Introduction

When large software application projects with long expected lifetimes have to be repeatedly rewritten for each new HPC architecture in a low-level, machine-specific language, this results in an unproductive development model– not simply because of the extra expenses in terms of man-hours, but also, because the resulting code becomes more error-prone. This is due to shortened lifetimes, multiple authors, and therefore, fewer chances to debug, test, and improve the code over time.[1–4] Furthermore, writing code that is closer to machine-level is a more difficult task, and thus fundamentally more prone to error.[4,5]

Classical molecular dynamics has become a ubiquitously-used computational modeling tool for a number of disciplines, from biology and biochemistry, to geochemistry and polymer physics.[6,7] Many of its algorithmic elements are embarrassingly parallel, and due to intense efforts from a number of developers over the past 50 years, several MD programs have been highly successful in achieving commendable efficiency and overall performance in the high-performance computing (HPC) setting.[8–16] However, as can be seen from the wealth of recent literature documenting recent porting efforts onto various HPC architectures and environments, significant amounts of development time and man-hours were expended for each retargeting of a particular application. While these heroic efforts are impressive and highly valuable not only in the end-product they provide, but also in the knowledge gained in the porting process, it is also important to consider if it is possible to avoid such re-development expenses with each novel HPC hardware solution. In other words, is it possible to create scientific HPC applications that are more portable?

Portability of a computational application can be defined by the following criterion: an application is said to be portable, if the cost of porting is less than the cost of re-writing.[1–3] Here, costs can include development time and personnel compensations, as well as error production (bugs), 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 - (cost to port/cost to rewrite)**.[1] 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, for instance, binary portability (ability of the compiled code to run on a different machine) and source portability (ability of the source code to be compiled on a different machine and then executed).[1–3] In this paper, we discuss solutions, accepted best practices, and our own efforts to create dedicated source-portable scientific computing applications for HPC, using a mini-application for molecular dynamics (MD) simulations of short-range non-bonded forces as a test case. We explore the application re-design process, the use of the directive-based API OpenACC,[17] and standard HPC libraries as part of this effort.

Learning from history: portability efforts over time

In 1978, Johnson and Ritchie of Bell Laboratories published a paper detailing their experiences with creation of 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.[18] As a result, both UNIX and the C language underwent changes to increase their portability. These dedicated efforts resulted in both a language and an operating system that remain pervasively used and highly portable to this day.

In many ways, the current problem of multiple dissimilar acceleration schemes for modern HPC systems mirrors the struggles of Johnson, Ritchie, and their coworkers, and it is well worth a re-visitation of their findings to determine what principles were discovered that can be applied to the existing situation. We first examine a few statements from that paper which were found to be 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. While many of the arguments for this position are independent of portability, portability is itself a very important goal; we will show how it can be achieved almost as a by-product of the use of a suitable language."[18]

The CUDA C API is, in essence, very close to machine-level code, in that memory transfers must be explicitly programmed, and hardware specifics such as optimal block-size often find "their way into user programs," analogously to the way byte size "of a particularly efficient I/O structure" from the UNIX version found its way into the C language after porting to other machines (and operating systems) during Johnson and Ritchie's early experiences. CUDA C currently can only be used to program GPGPUs that are CUDA-enabled. These devices are primarily provided by the NVIDIA corporation. While extremely efficient code can be created with enough time and understanding of the device, this degree of development-expense may not be optimal or even permissible, and furthermore, it must be repeated, at additional expense, each time the application is ported to a new machine type. Generally, possessing this degree of skill and expertise, and putting forth this level of effort, are neither desired by the scientist nor should be expected of him, as it would significantly detract from his scientific pursuits. In fact, such a deep understanding of machine-specific elements can possibly only be achieved by a professional programmer who devotes his career to this type of work: "the hardest part of moving the compiler is not reflected in the number of lines changed, but is instead concerned with understanding the code generation issues, the C language, and the target machine well enough to make the modification effectively."[18]

Nevertheless, scientific computing needs can often be very niche-specific and thus commercial applications may not provide an adequate computational solution.[19] Furthermore, modern science has advanced to a level that some amount of computing– often an appreciable portion– is required for both the theoretical and experimental branches. Computational science in the recent past has become recognized as the "third pillar" of science by national agencies such as the NSF;[20] recently, trends indicate that it is *essential* to the functioning of the other two.[21] Thus, it is of great importance that scientific computing initiatives have accessible programming tools to produce efficient code that can be easily ported to a number

of HPC architectures, and that the machine-level specifics and optimization of these tools is performed by API, system, and hardware developers by retargeting of the tool's back-end for each new architecture, while maintaining a consistent, unified front-end interface for the computational scientist to use. In this situation, a third type of portability, called "portability of experience" is provided:[1,2] the users become familiar with the design and structure of the application, and this information is carried over to the application when it is ported to a new environment.

In the place of "language," then, we can substitute "API," in the above quotes, and we see that exactly such an initiative has been launched by several projects. OpenACC, which was first developed to provide a high-level interface for GPU programming, now has been extended to multi-core machines, and conversely, OpenMP,[22] once specific to CPU-based threading, has now been extended to the GPU in versions 4.0 and 4.5, we refer these versions here as 4.X. Both are in rather developmental stages as yet. Both of these APIs offer directive- or pragma-based interfaces with which to wrap sections of code for parallelization; they both appear in a similar format to the syntax used by OpenMP, which has now become familiar to many programmers of all levels. While other APIs also exist, these two in particular are supported by a number of commercial hardware and compiler developers, and in addition, by the GNU project,[23] which provides one of the most widely used collections of free software, via their GCC compiler.[24]

> "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."[18]

This statement expresses the idea that the effort to design portable applications may not result in instantaneously optimal results, but that long term benefits will pay back these initial costs, and in addition, the work will provide unforeseen contributions to the field via increase in accessibility of the application and familiarity with the underlying effort.

One last lesson that can be learned from Johnson and Ritchie's work is that of separation, or encapsulation, of machine-dependent sections of the program or application. The UNIX kernel contained approximately 2000 lines of code that was either machine specific and written in assembler, or machine-specific device drivers, 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.[18] 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.

Hansen, Lawson, and Krogh, in 1973, envisioned a portable, standardized numerical linear algebra library to be called by FORTRAN, and whose back-end would be retargeted for optimal performance, in various assembler languages specific to the particular hardware, by system developers.[25] This idea became reality thanks to the combination of an effort to standardize the library's interface and to create benchmarking goals that a computer's performance could be quantified by. The continued use of the BLAS libraries and

their incorporation into every optimized scientific library package provided by computer manufacturers is a testament to the success of this portability effort.[26]

We see four major themes from this review of historical efforts which still apply with high relevance today: the choice of a high-level programming interface whose back-end is re-targetable by its developer for multiple architectures, formal standardization of the interface via a legal consensus to provide for its uniformity, encapsulation of machine specific sections of the program, and the worthiness of the portability effort, 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, and because of the value to the computing community that an increase in accessibility of the application delivers. The major caveat for the first principle is that the interface developer is both committed to the portability effort for the long-term, and supported by enough infrastructure to be able to maintain that commitment.

Modern guidelines

Modern-day experts in the field provide a set of guidelines for creating portable software applications. A number of these are the same as those noted above, namely **encapsulation of machine-dependent sections**, creation of a **unified high-level interface**, and **standardization of the language and API**. As noted above, along with the creation of standard structure, parameters, and subroutines by the interface developers, the effort of system developers is a necessary component of the effort.[1] These may be the hardest criteria to verify, and thus can lead to insecurity about devoting a significant effort to design an application that is bound to a particular interface. One way to reduce such insecurities is to make sure the interface developers are widely supported, publicly committed, and are backed by confirmed, large-scale initiatives. Alternately, if the interface itself follows a standard design, or involves small additions to standard programming languages, then it may be possible that switching the interface would not require too much effort. Luckily, the twofold efforts of OpenMP and OpenACC now provide such an option, in the case of heterogeneous HPC.

Modularity

Another critical element of portable applications has been found to be **modularity**.[1,2] Of course, some level of modularity is required to satisfy the encapsulation guideline. But in addition to this separation, modularity of all distinct subroutines of the application provide for swapping of these subroutines, use of specialized, standard libraries in place of some previously originally written sections for increased performance, and 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.[27]

Portable subset of the standard language

Together with the use of a standard programming language, one must determine what the **portable subset** of this language is.[1,2] Not all elements of even a standard language are guaranteed to be portable in all situations. For example, not all features of the standard C++ or FORTRAN languages are compatible with OpenACC, and thus would not included in the portable subset of those languages, if OpenACC is the interface chosen to create portable HPC code. These details will be discussed further in the subsequent sections.

Dedicated portable design effort and clear portability goals

While it may be impossible for very large, legacy codes to start from scratch and rebuild a dedicated portable version, it has been noted that for optimal results in creating a portable application, **the design, planning, and implementation processes should, if possible, address portability at every step**.[1,2] This facilitates effective implementation of the other guidelines. For instance, for a hardware-specific implementation of an application, decomposition into modular components may not be easy or possible, if the program was not created with portability explicitly in mind. An example is a program built with a heavily nested tree of dependent subroutines. Dedicated portability design efforts can often anticipate regions of the code that may require swapping, rearrangement, or redistribution of tasks or data.[27] Dedicated design efforts also include simplifying the algorithm if possible, excellent documentation efforts, especially for encapsulated machine-specific regions, choice of simpler coding syntax, and generally, an approach that facilitates the ease of future workers to understand and modify the code over extended application lifetimes.[1–3]

Besides portability across architectures, a goal of parallel directives like OpenMP or OpenACC is to provide a way to accelerate regions of serial code without involving a major rewriting. This is, in a sense, portability across programming models. However, in the case of the use of these same directives to create portable parallel code, trying to include compiler directives such as OpenACC around previously serial regions will not lead to optimal results. Parallel programming often requires complete rethinking of an algorithm, with new problems arising for routines that were trivial to code in serial.[27] Thus the creation of a **dedicated portable parallel HPC application**, which is our goal in this project, uses the parallel directive APIs in a different way than one would to simply try to accelerate existing serial code.

A clear statement of **portability goals** is also important for the effort. The type of portability desired should be explicitly decided, and what exact systems the code should be able to easily be ported to anticipated at each design step. Deciding on the portability goals ahead of time will facilitate the dedicated design efforts of the portability project.

Final result: a better product?

Johnson and Ritchie wrote that "portable programs are good programs for more reasons than that they are portable."[18] After scanning the above guidelines for portability, one

can see that if followed, the result should be a long-lasting, well-written, modification-accessible product that should be easier to debug, and easier to work on in a task-distributed manner. Furthermore, application and code *reusability* can come from modularity, documentation, clarity, and linearity. Finally, from increased simplification, documentation, and standardization come the increased possibilities of programming help from the computer science community, as now there may be a decreased need to understand the science explicitly, or even the function of the application as a whole.

The non-bonded forces calculation for classical molecular dynamics

The classical molecular dynamics algorithm involves three main components: the integration step, the calculation of short-range forces, and the calculation of long-range forces. The integration step is generally the quickest part of the calculation, and as it has some memory-intensive aspects, is often calculated using the CPU, in implementations using heterogenous architectures. The long-range forces calculation, in most implementations, involves an Ewald-sum, and requires Fourier transform methods, which are fast for smaller systems, but do not scale well for large systems. This is an active area of development and is not addressed in the current project. The major bottleneck for all system sizes is the short-range non-bonded forces (SNFs) calculation, as it involves a sum of pairwise interactions over multiple subsets of the particle space.

The SNFs consist of the Lennard-Jones interaction, and short-range electrostatic forces. The Lennard-Jones interaction is an empirical function created to approximate the dispersive, or van der Waals forces, which in reality are purely quantum effects. The functional forms for these two additive forces are:

$$F_{LJ}(r_{ij}) = \Big[12\big(\frac{\sigma_{ij}^{12}}{r_{ij}^{13}}\big) - 6\big(\frac{\sigma_{ij}^{6}}{r_{ij}^{7}}\big)\Big]\frac{\mathbf{r}_{ij}}{r_{ij}},$$

$$F_C(r_{ij}) = \frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}^2}\frac{\mathbf{r}_{ij}}{r_{ij}}.$$

Here $F_{LJ}(r_{ij})$ is the Lennard-Jones force on atom i due to atom j, σ is a parameter that depends on the atom type of both interacting atoms, and $F_C(r_{ij})$ is the analogous Coulomb force, with q_n being the point-charge value assigned to atom n, and ϵ_0 the permittivity of free space; both are functions of the inter-atomic distance.

The Lennard-Jones and short-range electrostatic forces rapidly decay to zero outside of a radius of about 10-14 angstroms. This creates an excellent means of reducing the total calculation by imposing a distance-based radial cutoff on each atom, outside of which no interactions are considered. Algorithmically, the SNF calculation thus usually consists of a spatial decomposition of the system into three dimensional cells, followed by a binning of the atoms into their associated cells, then some sort of sorting, after which pairwise forces on each atom can be calculated and summed. These forces, as can be seen from their

equations above, depend on the pairwise distances between an atom and all other atoms within the radial cut-off. If the spatial decomposition into cells is performed so that the cells' dimensions are close to the LJ cut-off distance, then only the interacting cell-cell pairs need to be searched for interacting atoms, for each central cell. In the non-periodic regime, the number of interacting cells for a particular cell varies, from seven for a corner, to 26 for a central cell. In the periodic regime, all cells have 26 neighbors. Figure 1 shows a central cell and its interacting cell neighbors.

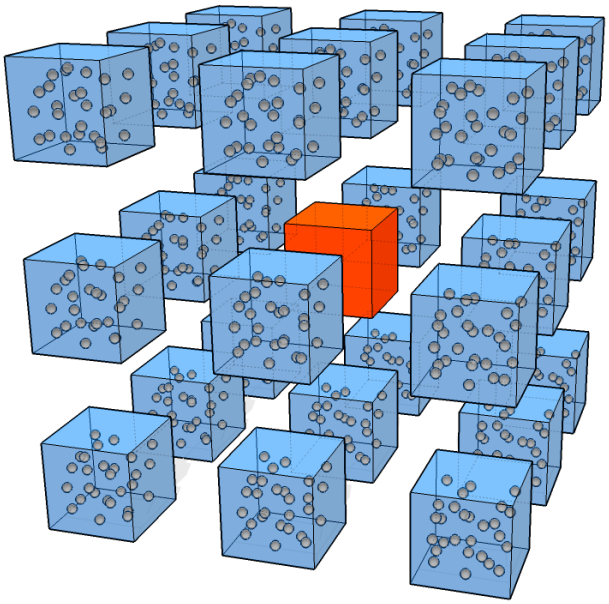

Figure 1: Schematic of the interaction neighbors for cell-cell interactions involved in the spatial decomposition in the molecular dynamics algorithm. The central box (orange), interacts with itself, and with its 26 immediate neighbors, creating a total of 27 interactions for each cell in the grid, if in a periodic system, or a range of interactions from 8-27 if in a non-periodic system. Boxes are exploded outward for visualization purposes, but sides are touching in the actual grid.

Traditionally, after determining which atoms are within the interaction radius for a particular atom, a list of these neighbors is tabulated for each atom. Since the early days of MD implementations on CPUs, it has been assumed that updating the neighbor list for each atom can occur every 10-20 time steps, because the atoms are not expected to move a significant amount each step. Originally, this approximation was seen as absolutely necessary, especially because the CPU-based programming model allowed more memory and communication operations and was more restricted by flops. In order to make sure that accuracy was preserved, a "Verlet skin" is also created. This is a region around the cutoff radius wherein the velocities of atoms are monitored for possible violation of the neighbor list assumption.

Portability goals: timings and architectures

Because of the very small time-step required (1-2 femtoseconds) for keeping the simulation from sustaining unacceptable drifts in energy, and because experimental time scales are generally of the order of milliseconds to seconds, MD developers have continuously pushed for increasingly shorter per-time-step execution rates, especially for the HPC MD applications. Currently, Gromacs[15] and NAMD[16] exhibit highly competitive timings per time-step, however, there is an interesting difference between the two. Gromacs developers have focused intensively on within-node threading with OpenMP, and thus their times for smaller systems (under 2 million atoms) are exceptionally fast, averaging about 1-2 ms per time-step, if not faster,[13] using up to approximately 500 nodes. However, above this number of atoms, communication overhead seems to overwhelm the calculation, and the timings per time-step drop significantly. For instance, for 7.7 M atoms, using approximately 3000 nodes, Gromacs attained only about 40 ms/time-step.[28] It is worth noting that the simulation of one million atoms is already an impressive milestone in molecular dynamics simulations, as it would have been an unfathomable goal for most researchers in past decades. Most biological and biochemical simulation needs do not usually require systems larger than 1-2 million atoms, but desire simulation time to be as long as possible, so that properties calculated from the simulation can be compared with experimental data. Furthermore, it is more difficult to obtain thousands of nodes at a time using commonly granted supercomputing resources. Thus, the Gromacs programming model is highly suitable for common academic pursuits. For NAMD, the use of the dedicated parallel programming library charm++ which focuses on medium-grained task distribution, scales exceptionally well for very large systems, especially when using thousands of nodes, as is the goal for the supercomputing environment. However, for systems under approximately 2 M atoms, the performance is sluggish, clocking about 11 ms/time-step for a 1 M atom system using 128 nodes.[29] 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. It is important to note, however, that this benchmarking information can be difficult to compare between one report and another, because most are reported in ns/day. However, the time-step taken can vary from 1-5 fs, and is often not reported. In addition, the type of ensemble used (NVE, NVT, or NPT) affects the timing, as does the details of the particle-mesh Ewald (PME) grid-size, and whether it is employed every time-step. These details are also not always reported. It is known that the PME calculation will not scale over about 512 nodes. For the above published benchmarks with 1 M, 21 M and 224 M atoms using NAMD, for instance, PME electrostatics were only calculated every 3 time-steps.[30] Thus it is unclear what portion of the linear scaling was due to the use of charm++, and what was a result of reduced numbers of PME calculations. On Titan, a 1.1 M atom system using Gromacs 5.1.3 times at about 1.2 ms/time-step using PME for every step, and a 2 fs time-step.

Nevertheless, while focusing on a dedicated portable version of the SNF calculation, and expecting some sacrifices in efficiency, **we are aiming to maintain the usability of the application as reflected by time-to-completion consistent with current expectations**.

In essence, we are requiring not only source portability, but also, *performance portability*. Explicitly, we are aiming for linear scaling, and rates under 10 ms/time-step for systems under 2 M atoms, and under 50 ms/time-step for systems approaching tens of millions of atoms, while hoping for rates to eventually drop to 3 ms/time-step in the smaller case, after optimization. We are also considering whether it is possible to achieve rates similar to Gromacs on small systems, and simultaneously to achieve scaling and NAMD-level rates on very large systems. It may be necessary for the program to switch kernels for varying system sizes, to balance communication overhead and workload. The modular nature of our application design should enable such swapping to be implemented.

We defined our portability goal to be the following: this application will be performance and source portable to a variety of HPC architectures using the Linux operating system and commonly provided compilers. Our initial test systems will include different heterogeneous CPU/GPU systems such as Titan and Summit, as well as multicore systems such as the Intel KNL, but we will keep in mind the possibility of additional systems that may develop, and attempt to design the application to anticipate various needs for data locality and communication models.

Choice of language/API

After deciding on the **use of accelerator directives such as OpenACC and OpenMP 4.X**, we decided to begin writing our initial test modules in the **C language**, as it enjoys native support on a variety of machines and is familiar to all programmers. This is especially pertinent due to the developmental stages that both directive models currently find themselves in. Thus, it should provide the least language-specific implementation problems for the two directive initiatives. For instance, currently OpenACC 2.5 is to be supported by the GNU Compiler Collection's version 6, however, there is still no complete GCC support for FORTRAN for OpenACC 2.5.[17,31] C++ functionality will be added sparingly as tests determine the the level of support for this extension. We believe that it is important to first identify the *portable subset of C* that works well with OpenACC and OpenMP 4.X, before extending the language, as is discussed in the subsequent sections.

Dedicated portable code: design deconstruction and creation of modular format

We began with a deconstruction of the SNF algorithm into its subtasks, to create the best system of modular components that would simultaneously allow for sub-algorithm and implementation swapping, individualized testing, communication graph rearrangement, swapping-out with standard parallel libraries, and, if necessary, creation of an encapsulated, dedicated kernel. One of the first places to receive a complete cut, for now, is the use of non-rectangular cells. Several highly optimized algorithms have been published that focused on the use of cells of varying complexity, from non-rectangular parallelepipeds to

sections of interlacing planes with highly complicated shapes.[15,32] While these interesting optimizations of the spatial-grid/communication interactions may have resulted in some amount of speed-up, these types of algorithms require more time to code, understand, and test, and thus are not a practical choice for a dedicated portability effort. **We decided to decompose the space only into a regular, rectangular grid that could be easily implemented**. Furthermore, once the grid is created, the location of each atom can very easily be calculated using a reduction over its three position coordinates. If an integer-based coordinate system for the grid is created, so that each cell has a 3-digit address in grid space, a one-digit address can also uniquely be determined if the direction of cell counting is fixed, exactly as occurs when a multidimensional array is"flattened" into a one-dimensional version, after the arrangement of the dimensions is decided.

We arrived at several **main computational modules**. The first module creates the cell-grid based on the minimum and maximum values of the atomic position along each dimension, and the cut-off radius given. We call this **"grid generation."** The second module creates the **"interaction list"** for the cell-cell interactions, using an operation of addition of -1, 0, and 1 to each of a cell's 3-digit-address components, and filtering those that go out of range, for the non-periodic implementation, and translating into the periodic spaces, for the periodic setting. The interaction list can be visualized as a two-dimensional list with the first dimension enumerating all cell IDs, and the second-dimension listing each interacting cell ID for each cell. For a constant-volume simulation (NVE or NVT), the interaction list only needs to be calculated one time. For a simulation with constant pressure, it must be calculated more frequently, potentially every time the volume is changed by the numerical "barostat." We began with a consideration of a constant-volume simulation, thus we have not focused at this time on parallelization of the grid-generation or interactions list; however, as they are both contained in isolated modules, their parallelization and optimization can be addressed in the future, if required.

The next module is the **"atom-binning."** Involved in this task is the assignment of each atom to its corresponding bin, referred to here as the "bin-assign" procedure. This was mentioned above, and is explained further in subsequent sections below. Additional steps involve counting the number of atoms in each bin, and the creation of the "permutation array," which is a list of the atom indices from the original list that correspond to each bin. The bin-assignment is a completely parallel task that is trivial to distribute and requires no redesign from an analogous serial algorithm. For the rest of the parallel binning algorithm, however, it is impossible to simply unroll the counting and the permutation array steps from a serial implementation: the very concept of counting, and the all-prefix-sums generation in order to keep track of the positions of bin divisions in the permutation array, are dependent on a sequential programmatic progression, and thus are inextricably tied to a serial implementation. This is an excellent example of how the use of OpenACC in a naïve way to speed-up a serial algorithm can fail completely, if the algorithm's serial version requires a complete restructuring in the parallel programming model.

The final two computational modules we defined are the **"pairwise-distance"** calculation, and the reduction of these distances into a single 3-dimensional force experienced

by each atom, the **"force-reduction."** We began by breaking down these calculations into parts that are *required* for the physics, and parts that are used traditionally for computational speed-up. The calculation of *all* pairwise distances of all atoms is the completely correct physical description of the forces, but this would be an impossible calculation even in the HPC regime. The decay of the force function makes the cut-off approximation both accurate and very computationally important. The cell-based spatial decomposition makes excellent use of the cut-off approximation.

The generation of the atomic neighbor list reduces the number of times the pairwise distances within the cell interactions are calculated, generally by ten times. However, they incur several additional overhead costs: potential inefficiency of many small data-structure accessing steps in their generation caused by increased launch-overhead and communication, increased bookkeeping requirements in the code, and the requirement to "batch" the calculations by hand on the GPU, leading to increases in code complexity and thus potential error-generation, and decreases in portability. Furthermore, the monitoring of the velocities of atoms in "shells" around each atom's cut-off radius lead to further bookkeeping, more small data structures, and more communication costs.

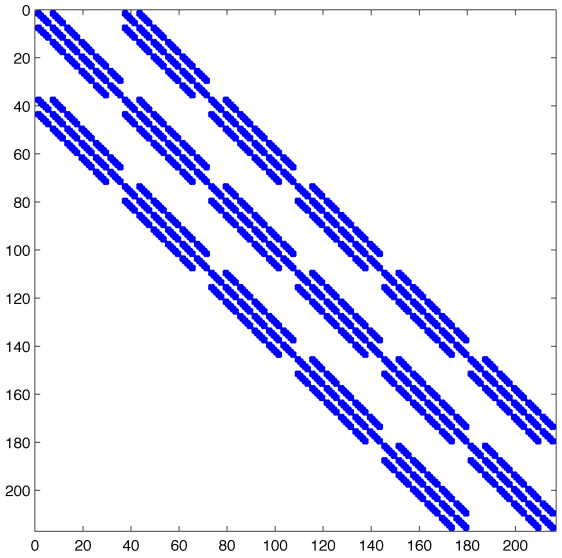

Figure 2: Distance matrix for cell-cell distances, for a solvated system of 30,000 atoms (small protein), with all distances outside of radial cut-off of 10 angstroms in white, and all within the cut-off in blue. The number of cells in each direction is 6, resulting in a total 46,656 cell-cell distances. 4096 of these are actually used due to the radial cut-off.

Figure 2 shows the distance matrix for all cell-cell interactions in the system, with those having distances greater than the cut-off colored white, and interacting cells colored blue. As can be seen, the cut-off creates a banded structure to the matrix, and reduces the number of cell-cell calculations by 90 %. We decided to begin with a temporary removal of the atomic neighbor list from the algorithm, to see how the code would time using an every-timestep cell-cell calculation. This allows for a separation of the two procedures, which

facilitates optimization of the cell-based routine, and exploration of different algorithmic methods with which to approach it. The incorporation of the atom-based neighbor list can then be added as a separate module, and may help to provide part of the kernel-switching requirements we envisioned for the small-to-large system size transition. We plan on revisiting the task with the intention of creating a more simplified, portable method of coding its implementation in the future. It is possible that, for a GPU-based implementation, however, the "flops are effectively free" programming model may lead to adequate performance despite the removal of the atomic-neighbor list. We will show in the subsequent section that preliminary tests support this idea. Furthermore, the idea that a majority of atoms will not change their cell locations in one time-step can be incorporated into the sorting algorithm used for binning, and thus some efficiency can possibly be regained from the use of that same physical property, but in a different way.

Building the modules: use of OpenACC

We began the building of our modules using OpenACC 2.5, as we are currently testing on the Titan and SummitDev machines. The Intel KNL with OpenACC/OpenMP 4.X functionality is still in its developmental stages. OpenACC was initially designed for GPU use, thus this should provide an API that has been tested more thoroughly for this type of system. OpenMP 4.X has more recently incorporated GPU-based functionality, thus its use will be tested in a second phase of this project, and comparisons with both APIs will include multicore CPU functionality as well.

Binning module: bin-assign, bin-count, all-prefix-sums and permutation array

Figure 3 shows the serial version of the bin-assign and bin-counting procedures. The most memory-efficient method, in serial, is to use a one-dimensional array of sorted atoms, and an accompanying all-prefix-sums array that keeps track of where each "bin" begins and ends in the sorted one-dimensional array. In a serial implementation, the bin-count can be accomplished in the same for-loop as the bin-assign.

For a parallel version, as noted above, "counting" is ill-defined, and this seemingly trivial computation in serial becomes a more difficult task in parallel. The serial version of the permutation-array generation step is also relatively trivial, and can incorporate the all-prefix-sums array generation and the bin counting as well. Figure 4 shows a version of this type of nested solution in a serial implementation.

Since the bin-assign procedure is independent for each atom, it is easily parallelizeable. The requirements simply involve a serial function with an OpenACC pragma to unroll it on the GPU. Figure 5 shows the implementation of its parallelization, in C using an OpenACC parallel loop pragma. It is further possible to potentially optimize this section using different OpenACC options. For 500,000 atoms, this section of the binning algorithm required 115 microseconds (0.115 ms), and for 30,000 atoms, 11 microseconds.

```
/* allocate bin_count, an array the length of the total number of bins,
with the number of atoms in each bin as the entry in each element of the
array: */
bin_count = (int *)malloc(numbins*sizeof(int));
 // initialize to zeros:
for(i=0;i<numbins;i++){
    bin_count[i]=0;
  }
  for (b = 0; b < NUMATOMS; b++) {
// read each atom's 3 coordinates and calculate the value of the 3-digit
address:
    temp[0] = floor((coords[b][0]/range[0]-kbinx)*num_divx);
    temp[1] = floor((coords[b][1]/range[1]-kbiny)*num_divy);
    temp[2] = floor((coords[b][2]/range[2]-kbinz)*num_divz);
// find the 1-digit address of the atom:
   n= num_divy*num_divz*(temp[0])+num_divz*(temp[1])+(temp[2]);
//enter the one digit address into that atom's index in the bins array:
    bins[b] = n;
// update the bin count in that bin's index in the bin_count array:

    bin_count[n]=bin_count[n]+1;
  }
```

Figure 3: Code snippet of a serial version of bin-assign, which includes a bin-element counting task in the same for-loop. The variable "bins" is a one-dimensional array the length of "NUMATOMS," the total number of atoms. Each element of bins contains the single-integer bin ID of the atom with that same index in its array, and the variable "coords" is a two-dimensional array allocated at compile time, containing the x, y, and z components of each atom's coordinates. The variable "bin_count" is a tally of the number of elements in each bin, and is an array the length of the number of bins, or cells, in our case.

```
count = 0;
  inc=0;
  for (b=0; b<numbins; b++){
    for (c = 0; c< NUMATOMS; c++){
      if(bins[c]==b){
        perm_array[count]=c;
        count++;
      }
    }
   ps[b]=count;
   bin_count[b]=(ps[b]+1)-inc;
   inc=ps[b]+1;
  }
```

Figure 4: Code snippet of a serial version of the permutation array ("perm_array") calculation, together with a nested prefix-sum ("ps") array generator and bin-element counter

Although the use of gangs, workers, and other data distribution keywords provided by OpenACC are defined for particular divisions of tasks, the actual performance of these various methods to create compiler-written code for a particular HPC architecture is highly system dependent. The most general pragma, the "kernels" directive, allows the API to determine what regions of the section can be parallelized, and to distribute these regions appropriately. A less general option is the parallel "loop" region, which specifically tells the compiler that the section is a loop to be unrolled. As can be seen in Figure 5, it was necessary to also tell the compiler that the variable "temp" was private to each parallel process, and thus would not have to be shared, copied, or reduced at any time.

Occasionally, the use of very specific options such as the gang and worker keywords

do *not* provide increased speed-up, and can even decrease the speed-up that a more-general option provides. Thus, these directive options must be tested in a somewhat trial-and-error fashion for various architectures. Fortunately, due to the unified interface of the API and the limited number of option possibilities, it should be rather easy to find the optimal options for each parallel section of code for a particular architecture, and finally, this change would still involve changing not more than a few lines in the program.

```
#pragma acc parallel loop private(temp)
  for (b = 0; b < NUMATOMS; b++) {
    temp[0] = floor((coords[b][0]/range[0]-kbinx)*num_divx);
    temp[1] = floor((coords[b][1]/range[1]-kbiny)*num_divy);
    temp[2] = floor((coords[b][2]/range[2]-kbinz)*num_divz);
    bins[b]= num_divy*num_divz*(temp[0])+num_divz*(temp[1])+(temp[2]);
  }

>> cc -acc -ta=tesla -Minfo=accel bin_acc.c -o bin_acc
main:
     60, Generating implicit copyout(bins[:])
         Generating implicit copyin(coords[:][:],range[:])
         Accelerator kernel generated
         Generating Tesla code
          61, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x
*/
     60, Local memory used for temp
```

Figure 5: Code snippet of bin-assign, parallel version, using the OpenACC parallel loop directive with addition of the "private" option, and compiler output.

Manual task division together with OpenACC pragmas

We also tested how some amount of manual splitting of the bin-assignment tasks would affect the speed-up. We separated the atoms into 5 evenly distributed blocks, and added OpenACC loops around both the blocks, and the inner bin-assign. Remarkably, this resulted in a 3-5 X speed-up, depending on the number of atoms. However, the speed-up may be system- and data-size- dependent, and it may be a difficult task to optimize this manual splitting by future users of the application. Nevertheless, we will continue to test this task distribution step and attempt to incorporate it into the program if it can be used without sacrificing large aspects of the portability.

Requirement for parallel algorithm design for binning: count and all-prefix-sums, and a need for encapsulation

The problematic parallelization of the bin-count procedure can be approached in several ways. We tried using an OpenACC directive to unroll the outer loop, over the bins. The compiler was able to perform this task, but it is unclear how this was accomplished; it is probable that the count array was copied to each of the task partitions and reduced at the end. However, it is questionable whether this is the optimal threading. The timing

for 30,000 atoms was far from ideal, and it is possible that the automated solution to this problem was not the correct one. To solve this, one can use an atomic-add for the bin-count array variable, which can be kept in a shared location in memory. These types of operations are supported by OpenACC's more advanced directive options. Alternately, one can create a type of merge-count, so that the bin-ID array is split into subarrays, each is counted in serial by parallel gangs, and the results are merged. It is also possible that for various architectures, there will be a different optimal solution for this step. Thus this is an example of a region of code that may require encapsulation, and heavy documentation, as well as several kernels to be used for specific systems.

For the all-prefix-sums calculation, after the bin counting is completed, the counts in each bin must be added and each partial sum must be recorded in a prefix-sum array. This, again is an inherently serial idea, and is non-trivial to design in an efficient parallel way. A theoretically efficient, parallel, sum-scan algorithm uses a binary-tree traversal to distribute the sums and then reduce them, but its efficient implementation can be hindered by hardware-specific memory access issues, and latency-management concerns on the GPU.[33] The solution requires a significant amount of effort, and thus is the type of procedure that would best be replaced with a call to hardware-specific libraries, that would each provide an optimized solution for a specific architecture. We will explore the use of various sorting procedures provided by HPC libraries such as Thrust[34] in ongoing work on this part of the binning module. This again is a region that must be encapsulated, and highly documented, because it may involve machine-specific solutions.

Alternate approach: two-dimensional run-time-allocated ragged array, OpenACC problems

An alternate approach to the use of a one-dimensional permutation array that requires the prefix-sum-based index calculation is to use a two-dimensional ragged array, with each top-level variable pointing to an array that contains the indices of the atoms in each corresponding bin. We attempted to try this approach, but discovered that a two-dimensional array that is allocated at run-time is not supported by OpenACC at this time, if using the simplest subset of directive options that can be called in a single line of code. On the other hand, run-time allocated one-dimensional arrays, and compile-time allocated two-dimensional arrays are supported. The problem is that in C, the pointer member can be any kind of data type, and the compiler cannot determine the length of the pointer members in order to copy them onto the device. Furthermore, allocated C pointer members are not guaranteed to be contiguous in memory, so the compiler cannot necessarily find the dereferenced elements.

Solutions to this problem do exist for OpenACC, in the form of some type of "deep copy." This can be a manual deep copy, where the top level of pointers is re-created on the device as a separate variable, the lower-level elements are also created manually as fixed to an orphaned object on the device, and then manually fixed to the top-level pointers on the device. This, however, is not simple, and seems to contradict the intended design goals in

using simple, single-line directives that do not require explicit creation of separate device variables. Without this type of procedure, however, many complex data structures, such as run-time allocated structs in C/C++, cannot be used. It is noteworthy, however, that the deep-copy procedure is currently under development by OpenACC in order to create a simpler, more user-friendly format.[17]

Alternatively, for systems that support it, the heterogeneously managed memory (HMM) functionality can be used. This is ultimately provided by CUDA's Unified Memory programming model which was initiated with CUDA compute capability 3.5, but has been significantly improved in compute capability 6 and the CUDA toolkit version 8 with the Pascal architecture.[35] In fact, although it was enabled in on GPUs such as the K20X on Titan, is was highly inefficient, and therefore, not recommended. Furthermore, it is just recently that OpenMP and OpenACC have incorporated its use. Therefore, although this solution is a possibility, it would only be an effective strategy for devices with compute capability 6.x. Therefore, a different solution would need to be developed for GPUs with lower compute capabilities, limiting the overall portable design of this module and increasing its complexity.

A final possibility for this problem could be the compile-time allocation of a padded two-dimensional array to hold the bin element indices, for instance, allowing each sub-array to be the length of the entire set of atoms. This solution appears to be a wasteful use of memory space, and further, the elements in the underlying contiguous memory space will then need to be accessed in a non-contiguous manner, due to the skipping over of empty array elements. However, it is a solution that may, in fact, be a viable option instead of requiring a separate library or kernel to perform an optimized all-prefix-sums calculation. Ongoing testing will determine how practical such an approach would be.

Portable subset of C with respect to parallel directives

We thus have determined, so far, that the HPC-portable subset of C that allows for the use of OpenACC on GPU, without more complex programmatic methods like manual deep copy, *does not include complex data structures such as run-time allocated double pointers*. The acceptable data structures include one-dimensional arrays, and compile-time allocated multi-dimensional arrays. It is possible that additional limitations will be found in future work on this application.

The pairwise distance calculation

Figure 6 shows the direct method of calculating all pairwise-distances between two sets of three-dimensional observations. This algorithm can be unrolled using a single OpenACC pragma, or optimized by dividing into batches, or specifying additional directive options.

```
void pairwise(double X[], double Y[], double D[], int ldX,
int ldY)
{
  int i;
  int j;
  double y[3];
  int k;
  double temp;
  for (i = 0; i < ldX; i++) {
    for (j = 0; j < ldY; j++) {
      for (k = 0; k < 3; k++)
      {
        temp =  X[k + 3 * i] - Y[k + 3 * j];
        y[k] = temp * temp;
      }
      temp = y[0];
      for (k = 0; k < 2; k++)
      {
        temp += y[k + 1];
      }
      D[i + ldX * j] = temp;
    }
  }
}
```

Figure 6: Code snippet, serial version, of the direct implementation of the pairwise-distance calculation.

Use of a BLAS-based algorithm to allow for HPC libraries

An interesting alternative to the direct version of the pairwise-distance calculation using higher-level matrix multiplication has been proposed in the past, and was used to create an accelerated all-pairwise-distance algorithm for use in Euclidean-distance-based clustering tasks.[36] Surprisingly, although this algorithm involves more total flops. However, because matrix operations such as matrix-matrix multiplication have been standardized by the BLAS (Basic Linear Algebra Subprograms)[37] initiative, they are included in most high-performing scientific libraries provided by system manufacturers, and therefore have also become benchmarks for measuring the performance of these systems. Thus they are competitively implemented in a highly optimized manner. Therefore, this version of the pairwise-distance calculation can result in faster performance than a direct version, even with compiler-based unrolling. Additionally, we have found that while as yet not a standard BLAS routine, a batched version of matrix-matrix multiplication (MM) for many small matrices exits in both the cuBLAS library, and the Intel MKL on the KNL. Furthermore, an open-source version of a batched MM solver provided by Magma,[38] an effort by the Dongarra group (which often achieves better performance than commercial versions, and pioneers new routines which are later adopted by commercial libraries),[36] adds additional kernels for varying matrix sizes. This is important, because long, skinny matrices often require different threading patterns to achieve maximum speed-up than square matrices do. On the other hand, for long, skinny MM, another performance factor to consider it the writing of the resulting product matrix, which is the size of the product of both long dimensions of the two ling, skinny matrices. Thus, it is important that the particular implementation of the MM algorithm does not involve an initialization step where a matrix full of zeros the size of the product matrix is written. This requires that the developers

create a separate kernel for the case where the optional weighting constant, which is part of the standard level-three BLAS dgemm (sgemm) routine, is zero.[37] The algorithm for the MM-based distance matrix calculation is shown in Algorithm 1.

Algorithm 1 Pairwise distance calculation using matrix operations, adapted form Li et al., 2011[36]

1: load matrices **A** and **B** and allocate memory for matrix **C**
2: **A** has dimension N by 3, **B** has dimension M by 3, and **C** has dimension N by M
3: note: $(\cdot)$ denotes elementwise multiplication
4: $\mathbf{v}_1 = (\mathbf{A} \cdot \mathbf{A})[1,1,1]^T$
5: $\mathbf{v}_1 = (\mathbf{B} \cdot \mathbf{B})[1,1,1]^T$
6: $\mathbf{P}_1 = [\mathbf{v}_1, \mathbf{v}_1, ..., \mathbf{v}_1]$ (dimension N by M)
7: $\mathbf{P}_2 = [\mathbf{v}_2, \mathbf{v}_2, ..., \mathbf{v}_2]^T$ (dimension N by M)
8: $\mathbf{P}_3 = \mathbf{A}\mathbf{B}^T$(dimension N by M)
9: $\mathbf{D}^2 = (\mathbf{P}_1 + \mathbf{P}_2 - 2\mathbf{P}_3)$, where $\mathbf{D}^2$ is the matrix of squared distances
10: pairwise distance matrix can be recovered from $\mathbf{D}^2$ by elementwise square-root

We proposed a possible solution for the calculation of all pairwise distances between all atoms in two interacting cells using this algorithm, to see if the optimized developer-provided batching would provide a way to avoid manual threading of the distance calculation. This method would also provide an easier way to divide the work over the cell-cell space, as the optimal batch number per GPU would decide how many cell-cell interactions could be handled at one time by a device. Because of the physics involved, each cell is expected to contain about 200 atoms. We implemented this algorithm using cuBLAS, the CUDA-based BLAS library provided by NVIDIA, initially with simply the matrix-multiplication section, in line 8 of Algorithm 1 ported to the GPU, where the matrices **A** and **B** are the atomic positions for atoms in two interacting cells. The remaining parts only involve sum parallel summations and can be parallelized using OpenACC. We checked timings for various sizes of N-by-3 matrices, and also, various numbers in the batch, for the batched dgemm (double-precision generalized matrix multiplication algorithm), as implemented by the cublasDgemmBatched routine. Figure 7 shows the results of the timing experiments for varying batch sizes of 200 by 3 matrices. It is worth noting that the host-to-device transfer need only occur one time for the entire SNF calculation, and all modules of the calculation can occur on the device.

As can be seen, the time increases linearly with batch size. For a system of about 30,000 atoms, which decomposes spatially into a 6 by 6 cell grid, there are about 2000 cell-cell interactions if the symmetry of the distance metric, or in physical terms, Newton's third law, is used, and about 4000 if it is not, for non-periodic calculations, and about 6000 for periodic. Thus, using 200 matrices per batch, and 10-20 nodes, depending on use of symmetry, all distances can be computed in under 2 ms, including data transfer time to the device. Even faster results can be achieved with fewer matrices in a batch, and the use of more

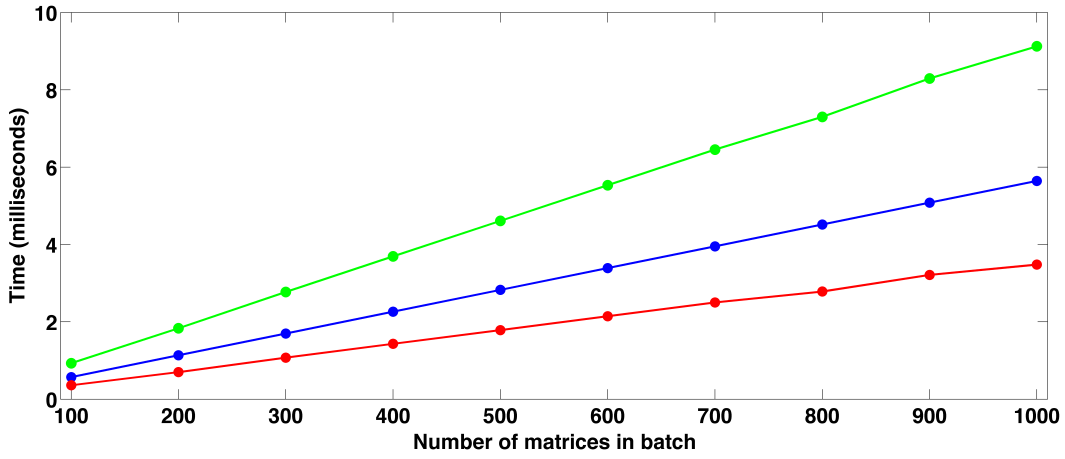

Figure 7: Timings of the batched dgemm calculation from cuBLAS, blue, host-to device transfer, red, and the total time, green, for matrices of size 200 by 3, for various batch sizes. Each time is the average over 100 batched calculations for each batch size.

nodes, however, this would involve more communication overhead. It is interesting to note that, even using 100 matrices in a batch, the total time is still under 10 ms, including data transfer. On the Summit machine, where multiple GPUs are available per node, this may be more efficient. Furthermore, the use of Magma's special kernels for smaller matrices may result in increased speed-ups. It can be seen that this task division should exhibit strong scaling efficiency, if not swamped by communication overhead. Additionally, to increase the cut-off distance by a factor of two, a new task distribution plan can easily be implemented by again changing the number of matrices per batch (and thus per node). For 800 by 3 matrices, the dgemm time for a batch of 100 was 8 ms. Further reduction in time can also be accomplished by the use of sgemm (single precision general matrix multiplication) instead of dgemm.

Use of OpenACC on the direct distance calculation

An alternative to using the MM method to compute the distances is to simply add OpenACC or OpenMP pragmas around the direct pairwise distance function shown in Figure 6. The drawback involves the loss of highly-optimized batching and threading, however, there is a decrease in number of floating-point operations. We are currently investigating optimal ways of applying OpenACC pragmas to this subfunction to achieve maximal performance.

Force reduction: precision considerations and the reciprocal square root

The final module involves using the pairwise distances to calculate the sum of all forces on each atom. For the LJ force, there is no need to compute a reciprocal square root (rsqrt), as the terms in the denominator are all even. For the Coulomb forces, however, the rsqrt must be computed. Many compilers have a fast version of the rsqrt. On a CUDA-enabled GPU,

a single-precision fast rsqrt is available. However, even this fast version requires 16 clock cycles, as compared to 4 for a simpler arithmetic operation such as addition. Furthermore, it has been argued that the use of single precision in the forces calculation can lead to unacceptable accumulated drifts in energy. Thus the use of the fast rsqrt must be carefully considered.

In addition, there are two possible design plans to approach the summed forces. The distance matrix can be read atom by atom, and a branching step performed if the squared distance is within the (squared) cut-off, upon which the force calculation is applied, and the running total is accumulated in a separate array. Unfortunately, however, this will require first tallying the total number to be summed to allocate the array, and then creating the sum in parallel, which again involves parallel-specific decisions similar to those discussed in the binning module. However, this avoids using the rsqrt too many times.

An alternate approach would be to simply compute all forces from all distances, regardless of whether they are actually within the cut-off, and allow for the functional form to return near-zero values for those outside of the cut-off. While this involves more rsqrt operations, it does not involve multiple counting steps, and the problems discussed above, and in addition, would greatly simplify the code for this region. It has yet to be seen if this is a viable solution for some systems, and will be determined through ongoing testing.

Communication and internode task distribution

The final tasks for this application will involve task distribution over multiple nodes, and optimization of this distribution for varying architectures. There are several possibilities to explore, including the use of a dedicated node to create the cell-grid and distribute cell-cell interaction to other nodes. For the GPU, data transfers require very high overhead, and thus, it will be optimal to limit the number of time the atomic position list is moved. It is possible that sending all atoms to each node to sort, and then choose their own set to calculate forces on, could be more optimal on some systems and for some range of system sizes.

The advantage to the modularity of our application design is that data and tasks can easily be distributed in multiple communication graphs for different systems, and tested for performance. It is also possible that some level of machine learning can be used to perform tests on additional machines and decide of a task distribution plan.

Conclusions

We have found that by adhering to accepted guidelines for portability, and maintaining a dedicated design process for an application, portable scientific computing applications can be systematically created. Challenges presented by the designing of HPC-portable applications involve difficulties in creation of parallel versions of serial programmatic routines, and often reveal the need for the use of high performance libraries created for each particular architecture by specialists. We have also found that through the use of carefully designed

modules and functions, together with accelerator directives such as OpenACC 2.5, acceptable performance for some tasks can be achieved relatively easily, and allow for a unified, portable interface for the application. Through encapsulation of the machine-specific regions such as those using dedicated library calls, a well-documented, code that is designed to be a simple as possible, and the unified interface, together with thorough consideration of possible future effects of each design decision with portability *and* performance always in mind, we believe that highly portable programs that have historically been designed in mostly machine-specific manners can be created.

In addition to the creation of these portable applications, we have found that the process involved in the design and implementation of such codes also creates subprograms and code snippets that provide examples of areas where the API has difficulties, or performs in a less-than-efficient way. These snippets can easily be extracted and tested due to the modular design required for portable applications. These examples, in turn, give the API developers test problems that may be outside of their usual testing routines, and thus help to maintain the cycle of interface-developer/system-developer collaboration that is seen as a requirement for the creation of portable, high-level interfaces for applications.

References

[1] James D Mooney. Bringing portability to the software process. *Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown WV*, 1997.

[2] James D Mooney. Developing portable software. In *IFIP Congress Tutorials*, pages 55–84. Springer, 2004.

[3] Stephen R Schach. *Object-oriented and Classical Software Engineering*, pages 215–255. McGraw-Hill, 2002.

[4] Claudio Bonati, Simone Coscetti, Massimo D'Elia, Michele Mesiti, Francesco Negro, Enrico Calore, Sebastiano Fabio Schifano, Giorgio Silvi, and Raffaele Tripiccione. Design and optimization of a portable LQCD Monte Carlo code using OpenACC. *International Journal of Modern Physics C*, 28(05):1750063, 2017.

[5] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. OpenACC–first experiences with real-world applications. *Euro-Par 2012 Parallel Processing*, pages 859–870, 2012.

[6] Tamar Schlick. *Molecular Modeling and Simulation: an Interdisciplinary Guide*, volume 21. Springer Science & Business Media, 2010.

[7] Giovanni Ciccotti, Mauro Ferrario, and Christof Schuette. Molecular dynamics simulation. *Entropy*, 16:233, 2014.

[8] Manaschai Kunaseth, Rajiv K. Kalia, Aiichiro Nakano, and Priya Vashishta. Performance modeling, analysis, and optimization of cell-list based molecular dynamics. In *CSC*, pages 209–215, 2010.

[9] W. Michael Brown, Peng Wang, Steven J. Plimpton, and Arnold N. Tharrington. Implementing molecular dynamics on hybrid high performance computers–short range forces. *Computer Physics Communications*, 182(4):898–911, 2011.

[10] Athanasios Anthopoulos, Ian Grimstead, and Andrea Brancale. GPU-accelerated molecular mechanics computations. *Journal of Computational Chemistry*, 34(26): 2249–2260, 2013.

[11] Andreas W Götz, Mark J Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross C Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 1. Generalized Born. *Journal of Chemical Theory and Computation*, 8(5): 1542–1555, 2012.

[12] Romelia Salomon-Ferrer, Andreas W Götz, Duncan Poole, Scott Le Grand, and Ross C Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs. 2. Explicit solvent particle mesh Ewald. *Journal of Chemical Theory and Computation*, 9(9):3878–3888, 2013.

[13] Carsten Kutzner, Szilárd Páll, Martin Fechner, Ansgar Esztermann, Bert L de Groot, and Helmut Grubmüller. Best bang for your buck: GPU nodes for GROMACS biomolecular simulations. *Journal of Computational Chemistry*, 36(26):1990–2008, 2015.

[14] W Michael Brown, Jan-Michael Y Carrillo, Nitin Gavhane, Foram M Thakkar, and Steven J Plimpton. Optimizing legacy molecular dynamics software with directive-based offload. *Computer Physics Communications*, 195:95–101, 2015.

[15] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C Smith, Berk Hess, and Erik Lindahl. GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1:19–25, 2015.

[16] John E Stone, Antti-Pekka Hynninen, James C Phillips, and Klaus 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*, pages 188–206. Springer, 2016.

[17] `www.openacc.org`, 2017. Accessed: 2017-07-14.

[18] Steven C Johnson and Dennis M Ritchie. UNIX time-sharing system: Portability of C programs and the UNIX system. *The Bell System Technical Journal*, 57(6):2021–2048, 1978.

[19] Rahul Nori, Nitin Karodiya, and Hassan Reza. Portability testing of scientific computing software systems. In *Electro/Information Technology (EIT), 2013 IEEE International Conference on*, pages 1–8. IEEE, 2013.

[20] Computational and data-enabled science and engineering. `https://www.nsf.gov`. Accessed: 2017-07-14.

[21] Matti Tedre and Peter J Denning. Shifting identities in computing: From a useful tool to a new method and theory of science. In *Informatics in the Future*, pages 1–16. Springer, 2017.

[22] `www.openmp.org`, 2017. Accessed: 2017-07-14.

[23] `www.gnu.org`, 2017. Accessed: 2017-07-14.

[24] `gcc.gnu.org`, 2017. Accessed: 2017-07-14.

[25] Richard J Hanson, Fred T Krogh, and CL Lawson. A proposal for standard linear algebra subprograms. *Technical Memorandum 33-660, National Aeronautics and Space Administration*, 1973.

[26] Jack Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A proposal for a set of level 3 basic linear algebra subprograms. *ACM Signum Newsletter*, 22(3):2–14, 1987.

[27] Gerassimos Barlas. *Multicore and GPU Programming: An integrated approach*. Elsevier, 2014.

[28] Sander Pronk, Szilárd Páll, Roland Schulz, Per Larsson, Pär Bjelkmar, Rossen Apostolov, Michael R Shirts, Jeremy C Smith, Peter M Kasson, David van der Spoel, et al. GROMACS 4.5: a high-throughput and highly parallel open source molecular simulation toolkit. *Bioinformatics*, 29(7):845–854, 2013.

[29] `www.ks.uiuc.edu/Research/namd/performance.html`. Accessed: 2017-07-14.

[30] James C Phillips, Yanhua Sun, Nikhil Jain, Eric J Bohm, and Laxmikant 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*, pages 81–91. IEEE Press, 2014.

[31] `gcc.gnu.org/wiki/OpenACC`. Accessed: 2017-07-14.

[32] Kevin J Bowers, Ron O Dror, and David E Shaw. Zonal methods for the parallel execution of range-limited N-body simulations. *Journal of Computational Physics*, 221(1):303–329, 2007.

[33] Mark Harris, Shubhabrata Sengupta, and John D. Owens. Chapter 39. Parallel Prefix Sum (Scan) with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*. Addison-Wesley, Boston, 2008.

[34] `thrust.github.io`. Accessed: 2017-07-19.

[35] CUDA 8 features revealed. `devblogs.nvidia.com/parallelforall/cuda-8-features-revealed`. Accessed: 2017-07-19.

[36] Qi Li, Vojislav Kecman, and Raied Salman. A chunking method for euclidean distance matrix calculation on large dataset using multi-gpu. In *Machine Learning and Applications (ICMLA), 2010 Ninth International Conference on*, pages 208–213. Ieee, 2010.

[37] BLAS (basic linear algebra subprograms). `www.netlib.org/blas`. Accessed: 2017-07-19.

[38] `icl.cs.utk.edu/magma`. Accessed: 2017-07-19.