Performance Analysis and Optimization of the RAMPAGE Metal Alloy Potential Generation Software

Philip C. Roth
Oak Ridge National Laboratory
Oak Ridge, Tennessee USA
rothpc@ornl.gov

Hongzhang Shan
Lawrence Berkeley National Laboratory
Berkeley, California USA
hshan@lbl.gov

David Riegner
The Ohio State University
Columbus, Ohio USA
driegner@gmail.com

Nikolas Antolin
The Ohio State University
Columbus, Ohio USA
nantolin@gmail.com

Sarat Sreepathi
Oak Ridge National Laboratory
Oak Ridge, Tennessee USA
sarat@ornl.gov

Leonid Oliker
Lawrence Berkeley National Laboratory
Berkeley, California USA
LOliker@lbl.gov

Samuel Williams
Lawrence Berkeley National Laboratory
Berkeley, California USA
swwilliams@lbl.gov

Shirley Moore
Oak Ridge National Laboratory
Oak Ridge, Tennessee USA
mooresv@ornl.gov

Wolfgang Windl
The Ohio State University
Columbus, Ohio USA
windl.1@osu.edu

Abstract

The Rapid Alloy Method for Producing Accurate, General Empirical potential generation toolkit (RAMPAGE) is a program for fitting multicomponent interatomic potential functions for metal alloys. In this paper, we describe a collaborative effort between domain scientists and performance engineers to improve the parallelism, scalability, and maintainability of the code. We modified RAMPAGE to use the Message Passing Interface (MPI) for communication and synchronization, to use more than one MPI process when evaluating candidate potential functions, and to have its MPI processes execute functionality that was previously executed by external non-MPI processes. We ported RAMPAGE to run on the Eos and Titan Cray systems of the United States Department of Energy (DOE)'s Oak Ridge Leadership Computing Facility (OLCF), and the Cori and Edison systems at the DOE's National Energy Research Scientific Computing Center (NERSC). Our modifications resulted in a 7× speedup on 8 Eos system nodes, and scalability up to 2048 processes on the Cori system with Intel Knights Landing processors. To improve maintainability of the RAMPAGE source code, we introduced several software engineering best practices to the RAMPAGE developers' workflow.

CCS Concepts • **Software and its engineering → Software performance**; *Collaboration in software development*; • **Computing methodologies** → Parallel algorithms; Molecular simulation;

Keywords Applications, performance engineering, Message Passing Interface

ACM Reference Format:
Philip C. Roth, Hongzhang Shan, David Riegner, Nikolas Antolin, Sarat Sreepathi, Leonid Oliker, Samuel Williams, Shirley Moore, and Wolfgang Windl. 2017. Performance Analysis and Optimization of the RAMPAGE Metal Alloy Potential Generation Software. In *Proceedings of 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems (SEPS'17).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3141865.3141868

SEPS'17, October 23, 2017, Vancouver, Canada

ACM ISBN 978-1-4503-5517-9/17/10…$15.00
https://doi.org/10.1145/3141865.3141868

1 Introduction

In this paper, we describe a recent collaboration between members of the Center for Performance and Design of Nuclear Waste Forms and Containers (WastePD) [3], an Energy Frontier Research Center (EFRC) supported by the United States Department of Energy (DOE) Office of Science, and members of the Institute for Sustained Performance, Energy, and Resilience (SUPER) project of the Office of Science's Scientific Discovery through Advanced Computing (SciDAC) program. The Rapid Alloy Method for Producing Accurate, General Empirical generation toolkit (RAMPAGE) [10, 13, 14]

is software that finds multicomponent interatomic potential functions for metal alloys. RAMPAGE is used to study the properties of metallic glasses and high-entropy alloys for use in nuclear waste containers. Our experience illustrates a collaborative approach to refactoring and re-engineering an existing application for improved parallelism, scalability, and maintainability. RAMPAGE uses a genetic algorithm approach in a master-worker organization, and its initial implementation could exploit multiple cores within a single system node. Thus a major focus for our collaboration was to enable RAMPAGE to use more than one system node, and to enable its underlying molecular (MD) dynamics simulations to use more than one process when evaluating candidate potential functions. We modified RAMPAGE to use the Message Passing Interface (MPI) [4, 5] for communication and synchronization, to support multiple MPI processes for each MD simulation, and to have its MPI processes execute functionality that was previously accomplished using externally-invoked, non-MPI processes. We ported the software to the OLCF's Titan Cray XK7 with graphics processing units (GPUs), OLCF's Eos Cray XC30 system, NERSC's Cori Cray XC40 system with Intel Knights Landing manycore processors, and NERSC's Edison Cray XC30 system. On the Eos system, our modifications resulted in a 7× speedup on 8 compute nodes. Based on our evaluation of these modifications, we developed several recommendations for future work to further improve RAMPAGE's performance and scalability.

In this experience paper, we describe our analyses, evaluate our modifications, and discuss our recommendations for future RAMPAGE improvements. We emphasize that the activities we describe were a true collaboration between the WastePD materials scientists and the SUPER computer scientists. We largely avoided the all-to-common pattern of HPC performance engineering efforts: computer scientists obtain an application snapshot and toy problem inputs, analyze and optimize the code for months or even years in isolation from the application developers, deliver their final modifications with claims of large performance gains to the application developers, and all involved are frustrated when the modifications are never incorporated into the application's code base. In contrast, throughout this effort we maintained close communication between the SUPER and WastePD team members, and members from both teams contributed code modifications several times throughout the project. WastePD team members tried intermediate versions produced by the SUPER team, and adapted their input problem based on its behavior (e.g., by identifying the reason why some configurations caused molecular dynamics simulations to fail and by adjusting the input problem's configuration to avoid that failure). We don't claim that our project interactions were perfect during this brief six-month project—for instance, we would have liked to progress more in terms of support for complex alloys—but overall we are very pleased with the collaboration and its outcomes. We present our experiences in this paper with the hope that application developers and performance engineers can adapt our collaborative approach and avoid our mistakes when working to overcome their own parallel performance challenges.

2 Background

To find a high quality potential function for a given alloy, RAMPAGE uses a *genetic algorithm* (GA) approach to generate and evaluate many candidate potential functions. Each candidate is constructed by combining potential functions for the individual components of the alloy. The software uses a traditional master-worker organization. A *fitting driver* (the master) generates an initial collection of candidate potentials, and distributes them one at a time to a collection of worker processes for evaluation. When a worker receives a candidate potential to evaluate, it uses the Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [9] to simulate the behavior of the alloy's atoms assuming the candidate potential correctly described that behavior. After the simulation terminates, the worker computes a few metrics that capture how well the candidate potential fared, and returns these evaluation metrics to the fitting driver. Once the fitting driver receives evaluation metrics for all of its initial candidate potentials, it "crosses" high quality potentials from the initial batch to create a new batch of candidate potentials. After a user-configurable number of iterations of this generate-distribute-evaluate cycle, it outputs the potential function with the highest quality metric values as the best potential for the alloy under consideration.

When we started our collaboration on RAMPAGE, it was implemented as a collection of C++ programs, Bash shell scripts, and Python scripts that used the file system and command line arguments to separately-spawned processes for communication and synchronization. Setting up a problem involved running several shell scripts that copied a set of template scripts and LAMMPS data files into a problem-specific directory. Those shell scripts configured the templates and data files for the specific problem using other shell scripts and Python scripts. When running RAMPAGE itself, the C++ fitting driver managed a collection of worker processes for evaluating candidate potential functions. The fitting driver ensured that no more than a user-configurable number of worker processes were running at any given time. The fitting driver used problem inputs to create a problem-specific shell script for evaluating a single candidate potential function. To evaluate each candidate potential, the fitting driver used the C standard library's `system()` function to spawn a separate shell process that executed a generic worker shell script to do the evaluation. This worker shell script evaluated its assigned candidate potential using other shell scripts (including the problem-specific shell script), Python scripts, and a single-process LAMMPS invocation. The fitting driver communicated problem definitions and inputs to the worker

process via parameters to the worker shell script and via the filesystem, and the worker communicated quality metric values to the driver via the filesystem. The implementation had no special support for load balancing such as via work stealing.

HPC experts focused on extreme scale computing might be tempted to criticize this initial RAMPAGE implementation as one that is obviously ill-suited for running on HPC systems, and that should have been developed differently. In fact, the software has a modular organization with evidence that the developers used several widely-accepted software development practices. The initial implementation was a plausible result of a long-term development activity by computational scientists that prioritized productivity over performance (e.g., as evidenced by the widespread use of shell and Python scripts). Its performance was adequate for small-scale problems on small-scale systems, and it was only when the developers attempted to run it on larger-scale systems that they encountered its performance and scalability limitations.

3 Modifications

After an initial kickoff meeting to determine objectives and discuss potential strategies, the team worked together closely through regular conference calls and use of a joint code repository. The primary objective for our RAMPAGE work was to improve its performance and scalability so that RAMPAGE users could obtain better quality alloy potential functions with a given amount of computing resources and in a given amount of time. Improvements to RAMPAGE's maintainability were important, but secondary. In this section, we describe how our modifications to the initial RAMPAGE implementation meet our objectives.

3.1 Multi-node Scalability

The initial RAMPAGE implementation used the standard C library's `system()` function to spawn external shell script processes, which in turn spawned single-process LAMMPS and Python interpreter processes as described in Section 2. Thus, the initial implementation had no support for creating worker processes on nodes other than the one running the fitting driver. To enable RAMPAGE to conduct more simultaneous candidate evaluations than can fit in one compute node, we modified the program to use the MPI library for communication and synchronization, allowing us to use the HPC system's parallel program launcher (e.g., `mpirun`) for efficient creation of driver and worker processes on more than one system node. Worker processes no longer interpret a shell script. Instead, they run the same executable as the fitting driver (though with different functionality). In addition to allowing RAMPAGE to use more worker processes than can comfortably fit on one compute node, this change had a more subtle benefit: rather than being spawned each time the driver needs to evaluate a candidate potential, worker processes are now long-lived and can evaluate multiple candidate potentials during their lifetimes, thus minimizing the overhead of creating and destroying worker processes.

3.2 Scalable, Efficient Communication

The initial RAMPAGE implementation used the file system and shell script command lines to communicate between the fitting driver and its worker processes. We replaced these methods with MPI communication operations. For instance, instead of using a shell script to copy problem inputs to locations in the file system where worker processes would use file naming conventions to find their inputs, the driver now uses MPI broadcast operations for efficient distribution of common problem inputs to all workers.

In the initial implementation, after the driver spawned worker processes it polled its workers to identify when one of them had written its candidate evaluation quality metrics to a global results file. The polling logic was complicated and not reliable, e.g., when the results file is stored on a Network File System (NFS) file system. We modified the driver to assign work to each worker, and the workers to return their results to the driver, using MPI point-to-point operations. After sending work to a worker, the driver issues a non-blocking receive for the result of that work. When not doing other work like recording a worker's quality metric results or generating new candidate potentials, the driver uses an MPI "waitany" operation that allows it to sleep until a results message arrives from any of its workers. This approach lets the driver behave in an asynchronous, event-driven manner similar to that of the initial RAMPAGE implementation, but without the inefficient and error-prone approach of polling the filesystem in search of results files written by each worker. However, as we discuss in Section 5, unless process placement is carefully managed, this modification also has the potential to cause a load imbalance within the program, or to cause process placement to be misaligned relative to compute node boundaries.

3.3 Multi-process LAMMPS

When a RAMPAGE worker evaluates a candidate potential, it uses LAMMPS to simulate the alloy's behavior assuming the candidate potential was the alloy's true potential function. Although LAMMPS can be built as a standalone parallel program that uses MPI for communication, workers in the initial RAMPAGE implementation only invoked single-process LAMMPS runs. When we first introduced MPI into RAMPAGE and started running it on batch-scheduled OLCF and NERSC systems, this design choice became a requirement: within a batch job, after the job's controlling shell script started RAMPAGE with the parallel program launcher, neither the resulting RAMPAGE processes nor the shell processes they spawned could then execute the parallel job

launcher again to start multi-process LAMMPS runs. Nevertheless, the WastePD team indicated that limiting LAMMPS runs to a single process performed too poorly for the complex alloy configurations and quality metric calculations they desired to perform. To enable simulation of more complex alloys and calculation of more demanding quality metrics, we added the ability for RAMPAGE workers to run LAMMPS simulations using multiple processes. Supporting this functionality required a surprising number of significant changes to RAMPAGE.

To use multiple processes, LAMMPS must be built to use the MPI library and, when started, must be provided with an MPI communicator that defines the collection of processes it should use for its simulation. When run as a standalone parallel program, a system's parallel program launcher creates the processes for LAMMPS to use, and these processes are accessible to LAMMPS via the default `MPI_COMM_WORLD` communicator. Because most batch-scheduled HPC systems do not allow one MPI program to spawn another MPI program, we could not rely on this approach for running multi-process LAMMPS simulations from within RAMPAGE. Fortunately, LAMMPS can be embedded as a C++ object into a host program, and manipulated using an application programming interface (API). In this mode, LAMMPS must be built as a library and linked into the RAMPAGE executable, and all RAMPAGE processes that cooperate to run a given LAMMPS simulation must be part of an MPI communicator that is provided to the LAMMPS C++ object when it is created.

Our decision to embed LAMMPS into RAMPAGE had a few ramifications for our RAMPAGE implementation. First, although the MPI standard defines the `MPI_Comm_spawn` function for dynamically creating new processes as an MPI program runs, few batch-scheduled HPC systems support it, and so the parallel program launcher must create all the processes RAMPAGE will use (including those for running LAMMPS) when it starts RAMPAGE. Given this restriction, we introduced a hierarchy of MPI communicators and a new "worker helper" process type into the RAMPAGE organization as shown in Figure 1. RAMPAGE now has one communicator for communication between driver and workers, and one "LAMMPS communicator" per worker that specifies the worker helper processes used for the workers' LAMMPS invocations. Because each worker must provide an MPI communicator when constructing its LAMMPS object, the RAMPAGE driver and worker processes partition the available processes in `MPI_COMM_WORLD` into these smaller LAMMPS communicators when the overall program is first started, and before creating their LAMMPS objects.

A second consequence of embedding LAMMPS is that it allows us to reduce the number of filesystem accesses needed to define LAMMPS simulation inputs and to obtain simulation results. LAMMPS uses a text-based input script to define and execute a simulation. We modified RAMPAGE workers to construct this input script in memory rather than writing it to a file for LAMMPS to read. Furthermore, because most of these input commands are the same for every candidate potential a worker will evaluate, RAMPAGE workers now construct the common part of the input script once during program initialization and reuse it for every LAMMPS invocation, further reducing the number of file system accesses compared to the initial RAMPAGE implementation. In addition to these input commands, LAMMPS reads a small number of data files during each simulation to evaluate a candidate potential function. Although we have not yet modified our RAMPAGE implementation to do it, we believe RAMPAGE could use the same construct-in-memory approach for these data files as we do with the LAMMPS input commands.

In the initial RAMPAGE implementation, each worker process executed a shell script that invoked yet another script to construct an input data file containing the candidate potential function to be used in the LAMMPS simulation. This additional script did some setup work and then invoked a C++ program to construct the LAMMPS file. To avoid losing the benefit of embedding LAMMPS into worker processes, we modified this C++ program so that it could be built as a library that exposes its potential generator as a function callable by RAMPAGE workers, with inputs and results passed in memory rather than via the file system.

3.4 Property Calculation Optimization

Embedding LAMMPS also allows RAMPAGE to extract simulation results directly from the LAMMPS object in memory rather than parsing them from a LAMMPS output file. The initial RAMPAGE implementation used shell and Python scripts to post-process LAMMPS output to compute metrics reflecting the quality of the candidate potential function used in the simulation. Just as we did not want to use external processes to construct LAMMPS input, we did not want to compute quality metrics from LAMMPS outputs using external processes. Instead, we modified the RAMPAGE workers to do the quality metric computation internally. Because the RAMPAGE developers still wanted to be able to write the quality metric computations using Python, we explored the feasibility of embedding a Python interpreter into each RAMPAGE worker. As with the common LAMMPS input commands, this strategy allows each RAMPAGE worker to parse the quality metric computation script once at initialization, rather than each time it evaluates a candidate potential, and allows the inputs and outputs to the metric calculation function to be passed in memory rather than via the file system. However, this strategy also greatly complicates RAMPAGE's build- and run-time configuration, and can be especially troublesome on HPC systems like those in the OLCF that default to statically-linked executables and do not allow access to a user's home directory from compute nodes.

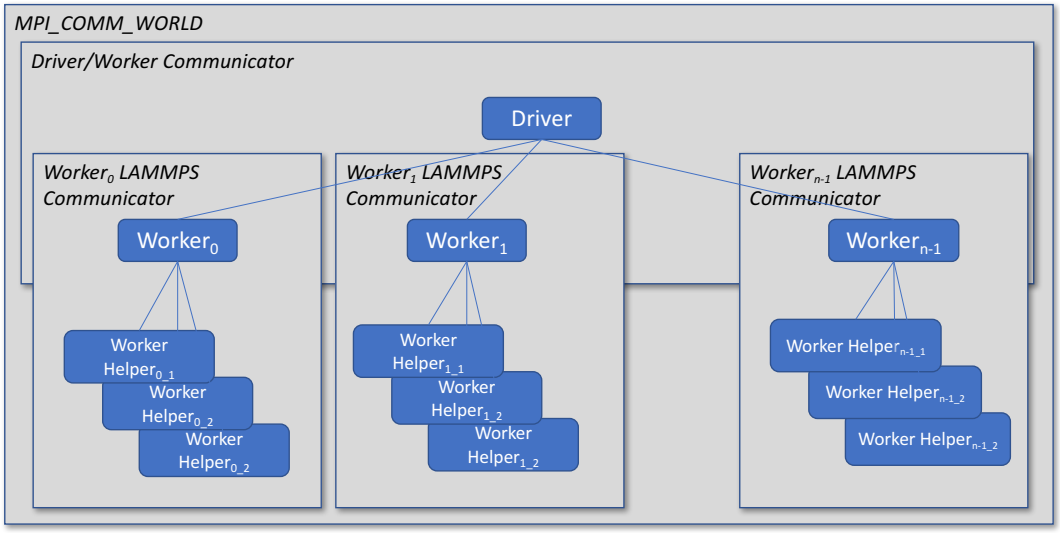

Figure 1. Hierarchy of MPI communicators and process types used by RAMPAGE. In this example configuration, RAMPAGE uses four processes for each LAMMPS simulation.

As an alternative, we investigated an approach that replicates the Python quality metric computation script's functionality within the C++ worker code itself. The Python property calculation script uses the `polyfit` function from the NumPy Python module [12] to fit a polynomial to LAMMPS output data. Because it is invoked every time a worker runs LAMMPS, it is run many thousands of times during a typical RAMPAGE run. In our alternative implementation, we implemented two variants of the least-squares fitting algorithm using the Linear Algebra PACKage (LAPACK) [1] and GNU Scientific Library (GSL) [2]. In addition to reducing the complexity of building and using RAMPAGE, this approach may also provide a performance benefit: because of their use in software from many problem domains, HPC system vendors often provide highly optimized versions of these libraries such as Cray's Scientific Libraries (LibSci), Intel's Math Kernel Library (MKL), and IBM's Engineering and Scientific Subroutine Library (ESSL).

We evaluate these approaches in Section 4 and discuss our recommendations for implementing property metric calculations in Section 5.

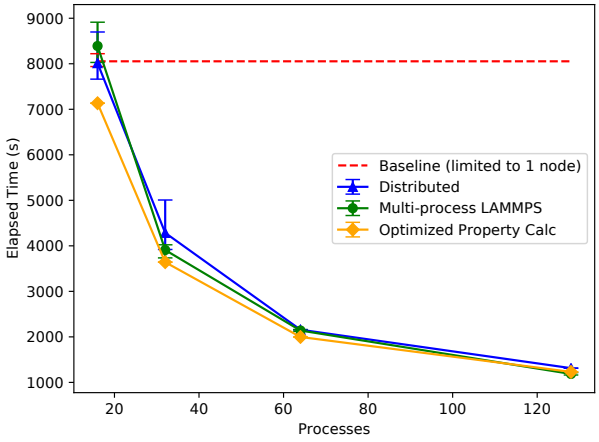

Figure 2. RAMPAGE elapsed time for evaluating 4000 Cu-Ni candidates running on the OLCF Eos system. Note that baseline version was limited to 16 processes on a single system node.

4 Impact

We saw substantial performance and scalability benefits from our RAMPAGE modifications. Figure 2 shows the elapsed time required to generate and evaluate 4000 candidate potentials for the Cu-Ni alloy running on the OLCF Eos system. Eos is a Cray XC30 with 736 nodes, each containing two 8-core Intel Xeon E5-2670 (Ivy Bridge) processors clocked at 2.6GHz and with support for two hardware threads. Each node also contains 64GB SDRAM but no local storage. The nodes are connected using Cray's Aries interconnect with a Dragonfly network organization [7]. Eos is similar to the National Energy Research Scientific Computing Center (NERSC) Edison system, but is smaller and uses an earlier generation of the Intel Xeon Ivy Bridge processor. On this system, we used the Intel version 17 compilers with the optimization flags `-O3 -xHOST -ip -no-prec-div`. The figure shows elapsed time for four versions:

- Baseline: Version with only enough modifications to run successfully on the OLCF Eos system.

- Distributed: Version that uses MPI for communication between fitting driver and worker processes. Invokes single-process LAMMPS as a separate process when evaluating candidate potentials, and uses external processes to set up and post-process each LAMMPS run.
- Multi-process LAMMPS: MPI-based distributed version that invokes LAMMPS as a library, supports multiple processes per LAMMPS invocation, and inlines LAMMPS pre- and post-processing.
- Optimized Property-Calc: MPI-based distributed version that invokes LAMMPS as a library and utilizes LAPACK-based least-squares fitting for computing quality metrics.

For the runs with the multi-process LAMMPS version, we configured the program so that most workers use two processes per LAMMPS invocation. (See Section 5 for further discussion about why it is *most* workers and not *all* workers.) In the figure, each data point indicates the average elapsed time over three runs, and the bars above and below each data point indicate the min and max elapsed time for those three runs. Also, the figure shows a horizontal line for the baseline performance to ease the comparison between the baseline version's performance and that of our modified versions, but recall that the baseline version was limited to running on a single node.

The figure demonstrates several positive performance and scalability benefits resulting from our RAMPAGE modifications. Most fundamentally, our modifications enabled the program to run on multiple system nodes, thus increasing its possible throughput for evaluating candidate potential functions. The resulting scaling behavior was very good: we observed more than 7× speedup when run on 8 Eos nodes. We expected good scalability because RAMPAGE evaluates each candidate potential independently, thus for most of its run it is operating in an "embarrassingly parallel" manner.

RAMPAGE's response to process affinity controls varied across the four versions we measured. The default process affinity policy on the Eos system is to bind each process to a specific processor core. Under this policy, any child processes that a process creates are bound to the same core as the parent. Because the original implementation's master process creates all worker processes as child processes, this default policy causes *all* RAMPAGE processes to share the same processor core. To avoid this extremely undesirable situation, we turned off process affinity when running the original implementation so that the compute node's operating system could migrate processes to any compute node core. For the "Distributed" version, we also achieved slightly better performance by turning off process affinity than with the default affinity policy. For the "Multi-process LAMMPS" and "Optimized Property-Calc" runs, we observed the opposite: in these runs, worker processes do not create child processes and we achieved better performance using the default process affinity policy.

When run on only one system node, the figure suggests that our modified versions did not demonstrate a substantial performance benefit over the baseline version. These results are slightly deceptive in that the baseline version used an old LAMMPS version whose force calculations are less accurate and less computationally demanding than those of the up-to-date LAMMPS version we used for our "Distributed," "Multi-process LAMMPS," and "Optimized Property-Calc" runs. The RAMPAGE developers prefer to use the newer version, but it is not backward-compatible and so we were not able to run our modified RAMPAGE versions using the older LAMMPS version without making more extensive modifications to the initial RAMPAGE version. We believe that if we were able to make a more direct comparison, our modified RAMPAGE versions would demonstrate a significant performance benefit over the baseline version even when using only a single node.

The results shown in the figure also suggest a slight performance benefit for our test problem when using two processes per LAMMPS invocation compared to the single-process LAMMPS configuration used for the "Distributed" runs. We expected a more dramatic performance benefit with the multi-process LAMMPS support, and we hypothesize that the Cu-Ni test problem and the potential evaluation metrics used in the current implementation are not computationally demanding enough to overcome the parallel overheads of using multiple processes for each LAMMPS run.

We used Cray's LibSci implementation of the LAPACK library for our "Optimized Property-Calc" runs. We obtained good performance gains from using the optimized least-squares fitting implementation compared to the Python implementation. The improvement is not as pronounced at higher processor counts for this problem because each worker process performed fewer polynomial fitting invocations. We expect that use of a vendor-optimized scientific library will provide a substantial performance benefit when used in production RAMPAGE runs.

4.1 Software Engineering Best Practices

In addition to our performance optimizations and scalability improvements, we also introduced a few engineering best practices to the RAMPAGE development workflow. In particular, we introduced the use of a source code repository to support change tracking and to simplify concurrent development by project team members. The RAMPAGE Git repository is currently hosted at Bitbucket.org, and although access is currently limited to project team members, it will be easy to enable public access whenever we decide to relax the access controls.

We also introduced a GNU autotools-based configure and build infrastructure to the RAMPAGE code base. This infrastructure eases the difficulty of configuring the software for

use on multiple systems, and is especially useful in situations where the source code tree is hosted on file systems that are accessible on multiple systems that differ in their hardware or software environment. It also unifies several disparate configuration approaches that were being used in the initial implementation, reducing the possibility that one part of the code might be configured to use a different compiler or compilation flags than another part.

5 Recommendations

We made many modifications to RAMPAGE that resulted in scalability, performance, and maintainability benefits. Based on our experiences, we identified several directions for further work that we believe will provide further benefit.

5.1 Batching and Load Balance

Currently, the RAMPAGE driver distributes work to its workers one candidate at a time. When a worker reports its results back to the driver, the driver gives that worker another candidate to evaluate. In both communication directions, the amount of data transferred is small. Because candidate evaluations are independent, and because the fitting driver constructs a large number of candidates at one time (as opposed to constructing a single new candidate only when it receives results from a worker), there may be some performance benefit to having the driver provide candidates to workers in large batches so as to reduce the number of messages and increase message size between driver and workers. At the extreme, when the driver generates C candidates for evaluation, and has S workers available, it could send C/S candidates to each worker.

Changes to the number of candidates per work assignment might also impact the workload balance across workers. The time required to evaluate each candidate potential function varies. It depends on factors such as whether its LAMMPS simulation terminates because it converged or because it reached a limit on the number of simulation time steps. RAMPAGE's current master-worker organization and its practice of distributing candidates one-by-one has some inherent load balancing capabilities. But, as shown in Figure 3, the current implementation can still suffer from significant load imbalances at the ends of candidate evaluation phases. That figure shows a Vampir visualization of an event trace collected using the Score-P [8] data collection library. The trace was collected from a RAMPAGE run for the Cu-Ni problem on one Eos node (16 cores), configured for 8 workers, two candidate batches, and 100 candidates per batch. The timeline part of the visualization clearly shows a load imbalance at the end of the candidate evaluation phases. Increasing the number of candidates per work assignment might exacerbate that load imbalance if a worker receives an "unlucky" work assignment containing several candidates that take a long time to evaluate. Further investigation is needed to determine how best to trade-off the messaging overhead reduction from increasing the number of candidates per work assignment against the potential for load imbalance.

5.2 True SPMD Organization

RAMPAGE currently uses a master-worker organization. Figure 3 clearly shows that while workers are evaluating candidates, the master is mostly idle. The figure also demonstrates a negative performance impact under our current practice of running RAMPAGE with the same number of processes as the number of available cores. To obtain the figure, we ran RAMPAGE with 16 processes and 8 workers on 16 cores using Eos' default process placement policy. Most workers used two processes for running LAMMPS, and the timeline visualization clearly shows that a worker and its corresponding helper process are idle in the same intervals. Because the driver process is allocated its own core, the first worker (labeled "Master thread:1" in the figure) has no helper. Interestingly, this worker was not necessarily the slowest to complete its work. This may reflect the inherent load-balancing properties of the "one candidate at a time" approach used in the current RAMPAGE implementation.

There are several ways to address the poor process placement problem. One is to oversubscribe the cores by requesting more processes than the number of cores, and to carefully specify process affinity to cores so that the master and first worker are bound to the same core. Another is to turn off all process affinity and let the node's operating system migrate processes among cores according to its policy. We might finesse the problem by using the *last* MPI rank as the driver instead of MPI rank 0, so that a system's default process affinity policy would be less likely to split worker/helper communicators across system nodes and sockets within a node.

Because candidate evaluation in RAMPAGE is embarrassingly parallel, a more attractive alternative for dealing with the process placement problem is to switch to a true Single Program Multiple Data (SPMD) organization. With this organization, each available MPI process would construct and evaluate their own pool of candidates independently. Each would repeat the generate-evaluate-communicate cycle until the desired number of candidates had been evaluated across all. Once done, the program would use a final reduction to identify the highest performing candidates and to report these to the user. This approach has several attractive properties: it eliminates the driver process that sits idle most of the time, it avoids workers sitting idle while the driver generates new candidates, and it is easier to map processes to the available cores without crossing system or processor socket boundaries.

5.3 Acceleration

For this project, we targeted systems at OLCF and NERSC. Both centers feature systems with accelerators: OLCF's Titan

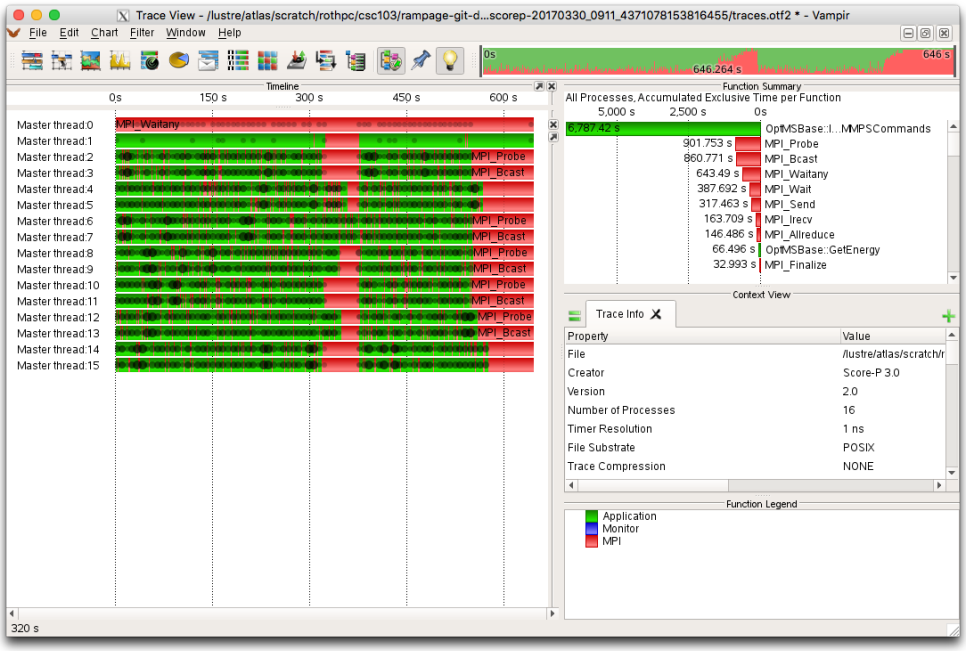

Figure 3. Vampir visualization of RAMPAGE event trace from Cu-Ni run on one node of OLCF Eos, showing load imbalance and poor mapping of processes to the available processor cores.

has graphics processing units, and NERSC's Cori has Intel Knights Landing Xeon Phis.

We used Eos for most of our work on OLCF systems, but we also verified that our modified RAMPAGE implementation runs on Titan. Because the profile pane of Figure 3 shows that the majority of RAMPAGE's total run time is spent within LAMMPS invocations, we targeted Titan's GPUs with a GPU-enabled LAMMPS library.When we ran our test Cu-Ni problem on Titan, we did not obtain a speedup compared to the non-GPU version. We attribute this to two factors. First, Ni-Cu candidate evaluation involves too little computation to overcome the overhead of transferring data to and from the system's GPUs. Second, because we mapped worker processes to the available system based on the number of CPU cores per node, we experienced significant contention for the single GPU per node. Our experience with Titan was useful, however, in that it suggests that RAMPAGE can support more complex and computationally demanding problems than the Cu-Ni problem, and that GPU-accelerated systems are feasible targets. Future work should involve evaluating these more demanding problems to GPU accelerated systems such as the forthcoming OLCF Summit system.

We also ported RAMPAGE to the NERSC Cori platform with Intel Knights Landing Xeon Phi processors. Each Cori node is a single-socket Knights Landing processor with 68 cores. Although each 1.4GHz core has smaller peak floating point performance than the 2.6GHz Intel Xeon E5-2670 (Ivy Bridge) processors in Eos by about 4.5×, each core supports 4 hardware threads and two 512-bit-wide vector processing units that provide 6× more available hardware threads and 6× more peak floating point throughput compared to the two 8-core processors in an Eos node. RAMPAGE scaled very well up to 128 MPI processes within a single Cori node, and as a result, RAMPAGE achieved similar performance to Eos when viewed from a "number of nodes" perspective. In fact, the "Distributed" implementation of RAMPAGE scaled well up to 16 nodes (16 × 128 MPI ranks) on Cori when run in cache mode, but larger configurations were limited by file system I/O performance. Throughout the project our modifications reduced the number of file system accesses needed during a RAMPAGE run, as described in Section 3.3 there remain opportunities to further reduce file system accesses when evaluating candidate potentials.

Also, because LAMMPS performance is by far the most important factor determining RAMPAGE performance, we expect a significant performance improvement from using a LAMMPS library built using aggressive Intel-specific optimizations [6] instead of library built using common compiler optimization flags for each tested platform.

5.4 Genetic Algorithm Framework

RAMPAGE uses a GA approach, and a significant part of the code base deals with the GA functionality (as opposed to candidate evaluation). While RAMPAGE supports a specific GA strategy, there exist heuristic optimization frameworks such as Optimus-Prime [11] that provide several heuristic search algorithms including GA and Particle Swarm Optimization. Optimus-Prime also provides problem-independent support

for distributing work across multiple system nodes and for results collection. Using such a framework requires a user only to provide functions that implement the problem-specific functionality and facilitate exploring several search algorithms to find a suitable match for their problem. Although we considered implementing RAMPAGE within the Optimus-Prime framework, we found that the initial implementation's extensive use of shell and Python scripts hindered our ability to quickly integrate it into the framework. Now that we have inlined most of RAMPAGE's functionality into its MPI processes, future work should revisit the integration of RAMPAGE into the Optimus-Prime framework.

5.5 Plug-in Architecture for Quality Metric Computation

In our current implementation, RAMPAGE workers compute three quality metrics from LAMMPS simulation results using either a Python function or a function from a scientific library such as an LAPACK implementation. The RAMPAGE developers prefer to implement their quality metric calculations in Python, and wish to compute more quality metrics than those currently implemented. However, we found that embedding a Python interpreter into RAMPAGE workers greatly increased the difficulty in building RAMPAGE and configuring it to run on batch-scheduled HPC systems. Future work should involve investigation of a plug-in architecture for alternative implementations of candidate quality metrics, with compile- or run-time configuration to select which implementation to use based on the user's performance goals and usage restrictions imposed by the target system. Integration into the Optimus-Prime GA framework might provide the benefits of this proposed plug-in architecture as a by-product of integration with the framework.

5.6 Out-of-source Build/Install

Although we added a configure and build infrastructure to RAMPAGE during this project, our current implementation does not support building the code outside the source tree or installing the built programs outside the source tree. Such capabilities are desirable on some HPC systems such as the OLCF systems because these systems only allow programs running on compute nodes to access a scratch parallel file system whose files are purged after a period of inactivity. On such systems, the RAMPAGE source tree must be stored in the parallel file system, and without careful management, it is easy for a developer to lose files from the source repository (including those needed by the repository management tool, Git). For better maintainability, it would be preferable for the source tree (which is persistent) to be placed in the developer's home directory, but the executables and supporting data files (which can be regenerated or copied from persistent templates) to be installed into the scratch parallel file system for program runs.

6 Summary

In a recent six-month collaboration between SUPER computer scientists and WastePD materials scientists, we conducted an evaluation of the performance, scalability, and maintainability of the WastePD project's RAMPAGE alloy potential generation software. By enabling it to run on more than one compute node using MPI-based parallelism, by enabling it to use more than one MPI process for molecular dynamics simulations, and by inlining functionality that previously had been done in separate processes, we greatly increased RAMPAGE's throughput capability for evaluating candidate potential functions. We ported the software to systems at two prominent DOE computing facilities, and observed greater than 7× speedup on 8 nodes of a Cray XC30 and scalability up to 2048 processes on a Cray XC40 KNL-based system. We also introduced software engineering best practices such as use of a source code repository to support concurrent development by multiple developers and an automated configuration and build infrastructure.

Based on our experiences with RAMPAGE, we made several recommendations for future work on the program. We suggested changing the program's master-worker organization to an SPMD organization in which each process generates and evaluates its own collection of candidate potentials, communicating only when necessary for its genetic algorithm implementation to cross promising candidates and to identify the program's best-known candidates. Because of the predominance of the molecular dynamics simulations within the program's overall run time, we recommended further investigation into using accelerated and optimized versions of LAMMPS on GPUs and KNL processors. And we recommended further consideration of implementing RAMPAGE in an existing genetic algorithm framework like Optimus Prime, so as to allow RAMPAGE developers to focus on implementing the problem-specific code.

During this project, we also identified several recommendations for collaborations between computer scientists and domain scientists, including:

- **Frequent communication is critical.** Every project member should be aware of how the code is changing and how those modifications impact team members' ideas about what has been done, what needs to be done, and what is possible.
- **Agree on terminology early.** The computer scientists and domain scientists may speak about the same concept but express it using different words. Don't just talk about the statement of work during the project kick-off meeting, spend time digging into each others' perspectives and make sure that when one part of the team says "thread," the other part of the team is thinking of the same concept.
- **Be a hands-on participant.** In our project, the domain scientists would not have identified problems

with some of the intermediate implementations unless they tried to use them on their own computing resources, and the computer scientists would not have been able to deliver something useful to the domain scientists if they did not track the domain scientists' modifications to the input problems.

- **Take advantage of software engineering technology.** Software code base management tools that allow one to track revisions and pursue multiple branches of development are a must. It is much more valuable to be able to talk about a version of the code as revision 1234 than it is to say it was the version that Susie was using on January 10th. A shared "to do" list is also helpful.
- **Trust.** Be willing to trust your project teammates, even if it is early in the project and they haven't quite earned it yet. The computer scientists can't do much to help with code or problem inputs they can't access for hands-on work, and domain scientists probably understand more about the computer science than many computer scientists assume.

We hope that by using our description of this project, and our recommendations for future work and future collaborations, others can adapt our experiences and recommendations for their own performance engineering work with applications targeting leading-edge HPC systems such as those deployed at the DOE OLCF and NERSC computing facilities.

Acknowledgments

This manuscript has been co-authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research under contract number DE-AC05-00OR22725. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Researchers from LBNL were funded by the Advanced Scientific Computing Research Program in the U.S. Department of Energy, Office of Science, under Award Number DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center (NERSC), which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

The contributions from OSU were supported as part of the Center for Performance and Design of Nuclear Waste Forms and Containers, an Energy Frontier Research Center funded by the U.S. Department of Energy, Office of Science, Basic Energy Sciences under Award # DE-SC0016584.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK Users' Guide* (third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

[2] Mark Galassi et al. 2009. *GNU Scientific Library Reference Manual* (3nd ed.). Network Theory Ltd. 592 pages.

[3] Gerald Frankel. 2016. Center for Performance and Design of Nuclear Waste Forms and Containers. https://science.energy.gov/bes/efrc/centers/wastepd/. (2016).

[4] William Gropp, Ewing Lusk, and Anthony Skjellum. 1999. *Using MPI: portable parallel programming with the message-passing interface* (2nd ed.). MIT Press, Cambridge, MA.

[5] William Gropp, Rajeev Thakur, and Ewing Lusk. 1999. *Using MPI-2: Advanced Features of the Message Passing Interface.* MIT Press, Cambridge, MA, USA.

[6] James Jeffers, James Reinders, and Avinash Sodani. 2016. *Intel Xeon Phi Processor High Performance Programming* (2nd ed.). Morgan Kaufmann. 662 pages.

[7] John Kim, William J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. In *Proceedings of the 35th International Symposium on Computer Architecture.* Washington, DC USA, 77–88.

[8] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir.* Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.

[9] Steve Plimpton. 1995. Fast parallel algorithms for short-range molecular dynamics. *J. Comput. Phys.* 117 (1995), 1–19.

[10] David Riegner, Logan Ward, and Wolfgang Windl. 2017. Welcome to the EAM Alloy Potential Generator. https://atomistics.osu.edu/eam-potential-generator/index.php. (2017).

[11] Sarat Sreepathi. 2012. Optimus: A Parallel Optimization Framework with Topology Aware PSO and Applications (poster). http://ft.ornl.gov/~sarat/files/sc12-SRC.pdf. (2012).

[12] S. van der Walt, S. C. Colbert, and G. Varoquaux. 2011. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering* 13, 2 (March 2011), 22–30.

[13] Logan Ward, Anupriya Agrawal, Katheraine M. Flores, and Wolfgang Windl. 2012. Rapid production of accurate embedded-atom method potentials for metal alloys. (2012). arXiv:cond-mat.mtrl-sci/1209.0619

[14] Logan Ward, Anupriya Agrawal, and Wolfgang Windl. 2017. Rapid Alloy Method for Producing Accurage, General Empirical (RAMPAGE) Potential Making Toolkit: Software Documentation and How-To Guide. https://atomistics.osu.edu/eam-potential-generator/potential-maker.tar.gz. (2017).