SAND2019-1134

SANDIA REPORT

SAND2019-DARMA
Unlimited Release
Printed January 2019

DARMA-EMPIRE Integration and Performance Assessment – Interim Report

J. Lifflander, M. Bettencourt, P. Miller, P.P. Pébaÿ, M. Perrinel, F. Rizzi, N. Slattengren, G. Templet

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

SAND2019-DARMA
Unlimited Release
Printed January 2019

DARMA-EMPIRE Integration and Performance Assessment – Interim Report

J. Lifflander[*], M. Bettencourt[⋆], N. Slattengren[*], G. Templet[*]
[*]08753, [⋆]01352
Sandia National Laboratories
[*]Livermore CA, [⋆]Albuquerque NM
U.S.A.

P. Miller, P.P. Pébaÿ, M. Perrinel, F. Rizzi
NexGen Analytics
Sheridan WY
U.S.A.

Abstract

This is the DARMA FY19-Q1 interim report.

This page intentionally left blank.

This page intentionally left blank.

Contents

List of Figures

This page intentionally left blank.

Chapter 1

Introduction

We begin by presenting an overview of the general philosophy that is guiding the novel DARMA developments, followed by a brief reminder about the background of this project. We finally present the FY19 design requirements.

1.1 Philosophy

As the Exascale era arises, DARMA is uniquely positioned at the forefront of asychronous many-task (AMT) research and development (R&D) to explore emerging programming model paradigms for next-generation HPC applications at Sandia, across NNSA labs, and beyond. The DARMA project explores how to fundamentally shift the expression (PM) and execution (EM) of massively concurrent HPC scientific algorithms to be more asynchronous, resilient to executional aberrations in heterogeneous/unpredictable environments, and data-dependency conscious—thereby enabling an intelligent, dynamic, and self-aware runtime to guide execution.

Although subsets of concurrent algorithms have provably-good mesh partitionings (e.g, communication-optimal decompositions for 2.5D matrix multiplication [6]), dynamic and/or heterogeneous environments can quickly render predetermined partitionings suboptimal. Even within ostensibly "homogeneous" compute clusters, researchers have documented substantial chip-to-chip variations [1] along with OS variability [2]. These machine characteristics engender a dynamic—and sometimes unpredictable—execution context even for the most static problems. DARMA seeks to enable runtime reconfigurability (e.g., load balancing) for a wide range of concurrent algorithms that may have data-dependent computation or are simply executing in a dynamic environment.

1.1.1 Pillars

Across the emerging programming models, two foundational pillars of efficient tasking exist: asynchronous execution and overdecomposition. While each technique may be employed individually, their combined synergy realizes the maximum performance portability through machine-independent algorithmic expression.

Asynchronous Execution: Asynchronous execution enables interleaving sequential kernels based on the current state, data availability, and concurrent coordination of the algorithm. Introducing asynchrony to a parallel algorithm, makes it naturally more tolerant to aberrations in the execution: faults, processor speed variations, or dynamic, data-dependent work loads.

Overdecomposition: While asychrony without overdecomposition provides benefits (e.g., relaxing arbitrary ordering requirements and utilizing network resources more effectively over time) a machine-dependent decomposition of a scientific algorithm limits runtime flexibility and portability. In contrast, by expressing an algorithm with a logical decomposition, or natural structure of the domain science (e.g., physical space, matrix dimensionality), the programmer gives the system a valuable degree of freedom for remapping and exposing communication edges in the computational DAG. Furthermore, it enables the domain scientist to program in an abstract, machine-independent way that is highly portable across diverse architectures—including ones that may not exist today.

1.1.2 Benefits

The two pillars of tasking may yield several significant benefits for an application; in particular: load balancing, compositionality, and fault tolerance.

Load balancing is the remapping of pieces of work and data in order to better equilibrate resource utilization. HPC load balancing schemes come in several flavors. Persistence-based load balancing schemes, studied extensively in particular in Charm++ [7], incrementally (every k iterations) use past iterative behavior as a prediction for the future. By exploiting iterative instrumentation, the runtime can predict future work loads for persistent work units and therefore redistribute tasks and data. Task remapping can be viewed as very general load balancing strategy, employed when tasks and data are migrated dynamically, such as in work stealing [4]. Localized heuristics, which may be domain- or application-specific, can be used for dynamic remapping. Memory bounds or other localized runtime constraints may impact remapping decisions.

Application load balancing problems range from known provably-good distributions (e.g., optimal cyclic distributions for very specific classes of sparse symmentical tensors) to fully data-dependent imbalances (e.g., PDES, discrete event simulation with arbitrary rollbacks). Between these two extremes lie partially static problems for which inspector-executor schedulers may be very effective. Even for the most static application domains, unpredictable architectural variability (e.g., dynamic processor speed variation induced by temperature variability, or a chip failure/error leading to localized performance degradation) can severely impact overall performance especially in the context of bulk-synchronous parallel algorithms. The two pillars of tasking therefore allow for effective load balancing: overdecomposition enables task migration while asynchrony mitigates parallel coordination overheads.

Compositionality is the ability to interleave multiple, possibly disjoint but related parallel components. *Tasking* allows for the dynamic composition of parallel components, without incurring performance penalties or fully re-engineering components to execute concurrently. For applications ranging from Monte Carlo calculations to multi-physics codes, compositionality enables proper isolation and separation of concerns, when engineering distinct components while enabling them to be mapped to overlapping hardware resources, without the need for these components to exchange information nor even to be aware of their respective existences. This is a significant benefit for multi-physics toolboxes, enabling powerful reconfiguration for the end user.

Fault Tolerance is the ability to tolerate soft/hard faults by explicitly divulging dependencies. While efficient fault tolerance is never free, tasking enables more efficient resiliency schemes without requiring expensive memory-dump checkpoints, full replication, or application-oblivious message logging. As the runtime gains more knowledge of the data dependencies, fault tolerance schemes generally require less resources and can isolate rollback regimes when faults occur.

Chapter 2

FY19 Strategy

The strategy for FY19 is oriented toward isolating DARMA research efforts from engineering production-oriented AMT solutions. The past DARMA strategy has combined the two, leading to a stack that attains neither goal optimally. Thus, the FY19 milestones are focused on balancing the work between these high-level goals and partitioning the effort accordingly.

2.1 Summary of DARMA Deliverables

2.1.1 AMT Core Production-Oriented Runtime

Our first aim is to design, develop, and deploy core runtime software on top of MPI, slated for production hardening long term, that includes the minimal core features required to support foundational DARMA capabilities: overdecomposition, asynchronous event-driven execution, and load balancing. The programming and execution model will be derived based on lessons learned over the past few years of research experience with DARMA programming and execution models.

2.1.2 Particle-in-Cell Move Empire Integration

We will utilize the DARMA interface in Empire to overdecompose the particle move to enable load balancing. We shall also conduct extensive performance study on ATS-class platforms to compare Empire implementation with DARMA to pure MPI. Furthermore, we will perform a detailed study on the costs (in terms of engineering) to fully support all Empire features; and, develop a long-term plan for DARMA and Empire integration.

2.1.3 Load Balancing Algorithm Development

This is a research-oriented part of the FY19 work, to study develop, and empirically evaluate scalable load balancing algorithms for ATS-class systems with a focus on PIC and Mini-Multiscale imbalances. Certain classes of electromagnetic or solid mechanics applications have highly dynamic physics. To solve mission-relevant scientific problems requires either: (1) strong scaling to decrease time-to-solution or (2) weak scaling to solve larger problems. Failing to properly balance load can cause a single node's workload to either not decrease (for strong scaling) or increase beyond resource constraints (weak scaling). Load balancing requires cost models or heuristics that can simultaneously consider locality and computational load under constraints like total memory usage. Fundamental algorithms research is required on scalable methods for computing cost models and heuristics on large, distributed systems. Then, we will test the scalability of one new distributed persistence-based load balancing algorithm with PIC as the primary example of performance.

2.1.4 Performance Analysis and Runtime Optimization

This is a level of effort in FY19 that shall not be performed in a particular chronological order but, rather, will be performed in conjunction with all other tasks. Its goal is to analyze and evaluate sources of runtime overhead and quantify tasking overheads for tasking frameworks on ATS-class systems, as well as to model theoretically (via performance bounds) and experimentally (via performance tools) overhead mitigation techniques for key applications (e.g., SimplePIC, Mini-Multiscale) at scale on ATS-class systems. While tasking runtime systems provide performance portability and performance optimization, they inevitably can increase overheads for certain operations. For example, in a tasking runtime, a message send may be more costly due to the dynamic nature of work mapping to hardware. Key overheads shall be identified—via performance analysis and simulation/emulation—that arise from programming and execution models. Semantic overheads for more "automatic" programming models should be measured and techniques developed to mitigate their cost. Execution model overheads related to distributed-memory execution shall be measured against other models and analyzed with respect to their theoretical scalability—whether memory or time increases when increasing the number of nodes.

2.2 Empire-DARMA Integration Strategy

Empire is an electromagnetic and plasma physics code. Our integration efforts have focused on the particle-in-cell (PIC) part of Empire. At a high level, the PIC-move algorithm consists of the following stages:

1. Charged particles are displaced based on how the existing fields act on these particles;
2. Electric/magnetic fields and currents are weighted; and,
3. Finally, the solution phase during which new fields are computed, for use in the next iteration.

The Empire code is implemented with MPI and Kokkos (for on-node parallelism) utilizing a SPMD (single program, multiple data) programming model. The mesh is decomposed into spatial blocks with a set of particles that resides in that spatial region. The partitioning of the mesh is performed *a priori* by a decomposition tool. This tool creates discrete spatial blocks of the entire mesh that are mapped to a specific MPI rank. The mesh decomposition tool attempts to create partitions that are even across the MPI ranks. When the density of the particles is constant across the domain the problem is mostly load balanced. However, for problems where the particles are non-uniform (perhaps unpredictably so), a dynamic load balancer is required to balance the work across nodes.

During the move stage of the PIC algorithm, for a given timestep, particles move across the mesh blocks may be migrated across MPI ranks depending on a particle's current spatial locality. The CFL condition indicates the maximum distance any particles may travel depending on velocity and dt. Thus, during a given iteration, if the CFL (c) is greater than 1, a particle may travel across c mesh blocks (i.e., MPI ranks). The MPI code performs an all-reduce every micro-iteration to determine if particles have settled in their final mesh block or are continuing to move. Based on the result of this data-dependent all-reduce (i.e., the global count of particles sent out of their current block), all the mesh blocks may either perform another move micro-iteration or transition the next step—executing the solver.

For a subset of PIC problems, the particle distribution across mesh blocks may be imbalanced with respect to an initial particle-agnostic mesh decomposition. For the initial target problem in Q1, the particle distribution evens over time until it is fully balanced over the mesh blocks. During the solve phase of PIC, even for an imbalanced particle distribution, the amount of work is even because it operates per mesh block rather than per particle.

2.2.1 Particle Move Asynchrony

The MPI code divides the macro-iteration into n distinct micro-iterations, where n is upper bounded by the CFL value. During each micro-iteration, particles move within their current spatial block. If a particle ends up at a boundary, it gets pushed into a send buffer for that direction. The MPI ranks exchange particles and all-reduce to determine the total number of migrated particles. This operation occurs iteratively during each micro-iteration until the total number of migrated particles is zero. As the move progresses, the number of particles moving drops off sharply. When the problem is load balanced, the extra micro-iteration synchronization is not costly. However, for load imbalanced problems, each micro-iteration's bulk synchronization adds significant cost because some MPI ranks will sit idle waiting for others to process their incoming particles.

The particle move can be reformulated as a termination detection (TD) problem with particles moving recursively. In VT or the DARMA *Virtual transport* runtime, the arrival of incoming particles is not synchronized nor formulated as an "exchange" between mesh blocks. Instead, incoming particles arrive via an asynchronous handler triggered on a mesh block. Incoming particle handlers may be recursively triggered until termination is reached.

```
void particleHandler(ParticleMsg* msg) {
  auto const& particles = msg−>getParticles();
  // Push particles into ParticleContainer: 'this−>pc'
  moveRecur(this−>pc);
}
```

```
void moveRecur(ParticleContainer pc) {
  // dispatch Kokkos move kernel (with start_ offset)
  for (i = 0; i < num_neighbors; i++) {
    // Create 'msg' with particles outside of this mesh block for neighbor 'i'
    vt::theMsg()−>sendMsg<ParticleMsg,particleHandler>(msg);
  }
}
```

An aggressive implementation could dispatch the Kokkos move kernel for every incoming ParticleMsg that arrives for the mesh block with handler particleHandler. While this is correct, this will cause the average grain size to be fairly small. Thus, in a non-synchronized implementation how often the moveRecur is dispatched is tunable. A less aggressive implementation could wait until all messages on a given node are scheduled before executing the kernel (moveRecur). In the current implementation, we have not explored the tradeoffs of latency and grain size—but this is an interesting tunable part of the code that can be explored in the future.

Termination detection [3] guarantees, across a distributed system, that no more work can be created due to the causal relationship between send/receive for diffusing computations. Diffusing computations characterize a subset of distributed operations where messages must be causally related (e.g., disallowing arbitrary message sends not related to a message receive). Thus, each message has a distinct ancestor and the computation can be tracked by the underlying runtime system. Four-counter termination detection performs asynchronous reductions (with control messages) across the distributed system globally counting consumed c_k and produced p_k message counts, where k is the iteration of the global reduction. When these counts are equal $((c_{k-1} = p_{k-1}) \wedge (c_k = p_k) \wedge (c_k = c_{k-1}))$ across two subsequent TD reductions global termination is provably guaranteed for a diffusing computation [5].

Even without any overdecomposition, replacing all-reduces and distinct micro-iterations with termination detection provides more asynchrony. VT provides a component to efficiently perform termination on an *epoch*. An epoch provides a distinct grouping of related tasks by marking them with a distinct label. In VT, epochs are implemented as 64-bit IDs stored in the envelope of a message. The termination detection VT component tracks the produced/-consumed counts of a given epoch, allowing a user-defined continuation to be registered when termination over that epoch is detection. Thus, for the particle move in Empire, the problem of detecting termination becomes a distributed runtime problem rather than relying on synchronized user-defined checks for termination.

The first step in our work was developing MigrateParticlesVT which uses VT to send/receive particles (with a send handler) and replacing the reductions with creating a new epoch and waiting for termination of that epoch. The current implementation serializes the particles multiple times (with the Kokkos kernel PackParticles and VT's message serialization). We plan to optimize this part of the code by using VT RDMA (Remote Direct Memory Access) module to perform direct transfers on a send buffer without packing/unpacking the particles considering that BasicParticle is byte serializable.

2.2.2 Mesh Overdecomposition

In general, overdecomposition involves dividing the computation into a number of units that is greater than the number of processors (or MPI ranks), thus yielding multiple work and data units assigned to each processing element. Overdecomposition enables overlapping communication and computation along with allowing a load balancer to remap medium-grained pieces to attain a good load balance. The efficacy of overdecomposition is limited by the increased memory usage related to the number of overdecomposed units (per rank) as well as scheduling overheads.

In contrast to bulk synchronous models, where concurrent execution flows through a sequence of MPI collectives and send/receive pairings, active-message-based execution triggers work when a message arrives on a node. By virtualizing the node with active messages, work and data units can be migrated by the runtime dynamically—no longer confining data to a fixed physical location.

To overdecompose the particle move of Empire, each mesh block would need to be further split starting from the initial decomposition performed out-of-band (before execution by the `decomp` tool). However, the work of finding the crossing point for particles (in the main move kernel) would require substantially more complex multi-level logic: first, across the initial domain decomposition; and, then, across the newly sub-divided domain. For the work documented in this report for FY19-Q1, we chose to *replicate* the initially decomposed mesh blocks to extract more concurrency rather than initially implementing a 2-level partitioning scheme.

Replication

Replication is possible for Empire PIC because as the particles move, and deposit a charge, they do not interact with each other (currently no collide). This means that a mesh block can be copied k times, possibly migrated to another node, and any particle that moves within the spatial region of that mesh block can safely choose any replicant (for that spatial region) to execute (they are equal from a correctness perspective). After the move completes, all replicants for a given spatial region, accumulate their locally deposited charge in a reduction to compute the total point-charge values.

Depending on the density of particles, the number of replicants for a certain area in space can vary to increase the computation resources assigned to that spatial region. In theory, this enables a very dynamic workflow—replicants can only be created when absolutely required to rebalance the load profile. Our implementation in Q1 only allows for a constant k across the domain, but we plan to allow for k to be variable across the domain and across time as we improve the implementation during the rest of the year.

2.2.3 Replicated Mesh Collection in DARMA

The VT package (in the DARMA toolkit) offers a *collection* abstraction to express an indexable, distributed set of objects, which can be arbitrarily indexed. A collection can be created and managed in a rooted or collective manner (similar to a set of MPI ranks). Each element in a VT collection is a C++ object. Messages can be sent between C++ objects, targeting a certain index or the whole collection (a broadcast), or part of the collection (a multicast). Reductions can be performed across the entire collection or a section of it.

The replicated mesh block collection uses a 2-D index. The first dimension is the original MPI rank that is being replicated. The second dimension is $(0, k]$—which replicant for a given mesh block the collection element represents. The collection is created collectively: each MPI rank creates all k replicated mesh blocks, instantiating k elements that have the data copied into them to perform the replicated computation.

Chapter 3

Empire-DARMA Implementation

The following sections describe our implementation of DARMA in Empire.

3.1 Empire Tasking

Much of our work in Q1 of FY 2019 was focused on refactoring Empire-PIC and Empire to allow the particle move portion of the PIC algorithm to be executed as a set of VT tasks. This required seralizing `ParticleLists::moveParticles` which involved a number of other structures in Empire as well as STK (Sierra Toolkit Mesh) types. To do this we refactored a number of Empire and Empire-PIC types to: (1) serialize types and operations involved in the particle move; (2) allow for subdividing particles and maintaining their relationship to the mesh partition that contains them; (3) and, adding new classes that facilitate use of DARMA. We approached this work in stages, each of which we tested as they were completed.

DARMA-VT uses serialization as a means of moving data and operations off-node. Empire was not written with serialization in mind, and if not for integration with DARMA it is not likely that serialization would have become a design goal (beyond the need to checkpointing). To overcome this, we worked to recursively serialize Empire data structures related to the particle move. Thus, we traced through the particle move operations and identified all types and kernels involved in the particle move. Obstacles to serialization were largely types not part of Empire that we could not easily refactor—especially STK types. Fortunately, Empire employs a class that serves as an proxy mesh class an we were able to eliminate references to STK types in the move algorithm altogether. The benefit of this work is two-fold:

1. Empire has a cleaner code base; and,
2. We are able to serialize an entire mesh required for the particle move phase.

The next step is to further subdivide the particle move work for cases in which the simulation becomes heavily unbalanced such that many particles are in one portion of the mesh. Our Q1 work takes a simple approach to the problem in that m particles are distributed to n DARMA-VT tasks (with $\frac{m}{n}$ particles in each) and each sub-partition gets a copy of the entire mesh partition. Admittedly, other mesh partitioning schemes may be more efficient here, but given the goal of completing this preliminary work in Q1 this will be done in follow-on work. This freed our team from mesh partitioning tasks to pursue major contributions to DARMA-VT itself as well as refactoring Empire-PIC for serialization. Later work will investigate how sub-partitioning the mesh can impact performance of the VT implementation by reducing memory usage. Give the progress above, we added some abstractions to vary the way VT executes a particles move, particularly in terms of what data is sent back to the governing MPI rank and whether particles remain in VT or are communicated back to the governing MPI process when the enter a different mesh partition.

3.2 VT Runtime

Over the last year, the DARMA team has developed *Virtual Transport* (or VT), a new runtime that abstracts MPI ranks as distinct C++ objects (either singletons or collections) with efficient data transfer with active messaging and RDMA. VT was designed to be a highly interoperable runtime interface that easily composes with MPI and Kokkos (by utilizing OpenMP for threading). With this functionality, VT provides load balancing on the overdecomposed/virtualized abstractions by enabling object migration across the system.

In Q1, our team implemented a new version of the the particle move control code in Empire in terms of a VT object collection rather than across MPI ranks.

3.3 Serialization

The majority of Empire data structures are wrapped with a `Kokkos::View<T>` enabling PIC kernels to be mapped to the CPU or GPU—providing performance portability. Thus, the bulk of serialization work in Q1 was implementing a general serializer for any `Kokkos::View<T>`. Kokkos views in general are n-dimensional, consisting of k static (prefixed) dimensions and $n - k$ dynamic dimensions. For example, `Kokkos::View<T**[16][10]>` contains 2 static dimensions (of size 16 and 10) and 2 dynamic dimensions which are specified at runtime.

VT utilizes a serialization library called checkpoint. The checkpoint library provides general serialization functionality to traverse arbitrary C++ data structures in a (non-)intrusive manner depending on usage requirements.

For example, if the user has an object A:

```
struct A {
  std::vector<int> x, y;
};
```

The object may be non-intrusively serialized by declaring the following ADL (Address Dependent Lookup) overload of the `serialize` method:

```
template <typename SerializerT>
void serialize(SerializerT &s, A& a) {
  s | a.x | a.y;
}
```

The checkpoint library provides efficient default serialization of all the C++ STL data structures commonly used.

To serialize a `Kokkos::View<T**[16][10]>`, we implemented a serialization overload for any `Kokkos::View` and `Kokkos::DynamicView`. For each dynamic dimension (out of $n-k$ dimensions), checkpoint obtains the span i_{span} for dimension i, serializing the size for reconstruction. For debugging/performance analysis Kokkos allows the assignment of an arbitrary (non-distinct) label (typed as a C++ string) for a given Kokkos view. The label is serialized along with the dynamic extents as meta-data for the Kokkos view.

In the checkpoint library we have implemented robust (intended for eventual production use) serialization code for `Kokkos::View<T>` and `Kokkos::DyanmicView<T>`. To test this capability, we have implemented 412 unit tests to test the serialization of many types of `Kokkos::View` types, ranging from 0-D to 3-D, including `const` and non-`const` data types. Additionally, we implemented a few more complex integration tests that utilize MPI to actually migrate the serialized view data and test equality of the view data on a different MPI rank after sending it over the network.

After serializing the meta-data, checkpoint traverses the data segments serializing the value of each elements. When the data is continuous, the **checkpoint** serializer allows taking a pointer and range to efficiently memcpy it into the system serialization buffer.

For all the data structures in Empire that may be migrated we have implemented non-intrusive serializers. These include (but not limited to) the following Empire data structures along with all other data structures that are recursively reachable:

- `empire::pic::ParticleContainer`
- `empire::pic::PICMeshInteraction`
- `empire::pic::ElementalParticleData`

To ensure correctness of the Empire-PIC serialization code, we have written extensive unit tests in Empire that recursively test all the new serialization code.

Chapter 4

Performance Experiments and Results

We now present the empirical result obtained with our Empire-DARMA implementation.

4.1 Timings

Experiments were conducted on the Kahuna system at scales of 1, 2, 4, 8, and 16 nodes. Each node of Kahuna has 2 Haswell processors and 256 GB of memory. We executed the code with 2 processes per node, 14 threads per process (one per core, without hyperthreading), with processes pinned to separate sockets and threads pinned to separate cores within the process's socket. This was found to provide the best performance in both the baseline MPI code and various configurations of the VT code.

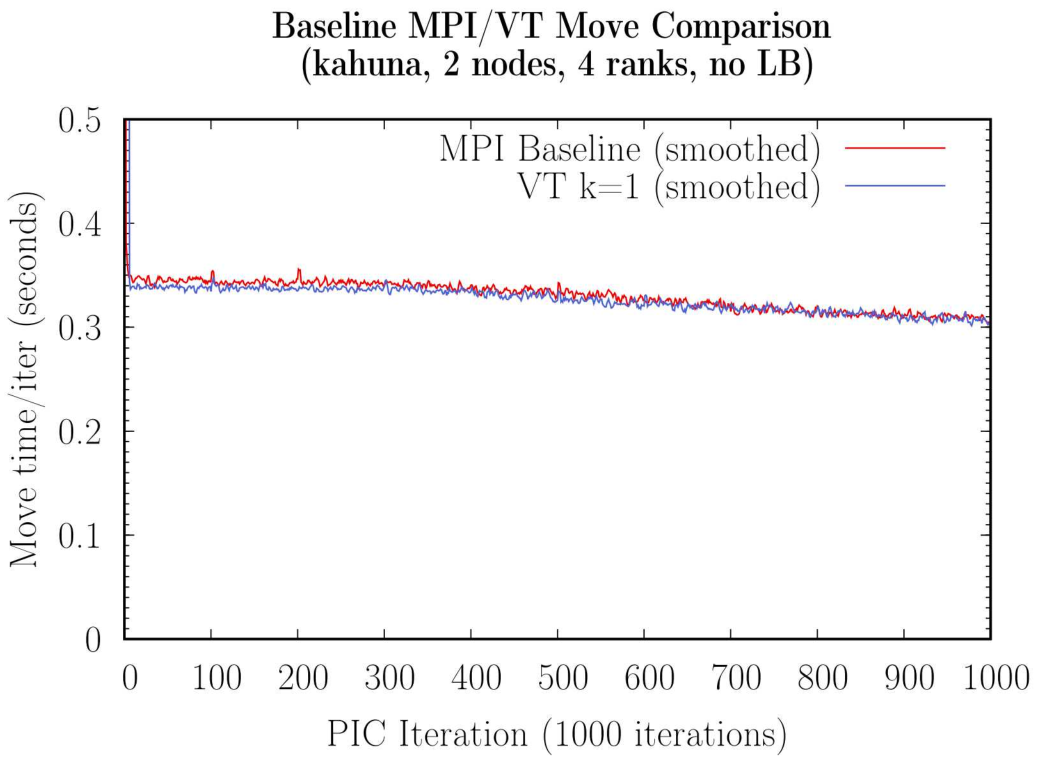
A comparison of per-iteration move times between the baseline MPI runs and the basic VT runs with no overdecomposition can be seen in Figures 4.1 and 4.2. Note that these and later figures show moving averages of individual iteration times where necessary to improve legibility. These illustrate that the VT code is an effective, faithful implementation. We believe the small offset between the MPI and VT codes is due to missing optimizations in the VT implementation. The update phase in the VT code can be further optimized to fuse together the "pre-move" operations without imposing extra synchronization. We find that the performance effects of overdecomposition alone range from small benefits to severe slowdowns.

We subsequently explored a range of replication factor parameters at each scale. Where a non-unity factor out-performs the basic non-overdecomposed run, on 1, 2, and 8 nodes, we can see the small impact of the best choice in Figure 4.3.

Furthermore, we show the overall scaling performance of the code with the best choices for overdecomposition factor at each scale in Figure 4.5. Activating load balancing provides substantial performance benefits at all scales. It also changes the ideal overdecomposition factor at several job size scales. At small scales of 1 and 2 nodes, the impact is nearly equivalent to running on 2 or 4 nodes, respectively, without the benefit of load balancing.

Confounding factors:

- For all of these runs, we used a mesh as decomposed for the target number of processes by the Trilinos `decomp` utility with its default arguments. We believe that variability in the division of work this created strongly and unpredictably affects the results obtained.
- The provided test problem presents an initially severely imbalanced distribution of particles throughout the simulation domain, that evolves toward a more uniform distribution as the run progresses. This gives the VT based code a single up-front opportunity to find a good assignment of work to processors, but makes dynamic adaptation more challenging. A more involved problem with ongoing, shifting imbalance would more strongly showcase the capabilities of a runtime-mediated measurement-based load balancing framework.



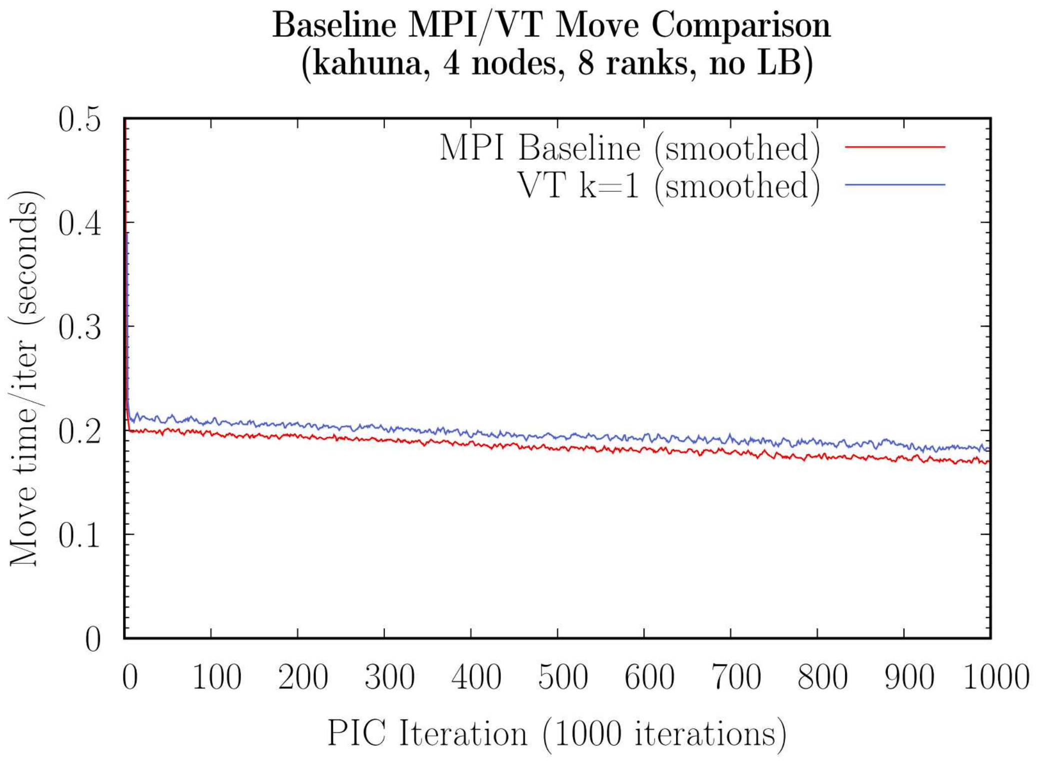
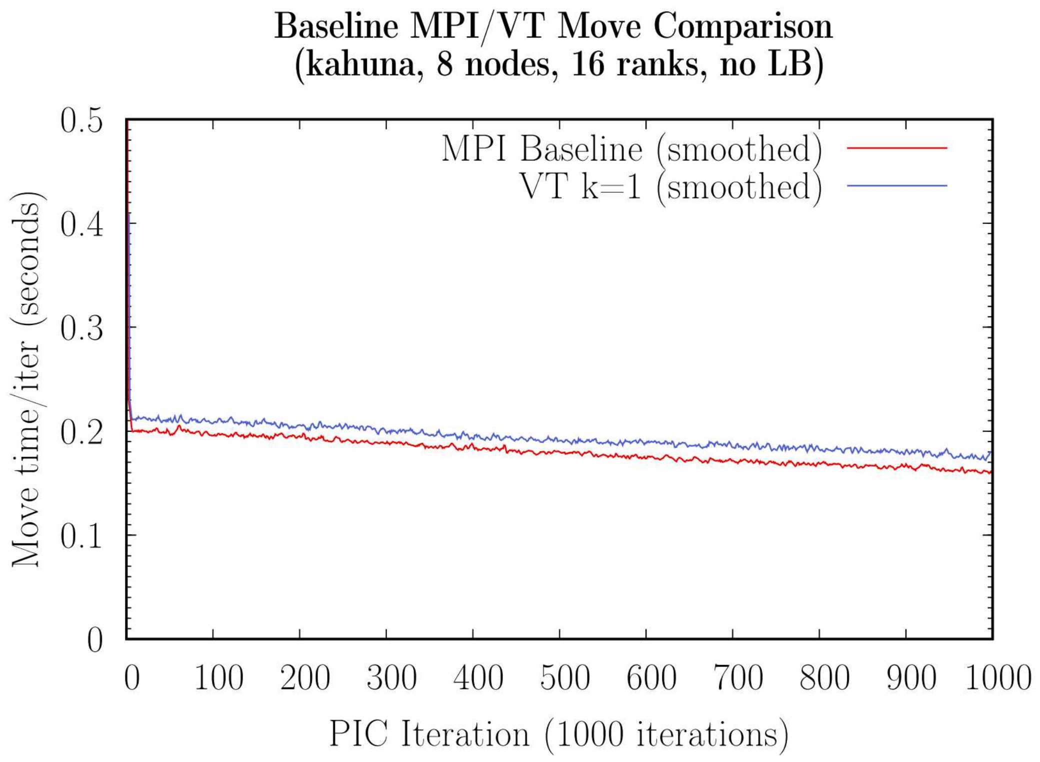


Figure 4.1. Comparison of per-iteration move times between the Baseline MPI runs and the basic VT runs without overdecomposition, for 2 and 4 nodes.



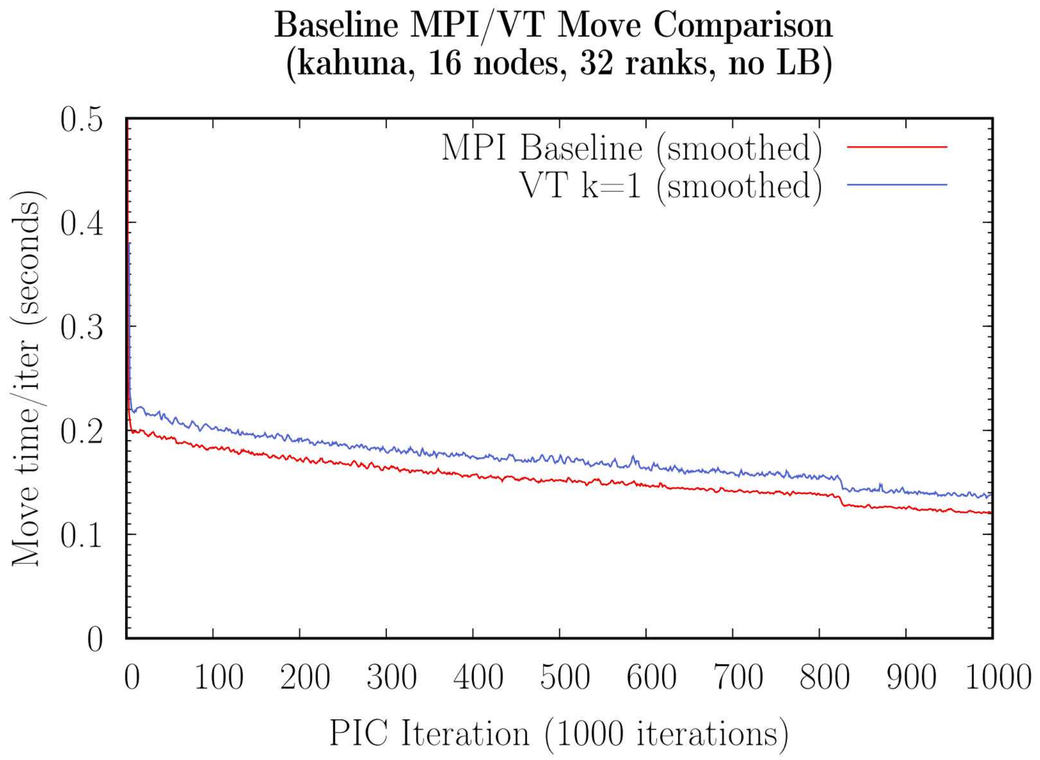
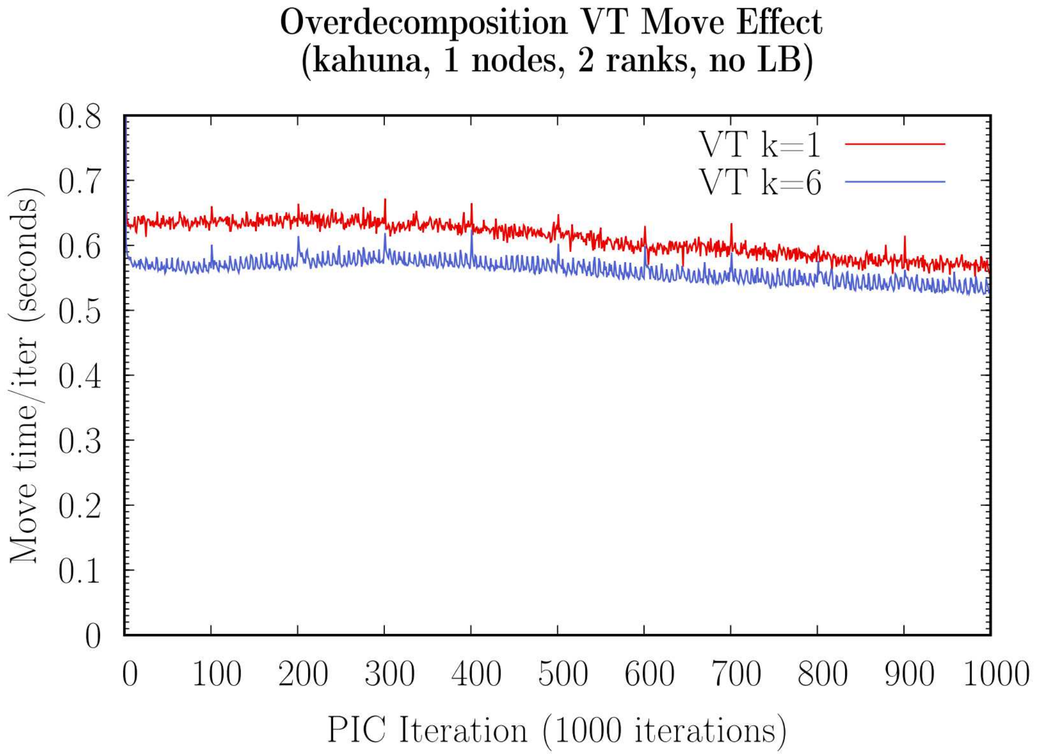


Figure 4.2. Comparison of per-iteration move times between the baseline MPI runs and the basic VT runs without overdecomposition, for 8 and 16 nodes.



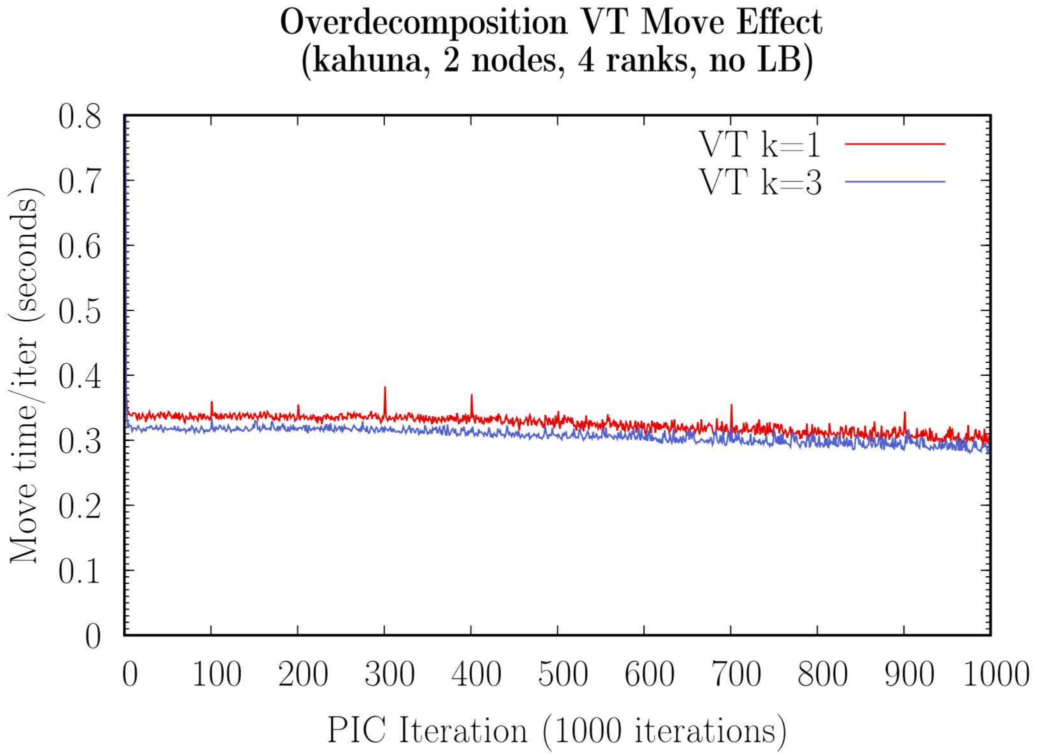


Figure 4.3. Comparison of replication factor parameters on 1, 2 nodes.

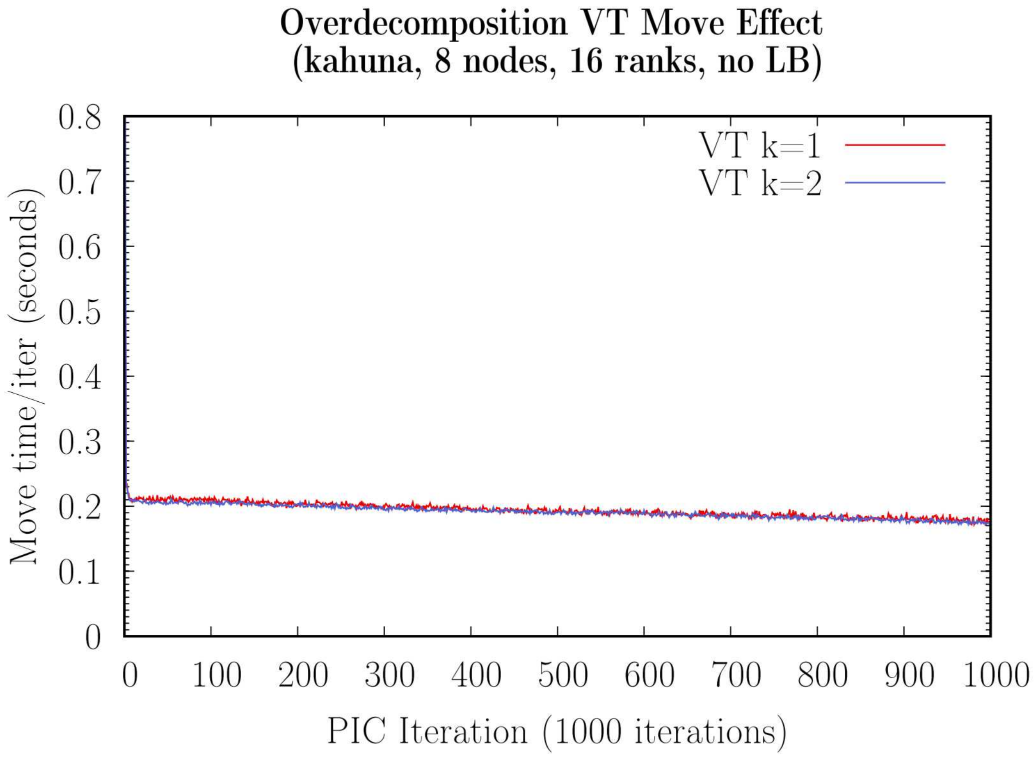


Figure 4.4. Comparison of replication factor parameters on 8 nodes.

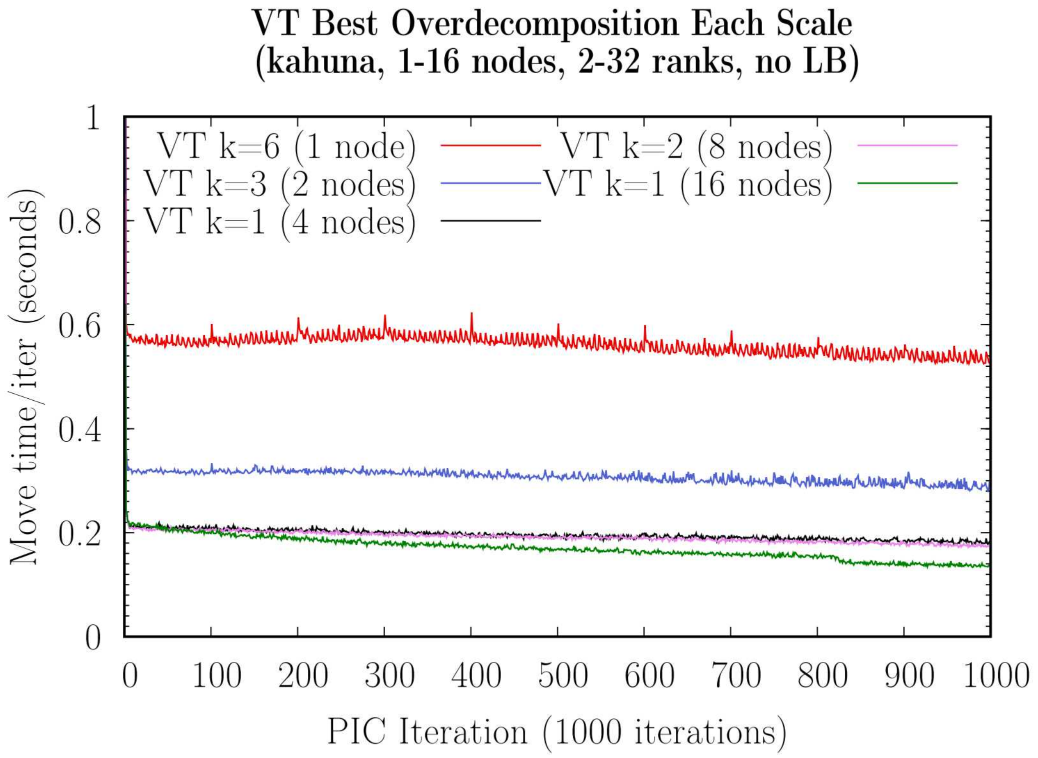


Figure 4.5. Overall scaling performance with 1 to 16 nodes.

4.2 Scalability

Scalability when taking into account load-balancing.

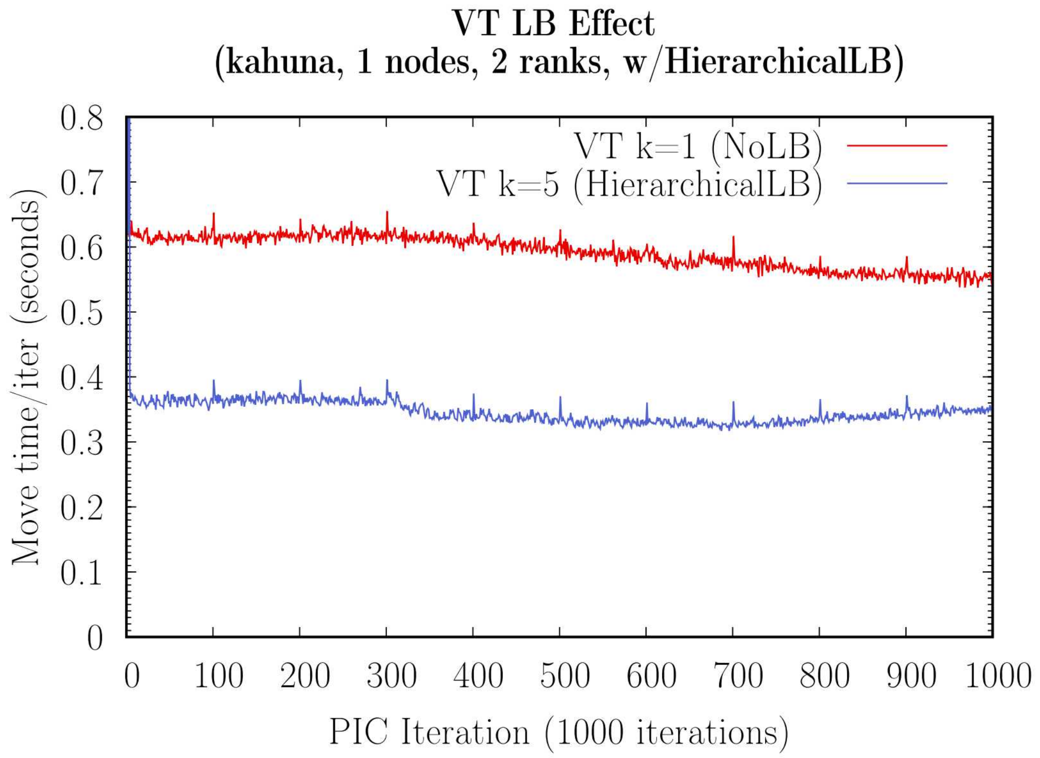
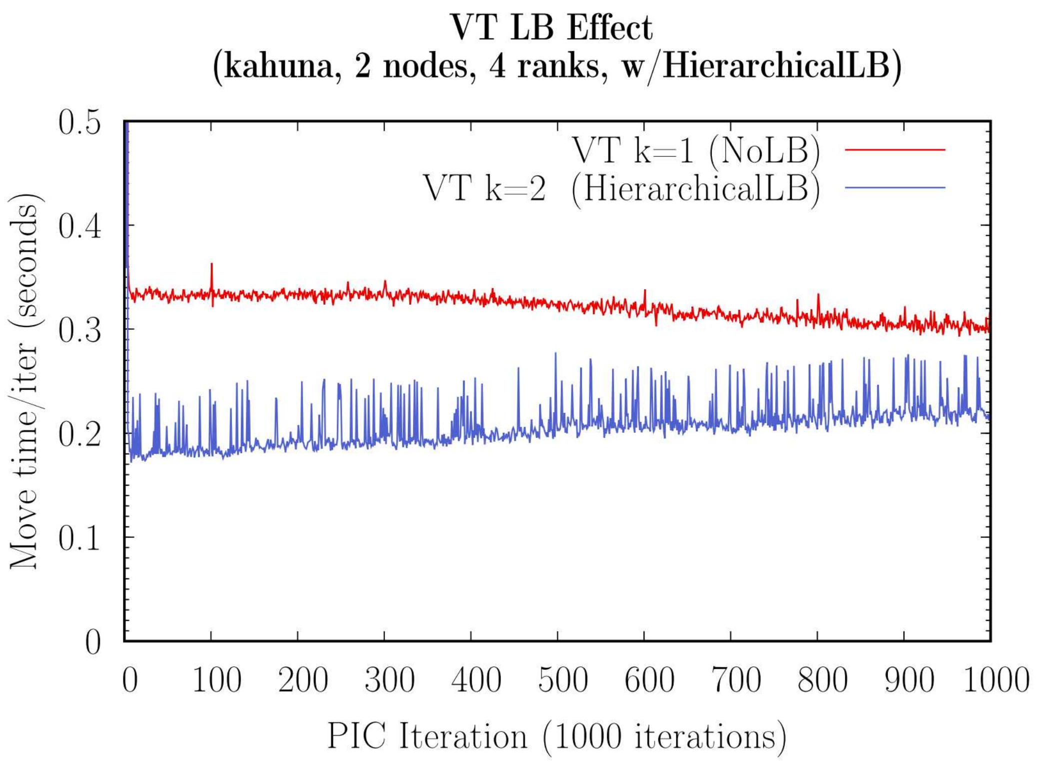


Figure 4.6. Effect of load balancing with a single node.



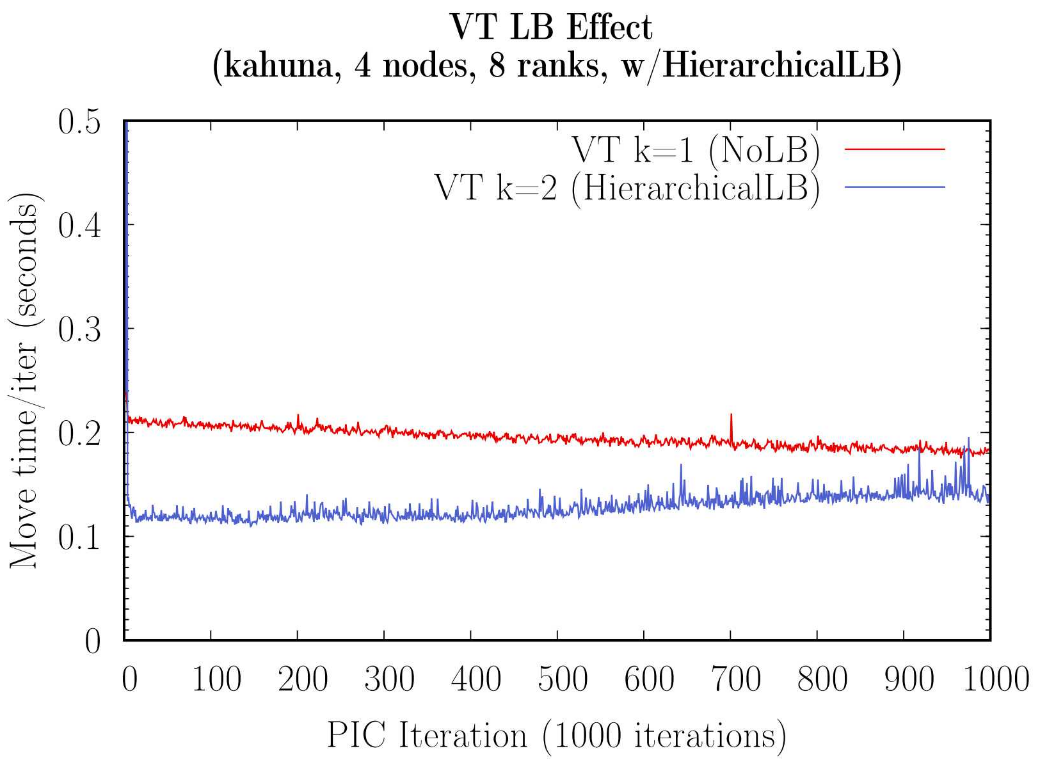
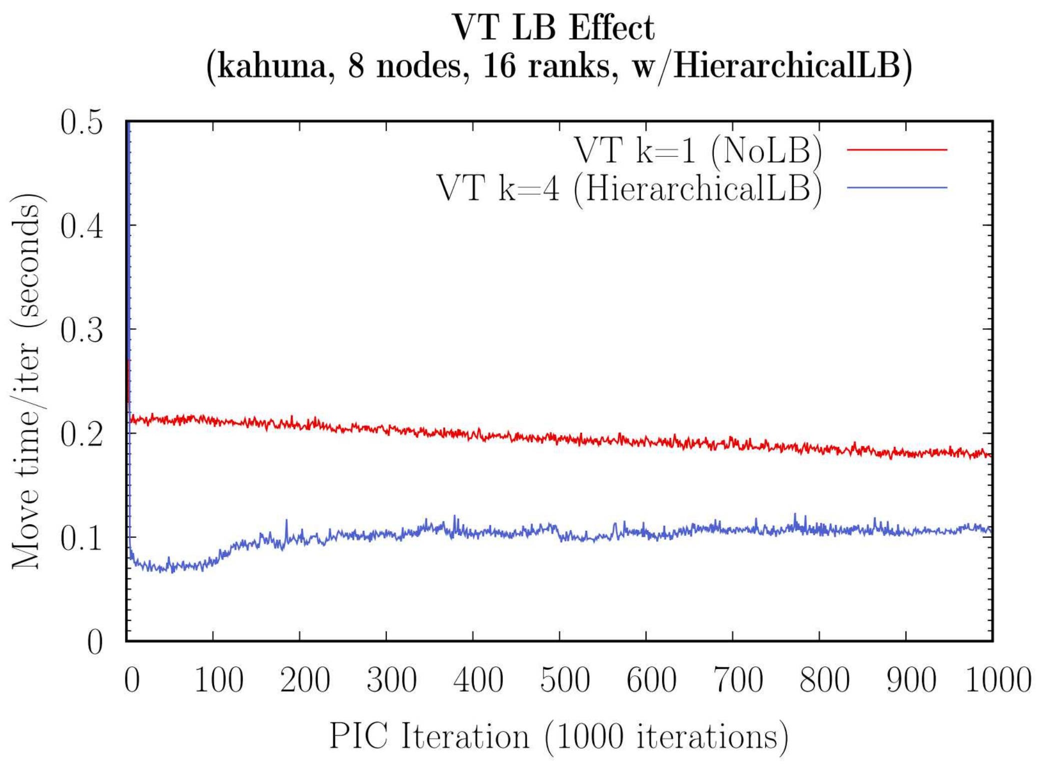


Figure 4.7. Effect of load balancing with 2 or 4 nodes.



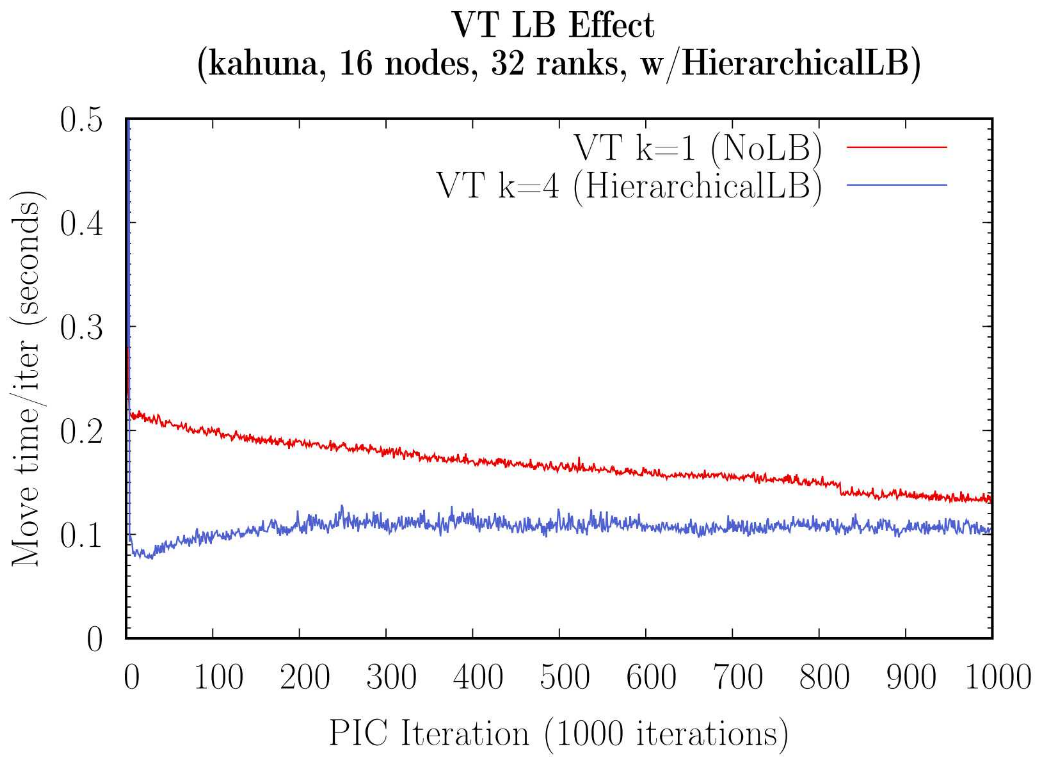


Figure 4.8. Effect of load balancing with 8 or 16 nodes.

Chapter 5

Conclusion

We conclude this report with a brief summary of what we have done so far and what we will do next.

5.1 FY19 Completed Work To-Date

In Q1 we have implemented a proof-of-concept taskification of Empire-PIC for the particle move phase using the VT library in the DARMA toolkit. Our implementation utilizes replication of the mesh blocks to create concurrent particle-move tasks that can be rearranged by the runtime—load balancing them across MPI ranks. Replication increases memory pressure and induces significant expense in updating per-iteration data for each replicant.

Even with these overheads and an un-tuned particle move implementation in VT, our empirical results show great promise—demonstrating that the load balancer can improve performance for an iteration up to 2x when the problem is significantly imbalanced.

5.2 Future Work

In the remainder of FY19, we plan to tune the VT implementation and reduce the memory pressure by actually splitting the mesh data into smaller blocks. Furthermore, we plan to dynamically overdecompose the problem instead of using a uniform decomposition across the entire domain. Thus, when load imbalance arises in a subset of the domain we can dynamically create smaller mesh chunks to effectively balance load without paying a high cost for the entire domain. Thus, the any extra costs imposed will be proportional to the benefit expected from balancing that part of the domain.

References

[1] Bilge Acun, Phil Miller, and Laxmikant V. Kale. Variation among processors under turbo boost in hpc systems. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 6:1–6:12, New York, NY, USA, 2016. ACM.

[2] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. Run-to-run variability on xeon phi based cray xc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 52. ACM, 2017.

[3] Edsger W Dijkstra and Carel S Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[4] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V Kale. Work Stealing and Persistence-based Load Balancers for Iterative Overdecomposed Applications. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 137–148. ACM, 2012.

[5] Friedemann Mattern. Algorithms for distributed termination detection. *Distributed computing*, 2(3):161–175, 1987.

[6] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5 d matrix multiplication and lu factorization algorithms. In *European Conference on Parallel Processing*, pages 90–109. Springer, 2011.

[7] Gengbin Zheng, Abhinav Bhatele, Esteban Meneses, and Laxmikant V. Kale. Periodic Hierarchical Load Balancing for Large Supercomputers. *International Journal of High Performance Computing Applications (IJHPCA)*, March 2011.

DISTRIBUTION:

1	MS	N/A	J. Lifflander, 08753
1	MS	N/A	M. Bettencourt, 01352
1	MS	N/A	N. Slattengren, 08753
1	MS	N/A	G. Templet, 08753
1	MS	0899	Technical Library, 8944 (electronic copy)

v1.40

Sandia National Laboratories