which involves the calculation of the forces on each atom using quantum mechanics, usually a self-consistent field (SCF) calculation with density functional theory (DFT) [15]. For each time step, an iterative SCF calculation has to converge before the forces can be calculated, therefore, the program involves two layers of necessarily serial algorithmic components. Furthermore, because the ab initio forces are based on a calculated electronic structure, it is the number of electrons that determines the system size, and not just the number of atoms. Here we examine CP2K [15], a BO-AIMD that can simulate thousands of atoms for tens of picoseconds, using over a thousand nodes, with the linear-scaling SCF algorithm (LS-SCF) [14].

III. REQUIRED PERFORMANCE FOR THE FOUR MD PROGRAMS

A. Classical MD performance

MD developers have continuously pushed for increasingly shorter per-time-step execution rates. Currently, GROMACS [12] and NAMD [13] exhibit highly competitive timings per time-step. GROMACS simulation of systems under 2 million atoms are exceptionally fast, averaging about 1-2 ms per time-step and [8] allowing for up to 200 ns/day [11], [49]. The simulation of one million atoms at 200 ns/day is an impressive milestone in molecular dynamics simulations, and would have been an unfathomable goal for most researchers in past decades. NAMD uses the dedicated parallel programming library Charm++ [37], which focuses on medium-grained task distribution, and scales exceptionally well for very large systems. However, for systems under approximately 2 M atoms, the performance lags behind GROMACS, clocking about 11 ms/time-step for a 1 M atom system using 128 nodes, at which point the performance increase seems to level off. For 21 M atoms, NAMD attained about 5 ms/time-step using 4096 nodes, and for a 224 M atom system, about 40 ms/time-step using the same number of nodes, for version 2.11 [50]. This translates to about 20-40 ns/day for these large systems. This benchmarking information can be difficult to compare between one report and another, when reported in ns/day (or days/ns), as the time-step taken can vary from 1-5 fs. In addition, the type of ensemble used (NVE, NVT, or NPT) affects the timing, as do the details of the PME grid-size, and whether it is employed every time-step. Some recent publications using of these programs include, for GROMACS, a simulation of a bacterial membrane model with $\sim 200,000$ atoms that was simulated for 1 μ s [51], and a ~ 24 M atom biomass model of lignin, cellulose and associated enzymes for 1.3 μ s [52], and for NAMD, a 1.1 μ s simulation of a 6 million atom HBV viral capsid [53] and an HIV capsid consisting of 64 million atoms for 1.2 μ s [54].

B. Classical MM Performance

LAMMPS performance can vary greatly depending upon the type of force field used, whether the system uses atoms or other types of atomistic units such as coarse-grained particles, and if there are additional aspects evaluated such as bond breaking and formation. For this reason, the program's benchmark website includes a large number of very different systems with very different times-to-solution [55]. Performance is reported in million atom steps/sec, which is closer to the more preferable and unambiguous metric of ms(or sec)/timestep. For a one billion atom Lennard-Jones system, timings of about 0.16 and 0.25 sec/timestep have been reported using thousands of cores on either IBM Blue Gene/L at Lawrence Livermore National Lab or Red Storm at Sandia National Lab (a Cray XT3), respectively [55]. Examples of work published by LAMMPS include a coarse-grained biological-membrane force field using a Lennard-Jones potential was used to model the changing shape of a red blood cell [56] for 1500 μ s, a simulation of collision-induced absorption of a gas mixture of argon and xenon [57], and a simulation of a radiation-induced cascade in a cubic polymorph of silicon carbide [58].

C. AIMD performance

Currently, using hundreds of nodes of a supercomputer, CP2K is able to simulate systems of 100-900 atoms for 50-80 ps [14], [59]–[63]. The LS-SCF uses an approximation based on the locality of the dominant forces applied at the level of the density matrix [14], [64] allowing for simulation of thousands of atoms: the energy of over 6,000 atoms can be calculated in under a minute [65], and AIMD can be performed at production-level for these system sizes with similar timings [59]. Currently, systems of a thousand or more atoms can be simulated at this level for a sufficient amount of time to compare results with experiments: researchers have recently been able to calculate important phenomena with this method [60]–[62], [66], [67].

IV. PERFORMANCE IN MD: THE IMPORTANCE OF SPEEDUP AND TIME-TO-SOLUTION

Source portability has been achieved by all four programs for most currently existing leadership-class supercomputing resources. All four codes use standard languages, either C/C++ or FORTRAN, standard APIs, and mostly standard libraries. In addition, each program is able to achieve high performance on a number of different HPC platforms, delivering the BOP in its subfield. Because of this, these programs' Pennycook scores would be 100%, however they would have achieved this score not by being portable, but by means of arduous porting to each platform. Thus when a new platform appears, there is no guarantee that this same level of performance would be quickly attained on it. In order to estimate the performance portability of these BOP programs on some new platform, we analyze their PP in terms of their programmatic components, using a metric similar to (2).

A metric for the performance portability of these four BOP MD programs with respect to the set of their main accelerating

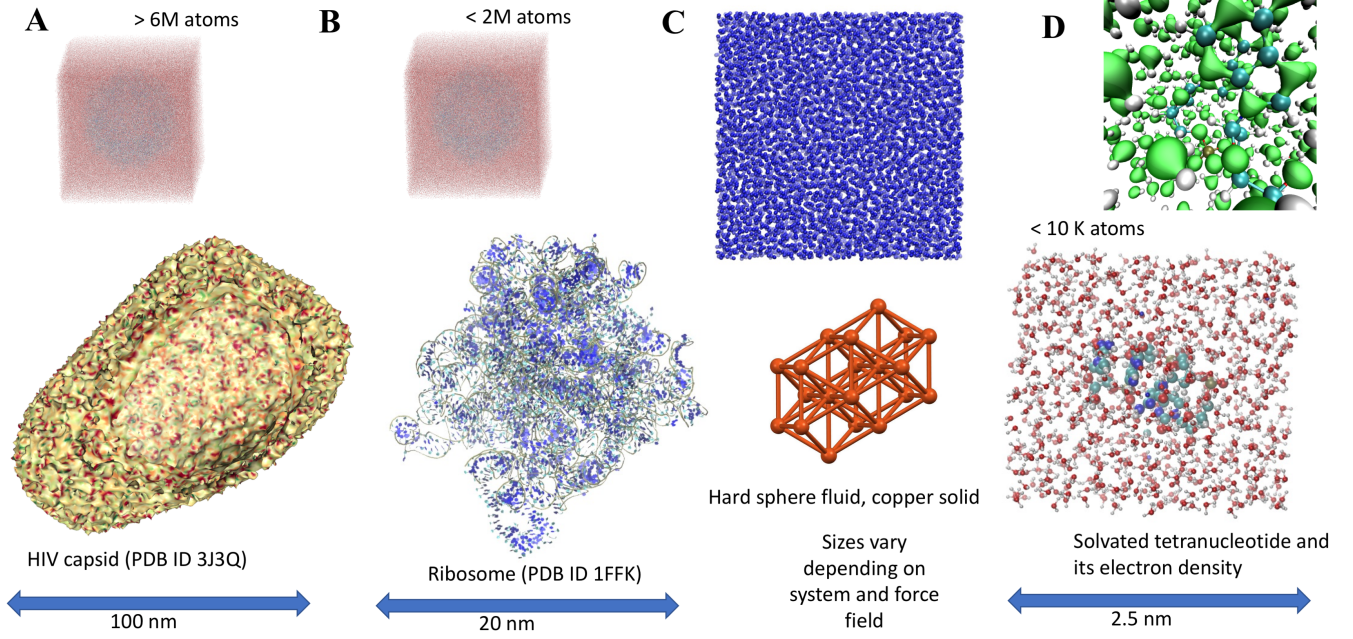


Fig. 1. Differences in system sizes and types for the four programs studied. A: NAMD, maximal performance for large systems, such as the HIV capsid, which was 64 M atoms when solvated. Image created using the NGL viewer interface [41] on the RCSB Protein Data Bank using PDB ID 3J3Q, [42]. B: GROMACS, maximal performance for systems under 2 M atoms, such as a ribosomal subunit. Image used PDB ID 1FFK [43]. A and B used the RCSB Protein Data Bank [44], [45]. C: LAMMPS, system sizes can vary from thousands to billions of atoms depending on the theoretical description (force field) used, such as a small solid copper crystal or a large box Lennard-Jones fluid. Copper structure, [46]. D: CP2K, forces on atoms calculated using a quantum mechanical description of the electrons, and a classical description of the nuclei. Maximal performance for systems with less than 10,000 atoms. All images used VMD, [47], [48].

programming components, or PP_{MD_c} , will be defined as follows:

$$PP_{MD_c}(a, p, Q) = \begin{cases} \frac{|Q|}{\sum_{i \in Q} S_i(a, p)}, & \text{if } |G| - |Q| \neq 0 \\ & \text{and } |G| - |Q| \neq |G| \\ 1, & \text{if } |G| - |Q| = |G| \\ 0 & \text{if } |G| - |Q| = 0, \end{cases} \quad (3)$$

where $|G|$ is the cardinality of the set G of algorithmic or programmatic components that significantly accelerate a best-performing application a , $|Q|$ is the cardinality of the set Q of all non-portable such accelerating components of a , p are the parameters used in a . S is the speedup that this non-portable component i of the application provides to the program: $S_i = T_0/T_i$, with T_i = the time-to-solution with the use of i , and T_0 the time-to-solution without i .

We define elements of G in a coarse-grained way, as one of the following acceleration levels: instruction-level (device or machine), thread-level, process-level, or library-level. At this time we assume the components do not influence each other and can be measured separately (which is indeed true for the components studied here). If a large majority of the performance of a program comes only from portable elements, each term in the sum in the denominator of (the first case of) (3) will be close to 1, and the performance portability PP_{MD_c} will be close to 100%, which is desirable. We see

the source portability requirement as essential to any further analysis of performance portability, and thus we set PP_{MD_c} if $|G| - |Q| = 0$ to zero in case three. If the values of PP_{MD_c} for a given BOP are not close to 100%, this indicates that a large number of accelerating components would have to be replaced with portable alternatives providing comparable speedup in order to achieve a performance-portable version of this application for future platforms. Careful analysis of each component can help fuel the development of portable tools which can deliver required acceleration at each of these levels. The PP_{MD_c} score does not depend on a concept of a theoretical peak of some type, which may be difficult to estimate.

Addition of a large number of components Q which only minimally accelerated performance would inflate the PP_{MD_c} score; however, there are a limited number of possible choices for all but the library level, thus $|Q|$ is bounded by a low number for all but libraries. We consider libraries as those elements not developed by the program developers themselves, so this increased score would not hide a large amount of extra worker hours. The definition in (3) does not yet include a way to use results from multiple machines. In this initial analysis, we decided to average the value of S_i obtained for each machine before entering this into (3) for each component i , to give the values for Final PP_{MD_c} listed in the tables below. For the analysis we perform in the following sections, the simulation details such as time-step size and ensemble,

mentioned above, will not matter as we only compare speedups in performance within a given program, for identical parameters.

V. EVALUATION OF PP_{MD_c} AND RESULTS

The following non-portable components help to achieve optimal performance: CUDA-C, architecture-specific SIMD intrinsic functions, interfaces with low-level communication APIs, and vendor-specific libraries. Sections of code written in CUDA-C, currently only compatible with NVIDIA GPUs, are not considered portable components: all sections of code currently using CUDA-C would have to be rewritten for a different GPU with competitive performance. For optimal performance on many-core, multicore, and the CPU portion of heterogeneous architectures, architecture-specific SIMD instructions, whether using intrinsic functions or vector instructions, are often required. In fact, without the use of SIMD, a majority of the processor’s capacity may be unused by a program, and compilers are not highly effective at auto-vectorizing code at compile time, due to uncertainty about whether various parallelization is allowable [68]. Highly tested and optimized SIMD instructions can take a long time to perfect and are architecture specific, thus they are also non-portable components. The use of standard high-performance libraries is often recommended as a performance portability solution [69], but can be problematic. While these libraries can provide a portable increase in performance without the need for rewriting by the domain scientist, a particular vendor may provide a version that is superior to those provided by other vendors for their particular machines, and thus performance may be greatly decreased on these other architectures.

A. NAMD

NAMD was allowed early access on OLCF Summit [70] as a part of the Center for Application Readiness (CAAR) program [71], so the program has been tested on Summit more than the other applications we studied. Thus we were able to assess a portion of PP_{MD_c} on this machine. Summit uses IBM Power System AC922 nodes, each containing two IBM POWER9 processors and 6 NVIDIA Volta V100 GPUs [72]. NAMD uses the Charm++ parallel framework for its parallelization [37], which is able to utilize MPI, or, for more efficient performance, the native lower-level communication API (LLCA) [37], [73], which is system dependent and therefore this interface with Charm++ is not portable. The LLCA for Summit is IBM’s Parallel Active Messaging Interface (PAMI) [74].

We built and tested NAMD on Summit with and without PAMI with Charm++. We used the GPU for these tests as well. To estimate the speedup with and without CUDA, new versions of which are currently under active development, we supplemented with an average over recently published reports on Titan, Blue Waters, and Stampede [50], [75], [76]. This number may actually be greater once new GPU work is completed; in fact, NAMD plans to use the GPU for the entirety of the calculation in future versions [77].

We used two standard NAMD benchmarks, the 21 M atom satellite tobacco mosaic virus (STMV) system [50], [76], [78], [79], with 40 nodes, and the 224 M atom STMV system with 400 nodes, with an MPI build, and with a PAMI build. Both builds used the latest version available as of June 2018, and GPU-offloading components that are currently being used in the Summit testing process [77]. Jobs used the following jsrun parameters: resources per host, 1; CPUs per resource, 42; GPUs per resource, 6; and the following NAMD run configuration: +ppn 41 +pemap 4-83:4,88-171:4 +commap 0. Both systems used a 2 fs timestep, and long-range electrostatics using PME every 3 steps, facilitated by the Verlet I r-RESPA multiple-time-stepping scheme, and using rigid bonds on all hydrogen atoms. Figure 2 illustrates the results, each averaged over 6 runs, for both systems tested, for MPI and PAMI versions. For both systems, use of the direct LLCA interface instead of MPI provided a speedup of $1.9 \times$. Table I displays the performance portability component scores and the final PP_{MD_c} score of 43%.

TABLE I
NAMD PP_{MD_c} SCORE COMPONENTS

Non-portable component	Speedup provided ($\times$)
CUDA-C ^a	2.7
Charm++ LLCA	1.9
Final $PP_{MD_c} = 43\%$	

^a Averages from published NAMD benchmarks, [50], [75]

B. GROMACS

We built and tested GROMACS version 2016.1 with a 1.1 M atom system on Titan, an OLCF Cray XK7 with AMD Interlagos CPUs, K20X NVIDIA GPUs, and a Gemini interconnect, with and without the GPU extension, and with and without the SIMD layer. GROMACS is written in C/C++, and uses MPI and OpenMP, as well as a heavy use of SIMD instructions via intrinsic functions. The SIMD “layer” is an interface used in the code, with a re-targetable back end; however this re-targeting is performed by the GROMACS developers. GROMACS also uses CUDA-C for its GPU port. We also tested the effects of the SIMD layer using one node of the NERSC Cori Cray XC40 Xeon Phi using a 134,177 atom system, with GROMACS version 2018.2, built with and without SIMD. The Titan version was built with gcc version 5.3.0, and the Cori version was built with the Intel compiler version,

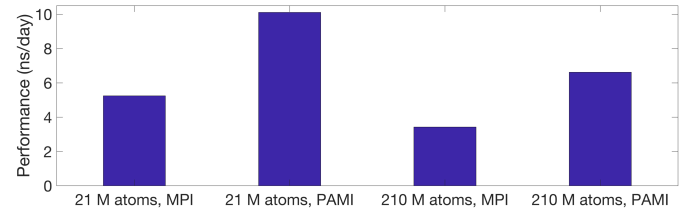


Fig. 2. Performance of NAMD with and without the use of the PAMI LLCA on OLCF Summit, for a 21 M atom system and a 210 M atom system

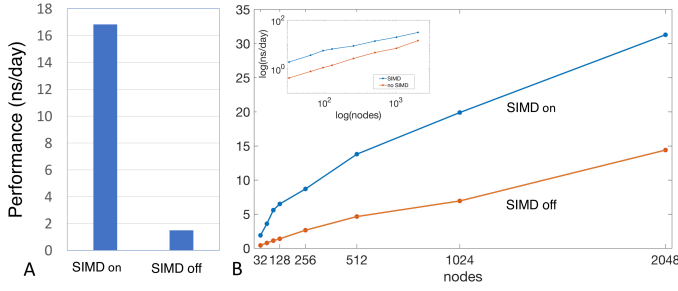


Fig. 3. Performance of GROMACS with and without the SIMD intrinsics layer. A: Using NERSC Cori (Cray XC40), a Xeon Phi, 134,177 atoms on one node, 16 MPI ranks with 4 OpenMP threads/rank, 2 fs timestep; performance is the average of five runs. B: Using OLCF Titan (Cray XK7), no GPU, 1.06 M atoms, 4 MPI ranks per node, 4 OpenMP threads/rank. 1.5 fs timestep.

18.0.1. In each case, the SIMD instructions can be turned off using the GMX flags “-DGMX_SIMD=None” in the cmake command line. The architecture-specific SIMD layer for Titan was turned on using “-DGMX_SIMD=AVX_128_FMA” and for Cori “-DGMX_SIMD=AVX_512_KNL.” GROMACS turns on O3 and all other required flags for automatic performance enhancement, even with the SIMD layer off. For the Titan build, we used the GROMACS-provided tuned, single-precision FFTW library, and for Cori, we used MKL’s FFTW. Both systems used MPICH for the MPI implementation. The GPU extension on Titan provided a $2.8 \times$ speedup, and the SIMD layer provided a speedup of $4.5 \times$, when averaged over all node counts tested. On Cori, the SIMD layer provided an astonishing $11.8 \times$ speedup. Therefore, we see that for GROMACS, the SIMD instructions can be more important to the performance than the GPU portions. Fig. 3 displays results for the SIMD layer. For the PP_{MD_c} score for GROMACS, a arithmetic mean over the SIMD results gives a speedup of $8.2 \times$; a harmonic mean may be more unbiased as it avoids over-emphasizing the KNL results, which may be outliers compared to other CPU-based systems (a more in-depth study would have used all machines possible for each program, while in this preliminary analysis we were unable to do so). The harmonic mean is $6.5 \times$. Thus for GROMACS the $PP_{MD_c} = 2/(6.5+2.8)=0.22$, or 21%, or with the arithmetic mean for SIMD, 18%; Table II displays this breakdown.

TABLE II
GROMACS PP_{MD_c} SCORE COMPONENTS

Non-portable Component	Speedup provided ($\times$)	
	<i>Titan</i>	<i>Cori</i>
CUDA-C	2.8	—
SIMD intrinsics	4.5	11.8
Final PP_{MD_c} = 21% (18%)		

C. LAMMPS

LAMMPS development has focused heavily on source portability, thus the choice to keep the code confined to a

portable subset of C++ and the use of only MPI for parallelization of the core program was made [4], [39], [80]. LAMMPS scales to thousands of nodes on DOE computers using MPI. All other parallelization, including threading with OpenMP, is accomplished by “USER” packages: separate modules that can be added to the LAMMPS build [40]. The Kokkos C++ library is an initiative at Sandia National Laboratories (SNL) to provide performance-portable acceleration options for applications [81]–[83], and has been used to create a Kokkos package for LAMMPS [84], [85]; currently, the only acceleration packages actively supported by the LAMMPS team at SNL are the Kokkos package, and the GPU package [86], (for which support is minimal, [87]). Threading within a node on the CPU can be accomplished with the Kokkos package [81], [84], [85], or alternately, the USER-OMP package [88] or the USER-INTEL package (which also provides SIMD vectorization support and support for the Xeon Phi) [89], and GPU-based acceleration can also use the Kokkos package, or the GPU package [4], [90]–[92]. The GPU package is able to use OpenCL or CUDA-C, but OpenCL threading on CPUs provides reduced performance compared to the USER-OMP package [90]. Furthermore, the support for this package has decreased and we did not test the OpenCL interface. We include this package as a non-portable component.

We tested two builds: the basic build and one with the GPU package, on Titan (Titan, section V-B); we also tried to test the Kokkos package, however, the out-of-the box build of Kokkos within the LAMMPS installation using the latest version of LAMMPS and the recently updated CUDA drivers on Titan was not successful (illustrating the fact that even achieving source portability of a program between various HPC systems is non-trivial). We supplement with the published benchmarks (“Accelerator benchmarks for CPU, GPU, KNL- Oct 2016”), using averages over the three systems for the USER-INTEL (not portable) versus the USER-OMP (portable) packages on the KNL in the PP_{MD_c} score [93]. This published benchmark focuses on one node; PP_{MD_c} scores are thus calculated for a single node only. For the GPU versus MPI-only test, we used the 32 K Lennard-Jones (LJ) benchmark [94], and 16 MPI ranks. The GPU package provided a $1.8 \times$ speedup, using the GPU and 16 MPI ranks, which gave the best performance. On a single node with GPU, LAMMPS achieves an average of 410.5 timesteps/second for this 32 K system with only LJ forces; this performance is much less than single-node performance of the other classical programs (compare to GROMACS, single-node, about 200-300 timesteps/second with approximately 80 K atoms, full electrostatics including PME, and bonded terms [8]). Table III displays these results, with a PP_{MD_c} score of 32%. It should be noted that if considering only the core MPI package PP_{MD_c} would be 1, however, accelerators would be ignored, which may exclude this program from the current analysis.

D. CP2K

CP2K uses its own sparse matrix multiplication library, Distributed Block Compressed Sparse Row (DBCSR), to run

TABLE III
LAMMPS PP_{MD_c} SCORE COMPONENTS

Non-portable component	Speedup provided ($\times$)
GPU package	1.8
Intel package (vectorization and threading) ^{a,b}	4.4

Final $PP_{MD_c} = 32\%$

^a From published LAMMPS benchmarks, [93]

^b USER-INTEL, speedup as compared to USER-OMP package

its LS-SCF routine. This library optimizes the multiplication of many small matrix blocks; it can use an auto-tuning library for small matrix multiplication (SMM) called libsmm, provided by the developers, for these small block multiplications [64]. DBCSR has been ported to the GPU using a CUDA-C based SMM implementation called libcusmm, and to the KNL [95], [96] using Intel’s version of this library, called libxsmm, that provides Intel-architecture-specific optimizations [97]. Parallel CP2K also uses ScaLAPACK and an FFTW library.

We used OLCF Titan (see section V-B), with the Cray LibSci library, to test the GPU port, and Eos, a Cray XC30 with Intel E5-2670 CPUs, which allowed us to use the MKL library and its Multi-threaded FFTW; we ran the LS-SCF 2048 water benchmark (6144 atoms) [65]. We used CP2K version 5.0-branch built with gcc 4.8.5 and CUDA toolkit 7.5, with MPICH for MPI, OpenMP, and the CP2K make flag DBCSR_ACC to turn on the GPU build. We tested CP2K without the libsmm library as this addition requires a time-consuming build procedure that can be difficult within the job scheduling and allocation environment on OLCF systems. We found that use of OpenMP decreased performance compared to using only maximum MPI ranks per node, and results use this configuration. We also considered published benchmarks to estimate the contribution from the Intel-specific libxsmm [65], [97] which we did not test. CP2K-provided, architecture agnostic libsmm seems to contribute approximately $1.4 \times$ speedup, and using MKL and libxsmm can greatly increase the performance of the LS-SCF, although some consideration must be made concerning the increased performance found for the Intel Sandy Bridge versus the AMD Opteron [98]. Our GPU build provided a $4.5 \times$ speedup versus the MPI-only build, using 64 nodes of Titan, which we also compared to published results from HECTOR. Using 1024 cores on Eos and Titan, we estimated the contribution of the Intel environment. We compared this to the Piz Daint results to estimate the contribution from libxsmm. Table IV displays these results.

VI. EXISTING RECOMMENDATIONS FOR PORTABLE DESIGN

We see that none of these BOP programs have a PP_{MD_c} score near 100%. Besides the speedup provided by these components, we can also investigate the code base and thus the programmatic decisions that were made to see how well they align with recommendations for portable software design. Here we review such recommendations from the literature. In

TABLE IV
CP2K PP_{MD_c} SCORE COMPONENTS

Non-portable component	Speedup provided ($\times$)
CUDA-C	4.3
Intel MKL libraries (and processor)	1.4
Intel libxsmm ^a	1.6

Final $PP_{MD_c} = 41\%$

^a estimated from published CP2K benchmarks, [65]

1978, Johnson and Ritchie of Bell Laboratories detailed their experiences creating the C language on the DEC PDP-11, and their attempts to port it to various other machines, first using the native operating system, and then using an accompanying porting of the UNIX operating system [99]. As a result, both UNIX and the C language underwent changes to increase their portability. The current problem of multiple dissimilar acceleration schemes for modern HPC systems mirrors these experiences. A few statements from that paper are especially relevant to present-day portability efforts;

“...we take the position that essentially all programs should be written in a language well above the level of machine instructions.”

The user should not have to be completely familiar with the system hardware in order to program the machine efficiently. In 1973 Hansen, Lawson, and Krogh, also envisioned a portable, standardized numerical linear algebra library to be called by FORTRAN, and whose back-end would be re-targeted for optimal performance, in various assembler languages specific to the particular hardware, by *system developers* [100]. This idea became reality thanks to the combination of an effort to standardize the BLAS library’s interface and the optimization of these routines by vendors for use with their architectures [101].

“Snobol4 was successfully moved to a large number of machines, and, while the implementation was sometimes inefficient, the technique made the language widely available and stimulated additional work leading to more efficient implementations,” [99].

While the effort to design portable applications may not result in instantaneously optimal results, long term benefits should pay back these initial costs.

Another lesson is that of the importance of “encapsulation” of machine-dependent sections of the program or application. The UNIX kernel contained approximately 2000 lines of code that were either machine-specific and written in assembler language, or machine-specific device driver, and written in C. The remaining 7000 or so lines of code for the kernel were in C and designed to be mostly portable for multiple machines, and proved to be so [99]. This encapsulation process allows for a clearly defined region that must be addressed for each machine, and greatly simplifies the porting process and its associated division of labor.

Three major themes from these historical efforts still apply with high relevance today:

- the choice of a high-level programming interface whose back-end is re-targetable by system developers for multiple architectures, and with a formal standardization of the interface via a consensus to provide for its uniformity,
- the encapsulation of machine specific sections of the program, and
- an agreement about the worthiness of the portability effort: that it is worthy even if it results in a temporary reduction in efficiency or functionality, because of the long-term improvements in the application such an effort provides.

Modern guidelines for creating portable software applications share aspects with early guidelines, namely encapsulation of machine-dependent sections, creation of a unified high-level interface, and standardization of the language or application programming interface (API), with a guarantee of community and industry support. In addition, the following recommendations are found in more recent literature:

A. Modularity

A critical element of portable applications is modularity [20], [21]. Of course, some level of modularity is required to satisfy the encapsulation guideline. But in addition to this separation, modularity of distinct subroutines and data structures in the application provide for a way to exchange them, and for easy rearrangement of tasks, data and communication procedures. This is especially important for portable parallel codes, as different architectures may require a different course-grained parallelization scheme [102].

B. Portable subset of the standard language or API

Together with the use of a standard programming language, one must determine what the *portable subset* of this language is [20], [21]. Not all elements of even a standard language are guaranteed to be portable in all situations.

C. Dedicated portable design effort and clear portability goals

While it may be impossible for very large, legacy codes to be rebuilt into a dedicated portable version, it has been noted that for optimal results in creating a portable application, the design, planning, and implementation processes should address portability at every step [20], [21]. Dedicated portability design efforts can often anticipate regions of the code that may require exchange, rearrangement, or redistribution of tasks or data [102]. A clear statement of portability goals is also important for the effort [20]–[22].

D. Summary of recommendations for portability

- High-level programming interface with re-targetable back end that is standardized and supported by a number of both commercial and open-source initiatives.
- Encapsulation of machine specific sections of the program.
- An agreement about the worthiness of the portability effort.

- Modularity of the algorithm sections and the data structures, with an ability to rearrange data storage and even to change the algorithm if required.
- Use of only the portable subset of the standard language.
- A dedicated portable design effort and clear portability goals.

The major caveat of the first recommendation, however, is that the interface developer must be both committed to the portability effort for the long-term, and supported by enough infrastructure to be able to maintain that commitment [20].

VII. FULFILLMENT OF PORTABILITY RECOMMENDATIONS BY THE MD PROGRAMS STUDIED

In this section we analyze the code base of each of the four programs to determine how well each satisfies the recommendations listed in section VI-D. The recommendations listed in VI-D have been condensed into four components of “fulfillment of portability recommendations,” (FPR): Use of high-level, portable languages and APIs, encapsulation of machine-specific sections, the modularity of the program, and the degree of emphasis placed on the use of portable interfaces that are re-targeted by professional programmers working for vendors or open source projects like GNU (and an agreement on the worthiness of this effort). For each, we try to assign a very rough score to quantify compliance: 0 if the recommendation is not met at all, 0.5 if the recommendation is addressed but not completely met, and 1 if the recommendation is well met.

A. NAMD

1) Use of high level programming interface with re-targetable back-end, portable subset of language or API:

The speedup provided by the use of the LLCA by Charm++ is considerable. The Charm++ developers are responsible for re-targeting Charm++ to each architecture: currently, IBM is working with Charm++ developers to optimize the PAMI interface for Charm++ on Summit [73], and the build has only been tested using the IBM-provided XL compiler. Charm++ is not a standard API that is ubiquitously provided by vendors and requires the work of an academic group for updates and support, so we rate this framework as only partially portable. NAMD also uses CUDA-C for its GPU extension. Charm++ implements its inter-node threading using POSIX threads which is a standardized execution model and thus is portable. We give this component of FPR a score of 0.5.

2) *Encapsulation of machine specific sections:* Encapsulation of the machine-specific portions of NAMD, at least for the GPU and LLCA components, is well accomplished. As noted above, the program can be built with an MPI-only interface and without GPU. A score of 1 is given for this component.

3) *Modularity of algorithms, code sections and data:* Because we are not as familiar with the code-base of NAMD, we were not able to estimate the degree of modularity of the program, aside from the contributions from encapsulation, therefore we did not score this item.

4) *Worthiness of portability effort, dedicated portability effort*: The worthiness of a portability effort is somewhat acknowledged by NAMD developers; the inclusion of an interface with MPI illustrates this. However, there are no alternatives provided for GPUs other than NVIDIA. Future development plans have included the use of OpenMP 4 for SIMD instructions on the CPU [37]. We give this item a score of 0.5.

B. GROMACS

1) *Use of high level programming interface with re-targetable back-end, portable subset of language or API*: GROMACS uses SIMD intrinsics for storage and math operations, as well as manual alignment of dynamically allocated variables. Encapsulated layers for the SIMD intrinsics, special allocation routines using new initialization functions, and a magic number for vector width have been created in order to allow for easier porting to each new architecture, while achieving optimal performance. The separated SIMD interface in which the math functions used in bottleneck regions can be written in a particular architecture's intrinsics can be considered a higher-level API. Each math operation's intrinsic function is tested extensively to determine the optimal intrinsic choice to minimize clock cycles. This can be a laborious process, as evidenced by the program's Gerrit code review history of commits: the SIMD intrinsics layer for the MIC architecture was worked on over the course of a full year [103]. Therefore, the amount of effort required from the developers is quite high. The CUDA-C interface which can be turned on or off both at compile time and run time (GROMACS has released a build with openCL that can make use of AMD GPUs [104]; this is still under development and does not function as effectively on NVIDIA GPUs as the CUDA-C version). We give this component of FPR a score of 0.5.

2) *Encapsulation of machine specific sections*: The machine-specific sections of GROMACS are well encapsulated, by the SIMD layer described above, and by separated GPU-based regions. In addition, different algorithmic components that are tuned for specific architectures, such as variable methods for dividing atoms into clusters, which are optimized for different architectures, are encapsulated in the code. We give this component of FPR a score of 1.

3) *Modularity of algorithms, code sections and data*: GROMACS is modular in some ways, as a result of encapsulation, and in addition, the FFTW library can be swapped out. However, certain algorithmic components cannot be easily exchanged. Within-node threading, currently using OpenMP, cannot be easily replaced. It was realized that it could be beneficial to be able to easily swap the threading libraries, when inadequate performance on some architectures was found to be related to threading. Newer threading libraries that can support thread overlap, and improved tasking and communication, may soon find their way into HPC programs. As a result, it was necessary to perform a manual change of the threading library for GROMACS, in order to incorporate the new Static Thread Scheduler (STS) threading library, which is being developed as

a possible OpenMP alternative [105]. This lack of portability of the threading implementation is evidence of insufficient modularity. We give this component a score of 0.5.

4) *Worthiness of portability effort, dedicated portability effort*: The porting layer cuts down the amount of work by a significant amount. For each architecture, only approximately 3000 lines of code need to be added; however, this work is dense, and technical, and each few lines is tested repeatedly until maximal performance is reached. The inclusion of an internal build of a single-precision FFTW version optimized for the program provides a fast library in the case that a vendor's version is not as optimized as that of other vendors. This is a solution to the problem of differences in performance of standard scientific libraries across platforms. The GROMACS project has recognized the worthiness of portability, and is deserving of a score of 1 here; however, these components of the program must still be ported by the (academic) developers (similar to the Charm++ LLCA interface in NAMD), and not by programmers working for the machine vendors. We must therefore give GROMACS a score of 0.5 for this component of FPR.

C. LAMMPS

1) *Use of high level programming interface with re-targetable back-end, portable subset of language or API*: LAMMPS core package uses only C++ with MPI, thus it is completely portable to Linux-based HPC systems. The Kokkos package is still under development, and the Intel and GPU packages are not performance portable to all architectures. We give this component of FPR a score of 0.5.

2) *Encapsulation of machine specific sections*: LAMMPS ports to specific architectures are encapsulated within the packages, giving a score of 1 for this component.

3) *Modularity of algorithms, code sections and data*: LAMMPS is very modular: threading, accelerators, and all other acceleration including Kokkos is added as a separate package. This component score is 1.

4) *Worthiness of portability effort, dedicated portability effort*: As we have mentioned above, LAMMPS has focused heavily on portability and thus this component of FPR is 1.

D. CP2K

1) *Use of high level programming interface with re-targetable back-end, portable subset of language or API*: CP2K uses OpenMP and MPI, and makes use of scientific libraries. In addition, the self-tuning SMM library is able to choose the most efficient small matrix multiplication method for different sized matrices for a particular architecture. However, the use of scientific libraries was found to provide greater increases in performance for particular vendor implementations, for instance, when using Intel's MKL with threaded FFTW, and the Intel version of SMM (libxsmm). CP2K also uses CUDA-C for its GPU port of DBCSR. An attempt was made to use the OpenMP SIMD constructs, however, performance was found to degrade with this addition [106]. We give this component a score of 0.5

2) *Encapsulation of machine specific sections*: Encapsulation is attained, with a separate CUDA build, and with all Intel-specific components found in separate libraries; we give a FPR component score of 1.

3) *Modularity of algorithms, code sections and data*: The majority of the LS-SCF algorithm is in the DBCSR library, thus the entire library could be exchanged. However, DBCSR itself does not seem to be modular to the point of allowing a different SMM library to be easily used instead of libsmm or libxsmm. We give this component a score of 0.5

4) *Worthiness of portability effort, dedicated portability effort*: Several aspects demonstrate the consideration of portability, but the use of CUDA-C, and the emphasis on Intel-specific speedups, diverge from this philosophy. We give this component a score of 0.5.

E. Gaps in Compliance

When looking at the FPR scores, we see deficits in a number of items. No program received a 0 for any items, but besides encapsulation, few components received a perfect score of 1, although LAMMPS scored 1 for 3 components. Extensive modularity was lacking, and programmatic choices were often made to maximize speedup over portability. These results point to areas to focus portability efforts.

VIII. DISCUSSION AND RECOMMENDATIONS

For these top-performing programs, optimization was achieved with a number of non-portable solutions. If high-level APIs such as compiler-directive-based interfaces begin to deliver performance that is very close to that achieved by these accelerating programming components, and if standard scientific libraries begin to offer more diverse routines, portability in scientific computing could benefit. Auto-vectorization by compilers is suboptimal [68]; this may be related to the underlying serial nature of the most commonly used languages in programming. Recently, we have seen the inclusion of SIMD constructs into OpenMP 4 [68], and the introduction of GPU-based threading using compiler directives, with both OpenMP and OpenACC [107], [108]. Obtaining SIMD-level optimization comparable to that achieved with intrinsic functions, but using OpenMP constructs, has yet to be tested. Performance of compiler directives on the GPU has varied greatly. With increased performance of these programming models, it may be possible to increase their use in HPC applications, especially given incentives as discussed below. We have also seen the creation of new accelerator-based matrix algebra subroutines such as batched BLAS operations [109]–[111], which could be incorporated as performance-portable solutions. A high level API for dynamic task management that could perform as well as Charm++ could be standardized and provided by each vendor with an optimized back-end for the particular machine, as well as tunable features to apply to different types of programs.

Sufficient program modularity on an algorithmic level was not found in the programs we studied. Parallel programming often requires complete rethinking of an algorithm, and

solutions may differ for different architectures [26], [27], [102]. Increased modularity of programs, via a more linear network of separated kernels that can be rearranged and replaced more easily, could increase the developer’s ability to replace threading procedures, linear algebra routines, and data-decomposition procedures.

Many languages have no syntax or semantics with which to express parallelization and to indicate safely parallelizable regions: the abstract machine for languages such as C is serial [112]. While compiler-directive-based APIs can provide necessary information to the compiler, it is possible that a fundamental paradigm shift in language will eventually be required. A parallel language could eliminate much of the difficulty involved in achieving good performance in an easy way on parallel systems, and may eliminate the need for so many machine-level, non-portable solutions.

Performance portability efforts may not be well-supported by the scientific field. As we have shown in this analysis, without the highly optimized, architecture-specific components, performance can be reduced as much as $5\text{--}10\times$. Recent advances in simulation time to multiple microseconds have provided some of the most stringent tests of the models; in some cases, such tests have uncovered errors that were not apparent over shorter simulation lengths [35], [36], [113], [114]. This type of result is highly important, and one that researchers may not be willing to sacrifice for investments in future portability. This contradiction has been noticed even in early studies of performance portability: “performance and portability are important but conflicting concerns,” [26]. Recently, the Overview of the DOE COE’s Performance Portability Meeting in 2017 included the following: “Making use of open standards, libraries, and software abstractions that allow for minimal code disruption without negatively impacting performance potential is the preferred path to programming, but it constitutes a large, as-yet-unsolved challenge,” [115]. We suggest that large granting foundations and supercomputing centers should provide more incentives for the use of performance-portable programs as opposed to simply high-performing programs. Otherwise, the motivation may not exist for domain scientists to suffer a possible loss of results, publications, and grants that could be caused by using a lower-performing but more portable version. Such incentives would help to increase the perceived worthiness of the portability effort, which we have found to be an important aspect of portable application design.

ACKNOWLEDGMENT

A. S. thanks Dmitry Lyakh and O. E. Bronson Messer for helpful discussions. The authors thank Nicholas Dean Smith for help with GROMACS building and optimization on Titan, and providing the large test system. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725, and the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of

REFERENCES

- [1] T. Schlick, *Molecular Modeling and Simulation: an Interdisciplinary Guide*. Springer Science & Business Media, 2010, vol. 21.
- [2] G. Ciccotti, M. Ferrario, and C. Schuette, "Molecular dynamics simulation," *Entropy*, vol. 16, p. 233, 2014.
- [3] M. Kunaseth, R. K. Kalia, A. Nakano, and P. Vashishta, "Performance modeling, analysis, and optimization of cell-list based molecular dynamics," in *CSC*, 2010, pp. 209–215.
- [4] W. M. Brown, P. Wang, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers—short range forces," *Computer Physics Communications*, vol. 182, no. 4, pp. 898–911, 2011.
- [5] A. Anthopoulos, I. Grimstead, and A. Brancale, "GPU-accelerated molecular mechanics computations," *Journal of Computational Chemistry*, vol. 34, no. 26, pp. 2249–2260, 2013.
- [6] A. W. Götz, M. J. Williamson, D. Xu, D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. Generalized Born," *Journal of Chemical Theory and Computation*, vol. 8, no. 5, pp. 1542–1555, 2012.
- [7] R. Salomon-Ferrer, A. W. Götz, D. Poole, S. Le Grand, and R. C. Walker, "Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit solvent particle mesh Ewald," *Journal of Chemical Theory and Computation*, vol. 9, no. 9, pp. 3878–3888, 2013.
- [8] C. Kutzner, S. Páll, M. Fechner, A. Esztermann, B. L. de Groot, and H. Grubmüller, "Best bang for your buck: GPU nodes for GROMACS biomolecular simulations," *Journal of Computational Chemistry*, vol. 36, no. 26, pp. 1990–2008, 2015.
- [9] W. M. Brown, J.-M. Y. Carrillo, N. Gavhane, F. M. Thakkar, and S. J. Plimpton, "Optimizing legacy molecular dynamics software with directive-based offload," *Computer Physics Communications*, vol. 195, pp. 95–101, 2015.
- [10] R. Schulz, B. Lindner, L. Petridis, and J. C. Smith, "Scaling of multimillion-atom biological molecular dynamics simulation on a petascale supercomputer," *Journal of Chemical Theory and Computation*, vol. 5, no. 10, pp. 2798–2808, 2009.
- [11] S. Pronk, S. Páll, R. Schulz, P. Larsson, P. Bjelkmar, R. Apostolov, M. R. Shirts, J. C. Smith, P. M. Kasson, D. van der Spoel *et al.*, "GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit," *Bioinformatics*, vol. 29, no. 7, pp. 845–854, 2013.
- [12] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1, pp. 19–25, 2015.
- [13] J. E. Stone, A.-P. Hynninen, J. C. Phillips, and K. Schulten, "Early experiences porting the NAMD and VMD molecular simulation and analysis software to GPU-accelerated OpenPOWER platforms," in *International Conference on High Performance Computing*. Springer, 2016, pp. 188–206.
- [14] J. VandeVondele, U. Borstnik, and J. Hutter, "Linear scaling self-consistent field calculations with millions of atoms in the condensed phase," *Journal of chemical theory and computation*, vol. 8, no. 10, pp. 3565–3573, 2012.
- [15] J. Hutter, M. Iannuzzi, F. Schiffmann, and J. VandeVondele, "Cp2k: atomistic simulations of condensed matter systems," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 4, no. 1, pp. 15–25, 2014.
- [16] V. V. Larrea, W. Joubert, M. G. Lopez, and O. Hernandez, "Early experiences writing performance portable OpenMP 4 codes," in *Proc. Cray User Group Meeting, London, England*, 2016.
- [17] M. G. Lopez, V. V. Larrea, W. Joubert, O. Hernandez, A. Haidar, S. Tomov, and J. Dongarra, "Towards achieving performance portability using directives for accelerators," in *Accelerator Programming Using Directives (WACCPD), 2016 Third Workshop on*. IEEE, 2016, pp. 13–24.
- [18] <https://www.lanl.gov/asc/doe-coe-mtg-2017.php>, accessed: 2018-08-20.
- [19] "NSF/Intel partnership on computer assisted programming for heterogeneous architectures (CAPA)," https://www.nsf.gov/funding/pgm_summ.jsp?pins_id=505319, accessed: 2018-08-20.
- [20] J. D. Mooney, "Bringing portability to the software process," *Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown WV*, 1997.
- [21] —, "Developing portable software," in *IFIP Congress Tutorials*. Springer, 2004, pp. 55–84.
- [22] S. R. Schach, *Object-oriented and Classical Software Engineering*. McGraw-Hill, 2002, pp. 215–255.
- [23] C. Bonati, S. Coscetti, M. D'Elia, M. Mesiti, F. Negro, E. Calore, S. F. Schifano, G. Silvi, and R. Tripiccone, "Design and optimization of a portable LQCD Monte Carlo code using OpenACC," *International Journal of Modern Physics C*, vol. 28, no. 05, p. 1750063, 2017.
- [24] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC—first experiences with real-world applications," *Euro-Par 2012 Parallel Processing*, pp. 859–870, 2012.
- [25] R. Nori, N. Karodiya, and H. Reza, "Portability testing of scientific computing software systems," in *Electro/Information Technology (EIT), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1–8.
- [26] J. Michalakas, R. Loft, and A. Bourgeois, "Performance-portability and the weather research and forecast model," *HPC Asia 2001*, 2001.
- [27] I. T. Foster, B. Toonen, and P. H. Worley, "Performance of massively parallel computers for spectral atmospheric models," *Journal of Atmospheric and Oceanic Technology*, vol. 13, no. 5, pp. 1031–1045, 1996.
- [28] R. O. Kirk, G. R. Mudalige, I. Z. Reguly, S. A. Wright, M. J. Martineau, and S. A. Jarvis, "Achieving performance portability for a heat conduction solver mini-application on modern multi-core systems," in *Cluster Computing (CLUSTER), 2017 IEEE International Conference on*. IEEE, 2017, pp. 834–841.
- [29] A. Sidelnik, S. Maleki, B. L. Chamberlain, M. J. Garzar'n, and D. Padua, "Performance portability with the chapel language," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 582–594.
- [30] S. J. Pennycook, J. D. Sewall, and V. Lee, "A metric for performance portability," *arXiv preprint arXiv:1611.07409*, 2016.
- [31] D. Marx and J. Hutter, "Ab initio molecular dynamics," *Parallel computing*, vol. 309, no. 309, p. 327, 2009.
- [32] R. Zwanzig and N. K. Ailawadi, "Statistical error due to finite time averaging in computer experiments," *Physical Review*, vol. 182, no. 1, p. 280, 1969.
- [33] G. R. Bowman, "Accurately modeling nanosecond protein dynamics requires at least microseconds of simulation," *Journal of computational chemistry*, vol. 37, no. 6, pp. 558–566, 2016.
- [34] X. Hu, L. Hong, M. D. Smith, T. Neusius, X. Cheng, and J. C. Smith, "The dynamics of single protein molecules is non-equilibrium and self-similar over thirteen decades in time," *Nature Physics*, vol. 12, no. 2, p. nphys3553, 2015.
- [35] C. Bergonzo, N. M. Henriksen, D. R. Roe, and T. E. Cheatham, "Highly sampled tetranucleotide and tetraloop motifs enable evaluation of common rna force fields," *Rna*, 2015.
- [36] K. El Hage, F. Hédin, P. K. Gupta, M. Meuwly, and M. Karplus, "Valid molecular dynamics simulations of human hemoglobin require a surprisingly large box size," *eLife*, vol. 7, p. e35560, 2018.
- [37] J. C. Phillips, L. Kale, R. Buch, and B. Acun, "Namd: Scalable molecular dynamics based on the charm++ parallel runtime system," in *Exascale Scientific Applications*. Chapman and Hall/CRC, 2017, pp. 119–144.
- [38] 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," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.
- [39] S. J. Plimpton, "The lammps molecular dynamics engine," <https://www.osti.gov/servlets/purl/1458156>, 2017.
- [40] <https://lammps.sandia.gov/>, accessed: 2018-08-20.
- [41] A. S. Rose and P. W. Hildebrand, "Ngl viewer: a web application for molecular visualization," *Nucleic acids research*, vol. 43, no. W1, pp. W576–W579, 2015.
- [42] G. Zhao, J. R. Perilla, E. L. Yufenyuy, X. Meng, B. Chen, J. Ning, J. Ahn, A. M. Gronenborn, K. Schulten, C. Aiken *et al.*, "Mature hiv-1 capsid structure by cryo-electron microscopy and all-atom molecular dynamics," *Nature*, vol. 497, no. 7451, p. 643, 2013.
- [43] N. Ban, P. Nissen, J. Hansen, P. B. Moore, and T. A. Steitz, "The complete atomic structure of the large ribosomal subunit at 2.4 Å resolution," *Science*, vol. 289, no. 5481, pp. 905–920, 2000.

- [44] H. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. Bhat, H. Weissig, I. Shindyalov, and P. Bourne, "The protein data bank nucleic acids research, 28: 235-242," *URL: www.rcsb.org Citation*, 2000.
- [45] <http://www.rcsb.org/>, accessed: 2018-08-22.
- [46] R. Wyckoff, "Interscience publishers, new york, new york rocksalt structure," *Crystal structures*, vol. 1, pp. 85-237, 1963.
- [47] W. Humphrey, A. Dalke, and K. Schulten, "VMD - Visual Molecular Dynamics," *Journal of Molecular Graphics*, vol. 14, pp. 33-38, 1996.
- [48] J. Stone, "An Efficient Library for Parallel Ray Tracing and Animation," Master's thesis, Computer Science Department, University of Missouri-Rolla, April 1998.
- [49] S. Pall, M. J. Abraham, C. Kutzner, B. Hess, and E. Lindahl, "Tackling exascale software challenges in molecular dynamics simulations with gromacs," in *International Conference on Exascale Applications and Software*. Springer, 2014, pp. 3-27.
- [50] www.ks.uiuc.edu/Research/namd/performance.html, accessed: 2017-07-14.
- [51] J. D. Nickels, S. Chatterjee, B. Mostofian, C. B. Stanley, M. Ohl, P. Zolnierczuk, R. Schulz, D. A. Myles, R. F. Standaert, J. G. Elkins, X. Cheng, and J. Katsaras, "Bacillus subtilis lipid extract, a branched-chain fatty acid model membrane," *The Journal of Physical Chemistry Letters*, vol. 8, no. 17, pp. 4214-4217, 2017.
- [52] J. V. Vermaas, L. Petridis, X. Qi, R. Schulz, B. Lindner, and J. C. Smith, "Mechanism of lignin inhibition of enzymatic biomass deconstruction," *Biotechnology for biofuels*, vol. 8, no. 1, p. 217, 2015.
- [53] J. A. Hadden, J. R. Perilla, C. J. Schlicksup, B. Venkatakrishnan, A. Zlotnick, and K. Schulten, "All-atom molecular dynamics of the hbv capsid reveals insights into biological function and cryo-em resolution limits," *eLife*, vol. 7, 2018.
- [54] J. R. Perilla and K. Schulten, "Physical properties of the hiv-1 capsid from all-atom molecular dynamics simulations," *Nature communications*, vol. 8, p. 15959, 2017.
- [55] https://lammps.sandia.gov/doc/Speed_bench.html, accessed: 2018-08-20.
- [56] S.-P. Fu, Z. Peng, H. Yuan, R. Kfoury, and Y.-N. Young, "Lennard-jones type pair-potential method for coarse-grained lipid bilayer membrane simulations in lammps," *Computer Physics Communications*, vol. 210, pp. 193-203, 2017.
- [57] W. Fakhardji and M. Gustafsson, "Molecular dynamics simulations of collision-induced absorption: Implementation in lammps," in *Journal of Physics: Conference Series*, vol. 810, no. 1. IOP Publishing, 2017, p. 012031.
- [58] K. Y. Fung, Y. R. Lin, P. J. Yu, J. J. Kai, and A. Hu, "Microscopic origin of black spot defect swelling in single crystal 3c-sic," *Journal of Nuclear Materials*, 2018.
- [59] A. Sedova, "Periodic DFT calculations of vibrational and molecular dynamics on large organic molecular systems using olcf computers," https://www.olcf.ornl.gov/wp-content/uploads/2017/11/Sedova_OLCF_user_meeting_2018_poster-1.pdf, accessed: 2018-08-27.
- [60] B. Milovanović, M. Kojić, M. Petković, and M. Etinski, "New insight into uracil stacking in water from ab initio molecular dynamics," *Journal of chemical theory and computation*, vol. 14, no. 5, pp. 2621-2632, 2018.
- [61] L. R. Pestana, N. Mardirossian, M. Head-Gordon, and T. Head-Gordon, "Ab initio molecular dynamics simulations of liquid water using high quality meta-gga functionals," *Chemical Science*, vol. 8, no. 5, pp. 3554-3565, 2017.
- [62] A. Hassanali, F. Giberti, J. Cuny, T. D. Kühne, and M. Parrinello, "Proton transfer through the water gossamer," *Proceedings of the National Academy of Sciences*, vol. 110, no. 34, pp. 13 723-13 728, 2013.
- [63] S. Roy, M. Galib, G. K. Schenter, and C. J. Mundy, "On the relation between marcus theory and ultrafast spectroscopy of solvation kinetics," *Chemical Physics Letters*, vol. 692, pp. 407-415, 2018.
- [64] U. Borštnik, J. VandeVondele, V. Weber, and J. Hutter, "Sparse matrix multiplication: The distributed block-compressed sparse row library," *Parallel Computing*, vol. 40, no. 5-6, pp. 47-58, 2014.
- [65] <https://www.cp2k.org/performance>, accessed: 2018-08-27.
- [66] M. J. Gillan, D. Alfè, and A. Michaelides, "Perspective: How good is dft for water?" *The Journal of chemical physics*, vol. 144, no. 13, p. 130901, 2016.
- [67] B. Sellner, M. Valiev, and S. M. Kathmann, "Charge and electric field fluctuations in aqueous nacl electrolytes," *The Journal of Physical Chemistry B*, vol. 117, no. 37, pp. 10 869-10 882, 2013.
- [68] J. Huber, O. Hernandez, and G. Lopez, "Effective vectorization with OpenMP 4.5, ORNL/TM-2016/391," Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States). Oak Ridge Leadership Computing Facility (OLCF), Tech. Rep., 2017.
- [69] S. Fang, Z. Du, Y. Fang, Y. Huang, Y. Chen, L. Eeckhout, O. Temam, H. Li, Y. Chen, and C. Wu, "Performance portability across heterogeneous socs using a generalized library-based approach," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 2, p. 21, 2014.
- [70] <https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/>, accessed: 2018-08-24.
- [71] <https://www.olcf.ornl.gov/caar/>, accessed: 2018-08-24.
- [72] <https://www.olcf.ornl.gov/for-users/system-user-guides/summit/system-overview/>, accessed: 2018-08-24.
- [73] <https://procurement.ornl.gov/rfp/CORAL2/CORAL-2-SOW-DRAFT-v16.pdf>, accessed: 2018-08-23.
- [74] <http://publib.boulder.ibm.com/epubs/pdf/a2314532.pdf>, accessed: 2018-08-24.
- [75] D. J. Hardy, "Improving namd performance on multi-gpu platforms, 16th annual workshop on charm++ and its applications," https://charm.cs.illinois.edu/workshops/charmWorkshop2018/slides/CharmWorkshop2018_namd_hardy.pdf, 2018.
- [76] 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 for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 81-91.
- [77] https://charm.cs.illinois.edu/workshops/charmWorkshop2018/slides/CharmWorkshop2018_namd_phillips.pdf, accessed: 2018-08-24.
- [78] S. B. Larson, J. Day, A. Greenwood, and A. McPherson, "Refined structure of satellite tobacco mosaic virus at 1.8 Å resolution," *Journal of molecular biology*, vol. 277, no. 1, pp. 37-59, 1998.
- [79] P. L. Freddolino, A. S. Arkhipov, S. B. Larson, A. McPherson, and K. Schulten, "Molecular dynamics simulations of the complete satellite tobacco mosaic virus," *Structure*, vol. 14, no. 3, pp. 437-449, 2006.
- [80] https://lammps.sandia.gov/tutorials/italy14/italy_overview_Mar14.pdf, accessed: 2018-08-24.
- [81] <http://www.apes-soft.org/sites/default/files/attachments/trott.pdf>, accessed: 2018-08-27.
- [82] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202-3216, 2014.
- [83] <https://www.osti.gov/servlets/purl/1365196>, accessed: 2018-08-27.
- [84] https://lammps.sandia.gov/doc/Speed_kokkos.html, accessed: 2018-08-27.
- [85] <https://www.osti.gov/servlets/purl/1365050>, accessed: 2018-08-27.
- [86] https://lammps.sandia.gov/doc/Speed_gpu.html, accessed: 2018-08-27.
- [87] Steven J. Plimpton, personal communication, 2018-03-02.
- [88] <https://github.com/lammps/lammps/commits/master/src/USER-OMP>, accessed: 2018-08-27.
- [89] https://lammps.sandia.gov/doc/Speed_intel.html, accessed: 2018-08-27.
- [90] https://lammps.sandia.gov/doc/Speed_gpu.html, accessed: 2018-08-28.
- [91] W. M. Brown, A. Kohlmeier, S. J. Plimpton, and A. N. Tharrington, "Implementing molecular dynamics on hybrid high performance computers-particle-particle particle-mesh," *Computer Physics Communications*, vol. 183, no. 3, pp. 449-459, 2012.
- [92] W. M. Brown and M. Yamada, "Implementing molecular dynamics on hybrid high performance computersthree-body potentials," *Computer Physics Communications*, vol. 184, no. 12, pp. 2785-2793, 2013.
- [93] <https://lammps.sandia.gov/bench.html>, accessed: 2018-08-27.
- [94] <https://lammps.sandia.gov/bench.html#lj>, accessed: 2018-08-27.
- [95] O. Schütt, P. Messmer, J. Hutter, and J. VandeVondele, "GPU accelerated sparse matrix matrix multiplication for linear scaling density functional theory," *Electronic Structure Calculations on Graphics Processing Units: From Quantum Chemistry to Condensed Matter Physics*, pp. 173-190, 2016.
- [96] I. Bethune, A. Gloess, J. Hutter, A. Lazzaro, H. Pabst, and F. Reid, "Porting of the dbcsr library for sparse matrix-matrix multiplications to intel xeon phi systems," *arXiv preprint arXiv:1708.03604*, 2017.
- [97] <http://grosser.es/presentations/2017-03-09--Hans-Pabst--libxsmm.pdf>, accessed: 2018-08-28.

- [98] D. Molka, D. Hackenberg, and R. Schöne, “Main memory and cache performance of intel sandy bridge and amd bulldozer,” in *Proceedings of the workshop on Memory Systems Performance and Correctness*. ACM, 2014, p. 4.
- [99] S. C. Johnson and D. M. Ritchie, “UNIX time-sharing system: Portability of C programs and the UNIX system,” *The Bell System Technical Journal*, vol. 57, no. 6, pp. 2021–2048, 1978.
- [100] R. J. Hanson, F. T. Krogh, and C. Lawson, “A proposal for standard linear algebra subprograms,” *Technical Memorandum 33-660, National Aeronautics and Space Administration*, 1973.
- [101] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, “A proposal for a set of level 3 basic linear algebra subprograms,” *ACM Sigum Newsletter*, vol. 22, no. 3, pp. 2–14, 1987.
- [102] G. Barlas, *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014.
- [103] <https://gerrit.gromacs.org/>, accessed: 2018-08-22.
- [104] <http://manual.gromacs.org/documentation/2018/install-guide/index.html>, accessed: 2018-08-23.
- [105] <https://github.com/eblen/STS>, accessed: 2018-08-22.
- [106] https://www.epcc.ed.ac.uk/sites/default/files/PDF/CP2K_vectorization_IPCC_summary.pdf, accessed: 2018-08-28.
- [107] www.openacc.org, 2017, accessed: 2017-07-14.
- [108] www.openmp.org, 2017, accessed: 2017-07-14.
- [109] icl.cs.utk.edu/magma, accessed: 2017-07-19.
- [110] <https://devblogs.nvidia.com/cublas-strided-batched-matrix-multiply/>, accessed: 2018-08-24.
- [111] <https://docs.nvidia.com/cuda/cublas/index.html>, accessed: 2018-08-24.
- [112] D. Chisnall, “C is not a low-level language,” *Queue*, vol. 16, no. 2, p. 10, 2018.
- [113] M. Krepl, M. Havrila, P. Stadlbauer, P. Banas, M. Otyepka, J. Pasulka, R. Stefl, and J. Sponer, “Can we execute stable microsecond-scale atomistic simulations of protein–rna complexes?” *Journal of chemical theory and computation*, vol. 11, no. 3, pp. 1220–1243, 2015.
- [114] S. Vangaveti, S. V. Ranganathan, and A. A. Chen, “Advances in rna molecular dynamics: a simulator’s guide to rna force fields,” *Wiley Interdisciplinary Reviews: RNA*, vol. 8, no. 2, p. e1396, 2017.
- [115] <https://www.lanl.gov/asc/doe-coe-mtg-2017.php>, accessed: 2018-08-22.