SCC Summer 2017: LAMMPS Scalability and Publication Replication

Ryan Prendergast, Alexander Ballmer

August 11, 2017

1 Abstract

The Student Cluster Competition provides five specific applications to run over a period of 48 hours. One of the applications is the Large Scale Atomic/Molecular Massively Parallel Simulator (LAMMPS), a molecular dynamics simulator.

This paper outlines our work over the summer to optimize LAMMPS: we sought to determine the compilation and runtime parameters which maximize efficiency. Tested parameters include compilation flags, neighbor list vectorization, per-core scalability through MPI ranks and OpenMP threads, and multi-core scalability.

2 LAMMPS Overview

2.1 Molecular Dynamics Simulations

LAMMPS is a molecular dynamics (MD) simulator, meaning it simulates the interactions between particles in a two or three dimensional space. The term "particle" here refers to any structure that can be represented by a one-dimensional point, not the elementary particles in a particle physics experiment. The particles in a real world simulation could represent anything, from fish to hair to atoms.

MD simulators provide a wide variety of applications, such as chemistry, bioinformatics, and materials science. LAMMPS specifically was written in the 1990s as a joint cooperative between three companies and two national laboratories to provide a highly parallelizable solution for extremely large simulations. The original program was written in Fortran 77 but was later rewritten in C++. It supports a system of plugins, fixes, and extensions to add new types of atoms and interactions between atoms. [3]

Open Source: The associated LAMMPS code is available at [2].

2.2 LAMMPS Computation

Each particle in a molecular dynamics simulation has specific properties associated with it. The particular types of data in a given simulation depend on which packages LAMMPS loads, but each particle typically has coordinates, velocity, and a set of functions defining how it interacts with other particles. Interactions between particles apply net forces, and the velocity and position can be updated according to Newtonian mechanics.

LAMMPS is designed to trace the interactions between thousands to billions of atoms. To do this, it must effectively parallelize over large core counts and many nodes without saturating the network bandwidth.

LAMMPS achieves this scalability through spatially decomposed "neighbor lists", which partition particles into largely autonomous groups such that the particles within a list have a high probability of interacting. A neighbor list N_i for a particle i, then, consists of all surrounding atoms j within a cutoff distance r_c, or

$$N_i = \{j : r_{ij} < r_c, j \neq i\}$$

Neighbor lists allow data locality and a heavy reduction in MPI cross-talk between separate ranks. Without a bottleneck in MPI, LAMMPS becomes purely compute bound. [4]

2.3 Outline of one Timestep

In general, a LAMMPS simulation consists of five steps.

1. Set initial positions r_0, velocities v_0, accelerations a_0, and time t_0.
2. For each atom i, compute the interatomic potential V_i with all neighboring atoms.
3. Update force F_i for each atom i by differentiating the interatomic potential with respect to distance (force is derivative of energy).
4. Using this force, solve Newtonian mechanics and update mutable properties like position and velocity.
5. Repeat for each timestep $t_i < t_{max}$

LAMMPS recomputes the total interatomic potential V every timestep until the end of the simulation. This computation involves a double summation: over each atom i and over each atom j in N_i, as shown bellow.

$$V = \sum_i \sum_{j \in N_i} \phi$$

The ϕ term is unique to each simulation; it can be a two-body potential $\phi(r_{ij})$ that depends exclusively on relative distance, or a multi-body potential which involves three or more particles.

2.4 Input

The LAMMPS source consists of a series of hierarchical subprograms represented as C++ classes. This design allows modularity and forms the basis of input scripts. To accept input, LAMMPS can run either as an interpreter, taking commands one by one, or read from a prewritten file. A single input command take the general form shown below.

input_parameter_1 value_1

For example, the below command sets the pair style of a simulation to the Tersoff multibody.

pair_style tersoff

The modular nature of LAMMPS means there typically is a one-to-one ratio of input parameters to C++ source files, and that adding new parameters is a matter of including their corresponding source files.

2.4.1 Input Parameters

To ensure comparable results across tested paramters, we used the in.si-bulk input file for all tests. This simulation creates a lattice of silicon atoms which interact according to the Tersoff multi-body potential. Some of the most important input paramters are explained below.

- Units: metal. This defines the style of units gives mass in g/M, distance in Angstroms, time in pS, energy in eV, etc
- Atom Style: atomic. Declares the particle type, in this case an interaction between atoms, and which attributes will be associated.
- Pair_style: tersoff. Formula used for interatomic potential calculations.
- Pair_coefficients: Si.tersoff. Specify the file containing coefficients for the particular potential calculation.
- Timestep: .0005. How often (in time set by units) to recalculate interatomic potentials.
- Run: 350. How much time (in time set by units) to run the simulation.

2.5 Compilation

LAMMPS includes a system of prewritten makefiles for a variety of supported CPU architectures, GPUs, and coprocessors.

Because LAMMPS was written to be modular, it relies heavily on packages. Many input files or makefiles require source dependencies that are not included by default; these packages are enabled individually. The command make yes-PACKAGE compiles and attaches that particular package to the binary.

Before compiling LAMMPS, we enabled three packages: USER-OMP, MANYBODY, and USER-INTEL. USER-OMP is required for the intel_cpu makefile and provides support for OpenMP threading. USER-INTEL, also required for make intel_cpu, provides Intel specific processor optimizations and allows variable precision. The MANYBODY pacakge supports multi-body potentials like Tersoff, which is necessary for the in.si-bulk input file.

2.5.1 intel_cpu Platform

We compiled LAMMPS for the intel_cpu platform, using -g for debugging and the -O2 optimization level. Some of the most important other compiler flags are explained below.

- -Xhost: Use the highest (AVX2) vector instruction set.
- -fp-model fast=2: Setting 2 enables aggressive floating point optimizations at the cost of accuracy.
- -no-prec-div: Enables optimizations that give less precise division results.

2.5.2 Further Compiler Optimizations

With highly aggressive compiler optimizations already set, we wanted to test whether further optimizing would decrease the runtime of the program. We tested the runtime of two binaries: one with -O3 and the other -O2. The binaries and input files were otherwise identical.

After running the test over two trials, we found an average runtime of 31:30 minutes for the O3 and 30:47 minutes for the O2 binary. The higher optimization was actually negligably detrimental to the runtime; for future tests we used O2.

3 Serial Run

The run conditions for the serial run consisted of a Dell Poweredge R630 computer with a Intel Xeon E5-2670 v3 processor. We used a single MPI rank and did not set OMP_NUM_THREADS. The input file was in.si-bulk and the precision was set to mixed.

Table 1 shows the VTune profiling results of a serial run.

Table 1: VTune Hotspots

Function	Time (s)
IntelKernelTersoff:kernal	1447.521
IntelKernelTersoff:kernal step	51.750
lmp intel:vector_ops	33.169
LAMMPS NS	30.720

Unsurprisingly, considering there was one MPI rank, LAMMPS spends the vast majority of its time in compute functions and very little in any MPI communication.

Allinea perf-report confirmed this finding, determing the serial run to be 99.9% compute bound and 0.1% MPI bound.

4 Tersoff Multi Body Potential

The specific task for using LAMMPS in the Student Cluster Competition is to replicate the results of a paper, "The Vectorization of the Tersoff Multi-body Potential: An Exercise in Performance Portability".

4.1 Overview

The paper we are supposed to replicate, , implements the Tersoff potential. The Tersoff potential is a many-body potential which means that it is composed of three or more potentials. As opposed to a pairwise potential (two potentials) like the Coulomb potential, the Tersoff manybody potential is more accurate in modeling the interaction of atoms because it includes calculating the neighbor lists of two atoms. This means that the Tersoff potential not only calculates the distance between two atoms but also the relative distance between the neighbor atoms and the angle between the three atoms.

The algorithmic complexity of the Tersoff potential is $\mathcal{O}(n^3)$. because there are 3 summations in the mathematical equation of the Tersoff potential which translates into 3 for loops I,J, and K. The first loop I goes over all the atoms, the second loop J goes over the neighbor list, and the third loop K goes over the same neighbor list.

Open Source: The associated Tersoff code is available at [1].

4.2 Replication of Experiment

The conditions to replicate the experiment were the same as the conditions for the serial run.

The LAMMPS vectorization experiment is replicated through a python and bash script based framework that automatically launches jobs. Replicating the experiment consists of compiling the binaries with set precisions, starting the scripted test runs, and reducing the data to an SQLite database for analysis.

We found that using single and mixed precision with vectorization decreased the wall time. However, using double precision with vectorization actually increased the wall time. These results were consistent with the published paper. We also saw that vectorizing the I loop and parallelizing the J loop versus the other way around does not significantly affect runtime. We hypothesize that the very large numbers of neighbors (atoms within a cutoff range of surrounding atoms) likely oversaturates the vector units.

5 MPI Strong Scaling

The conditions of the strong scaling runs were the same as the conditions for the serial run. Scaling was done on a single node up to the total number of hardware threads, and on multiple nodes. The input file was in.si-bulk, replicated to give 30 minutes of runtime when run in serial. Data was collected on power usage and runtime.

5.1 Single Node

We scaled the application on a single node from a serial run up to 48 threads, equal to the number of hardware threads on the node.

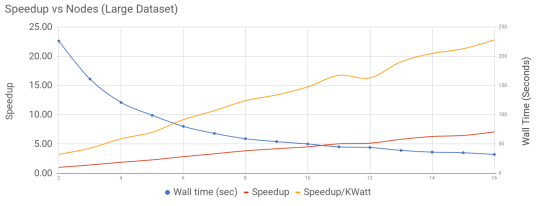
The fastest runtime on a single node as at 24 cores, with a 90 second runtime. After the number of MPI ranks exceeded the number of physical cores, runtime started to increase.

Power usage followed a similar pattern, increasing until usage hit 24 ranks, and then decreasing slowly after 24 ranks up to 48. This could be that LAMMPS was less efficient at running using MPI and hardware threads than MPI and physical cores.

From all of this data, we found that the maximum speedup occurred around 24 cores, as MPI does not seem to use hardware threads as efficiently as physical cores.

5.2 Multi Node

Strong scaling runs were run using the same si-bulk input file as the single node scaling runs. Scaling was done by adding nodes with 48 ranks per node. LAMMPS scales very well across multiple nodes, close to logarithmically. (Figure 5.2).



[t]

5.3 Allinea Perf-Report

We used the Allinea profiling software to generate an automated report of LAMMPS running at 24 cores on a single node. Allinea's report showed that LAMMPS is

purely CPU bound, spending very little time in either IO or MPI.

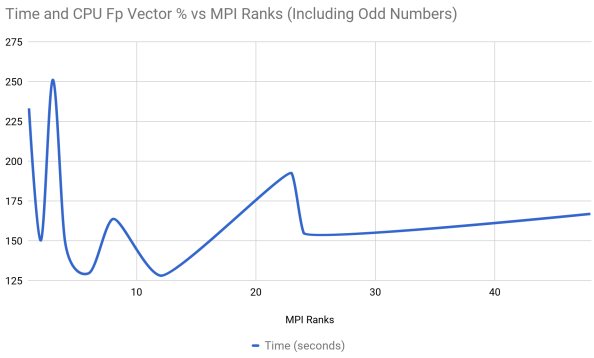
6 MPI/OpenMP Strong Scaling

For both single and multi node experiments, we used the Chameleon compute node hardware and ran our experiments on bare metal provisioning. The input script we used was in.si-bulk and we ran up to 48 threads to maximize performance. We split the test cases into odd numbers and even numbers. The final result was averaged over two trials.

6.1 Single Node

We scaled the application with 2 odd number MPI ranks and OpenMP threads and the other 8 trials with even numbers to test the hypothesis that odd numbers will result in decreased performance.

We found that LAMMPS ran best with 6 or 12 MPI ranks. With 6 ranks, it was strongly compute bound. We believe this is because LAMMPS conflicts with NUMA nodes.



6.2 Allinea Perf-Report

In this report, we found that LAAMPS is 96.5% compute bound, 3.4% MPI, and 0.1% I/O. Thus, it is mainly CPU bound.

7 Acknowledgements

Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation

References

[1] https://github.com/hpac/lammps-tersoff-vector.

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

[3] Paul Stewart Crozier. A brief survey of the lammps particle simulation code: introduction case studies and future development. Technical report, Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2011.

[4] Markus Höhnerbach, Ahmed E. Ismail, and Paolo Bientinesi. The vectorization of the tersoff multi-body potential: An exercise in performance portability. *CoRR*, abs/1607.02904, 2016.