SAND2018-9853R

ECP Milestone Report
DARMA-MPI Interoperability
WBS 2.3.1.04, Milestone 15

Jeremiah Wilke, Sandia National Labs

August 19, 2018

ECP Milestone Report
DARMA-MPI Interoperability
WBS 2.3.1.04, Milestone 15

Office of Advanced Scientific Computing Research
Office of Science
US Department of Energy

Office of Advanced Simulation and Computing
National Nuclear Security Administration
US Department of Energy

August 19, 2018

ECP Milestone Report
DARMA-MPI Interoperability
WBS 2.3.1.04, Milestone 15

Approvals

Submitted by:

Jeremiah Wilke, Sandia National Labs 15	Date

Approval:

Douglas B. Kothe, Oak Ridge National Laboratory Director, Applications Development Exascale Computing Project	Date

Revision Log

Version	Creation Date	Description	Approval Date
1.0	August 19, 2018	Original	

EXECUTIVE SUMMARY

DARMA (Distributed Asynchronous Resilient Models for Applications) is a runtime library developed as part of the the Sandia ATDM (Advanced Technology Development and Mitigation) program. DARMA supports applications within 2.2.5.03 ADNN03-ASC ATDM SNL Application, which includes a number of applications featuring dynamic physics which requires load balancing and asynchronous communication for high performance. We have implemented a modern C++ programming model that can enable dynamic, asynchronous communication on top of existing data structures from a serial or MPI code. DARMA development has occurred in parallel with a verification milestone for ATDM in FY18. For FY19, DARMA will impact ATDM by enabling the performance benefits of a dynamic runtime through only incremental changes to existing verified MPI codes. The results presented here demonstrate the DARMA development process for an MPI mini-app, showing a 3-4x improvement in performance for a challenging problem relative to the parent MPI code and coming within 25 percent of the theoretically optimal performance achievable from a perfect, fine-grained load balancer for most cases. The code is released open-source at `https://github.com/DARMA-tasking/darma-futures`

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

DARMA (Distributed Asynchronous Resilient Models for Applications) aims to simplify the development of applications exploiting asynchronous, deferred execution using modern C++. Deferred execution enables task parallelism, communication overlap, and load balancing by declaring and enqueuing work rather than explicitly stating where and when computational work should take place. This represents a shift from *imperative* to *declarative* models. We identify two major performance and productivity drivers:

1.1 Communication Overlap

Consider an archetypal code example showing computation/communication overlap:

```
MPI_Isend(...);
MPI_Irecv(....);
do_some_work();
MPI_Waitall(...);
```

How much work (and which work) should go in the function `do_some_work`? Does `MPI_Isend` even guarantee forward progress before the `MPI_Waitall`? Rather than explicitly defining what work to overlap, an alternative approach (shown below) would simply enqueue tasks and communication and let a dynamic runtime overlap *as much as possible.*

1.2 Dynamic Load-balancing Without Repartitioning via Overdecomposition

Much of the work in developing an application is decomposing the problem across MPI ranks. Applications can then only load balance in a "synchronous" manner that involves re-partitioning the problem. For some applications, this can create performance problems as "partial chunks" cannot be quickly rebalanced across processes. For applications exhibiting persistence (load balance changing slowly), the re-partitioning problem may present more of a productivity than performance challenge. Overdecomposition creates a single problem decomposition at a finer granularity than an entire MPI rank. Work chunks can then be freely exchanged between processes to distribute load without requiring a synchronous re-partition.

1.3 MPI Interoperability Problem Statement

DARMA creates challenges for interoperating with MPI. DARMA provides a "virtual" context. Although the code is oblivious to load balancing, work patches can freely migrate around the system to enable load balance. If interoperating with an existing MPI code, a DARMA kernel must "borrow" and "return" the data from/to its original location in the parent MPI application. This requires cleanly defining 1) ownership semantics for data moving between DARMA and MPI modes and 2) runtime infrastructure to track data migration during DARMA load balancing. The latest version of DARMA is implemented using MPI as the underlying communication layer, avoiding resource sharing and runtime compatibility difficulties in previous versions. While the runtime infrastructure is now fully compatible with inter-operating, we must still define the handoff semantics in the programming model.

2. DARMA C++ ABSTRACTIONS

2.1 Asynchronous References

The most significant transformation from a "simple" C++ code to DARMA C++ is the use of `async_ref` template wrappers.

```
async_ref_mm<int> myInt;
```

A full discussion of DARMA semantics is beyond the scope of this report. However, the `async_ref` enforces migratability (serialization) of the underlying type and acts as a future enabling deferred execution through asynchronous tasks.

2.2 Deferred Execution and Move Semantics

Rather than directly calling functions, DARMA enqueues work. A parent MPI application first creates a DARMA context using a given communicator.

```
auto dc = allocate_context(MPI_COMM_WORLD);
```

For that context, data and tasks are created:

```
auto myInt = dc->make_async_ref<int>(0);
auto deferredInt = dc->create_work<MyFunction>(std::move(myInt));
...
auto finalResult = dc->create_work<Final>(std::move(deferredInt));
```

The `create_work` function will invoke the *functor* `MyFunction` when the inputs become available. The call to `create_work` returns an `async_ref` (i.e. future) that acts as a handle for the future state of `myInt`. Because work is deferred, not executed immediately, the input `myInt` must be *expired*. C++ move semantics are required to transfer of ownership from the application to the DARMA runtime.

2.3 Collections

Collections are the main vehicle for distributed memory. They create multiple instances of the same object across the system.

```
int pieces = ...;
auto coll = dc->make_collection<int>(npieces);
```

In contrast to an MPI communicator whose size is tied to the number of launched processes, the size of the collection is arbitrary. Overdecomposition is enabled by creating more work patches than MPI ranks.

2.4 Accessors and Serialization

Certain operations (send, recv, load balancing, return to MPI) all have need to access different members of a data structure. A send operation, e.g., only needs to access and send ghost data. MPI interoperability may only need to transfer a subset of the data, for example charge densities for a solver (see below). Load balancing must transfer all of the data for a work patch. DARMA includes a serialization library. An accessor must implement the required serialization.

```
struct Accessor {
  template <class Archive> static void compute_size(Swarm& s, Archive& ar){}

  template <class Archive> static void pack(Swarm& s, Archive& ar){}

  template <class Archive> static void unpack(Swarm& s, Archive& ar){}
};
```

The data required is placed into a generic archive object, which enables type-safe data movement optimizations, particularly for intra-process versus inter-process data movement.

3. PARTICLE-IN-CELL MINI-APP

3.1 Overview

The main application driver is the particle-in-cell (PIC) application for Sandia ATDM (Advanced Technology Development and Mitigation), which is called EMPIRE (2.2.5.03 ADNN03-ASC ATDM SNL Application). PIC poses serious load balancing challenges since it combines particle data with mesh data. An optimal, balanced distribution of particles is not the same as an optimal, balanced distribution of the mesh. DARMA aims to provide a productive programming model that easily and transparently enables load balancing. A

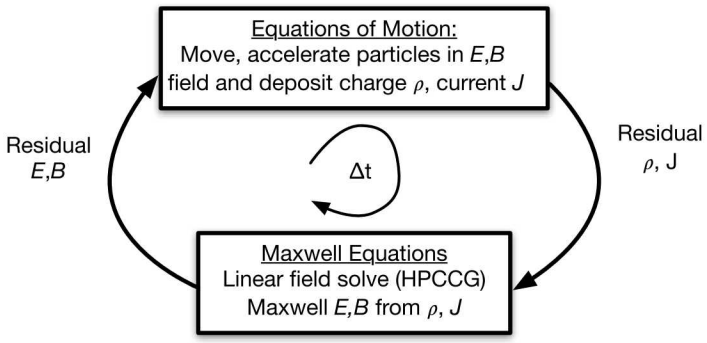

Figure 1: Overview of the iterations in the PIC algorithm showing the relationship between the particle move kernel and the field solve. Particles accelerate in the field, depositing charges and currents as they move. The residual charge and current deposited is used to update the fields for the next iteration.

parent MPI mini-app (9K lines) was developed first, with no load-balancing and a basic bulk-synchronous communication pattern. 500 lines of "wrapper" DARMA code was implemented in a single source file to add load balancing and asynchronous communication. This initial exercise demonstrates a workflow to be repeated with EMPIRE following its verification milestone in FY18. Both the DARMA runtime and the PIC mini-app can be obtained via the DARMA github repository: `https://github.com/DARMA-tasking/darma-futures`

Figure 1 shows the steps involved in the PIC application. The particles move through space, interacting and accelerating through electric and magnetic fields associated with cells in the mesh. Residual charge densities and currents are updated based on the particle movement. After each timestep, electric and magnetic fields are updated through a solver. The solver phase is easy to decompose into a balanced work distribution in standard MPI code. The particle move phase is highly irregular with dynamic particle trajectories and changing particle densities and is implemented as a DARMA kernel.

3.2 Data Distribution between DARMA/MPI

Certain data structures are required only for the MPI solver, only for the DARMA particle move, or are needed in both the DARMA and MPI phases. These data structures are summarized in Figure 2. Data movement between MPI and DARMA is not "symmetric." The residual densities generated by the move kernel must be passed to the MPI solver, but are not sent back when the DARMA kernel resumes. Similarly, the residual fields must be sent from the MPI solver to the DARMA move kernel, but are not sent back. The particles, which consume most of the memory and computational time (see below) are not directly required by the solver.

4. RESULTS

We wish to understand 1) the performance overheads of DARMA for a balanced problem without load balancing as a baseline case and 2) the performance benefits enabled by DARMA for an unbalanced problem. The critical parameter in both of these is the overdecomposition factor, which gives flexibility to the load balancer. Increasing overdecomposition gives more freedom to the DARMA runtime for communication overlap and better load balancing. However, increasing the number of work units to schedule increases runtime overhead. All results were collected on the Haswell partition of the Mutrino platform at Los Alamos National Lab.

4.1 DARMA Overheads for Balanced Problem

The balanced problem simulates a uniform particle density. Particles migrate through space, but an overall uniform density is maintained. Without load balancing and with no overdecomposition, the DARMA applica-

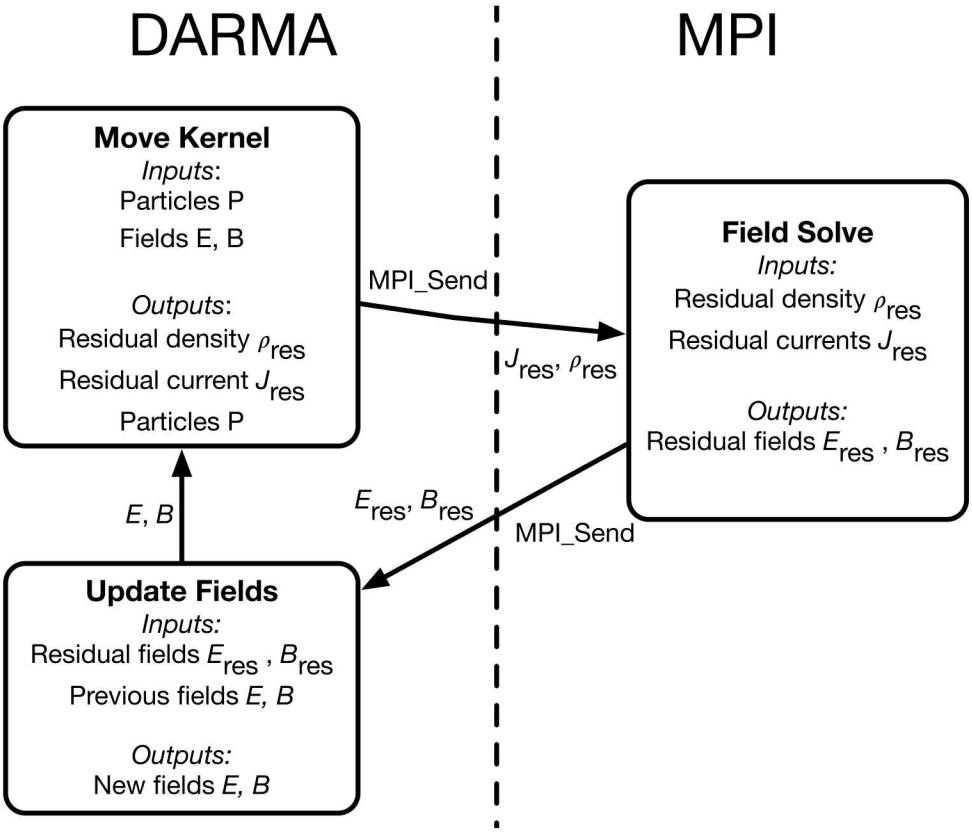

Figure 2: Overview of DARMA-MPI interoperability in the PIC application. The solver kernel assumes a fixed data distribution created by MPI. DARMA dynamically migrates data to achieve load balance in the move kernel. When interoperating between the two phases, only a small subset of data must be transferred from the DARMA location to the MPI location.

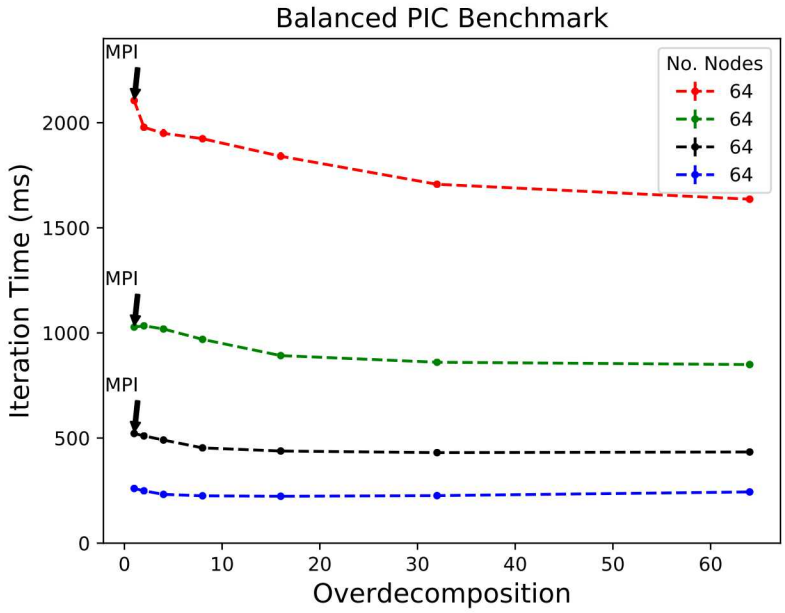

Figure 3: Performance of DARMA relative to MPI for the PIC mini-app for balanced strong-scaling problem with changing overdecomposition. All results collected on the Haswell partition of the Mutrino platform. The parent MPI application executes with an overdecomposition of 1.

tion is equivalent to an MPI-only application. Figure 3 shows the per-iteration time as overdecomposition increases from 1 to 64 for 8 to 64 nodes.

4.2 Load Balancing on Unbalanced Problem

The unbalanced problem simulates a large emission of charged particles from a 2D plate. A dense particle density exists in a small part of space while a low particle density exists in most cells. Figure 4 shows the effect of overdecomposition on improving the load balance for the move kernel. Note that too little overdecomposition does not grant sufficient flexibility but too much overdecomposition incurs performance overhead. The optimal overdecomposition shifts at larger node counts, with 8-32 generally being a good choice.

Figure 5 shows the time to execute the complete iteration, the solver portion, and the move kernel with and without load balancing assuming an overdecomposition factor of 32. In general, the MPI handoff is only a small portion of the overall time. Overall, at the largest scales, DARMA load-balancing improves performance by 3-4x relative to the MPI code. A theoretically optimal performance improvement can be estimated by DARMA by comparing the maximum task size to the average task size in the system. Additionally, the load balancer can estimate the "expected" improvement based on the computed distribution. These results are shown in Figure 6. The DARMA load balancer is generally able to compute a work distribution within 5-10% of a theoretically optimal distribution. The observed speedups, though, are quite different from the expected based on the load balance computation. Deviations from the expected distribution demonstrate either 1) runtime overheads that are not possible to load balance or 2) inaccuracies in the metrics used in the load balancer or changes in the optimal distribution over time. While no comparisons are made here to an MPI load-balancing strategy, DARMA at last achieves performance within 25% of optimal for most node counts. Performance analysis at the largest node counts is needed to identify discrepancies between the load balancer prediction and the observed speedups.

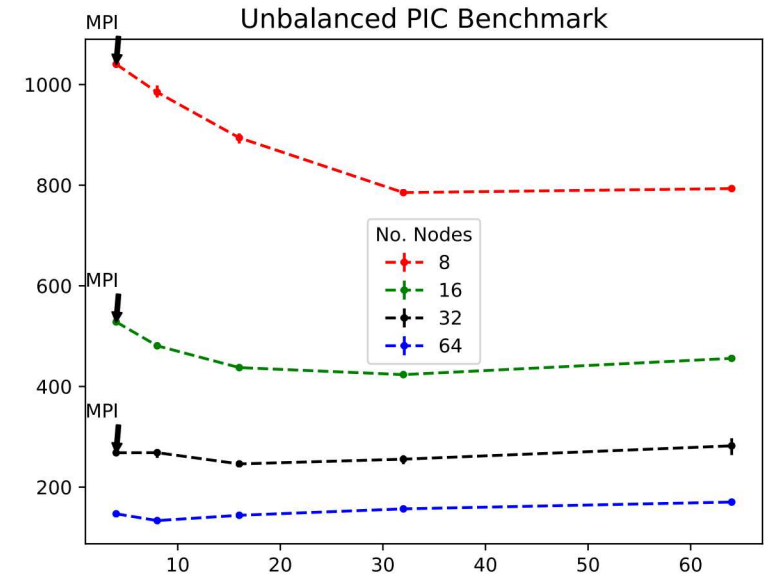

Figure 4: Performance of DARMA load balancing for unbalanced strong-scaling problem with changing overdecomposition. All results collected on the Haswell partition of the Mutrino platform.

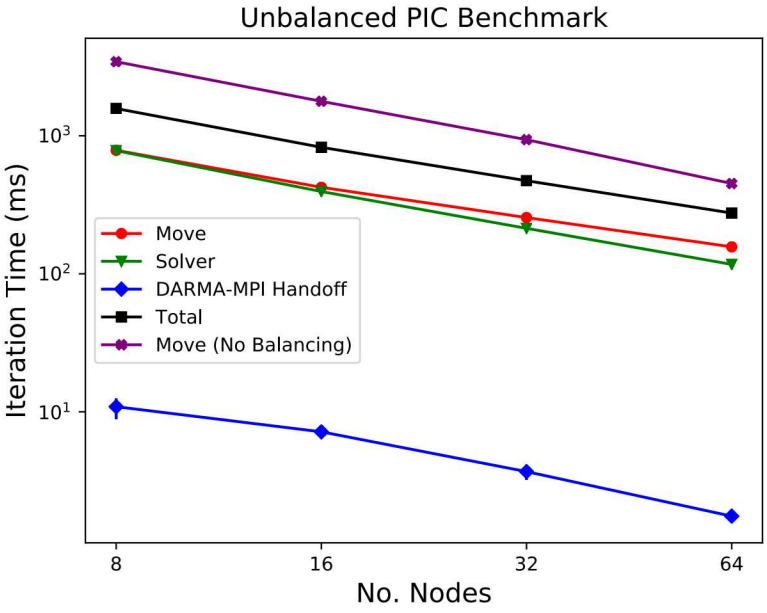

Figure 5: Performance of DARMA load balancing for unbalanced strong-scaling problem with overdecomposition 32 relative to MPI with static distribution. Scaling of the individual phases (move kernel, solver, DARMA-MPI handoff) are shown separately.

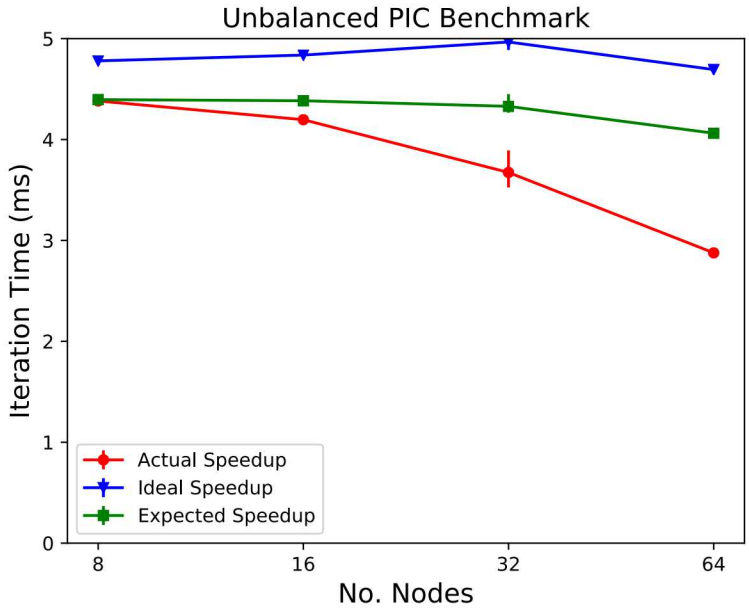

Figure 6: Performance of DARMA load balancing for unbalanced strong-scaling problem with overdecomposition 32 relative to a theoretically optimal load balanced distribution.

5. CONCLUSION AND FUTURE WORK

The results presented here show new runtime infrastructure and programming models support for interoperating a DARMA kernel with an MPI application. Modern C++ ownership models and future-like template wrappers provide the needed programming model semantics. Additional runtime infrastructure providing load balancer and data migration between the DARMA and MPI phases was implemented. Although not part of the milestone, missing is an MPI-based synchronous load balancer (e.g. Zoltan) as a baseline for load-balancing performance. Future work will adapt the MPI-only version of the PIC mini-app for bulk-synchronous load balancing.

ACKNOWLEDGEMENTS

This research was supported by the Exascale Computing Project (ECP), Project Number: 17-SC-20-SC, a collaborative effort of two DOE organizations—the Office of Science and the National Nuclear Security Administration—responsible for the planning and preparation of a capable exascale ecosystem—including software, applications, hardware, advanced system engineering, and early testbed platforms—to support the nation's exascale computing imperative.