PERFORMANCE ENHANCEMENTS FOR INTERATOMICPOTENTIALS.JL

JEREMIAH DEGREEFF*

Abstract. This project consists of improvements to the existing InteratomicPotentials.jl package. The first phase of improvements focused around the quality and consistency of the package, resulting in an upgrade from version 0.1 to a new version 0.2. The second phase of the project involved profiling of the code and a variety of both serial and parallel performance optimizations, resulting in a 20.2x speedup on my 6-core machine. Ongoing work for the project includes GPU acceleration and more efficient and flexible neighbor-list implementations.

*[**Note to grader:**] I became ill for over a week at the very end of the semester which necessitated a reduction in the scope I was hoping to cover with this project. I am continuing with this project after the conclusion of the semester and intend to ultimately complete the full scope, so this document primarily serves as a report of the progress made thus far with a discussion of the remaining areas of research in the Ongoing Work section. This reduction in scope was discussed with and approved by Prof. Edelman.*

1. Introduction. InteratomicPotentials.jl[1] is a relatively young package primarily authored by Dallas Foster[2] of the MIT Uncertainty Quantification group within the Aerospace Computation Design Laboratory. This package provides an interface for interatomic potentials which describe the forces, energies, and virial stresses between atoms or molecules in a particular configuration. This package is being developed as part of the Center for Exascale Simulation of Materials in Extreme Environments (CESMIX)[3]. Further background on the package and the ecosystem it is a part of is in section 2.

The scope of this project is twofold. First, to convert the experimental version 0.1 of InteratomicPotentials.jl into a more generalized, tested, and documented version 0.2 that is ready for public use. Second, to analyze the performance limitations of the current implementations and implement improvements. A discussion of the preliminary improvements made to the package is in section 3. The original goal for the performance improvements was to implement serial improvements, multithreaded improvements, and GPU support. As of the time of writing, only the serial and multithreaded improvements are complete. This progress report focuses on those enhancements in section 4. GPU acceleration and improvements to the neighbor-list calculations are discussed in section 5.

2. Background. CESMIX is a large, cross-disciplinary effort to advance the state of the art in predictive quantum and molecular simulations of materials, particularly in extreme environments. Due to the immense scale of our systems of interest, all software developed for the project must ultimately scale very well to run on a distributed system.

There are multiple existing codes in this space, the largest of which is LAMMPS[4], developed since the mid 1990s at Sandia National Labs. While these codes are well established and proven by decades of use and development, they are also monolithic and very difficult to augment with modern features. We saw a need to create a new package ecosystem that is more modular and supports features such as automatic

*Julia Lab, MIT CSAIL, Cambridge, MA (degreeff@mit.edu)

[1] https://github.com/cesmix-mit/InteratomicPotentials.jl

[2] https://github.com/dallasfoster

[3] https://computing.mit.edu/cesmix/

[4] https://www.lammps.org

This manuscript is for review purposes only.

differentiation and uncertainty quantification which are critical to the research goals of CESMIX. We accept that this ecosystem will not natively have all of the features present in the current state-of-the-art codes, but we focus preliminarily on the features that are essential for simulating our systems of interest. We also accept the trade-off that designing our codes for differentiability and uncertainty quantification will sacrifice some amount of peak performance.

What we are developing is a Julia ecosystem of packages which form a set of three composable workflows for classical molecular dynamics, ab initio molecular dynamics, and classical molecular dynamics with an actively learned potential[5]. One of the primary design goals of this work is composability in the sense that a user can easily swap in and out modules with the same role. This goal is achieved by defining simple Julia interface packages for each major component of the workflow. Any code, be it a Julia package or a wrapper around an existing C or Python code, that implements the specified interface can be composed into the ecosystem automatically. For example, the package Atomistic.jl[6] provides an interface for molecular dynamics simulators. We currently have implementations of this interface for NBodySimulator.jl[7] and Molly.jl[8], two Julia packages which offer different trade-offs and features for users. We also plan to provide a wrapper for LAMMPS.jl[9] in the near future.

InteratomicPotentials.jl is another such interface package. It defines an interface for specifying a potential between a configuration of atoms or molecules. Atomic configurations are provided by any implementation of the generic interface specified in AtomsBase.jl[10]. Implementations must specify methods for calculating the `potential_energy`, `force`, `virial_stress`, and `virial` of the configuration. The second part of the interface specifies the `parameters` and `hyperparameters` of the potential to be used for potential learning applications in the third composable workflow which are currently under development in PotentialLearning.jl[11]. The package includes implementations of five simple empirical potentials, a simple neighbor-list implementation that utilizes the `BallTree` implementation from NearestNeighbors.jl[12], and functionality for constructing a linear combination of multiple potentials. To our knowledge, other implementations of the InteratomicPotentials.jl interface currently exist in three packages: InteratomicBasisPotentials.jl[13], ASEPotential.jl[14], and DFTK.jl[15].

3. Preliminary Improvements. For the first month of the project, I focused on improving the quality and consistency of the InteratomicPotentials.jl codebase, resulting in a public version 0.2 release. The areas that I focused on were Unitful.jl[16] support, generalization of the parameter and hyperparameter interface, removal of dead experimental code, improvements to unit test coverage, and documentation.

The first major consistency improvement that I made was to add proper support

[5] https://github.com/cesmix-mit/ComposableWorkflows
[6] https://github.com/cesmix-mit/Atomistic.jl
[7] https://github.com/SciML/NBodySimulator.jl
[8] https://github.com/JuliaMolSim/Molly.jl
[9] https://github.com/cesmix-mit/LAMMPS.jl
[10] https://github.com/JuliaMolSim/AtomsBase.jl
[11] https://github.com/cesmix-mit/PotentialLearning.jl
[12] https://github.com/KristofferC/NearestNeighbors.jl
[13] https://github.com/cesmix-mit/InteratomicBasisPotentials.jl
[14] https://github.com/jrdegreeff/ASEPotential.jl
[15] https://github.com/JuliaMolSim/DFTK.jl
[16] https://github.com/PainterQubits/Unitful.jl

for unit-annotated quantities throughout the package[17]. Specifically, we use the Unitful.jl package to specify all units on the inputs and outputs of the calculations. We want to enforce units at the API layer in order to address one of the biggest problems in this space which is that people from different scientific backgrounds have different default unit assumptions. Explicitly labelling the units on the values we produce eliminates any confusion. The version 0.1 interface allowed units on the system configuration input (as this is part of the AtomsBase.jl API), but it naïvely striped them using `ustrip` without checking for consistency. This lead to invalid results whenever the units within the system or between the system and the parameters of the potential didn't all match. Version 0.1 also didn't allow initialization of potential structs with unitful quantities. We still want to use raw numbers internally for more efficient computations, but for version 0.2 we now assume that all raw floating point numbers represent Hartree atomic units[18]. We achieve this by stripping all unitful quantities with the `austrip` function provided by UnitfulAtomic.jl[19]. We provide two constructors for each interatomic potential: one which accepts unitful values (which are then stripped via `austrip`) and one that accepts raw floating point numbers (which we assume to be in atomic units). Once the calculations are complete, all quantities returned from our implementations are labelled with the corresponding atomic units. These design choices are consistent with other packages in the ecosystem such as DFTK.jl and Atomistic.jl. Note that by explicitly annotating the outputs, we leave room for packages that implement the API to use other unit conventions and still be able to integrate seamlessly within the ecosystem.

The second major improvement was to redesign and generalize the parameter and hyperparameter interface[20]. In version 0.1, the implementation of these features was a prototype and was inconsistently applied across the existing examples in the codebase. After discussing the goals of this functionality with Dallas, I was able to redesign the abstraction for version 0.2 to be generic enough that in most cases new implementations can just specify the symbols of their parameters and hyperparameters and accept the default implementations. The way that this was achieved is by introducing two new types into the abstract type tree. Specifically underneath the root of the type hierarchy (`AbstractPotential`) are now `TrainablePotential{P,HP}` and `NonTrainablePotential`. The two existing abstract potential subtypes are now subtypes of `TrainablePotential`. This extra layer of abstraction allows us to have default behaviors for the parameter and hyperparameter interface according to the type parameters `P` and `HP`, eliminating a significant amount of boilerplate code in the implementation of each concrete trainable potential[21]. Potentials which don't have any trainable parameters such as the `DFTKPotential`[22] in DFTK.jl can subtype `NonTrainablePotential` to have default implementations of the interface and avoid compatibility issues in potential learning workflows.

Lastly, I made a number of usability and quality improvements throughout the repository. I first discussed the status of various pieces of unused code with Dallas and removed the experimental code that had become obsolete[23]. In the final re-

[17] https://github.com/cesmix-mit/InteratomicPotentials.jl/pull/29
[18] https://en.wikipedia.org/wiki/Hartree_atomic_units
[19] https://github.com/sostock/UnitfulAtomic.jl
[20] https://github.com/cesmix-mit/InteratomicPotentials.jl/pull/32, https://github.com/cesmix-mit/InteratomicPotentials.jl/pull/35
[21] https://github.com/cesmix-mit/InteratomicPotentials.jl/commit/eeec524
[22] https://github.com/JuliaMolSim/DFTK.jl/blob/master/src/external/interatomicpotentials.jl
[23] https://github.com/cesmix-mit/InteratomicPotentials.jl/commit/4251105,

lease of version 0.1, the code coverage from the unit tests was approximately 40%. I rewrote all of the existing unit tests in order to test the correctness of each module in isolation[24], including the use of standard software development techniques such as mocking. In the process of thoroughly testing all of the existing features, I uncovered and patched many bugs in the codebase. I also fully tested all new features, and by the time of the official version 0.2 release achieved 100% code coverage[25]. I also worked with Dallas to write more extensive documentation for all of the modules within the package as well as instructions for people looking to use the interface or write their own implementations[26]. I wrote all of the boilerplate and most of the API documentation for individual types, functions, and constants, and Dallas filled in the scientific details and wrote most of the user instructions. There is still some open work to be done in this area[27], but the current stable docs can be found at https://cesmix-mit.github.io/InteratomicPotentials.jl/stable/.

4. Performance Analysis. In the context of a molecular dynamics simulation, the most important function provided by the InteratomicPotentials.jl API is the `energy_and_force` function which simultaneously calculates the energy and the force of an atomic configuration with respect to a particular potential. This combined function is used as a performance optimization because calculating the energy in addition to the force is usually negligibly more expensive than just calculating the force on its own. For all of the analysis in this paper, I focus on a cluster of argon atoms in a box with Dirichlet zero boundary conditions. I generate clusters of various sizes such that they have an even radial distribution. The algorithm for a scientifically reasonable cluster generation was provided by Dallas. For some tests I also used a pre-generated 1000-atom system, also provided by Dallas. The potential used in all of my tests is a Lennard Jones potential with scientifically accurate parameters again provided by Dallas. This is the simplest example potential for this type of system, but performance improvements made for this potential should also directly benefit all of the other empirical potentials and potentially inform performance enhancements for potentials in other packages down the line. All performance tests in this report were performed on a MacBook Pro with a 6-Core Intel Core i7 at 2.6 GHz and 16 GB of memory. I plan to repeat these tests on the MIT Supercloud[28], but have not yet been able to for reasons described in section 5 below.

To profile the performance of version 0.2.0 of InteratomicPotentials.jl, I focused on two things: the scaling of the `energy_and_force` with increasing number of atoms N as calculated with the `@belapsed` macro from BenchmarkTools.jl[29] and the profile of the `energy_and_force` as produced and visualized with the `@pprof` macro from PProf.jl[30]. The result of the scaling analysis[31] for values of $N = 250 : 250 : 10000$ can be seen in Figure 1. Over these 40 data points, I found that both a N^2 and a $N \log N$ regression fit the trend very closely (R^2 of 0.9974 and 0.9977 respectively), so it is not necessarily clear which more accurately describes this algorithm. This

https://github.com/cesmix-mit/InteratomicPotentials.jl/commit/9f64671

[24] https://github.com/cesmix-mit/InteratomicPotentials.jl/pull/30

[25] https://app.codecov.io/gh/cesmix-mit/InteratomicPotentials.jl

[26] https://github.com/cesmix-mit/InteratomicPotentials.jl/pull/41

[27] https://github.com/cesmix-mit/InteratomicPotentials.jl/issues/40

[28] https://supercloud.mit.edu

[29] https://github.com/JuliaCI/BenchmarkTools.jl

[30] https://github.com/JuliaPerf/PProf.jl

[31] https://github.com/cesmix-mit/InteratomicPotentials.jl/blob/performance-enhancements/benchmarks/scalability.jl

makes sense because the naïve algorithm without the neighbor-list is clearly $O(N^2)$, but adding the neighbor-list could theoretically reduce the complexity to $O(N \log N)$ for certain distributions of particles. Improvements to the neighbor-list which could help the best-case are discussed in section 5 below.

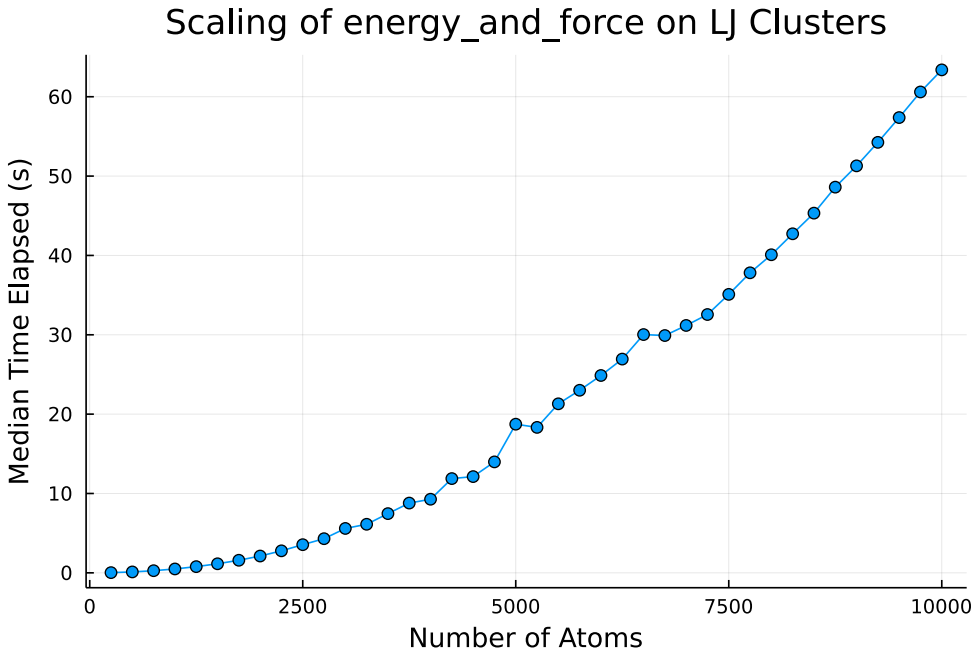

FIG. 1. *Scaling of* `energy_and_force` *in version 0.2.0 of InteratomicPotentials.jl*

I profiled the execution of `energy_and_force`[32] on systems of sizes $N = 1000$[33] and $N = 10000$[34]. I ran the $N = 1000$ calculation 100 times to have a similar sample size for both cases. The results were relatively similar for both as shown in Table 1, so I decided to focus on the case of $N = 1000$ as the standard benchmark[35] for my performance engineering as this simulation takes about two orders of magnitude less time to run than the larger system. My baseline median benchmark time with samples generated over 60 seconds to reduce noise was 524.964 ms.

I first optimized the serial performance of the functions `energy_and_force` and `neighborlist`[36]. One of the first inefficiencies that I saw was that we were pre-computing all of the norms of the (up to N^2) displacement vectors and allocating memory to store them in N different arrays. Because we only ever use these values once, it makes more sense just to calculate the norms when we need them. A similar

[32] https://github.com/cesmix-mit/InteratomicPotentials.jl/blob/performance-enhancements/benchmarks/profile.jl

[33] https://github.com/cesmix-mit/InteratomicPotentials.jl/blob/performance-enhancements/benchmarks/results/0.2.0/1000.pb.gz

[34] https://github.com/cesmix-mit/InteratomicPotentials.jl/blob/performance-enhancements/benchmarks/results/0.2.0/10000.pb.gz

[35] https://github.com/cesmix-mit/InteratomicPotentials.jl/blob/performance-enhancements/benchmarks/lj_1000.jl

[36] https://github.com/cesmix-mit/InteratomicPotentials.jl/commit/89230f4

function	% samples ($N = 1000$)	% samples ($N = 10000$)
`energy_and_force`	88.88%	89.12%
`neighborlist`	23.54%	32.30%
`atomic_symbol`	12.02%	12.06%
`getindex`	9.18%	9.75%
`inrange_point!`	4.71%	8.61%

TABLE 1
Five most commonly sampled functions in version 0.2.0 of InteratomicPotentials.jl

argument could be made for the displacements themselves, but I decided to hold off on this change until we decide on a final interface for the neighbor-list, as discussed in section 5. I also removed the sorting flag from the call to `inrange` because the sorting of neighbors just adds extra complexity without impacting correctness. In the `energy_and_force` function, I first extracted the looping logic into a separate function, which I called `_perform_pairwise`, because this logic is also shared with the `virial_stress` function. The main change in the looping logic was to hoist the call to `atomic_symbol` outside of the loop. I then split the loop into two versions. If the symbol check will always return true, we can get rid of the branching inside the loop entirely. Otherwise, we can still improve on the previous performance by pruning the `i` element early, not needing to check for `missing`, and loading the symbols from the cached local copy. This substantially decreases the time spent in `atomic_symbol` and `getindex` which were more than 10% of the version 0.2.0 execution time. Lastly, I used `@code_warntype` to analyze the type stability of each function and added type assertions where necessary to mitigate warnings. Running the same 1000-atom benchmark for 60 seconds, the median benchmark time for this version (hereafter called version 0.2.1-serial) was 95.285 ms for a speedup of 5.5x over version 0.2.0.

I then turned to parallel optimizations[37]. Specifically I threaded the main loops in both `neighborlist` and `_perform_pairwise`. The former was embarrassingly parallel, and the latter required a relatively straightforward reduce step to combine the results of the respective threads. I measured the median time to complete my standard benchmark system for 1 to 12 threads on my laptop with 6 physical cores. The resulting number of calls per second is displayed in Figure 2 with a comparison to the perfect speedup. We achieve roughly half of the ideal speedup until the number of physical cores is saturated. This is reasonable behavior because some of the computation (e.g. the construction of the `BallTree`) is not parallelizable, and there is also non-negligible overhead due to task management and memory allocation, particularly in the duplicated accumulator memory in the parallelized `_perform_pairwise`. With the optimal choice of thread count on my computer, I measured a median time of 26.022 ms on my standard benchmark for this version (hereafter called version 0.2.1-parallel) for a speedup of 3.7x over version 0.2.1-serial and a speedup of 20.2x over version 0.2.0. Figure 3 compares the results of the scaling analysis for clusters of $N = 250 : 250 : 10000$ atoms for each of the three versions.

5. Ongoing Work. To this point we have already started seen significant performance improvements when running on a single CPU. The next step for this project is to extend this performance engineering to a supercomputing cluster. I am currently looking into whether a distributed CPU implementation or a GPU implementation

[37] https://github.com/cesmix-mit/InteratomicPotentials.jl/commit/fc33b5a

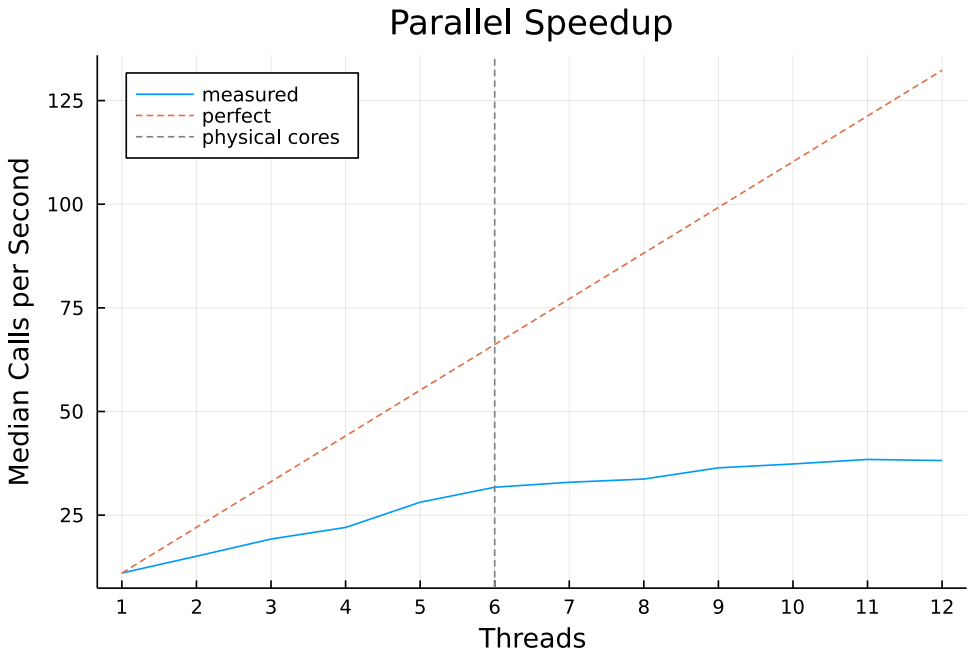

FIG. 2. *Parallel speedup of* `energy_and_force` *on a 6-core machine*

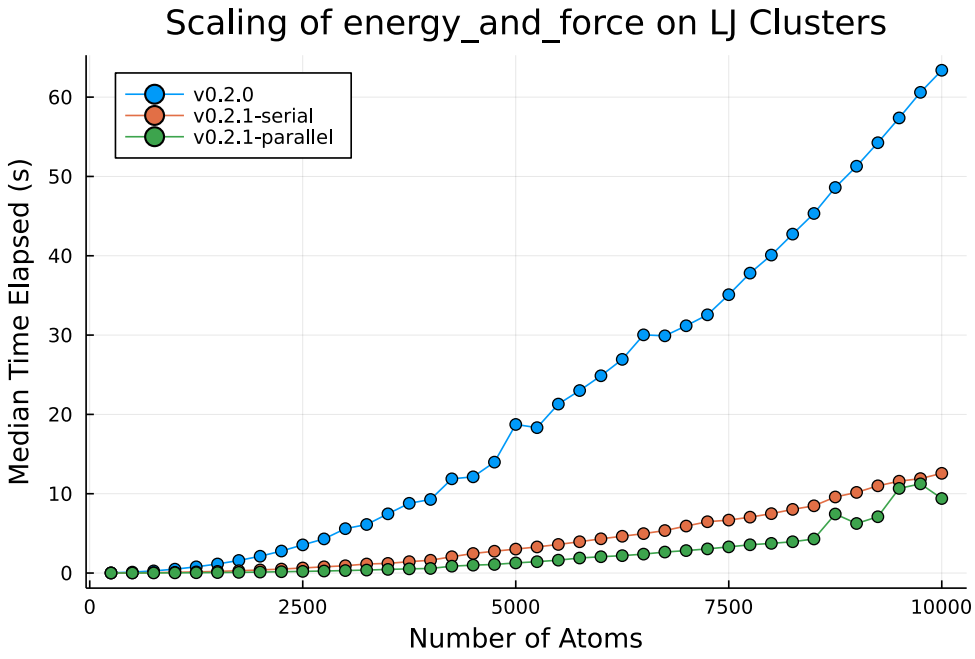

FIG. 3. *Scaling of* `energy_and_force` *in versions 0.2.0 and 0.2.1 of InteratomicPotentials.jl*

would be most desirable. I suspect that a GPU implementation will be the way to go because of the repetitive nature of the computation. In an uncertainty quantification setting, we will also likely want to run many full molecular dynamics simulations in parallel, so the force evaluation call in the inner loop of the simulation will likely be best constrained to a single processor. One concern that I have is whether we may run into memory issues on the GPU once we scale the system past a certain point. I plan to gather input on these design decisions this week from Valentin Churavy[38] and James Schloss[39] who are very experienced in this area. My current plan is to use the MIT Supercloud for doing performance engineering for these developments, but the versions of Julia supported by the Supercloud are currently not compatible with InteratomicPotentials.jl. Specifically, they only support up to version v1.6.1 currently, but I am actively in contact with the Supercloud administrators to find a solution for setting up either the LTS release (v1.6.6) or some version v1.7.X release.

The second piece of ongoing work is to redesign the neighbor-list computation within our workflows. Currently, InteratomicPotentials.jl contains a simple function called `neighborlist`[40] which constructs a neighbor-list from an AtomsBase.jl `AbstractSystem` using a NearestNeighbors.jl `BallTree`. This implementation is functionally correct, but is not the most effective within a molecular dynamic simulation context. The calculation of the neighbor-list is generally one of the more expensive parts of any pairwise interatomic potential calculation, so it is desirable to reuse the results within a molecular dynamics run. This could take the form of only recalculating the neighbor-list every so many timesteps or could involve more clever analysis of particle distribution and movement. Molly.jl[41] has a catalogue of a few different neighbor-list implementations[42] which take in AtomsBase.jl-compatible systems but produce a different representation than is currently in use within InteratomicPotentials.jl. There is also a less-developed existing package called NeighbourLists.jl[43] which has a couple of implementations but doesn't utilize the AtomsBase.jl API. In collaboration with Joe Greener[44] (primary author of Molly.jl) and Christoph Ortner[45] (primary author of NeigbourLists.jl), Dallas and I are currently designing an overhauled neighbor-list package that will incorporate all of these existing implementations. The package will provide a unified neighbor-list interface with multiple implementations for users to choose from that have different trade-offs. Users will also be able to define a custom implementation that fits the interface if desired. The common neighbor-list interface will be incorporated within the existing Atomistic.jl and InteratomicPotentials.jl interfaces to allow for a fully modular and composable design in which all three components are swapable. I plan to carry out this implementation within the next few weeks, pending consensus from all concerned parties.

Acknowledgments. I would like to thank Dallas Foster for providing frequent design consultation and code reviews throughout this project. Thanks also to Valentin Churavy for providing advice regarding profiling. Design and Development of the Julia CESMIX Ecosystem has been a collaborative effort with contributions from Alan Edelman, Valentin Churavy, Emmanuel Lujan, Michael Herbst, and Dallas Foster.

[38] https://github.com/vchuravy
[39] https://github.com/leios
[40] https://github.com/cesmix-mit/InteratomicPotentials.jl/blob/main/src/nnlist.jl
[41] https://github.com/JuliaMolSim/Molly.jl
[42] https://github.com/JuliaMolSim/Molly.jl/blob/master/src/neighbors.jl
[43] https://github.com/JuliaMolSim/NeighbourLists.jl
[44] https://github.com/jgreener64
[45] https://github.com/cortner