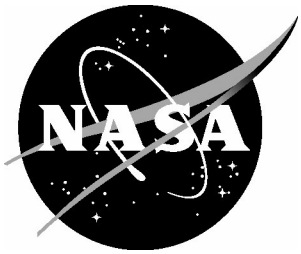
NASA/TM–2019-220409



High-Performance Computing Optimization for Aladyn – Adaptive Neural Network Molecular Dynamics Mini-Application

Vesselin I. Yamakov
National Institute of Aerospace, Hampton, Virginia

Gabriele Jost
NASA Ames Research Center, Moffett Field, California

Daniel Kokron
Redline Performance Solutions, Moffett Field, California

Yuri Mishin
George mason University, Fairfax, Virginia

Edward H. Glaessgen
Langley Research Center, Hampton, Virginia

September 2019

NASA STI Program . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA scientific and technical information (STI) program plays a key part in helping NASA maintain this important role.

The NASA STI program operates under the auspices of the Agency Chief Information Officer. It collects, organizes, provides for archiving, and disseminates NASA's STI. The NASA STI program provides access to the NTRS Registered and its public interface, the NASA Technical Reports Server, thus providing one of the largest collections of aeronautical and space science STI in the world. Results are published in both non-NASA channels and by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA Programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counter-part of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services also include organizing and publishing research results, distributing specialized research announcements and feeds, providing information desk and personal search support, and enabling data exchange services.

For more information about the NASA STI program, see the following:

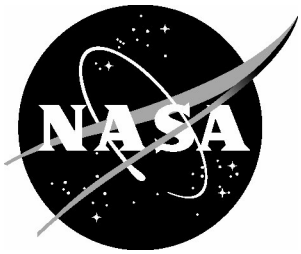
- Access the NASA STI program home page at http://www.sti.nasa.gov

- E-mail your question to help@sti.nasa.gov

- Phone the NASA STI Information Desk at 757-864-9658

- Write to:
  NASA STI Information Desk
  Mail Stop 148
  NASA Langley Research Center
  Hampton, VA 23681-2199

NASA/TM–2019-220409



High-Performance Computing Optimization for Aladyn – Adaptive Neural Network Molecular Dynamics Mini-Application

Vesselin I. Yamakov
National Institute of Aerospace, Hampton, Virginia

Gabriele Jost
NASA Ames Research Center, Moffett Field, California

Daniel Kokron
Redline Performance Solutions, Moffett Field, California

Yuri Mishin
George mason University, Fairfax, Virginia

Edward H. Glaessgen
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

September 2019

Acknowledgements

The development of the Aladyn mini-application software was initiated through funding from the NASA High-Performance Computing Incubator project. V. Yamakov is sponsored through cooperative agreement NNL09AA00A with the National Institute of Aerospace.

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Abstract

This report provides a description and performance evaluation of the optimization techniques for high performance computing (HPC) implementation of the open source Computational Materials mini-application Aladyn (https://github.com/nasa/aladyn). Aladyn is a basic molecular dynamics code written in FORTRAN 2003, which is designed to demonstrate the use of adaptive neural networks (ANNs) in atomistic simulations. The role of ANNs is to efficiently reproduce the very complex energy landscape resulting from the atomic interactions in materials with the accuracy of the more expensive quantum mechanics-based calculations. The ANN is trained on a large set of atomic structures calculated using the density functional theory (DFT) method. While achieving orders of magnitude faster computational performance than DFT, the ANN-based approach was still very computationally demanding compared to the conventional approach of using empirically fitted energy functions. After its initial development, Aladyn was evaluated and optimized by experts at the NASA Advanced Supercomputing (NAS) division to exploit modern supercomputer architectures. The code has been optimized for execution on multicore central processing units (CPUs), including Intel® Skylake microarchitecture, and on graphic accelerators, such as Nvidia® V100 graphic processing units (GPUs), using Open Multi-Processing (OpenMP) and Open Accelerators (OpenACC) programming interfaces. The optimization achieved a speedup of 4.7 times the baseline version on CPU performance and an additional 2.4 times on CPU+GPU performance.

1. Introduction

Atomistic computer simulations are a fundamental tool in materials research to model material properties form physics-based first principles. Atomic interaction, governed by Quantum Mechanics (QM) require sophisticated and highly computationally demanding mathematical models to calculate [1]. Classical methods use approximate functional forms, empirically fitted through a set of variable parameters to emulate atomic energies as direct functions of atomic coordinates [2]. While empirical potentials are computationally much simpler, allowing simulations of large-scale systems of up to a trillion (10^{12}) atoms [3], they are substantially less accurate compared to quantum calculations and applicable only to very specific atomic configurations or predefined crystallographic phases. A recently suggested approach is to use heuristic machine learning methods [4], such as those based on Adaptive Neural Networks (ANNs) to predict atomic energies, after being trained on a sufficiently large database of QM-calculated structures [5,6]. This approach reduces significantly the computational complexity, allowing for simulations of orders of magnitude larger systems compared to QM-based methods without compromising accuracy. Still, compared to classical methods using empirical energy functions, ANN methods remain two- to three orders of magnitude more computationally demanding. Hence, the computational cost of simulations, together with the need for extensive training of ANNs, still makes the practical implementation of ANN-based methods quite challenging.

The purpose of the Aladyn mini-application software [7], available as open source

at https://github.com/nasa/aladyn, is to be a testbed for exploring possible optimization strategies to develop highly scalable parallel algorithms for ANN-based atomistic simulations. Aladyn is aimed at utilizing the architecture of the high-end modern high-performance computing (HPC) hardware based on multicore central processing units (CPUs) equipped with graphic processing unit (GPU) accelerators. Specifically, the goal is to optimize the performance on a single HPC compute node, before implementing scaling to multi-node parallelization using message passing interface (MPI). At the same time, the open source code of Aladyn can serve as a training model for students and professors in academia.

2. Code Description and Algorithm

Aladyn is a basic molecular dynamics simulation [8] code to demonstrate the use of ANNs in calculating atomic energy and forces in a given atomic structure and performing a step integration of the equations of motion of all atoms to simulate structure evolution. In this approach, the ANN predicts the energy of an atom based on its local environment. Interatomic forces are calculated based on the spatial gradient of the energy and used to solve the Newtonian equations of motion to evolve the system in a classical molecular dynamics algorithm. The level of conservation of the total energy (kinetic and potential) of the system is used as a criterion for the correct execution of the simulation. To simplify the algorithm complexity of this mini-application, only single element systems, representing monocrystalline aluminum are considered.

The compilation and execution of the code, with the required input files are described in detail in ref. [8], as well as in a text file attached to its release at https://github.com/nasa/aladyn. This report gives higher focus on the mathematical algorithm and its computational implementation.

The neural network implementation algorithm in Aladyn follows the work by Behler and Parrinello [5]. The local environment is described through a set of Local Structure Parameters (LSPs) [5,6] defined for each atom as functions of the relative positions of its neighbors contained in a sphere of radius, r_c - the cut-off radius, which defines the interaction range. A fast search for neighbors in the vicinity of r_c is performed by applying the link-cell method [8] where the system box is divided into approximately cubic shape cells of size slightly larger than r_c (Figure 1). A list of atoms is maintained for each cell. Hence, the search for a neighbor in the interaction range of an atom does not have to exceed the nearest neighbor link-cells (marked in gray in Figure 1), limiting it to only 27 link-cells in 3D, rather than the whole system.

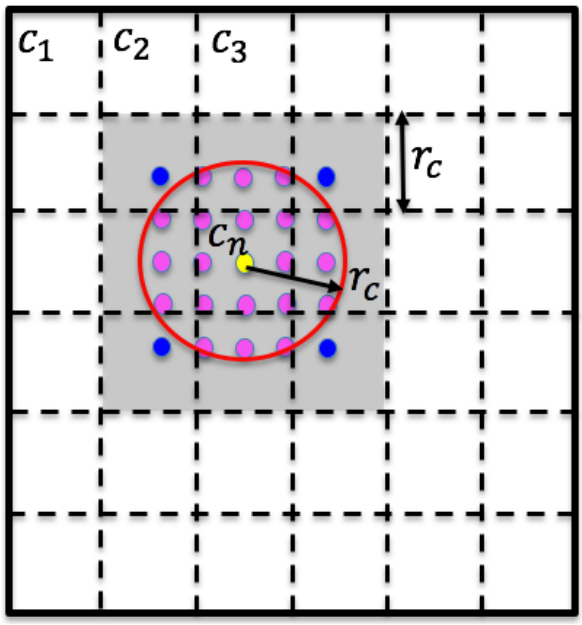


Figure 1. Schematic representation of the link-cell volume decomposition. Bold lines indicate the simulated system box boundaries. Dotted lines indicate the link-cell mesh of cells, $c_1, c_2, \ldots$. The cells in grey indicating the nearest cells, among which a search for neighbors of the central atom (in yellow, in cell, c_n) is performed.

After identifying all of the cut-off neighbors (*j*) for each atom (*i*) in the system, the code calculates individual LSP coefficients, G_i, for this atom as functions of the interatomic distances, r_{ij}, between atom (*i*) and its neighbors (*j*) as [9]

$$G_i^{(l,s)}(r_{ij}) = \sum_{j,k\neq i}^{r_{ij}\leq r_c} P_l(\cos\theta_{ijk}) f_s(r_{ij}) f_s(r_{ik}) \quad (l = 0,1,2,4,6;\ \ s = 1,2,..\ 12), \qquad (1)$$

where

$$f_s(r_{ij}) = e^{-(r_{ij}-r_s)^2/\sigma^2} f_c(r_{ij}), \qquad (2)$$

and

$$f_c(r_{ij}) = \begin{cases} \frac{(r_{ij}-r_c)^4}{d_c^4+(r_{ij}-r_c)^4} & : \quad r_{ij} \leq r_c \\ 0 & : \quad r_{ij} > r_c \end{cases}. \qquad (3)$$

In the above equations, $\sigma, r_{s=1,2,..12}, r_c$, and d_c, are model specific parameters, defined in the provided neural network potential file, ANN.dat. The functions, $P_l(x)$, are Legendre polynomials of order (l) defined through the iteration:

$$P_{l+1}(x) = [(2l+1)xP_l - lP_{l-1}]/(l+1); \quad P_0(x) = 1; \quad P_1(x) = x, \qquad (4)$$

and θ_{ijk} is the bond angle between the (*i*-*j*) and (*i*-*k*) bonds of atom (*i*), which expressed through the relative cartesian interatomic coordinates $(x_{ij} = x_j - x_i,\ \ y_{ij} = y_j - y_i,\ z_{ij} = z_j - z_i)$, and $(x_{ik} = x_k - x_i,\ \ y_{ik} = y_k - y_i,\ z_{ik} = z_k - z_i)$, is:

$$\cos\theta_{ijk} = \left(x_{ij}x_{ij} + y_{ij}y_{ij} + z_{ij}z_{ij}\right)/\left(r_{ij}r_{ik}\right). \quad (5)$$

The specific choice of (l, s)-set of values in Eq.(1) is determined on a case-by-case basis during training of the ANN for a given system. The resulted set of LSPs coefficients, $G_i^{(M_0=1,..60)}$, where M_0 counts all (l, s)-combinations - 60 in total, as given in Eq. (1) - are supplied as an input vector to the first input layer of the ANN.

The implemented ANN is a forward propagating neural network [6], consisting of an input first layer, one or more hidden layers, and an output layer. Each n-th layer of atom (i) can be represented as a vector $\vec{\boldsymbol{u}}^{(n)}(i) = \left(u_1^{(n)}(i), u_2^{(n)}(i), \ldots\ u_{M_n}^{(n)}(i)\right)$ of length M_n, with $M_0 = 60$ set as the length of the $\vec{\boldsymbol{G}}_i$ vector. The mathematical form of the ANN is expressed in matrix form through the iterations

$$\vec{\boldsymbol{u}}^{(1)}(i) = \vec{\boldsymbol{G}}_i * \hat{\boldsymbol{w}}^{(0,1)} + \vec{\boldsymbol{b}}^{(1)} \quad (6a)$$

$$\vec{\boldsymbol{u}}^{(n)}(i) = \vec{\boldsymbol{f}}\left(\vec{\boldsymbol{u}}^{(n-1)}(i)\right) * \hat{\boldsymbol{w}}^{(n-1,n)} + \vec{\boldsymbol{b}}^{(n)}; \quad n > 1. \quad (6b)$$

The first layer, $\boldsymbol{u}^{(1)}(i)$ in Eq. (6a), takes as an input the LSPs, $\vec{\boldsymbol{G}}_i$, of atom (i), weighted by the dot product $(*)$ with the weight matrix $\hat{\boldsymbol{w}}^{(0,1)}$ of size $(M_0 \times M_1)$. Next layers, $\vec{\boldsymbol{u}}^{(n)}(i)$, are calculated using Eq. (6b), where the input from the previous layer, $\vec{\boldsymbol{u}}^{(n-1)}(i)$, is modified through a transfer function

$$f(\boldsymbol{u}) = \frac{1}{1+e^{-\boldsymbol{u}}}. \quad (7)$$

The last layer consists of only one coefficient, giving the predicted potential energy of atom (i),

$$E_i = u^{(last)}(i). \quad (8)$$

The total system potential energy, E, is obtained as a sum of the potential energies of all atoms

$$E = \sum_i E_i. \quad (9)$$

The forces, acting on atom (i), are calculated as the spatial derivatives of E. A detailed description of the analytical differentiation of E, with the force and stress [10] calculation is given in Appendix A. The computational implementation of the analytical calculations is given in Appendix B and discussed in detail in Section 3.

Once the forces are known, a high precision 5-th order predictor-corrector scheme [11] is used to integrate the Newtonian equations of motion for each particle. The use of a high-order predictor-corrector integrator allows for accurate monitoring of the energy of the system [12] to identify any erroneous deviations from the energy conservation law as the system evolves.

The block scheme of the algorithm is given in Figure 2. At the beginning of the simulation, Aladyn reads the input structure as a list of atomic coordinates and velocities, together with the parameters of a trained ANN. The atomic velocities, v_i, define the initial temperature of the system, T, through the atomic kinetic energy, Q_i, as

$$Q_i = \frac{m\vec{v}_i^{\,2}}{2}, \tag{10}$$

so that

$$T = \frac{2}{3k_B}\sum_N Q_i, \tag{11}$$

where the summation is over all *N*-number of atoms, m is the atomic mass, and k_B is the Boltzmann constant.

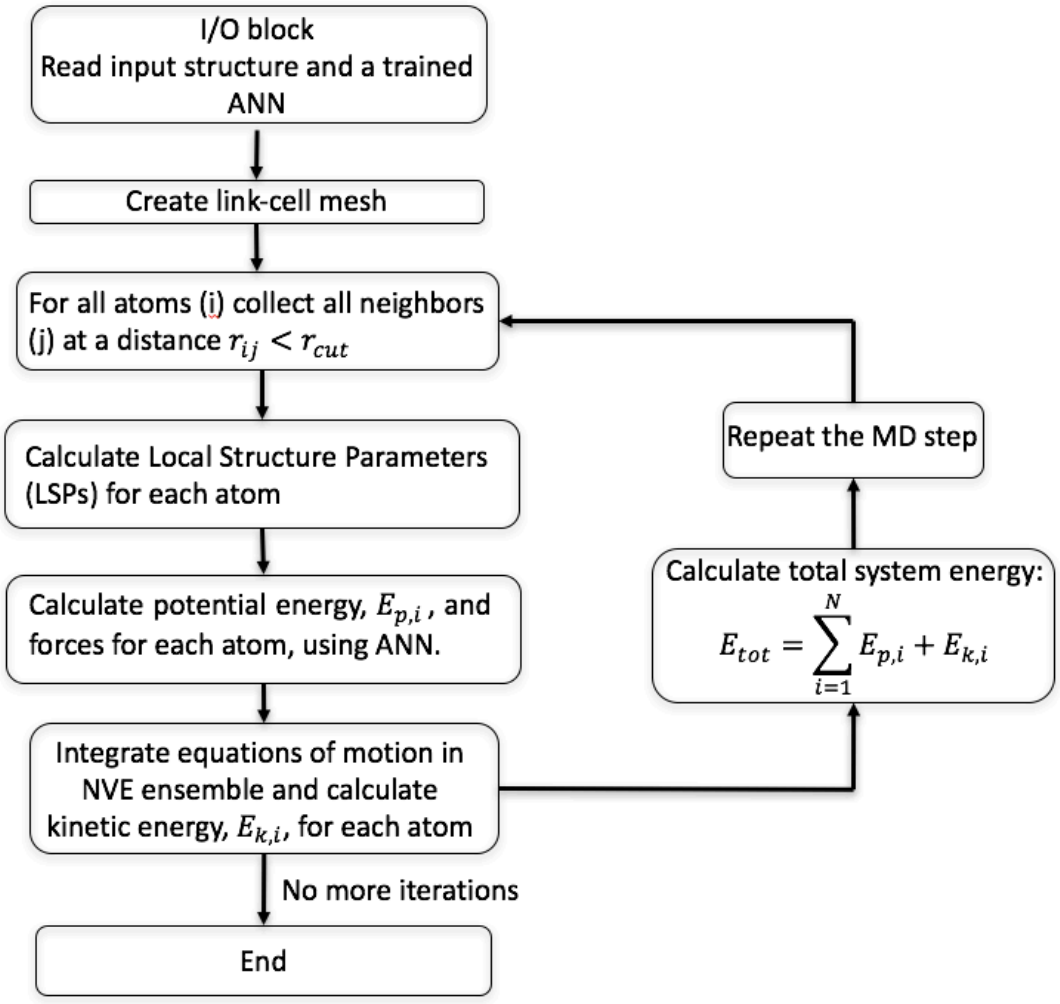


Figure 2. Flowchart summarizing the algorithm implemented in Aladyn for performing ANN based molecular dynamics simulation.

Based on the input system geometry, the algorithm creates a link-cell mesh over the entire system box. An MD step (or a time-step) starts with identifying all neighbors in a cut-off radius, r_c, around each atom of the system, using the link-cell list. Using the prepared list of neighbors, the LSPs are calculated for each atom, and supplied as an input to the ANN for energy and force calculation. The calculated forces are used to integrate the equations of motion for each atom and evolve the system by one MD step.

The updated atomic velocities, $\vec{v}_i$, resulting from the integration, are used to calculate the new kinetic energy of each atom, and update the overall temperature of the system, using Eq. (10) and Eq. (11), respectively.

The total system energy, calculated as

$$E_{tot} = \sum_{i=1}^{N}(E_i + Q_i) = const. \quad (12)$$

is reported and used as a verification test of the simulation since it must remain constant during the simulation. The updated atomic positions are used to calculate new LSPs, and the next steps repeat until the end of the simulation.

3. Code Optimization

3.1. Hardware and test case description

The code has been tested on two systems: A dual socket 20-core Xeon Gold 6148 model with a base clock speed of 2.4 GHz and a dual socket 18-core Intel Xeon Gold 6154 model with 4 Nvidia V100 GPU cards.

The code was compiled using the ifort v19 compiler with OpenMP programming interface [13,14], and the PGI v19 compiler. The OpenACC 2.6 programming interface [15] was used to enable GPU acceleration. The test case was an MD simulation of an aluminum crystal of 192,000 atoms simulated for 300 time-steps (MD steps). MPI was not used in the parallelization scheme, because the purpose of Aladyn is to optimize the performance on a single modern HPC compute node. This does, however, limit the code to the use of only 1 GPU card, thereby not fully utilizing its resources in terms of GPU accelerators. In a production code, MPI can be used to scale the performance and system size over multiple compute nodes or multiple GPUs on a single compute node by treating it as multiple MPI nodes (MPI processes).

3.2. Algorithm optimization

It was found that the optimal algorithmic realization of the calculations as outlined in Appendices A and B differs for the OpenMP and OpenACC programing models.

The first part, involving the nearest neighbor search (Loop 0: in Appendix B), was realized through two nested loops. The first loop is over the link-cells, $c_1,\ c_2, \ldots\ c_n, \ldots$ (Fig. 1). The second loop, inside the first one, is over the atoms of one cell, to find their nearest neighbors among atoms of that cell and its nearest neighbor cells. Because of the involved

intensive search in a large non-structured, and dynamically changing array of atoms (containing atomic coordinates and chemical type) those two loops were found inefficient when implemented with OpenACC, but benefited substantially from an OpenMP programing model and were accelerated using this model only.

The following parts (Loops 1 to 6 in Appendix B), which perform energy and force calculations, were realized within both OpenMP and OpenACC programing models, but some substantial algorithmic differences had to be introduced for both models.

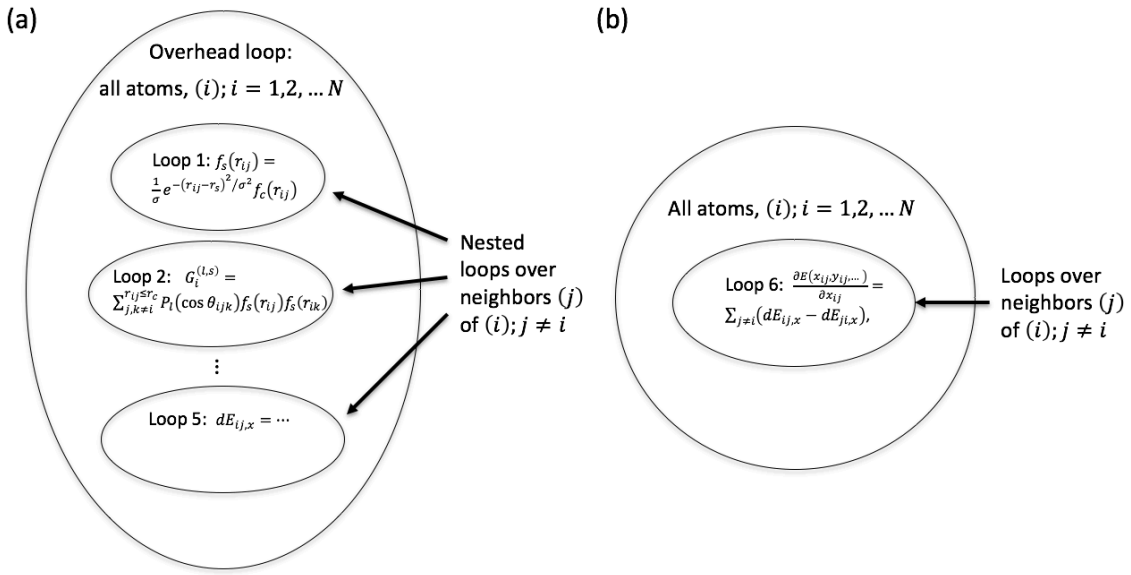


Figure 3. Intrinsic loop arrangement in the energy and force calculation algorithm implemented in Aladyn for (a) Loops 1 through 5, and (b) for Loop 6.

3.2. OpenMP multicore parallelization

The OpenMP implementation, using the ifort (2019.3.199) compiler, for energy and force calculation was done on two versions of the algorithm starting from Loop 1 to Loop 6, as defined in Appendix B. Both versions were tested and compared.

In the first, called the intrinsic version (see Fig. 3), Loops 1 through 5 were performed for each atom individually, rather than for all atoms. To perform the calculations for all atoms, those loops were combined under one overhead loop over all atoms. The benefits of this arrangement are twofold. First, because the calculations in Loops 1 to 5 involve individual atoms only, the iterations in the global overhead loop are independent of each other and the loop is parallelizable. Since this loop is the largest one over all atoms in the system, its parallelization gives good scaling with the system size or number of CPUs. Second, Loops 1 to 5 can use local temporary arrays to store the intermediate results for each atom separately. The size of these arrays is proportional to the number of neighbors

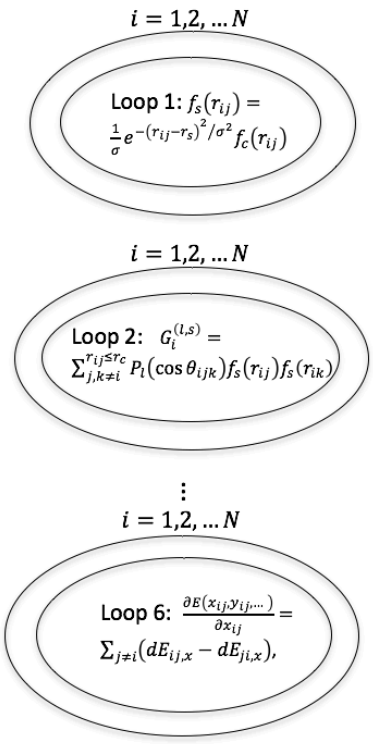


Figure 4. Extrinsic loop arrangement in the energy and force calculation algorithm implemented in Aladyn.

per atom, n_b, and remains relatively small ($n_b < 100$), which significantly improves memory management and caching. Since these arrays are used for each atom, they can be allocated at the beginning of the overhead loop and used as "private" arrays during loop parallelization. The form of Eq. (A12), does not allow Loop 6 also to be included under the overhead loop together with Loops 1 through 5. Loop 6 remains as a separate double nested loop over all atoms (Fig 3b). Since atomic stresses are not always needed in an MD simulation, Loop 6 can have two versions, one which calculates forces only (Eqs. A1(a-c)), and one which calculates forces with the stress components (Eqs. A1(a-d)). In Aladyn only the force calculations are used.

In the second, called the extrinsic version of the energy and force calculation, there is no overhead loop, and all loops from Loop 1 to 6 are iterated as nested loops over all atoms and all of the atom neighbors (Fig 4). The advantage of this arrangement is that the calculations are performed in a series of nested loops that can be collapsed* to form numerous but simple elementary loops, where each loop performs one, or a few, elementary calculations on all atoms at a time in a conveyor-like fashion. After a collapse of the nested loop in one master loop, the number of iterations in the master loop scales as $N \times n_b$, which can be on the order of 10^7 iterations or more. The simplicity of the resulted loops makes the extrinsic version most suitable for massive parallelization and vectorization. Under the OpenMP model, it was found that these loops were best parallelized using static, rather than dynamic scheduling [13,14].

A challenge with the extrinsic version is the need for large temporary arrays, that scale with the number of iterations (e.g., ~10^7 for 10^5 atoms). Another challenge is the dynamic number of neighbors of each atom. This number varies from atom to atom and from time-step to time-step. Having a different number of neighbors prevents the collapse of the inner loops over neighbors, with the outer loop over atoms. One solution is to find the maximum number of neighbors, $n_{b,max}$, at each time-step, and introduce "ghost" neighbors for atoms with less than the maximum number of neighbors. If the system density is relatively uniform, then this scheme does not lead to a large overload in terms of

* Loop collapsing is a loop transformation which combines loops that are perfectly nested into one single loop.

ghost atom calculations. It must be noted that $n_{b,max}$ can vary from time-step to time-step, and the affected arrays have to be reallocated at each time-step. Alternatively, a conservative overestimate can be made for $n_{b,max}$, and used throughout the entire simulation. In this case, array allocation can be done once at the beginning of the simulation. The drawback is that the conservative estimate (based on some maximum possible density of the simulated material) can be very large and can lead to unnecessarily large arrays, and a loss of efficiency. For systems of uniform and almost constant density (< 1% variations), such as solid body simulations, tests show that eliminating array reallocation at each time-step saves up to 20% of the CPU time. This result may differ substantially for systems with large density fluctuations, such as multiphase systems with solid-liquid or liquid-gas interfaces.

Additional optimization strategies that were tested include as follows: (i) Ensure that $n_{b,max}$ is a multiple of 8 (even if this leads to a slight overestimate of $n_{b,max}$), and instruct the compiler that it does not need to generate a remainder loop (use !DIR$ ASSUME (mod(max_nbrs,8).eq.0); (ii) Eliminate loop peeling on short loops (use !DIR$ VECTOR UNALIGNED); (iii) Force the compiler to use 256-bit Advanced Vector Extensions (AVX2) instructions, or 512-bit (AVX512) instructions [16]; (iv) Split vector arrays into scalar and reduce array dimensions when possible (e.g., instead of assigning a vector, $r(3)$, for atomic coordinates, use (x, y, z) scalars).

Figure 5 gives the results for the execution time of a 192,000 aluminum atom system for 300 time-steps using different optimization options.

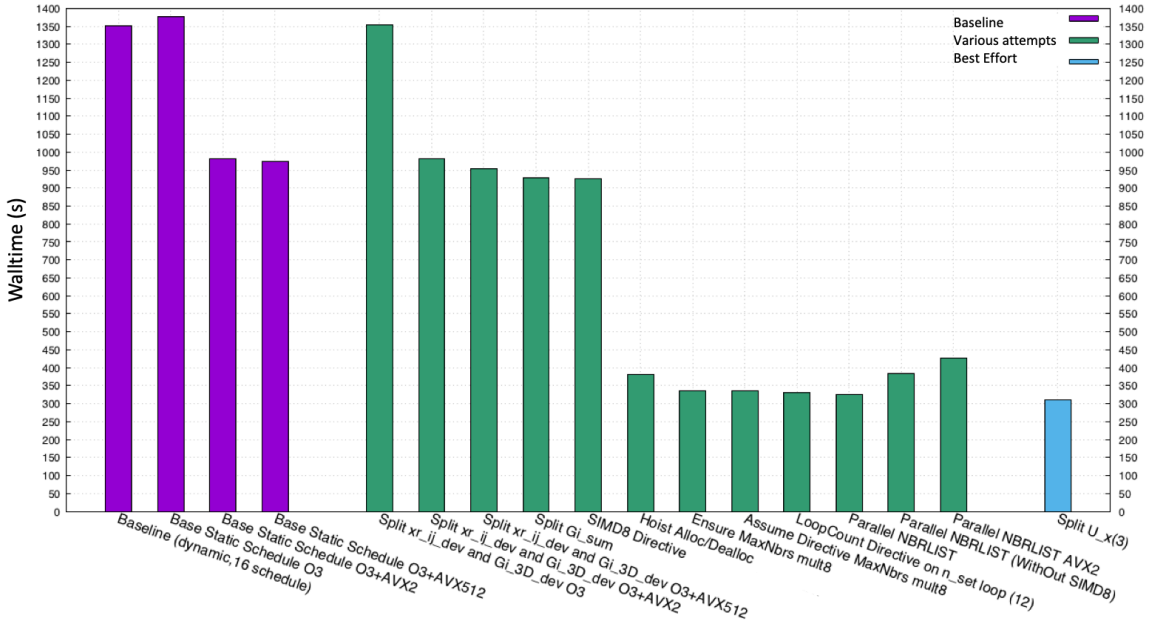


Figure 5. Results for execution time of a 192,000 atom system for 300 time-steps using different optimization options, as indicated with the explanations given in Table 1.

Optimization Option Name	Meaning
O3	Use of -O3 compiler option
Baseline (dynamic; 16 schedule)	OpenMP loops using directive SCHEDULE (DYNAMIC, 16)
Base Static Schedule	OpenMP loops using directive SCHEDULE (STATIC)
AVX2	Use of AVX2 instructions
AVX512	Use of AVX512 instructions
Split array_name	Reduce dimensions of listed arrays
SIMD8 Directive	Use of SIMD8 compiler directive where possible [16]
Hoist Alloc/Dealloc	Use global allocation/deallocation of frequently used arrays
Ensure MaxNbrs mult8	Ensure that MaxNbrs variable is a multiple of 8 to avoid generation of a remainder loop
Assume Directive MaxNbrs mult8	Instructs the compiler that it does not need to generate a remainder loop for MaxNbrs: !DIR$ ASSUME(mod(max_nbrs,8).eq.0)
LoopCount Directive on n_set loop (12)	Instruct compiler of a fixed number of loop cycles: !DIR$ LOOP COUNT (12)
Parallel NBRLIST	Parallelizing neighbor count in get_neighbors subroutine

Table 1: Definitions of the applied optimization options as listed in Fig. 5.

Some major finding with several profilers (op_scope (4.13), vtune (2019.3), and Advisor (2019.3)) after all optimizations were applied are as follows:

1. Compilation flags and directives force 512-bit instructions in critical loops (using op_scope (4.13))
2. Memory access (using vtune (2019.3))
   - Decreased NUMA access ratio: from 51% down to 15.4%
   - Very low average latency (9 cycles) indicating efficient cache utilization
   - Resource usage closer to capabilities of the hardware (DRAM peak 202GB/s of 230GB/s)
3. Microarchitecture (μarch) exploration
   - Increased Clock per Instruction (CPI) rate: from 0.416 to 0.465 (ideal is ~.25)
   - Increased Vector Capacity Usage (FPU): from 24.8% to 89.0%
   - Decrease of the Average CPU Frequency: from 3.045 GHz to 2.339 GHz

(Nominal: 2.200 GHz for AVX512)

4. Threading
   - Increased effective CPU utilization factor from 33.3 out of 80 logical CPUs to 38.4, or 47.9% of CPU usage.
   - Decreased serial time (outside parallel regions) from 16.5% to 1.3% of CPU Time
   - Decreased spin and overhead time from 5.8% to 2.7% of CPU Time
5. Compute efficiency: 1159 GFLOP/s (~40% of DP peak) (using Advisor (2019.3))

A comparison between the Intrinsic and Extrinsic versions of the algorithm showed that there was no noticeable difference in the performance speedup between the two versions. The major difference was in the substantially large DRAM usage of the Extrinsic version of ~23 GB for 192,000 atoms compared to ~750 MB for the Intrinsic version. Similarly, the DRAM bandwidth was much higher for the Extrinsic version compared to the Intrinsic version (~73 GB/s vs 2.3 GB/s, respectively). Thus, because the Intrinsic version allows for a much bigger system size to be simulated on one compute node with less bandwidth overload, it is the preferable version for multicore execution, compared to the Extrinsic version.

3.3. OpenACC for GPU acceleration

The OpenACC implementation of Aladyn used only the extrinsic version (Fig. 4) of the energy and force calculation, because it allowed for much more efficient massive parallelization on a GPU device. Some notable differences in the algorithm implementation have been found when optimizing for GPU-accelerator using OpenACC model, compared to the multicore OpenMP model optimization. For example, the intrinsic version was very inefficient on a GPU device, while on the multicore architecture, the intrinsic and extrinsic versions were found to be equally efficient in terms of speed. The inefficiency for the GPU device was found to be due to the extensive use of too many registers. The dynamic array allocation, where temporary arrays were created on the GPU device at each time-step was found to have no performance impact compared to the global allocation at the beginning of the simulation. This is in contrast to the multicore OpenMP results, where the initial global array allocation was found to be substantially more efficient.

The OpenACC code does not spend much time in memory transfer between GPU and CPU. We therefore focused our effort on optimizing the parallel execution of the individual kernels. Of a particular importance was the finding that in the calculation of the LSPs coefficients, $G_i^{(l,s)}$, of atom (i) in Loop 2 (Eq. 1), a collapse directive of a double (j,k)-loop over the same set of neighbors caused strided memory access for the k-dimension

due to the collapse of the (j,k)-indices into one. Eliminating the collapse directive resulted in a speed up between 20% and 30% of the overall application. The loop itself sped up by a factor of 7.

Metric Name	Original Loop 2	Optimized Loop 2
achieved_occupancy	0.624169	0.374456
stall_exec_dependency	43.33%	29.16%
flop_dp_efficiency	7.14%	41.77%
gld_transactions	8224805084	483291812
gst_transactions	3840000	3840000
l2_read_transactions	51709034	50929534
l2_write_transactions	3981272	4007482
dram_read_transactions	14353149	14337805
dram_write_transactions	1186837	1212392
flop_count_dp	79724544000	67928064000
l2_write_throughput	856.234670 MB/s	5.782475 GB/s
l2_read_throughput	10.859982 GB/s	73.495880 GB/s
dram_read_throughput	3.012424 GB/s	20.687079 GB/s
dram_write_throughput	255.235570 MB/s	1.749335 GB/s

Table 2: Performance statistics for Loop 2 before and after removing the inner loop collapse. Description of the listed metrics is given in Table 2, following Ref. [17].

To understand the performance difference, we looked into some important performance statistics of the original and modified code. The statistics are shown in Table 2. A contention on the floating point (FP) units was noticed, which contributed to low performance of the original code. This was indicated by a high percentage of stalls due to execution dependencies, most likely the FP units. Also, the modified code contains far fewer double precision (DP) add operations and much greater FP efficiency. The overall instruction count is much lower and there is much better L2 cache utilization. In general, removing the inner loop collapse allowed the compiler to generate more efficient code and exploit the GPU more effectively.

Metric Name	Description
achieved_occupancy	Ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
stall_exec_dependency	Percentage of stalls occurring because an input required by the instruction is not yet available

flop_dp_efficiency	Ratio of achieved to peak double-precision floating-point operations
gld_transactions	Number of global memory load transactions
gst_transactions	Number of store memory load transactions
l2_read_transactions	Memory read transactions seen at L2 cache for all read requests from L1 cache.
L2_write_transactions	Memory read transactions seen at L2 cache for all write requests from L1 cache.
Dram_read_transactions	Device memory read transactions.
Dram_write_transactions	Device memory write transactions.
Flop_count_dp	Number of double-precision floating-point operations executed by non-predicated threads (add, multiply and multiply-accumulate). Each multiply-accumulate operation contributes 2 to the count.
L2_write_throughput	Memory write throughput seen at L2 cache for all write requests
l2_read_throughput	Memory write throughput seen at L2 cache for all read requests
dram_read_throughput	Device memory read throughput
dram_write_throughput	Device memory write throughput

Table 3: Description of the metrics used in Table 2, following Ref. [17].

A complete description of all available metrics is given in Ref. [16]. For convenience we provide a subset in Table 3. Further performance improvements were obtained via array transpositions and added vectorization in Loop 3, reducing array dimensions, and adding vectorization with some other minor modifications. Hoisting allocation/deallocation of memory to an outer level, as in the multi-core version was also implemented here. This however did not yield any noticeable speed-up. Trying to reduce register pressure in Loop 3, by splitting up the loop into several smaller loops by splitting the calculation of the sum in Eq. A10 in parts for the two terms containing $P_l(\cos\theta_{ijk})\dot{f}_s(r_{ij})$, and $\frac{f_s(r_{ij})}{r_{ij}}\dot{P}_l(\cos\theta_{ijk})$, respectively, actually decreased the performance.

4. Summary

As part of its educational effort with NASA Langley Research Center, the High-End Computing Capability's (HECC) Applications Performance and Productivity (APP) team at NASA Advanced Supercomputing (NAS) Division improved the performance of the Aladyn miniapp by a factor of 4.7 for multi-core CPU, and a factor of 2.4 for GPU

execution. The APP team achieved the CPU speedup on a Skylake 6154 CPU with 40 threads by: (i) moving allocate/deallocate statements to an outer level, (ii) using Intel compiler directives to increase the use of avx512 instructions, and (iii) changing array layouts to speed memory accesses. The GPU speedup was achieved by increasing vectorization and applying array layout changes.

Appendix A

Analytic Force Calculation for ANN Potential

Equations (1) through (9) define the total potential energy of an atomistic system, $E = \sum_i E_i$, as expressed through the relative interatomic coordinates $\{x_{ij}, y_{ij}, z_{ij}, \ldots\}$ between pairs of atoms within the cut-off distance, $r_{ij} \le r_c$. The components of the force, $\boldsymbol{F}_i = (F_{xi}, F_{yi}, F_{zi})$, acting on atom (i) are given as spatial derivatives of E, which for the x-component is:

$$F_{xi} = -\frac{\partial E(x_{ij}, y_{ij}, \ldots)}{\partial x_i} = -\sum_{j\neq i} \frac{\partial E(x_{ij}, y_{ij}, \ldots)}{\partial x_{ij}} \frac{\partial x_{ij}}{\partial x_i} = \sum_{j\neq i} \frac{\partial E(x_{ij}, y_{ij}, \ldots)}{\partial x_{ij}} = \sum_{j\neq i} F_{x,ij}. \quad \text{(A1a)}$$

Note that due to the definition, $x_{ij} = x_j - x_i$, $\frac{\partial x_{ij}}{\partial x_i} = -1$.

Similarly, the other force components are given as

$$F_{yi} = -\frac{\partial E(x_{ij}, y_{ij}, \ldots)}{\partial y_i} = \sum_{j\neq i} \frac{\partial E(x_{ij}, y_{ij}, \ldots)}{\partial y_{ij}} = \sum_{j\neq i} F_{y,ij}, \quad \text{(A1b)}$$

$$F_{zi} = -\frac{\partial E(x_{ij}, y_{ij}, \ldots)}{\partial z_i} = \sum_{j\neq i} \frac{\partial E(x_{ij}, y_{ij}, \ldots)}{\partial z_{ij}} = \sum_{j\neq i} F_{z,ij}. \quad \text{(A1c)}$$

Using Virial stress formulation [10], the atomic stress can be calculated as:

$$\sigma_{\alpha\beta} = \frac{1}{V}\sum_{i\in V} \sigma_{\alpha\beta}^{(i)} = \frac{1}{2N\Omega}\sum_{i\in V}\left(m_i v_\alpha^{(i)} v_\beta^{(i)} - \sum_{j\neq i} \frac{\partial E}{\partial \alpha_{ij}} \beta_{ij}\right) \quad \text{(A1d)}$$

where α, β stand for x, y, or z Cartesian coordinates, and Ω is the system volume.

Since $E(x_{ij}, y_{ij}, \ldots)$ is a complex function build of all the functions given by Eqs. (1) through (8) as

$$E(x_{ij}, y_{ij}, \ldots) = \sum_i E_i = \sum_i \boldsymbol{u}^{(n)}(i) = \sum_i \boldsymbol{u}^{(n)}\left(\boldsymbol{u}^{(n-1)}\left(\ldots\left(\boldsymbol{u}^{(1)}\left(G_i^{(l,s)}(r_{ij}, r_{ik}, \ldots)\right)\right)\right)\right), \quad \text{(A2)}$$

its partial derivatives are obtained through the chain rule

$$\frac{\partial E(x_{ij},y_{ij},\dots)}{\partial x_{ij}} = \sum_k \frac{\partial E_k}{\partial x_{ij}} = \sum_k \frac{\partial \boldsymbol{u}^{(last)}(k)}{\partial x_{ij}} = \sum_k \frac{\partial \boldsymbol{u}^{(last)}}{\partial \boldsymbol{u}^{(last-1)}} \dots \frac{\partial \boldsymbol{u}^{(1)}}{\partial G_k} \frac{\partial G_k}{\partial x_{ij}}, \quad \text{(A3)}$$

where the summation is over all atoms (k), including $k = i$, and $k = j$.

Solving (A3) would be easier if Eqs. (6a) and (6b) are presented by components:

$$u_p^{(1)}(i) = G_i^{(q)} w_{q,p}^{(0,1)} + b_p^{(1)} \quad \text{(A4a)}$$

$$u_p^{(n)}(i) = f\left(u_q^{(n-1)}(i)\right) w_{q,p}^{(n-1,n)} + b_p^{(n)}; \quad n > 1, \quad \text{(A4b)}$$

where the Einstein summation convention over repeated indices is assumed.

Differentiating Eq. (A4b) gives the iteration equation for the derivatives of a neural network's layer (n)

$$\frac{\partial u_p^{(n)}(k)}{\partial x_{ij}} = \dot{f}\left(u_q^{(n-1)}(k)\right)\left(\frac{\partial u_q^{(n-1)}(k)}{\partial x_{ij}}\right) w_{q,p}^{(n-1,n)}. \quad \text{(A5)}$$

When enrolled from the *n*-th layer down to the first layer, one gets:

$$\frac{\partial u_{m_n}^{(n)}(k)}{\partial x_{ij}} = \dot{f}\left(u_{m_{n-1}}^{(n-1)}(k)\right)\dot{f}\left(u_{m_{n-2}}^{(n-2)}(k)\right)\left(\frac{\partial u_{m_{n-2}}^{(n-2)}(k)}{\partial x_{ij}}\right) w_{m_{n-2},m_{n-1}}^{(n-2,n-1)} w_{m_{n-1},m_n}^{(n-1,n)}$$

$$\dots$$

$$\frac{\partial u_{m_n}^{(n)}(k)}{\partial x_{ij}} =$$

$$\dot{f}\left(u_{m_{n-1}}^{(n-1)}(k)\right)\dot{f}\left(u_{m_{n-2}}^{(n-2)}(k)\right)\dots\dot{f}\left(u_{m_1}^{(1)}(k)\right)\left(\frac{\partial G_k^{(m_0)}}{\partial x_{ij}}\right) w_{m_0,m_1}^{(0,1)} \dots w_{m_{n-2},m_{n-1}}^{(n-2,n-1)} w_{m_{n-1},m_n}^{(n-1,n)}. \quad \text{(A6)}$$

Consequently, Eq. (A3) becomes (noting that the last layer has only one element, $\frac{\partial u_1^{(n)}(k)}{\partial x_{ij}} = \frac{\partial E_k}{\partial x_{ij}}$):

$$\frac{\partial E_k}{\partial x_{ij}} =$$

$$\dot{f}\left(u_{m_{n-1}}^{(n-1)}(k)\right)\dot{f}\left(u_{m_{n-2}}^{(n-2)}(k)\right)\dots\dot{f}\left(u_{m_1}^{(1)}(k)\right)\left(\frac{\partial G_k^{(m_0)}}{\partial x_{ij}}\right) w_{m_0,m_1}^{(0,1)} \dots w_{m_{n-2},m_{n-1}}^{(n-2,n-1)} w_{m_{n-1},1}^{(n-1,1)}, \quad \text{(A7)}$$

where

$$\dot{f}(u) = f(u)\big(1 - f(u)\big) = \frac{e^{-u}}{(1+e^{-u})^2}\,. \tag{A8}$$

Differentiating Eqs. (1-5), one gets for a particular $m_0 = (l, s)$ combination:

$$\frac{\partial G_k^{(m_0)}}{\partial x_{ij}} = \frac{\partial G_k^{(l,s)}}{\partial x_{ij}} = g_{x,\,kj}^{(l,s)}\delta_{ki} - g_{x,\,ki}^{(l,s)}\delta_{kj}, \tag{A9}$$

where δ_{ij} is the Kronecker delta symbol, $\delta_{ij} = 1$ if $i = j$, and 0 otherwise, and

$$g_{x,\,ij}^{(l,s)} = \\ 2\sum_{k\neq i}^{incl.\ k=j} f_s(r_{ik})\left[P_l\big(\cos\theta_{ijk}\big)\dot{f}_s\big(r_{ij}\big)\frac{x_{ij}}{r_{ij}} + \frac{f_s(r_{ij})}{r_{ij}}\dot{P}_l\big(\cos\theta_{ijk}\big)\left(\frac{x_{ik}}{r_{ik}} - \frac{x_{ij}}{r_{ij}}\cos\theta_{ijk}\right)\right], \tag{A10}$$

with

$$\dot{P}_l(x) = \big[(2l+1)\big(P_l + x\dot{P}_l\big) - l\dot{P}_{l-1}\big]/(l+1)\,; \quad \dot{P}_0(x) = 0; \quad \dot{P}_1(x) = 1. \tag{A11}$$

After inserting Eq. (A9) into (A7), the final form for the gradient of the total energy can be expressed as a sum of two terms,

$$\frac{\partial E\big(x_{ij}, y_{ij}, \ldots\big)}{\partial x_{ij}} = \sum_{j\neq i}\big(dE_{ij,x} - dE_{ji,x}\big), \tag{A12}$$

where

$$dE_{ij,x} = \\ \dot{f}\left(u_{m_{n-1}}^{(n-1)}(i)\right)\dot{f}\left(u_{m_{n-2}}^{(n-2)}(i)\right)\ldots\dot{f}\left(u_{m_1}^{(1)}(i)\right)w_{m_0,m_1}^{(0,1)}\ldots w_{m_{n-2},m_{n-1}}^{(n-2,n-1)}w_{m_{n-1},1}^{(n-1,1)}g_{x,\,ij}^{(l,s)}. \tag{A13}$$

The form of Eq. (A12) guarantees that the force, predicted by the neural network will satisfy Newton's third law: $\frac{\partial E}{\partial x_{ij}} = -\frac{\partial E}{\partial x_{ji}}$.

Equation (A13) represents another ANN, defined as

$$U_{m_1}^{(1)}(ij, x) = g_{x,\,ij}^{(m_0)}w_{m_0,m_1}^{(0,1)} \tag{A14a}$$

$$U_{m_n}^{(n)}(ij, x) = \dot{f}\left(u_{m_{n-1}}^{(n-1)}(i)\right)U_{m_{n-1}}^{(n-1)}(ij, x)w_{m_{n-1},m_n}^{(n-1,n)}; \quad n > 1, \tag{A14b}$$

in which the weights, $w_{m_{n-1},m_n}^{(n-1,n)}$, are the same as in the non-differentiated ANN, but there

is no bias term, $b_{m_n}^{(n)}$, and the transfer function is a multiplication with a constant, equal to the derivative of the transfer function from the respective layer of the non-differentiated ANN for atom (i). In addition, the input for this ANN are the derivatives of the LSPs, with respect to the pair distances, (x_{ij}, y_{ij}, z_{ij}), which makes a total of $3\,N(N-1)/2$ different ANNs to be computed to get the forces in a system of Natoms, rather than only N different ANNs for the energy calculation.

Appendix B

Computational Implementation of Force Calculation

Efficient calculation of forces for a fast MD simulation depends significantly on the way calculations for the spatial derivatives, given in Appendix A are organized and performed in an HPC code. The following describes how these calculations are implemented in Aladyn.

There are two parts in the force calculation. The first part is to search and identify all neighbors of an atom inside the interaction range, r_c. This is performed in subroutine get_neighbors in aladyn.f source file. The second part is the actual force calculation using the equations in Appendix A. These calculations are performed in subroutine Frc_ANN_OMP and in Frc_ANN_ACC in the aladyn_ANN.f file. All of the calculations are organized in a series of loops, which will be described here.

Nearest neighbor search.

Loop 0: For all atoms (*i*) identify and store their neighbors, (*j*), at a distance $r_{ij} \leq r_c$. The search for neighbors is performed using the link-cell technique, as described in Sec. 2.

Calculation of LSPs.

Loop 1: For all (*i,j*)-pairs, calculate and store $f_s(r_{ij}); s = 1,2,..12$, from Eq. (2).

Loop 2: For all atoms (*i*) use double loops over their neighbors, (*j*) and (*k*), to calculate and store the LSPs, $G_i^{(l,s)}$; $(l = 0,1,2,4,6;\ \ s = 1,2,..\ 12)$, from Eq. (1), using the pre-calculated $f_s(r_{ij})$ from Loop 1.

Loop 3: For all (*i,j*)-pairs, calculate and store $g_{x,ij}^{(l,s)}$, $g_{y,ij}^{(l,s)}$, and $g_{z,ij}^{(l,s)}$, from Eq.

(A10).

Energy calculation.

Loop 4: For all atoms (i) use ANN from Eqs.(A4a) and (A4b) (or Eqs. 6a, 6b) to calculate the potential energy of atom (i), E_i, from Eq. (8). In addition, calculate and store the derivative of the transfer function, $\dot{f}\left(u_{m_n}^{(n)}(i)\right)$ (Eq. A8), at each node, m of layer $n > 1$ of the ANN. Get the total potential energy of the system, $E = \sum_i E_i$, as a sum of all $E_{i=1,..N}$ (Eq. 9).

Force calculation.

Loop 5: For all (i,j)-pairs, start the pair ANNs, defined by Eqs. (A14a) and (A14b) to calculate and store $dE_{ij,x}$, $dE_{ij,y}$, and $dE_{ij,z}$, from Eq. (A13), using the pre-calculated $g_{x,\,ij}^{(l,s)}$, $g_{y,\,ij}^{(l,s)}$, and $g_{z,\,ij}^{(l,s)}$ from Loop 3, and $\dot{f}\left(u_{m_n}^{(n)}(i)\right)$ from ANN 1.

Loop 6: For each atom (i), use a loop over all its neighbors (j) to calculate the pair forces, $\frac{\partial E}{\partial x_{ij}}$, $\frac{\partial E}{\partial y_{ij}}$, $\frac{\partial E}{\partial z_{ij}}$, from Eq. (A12), and get the total force vector $\left(F_{xi}, F_{yi}, F_{zi}\right)$, acting on atom ($i$) from Eqs (A1a-c). If needed, use $\frac{\partial E}{\partial x_{ij}}$, $\frac{\partial E}{\partial y_{ij}}$, and $\frac{\partial E}{\partial z_{ij}}$, calculated in this loop to get the atomic stress components, $\sigma_{\alpha\beta}^{(i)}$; $(\alpha, \beta = x, y, z)$, for each atom (i), and the total stress of the system, $\sigma_{\alpha\beta}$. (Eq. (A1d)

References

[1] Lejaeghere, K., et al., "Reproducibility in Density Functional Theory Calculations of Solids", Science 351 (2016) aad3000-1-7.

[2] Brenner, D. W., "The Art and Science of an Analytical Potential", Phys. Stat. Sol. (b) 217, (2000) 23-40.

[3] Niethammer, C., Glass, C. W., Bernreuther, M., Becker, S., Windmann, T., Horsch, M. T., Vrabec, J., Eckhardt, W., "Innovative HPC Methods and Application to Highly Scalable Molecular Simulation (IMEMO)", In Inside - Innovatives Supercomputing in Deutschland, Volume 10(1), April 2012.

[4] Mueller, T., Kusne. A. G., Ramprasad, R., "Machine Learning in Materials Science: Recent Progress and Emerging Applications", in: Parrill, A. L., Lipkowitz, K.B. (Eds.), Reviews in Computational Chemistry, 29, Wiley (2016) 186-273.

[5] Behler, J., Parrinello, M., "Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces", Phys. Rev. Lett. 98 (2007) 146401-1-4.

[6] Behler, J., "Perspective: Machine Learning Potentials for Atomistic Simulations", J. Chem. Phys. 145 (2016) 170901-1-9.

[7] Yamakov, V., Glaessgen E. H., "Aladyn – Adaptive Neural Network Molecular Dynamics Simulation Code: Coputational Materials Mini-Application", NASA/TM-2018-220104. (https://github.com/nasa/aladyn)

[8] Frenkel, B., Smit, B., "Understanding Molecular Simulation", Academic Press, London, (2001).

[9] Pun, G. P. P., Batra, R., Ramprasad, R., Mishin, Y., "Physically Informed Artificial Neural Networks for Atomistic Modeling of Materials", NATURE COMMUNICATIONS https://doi.org/10.1038/s41467-019-10343-5.

[10] Cormier, J., Rickman, J. M., Delph, T. J., "Stress Calculation in Atomistic Simulations of Perfect and Imperfect Solids", J. Appl. Phys. 89 (2001) 99-104.

[11] Gear, C. W., "The Numerical Integration of Ordinary Differential Eq.s of Various Orders", Technical Report ANL 7126 (1966) Argonne National Laboratory, Argonne, IL.

[12] Schlick, T., Skeel, R. D., Brunger, A. T., Kale, L. V., Hermans, J., Schulten, K., "Algorithmic Challenges in Computational Molecular Biophysics", J. Comp. Phys. 151 (1999) 9-48.

[13] OpenMP Home Page, https://www.openmp.org/

[14] Gerber, R., "Getting Started with OpenMP", Intel Software Developer Zone, https://software.intel.com/en-us/articles/getting-started-with-openmp.

[15] OpenACC Home Page, https://www.openacc.org/

[16] Intel Developer Zone, Intel®AVX, https://software.intel.com/en-us/isa-extensions/intel-avx

[17] Nvidia Profiler Users Guide, https://docs.nvidia.com/cuda/profiler-users-guide/index.html#metrics-reference.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*
1-10-2019

2. REPORT TYPE
Technical Memorandum

3. DATES COVERED *(From - To)*

4. TITLE AND SUBTITLE

High-Performance Computing Optimization for Aladyn-Adaptive Neural Network Molecular Dynamics Mini-Application

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Yamakov, Vesselin I.; Jost, Gabriele; Kokron, Daniel S.; Mishin, Yuri; Glaessgen, Edward, H.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER
698259.02.07.07.03.01

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-21058

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)
NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
NASA-TM-2019-220409

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-
Subject Category 24
Availability: NASA STI Program (757) 864-9658

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report provides a description and performance evaluation of the optimization techniques for high performance computing (HPC) implementation of the open source Computational Materials mini-application Aladyn (https://github.com/nasa/aladyn). Aladyn is a basic molecular dynamics code written in FORTRAN 2003, which is designed to demonstrate the use of adaptive neural networks (ANNs) in atomistic simulations. The role of ANNs is to efficiently reproduce the very complex energy landscape resulting from the atomic interactions in materials with the accuracy of the more expensive quantum mechanics-based calculations.

15. SUBJECT TERMS

High performance computing; atomistic simulation; metal alloy; moleuclar dynamics

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
U	U	U	UU	25	19b. TELEPHONE NUMBER *(Include area code)* (757) 864-9658