SANDIA REPORT

SAND2017-9980
Unlimited Release
Printed September 2017

ASC ATDM Level 2 Milestone #6015: Asynchronous Many-Task Software Stack Demonstration

Janine C. Bennett, Matthew T. Bettencourt, Robert L. Clay,
Harold C. Edwards, Micheal W. Glass, David S. Hollman,
Hemanth Kolla, Jonathan J. Lifflander, David J. Littlewood,
Aram H. Markosyan, Stan G. Moore, Stephen L. Olivier,
J. Antonio Perez, Eric T. Phipps, Francesco Rizzi,
Nicole L. Slattengren, Daniel Sunderland, Jeremiah J. Wilke

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

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology and Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2017-9980
Unlimited Release
Printed September 2017

ASC ATDM Level 2 Milestone #6015: Asynchronous Many-Task Software Stack Demonstration

Janine C. Bennett, Matthew T. Bettencourt, Robert L. Clay,
Harold C. Edwards, Micheal W. Glass, David S. Hollman,
Hemanth Kolla, Jonathan J. Lifflander, David J. Littlewood,
Aram H. Markosyan, Stan G. Moore, Stephen L. Olivier,
J. Antonio Perez, Eric T. Phipps, Francesco Rizzi,
Nicole L. Slattengren, Daniel Sunderland, Jeremiah J. Wilke

Abstract

This report is an outcome of the ASC ATDM Level 2 Milestone 6015: Asynchronous Many-Task Software Stack Demonstration. It comprises a summary and in depth analysis of DARMA and a DARMA-compliant Asynchronous Many-Task (AMT) runtime software stack. Herein performance and productivity of the overall approach are assessed on benchmarks and proxy applications representative of the Sandia ATDM applications. As part of the effort to assess the perceived strengths and weaknesses of AMT models compared to more traditional methods, experiments were performed on ATS-1 (Advanced Technology Systems) test bed machines and Trinity. In addition to productivity and performance assessments, this report includes findings on the generality of DARMA's backend API as well as findings on interoperability with node-level and network-level system libraries. Together, this information provides a clear understanding of the strengths and limitations of the DARMA approach in the context of Sandia's ATDM codes, to guide our future research and development in this area.

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525. Funding was provided as part of the Advanced Technology Detection and Mitigation (ATDM) program funded as part of the Advanced Simulation and Computing (ASC) program of the National Nuclear Security Administration (NNSA). Additional contributions come from the Exascale Computing Program (ECP) jointly funded by the U.S. Department of Energy (DOE) Office and Science and NNSA.

Access to Trinity and Mutrino systems was obtained through the Advanced Computing at Extreme Scale (ACES) program under allocation 137908. Special thanks goes to Ann Gentile, Jason Repnik, Nathan Gauntt, Jim Brandt, Joel Stevenson, David Kratzer, and Peter Lambhorn for technical support throughout the project. Additional runs were performed at the National Energy Research Scientific Computing Center (NERSC).

Special thanks goes to those who have provided insight through technical discussions throughout the full development of project including Phil Miller, Sanjay Kale, Sean Treichler, Timothy Mattson, David Richards, and Jeff Keasler.

Contents

List of Figures

List of Tables

Executive Summary

Introduction and Milestone Description: This milestone report presents an in-depth analysis of DARMA and a DARMA-compliant asynchronous many-task (AMT) runtime software stack to inform the Sandia Advanced Simulation and Computing (ASC) code strategy in regards to next generation platforms. As written in the ASC Implementation Plan, the milestone description is as follows:

Next generation platforms (NGP) will require us to fundamentally rethink our programming models due to a combination of factors including extreme (e.g., billion-way) parallelism, data locality issues (for managing both performance and energy usage), and resilience (e.g., application-level fault tolerance). As seen in the PSAAP II program projects, and in the broader computational sciences community, the asynchronous, many-task programming model is emerging as a leading paradigm, with many variants of this model being proposed. However, although this paradigm presents tremendous potential to address key challenges posed by NGP, an FY15 ATDM L2 milestone (#5325) highlighted gaps with respect to productivity in leading AMT runtimes. In particular, the work done for milestone #5325 highlighted requirements gaps and deficiencies in existing application programming interfaces (APIs).

DARMA *is an AMT portability layer and specification effort aimed at addressing gaps identified in milestone #5325. Specifically,* DARMA *is designed to 1) insulate application from runtime system and hardware idiosyncrasies, 2) improve AMT runtime programmability by co-designing an API directly with application developers, 3) synthesize application co-design activities into meaningful requirements for runtimes, and 4) facilitate AMT design space characterization and definition, accelerating the development of AMT best practices.*

This milestone will evaluate a DARMA*-compliant AMT runtime software stack comprising ATDM ASD software components and existing AMT runtime technology (e.g., Charm++). We will assess the performance and productivity of this software stack on kernels and proxy applications representative of the Sandia ATDM applications. As part of the effort to assess the perceived strengths and weaknesses of AMT models compared to more traditional approaches, experiments will be performed on test bed machines and one or more ATS-x system (target is Trinity). The outcomes of this effort will include 1) an initial* DARMA*-compliant AMT software stack, 2) a clear understanding of the strength and limitations of the co-designed AMT API and the associated runtime implementation in the context of SNL's ATDM codes, 3) and information to guide our future R&D in this area (e.g., system performance issues, portability issues, usability issues on realistic apps, etc.).*

Specific deliverables include:

1. *Initial implementation of an integrated* DARMA*-compliant AMT software stack (including ATDM ASD components) on NGP tests beds and one or more ATS-x system (intent is to use Trinity).*
2. *Implementation of ATDM application kernels and proxies developed for the AMT system.*
3. *An analysis of the productivity, performance, scalability, and dynamic load balancing capability for the* DARMA*-compliant runtime on those ATDM application kernels and proxies.*
4. *A report to inform the code development road map guiding the (Sandia) ASC code strategy.*

Summary of Work and Findings: The DARMA abstraction layer facilitates the expression of tasking while mapping to a variety of underlying AMT runtime systems. It eases the shift from an imperative programming style, in which programmers use explicit statements to control their program state, towards a declarative regime, in which the programmer expresses algorithm logic while leaving control-flow management to an underlying runtime system. This milestone report presents an assessment of the efficacy of the overall DARMA approach.

Over the course of this milestone, a fully-distributed CHARM++ backend was developed, that served as the focus of performance and productivity assessments. Prototype HPX and ONNODE backends served as development tools, enabling a more thorough evaluation of the generality of the DARMA's backend API and of interoperability challenges with node-level libraries (using KOKKOS, our focus was interoperability with OpenMP on ATS-1). Three benchmarks, written by DARMA team members, were developed to investigate benefits and limitations of the approach. Three proxy applications, written by application developers, served to aide in the co-development of APIs, acquisition of subjective feedback, and application requirements for use cases important to the ASC/ATDM program. Using the DARMA-CHARM++ software stack, many 1000s of runs were performed with the benchmarks and proxies on ATS-1 systems (Trinity and Mutrino) on both Haswell and KNL partitions using two compilers: GCC6.3.0 and ICC18.0.0beta. On Mutrino, scaling and profiling studies were performed up to 64 nodes (2K cores Haswell), (4K cores KNL) and on Trinity KNL, scaling studies up to 2K nodes (131K cores) were performed. Productivity was evaluated via subjective feedback gathered from application developers and an assessment of semantic information gain in DARMA program specification over an imperatively written code.

The following findings emerge: Performance experiments up to 2K nodes on Trinity demonstrate the scalability of the DARMA-CHARM++ backend. Although DARMA's deferred execution and task model imposes overheads over imperative MPI implementations (typically 10-20% for balanced use cases), the performance gains for irregular, load-imbalanced problems are significant, while requiring minimal effort from the application developer. From a productivity perspective, DARMA's declarative programming style enabled three-capabilities to be managed by the DARMA-CHARM++ backend: 1) communication/computation overlap, 2) tunable granularity in which data decomposition did not necessarily match the execution resources, and 3) load balancing. Although, by design, DARMA's backend API captures a declarative representation of an application that can map to a variety of runtime implementations, this year's work highlighted opportunities for componentization within both the frontend and backend that could result in improved scheduling and optimization strategies long-term. Interoperability with node-level libraries was found to be complicated as, similar to other AMT frameworks, CHARM++ is centered around explicitly managed Pthreads. In particular, difficulties must be managed when using, *e.g.*, the OpenMP affinity layer for resource management and arbitration between the AMT runtime and node-level libraries.

Impact and Path Forward: Overall, this milestone report clearly identifies the overarching strengths and limitations of DARMA and the DARMA-CHARM++ backend in the context of several Sandia ATDM use cases. Together with the initial DARMA-compliant stacks that have been developed, the accumulated information highlights the potential of the overall approach while clearly identifying focus areas for future work. These include productizing and hardening efforts such as productivity tools (*e.g.*, timers, performance profilers, debugging aides), tuning of critical overheads (which can be ameliorated in part via optimizations in collectives and certain task scheduling operations), general documentation, and testing. Focused research and development on interoperability with node-level libraries is crucial and will play a major role in next year's efforts. Lastly, there are a number of opportunities for componentization that present opportunities for optimization strategies and, perhaps more importantly, increase the participatory nature of the DARMA architecture. Enabling and garnering broader community engagement would facilitate development of AMT best practices, which are ultimately required for vendor-supported solutions long-term.

Chapter 1

Introduction

1.1 Programmatic History

Within the Department of Energy's National Nuclear Security Administration (DOE/NNSA), the Stockpile Stewardship Program (SSP) is an integrated technical program for maintaining the safety, surety, and reliability of the U.S. nuclear stockpile. The Advanced Simulation and Computing (ASC) provides the SSP with simulation capabilities and computational resources that enable the qualification and certification of the nation's nuclear deterrent in the absence of underground, full-system nuclear weapons tests. ASC comprises 6 sub-programs: Integrated codes (IC), Physics and Engineering Models (PEM), Verification and Validation (V&V), Computational Systems and Software Engineering (CSSE), Facility Operations and User Support (FOUS), and Advanced Technology Development and Mitigation (ATDM). Established in 2014, ATDM, the newest of the subprograms, is focused on mitigating the risk that current simulation capabilities will not be able to effectively utilize future high performance architectures. The ATDM subprogram sees application developers work closely with computer scientists on next generation Code Development and Applications (CDA) and Architecture and Software Development (ASD) activities.

Sandia's CDA effort is developing two production-prototype applications that are performant on emerging new computer architectures (including exascale) and are suitable for guiding future application development and changes to current production applications. In addition to being designed for performance portability on next generation platforms (NGPs), next generation simulation (NGS) capabilities are also being developed and included in the new applications. One such new capability is embedded analysis for sensitivities, uncertainty quantification (UQ), and optimization. The teams leverage a component-based software design philosophy to enable agility, re-use, and reduced costs. Research and development activities aim to address challenges posed by next generation platforms, while simultaneously enabling new design and decision support capabilities for nuclear weapons applications. Sandia's ASD research and development efforts can be largely grouped into two categories: 1) performance abstractions (including node-level performance portability, dynamic asynchronous tasking, and data-management), and 2) embedded analysis (geometry, meshing, UQ, optimization, and analytics, and multi-scale, multiphysics support).

The two application spaces are:

1. "Next Generation Electromagnetics Simulation of Hostile Environment"
   (a) Self consistent simulation from a hostile builder device, radiation transport, plasma generation and propagation to NW system circuits, cables and components with uncertainties.
   (b) Coupled Source Region ElectroMagnetic Pulse (SREMP) to System Generated ElectroMagnetic Pulse (SGEMP) simulation. Physical spatial domain on the order of kilometers down to system geometry down to millimeters.
   (c) Efficient radiation transport and air chemistry through Direct Simulation Monte Carlo (DSMC)

in rarified domains and condensed time history in thick regions.
   (d) Embedded sensitivity analysis, uncertainty quantification and optimization.
2. "Virtual Flight Testing for Hypersonic Re-Entry Vehicles"
   (a) Virtual flight test simulations of re-entry vehicles from bus separation (exo-atmospheric) to target for normal and hostile environments.
   (b) DSMC-based simulation of exo-atmospheric flight regime with hand-off to continuum Navier-Stokes at appropriate altitude.
   (c) Time-accurate wall-modeled LES of high Reynolds number (100k-10M) hypersonic gas dynamics.
   (d) Fully-coupled simulation of re-entry vehicle ablator/thermal (shape change, ablation products blowing) and structural dynamic (random vibration) response.
   (e) DNS and DSMC enhanced reacting gas models and turbulence models via a-priori and on-the-fly model parameter calculations.
   (f) Embedded sensitivity analysis, uncertainty quantification and optimization.

This milestone report summarizes Sandia's recent research and development regarding asynchronous many-task (AMT) programming models and runtime systems. Findings and lessons learned are presented as they pertain to our ATDM applications, embedded analysis, and other performance abstraction capabilities.

1.2 Exascale Drivers

The shift from petascale to exascale computing is introducing fundamental changes in high performance computing architectures, including increased heterogeneity, extreme (e.g., billion-way) parallelism, data locality issues (for managing both performance and energy usage), and resilience (e.g., application-level fault tolerance) [3, 4]. These changes are driving application developers to reconsider how they specify and execute their algorithms so as to make productive and performant use of future architectures.

To elaborate on this more precisely, we first define a few terms. A parallel programming model is an abstract view of a machine and set of first-class constructs for expressing algorithms. A programming model focuses on how problems are decomposed and expressed. A parallel execution model specifies how an application creates and manages concurrency. A runtime system primarily implements portions of an execution model, managing how and where concurrency is managed and created. Runtime systems therefore control the order in which parallel work (decomposed and expressed via the programming model) is actually performed and executed.

Communicating sequential processes (CSP) is the most popular concurrency execution model for science and engineering applications. We note that often people use the term Message Passing Interface (MPI) synonymously with CSP to describe code that has been decomposed into a set of MPI ranks, each of which executes sequential code segments that coordinate via messages. CSP is an imperative style of programming, in which application developers write statements that explicitly change the state of a program to produce a specific result (prescribing the order and manner in which operations are performed).[1]

As we look ahead to future platforms, imperative programming requires the management of system heterogeneity, fault tolerance, and increasingly complex workflows at the *application-level*. Alternatively, declarative programming – in which the application developer specifies their desired result but does not prescribe

[1] We note that in imperative styles of programming, the programming model and execution model are closely tied and often not distinguished. The non-specific term *parallel model* can be applied in these settings.

the manner and order in which work is performed – designates the management of some of this system complexity to an underlying runtime system. As seen in the Predictive Science Academic Alliance Program II (PSAAP-II) projects [5], and in the broader computational sciences community, the asynchronous many-task (AMT) model and associated runtime systems [6–14] are emerging as a declarative alternative to the imperative CSP approach. An AMT model is a categorization of programming and execution models that decompose applications into small, transferable units of work (many tasks) with associated inputs (dependencies or data blocks) rather than simply decomposing at the process level (MPI ranks). An asynchronous many-task (AMT) execution model can be viewed as the coarse-grained, distributed memory analog of instruction-level parallelism, extending the concepts of data prefetching, out-of-order task execution based on dependency analysis, and even branch prediction (speculative execution). Rather than executing in a well-defined order, tasks can execute when inputs become available. AMT runtimes often operate with a directed acyclic graph (DAG) that captures relationships between application data and interdependent tasks. A DAG that represents an application can be annotated to capture additional information (such as a tasks' read or write usage of data, and whether a task needs only a subset of a data structure).

A declarative AMT approach provides several key advantages over a traditional imperative program. By breaking up work into many work units that can be executed asynchronously, a runtime has the flexibility to perform load balancing for imbalanced problems. A DAG facilitates the execution of task parallelism, accelerating applications by allowing multiple tasks to execute simultaneously. Task parallelism is particularly beneficial for applications that have a wide task graph (multiple tasks that can be performed in parallel). However, the "lookahead" provided by a task graph is another advantage that applies universally to applications. Specifically, a data-task graph grants a runtime with the ability to see many pending operations simultaneously (in contrast to imperative programs that see only the current function.)[2] Lookahead enables the runtime to anticipate the optimal task schedule (where and when tasks and data movement execute) even on complex heterogeneous platforms.

We conclude this Section by noting that the interchangeable use of the terms MPI and CSP has led to a common misperception that AMT research seeks to identify "alternatives to MPI" for future software stacks. The AMT goal is to provide a more productive programming model that focuses on describing data effects rather than imperatively specifying execution. MPI is a transport layer and AMT runtimes can and do use MPI as a communication layer within their software stack [8, 9]. It is also true that AMT runtimes often leverage lower-level network APIs (e.g., GNI [15], PAMI [16], libfabrics [17]). By coding directly to hardware primitives, particularly RDMA, data movement can be optimized to avoid certain semantic and protocol overheads inherent to MPI point-to-point sends. While low-level transport layers may provide improved performance for AMT models, their use also furthers the misperception that AMT research seeks "alternatives to MPI". AMT research seeks to find a set of declarative, data-centric abstractions *above* the transport layer to facilitate the expression and management of asynchrony and task parallelism.

1.3 FY15 ATDM Level 2 Milestone

In FY15, Sandia's ATDM program delivered a Level 2 milestone (#5325) [18] that examined leading AMT research solutions in the context of ASC workloads. The primary goal of the milestone was to gather data to inform Sandia's ASC strategy in the context of next generation programming and execution models. After performing a broad survey of many AMT runtime systems, the team narrowed their focus to deliver

[2] With a static DAG that does not change during run-time, all pending operations are known. When a DAG is dynamically generated during run-time, lookahead is still achievable and typically is maintained for a limited window of operations.

a thorough examination of three exemplar runtimes, chosen to cover a spectrum of low-level flexibility to domain-specific expression: Charm++ [8], Legion [7], and Uintah [9]. Charm++ implements an actor model with low-level flexibility, replacing message passing with remote procedure invocations. Legion is a data-centric task model with higher-level constructs, representing a strong shift from the procedural style of MPI and Charm++ to a highly declarative program expression. Uintah is a scientific domain-specific system for solving partial differential equations on structured grids using thousands of processors. While not a true domain specific language (DSL), it was chosen to highlight the potential optimization of a domain-specific runtime. In addition to providing three very different implementations, application programmer interface (API)s, and abstractions, the runtimes were selected because they had demonstrated results on science applications at-scale.

MiniAero [19, 20], available online at [21] served as the application code basis for the milestone study. It is a compressible Navier-Stokes, three-dimensional, unstructured mesh, finite volume, explicit, computational fluid dynamics (CFD) mini application. The functionality of MiniAero was implemented using each of the Charm++, Legion, and Uintah runtimes. Using these implementations, the three runtimes were evaluated with respect to three main criteria:

Programmability: Does this runtime enable the efficient expression of ASC/ATDM workloads?
Performance: How performant is this runtime for ASC/ATDM workloads on current platforms and how well suited is this runtime to address future architecture challenges?
Mutability: What is the ease of adopting this runtime and modifying it to suit ASC/ATDM needs?

Several overarching findings emerged via the experiments and analysis summarized in the final report [18]. From a performance perspective, AMT runtimes show tremendous potential for addressing extreme-scale computing challenges. Empirical studies showed that an AMT runtime system can mitigate performance heterogeneity inherent to the machine itself[3] and that the imperative use of MPI and AMT runtimes perform comparably under balanced conditions. From a programmability and mutability perspective however, the report highlighted requirements gaps for Sandia applications and deficiencies in existing APIs. The report concluded by highlighting the critical need for best practices and eventual standards that preclude widespread adoption of these runtimes by the ASC application community. The FY15 report recommendations led to the development of DARMA, a C++ abstraction layer for AMT runtimes. DARMA provides a set of abstractions to facilitate the expression of tasking that map to a variety of underlying AMT runtime system technologies. This FY17 milestone report presents the DARMA abstraction layer and provides a detailed analysis of a DARMA-compliant software stack, capturing data to to further inform the technology roadmap that shapes Sandia's ASC code strategy.

1.4 FY17 Milestone Outline

This milestone is a direct followup to the FY15 ATDM L2 milestone #5325. Herein we evaluate a DARMA-compliant AMT runtime software stack comprising ATDM ASD software components and existing AMT runtime technology. We assess the performance and productivity of this software stack on benchmarks and proxy applications representative of the Sandia ATDM applications. As part of the effort to assess the perceived strengths and weaknesses of AMT models compared to more traditional approaches, experiments are performed on ATS-1 (Advanced Technology Systems) test bed machines and Trinity. The outcomes of

[3] Although the experiments in the report were with static workloads, there are other studies that show AMT runtimes can mitigate performance heterogeneity inherent in the application [22–24].

this effort include:

1. an initial DARMA-compliant AMT software stack,
2. a clear understanding of the strength and limitations of the co-designed AMT API and the associated runtime implementation in the context of Sandia's ATDM codes, and
3. information to guide our future research and development in this area (e.g., system performance issues, portability issues, usability issues on realistic apps, etc.).

The milestone comprises four specific deliverables:

Deliverable 1: Initial implementation of an integrated DARMA-compliant AMT software stack (including ATDM ASD components) on next generation platform testbeds and Trinity. We have three different AMT software stacks in various stages of development. The first, a DARMA-CHARM++ software stack that is fully distributed, is the focus of this milestone's performance analysis study. A DARMA-ONNODE backend serves as a development tool for both application and runtime system developers. We also have prototype DARMA-HPX implementations for both HPX3 and HPX5 (these fall under exceeds criteria for the milestone). The DARMA-ONNODE backend supports interoperability with KOKKOS using the OpenMP affinity layer for resource management and there is ongoing research and development joint with the KOKKOS and Resource manager ASD teams to support KOKKOS interoperability in the DARMA-CHARM++ backend. All performance analyses using the DARMA-CHARM++ backend was performed on Trinity (ATS-1) and Mutrino (a Trinity testbed).

Deliverable 2: Implementation of ATDM application benchmarks and proxies developed for the AMT system. Three benchmarks, written by DARMA developers, were developed to highlight benefits and limitations of the programming model and runtime system.

Jacobi: has memory-bound computation, latency-bound communication to expose overheads.
Molecular dynamics: is compute-bound with more bandwidth-intensive communication to complement Jacobi.
Simulated Imbalance: designed to assess load balancing capabilities.

There are three proxy applications, written by application developers. These proxy applications served to aide in the co-development of APIs, acquisition of subjective feedback, and application requirements for several use cases important to the ASC/ATDM program:

PIC**:** This was a direct collaboration with EMPIRE application team on two proxy applications: SIMPLEPIC, MINIPIC (exceeds criteria).
UQ: Embedded analysis is a capability used by both ATDM applications.
Multiscale: Ties back to Sandia's ASC/IC program (Sierra).

Deliverable 3: An analysis of the productivity, performance, scalability, and dynamic load balancing capability for the DARMA-compliant runtime on those ATDM application benchmarks and proxies. Over the course of the milestone, many 1000s of runs were performed on ATS-1 systems (Trinity and one of its testbeds, Mutrino) on both Haswell and KNL partitions. On Mutrino, scaling and profiling studies up to 64 nodes (2K cores Haswell), (4K cores KNL) were performed. On Trinity KNL, scaling studies up to 2K nodes (131K cores) were performed. While limited system access precluded full scaling studies to 4K

nodes (262K cores), spot runs were run to completion at 4K nodes. We had limited access to the Trinity Haswell partition during the course of our study, and thus present limited scaling results for this partition. Studies were performed for two compilers: GCC6.3.0 and ICC18.0.0beta (ICC results NDA for now and thus not included within the report). Our resulting scaling studies and performance profiling are used to assess: 1) overheads in balanced use cases with imperative baselines (MPI-only) 2) scaling trends (focus on strong, weak scaling as time permitted), and 3) load-balancing capabilities for load-imbalanced use cases. Lastly, we gathered subjective feedback from application developers on productivity and provide a summary of semantic information gain in DARMA program specification.

Deliverable 4: A report to inform the code development road map guiding the (Sandia) ASC code strategy for next generation platforms, in the context of using alternative programming models. This report meets the fourth deliverable criteria directly and is outlined as follows.

We begin the report with an overview of DARMA in Chapter 2, followed by a series of Chapters comprising findings and lessons learned over the course of this milestone. Where appropriate, details are highlighted as they pertain to ATDM applications, embedded analysis, and other performance abstraction capabilities within the ATDM subprogram. In Chapter 3 we examine the ability of DARMA to map to a variety of AMT runtime technologies. We provide a summary of existing DARMA-compliant backends. We further explore how DARMA could map onto other runtime system technologies. We conclude Chapter 3 with summaries of lessons learned regarding the generality of the backend API and potential changes that could occur based on our experiences this year. In Chapter 4 we present findings on interoperability of a DARMA-compliant software stack with node- and network-level libraries and runtime components. We focus a majority of the discussion on interoperability with KOKKOS [25], a node-level performance portability layer that is part of Sandia's ATDM program.

Chapter 5 provides a performance and productivity assessment of the overall DARMA approach. We begin Chapter 5 by focusing broadly on productivity benefits of DARMA's programming model. Next, we move to a discussion of system details informing our performance results, including the architectures and compilers used within our performance studies. This leads to a presentation of benchmark- and application- specific performance and productivity findings. The benchmarks presented herein were written by DARMA runtime developers, and they abstract away much of the complexity of a real application, leaving only a small portion designed to highlight specific benefits and limitations of the programming model and/or runtime system. Specifically, they are used to explore:

- At what granularity of work do runtime overheads become apparent for latency-intolerant algorithms?
- Are these overheads masked effectively as you expose greater asynchrony in the underlying algorithm (enabling greater overlap of communication and computation)?
- How effective is the runtime at mitigating load-imbalance?

Data gathered includes quantitative performance data in the form of weak and strong scaling studies and graphical views of performance traces that capture, e.g., overlap of communication and computation. For each benchmark considered, there are both Haswell and KNL results. We note that within our experiments, we have fixed the high-level algorithm for solving a given problem. However, the semantic information captured differs between MPI and DARMA, and we discuss the relevant differences for each benchmark in its corresponding Section.

In addition to the benchmarks, three proxy applications, written by application developers, are considered in various levels of detail: Particle In Cell (PIC), Uncertainty Quantification (UQ), and Multiscale. Each of

these proxies is interested in some subset of the following capabilities provided by a DARMA-CHARM++ backend:

- Runtime-managed load balancing
- Effective overlap of communication and computation
- Effective re-use/sharing of data across simulation runs

Similar to the benchmarks, we gather performance data in the form of scaling plots and views of performance traces. From a productivity perspective, we present a semantic assessment (as is done for the benchmarks). We also summarize the co-design process with each application and include subjective feedback from each application developer. We end the report with conclusions and recommendations in Chapter 6.

This page intentionally left blank.

Chapter 2

DARMA Overview

DARMA is a C++ abstraction layer for asynchronous many-task (AMT) runtimes. It provides a set of abstractions to facilitate the expression of tasking that map to a variety of underlying AMT runtime system technologies. Figure 2.1 presents an overview of a generic DARMA-compliant software stack. DARMA comprises three components: an application-facing frontend API, a runtime-facing backend API, and a translation layer that converts application code into a series of backend API calls to an AMT runtime. Because all runtime systems do not provide the same functionality, there will be some amount of "glue code" required to map backend API calls to a particular runtime.

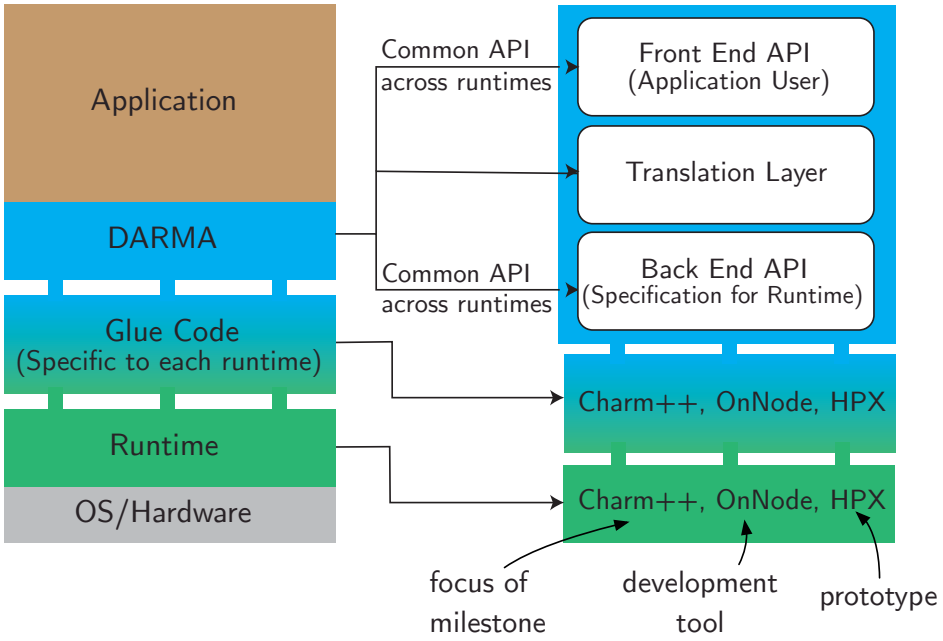


Figure 2.1: The components of DARMA within the context of a generic DARMA-compliant software stack.

2.1 Design Considerations

Over the course of the last two years, the DARMA team has worked closely with application and system software developers who directly inform DARMA's frontend API and backend specification via co-design interactions and feedback. This Section summarizes high-level considerations that have been raised during these interactions that directly inform DARMA's design.

AMT runtime systems operate with a directed acyclic graph (DAG) that captures relationships between application data and inter-dependent tasks (see Figure 2.2 as an example). DAGs can be annotated to capture additional information such as whether data is accessed in read/write mode or whether a task needs a subset of a particular piece of data only. As was mentioned in Section 1.2, a data-task dependency graph grants a runtime system "lookahead" when executing an application. If the application is describable in terms of a static task graph, many optimizations can be made even before the application executes. However, most nontrivial applications require dynamic task graphs, generated concurrent to the program's execution. In this case, the runtime system only has a portion of the whole application's graph at any given time; the portion of the task graph visible to a runtime but not yet executed describes its "lookahead" on an application. In addition to increased lookahead, any additional annotations enable a runtime system to reason more completely about when and where to execute a task and whether to load balance. We note that existing runtime systems leverage DAGs with varying degrees of annotation.

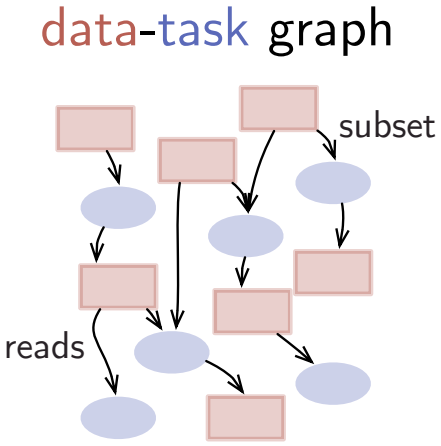


Figure 2.2: AMT runtimes operate with a directed acyclic graph (DAG) that captures relationships between application data and inter-dependent tasks. DAGs can be annotated to capture additional information such as a tasks' read/write usage of data, or whether a task needs a subset of a particular piece of data only. This additional information enables a runtime to reason more completely about when and where to execute a task and whether to load balance.

A data-task dependency graph is a declarative specification of a program that is a departure from the imperative program specification in many applications today. One of DARMA's primary objectives is to simplify the shift from imperative to declarative programming styles. Using C++ template metaprogramming, DARMA captures data-task dependencies from user code. Developers write code that maintains the sequential, imperative "look and feel" of C++. However, by using C++ wrapper classes provided in DARMA's header-only library, DARMA is able to automatically build the task graph without burdening developers with explicit programming in tasks and dependencies. The information contained in the task graph is what underlying AMT technologies leveraged by DARMA use to schedule work and manage data movement. The transformation of imperative code to a declarative data-task DAG occurs primarily at run-time (i.e., DARMA produces a dynamic task-data DAG), although there are compile-time aspects (Section 2.3).

2.1.1 Categories of Application Task Graphs

Task graphs can be broadly categorized as static or dynamic, with an intermediate "semi-static" sometimes also used. Static graphs are known completely at compile-time. For example, certain matrix assembly task

graphs might be characteristic of the underlying equations only and independent of the input problem or discretization. Dynamic task graphs, however, depend on the input parameters and simulation data. The exact task graph can only be discovered at runtime. Semi-static task graphs are those with regular characteristics such as the Jacobi solver benchmark considered in Section 5.4.1. Although the exact discretization, problem size, and number of iterations is not known until runtime, execution is highly regular and almost entirely known at compile time. Semi-static problems are characteristic of many MPI applications, particularly those encountered in non-adaptive engineering codes. Problems are relatively easy to decompose across MPI ranks and remain load-balanced throughout the entire execution.

DARMA is geared in particular towards dynamic problems with changing execution loads that are not easily predicted *a priori*, which can frequently benefit from overdecomposition and load balancing. Load balancing of dynamic applications will be be explored more deeply when discussing the SIMPLEPIC proxy application (Section 5.5) and synthetic imbalance benchmark (Section 5.4.3).

2.2 DARMA Execution Model

The DARMA execution model is not a strict requirement for how runtimes execute — merely a model for guiding how we understand the relationship between the frontend and backend (here the term frontend refers to DARMA in Figure 2.1 while backend refers to the Glue Code + Runtime layers in Figure 2.1).

Specifically, DARMA is a generalization of the producer-consumer work problem [26]. The frontend is a producer of tasks and the backend is a form of consumer. The producer puts new tasks into a work queue, which the consumer pulls off and executes. The producer-consumer model assumes simultaneous execution of producer and consumer with atomic access to shared state (the queue). In the case of DARMA, instead of a simple queue, the shared state is the task-DAG. This generalization of a producer-consumer model enables *deferred execution*. The application and frontend (i.e., the producer of new tasks) does not immediately execute work as would be done in an imperative model. Instead, the application *defers* execution by appending new tasks (vertices and edges) into the task-DAG. In parallel, the runtime is analyzing and running (i.e. consuming) the existing vertices and edges in the task-DAG. This parallel task creation and task execution linked through deferred execution of a task-DAG is the basis of the DARMA execution model. Figure 2.3 depicts simultaneous producers and consumers running active tasks, scheduling pending tasks and appending new tasks to execute.

2.3 DARMA Frontend Concepts and Components

The DARMA programming model shifts focus from execution to data. The effects a task will have on data are presented to the runtime system, encoded in the data-task DAG. Data effects are usually read or write permissions, but can also include commutative modes often found in reductions or streaming applications. These data effects enable the runtime system to reason about what can be run in parallel (e.g., multiple tasks can simultaneously read data, but only one task can safely modify data a time). In this way tasks create a "data flow" as data blocks are updated with new values and used in future tasks.

DARMA (jointly with the application) is the frontend producer of tasks, appending them as vertices and edges to the data-task graph. DARMA provides three services to simplify the process of producing tasks from user code:

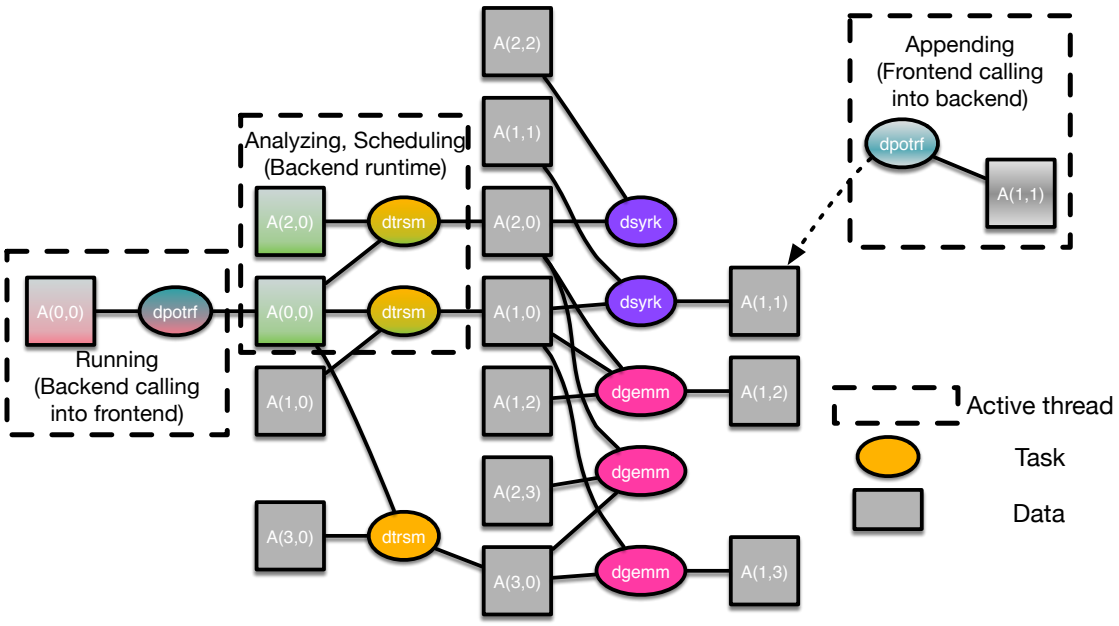


Figure 2.3: The DARMA execution model showing simultaneous consuming and producing of a task-DAG for Cholesky decomposition. Three parallel threads are shown that are 1) running an active task, 2) scheduling pending tasks, and 3) appending new tasks to execute. Note this is a conceptual model only and not a strict requirement.

- Compile-time *capture hooks*
- Run-time *dependency capture*
- Run-time *dependency analysis*

In DARMA, the template metaprogramming in the translation layer causes the C++ compiler to insert capture hooks when creating tasks. The capture hooks are used at run-time to capture each variable's use in a function at run-time, along with other pertinent information (like whether the variable is being updated or just read). Actually deriving what tasks can run in parallel and their dependency relationships occurs through a run-time dependency analysis phase. After dependency analysis, the sequential C++ code is finally converted into a task graph representation and passed to the backend for scheduling and execution.

Note, the functionality supported in DARMA by these three components is an extension of what occurs with a standard C function call. In C, the compiler inserts stack allocation calls and the instructions for moving variables onto the stack (these are analogous to DARMA's capture hooks). At run-time, the function arguments are then pushed onto the stack and made available to the called function (e.g., run-time dependency capture). All of this is transparent to the C programmer. Managing the stack across function calls is the job of the compiler and the assembler. We note that there is no run-time dependency analysis in this example, because in standard C, there is no "meta" layer that understands the arguments passed into a function. Furthermore, there is no connection made between uses of a variable across multiple functions.

DARMA's current implementation combines these three components in a single "translation layer" between a frontend API and backend API, implemented as a header-only C++ library. However, through our initial implementation effort, we see a natural decomposition into the three distinct components above and future development of DARMA will explore splitting this functionality more cleanly into distinct software components. In the remainder of this section we elaborate on the three components, discussing alternative implementation strategies as potential future work. More rigorous concepts for fully understanding the

translation layer is contained in the Appendix.

2.3.1 Capture Hooks

The process of inserting capture hooks involves compile-time information only. This process is currently managed through template meta-programming operations on special template wrappers that the application developer uses within their code (more details in Section 2.4). This directly embeds the capture hooks in a C++ application using only the meta-programming tools in a standard C++ compiler. Because all the information required for inserting capture hooks is available at compile-time, an alternative implementation could insert these capture hooks using a source-to-source transformation.

2.3.2 Dependency Capture

Dependency capture occurs at run-time. This process associates groups of variable uses with a function that uses them. DARMA has a strict requirement that run-time capture records task ordering constraints such that the program produces results equivalent to a sequential execution of the source code. The current DARMA dependency capture records 1) whether variables are used in a read/write mode and 2) ordering relationships of these reads and writes. The principle requirement on dependency capture is only that it contains enough information to preserve sequential execution. Thus, a dependency capture layer may capture more or less detailed information about variable accesses, depending on the complexity of the analysis supported by the underlying dependency analysis component.

2.3.3 Dependency Analysis

Dependency analysis occurs at run-time. It is the process of deriving which tasks can run in parallel and the precedence constraints between tasks that cannot run in parallel. The results of the dependency analysis are delivered to the runtime system via backend API calls. The dependency analysis layer must receive ordering constraints and access permissions (read/write) on the variables used in tasks. The analysis layer may also receive information about aliasing or subsetting uses of variables.

A very simple dependency analysis layer could ignore all dependency capture information except ordering requirements, executing all tasks in source-code order. On the other extreme, a full dependency analysis layer could support full read/write task parallelism with data aliasing analysis. If a dependency capture component provides more information than the analysis layer can use, the analysis layer is free to ignore information not required to preserve sequential semantics. Similarly, if the capture layer provides less information than could be known (e.g., no distinguishing read/write usages) the analysis layer should simply operate with the simplified information.

2.4 DARMA Abstractions and Simple Examples

The application developer uses DARMA frontend API to annotate their code using abstractions for data and tasks. Their code maintains the sequential, imperative "look and feel" of C++, however the abstractions are

used by the translation layer to insert capture hooks, capture dependencies, and perform dependency analysis required to build the data-task DAG. The information contained in the task graph is what underlying AMT technologies leveraged by DARMA use to schedule work and manage data movement.

In DARMA, asynchronous smart pointers wrap user data and track meta-data used to build and annotate the DAG:

- `darma::AccessHandle<T>`
- `darma::AccessHandleCollection<T>`

These smart pointers enforce sequential semantics: they use the order in which data is accessed in your program and how it is accessed (read/write/etc.) to automatically extract concurrency.

Tasks are annotated via several interfaces:

- `darma::create_work`
- `darma::create_concurrent_work`

Figure 2.4 (a) and (b) illustrates two simple DARMA applications and the associated data-task dependency graphs that are generated.

In Figure 2.5 we show an example DARMA code that performs a distributed SAXPY operation over three collections that contain `A`, `x`, `y`. The functor `SAXPYDistVec` is executed over the index range `Range1D<int>(num_blocks)` that describes the extents of the three collections along with the task collection spawned on line 49.

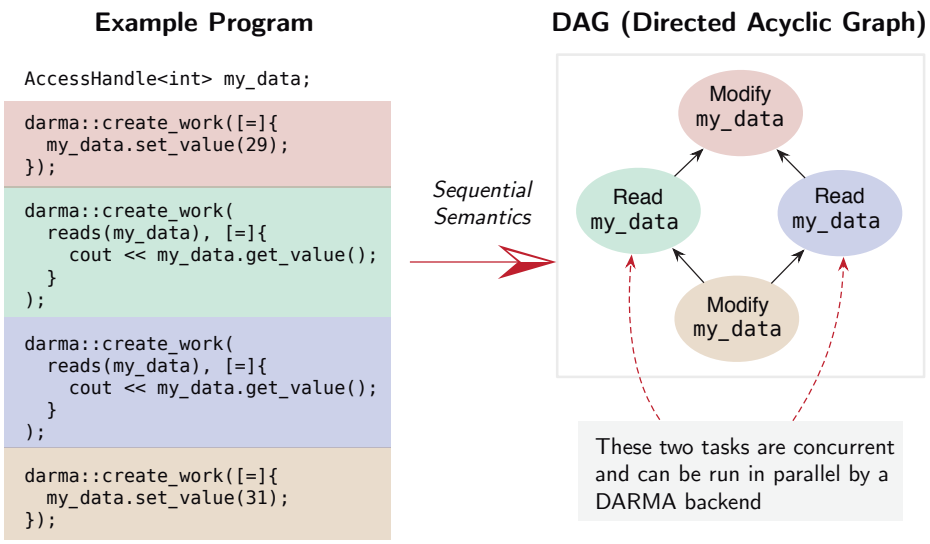
2.5 Backend Concepts and Components

The runtime-facing portion of DARMA (which we call the *backend API* or *backend abstractions* in this document) is designed primarily to present the application to the runtime in a way that allows the runtime to reason about the application work in terms of tasks and dependencies. Application code is "translated" into calls to the backend API that express the application workload in a form that can easily be executed by a DARMA backend implementation. The translations of application code to these backend abstractions happens in a layer that takes advantage of, among other things, common C++ template metaprogramming idioms to perform this translation with very low overhead; these metaprogramming techniques and their application to the DARMA API are beyond the scope of this report.

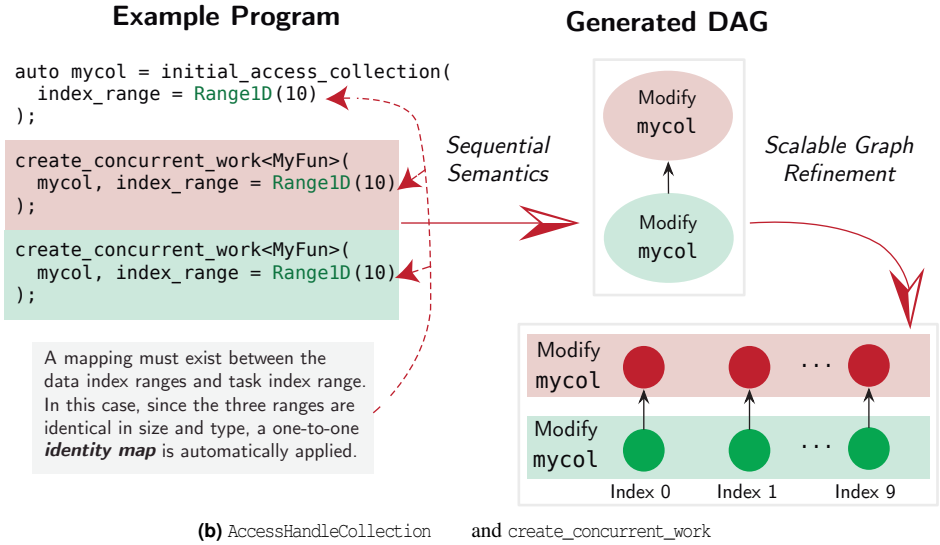
The DARMA translation layer expresses application code to the backend in terms of a task-data directed acyclic graph (DAG). `Flow` s and `AntiFlow` s are backend-provided objects (opaque typedefs from DARMA's perspective) that the translation layer uses to express data-flow dependencies and anti-dependencies[1], respectively, between uses of data in the application. A `Use` object (frontend-provided[2]) summarizes these relationships by grouping up to two `Flow` s (an `in` flow and an `out` flow) and up to two `AntiFlow` s (an `anti-in` and an `anti-out`) with permissions that describe the relationship between various accesses to a piece of underlying data. A `Task` is then just a collection of `Use` s and a method (`Task::run()`)

[1] An anti-dependency is also known as a Write after read (Write-After-Read (WAR)) data hazard where a task has required input(s) that are later changed. An anti-dependency can be removed by making a copy of the data

[2] The terms "frontend-defined" or "frontend-provided" refer to objects or abstractions that the backend runtime implementer does not provide an implementation of; rather, it is implemented in the translation layer (sometimes called the frontend in this section; the terms may be used interchangeably) to express some set of information to the backend.



(a) AccessHandle and create_work



(b) AccessHandleCollection and create_concurrent_work

Figure 2.4: Simple examples putting data and tasks together.

that accesses the data and performs computation with it.

All DARMA programs start with a single top-level Task , which the backend should run in one place. When the top-level Task and all Task s generated by the top-level task (and all Task s generated by those Task s, and so on) finish executing, the program is complete. The top-level task may generate simple Task s or it may generate TaskCollection s — large data-parallel launches of many tasks running concurrently. Task collections in DARMA, in contrast to other systems like TASCEL with task collections, do not require that tasks are independent. Concurrent tasks (i.e., members of the same TaskCollection) may send and receive data to each other through publish operations. Backends must therefore implement concurrency or non-blocking publish operations to ensure that tasks within a collection are guaranteed forward progress. However, due mostly to the constraints in the DARMA programming model, the concurrent progress requirements for members of TaskCollection s are significantly less strict and constraining than many other systems (for instance, std::thread in the C++ standard library). In addition to a correctness requirement

```
constexpr double INIT_VAL_A = 1.0, INIT_VAL_X = 2.0, INIT_VAL_Y = 3.0

// Typedefs to save some space
using index_t = Index1D< int>;
using block_t = std::vector<double>;
using AHC_block_t = AccessHandleCollection<block_t, Range1D< int>>;

struct InitDistVec {
  void operator()(index_t index, AHC_block_t dist_vec, int blk_sz, double val) {
    block_t& local_block = dist_vec[index].local_access().get_reference();
    local_block.resize(blk_sz);
    for (auto i = 0; i < blk_sz; i++) {
      local_block[i] = val;
    }
  }
};

struct SAXPYBlock {
  void operator()(block_t const& A, block_t const& x, block_t y, int blk_sz) {
    for (int i = 0; i < blk_sz; i++) {
      y[i] = A[i] * x[i] + y[i];
    }
  }
};

struct SAXPYDistVec {
  void operator()(index_t index, AHC_block_t A_col, AHC_block_t x_col,
      AHC_block_t y_col, int blk_sz) {
    // perform the SAXPY calculation on the local block of A, x, y
    create_work<SAXPYBlock>(
      A_col[index].local_access(), x_col[index].local_access(),
      y_col[index].local_access(), blk_sz);
  }
};

void darma_main_task( std::vector<std::string> args) {
  int const num_blocks = std::atoi(args[1].c_str());
  int const blk_sz = std::atoi(args[2].c_str());
  // The index range that describes the extents of the AccessHandleCollection
  auto range = Range1D< int>(num_blocks);
  // create AccessHandleCollections for distributed vectors A, x, y
  auto A = initial_access_collection<block_t>(index_range=range);
  auto x = initial_access_collection<block_t>(index_range=range);
  auto y = initial_access_collection<block_t>(index_range=range);
  // initialize distributed vectors with some initial values
  create_concurrent_work<InitDistVec>(A, blk_sz, INIT_VAL_A, index_range=range);
  create_concurrent_work<InitDistVec>(x, blk_sz, INIT_VAL_X, index_range=range);
  create_concurrent_work<InitDistVec>(y, blk_sz, INIT_VAL_Y, index_range=range);
  // perform the SAXPY calculation
  create_concurrent_work<SAXPYDistVec>(A, x, y, blk_sz, index_range=range);
}
```

Figure 2.5: Example DARMA code that performs a SAXPY on the distributed collections: `A`, `x`, `y`.

(concurrent forward progress), `TaskCollection` s in DARMA also imply a performance requirement of scheduling and launching collections scalably across a large system.

The operations a backend is required to implement can be understood by operations it must perform with `Use`, `Flow`, and `Task` objects (with special relationships for those used in `TaskCollection` s). The DARMA specification provides rules on `Flow`, `Use`, and `Task` that must be obeyed. The challenge for the backend implementation is to execute an application efficiently while obeying the requirements codified in `Flow`-`Use`-`Task` relationships (equivalently, while making a legal traversal of the application's task-data DAG). We can identify the following responsibility that *must* be implemented correctly according to the specification and *should* be implemented performantly.

- Distributed dependency management (mostly through scalable distributed reference counting) of `Flow` and `AntiFlow` objects to determine when a `Task` is safe to run.
- Migrating `Use` and `Task` objects (and their associated application data) between DARMA processes when launching a `TaskCollection`, publishing within a `TaskCollection`, load balancing, and other distributed operations on the coarsened `FlowCollection`-`UseCollection`-`TaskCollection` representation of the application.
- Guaranteeing forward progress between members of a `TaskCollection`
- Collectives

Each of these responsibilities is currently implemented within a single component for the DARMA-CHARM++ backend (see Section 3.1). Future work will consider how they could be split into independent components. A more precise specification of backend requirements is given in the Appendix.

2.5.1 Future Work: Shifting Frontend and Backend Responsibilities

The C++ types underlying `AccessHandle` objects are currently opaque (type-erased) to the backend. It instead receives an abstract C++ class, `Handle`, that encapsulate type information (along with a few helper classes like `SerializationManager` and `ArrayConceptManager`). Runtime type registries for serialization and the task queue are currently maintained entirely in the frontend. Componentization of the frontend may allow backends to receive type-specific information in the future, allowing, e.g., serialization optimizations or type-optimized collectives.

This page intentionally left blank.

Chapter 3

Findings on the Generality of the DARMA Backend API

By design, DARMA's backend API captures a declarative representation of an application that does not prescribe exact execution. This enables a variety of runtime implementation, scheduling, and optimization strategies using existing software stack technologies. While not prescriptive about execution, a DARMA-compliant runtime stack must satisfy the following core execution model requirements:

- The runtime must manage data dependencies between tasks (data inputs and outputs).
  - Data usage (write/read/advanced modes) and sequencing information is provided by the frontend to facilitate task scheduling without data conflicts.
  - The runtime should schedule decisions based on current state to copy, move, or stall data accesses to optimize performance and memory usage.
- The runtime must track placement of data, tasks, and task collections across distinct memory spaces.
  - This includes support for reference counting of data to determine task readiness for scheduling purposes.
  - The location of task collection elements must be managed to efficiently transfer data for publishes (sends) and fetches (receives) between elements.
- The runtime must coordinate data movement utilizing its underlying communication transport layer.
  - The frontend interface provides tools to serialize/de-serialize arbitrarily typed objects when moving C++ objects across memory spaces.
- The runtime must implement collective operations (currently only `reduce` and `allreduce`).

Figure 2.1 captures an overview of a DARMA-compliant software stack. The amount of direct support for the requirements above plays a crucial role in determining the thickness of the "glue layer" in the figure. The DARMA frontend supplies the majority of data dependency and usage information directly to the backend for the first requirement. However, a backend can make decisions regarding task dependency resolution (e.g., copying data) based on current distributed state (or past persistent information collection) as it strives to maximize performance. The second requirement engenders the most complexity: how is the placement of data, tasks, and task collections managed? In shared-memory (e.g., flat memory hierarchy) contexts this is trivial, but on large-scale distributed-memory architectures, location management is the dominant problem the backend must tackle to efficiently execute tasks, distribute task collections, and send and receive data between (potentially) dynamically placed entities within the system. The third requirement - coordinating data movement, data marshaling, and task migration itself - is perhaps the most straightforward concern, as backend implementations often have clear mechanisms for moving bytes across memory spaces efficiently. Finally, collective operations are where many of the performance portability challenges may be encountered. Because DARMA has very specific semantics surrounding collective operations, a semantic mismatch or lack of collective implementation may require significant effort to ensure high performance. For simulations

involving tightly-coupled physics, collectives can have an enormous impact on performance despite only occupying a small fraction of the application source code. A slightly inefficient implementation may have significant performance ramifications.

Note that load balancing and work stealing are not strictly part of the backend interface. A backend runtime has complete flexibility in how and when load balancing occurs, provided only that it obeys DARMA semantics (enforcing dependencies). The transformation from a complex algorithm in the frontend that must satisfy sequential semantics to a simple set of dependency reference counting rules is quite a dramatic transformation. It provides a representation of the algorithm amenable to task reordering and transformation without being overly prescriptive on how a backend actually implements it.

There are multiple DARMA-compliant software stacks in various stages of development. In the Sections that follow, we explore the compatibility and implementation strategy for these backends. We first present an overview of the underlying runtime technology followed by a discussion of strategies and challenges in regards to 1) distributed reference counting, 2) collectives, and 3) load balancing capabilities. We include the most detail on the DARMA-CHARM++ backend, which is the focus of performance analysis within this milestone report. We also summarize implementation strategies for DARMA-ONNODE and DARMA-HPX and for other potential backends, including MPI and Legion/REALM, highlighting compatibility issues where relevant. We conclude with Table 3.1 which provides an overview DARMA implementation strategies for multiple runtime technologies, capturing how direct the mapping is between DARMA's requirements and the runtime system's underlying functionality.

3.1 CHARM++

3.1.1 Introduction

A key objective in building a CHARM++ backend for DARMA is to enable applications to utilize the flagship features of CHARM++: a highly-scalable implementation, asynchrony, load balancing, checkpoint/restart, and modularity. CHARM++ is a very specific breed of programming model: it is best characterized as a parallel object-oriented model, sometimes known in theoretical worlds as an *actor model*. While in some ways *actor-based* may be used to describe it, it remains clearly out of the mainstream of actor models due to the low emphasis placed on the theoretical guarantees that actor systems typically prescribe. Additionally, the focus of actor systems has not been high-performance computing; in fact, typically the opposite is true, that actor models are used for security, engineering, web programming: domains in which computational model rigor is paramount.

CHARM++ was developed semi-concurrently with the development of actor models originally studied by Prof. Carl Hewitt (MIT) and followed by Prof. Gul Abga (UIUC) in the mid-1980s, culminating in his dissertation, *Actors: A Model of Concurrent Computation in Distributed Systems*. While the focal point of actor systems was typically more theoretical and rigorous, as a deep study of concurrent systems, CHARM++ evolved out of developing mechanisms to scalably solve more practical large-scale logic problems engendering the development of the Chare Kernel in the late 1980s by Prof. Laxmikant V. Kale. The Chare Kernel was a C-based programming model that enabled concurrent objects, called *chares*, to be created and messages to be sent to them causing asynchronous method invocation. By 1993, the Chare Kernel had evolved into a C++ programming tool, shipping with a custom translator to generate code to promote usability by reducing program verbosity. Also during this time, a new parallel interoperable runtime system called Con-

verse was developed, separating the message-driven runtime system CHARM++ from the actual language implementation. Over the next 10 years, CHARM++ would be rewritten several times; load balancing and migratability was actively researched and developed; and it evolved into a production-quality system as domain science collaborations were fostered. Along the way, CHARM++ remained, at heart, a concurrent, object-oriented system, but fancier idioms were researched and developed, such as *chare arrays*. Chare arrays are potentially sparse or dense, multi-dimensional, custom indexable, on-demand-created entities that make CHARM++ a powerful language and runtime system adept at scaling a host of computational science algorithms on the top supercomputers.

In CHARM++, a chare is a C++ object for which methods can be remotely invoked by any other chare via a *proxy*. A proxy provides a mechanism to virtually name an instance of a chare or chare array to enable *entry methods*, explicitly marked C++ methods, as remotely invocable. On the processor where the chare resides, messages that cause entry methods to invoke can arrive in any order and can thus execute in any order. Priorities can be supplied for method invocations, but they are only followed as a heuristic—CHARM++ is not obligated to strictly obey them semantically. The programmer is responsible for ensuring that any execution order is correct if there are potential races. A higher-level language, called Structured Dagger (abbreviated SDAG), can be used to define valid (partial) orders of message execution. In the sense that only one method can execute at a time, CHARM++ incorporates "sequential semantics". According to CHARM++ semantics, a chare has a well-defined location (processor) at any point in time, but it may be migrated during the course of the program in between method invocations. A virtual proxy is sufficiently powerful to send a message (i.e., invoke a method) on the chare without knowing the exact location of the chare. The location is resolved when a message is sent to the chare—the runtime is responsible for managing the location and ensuring that all messages arrive (even in the face of migrations). If a chare keeps migrating it is possible that a message takes many hops to eventually arrive on the processor where the chare eventually resides. Many different protocols can be employed for location management—an area that research has extensively studied in the past.

DARMA's sequential semantics are quite different than those of CHARM++. DARMA uses the type of access to the data (e.g. read/write/modify) and the sequence and nesting for which the tasks are defined to produce a task graph. Thus, the backend DARMA semantics are not a direct mapping to CHARM++'s object-oriented semantics. Furthermore, CHARM++ empowers objects to provide a locus of control and data persistency, which is represented differently in DARMA. The challenges mapping DARMA constructs to those provided by CHARM++ are explored in detail below.

3.1.2 Challenges

Due to the substantial semantic differences between CHARM++ and DARMA in how data and work (tasks or object methods) are defined, several challenges must be addressed to effectively map DARMA to CHARM++. First, DARMA separates data (`AccessHandle<T>`) and control flow (tasks via `create_work(...)` and task collections via `create_concurrent_work`) more clearly than CHARM++, and thus neither are directly semantically compatible with CHARM++'s chare arrays, which provide the foundation of persistence-based (or measurement-based) load balancing. Most scientific applications decompose the domain into several chare array types and create instances of those to compute. The iterative nature of many scientific problems is exploited by CHARM++ through heuristics that operate based on the assumption of persistency in the model: *each chare array element is likely to perform similar work in the future as it did in the past*. Thus, CHARM++ instruments each element's entry methods of the chare array with timers and other hooks to measure the time and communication patterns that arise during an iteration. Iterations are

marked by the user by explicit calls to `AtAsync()` , which denote a collective phase shift in an array's computation. Load balancing utilizes instrumentation collected from all the live chare arrays instances (along with other statically placed entities as static constraints) to compute a new element-to-processor distribution that minimizes imbalance and communication overhead given a network topology.

The backend DARMA-CHARM++ glue layer tackles these semantic differences by using chare and chare arrays in non-traditional ways to provide a mapping from DARMA to CHARM++ that intends to maximize the usability of the feature set in CHARM++ while striving for the best performance possible. The DARMA-CHARM++ layer is thus fairly thick—about 20k lines of CHARM++ code (mostly C++ code, with some Charm Interface files). The majority of the complexity lies in mapping data and task collections to chare arrays that retain persistency and implement publish/fetch semantics provided by DARMA, which are distinct from entry method invocations as a data transfer mechanism.

3.1.3 Data and Task Collections

A task collection in DARMA (`create_concurrent_work`) is mapped to an instance of a chare array `TaskCollection` built in the DARMA-CHARM++ backend. When a (`create_concurrent_work`) is encountered, the backend keeps a hash map of current live instances of the `TaskCollection` chare array. Each live instance is accessible based on the data inputs to the DARMA task collection. In particular, if multiple task collections in DARMA have the same data inputs (in write mode), the same chare array instance is used as was used in the past. By re-using a previous `TaskCollection` chare array instance, the CHARM++ runtime system is able to extract the persistency that may be in an iterative DARMA scientific application.

Data collections in DARMA are expressed in the front-end API by `AccessHandleCollection<Range<U>,T>` that map to a collection of `Flow` objects in the backend. The DARMA-CHARM++ backend manages `Flow` s with custom C++ data structures for containment, but does not have any direct mapping in CHARM++. Instead, data collection in DARMA may reside (for a given generation) in many chare array instance at a given time, depending on the task collections that are executing.

In the following code snippet of DARMA, a single collection of data called `data_collection` is created and then a task collection is instantiated that runs the functor of type `DoWork` in a for loop `num_timesteps` times:

```
1 void darma_main_task(  std::vector<std::string> args) {
2   Range1D< int> domain(num_elemnts);
3   AccessHandleCollection<   std::vector<double>, Range1D< int>> data_collection =
4     initial_access_collection<   std::vector<double>>(index_range=domain);
5   for (int i = 0; i < num_timesteps; i++) {
6     create_concurrent_work<DoWork>(i, data_collection, index_range=domain);
7   }
8 }
```

The first time that the `create_concurrent_work` is encountered, the DARMA-CHARM++ backend will instantiate a `TaskCollection` chare array that relies on input `data_collection` , a particular data collection with a unique name in the backend. This same chare array will be used to perform the computation defined in the functor `DoWork` each iteration because the input is the same. Thus, when load balancing is invoked, CHARM++ will be able to rely on past computational work of that chare array instance to predict the future.

Complexity arises when inputs are read-only instead of write. Re-using the chare array instance in this case might reduce concurrency. Furthermore, if the inputs are not exactly the same but there is some amount of overlap between them, the backend must make a more complex decision of whether to re-use the chare array instance.

3.1.3.1 Load Balancing

As explained in the above section, the DARMA-CHARM++ backend is careful to ensure that DARMA code with task collections use chare arrays such that persistence is exposed to the CHARM++ runtime. However, the DARMA-CHARM++ backend must also decide what constitutes a phase (or iteration) and when to run the load balancer. In CHARM++ programs, the user defines a phase (or iteration) by invoking `AtAsync()` on the chare array element. Every chare array element of a particular instance must make this invocation in a coordinated manner, but it does not necessarily imply or invoke synchronization on the program's execution. CHARM++ uses this to distinguish between phases so that instrumentation is properly associated with a distinct phase. When the load balancer is actually invoked, CHARM++ may synchronize the system to collect a clean snapshot for that phase, run the load balancing algorithm, and then begin migration to establish a new load distribution.

In theory, a DARMA backend may be able to automatically infer the phases (or iterations) of a program, but this remains an open research problem for arbitrarily complex codes, especially in the presence of hierarchical iteration space, such as layers of macro- and micro-iterations. Thus, we currently provide a hint to the backend of a phase by adding a label to the `create_concurrent_work` that indicates a new phase will begin:

```
5  for (int i = 0; i < num_timesteps; i++) {
6    create_concurrent_work<DoWork>(i, data_collection,          name=BeginIter, index_range=domain);
7  }
```

The keyword argument `name` allows the user to supply an arbitrary tuple of variably typed entities that is passed to the backend without any frontend semantic interpretation. Thus, it is used during DARMA development to extend the interface with backend-specific indicators without introducing new syntax and semantics to the frontend for a backend-specific hint. Note, that this code will compile and run with any backend, but it can be ignored by other backend implementations.

With this hint in place, the DARMA backend inserts dynamically an `AtSync()` call to CHARM++ for each chare array instance that was active during the current phase. With this in place, the user can invoke a CHARM++ load balancer for the DARMA program by simply adding it to the command line.

Lastly, because the DARMA-CHARM++ backend does more than just run the user's program when chare arrays execute, the CHARM++ instrumentation cannot be directly relied on to accurately reflect the time spent executing DARMA tasks that belong to the element of the task collection. To solve this, the DARMA-CHARM++ backend leverages a feature of CHARM++ that allows for CHARM++'s instrumentation to be overridden (or turned off). Instead, times or weights are provided explicitly to CHARM++ as part of an application model. This mode of CHARM++ is typically used in cases when an application has clairvoyant knowledge of its current or future behavior that can be directly input to the load balancer database to increase accuracy or decrease instrumentation overheads. The DARMA-CHARM++ backend instruments all DARMA subtasks that are children of a task collection indexed element by using wall timers while a DARMA task is running (because a DARMA task can cause calls back into the backend, the backend im-

plementation must be careful to only record the time in the user's code). Once the times are aggregated for all subtasks, this time is provided explicitly to CHARM++ as a *model-based LB* timing.

3.1.4 Termination Detection

One of the key problems in executing a DARMA program correctly is detecting when it is finished executing all tasks transitively. A DARMA program is a *rooted task* model: it begins execution with a *root task*, from which all other tasks and task collections are spawned in the system. The problem of termination detection in DARMA is detecting in a distributed system when the root task is finished: all its work and subtasks transitively have finished execution. Although, on the surface, this may appear to be a simple problem that a few counters would solve, it is complicated by the fact that, in a distributed system, messages in flight may create more work and may arrive in any order or be arbitrarily delayed on the wire. Thus, employing a reduction that checks for all work queues to be empty is insufficient. Even implementing a reduction that checks if all work produced and consumed is equivalent can be proven to be incorrect. Nevertheless, years of extensive research have produced a vast range of algorithms that are effective at solving this problem efficiently.

The DARMA backend uses an algorithm called the *four-counter method*, to effectively solve this problem. Each DARMA backend uses counters on every processor core to count tasks and task collections (represented as a count of the number of tasks in the collection) that are produced and consumed. For instance, if a processor encounters a `create_concurrent_work` , it increases its produced count by the number of tasks in the task collection. Note that each task collection task may generate more tasks. Then, the *four-counter method* is applied by utilizing a series of reductions that aggregate produced and consumed counts across the whole system. The algorithm requires that two sequential waves of reductions have exactly identical produced and consumed counts to deduce that termination has been detected. The DARMA-CHARM++ backend optimizes this algorithm by only propagating a reduction when termination is likely: i.e., when the reduction reaches a processor, it only contributes to the reduction if its work queue is empty and several other indicators in the backend are flagged.

3.1.5 Collectives

The DARMA frontend interface for collectives currently provides two mechanisms: an `allreduce` among elements of a task collection (not directly tied to data) and a `reduce` that operates directly on a `AccessHandleCollection` instance. DARMA allows the contribution to a given collective instance (e.g., a single `allreduce`) to be demarcated with a unique tag provided by the user. This tag coordinates the reduction so each contributed piece is combined correctly with other contributions in the task collection. The following code snippet in DARMA is an example of two distinct `allreduce` s in the same task collection with different tags. Note that although each element in this simple example calls `allreduce` in the same order, this is not a requirement due to the existence of tags for coordination.

```
1  struct MyFunctor {
2    void operator()(ConcurrentContext<IndexID<     int>> ctx)  const {
3      auto han1 = initial_access<   int>();
4      auto han2 = initial_access<   double>();
5
6      create_work([=]{
7        han1.set_value(29); han2.set_value(37.5);
8      });
```

```
9
10        ctx.allreduce<Add>(in_out=han1, tag=1);
11        ctx.allreduce<Max>(in_out=han2, tag=2);
12      }
13    };
```

In contrast, CHARM++ has more restricted reduction semantics: it requires that each contribution (for a given chare array instance) be ordered the same way across every element in that instance. This semantic difference makes directly mapping DARMA's `allreduce` and `reduce` to a chare array reduction in CHARM++ difficult.

To address this disparity, the DARMA-CHARM++ backend implements a custom CHARM++ reduction operation that uses the built-in CHARM++ spanning trees, but can combine interleaved reduction tags correctly by packing/combining contributions up the reduction tree in a smart, dynamically sized, tag-aware container. To ensure that deadlock does not occur, a ready DARMA contribution immediately contributes to a CHARM++ reduction on the related chare array that manages the encapsulated task collection. This contribution includes the reduction data, tag, and `reduce` operation and is immediately propagated up the tree regardless of the tag. At each intermediary combine node in the CHARM++ reduction tree, the set of tags for each child (of the reduction tree node) are sorted and the reduction operation is applied for identical tags. By allowing arbitrary tag interleaving on a single CHARM++ reduction, the final result at the reduction root may contain multiple partial, incomplete reductions. The DARMA-CHARM++ backend tracks and buffers the partial results, coordinating future reductions to eventually produce a final result for each tag. The custom collective implementation in the DARMA-CHARM++ backend is over 600 lines of code.

3.2 OnNode

In the course of designing DARMA, it has been helpful to create a "simplest possible" backend to the DARMA API. In addition to providing a bare-bones backend for applications to use for preliminary debugging, the DARMA-ONNODE backend provides a sandbox for the DARMA team to experiment with new features and a simplified implementation that is helpful in elucidating fundamental semantics of DARMA backend API constructs. The DARMA-ONNODE backend only runs in shared memory, which significantly simplifies the implementation process. In this context, much of the backend API can be expressed in terms of a relatively simple concurrent programming construct, which for the purposes of this section we will call a *join counter*. In order to represent a `Flow` correctly, join counters in the DARMA-ONNODE backend need to implement the following semantics:

- The value of the join counter is always non-negative and goes from non-zero to zero exactly once.
- No increments may occur after the counter goes from non-zero to zero. (Thus, the final decrement of the join counter does not race with any increments.)
- A list of callbacks (not necessarily ordered) can be attached to the join counter that are triggered when the counter's value reaches 0. Attachment of callbacks to this list may race with any increments or decrements to the join counter's value, including the final decrement.

With this construct, much of the shared memory implementation of the DARMA API is relatively straightforward. A `Flow` can be implemented as a join counter that is (essentially) incremented when a producer `Use`

is registered and decremented when that `Use` is released (a more precise description is given in Section 7.2.2). The readiness of a `Task` can then be determined from the readiness of the `Flow` s and `AntiFlow` s it depends on. Thus, a callback that enqueues a `Task` that is ready to run can be attached to a join counter. The value of the join counter should start at the number of in/anti-in `Flow` s for dependency and anti-dependency `Use` s of that `Task` . Decrements to the `Task` 's join counter are attached as callbacks to the join counters of those in/anti-in `Flow` s. Worker threads then simply pop tasks off of this ready queue and execute their `run()` methods until all of the `Task` s in the program have been completed.

When working in a shared memory environment, simple implementations of collectives and load-balancing are relatively trivial. The current DARMA-ONNODE backend implementation uses randomized work stealing to balance the workload across the worker threads (though the work-stealing details are well-separated enough from the rest of the implementation that other strategies could be implemented with little additional effort). Collectives, too, are relatively straightforward. Since the number of expected contributions must be given in the collective call, reductions can be simply implemented with a join counter that starts with the number of expected contributions and has an attached callback that performs the reduction and releases the collective's `Use` s.

3.3 HPX

HPX-5 is a low-level runtime system project out of Thomas Sterling's group at Indiana University. The runtime system is based on the ParalleX execution model, which relies heavily on futures (or *lightweight control objects* (LCOs), a generalization of a future) as a mechanism for programmatic dataflow. Distributed computing and data transfer is enabled by an *active global address space* (AGAS), which is similar to PGAS (partitioned) distributed models. In this computational model, each processor has access to a virtualized global address space, and uses put and get operations to transfer data at a low level. Futures are the mechanism used to describe parallel dataflow algorithms as the expectation of receiving a piece of data in the future. By fusing futures together with triggered actions (continuations), parallel dataflow algorithms can be effectively expressed. Lightweight threads, or user-level threads, are used on-node to schedule a program by suspending when data is not available (a future is not fulfilled) and context switching to a ready task.

Some of DARMA's data and computational model have fairly direct mappings to future-like semantics. An `AccessHandle<T>` in DARMA, can be viewed as a series of futures (essentially a dataflow) with a common name, which matches well with LCOs. Since LCOs in HPX-5 are associated with a unique address in the global address space, their distributed reference counting semantics need to implement `Flow` s and `AntiFlow` s for the DARMA backend API have a relatively clean mapping onto HPX-5. However, other parts of the DARMA model to support efficient distributed computation, such as task collections and collectives, do not map as seamlessly to the HPX-5 model. In particular, because HPX-5 does not have powerful, higher-level distributed constructs, the work of efficiently mapping DARMA's collection-based constructs is substantial. Moreover, HPX-5 does not offer direct support for distributed collectives (because they are not part of the ParalleX model), so that part of the implementation required substantial additional effort. HPX-5 also does not currently provide any direct support for load balancing, so any backend-provided load-balancing will either require a direct implementation in the glue code or additional work by the HPX-5 team. Nonetheless, an initial implementation of the current backend API has been completed this year, though much in the way of hardening, debugging, and optimization still needs to be done.

Several revisions were made to the backend API based on feedback from this collaboration. One development was the realization that the `Flow` abstraction (and the corresponding `Use` abstraction on the frontend

side that groups Flow s together) can express dependency relationships, but it was difficult for the backend to discern anti-dependency relationships from Use s and Flow s alone. To this end, we introduced a new but analogous backend API abstraction called an AntiFlow that represents connections between Use s that have an anti-dependency relationship. This abstraction significantly simplifies the backend logic, particularly with respect to semantic reasoning, and enables the backend developer to focus more effort on optimal execution of the program task DAG (and less effort on correct execution).

Lastly, there is another work-in-progress prototype DARMA backend based on the ParalleX model, leveraging HPX-3, which is developed out of Hartmut Kaiser's Stellar group at Louisiana State University.

3.4 MPI

While CHARM++ and HPX provide highly asynchronous frameworks, a runtime based on MPI is perfectly capable of implementing the core backend API. While any complexity can theoretically be built using MPI, we summarize here the "simplest" implementation that uses as many existing features and libraries as possible.

An MPI backend is characterized by static or semi-static data distributions. Unlike HPX and CHARM++ that have active global address spaces (AGAS) or chare arrays with significant built-in infrastructure for dynamic data distribution, no such infrastructure exists within MPI. Thus, a static mapping from Flow to physical location (rank, address) must be maintained on all ranks. The vast majority of reference counting on dependencies is local. The distributed part of dependency counters comes in publish/read or collective operations. Because the MPI backend maintains a static data distribution, all publish/read operations can be directly mapped onto MPI Isend/Irecv calls. Reference counters simply need to be associated with an MPI request and can be decremented when a corresponding MPI_WaitX call completes. Because publish/read operations map onto non-blocking sends, the MPI backend can still provide some amount of communication/computation overlap, although likely less than, e.g. CHARM++.

Collectives are again mostly straightforward. Static data distributions can be associated with an MPI communicator via MPI_Comm_split . Leader-based reductions, e.g. can first collect all contributions within a node and then directly call an MPI_Reduce on the associated communicator. Some loss of efficiency may occur for custom data types since arbitrary C++ data types cannot be easily mapped onto the C-style structs required for MPI datatypes.

The semi-static data distribution is still compatible with load balancing. "Semi-static" indicates that it can only be updated in a globally synchronous manner in which all ranks agree on a distribution. This contrasts with a fully dynamic distribution in which data and task migration can occur locally between two ranks without all ranks knowing. While this semi-static data and task distribution may seem incompatible with load balancing, it actually achieves the state-of-the-art load balancing used in many applications. The "gold standard" for MPI load balancing is often Zoltan or Metis partitioning software to break up complex workloads into uniform chunks. As workloads evolve over many timesteps (e.g. in adaptive mesh refinement), Zoltan or Metis can be re-run to rebalance the evolved workload. These graph partitioners require a global synchronization to run and migrate data. While DARMA does not directly have global barriers as a programming model concept, certain task launches can be intermittently executed as a global synchronization with a load balancing phase.

DARMA already requires serialization to be defined for every piece of data. Additionally, the task/depen-

dency model of DARMA can be combined with timers or performance counters to transparently build a weighted graph representation of the global workload that is immediately compatible with Zoltan or Metis. Thus, with no hints or input from the application developer on the optimal load balance, DARMA can automatically perform a best-guess load balancing relying on the established heuristics within Metis to generate balanced distributions.

3.5 Legion/REALM

Of all the existing AMT runtime work, the DARMA programming model is most similar to that of Legion. In spirit, Legion's programming model is similar to DARMA in that it provides sequential semantics and leaves concurrency extraction to the programming system, though there are several key aspects of the description and semantics of data and control (tasks and task collections) are different in Legion (for instance, Legion has no direct equivalent of DARMA's publish/fetch semantics on `AccessHandleCollection` s). Because of the semantic similarity to DARMA's frontend programming model, it makes more sense to discuss a mapping of DARMA *frontend* semantics onto Legion (rather than the DARMA backend API). However, exposing an interface to the DARMA frontend is beyond the scope of the work undertaken this year, and it is hard to speculate on the efficiency of doing so with a reasonable degree of confidence.

Legion mainly implements the higher level logic of deriving concurrency, which is the role of the translation layer in DARMA. As such, Legion does not provide a suitable backend target but rather an alternative implementation of a dependency analysis component (see Section 2.3) within DARMA. Instead, DARMA has a relatively straightforward mapping onto the low-level runtime for Legion, which is called Realm [27]. Realm is an event-based, low-level runtime system designed to execute programs from deferred execution programming models (like Legion or DARMA) on heterogeneous, distributed memory machines. We have briefly explored what an implementation path forward might look like for DARMA on top of Realm, and initial indications are promising.

Realm has an asynchronous countdown trigger abstraction called a `Realm::Barrier` (which is actually quite different from, e.g., an MPI barrier) which maps pretty directly onto DARMA's concept of `darma::Flow` s and `darma::AntiFlow` s.[1] It has a mechanism for creating single and group execution of a closure (`Realm::Processor::spawn()` and `Realm::Runtime::collective_spawn()`, respectively) dependent on one or more `Realm::Event` s (of which `Realm::Barrier` s are a special case); the mapping from DARMA's `darma::Task` and `darma::TaskCollection` constructs to these Realm analogues is therefore straightforward. Dynamic dependency creation between sibling tasks (via `publish()` and `read_access()` in the DARMA frontend) is not directly supported, but can be easily emulated using tables of `publish()` entries (for each index in a given `darma::AccessHandleCollection` protected by Realm `Realm::Reservation` constructs (the authors of Realm describe `Realm::Reservation` s as "a deferred execution version of locks that do not block"). The interaction of the DARMA backend API with the Realm data model is also straightforward. Broadly speaking, data in Realm is managed by instances of `Realm::PhysicalRegion` objects (each tied to a given `Realm::Memory`) that represent a piece memory addressable by some subset of the `Realm::Processor` s on the `Realm::Machine`. Like almost everything in Realm, copies of data between two `Realm::PhysicalRegion` s is a deferred op-

[1] In this section, for brevity and clarity, we use the namespace prefix `darma::` for symbols that are relevant to the DARMA API, even if it is not their formal namespace (for instance, we use `darma::TaskCollection` for the symbol that is formally in qualified as `darma_runtime::abstract::frontend::TaskCollection`, and we use `darma::Flow` to mean the type defined by the backend implementer via `darma_runtime::types::flow_t`). Realm symbols use their actual qualified names as seen in code.

eration that returns an `Realm::Event` representing the completion of the copy, on which other operations (such as a `Realm::Processor::spawn()`) can be predicated. The DARMA-Realm backend would merely have to create copy operations for each dependency `darma::Use` that is not accessible on a given `Realm::Processor` that the backend wants to execute a `darma::Task` on and then predicate the `Processor::spawn()` on those copy operations (in addition to any `Realm::Event`s from the `darma::Flow`s and `darma::AntiFlow`s in the `darma::Task`'s `darma::Use`s).

While the DARMA backend API generally matches the Realm runtime system, some backend details are awkward to implement with Realm. Specifically, the implementation of `darma::Runtime::get_running_task()` in the DARMA API does not appear to be straightforward in the API provided by the Realm runtime. Careful inspection of the implementation details of Realm reveals that it should be safe to use a C++ `thread_local` variable for such purposes, since Realm itself uses a thread local variable for its implementation of `Realm::Processor::get_executing_processor()` (and, presumably, an operation spawned on a `Realm::Processor` does not migrate during its execution in general). However, there does not appear to be a guarantee in the Realm specification that such use of `thread_local` will always be safe, and operating outside the bounds of the specification for a given runtime is not ideal. Unfortunately, the implementation of `darma::Runtime::get_running_task()` (and the efficiency thereof) is critical to the DARMA translation layer and is unlikely to be removed from the backend API in the foreseeable future. Other DARMA backend API methods that do not have much contextual information conveyed through arguments, such as `darma::MemoryManager::allocate()` and `darma::MemoryManager::deallocate()`, have similar issues. Again, implementations of all of these methods can be done using Realm, but it is unclear where the most straightforward implementations lie with respect to compliance with the Realm specification.

By far the most awkward aspect of a theoretical implementation of the DARMA backend API using Realm, however, is the mismatch in the data models. Data structures in Realm are, in general, very statically sized and shaped, while data in DARMA is expressed through the C++ object model, which can often have many dynamically sized and shaped components. While Realm does offer support for arbitrary serialization and deserialization of objects, its interface for this is a poor match for the DARMA backend API, since it relies on the preservation of C++ type information, which is currently stripped out by the translation layer before it reaches the backend API. Furthermore, Realm assumes that application data is thoroughly described in terms of index and field decompositions. DARMA's data model currently does not expose this sort of information (which can often be implemented with, for instance, `std::vector<T>` for indexing and data members of a C++ object for fields, where DARMA is currently only informed of the object itself and not the field or index decompositions).

3.6 Summary of Findings

Table 3.1 which provides an overview DARMA implementation strategies for multiple runtime technologies, capturing how direct the mapping is between DARMA's requirements and the runtime system's underlying functionality. Although, by design, DARMA's backend API captures a declarative representation of an application that can map to a variety of runtime implementations, this year's work has highlighted opportunities for componentization within both the frontend and backend that could result in improved scheduling and optimization strategies long-term. In particular, as the the DARMA-CHARM++ backend was built, we discovered that a few subsets of the backend interface had a very direct mapping to CHARM++, but the majority required significant implementation effort due to the semantic disparity between a heavily

data-centric model (DARMA) and a message-driven, concurrent object model (CHARM++). Ostensibly, to reduce backend implementation complexity, some pieces of the backend glue layer may be generalizable and reused across different runtimes to potentially simplify the development of new backend glue layers. Note that the following is not an extensive, rigorous investigation of this topic, but more of a cursory assessment.

Consider the part of the DARMA interface that provides collective support for task collections and data collections (i.e., `AccessHandleCollection`). Currently the DARMA frontend API only supports two distinct collective operations (`reduce` and `allreduce`), but many more are planned such as `gather` /`allgather` , `scatter` /`allscatter` , etc. Depending on the backend runtime system, these may already exist with similar semantics to DARMA, but is unlikely due to the asynchronous nature of DARMA's collectives. Thus, to fully implement the DARMA backend API, each glue layer may have to re-implement each collective with the proper DARMA semantic. Decades of past research has extensively studied how to efficiently support different collectives with varying objectives that all may be applicable depending on runtime knowledge (e.g., topologically aware, network-specific algorithms or bandwidth-minimizing algorithms). It is unrealistic to expect backend glue writers to provide a highly-tuned, data-dependent implementations if the targeted runtime is deficient.

This problem can be mitigated by building a general backend collective component that matches DARMA semantics but is runtime agnostic (*e.g.*, only requires implementing a `send` function). Such a component may even be universally applicable beyond DARMA, improving runtimes as the broader community develops best practices and collectively develops general libraries for HPC.

	Manage data dependencies between tasks (data inputs and outputs)	**Determine and track placement of data, tasks, and task collections across distinct memory spaces**	**Coordinate data movement utilizing the underlying communication transport layer**	**Implement collective operations**
CHARM++	*Not a direct mapping*: Implements local and distributed schedulers in CHARM++ user-space to schedule and track DARMA data.	*Not a direct mapping*: Utilizes CHARM++'s groups, nodegroups, and chare arrays to manage DARMA tasks and data; carefully maps related task collections in DARMA to persistent CHARM++ chare arrays for effective LB.	*Close mapping*: Uses CHARM++'s native, platform-specific network layers (ugni, ibverbs, TCP/IP, MPI) to transfer data; performs (de-)serialization by utilizing CHARM++'s extensive PUP (Pack-/UnPack) interface.	*Weak mapping*: CHARM++ has a native `reduce` support, but not an `allreduce` . Since CHARM++ has more restricted `reduce` semantics, implementing DARMA's `reduce` /`allreduce` requires a complex, custom CHARM++ reduction operation, but reuses CHARM++ topological spanning trees.
HPX-5	*Close mapping*: HPX-5 provides lightweight control objects (LCOs) as a first-class abstraction that the execution of tasks can be easily predicated on.	*Weak mapping*: Tracking placement of data is relatively easy since an address in the active global address space (AGAS) can be associated with the data, but determining the placement must be done manually in the glue layer (HPX-5 only has a few primitive scheduling algorithms implemented).	*Weak mapping*: Transfer of data can be done via the AGAS, but there is no serialization layer to hook into and interfacing with serialization in DARMA must be done manually	*Not a direct mapping*: HPX-5 has only minimal support for collectives, as they are not part of the ParalleX execution model
Legion/Realm	*Very close mapping*: Realm contains constructs designed to implement very similar semantics to those of the DARMA frontend	*Weak mapping*: Tracking placement of data and tasks can be done easily enough using Realm's system of unique global IDs, but determining placement requires careful interfacing with Legion's Mapper layer (and for a custom Mapper to be written)	*Not a direct mapping*: Legion has a mostly static view of data, and moving around data in DARMA that has the full flexibility of the C++ object model is awkward.	*Close mapping*: Collectives in Realm are deferred and asynchronous just like in DARMA, so the mapping is excellent.
Basic MPI Backend	*MPI provides no infrastructure*: Glue layer must implement all functionality. A minimal implementation would execute all tasks sequentially with no dependency analysis. Some context switching or deferred execution is required to satisfy concurrency model.	*MPI provides no infrastructure*: Glue layer must implement all functionality. A minimal implementation would use a semi-static data mapping.	*MPI provides no infrastructure*: Glue layer must implement all functionality. A minimal implementation would use buffered sends for non-blocking communication and the semi-static data distribution to map all publish operations directly to `MPI_Isend`	*Implementation-dependent*: Given type information and semi-static mapping, MPI could directly leverage non-blocking collections for built-in datatypes.

Table 3.1: Summary of Backend Mappings

This page intentionally left blank.

Chapter 4

Findings on Interoperability

Sandia's ATDM program embraces a component-based approach to application development that stems from a rich history of component-based software engineering research and development within the broader HPC research community[1]. However, recent increases in system parallelism and heterogeneity pose challenges with regards to component *interoperability*, or the ability of separate software components to efficiently share execution resources, share memory resources, and exchange information. This is due to the fact that increases in system parallelism and heterogeneity are driving corresponding increases in

1. the number of performance tools, runtimes, and languages aimed at gleaning performance from new architectures, and
2. the complexity of the component-based systems that are developed and deployed.

The core problem is an underlying assumption by many frameworks that all system resources are available for their exclusive use. These interoperability challenges are not unique to AMT runtime systems. Furthermore, even though different thread models (OpenMP, Pthreads, or std::threads) expose similar concepts and data structures such as thread id, thread private data and thread synchronization, interoperability is not well defined in the standards of the various models. The differences in behaviors can grow combinatorially with the number of supported thread models, computer architectures, and compiler versions. For instance, thread private data allocated using a particular thread model can be in an undefined or inconsistent state when accessed from threads created by a different model (this is particularly true for initial values and default construction). Also, the various models make certain assumptions about how locks (mutexes) can be acquired and released and what happens when they change state. These differences can lead to subtle, hard to discover bugs which are compiler and architecture specific. Consequently, an application comprising components that use different node-level libraries will require arbitration strategies. For example, thread private data can be replaced with lockfree map from thread id to the data, but this technique is often slower than the vendor-provided APIs and more cumbersome to use. Beyond the existing challenges raised by more traditional software stack components, however, the asynchrony of AMT runtimes raises additional interoperability challenges. In this Section, we explore interoperability challenges and solutions at both the network- and node-level within a DARMA software stack. This Section presents preliminary results and findings that are a part of longer-term research efforts between several research teams within Sandia's ATDM program (KOKKOS [25], NoRMa, a Node-level Resource Manager [29], DARMA) and CharmWorks [30].

[1] dating back twenty years to the Common Component Architecture Forum [28]

4.1 Node-level Resources

4.1.1 KOKKOS: Performance Portability

There are a number of programming model and library options for exploiting node-level concurrency. These include architecture- and vendor-specific solutions that pose a performance portability challenge for application codes, as well as a number of broader efforts targeting performance portability. KOKKOS [25] is a C++ library that provides programming model abstractions for performance-portability across many-core architectures. Specifically, KOKKOS provides abstractions to:

- Identify / encapsulate grains of data and parallelizable operations
- Aggregate these grains with data structure and parallel patterns
- Map aggregated grains onto memory and cores / threads

4.1.2 AMT+KOKKOS: Design Challenges and Considerations

KOKKOS is designed to be agnostic to inter-node parallelism. It integrates most easily with MPI programs, but can work with any distributed model. KOKKOS used to make the assumption that a single master thread would launch data parallel work on a compute node. This limited its use within an AMT framework so this restriction was relaxed in the OpenMP backend. Using `Kokkos::partition_master` (or nested OpenMP regions), control flow can be split into multiple master threads so that an AMT task scheduler can execute multiple data parallel tasks simultaneously. The `Kokkos::Thread` backend is being refactored to also support the `partition_master` API. Uintah [9] is using `partition_master` and the OpenMP backend to create a new task scheduler to run their AMT tasks. This new scheduler solves many of the problems encountered by Uintah's current `pthread/std::thread` schedulers, namely incompatibility with BLAS and Hypre which depend on OpenMP.

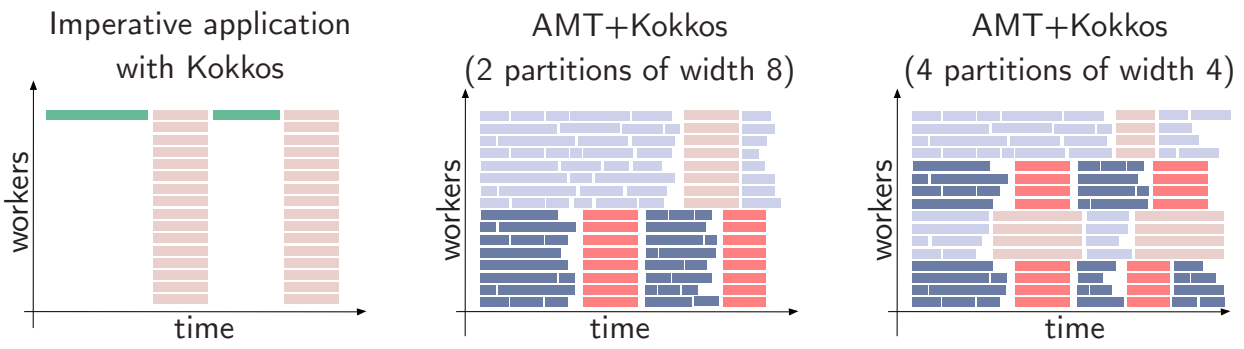


Figure 4.1: A comparison of worker activity over time for several application scenarios. On the left, a simple imperative application code is shown with serial work in green and KOKKOS work in red. The middle and right images illustrate AMT+KOKKOS for two different partitionings of resources. The threaded AMT work is shown in blue and the KOKKOS work is shown in red.

Figure 4.1 illustrates worker activity over time for node-level resources in several application scenarios. On the left is a traditional, imperative application code. Within the node, the "hand-off" between the serial application code and the KOKKOS backend is a fork-join of system resources (serial work is denoted in green and KOKKOS work in red). AMT runtimes, however, are frequently threaded, with work dynamically being scheduled to work queues on different threads. The middle and right-hand figures illustrate AMT work de-

noted in blue and KOKKOS work shown in red. These illustrate the use of `Kokkos::partition_master` to partition the node into distinct collections of resources belonging to independent execution space instances (in this case two and four partitions are shown with widths eight and four respectively). The partitioning of resources enables the overlap of AMT tasks and KOKKOS parallel patterns across different partitions. Partitioning can be nested, resulting in a node being partitioned into execution space instances of different widths. The extent to which nested and/or dynamic partitioning is achievable and useful for ATDM applications will be explored in FY18. In the AMT+KOKKOS images, the hand-off between node resources introduces a synchronization, which can be costly depending on the current state of the dynamic scheduler.

Efforts to coordinate between DARMA and KOKKOS for management of execution resources led to the development of a new component, the Node Resource Manager (NoRMa) [29]. NoRMa presents two levels of interfaces. The lower layer maintains an inventory of available resources (cores and hardware threads) and responds to requests from other software components for those resources. Once resources are reserved, threads may be launched onto those resources either directly by the owning component or using an optional higher level NoRMa interface. This higher level interface allows the "donation" of threads, bound to particular cores, from one component to another. This would accommodate, for example, the transfer of threads from DARMA to a KOKKOS execution space instance. The lower level interface is implemented using the hardware locality (HWLOC) library [31], while the higher level interface is implemented using C++ `std::threads` .

While designing and implementing the approach described above was a useful exercise, ultimately it was not adopted as the solution for DARMA +KOKKOS interoperability going forward. Instead, the OpenMP affinity layer was chosen as the mechanism to manage execution resources. Reasons for this choice include the need to work with native vendor OpenMP libraries for mixed KOKKOS and OpenMP node programming (such as with OpenMP-enabled Intel Math Kernel Library calls), preferences for OpenMP among Sandia Integrated Codes developer teams, the precedent of using OpenMP affinity for Uintah+KOKKOS integration, and the CHARM++ runtime library's interference with HWLOC controls. The integration effort using OpenMP affinity, described in the next section, still leaves considerable work to be done in the area of resource management for efficient execution.

4.1.3 DARMA +KOKKOS: Strategy and Implementation Details

Within Sandia's ATDM program, we will rely upon vendor-supported OpenMP runtimes to achieve node-level resource management and arbitration. With regards to DARMA-CHARM++ + KOKKOS, this poses a challenge because the CHARM++ runtime currently does not work with threads spawned and managed by vendor-supported OpenMP. As part of an on-going collaboration with CharmWorks this year and next, the CharmWorks team is prototyping CHARM++ runtime system modifications to use threads spawned and managed by vendor-supported OpenMP runtime libraries instead of internally-spawned platform-specific threads.

In the meantime, we have implemented a prototype demonstrating KOKKOS interoperating with the DARMA-ONNODE backend. In this prototype, the exiting and re-entering of DARMA and KOKKOS work queues happens at function boundaries. The synchronization required before launching one or more tasks containing KOKKOS is accomplished by closing a structured block indicating an OpenMP parallel region. Figure 4.2 demonstrates the use of a `Kokkos::parallel_reduce` in the Jacobi benchmark to compute the residual. Table 4.1 shows timings running the code using DARMA-ONNODE +KOKKOS backend for various partition sizes on a Mutrino KNL node.

```
create_work(reads(prev), is_data_parallel=      true, [=]{
  // These could also be held in, e.g., Kokkos::Unmanaged Views
  double const* prev_buf = prev.get_value().data();
  double* next_buf = next.get_reference().data();
  max_residual.set_value(-1e12);
  // since we marked this task as data_parallel , we can call
  // Kokkos as usual here:
  Kokkos::parallel_reduce(size_x * size_y, [=](      int ij, double& resid){
    int i = ij / size_y;
    int j = ij \% size_y;
    if(not (i == 0 || j == 0 || i == size_x-1 || j == size_x-1)) {
      // compute the next iteration's values
      next_buf[ij] = 0.2 * (
        prev_buf[INDEX(i,j )] + prev_buf[INDEX(i-1,j)]
        + prev_buf[INDEX(i,j-1)] + prev_buf[INDEX(i+1,j)]
        + prev_buf[INDEX(i,j+1)]
      );
      // compute the local residual
      double local_diff = fabs(next_buf[ij] - prev_buf[ij]);
      // and see if it's the maximum
      resid =  std::max(local_diff, resid);
    }
  }, Kokkos::Experimental::Max<    double>(max_residual.get_reference()));
);
```

Figure 4.2: Example code comprising both KOKKOS and DARMA.

Table 4.1: Timing results for the code shown in Figure 4.2 for a variety of different partition sizes

# of Partitions	Partition size	Average Time Per Iteration (s)
8	8 cores	2.115
16	4 cores	1.827
32	2 cores	1.985
64	1 cores	2.487

FY18 will see completion and hardening efforts with the DARMA-CHARM++ backend and KOKKOS with vendor-supported OpenMP supplying underlying thread resources. This year's efforts have focused on execution resource management and arbitration and next year we will extend our focus to include interfacing of data management abstractions, motivated by the PIC application use case. Additional research activities will explore nested partitioning of node resources (beyond the single-level implemented in the current prototype). Possible future research avenues also include a more integrated strategy between DARMA and KOKKOS whereby synchronization points are avoided in the hand-off between runtimes. At a minimum, to avoid the synchronization at the start of a task containing KOKKOS work, the KOKKOS worker threads would need to be able to overlap with DARMA worker threads doing potentially unrelated work within a partition. One approach to this would require a means for the DARMA runtime to start up KOKKOS worker threads asynchronously. Other optimizations would require an expression of the KOKKOS kernel itself in a form that can be scheduled separately. These optimizations could result in significant performance improvements as the as the number of threads operating on a memory space increases.

4.2 Network Resources

Within our ATDM applications we will have components that rely on a distributed DARMA software stack coupled with components written in a traditional imperative MPI-style (e.g., solvers using Trilinos [32]). While a demonstration of this capability is not planned until FY18, we include a brief discussion here nonetheless regarding the planned approach. We note that all of the leading AMT runtimes provide a hand-off mechanism to support phases of computation in which ownership of network resources toggle between the AMT runtime and the imperatively written MPI code. This phased approach is not invasive to the imperative MPI code and CHARM++ provides smooth integration with existing MPI code in this manner. In DARMA, a natural place to perform such a hand-off is after a `create_concurrent_work` . In summary, the abstractions and the underlying technologies are available and in FY18 we will be performing the engineering work within the DARMA-CHARM++ backend to support this capability.

This page intentionally left blank.

Chapter 5

Findings on Performance and Productivity

In this chapter, we present a variety of performance and productivity findings. We begin by focusing broadly on productivity benefits of DARMA's programming model. Next, we move to a discussion of system details informing our performance results, including the architectures, affinity settings, and compilers used within our studies. This leads to a presentation of benchmark- and application-specific performance and productivity findings. For each benchmark and proxy application considered, there are both Haswell and KNL results. We present strong and weak scalability results, along with other detailed performance analysis views, e.g. communication graphs over time, to give a detailed view into the DARMA-CHARM++ results. We note that within our experiments, we have fixed the high-level algorithm for solving a given problem (using the same high-level approach across runtime systems). However, the semantic information captured differs between the imperative MPI and DARMA implementations, and we discuss the relevant differences within relevant Sections.

5.1 Productivity within the DARMA Programming Model

The key "asynchronous" ingredient in DARMA is the complete lack of blocking function calls or thread locks. It is exactly the blocking `MPI_Wait` call that creates the most difficulty in making MPI more asynchronous. While the mechanics of `MPI_Isend` and `MPI_Irecv` calls themselves are not difficult to use and understand, actually implementing an optimal overlap of communication and computation can be difficult depending on the application. Some issues include:

1. How much computation is appropriate to interleave between `MPI_Wait` calls? Too little computation and execution will stall waiting for a send to complete. Too much computation and not enough forward progress will be made on other communications.
2. The MPI specification makes no guarantees on forward progress between an `MPI_Isend/Irecv` and the corresponding wait. For certain network protocols, this creates the illusion of communication/-computation overlap. However, the full communication delay still occurs inside the `MPI_Wait` .
3. Overdecomposition to tune data block granularity or improve load balancing in MPI requires the application to explicitly manage a multi-level decomposition. Decomposing problems across MPI communicators with fewer ranks than `MPI_COMM_WORLD` is straightforward. Even with work on MPI shared memory and MPI endpoint extensions, creating an MPI communicator with *more* ranks than `MPI_COMM_WORLD` is not easily done. Thus applications must manage a problem decomposition across ranks and then also problem decomposition within ranks.
4. In MPI, data migration during the load balancing and consequent connectivity information update is explicitly performed and managed by the application itself. This will potentially introduce additional synchronizations.

5. Interleaving MPI calls within an OpenMP or other multithreaded region can result in poor performance if the `MPI_THREAD_MULTIPLE` implementation uses global locks. Explicit thread synchronization is required for other MPI implementations.

Given these programming productivity challenges, we can better understand what DARMA provides:

- A fully non-blocking programming model that a programmer can reason about via simple means, as if it were executed sequentially.
- Asynchronous progress on communication with no concern for thread safety or optimal task sizes for interleaving.
- Easy separation of problem decompositions from both the number of nodes and number of processes physically used to run.
- Mechanisms to enable load balancing capabilities supported by backend runtimes with little to no effort from an application programmer.

In the programming model itself, MPI most naturally implements a communicating sequential processes (CSP) model, although the notion of "sequential" is now more complicated with emergence of MPI+X models. All inter-process parallelism must be explicitly managed, leading to the explicit synchronization (blocking calls) apparent in MPI code. DARMA instead follows a "root" model. The code can essentially be read as if it were executing on a single thread. The application developer annotates their code with calls to `create_work` and `create_concurrent_work` to indicate sections of code where parallelism is useful. The runtime is then free to implement the parallel execution as best it can given available machine resources. While this has the appearance of auto-parallelization, the application developer must still define task granularity and indicate where parallelism is most important - operations that are exceedingly difficult for an auto-parallel compiler. DARMA therefore aims to avoid explicit threading and blocking constructs in the programming model while still requiring extra semantic information missing in a serial C/C++ code.

DARMA also captures logical labels for all data structures, which is lacking in MPI. MPI is simply C/C++Fortran code. While some memory can be allocated through the MPI runtime, MPI has no concept of a block of memory being logically associated with a given data structure. DARMA, however, assigns each data structure (access handle) with a unique logical label that allows it be globally identified. Similarly, while MPI allows C-style struct definitions of MPI datatypes, it does not allow arbitrary C++ type information. DARMA, however, can associate arbitrary type information (critically serialization info) with each global identifier.

Being able to logically track data structures and move them at will using the associated serialization information allows DARMA backends to transparently load balance in a way that MPI (without extra libraries) cannot. A DARMA backend can perform load balancing either on-the-fly through work stealing, as a global reshuffle, or as a hybrid approach using persistence-based work stealing. All of the necessary information for all these implementations is inherent in the extra semantic information DARMA captures.

Can DARMA be considered a superset of MPI that adds functionality? Said another way, given a source-to-source compiler, could a DARMA application be transformed into an MPI application? The DARMA programming model assumes a root task that decides what tasks to perform and spawns parallel work. This differs dramatically from the "democratic" CSP model in which there is no root. Based on something like a rank ID, workers agree in a distributed way on the work distribution. In the most obvious mapping, DARMA code could transformed to MPI as:

```
1 if (rank == 0){
2  //do \darma root work
3 }
4 MPI_Bcast(...);    //create concurrent work
5 if (rank == 0){
6  //do \darma root work
7 }
```

in which rank 0 takes over all root work and all other ranks wait on a broadcast for concurrent work.

The transformation from a rooted model like DARMA to a CSP model like MPI touches on the question of the theoretical efficiency of `create_concurrent_work` , which is the most expensive task scheduling operation within DARMA. The notion of *control replication* is inherent in MPI. Usually, every rank is its own root and makes decisions for itself on what to execute. Given a predefined, semi-static work distribution based on MPI rank, every worker can select the next to run without a root broadcast. This saves significant communication overhead. DARMA is currently not implemented with or specified to allow control replication. Control replication requires certain restrictions be placed on what a root task can do at certain times. A DARMA specification and corresponding implementation that enables control replication is being considered, particularly in the context of a DARMA-MPI backend.

5.2 Trinity: Advanced Technology System - 1

Within the ASC program, the Advanced Technology Systems (ATS) are systems that serve to foster technical capabilities beneficial to the ASC program while simultaneously helping the ASC program prepare for future system architectures. Trinity, the first ATS system (commonly referred to as ATS-1), is the target for the performance analysis within this Level 2 milestone. It was acquired and is managed by the New Mexico Alliance for Computing at Extreme Scale (ACES), a joint collaboration between Los Alamos National Laboratory and Sandia National Laboratories.

Trinity is based on the Cray XC30 architecture, with a mix of Haswell and KNL processor types and an Aries interconnect with Dragonfly topology. Figure 5.1 presents an overview of the Trinity system (image courtesy of ACES). The mix of processors enables support for current ASC programs while facilitating the development and use of emerging programming models and workflows. In this report we show performance results on both Haswell and KNL architectures.

The Haswell partition comprises 9436 compute nodes. Each Haswell compute node features two 16-core Haswell processors operating at 2.3 GHz, along with 128GiB of DDR4-2133 memory, spread across 8 channels (4 per CPU). The KNL partition has 9984 Xeon Phi Knights Landing (KNL) based compute nodes. Each KNL compute node comprises a single KNL processor with 16 GiB of on-package memory and 96 GiB of DDR4-2400 memory. Figure 5.2 provides an illustration of the Haswell and KNL compute nodes. The Haswell partition delivers 1 PiB of memory capacity and 11 PF/s of peak performance and the KNL partition delivers 1 PiB of memory capacity and 29 PF/s peak performance.

For practical reasons, unless otherwise stated, all smaller-scale performance results (up to 64 nodes) were run on Mutrino, an Application Regression Test (ART) System for Trinity residing on Sandia's unclassified network. Mutrino is intended to replicate Trinity architecture at a smaller-scale, see Figure 5.3.

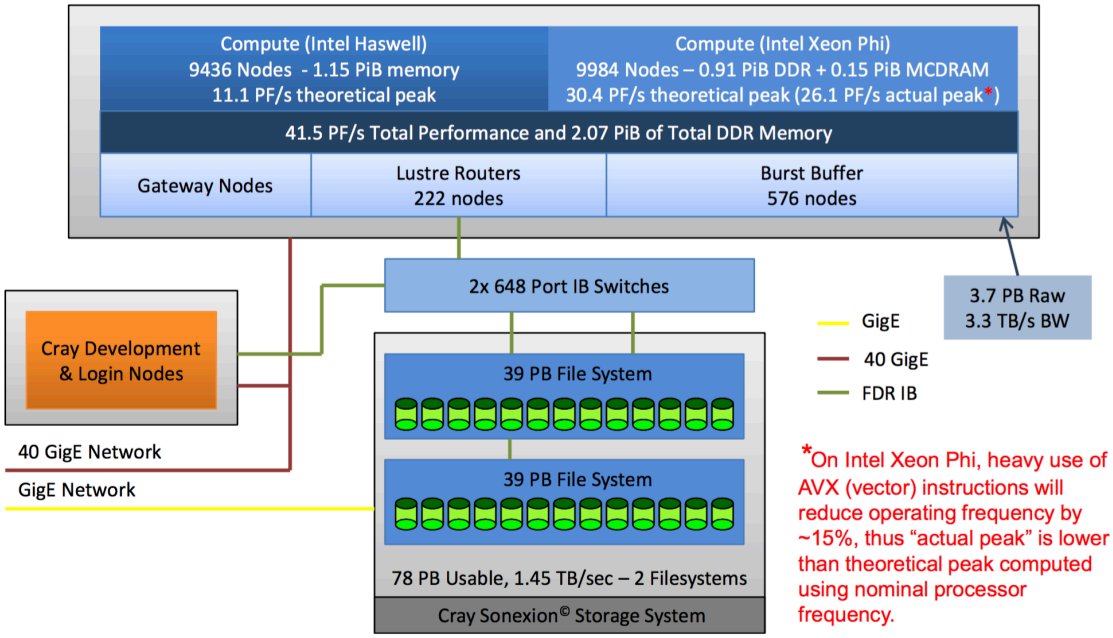
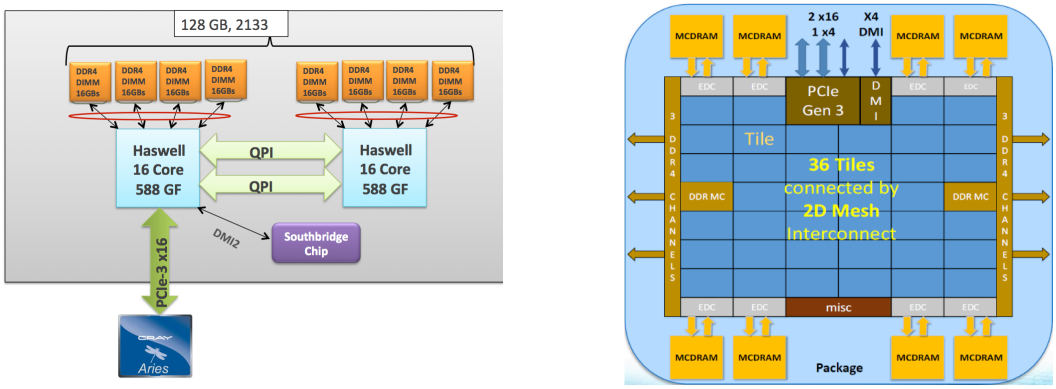


Figure 5.1: An overview of the Trinity Architecture (image courtesy of ACES)



(a) Haswell **(b)** KNL

Figure 5.2: Diagrams of Haswell and KNL compute nodes on the left and right respectively (images courtesy of ACES).

5.2.1 Haswell and KNL Differences

Haswell is a "standard" x86 CPU that supports 2-way hyperthreading and has 256-bit vector extensions. On Mutrino and Trinity, there are two sockets each with 16 cores at 2.3GHz. Knight's Landing (KNL), in contrast, is a many-integrated core (MIC) architecture based on Xeon Phi processors. KNL emphasizes massive on-node parallelism, providing 68 cores each with 4-way hyperthreading and 512-bit vector extensions. Each KNL provides high-bandwidth Multi-channel Dynamic Random Access Memory with more than 460GB/s to feed the as-many-as 272 threads, compared to only 68 GB/s on Haswell combined over 4 DIMMs. Although the cores in Xeon Phi are "lightweight" relative to Haswell (1.4GHz), they are still x86 superscalar cores. Given the massive combined thread and vector parallelism on KNL and increased memory bandwidth, it should surpass Haswell on highly parallel, numerically intensive code. One could

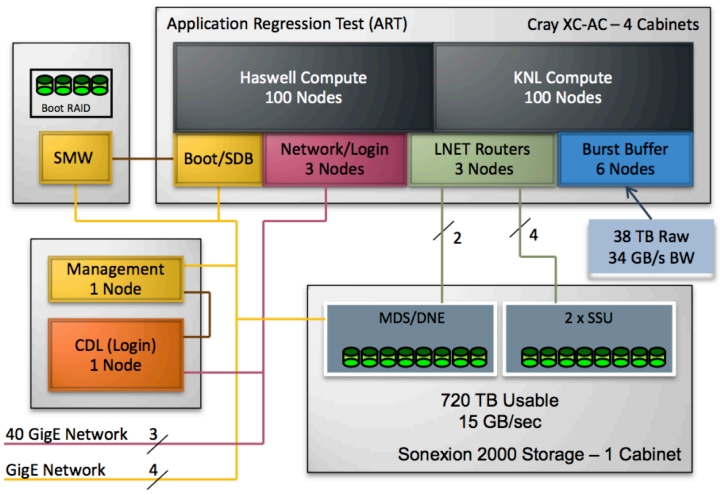


Figure 5.3: An overview of the Mutrino Architecture (image courtesy of ACES)

expect individual Haswell cores, however, to have better serial performance and perform better completing system tasks like communication.

5.3 C++ Compilers

Within our studies we ran a series of experiments using the GNU (GCC 6.3.0) and Intel (ICC 18.0.0 beta) C++ compilers. Due to a non-disclosure agreement with Intel, we are only able to provide results for code compiled with the GNU compiler in this report. However, at the granularity of DARMA tasks, differences in performance across different compilers should be relatively negligible compared to the differences between compilers in other parts of the code. For instance, loop vectorization and unrolling takes place at a *much* smaller granularity than that of DARMA tasks, but the efficiency of these sorts of optimizations can vary significantly between compilers. The DARMA frontend has been carefully designed such that the parts of DARMA that the user could reasonably interact with in an inner-loop context (for instance, `AccessHandle::get_value()`) should not interfere with the efficiency of the optimizer. We do not expect that the higher-level abstractions used in DARMA should have much effect on the efficiency of compiler optimization for low-level kernels, and thus far we have seen no evidence to the contrary.

With respect to an application's usage of DARMA, a much more noticeable difference between compilers is the cost of the compilation itself. DARMA makes extensive use of some newer C++ features from the C++11 and C++14 standards, and many compiler frontends are not yet optimized for handling some of these features. However, the recent trend towards a reduction in the overall number of compiler frontends (for instance, several commercially available compilers from hardware vendors have recently begun using the Clang frontend to interact with modern C++) lowers the risk of this decision, since the majority of the new features used by DARMA are only visible in the foremost passes in most compiler frontend implementations. Furthermore, as compiler implementation efficiency improves and the DARMA translation layer becomes more hardened, we expect that compilation times will decrease; thus, the current compilation cost of applications written in DARMA can be considered a reasonable estimate of an upper bound.

5.4 Benchmarks

The benchmarks presented herein - Jacobi, Molecular Dynamics, and Simulated Load Imbalance - were written by DARMA runtime developers to abstract away much of the complexity of a real application, leaving only a small portion designed to highlight specific benefits and limitations of the programming model and/or runtime system. Specifically, the benchmarks are used to explore:

- At what granularity of work do runtime overheads become apparent for latency-intolerant algorithms?
- Are these overheads masked effectively as you expose greater asynchrony in the underlying algorithm (enabling greater overlap of communication and computation)?
- How effective is the runtime at mitigating load-imbalance?

Data gathered includes quantitative performance data (both Haswell and KNL) in the form of weak and strong scaling studies and graphical views of performance traces that capture, e.g., overlap of communication and computation. A summary of the semantic information captured in the DARMA implementation is provided for each benchmark, along with a discussion of how that information is (or could be) used by a backend to optimize performance.

5.4.1 Jacobi Benchmark

The Jacobi benchmark has memory-bound computation with latency-bound communication, providing a stress test to highlight overheads in the system. It performs a sparse linear solve $Ax = b$ for a fixed matrix A derived from the heat equation on a 2D structured grid. With sparsity, the Jacobi solve for the $k+1^{\text{th}}$ iteration reduces to:

$$x_{i,j}^{(k+1)} = \frac{1}{A_{ij}} \left(x_{i+1,j}^{(k)} + x_{i,j+1}^{(k)} + x_{i-i,j}^{(k)} + x_{i,j-1}^{(k)} \right)$$

where A_{ij} is the diagonal element. The solve computation requires nearest neighbor communication of a halo box of elements $x_{i,j}$ from 4 neighbors (up, down, left, right) followed by a 2D stencil computation. At regular intervals, a reduce collective is required to check a residual for convergence. The implementation of the Jacobi algorithm decomposes the problem into a two-level grid: 1) an $n \times m$ grid of boxes and 2) an $N \times M$ grid within each box. For boxes of size $N \times N$, each box has a communication volume of $\mathcal{O}(N)$ and a computational cost of $\mathcal{O}(N^2)$. Thus, larger box sixes amortize the cost of communication and scheduling overheads. However, overly large boxes may limit opportunities for communication-computation overlap and cache blocking.

Without employing complex locality optimizations, such as time-tiling to recursively optimize cache locality, the computational part of Jacobi is typically highly memory-bound due to the limits of memory hierarchy bandwidth on modern HPC architectures. The halo regions communicated are small (latency-bound rather than bandwidth-bound communication) and the computation per grid point is small. Thus, *a priori*, we would expect performance to be dictated by (1) memory bandwidth and cache blocking and (2) latency hiding of communication.

The MPI implementation of Jacobi is a fairly standard MPI code, but features overdecomposition to directly compare to DARMA. It employs `MPI_Testall` to check for completed requests associated with each overdecomposed block to dynamically launch work on each block as they become ready. The majority of the experiments with MPI are without overdecomposition, providing very little overlap of communication and computation.

5.4.1.1 CPU Binding and Affinity Study

For each of the benchmarks we experimented with a variety of CPU binding and affinity strategies. Our most in-depth study (with the broadest set of parameter options) was performed with Jacobi. The summary of our approach along with our findings for Jacobi are included here. For the other benchmarks and proxy applications we include affinity settings used within the reproducibility notes for their associated plots.

While Haswell has many cores (32) with 2-way hyperthreading, the KNL architecture provides many more possibilities for how exactly on-node parallelism is configured. In our experiments we explored several variables in the configuration including: number of processes per node, number of threads per process, and choice of thread affinity (binding of core to physical core to hypercore). For simplicity on Haswell, we chose to run without hyperthreading for both DARMA and MPI with each task (process) bound to a specific core. For KNL, we scanned numerous options and results are shown Figure 5.4. The main trade-offs to consider are:

- Increasing number of *communication threads*: fewer threads are available for computation, but communication is driven forward more quickly. A communication thread is a specialized CHARM++ thread that is dedicated to performing scheduling operations and data transfer on and off the network.
- Increasing number of hyperthreads per core: more threads actively computing may increase potential cache conflicts and weaken serial performance for each thread.
- CPU binding: Binding tasks to physical cores only or binding tasks to specific hyperthreads.

For MPI, the best performance is actually achieved with no hyperthreading with a single MPI rank bound to a physical core. We observed, for the configurations tested, that increasing to 128 total threads ($2\times$ hyperthreading) slightly degrades performance. In the MPI implementation of Jacobi OpenMP is not used. Increasing the number of hyperthreads utilized requires launching more MPI ranks on the same node and pinning their affinity to a particular hypercore. For the configurations that we explored for DARMA, the optimal choice is 13 processes per node with 16 compute threads (4 compute cores) and 1 communication thread each. Despite dedicating 13 cores to communication, this achieves the best performance. We observed that hyperthreading on compute cores improves performance; $4\times$ hyperthreading per core in the $13\times$ configuration attained better performance than the 13×8 configuration which has only $2\times$ hyperthreading.

5.4.1.2 Mutrino: 64 nodes

Figure 5.5 show the scaling results for Jacobi on up to 64 nodes on Mutrino for both the Haswell and KNL partitions. In general, the Jacobi benchmark in both DARMA and MPI have excellent scalability. Compared to the MPI implementation, DARMA has a relatively constant overhead but scales very well. This result demonstrates that both the DARMA runtime implementation and the Jacobi implementation in DARMA are highly scalable; this is notable due to the nature of the Jacobi benchmark without any overdecomposition: it has very low tolerance of any latency. For Haswell, a nearly straight scaling curve is seen. As the total problem size increases to 128 billion cells, the gap between DARMA and MPI further shrinks. The story is a bit more complicated for KNL. Both MPI and DARMA show super-linear scaling as the problem size per node shrinks down the strong scaling curve. This is likely related to a combination of cache effects and more efficient use of high-bandwidth MCDRAM as the memory footprint per process decreases.

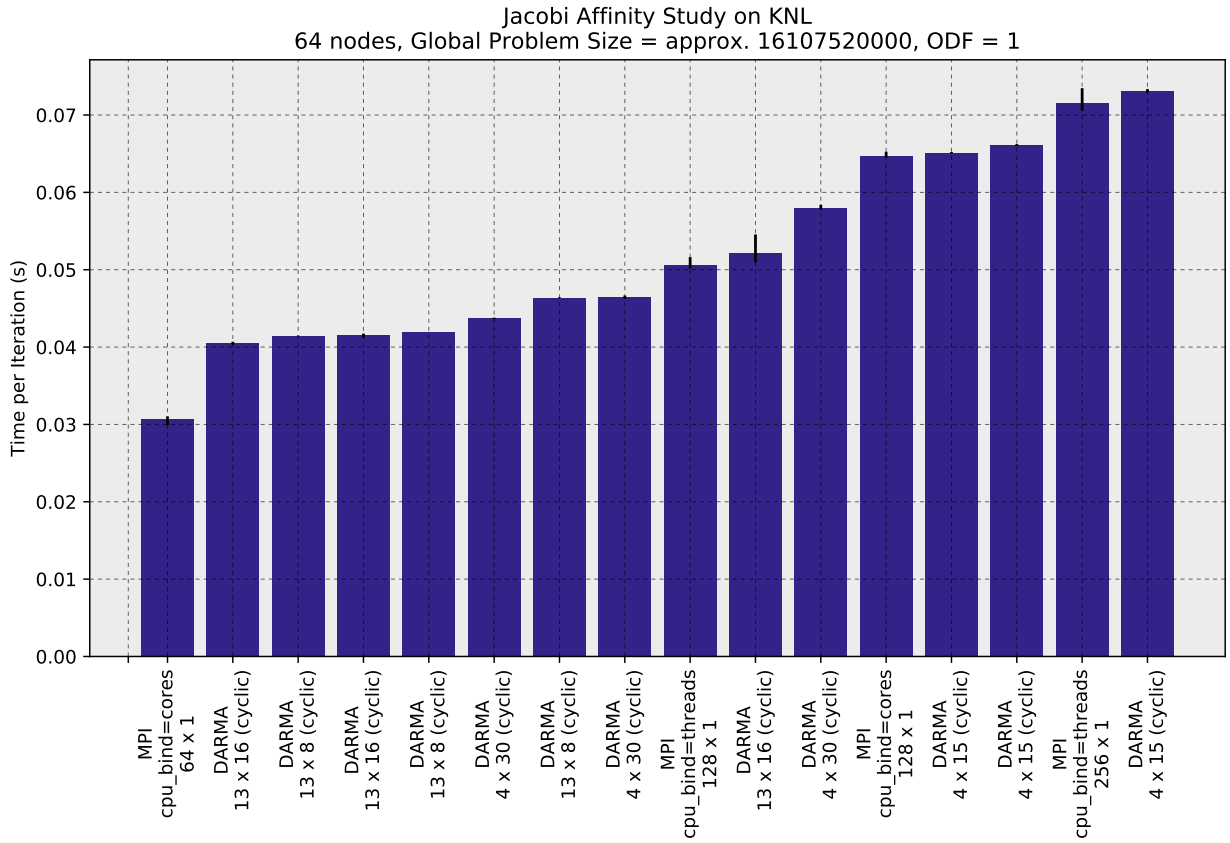


Figure 5.4: Observed performance for varying CPU binding/affinity configurations on Haswell for both MPI and DARMA on Jacobi benchmark for 64 nodes of KNL. Error bars show min and max values observed from multiple trials. Binding configurations examine number of processes per node, number of threads per process, and use of hyperthreading.

5.4.1.3 Detailed Performance Analysis for DARMA

Figure 5.6 shows several figures including: a timeline view that incorporates a sample of the work individual threads are performing across time (white is idle time, maroon is application work), and a graph that histograms the messages sent over time. Jacobi running with DARMA on 64 nodes of both Haswell (left) and KNL (right) are shown in this figure each showing two levels of overdecomposition.

In Figures 5.6a and 5.6b, we present a timeline view of Jacobi with no overdecomposition (ODF=1) for 5 iterations of execution. Surprisingly, regular idle gaps (shown in white) are prominent in the figure, exposing the cost of global synchronization after each iteration in DARMA with the grain size selected. If the grain size is sufficiently large, the global synchronization cost can be hidden, but at this grain size the synchronization becomes a limiting factor in performance.

Note that at 64 nodes we are at the far end of the strong scaling curve for the problem size selected. Additionally, in Figure 5.6b, the execution is not consistent, although the decomposition is perfectly regular. This effect may be due to noise on the machine or how threads are mapped to cores. More detailed examination of the performance with ODF=1 on KNL is necessary to determine the cause of the irregular gaps. Figures 5.6c and 5.6d histogram the number of messages sent in the same time range as the timeline in Figures 5.6a and 5.6b. These graphs are as expected: a burst of communication is present at the end of each

of the 5 iterations.

In the next set of figures, we graph the same configuration, but increase the overdecomposition factor per processor to 4 (ODF=4). Figures 5.6e and 5.6f show the timeline for ODF=4. First, note that with ODF=4 compare to ODF=1, the execution time increases for Haswell by about 32% (0.61s to 0.81s). This is primarily due to overheads in the system that increase with ODF; in particular, the cost of synchronization is increased, demonstrated by the increase in idle time (white) between iterations. With increased overdecomposition, the global `allreduce` operation over all blocks becomes more expensive (more blocks to reduce), increasing the intermittent idle time. The global synchronization further degrades performance by limiting the overlap of halo communication with computation that should be possible across iterations. However, for KNL in Figure 5.6f, overdecomposition does not decrease performance but it remains almost constant due to benefits of smoothing out of irregularities with ODF=1. In general, overdecomposition can have very positive effects on any irregularities that arise within a system even without explicit load balancing enabled. Figures 5.6g and 5.6h show the message sent histogram for ODF=4. As expected, the total number of message increases (increase in halo exchange messages), but the messages sent are better spread out in time. Overdecomposition has the positive effect of spreading communication out better over time, increasing the efficacy of network usage and creating communication/computation overlap.

In its current implementation, the DARMA `allreduce` therefore has a major effect on performance. First, the MPI `allreduce` implementation (without overdecomposition) is heavily optimized compared to the DARMA implementation (a CHARM++ reduction followed by a CHARM++ broadcast). Furthermore, the DARMA `allreduce` does not fully exploit shared memory optimizations. The `allreduce` must occur over a logical chare array in contrast to MPI which reduces across actual physical network endpoints. Thus, there is cost associated with resolving the location of each virtualized chare array element and applying the `reduce` operation. This layer of "virtualization" adds significant overhead. Second, the DARMA scheduler cannot look beyond the `allreduce` since the residual result predicates whether to continue executing (is inside the `while` loop conditional). This limited lookahead inhibits scheduling optimizations for DARMA.

Extra asynchrony can only be added by speculating on the result of the `allreduce` . The while loop must run for many, many iterations until the residual is below the cutoff. Thus, a scheduler can almost always assume the while loop should continue. The ideal solution to the problem would be having the DARMA runtime automatically execute speculatively. This would be the DARMA runtime equivalent to branch prediction in today's superscalar processors. Speculative execution for more asynchrony was not possible to implement in the time constraints. However, we can "simulate" its effect by only performing an `allreduce` convergence check every *N* iterations. Figure 5.7 clearly demonstrates the performance benefits of extra asynchrony from speculation. The computation pipeline has much less frequent stalls.

In Figures 5.7a and 5.7b, we present a timeline view without any asynchronous iterations for ODF=4. This can be contrasted to Figures 5.7c and 5.7d with are the timeline view only computing the residual every 10 iterations and using the application work to overlap with the high latency of the `allreduce` in DARMA. These figures show that the idle time (white) is greatly reduced, resulting in much better performance for both Haswell and KNL; on Haswell the execution time for 5 iterations decreases from 0.81s to 0.54s (a 50% improvement); on KNL the execution time decreases from 0.65s to 0.485s (a 34% improvement). Compared to the best performance achieved previously with ODF=1, Haswell improves by 13% and KNL improves by 34%.

Additionally, by examining the messages sent over time in Figures 5.7e and 5.7f, the communication/computation is even more pronounced and network utilization is highly optimized as the injection bandwidth stress incurred is lowered.

Grain Size, Overdecomposition, and Asynchrony In Figure 5.8, we present processor utilization graphs and communication graphs for two different problem sizes: 30bn and 7bn total cells. The columns from left-to-right compare 4 distinct Jacobi configurations: 30bn, ODF=1; 30bn, ODF=4; 7bn, ODF=1; 7bn, ODF=4. The processor utilization graphs are generated by creating a very dense histogram (small time bins) of how time is spent across all the worker cores (note this does not include communication threads) by the type of activity. Maroon represents application work; white, idle time; orange, dependency analysis; and, blue/green, communication time (publishing and fetching data). By building a histogram over all the workers (1920 workers on Haswell and $13,312$ on KNL) across time, the graph displays compactly how time is generally spent, even at large scale.

In the first two columns (30bn problem size), utilization is better than the latter two columns (7bn problem size). This is expected because as the problem size decreases, less work is present to amortize runtime and other intrinsic communication overheads. The second row in the figure zooms into a single iteration of Jacobi for all the studied configurations. Overall, the patterns observed are similar to the previous: overdecomposition itself is not particularly effective but, an effective combination of overdecomposition and asynchrony, optimizes performance greatly. Asynchrony is even more important for the smaller problem (7bn). For the 30bn problem, performance is increased by 2%, whereas for the 7bn problem size performance is increased by 64%.

5.4.1.4 Trinity: 2048 nodes

Figures 5.9 show scaling results on Trinity up to 2K nodes. As expected, DARMA exhibits some minor overheads relative to MPI given the lack of optimization in collectives and task scheduling overhead. For the strong scaling, superlinear behavior is actually observed due to shrinking memory footprints as the problem size per node shrinks. For weak scaling, excellent performance is seen up to 1K nodes and then suddenly degrades at 2K. More study is required here: most data points for weak and strong scaling were collected on a dedicated reservation. However, due to system failures, some data points had to be collected on a noisy system with other jobs. These data points must be repeated to assess whether a scaling bug exists or the data points were extraneous due to noise.

5.4.1.5 Productivity and Semantic Information Gain

Table 5.1 shows the number of lines of code for the Jacobi implementation in DARMA and MPI: they are nearly identical. On the surface level, the codes approximately support the same features: overdecomposition, asynchronous iterations. The MPI code uses `MPI_Testall` to explicitly test in a while loop whether a given overdecomposed block is ready (the boundary data has been received from all neighbors). However, in the MPI code, the user has to manually manage scheduling each block (waiting for it to become ready, selecting a correct/matching send/receive tag for each block, and explicitly posting sends and receives). This MPI code, while supporting overdecomposition, is not composable with any other code. If another component of a program was executing along side the Jacobi, especially if the other component was dynamic, the scheduling code would become much more complex. In general, without knowing all the components in a program in MPI, the programming must either implement a general scheduling algorithm (which is difficult and duplicates effort on runtime systems), or will rely on a static distribution of work to processors. Furthermore, the MPI Jacobi implementation does not support load balancing, fault tolerance, or data-driven execution. In contrast, all these features are supported by the DARMA execution model: the DARMA code is data-centric, fully composable, and can be load balanced. Additionally, due to the DARMA pro-

Code	SLOCCount (lines of code)*
MPI	395
DARMA	399

* From David A. Wheeler tool that counts lines of code.

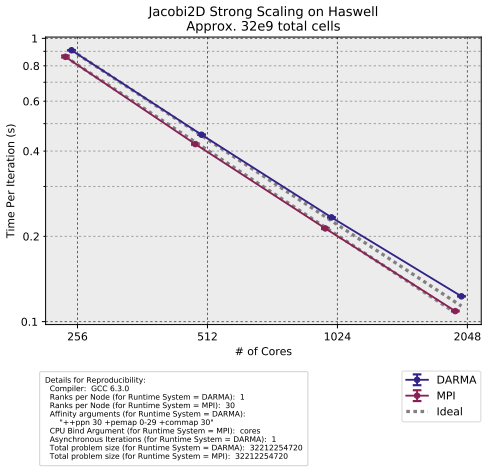
Table 5.1: Lines of Code for Jacobi in MPI vs. DARMA.

gramming model, the Jacobi code in DARMA will not have data races or deadlocks, which can easily occur in an imperative MPI programming model, especially in the face of overdecomposition.

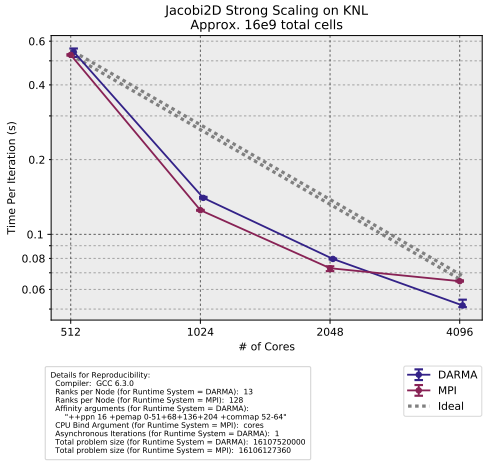
Figure 5.10 shows the implementation of the boundary packing and send/publish code in MPI (Figure 5.10b) and DARMA (Figure 5.10a). Although the codes are similar in length, the DARMA code expresses four tasks that pack the boundary data (`create_work<PackBoundary>(...)`). Due to the permissions expressed, the DARMA system has the opportunity to pack all the boundaries concurrently and publish them. However, in the MPI code, this would be difficult to express—the user would have to employ OpenMP primitives and use `MPI_THREAD_MULTIPLE` to actually make packing and sending the data happen concurrently. By enabling this finer concurrency to be expressed, the DARMA programming model can future-proof implementations in a way that is difficult in the imperative MPI code.

Figure 5.11 shows the outer timestepping loop for MPI (Figure 5.11a) and DARMA (Figure 5.11b). The DARMA code uses the `create_work_while` construct to describe a taskified while loop where each iteration is dependent on the global residual calculation for each iteration. To increase asynchrony, the DARMA code has a sequential inner for-loop (within the body of the `create_work_while`) that spawns up to `async_factor` iterations without checking the residual. As demonstrated by the preceding empirical results, increasing asynchrony substantially improves performance by enabling overlap between iterations. In the DARMA code, this optimization is simple to implement: the sole modification to this code is the addition of the for-loop (see Figure 5.11b, lines 14–20. The ease of implementing this optimization is due to the data-centric nature of DARMA, the automatic communication/computation overlap that is default when uninhibited by synchronization, and the automatically tagging of data that ensures that data between multiple iterations does not conflict. To implement asynchronous iterations in MPI along with overdecomposition, each boundary exchange send/receive would require a unique MPI tag. Using a tag that is unique for each overdecomposed block is not sufficient for correctness with the increased asynchrony. Each MPI tag generated must also ensure that future iteration boundaries receives (from a neighboring overdecomposed block that may be on the next iteration) do not conflict with the current iteration boundary exchange.

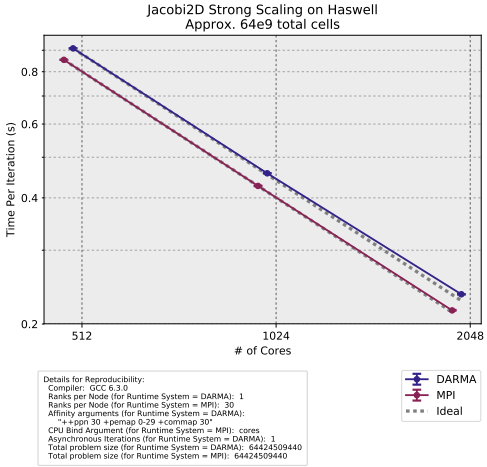
Figure 5.11a shows the timestep loop for MPI, including code to manage a set of `active_blocks` each iteration, which are overdecomposed blocks that have not received all their boundary data and executed the kernel for this iteration. On lines 2–6, the code posts receive and send calls for each block in `active_blocks` (at this point, all blocks designated to this rank are active). On lines 9–25, the MPI code loops through each active block, checks to see if the block is ready (using `MPI_Testall`), and if so runs the kernel, calculates the local residual, and moves the block to `inactive_blocks`. The implementation of the timestepping loop in MPI essentially amounts to implementing a custom user-level scheduler. Implementing a custom scheduler like this directly in application code greatly limits the composability and flexibility and substantially increases the burden on the application developer.



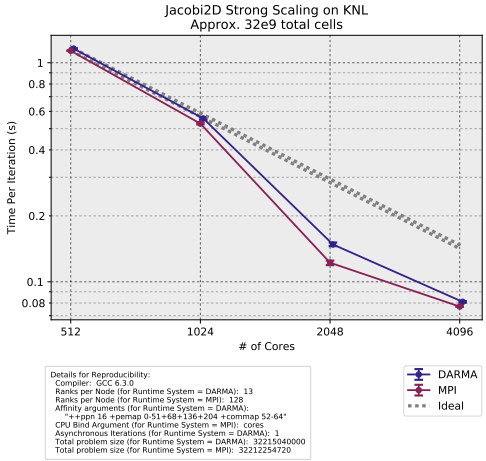
(a) 32B total cells



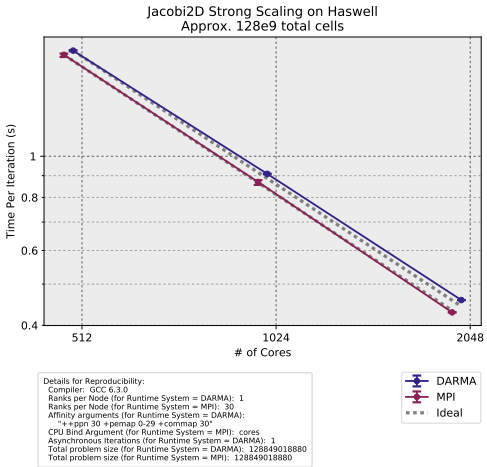
(b) 16B total cells



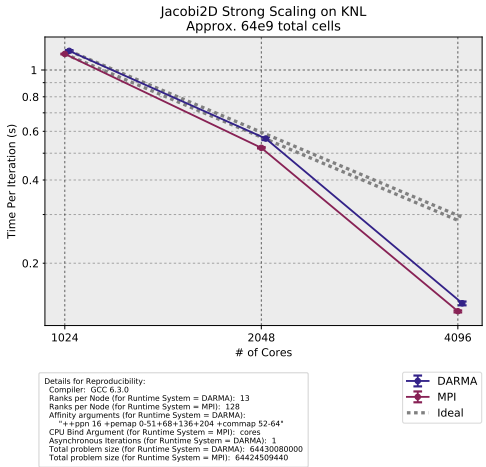
(c) 64B total cells



(d) 32B total cells



(e) 128B total cells



(f) 64B total cells

Figure 5.5: Strong scaling performance of Jacobi benchmark for MPI and DARMA-CHARM++ for increasing total cells on Mutrino for Haswell (2048 cores): (a) 32B , (c) 64B, and (e) 128B and KNL (4096 cores): (b) 16B, (d) 32B, and (f) 64B. Note scales do not begin at 0. Error bars show min and max values observed from multiple trials. All-Reduce convergence checks are performed each iteration.

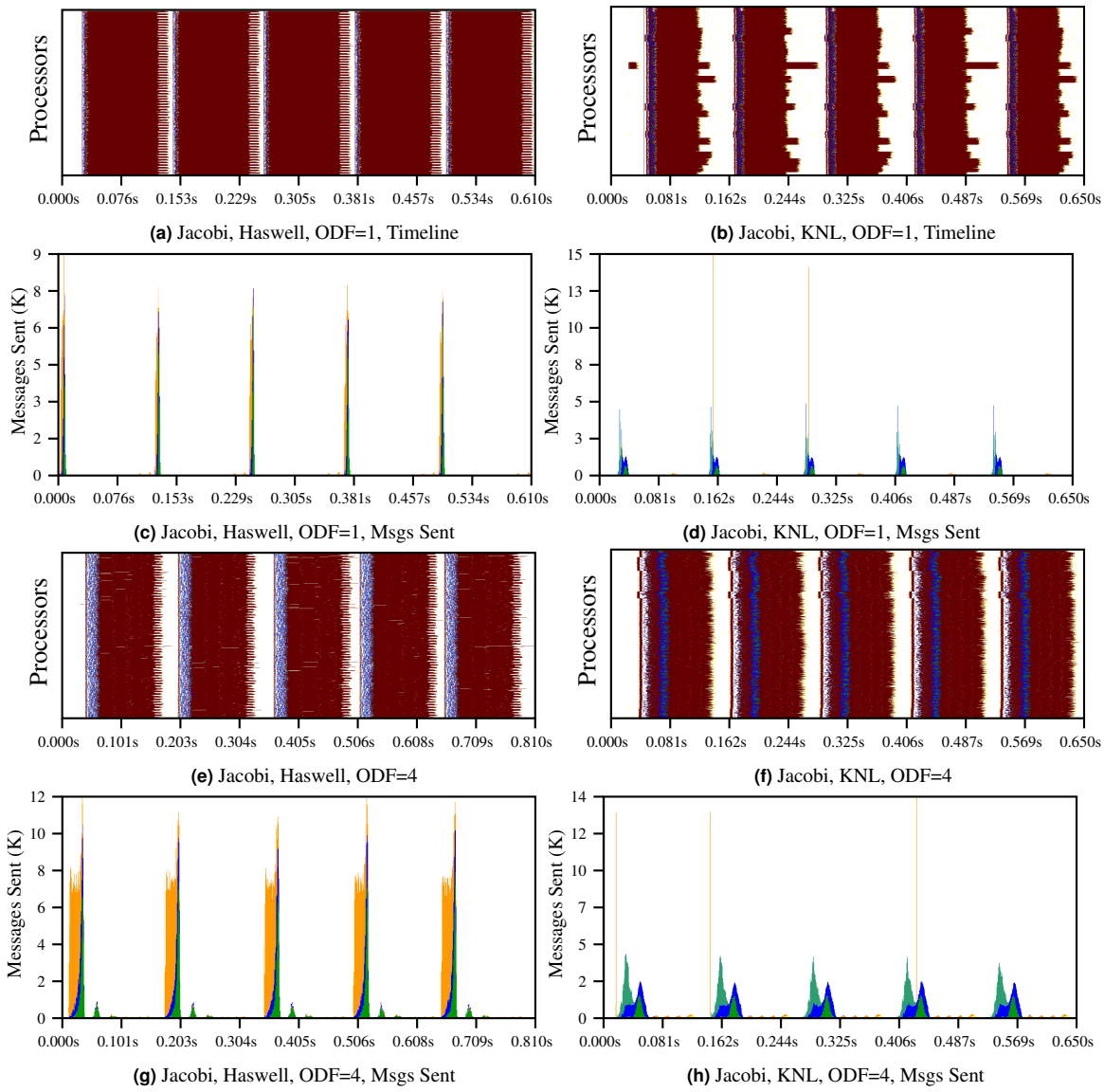


Figure 5.6: Timeline views and communication histograms for a sample of threads over time of the Jacobi benchmark (in DARMA) on 64 nodes of Mutrino (Haswell and KNL) for varying levels of overdecomposition. In the timeline views, task execution is indicated in maroon while idle time is shown in white. Computation stalls at regular intervals. The timeline clearly shows a gap where task execution stalls which corresponds to the residual `allreduce` . By increasing overdecomposition, messages sent are more spread out over time, causing better network utilization, effective communication/computation overlap, but the cost of dependency analysis (in orange) slightly increases.

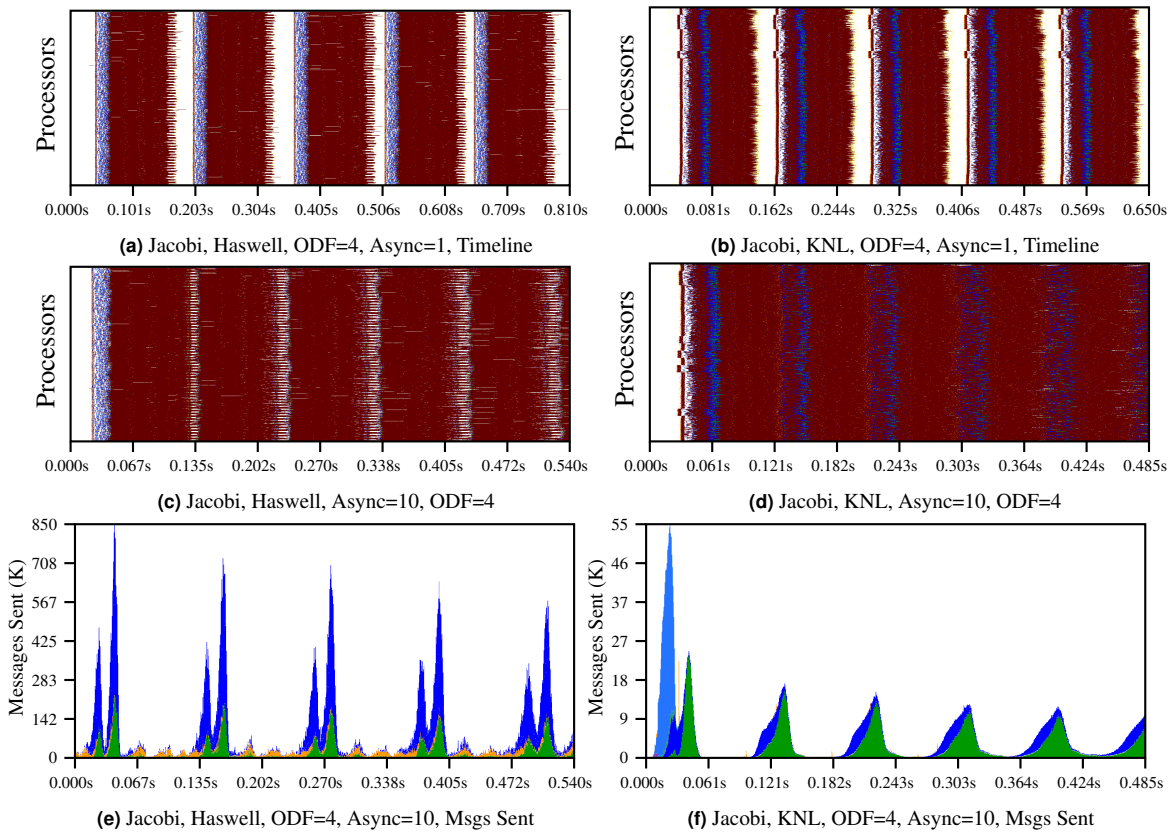


Figure 5.7: Timeline views and communication histograms for a sample of threads over time of the Jacobi benchmark (in DARMA) on 64 nodes of Mutrino (Haswell and KNL) for varying levels of asynchrony in the benchmark, achieved by incorporating speculative execution (async=1 or async=10). Computation is shown in maroon while idle time is in white. The overdecomposition factor (ODF) is fixed at 4. Asynchrony has a dramatic effect on the amount of idle time in the Jacobi benchmark. By reducing the hard, global synchronization each iteration the idle time is reduced and the performance increases (compared to the previous best performance: async=10 is 13% faster on Haswell and 34% faster on KNL).

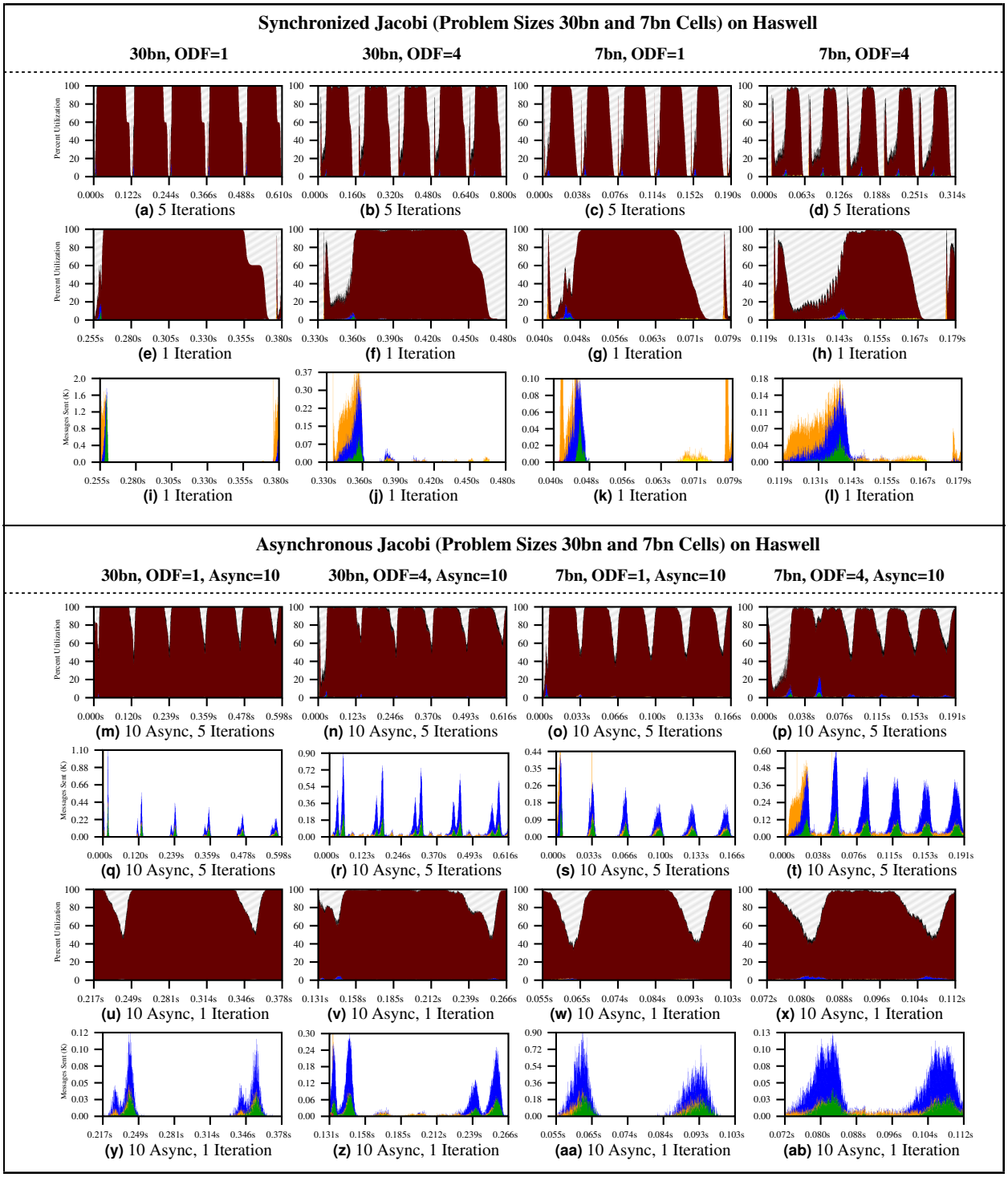
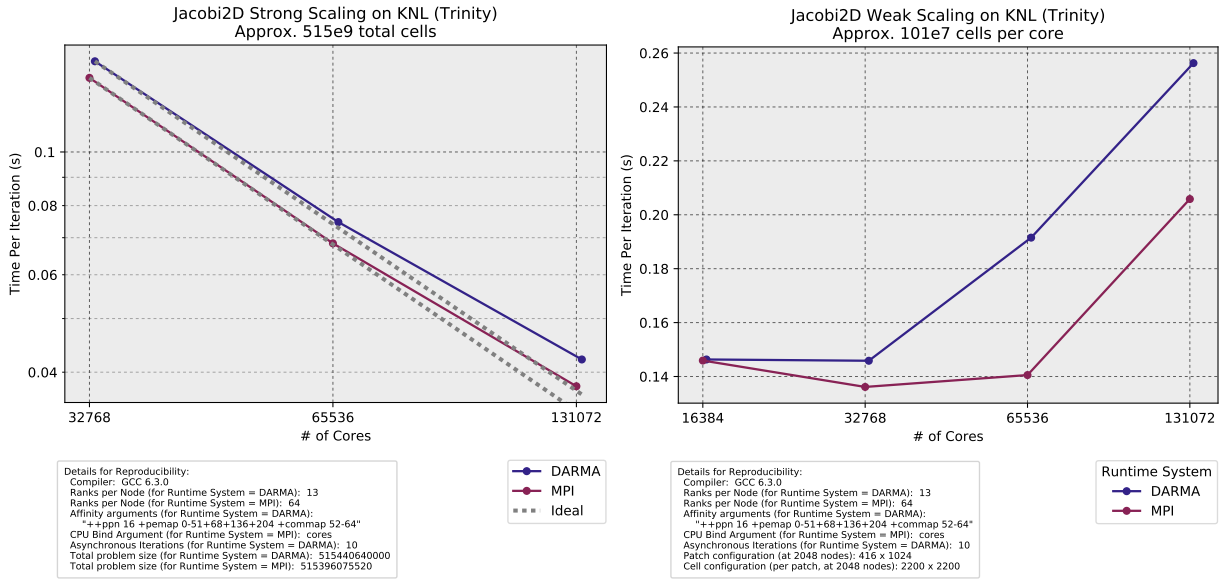


Figure 5.8: Processor utilization and communication histograms for a sample of threads over time of the Jacobi benchmark (in DARMA) on 64 nodes of Mutrino on Haswell. Different columns explore varying levels of overdecomposition for two problems sizes (7bn total cells and 30bn total cells). Different rows explore varying levels of asynchrony exposed in the benchmark. Computation is shown in maroon while idle time is in white.



(a) Strong Scaling

(b) Weak Scaling

Figure 5.9: Strong and Weak scaling performance of Jacobi benchmark for MPI and DARMA-CHARM++. Scaling is shown up to 2048 nodes on Trinity for KNL (139K cores). Error bars show min and max values observed from multiple trials. The overdecomposition factor is set to 4 for DARMA.

```
void post_neighbor_sends(   int const t) {
  double* const buf = &data[fst][0];
  for_each_boundary(index, [&](Boundary b,       int bound_size) {
    auto const off = boundary_dir[b];
    auto const tag = compute_tag(index, b, t);
    auto const rpe = block_to_pe({index.x - off.first, index.y - off.second});
    switch (b) {
    case Bottom:
      MPI_Isend(buf + INDEX(size_bound.x-2,0), bound_size, MPI_DOUBLE, rpe, tag, MPI_COMM_WORLD, &req[cur_req++]);
      break;
    case Top:
      MPI_Isend(buf + INDEX(1,0), bound_size, MPI_DOUBLE, rpe, tag, MPI_COMM_WORLD, &req[cur_req++]);
      break;
    case Right: {
      auto right_ptr = right.get();
      for (auto i = 0; i < bound_size; i++)
        right_ptr[i] = buf[INDEX(i,size_bound.y-2)];
      MPI_Isend(right_ptr, bound_size, MPI_DOUBLE, rpe, tag, MPI_COMM_WORLD, &req[cur_req++]);
      break;
    }
    case Left: {
      auto left_ptr = left.get();
      for (auto i = 0; i < bound_size; i++)
        left_ptr[i] = buf[INDEX(i,1)];
      MPI_Isend(left_ptr, bound_size, MPI_DOUBLE, rpe, tag, MPI_COMM_WORLD, &req[cur_req++]);
      break;
      }
    }
  });
}
```

(a) MPI Jacobi2D code to send boundaries in overdecomposed implementation.

```
  //pack  and  publish  boundaries
  for_each_boundary(
    index, size_x, size_y,
    [&](Boundary b,   int bound_size,   int bound_off) {
      Index3D boundary_idx = index.increase_dim(bound_off, r_bound);
      auto boundary = boundariesAHC[boundary_idx].local_access();
      create_work<PackBoundary>(index, prev, boundary, b, size_x, size_y, bound_size);
      boundary.publish(version=iter, n_readers=1);
    }
  );

// ...

void PackBoundary::  operator()(
  Index2D index, ReadAH_VecDouble prev, AH_VecDouble boundary, Boundary b,           int size_x,  int size_y,  int bound_size
) {
  const double* prev_buf = prev->data();
  double* bound_buf = boundary->data();
  switch (b) {
  case Top:
    std::memcpy(bound_buf, prev_buf + INDEX(1,0), bound_size*        sizeof(double));
    break;
  case Bottom:
    std::memcpy(bound_buf, prev_buf + INDEX(size_x-2,0), bound_size*         sizeof(double));
    break;
  case Left:
    for (auto i = 0; i < bound_size; i++)
      bound_buf[i] = prev_buf[INDEX(i,1)];
    break;
  case Right:
    for (auto i = 0; i < bound_size; i++)
      bound_buf[i] = prev_buf[INDEX(i,size_y-2)];
    break;
  }
}
```

(b) DARMA Jacobi2D code to publish boundary data.

Figure 5.10: MPI and DARMA Jacobi2D code to pack and send boundaries in overdecomposed implementations.

```
do {
  for (auto&& b : active_blocks) {
    // first post all the receives and sends for this timestep
    b.post_neighbor_receives(timestep);
    b.post_neighbor_sends(timestep);
  }

  double max_residual = -1e12;
  do {
    // loop through all active blocks assigned to this rank and test if they are ready to execute the kernel
    for (auto it=active_blocks.begin(), it_end=active_blocks.end(); it!=it_end; ++it) {
      auto& b = *it;
      bool const ready = b.test_ready();
      if (ready) {
        b.reset_ready();
        b.copy_recv_buffers_in();
        double local_max_residual = b.do_kernel();
        max_residual =   std::max(local_max_residual, max_residual);
        b.swap_arrays();
        auto cur = it;
        inactive_blocks.splice(inactive_blocks.begin(), active_blocks, cur, ++it);
        break;
      }
    }
  } while (active_blocks.size() != 0);

  // check for global convergence
  double global_max_residual = 0;
  MPI_Allreduce(&max_residual, &global_max_residual, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);

  std::swap(active_blocks, inactive_blocks);
  converged = global_max_residual < error_threshold;
  timestep++;
} while (timestep < max_iter and    not converged);
```

(a) MPI Jacobi2D outer timestepping loop with overdecomposition.

```
auto max_residual = initial_access<    double>();
auto iter = initial_access<   int>();

create_work([=]{
  max_residual.set_value(error_threshold + 1.0);        // set to large initial value
  iter.set_value(0);
});

create_work_while([=]{
  return max_residual.get_value() > error_threshold && iter.get_value() < max_iter;
}).do_([=]{
  int next_iter = *iter;
  //do a number of substeps without checking residual
  for (int substep=0; substep < async_factor; ++substep, ++next_iter){
    auto& prev = next_iter % 2 == 0 ? jacobi_v1 : jacobi_v2;
    auto& next = next_iter % 2 == 0 ? jacobi_v2 : jacobi_v1;
    create_concurrent_work<JacobiStep>(
      size_x, size_y, next_iter, prev, next, boundaries.mapped_with(dim_reduce), residuals, index_range=range_2d
    );
  }
  residuals.reduce<Max>(output=max_residual);

  // update the iteration number for next round
  *iter = next_iter;
});
```

(b) DARMA Jacobi2D outer timestepping loop with overdecomposition and substeps (asynchronous iterations).

Figure 5.11: Comparison of the DARMA and MPI Jacobi2D code for the timestepping loop. Both implementations include overdecomposition and a global convergence check via a `reduce` operation, The DARMA implementation includes asynchronous iterations (lines 14–20), but the MPI code is synchronized every iteration.

5.4.2 Molecular Dynamics Benchmark

The molecular dynamics benchmark complements the Jacobi benchmark by providing a compute-bound kernel with more bandwidth-intensive communication.

Molecular dynamics codes have the following stages:

- Compute "self" forces between atoms within each box
- Exchange particles in neighboring cells (communication)
- Compute forces acting on atoms between neighbor boxes
- Accelerate particles and update atom positions
- Migrate particles that move outside their original cell (communication)

We have implemented a basic molecular dynamics code that computes pairwise forces using a Lennard-Jones potential (LJP), similar to that seen in MiniMD.

While the molecular dynamics benchmark shares some similarities with Jacobi (nearest neighbor communication), all atoms in neighboring cells are exchanged, not just a halo region. Thus communication is more bandwidth-intensive than in Jacobi. Similar to Jacobi, space is decomposed into cells or boxes on a grid of size $n \times m$. There is no atom grid, but as box sizes are decreased the number of atoms within a box decreases. For every N atoms within a box there are N^2 particle forces computed, leading to a much greater computational intensity than Jacobi. Particle migration between boxes can produce some load imbalance, but this tends to be modest.

The MPI version of the molecular dynamics benchmark attempts to mimic the asynchronous communication within the DARMA benchmark. All sends and receives are posted asynchronously. The entire list of MPI `Requests` is then passed to `MPI_Waitany` to check for any completed request. The corresponding computation pending on that request is then performed. Such a communication pattern is difficult to implement efficiently within MPI for larger problems and is error prone. At large scale, this manifested as a race condition bug for the large-scale MPI runs, leading to an incomplete set of MPI results in Figures 5.16 and 5.15.

5.4.2.1 Mutrino: 64 nodes

Figures 5.12 show scaling results on Mutrino for the molecular dynamics benchmark on both the Haswell and KNL partitons. In most cases, DARMA scales as well as or better than MPI for this benchmark. In several cases (mostly those with higher computational intensity), DARMA not only scales better than MPI, but also actually *performs* better than MPI at the largest scale. More profiling is needed to assess the leveling off of MPI strong scaling performance. A likely contributor is less efficient asynchronous communication. With overdecomposition, several hundred or even thousand halo boxes must be sent by a single MPI rank. This severely stresses MPI functions like `MPI_Waitany` and `MPI_Testsome`, which must traverse the entire list of requests to search for a completed communication. Polling with `MPI_Probe` may have provided a more scalable implementation, but this was not explored. The goal was to write a simple molecular dynamics MPI code of "equivalent complexity" as the analogous DARMA code.

5.4.2.2 Detailed Performance Analysis for DARMA

Figures 5.13 and 5.14 show detailed timelines for the execution of the DARMA molecular dynamics benchmark. The figures clearly demonstrate the tradeoffs of task granularity, with Figure 5.13 having larger tasks (400 particles per box) and Figure 5.14 having smaller tasks (100 particles per box). Large tasks better amortize runtime overheads, but decrease communication pipelining and overlap with computation. For 400 particles per box, very little idle time is observed except in bursty communication phases. Most messages are sent in a very narrow time window followed by nearly 100% processor utilization for computation. For 100 particles per box, there are still evident bursts of communication, but they are much more distributed in time. Processor utilization dips for longer periods, though, as the runtime overhead of the smaller tasks offsets the gains from pipelined communication.

5.4.2.3 Trinity: 2048 nodes

Figures 5.15 and 5.16 show scaling results on Trinity up to 2K nodes. These results are consistent with the Mutrino results observed above. DARMA scales well (sublinearly, in fact) while MPI degrades significantly at the largest scale. As discussed above, the likely contributor is inefficient asynchronous communication. On the strong-scaling curve in particular, surface area to volume ratios increase as total problem size shrinks per physical node. A much larger percentage of box interactions are inter-node rather than intra-node, leading to MPI request lists for hundreds of box communications. Although further testing is required, the most likely explanation seems that DARMA is able to scalably handle hundreds or even thousands of box communications. In contrast, the MPI algorithm must be optimized to avoid these long request lists.

5.4.2.4 Productivity and Semantic Information Gain

DARMA provides a data-centric decomposition of the molecular dynamics problem rather than an processor- or execution-centric version. MPI codes wishing to express the algorithm in terms of well-defined cells with multiple cells per MPI rank must express a two-level algorithm: intra-node neighbors cells and inter-node neighbor cells. When trying to combine with asynchronous communication and asynchronous dispatch (run tasks as soon as ready), an enormous amount of atom and cell metadata must be managed explicitly in the MPI application. In contrast, the DARMA algorithm is very succinctly expressed as a single-level algorithm directly in terms of boxes or cells. No complexity due to inter- and intra-node interactions is apparent in the application. Data-flow occurs naturally without using non-deterministic MPI functions like `MPI_Waitany` .

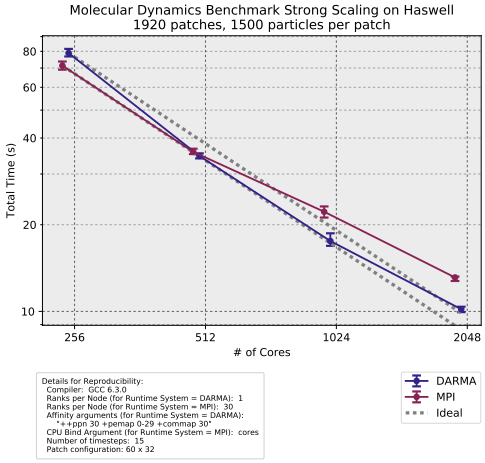
Code	SLOCCount (lines of code)*
MPI	765
DARMA	621

* From David A. Wheeler tool that counts lines of code.

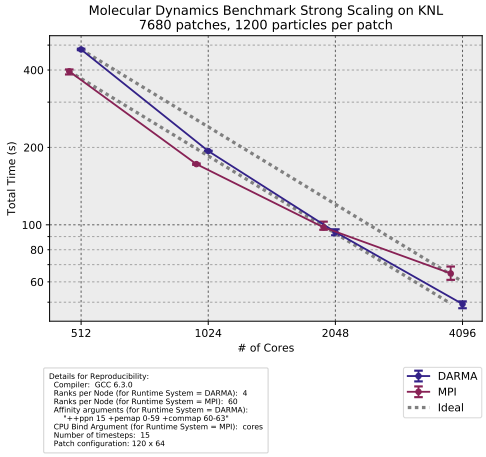
Table 5.2: Lines of Code for Molecular Dynamics Benchmark in MPI vs. DARMA.

While Table 5.2 shows lines of code, the MPI code is far more error-prone and complex. Thus the discrepancy does not fully illustrate the extra complexity of the MPI version. Figure 5.17 shows one portion of the MPI code for performing neighbor cell exchanges: the receive progress code. First, the number of particles

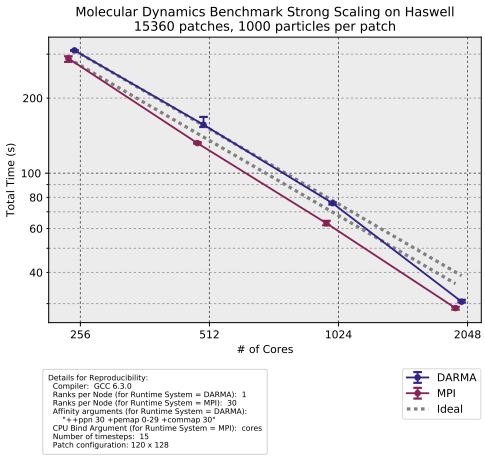
must be sent as this can change from iteration to iteration. Once the number of particles is received, a receive can be posted for the correct number of particles. Boxes coming from the same rank should be distinguished by tag. Computation cannot occur on a box until all sends are completed (not shown). In DARMA, however, a simple loop over neighbors creates a task for each interaction. All data movement occurs automatically. Interactions are computed once all sends and receives involving a box are completed.



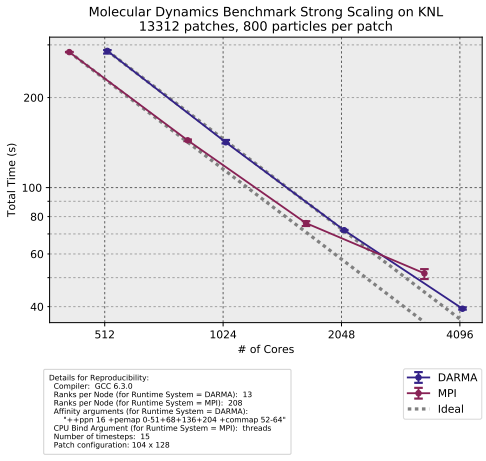
(a) 1.92K patches, 2.88M particles total



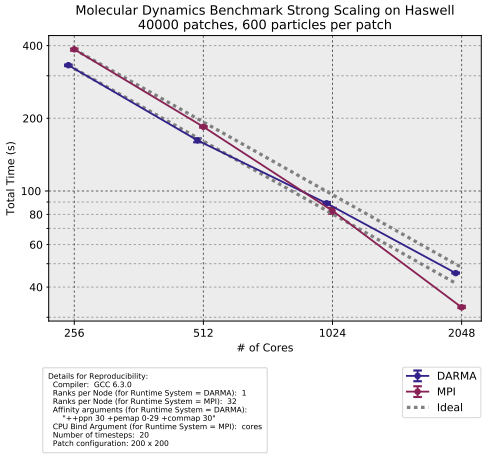
(b) 7.68K patches, 9.22M particles total



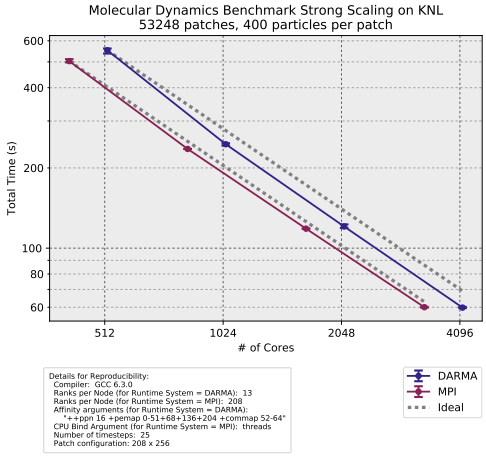
(c) 15.36K patches, 15.36M particles total



(d) 13.31K patches, 10.65M particles total

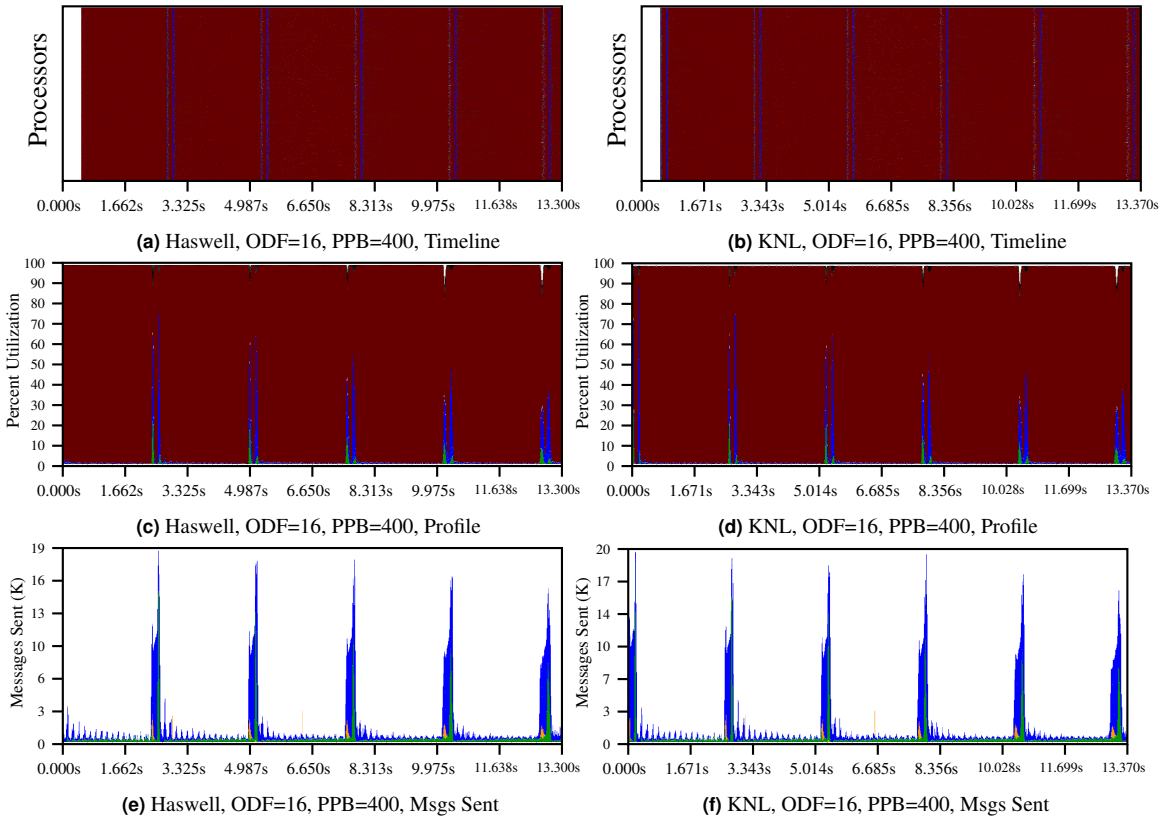


(e) 40K patches, 24M particles total



(f) 53.24K patches, 21.3M particles total

Figure 5.12: Strong scaling performance of molecular dynamics benchmark for MPI and DARMA-CHARM++ for varying problem sizes. Scaling is shown up to 64 nodes in the left column for Mutrino for Haswell (1920 cores for MPI, 1984 cores for DARMA) and in the right column for KNL (3840 cores for MPI, 4096 cores for DARMA). Error bars show min and max values observed from multiple trials.



(a) Haswell, ODF=16, PPB=400, Timeline

(b) KNL, ODF=16, PPB=400, Timeline

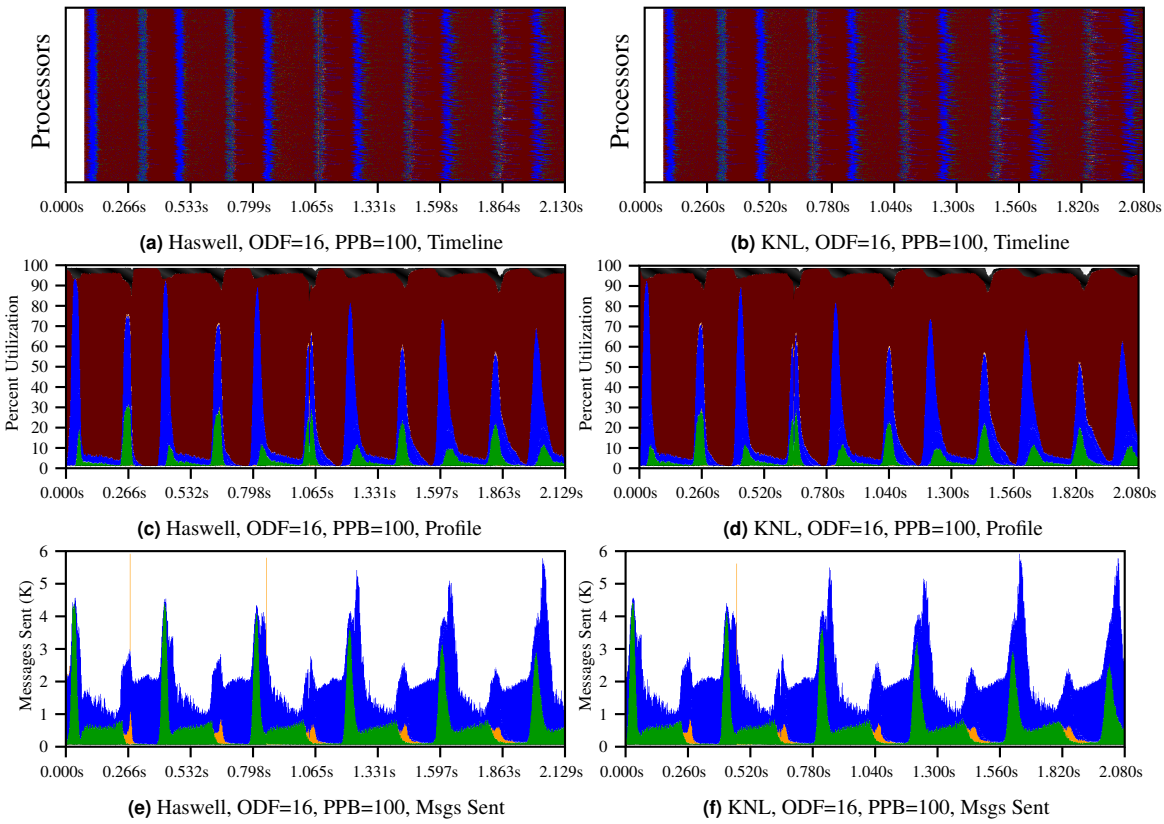
(c) Haswell, ODF=16, PPB=400, Profile

(d) KNL, ODF=16, PPB=400, Profile

(e) Haswell, ODF=16, PPB=400, Msgs Sent

(f) KNL, ODF=16, PPB=400, Msgs Sent

Figure 5.13: Timelines showing activity of the molecular dynamics benchmark with 400 particles per box on Haswell and KNL partitions of Mutrino. (a) and (b) show compute/system/idle activity over time on a per-processor basis. Maroon indicates task computation while blue indicates idle time or system work. (c) and (d) show percent utilization of the processors over time, with periods of intense computation (all maroon) interspersed with idle and communication activity. (e) and (f) show messages sent in certain time windows, demonstrating communication activity.



(a) Haswell, ODF=16, PPB=100, Timeline

(b) KNL, ODF=16, PPB=100, Timeline

(c) Haswell, ODF=16, PPB=100, Profile

(d) KNL, ODF=16, PPB=100, Profile

(e) Haswell, ODF=16, PPB=100, Msgs Sent

(f) KNL, ODF=16, PPB=100, Msgs Sent

Figure 5.14: Timelines showing activity of the molecular dynamics benchmark with 100 particles per box on Haswell and KNL partitions of Mutrino. (a) and (b) show compute/system/idle activity over time on a per-processor basis. Red indicates task computation while blue indicates idle time or system work. (c) and (d) show percent utilization of the processors over time, with periods of intense computation (all maroon) interspersed with idle and communication activity. (e) and (f) show messages sent in certain time windows, demonstrating communication activity.

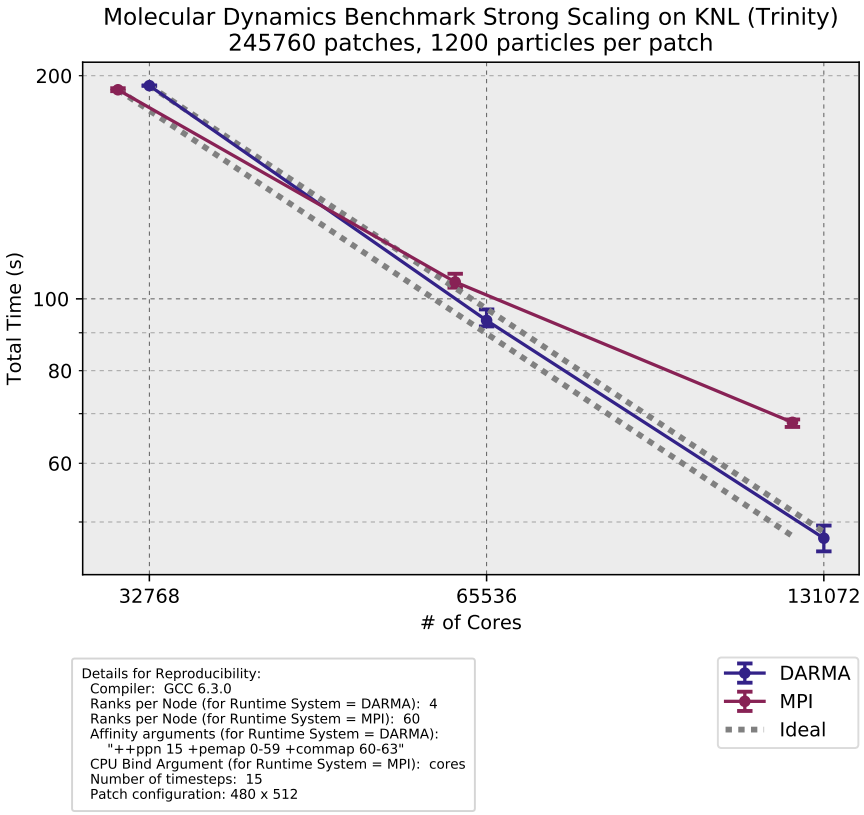
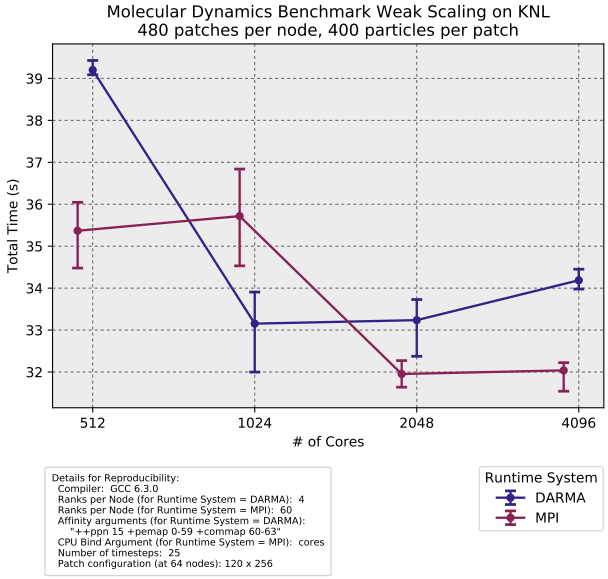
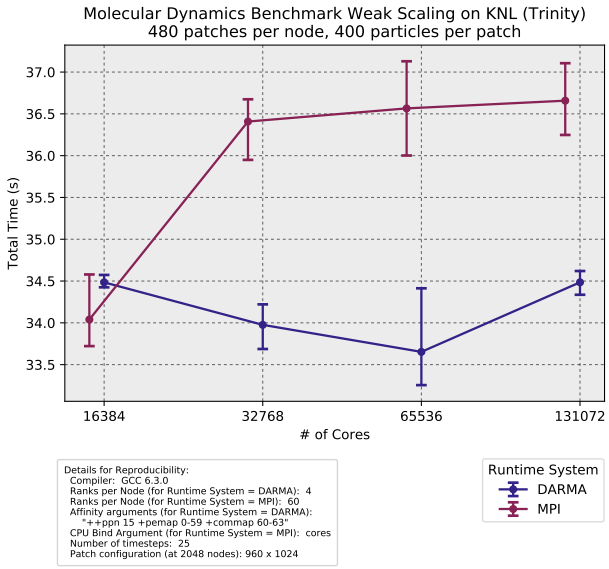


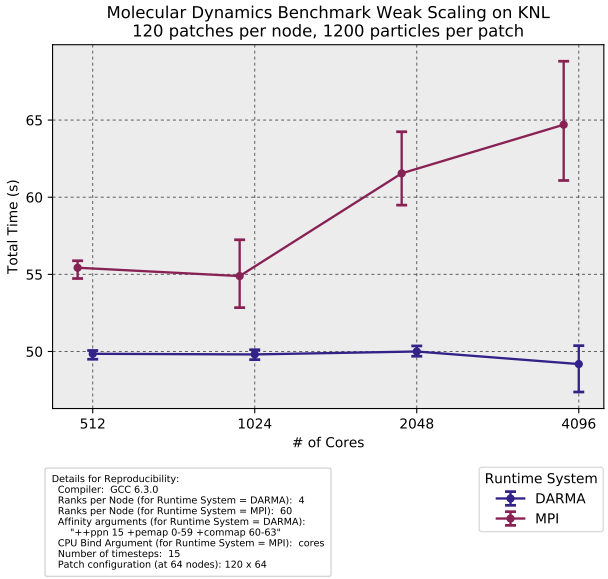
Figure 5.15: Strong scaling performance of molecular dynamics benchmark for MPI and DARMA-CHARM++. Scaling is shown up to 2048 nodes on Trinity for KNL (139K cores). Error bars show min and max values observed from multiple trials.



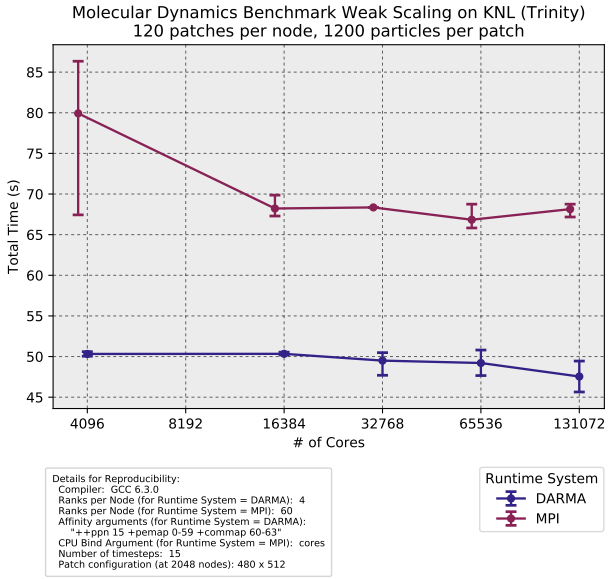
(a) 400 Particles/Patch Mutrino



(b) 400 Particles/Patch Trinity



(c) 1200 Particles/Patch Mutrino



(d) 1200 Particles/Patch Trinity

Figure 5.16: Weak scaling performance of molecular dynamics benchmark for MPI and DARMA-CHARM++. Scaling is shown up to 64 nodes (2K cores) on Mutrino KNL (left column) and up to 2048 nodes (139K cores) on Trinity KNL (right column) for 400 particles per patch (first row) and 1200 particles per patch (second row). Error bars show min and max values observed from multiple trials. Note the y axes are not aligned in these plots.

```
int neighborProgressLoop(config& cfg,      std::vector<ParticleBox>& myBoxes,
  BoxComm& comm,   std::map<int, std::list< int>>& pendingComputes)
{
  int index_done;
  int num_requests = comm.numRecvs;
  MPI_Status stat;
  MPI_Waitany(num_requests, comm.reqs, &index_done, &stat);
  int num_ops_done = 0;
  ActiveSend& send = comm.sends[index_done];

  if (send.isMeta){
    if (send.numParts > 0){
      int source = stat.MPI_SOURCE;
      int rel_box_id = index_done - comm.numSends;
      ParticleBox& inBox = comm.boxes[rel_box_id];
      inBox.localParticles.resize(send.numParts);
      MPI_Irecv(&inBox.localParticles[0], send.numParts*        sizeof(Particle),
          MPI_BYTE, source, send.tag, MPI_COMM_WORLD, &comm.reqs[index_done]);
    } else {
      ++num_ops_done;    //nothing  to  do
    }
  } else {
    int glbl_box_id = send.boxID;
    int comm_box_id = index_done - comm.numSends;
    ParticleBox& inBox = comm.boxes[comm_box_id];
    inBox.id = glbl_box_id;
    inBox.x = glbl_box_id % cfg.num_box_x;
    inBox.y = glbl_box_id / cfg.num_box_y;
    for (int local_box_id : comm.incoming[glbl_box_id]){
      ParticleBox& pBox = myBoxes[local_box_id];
      if (pBox.joinCounter == 0){
        neighborInteract(pBox, inBox);
      } else {
        pendingComputes[local_box_id].push_back(comm_box_id);
      }
    }
    ++num_ops_done;
  }
  return num_ops_done;
}
```

(a) MPI Molecular Dynamics Communication Loop

```
void ParticleTimestep::  operator()(Index2D my_index, AHC_VecParticle2D particles_col,
    AHC_VecIndex2D neighbors_lst, AHC_VecParticle3D migrate_parts,          int iter)
{
  //make  my  local  particles  available  to  all  other  boxes
  auto local_particles = particles_col[my_index].local_access();
  local_particles.publish(version=iter, n_readers=NUM_NEIGHBORS);

  const auto& neighbors = *neighbors_lst[my_index].local_access();
  //loop  geometry  info  and  create  tasks  to  interact  with  each  neighbor
  for (auto&& n : neighbors) {
    auto neighbor_particles = particles_col[n].read_access(version=iter);
    create_work<NeighborInteract>(local_particles, neighbor_particles);
  }
```

(b) DARMA Molecular Dynamics Task Setup

Figure 5.17: Molecular dynamics setup code performing particle exchange prior to computing neighbor interactions

5.4.3 Simulated Load-Imbalance Benchmark

In order to study the interplay of load balancing overheads and quality of load balancing, we construct a simulated benchmark with minimal communication. This isolates the load balancer itself as the main concern. The benchmark generates an adversarial imbalanced work distribution. The work distribution, however, is constructed based on a linear imbalance function with a known optimal solution. The benchmark can therefore run in 3 modes:

1. A perfectly balanced mode which *a priori* generates the known optimal distribution
2. An adversarial imbalanced initial distribution with no load balancing
3. An adversarial imbalanced initial distribution with load balancing after the first few iterations

Cases (1) and (2) provide "best-case" and "worst-case" baselines for comparison. Case (3) should ideally match case (1) with minimal load-balancing overheads. As we will demonstrate below, different load balancers present a trade-off between quality (improving the work distribution) and overhead (cost to run and redistribute work).

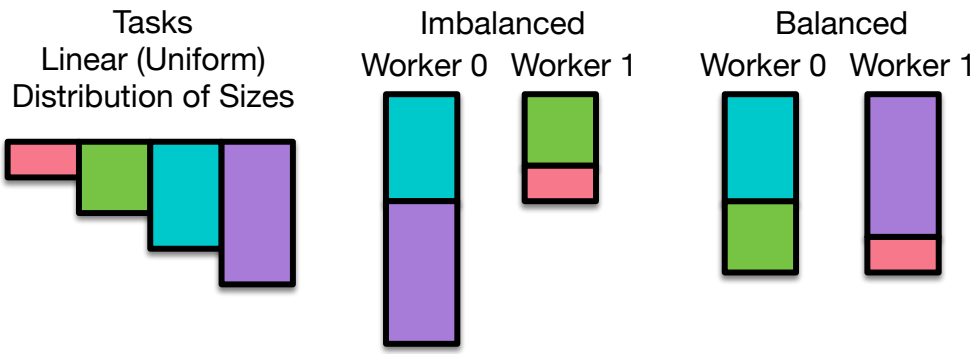


Figure 5.18: High-level description of the synthetic imbalance benchmark. Task sizes span a uniform distribution. The optimal balance is achieved by pairing a small task with a large task on the same node.

Figure 5.18 illustrates at a high-level the distribution of tasks and how the optimal solution can be constructed. The optimal solution is not known to the DARMA load balancers and only application-agnostic load balancers are tested. Generating the optimal distribution requires at least a 2x overdecomposition (2 tasks per process). By construction, both MPI and DARMA versions generate exactly the same tasks with the same overdecomposition. DARMA is able to generate dynamic distributions while MPI generates a static task distribution. The MPI code cannot load balance and therefore is only able to run cases (1) and (2). However, it provides another baseline for best- and worst-case performance without scheduling and load-balancing overheads.

5.4.3.1 Mutrino: 64 nodes

Figure 5.19 shows strong scaling results on Mutrino up to 64 nodes for Haswell and KNL for a different problem sizes and load balancing strategies.

CHARM++ provides a variety of load balancing strategies as part of its runtime system and the application developer can leverage these with very little effort, as described in Section 5.4.3.4. The CHARM++ load balancers tested within these scaling studies are listed in Figure 5.20 along with their associated high-level description.

Load balancers occur in roughly two groups: centralized balancers that are expensive, but high-quality (Greedy,Refine) and scalable balancers that are less expensive on larger runs, but provide more approximate, heuristic load balancing (Distributed, Hierarchical). Hybrid tries to combine these strategies by performing centralized balancing within small groups of processes.

Refine and Greedy clearly provide the best performance. At 64 nodes and below, load balancing overheads are small and the quality of the solution is more critical than scalability of the balancing algorithm. However, even at 64 nodes, the Refine balancer scaling levels off. In Section 5.4.3.3 showing Trinity results, both Refine and Greedy become far too expensive. Hybrid provides almost equivalent results to Refine, but this is to be expected since the entire system is a "small group," making Hybrid and Refine essentially the same algorithm. However, while Refine performance tapers off at 64 nodes, Hybrid maintains linear strong scaling throughout.

In these plots, we find that all load balancers are relatively scalable up to 64 nodes, however with different quality solutions. Only the Greedy and Hybrid load balancers are truly competitive with the optimally-balanced baseline. However, all load balancers perform better than the worst-case baseline with no load-balancing.

5.4.3.2 Detailed Performance Analysis for DARMA

Figure 5.21 examines the execution over time of various load balancing strategies. Without load balancing, iterations have a "jagged edge" with several threads idling waiting for other threads to finish. The hierarchical and distributed load balancers have the same pattern in the first iteration. Although they reduce idle time in future iterations, significant idle time remains. The hybrid and greedy load balancers, however, generate a nearly optimal solution, with almost no significant idle time visible after the first iteration.

As stated above, overdecomposition gives flexibility to the load balancer to improve overall. In this synthetic example, an ideal load balance is achieved by placing one heavyweight and one lightweight task on each core (Figure 5.19). The load balancers underneath DARMA, however, are application-agnostic. They have no knowledge of this pattern and thus make a best guess based on profiling during a warm-up phase. Even though an ODF of 2 is theoretically sufficient to obtain ideal load balance, more overdecomposition may still help the application-agnostic load balancer in DARMA. Increased overdecomposition flexibility comes at the cost of increased scheduling and load-balancing overheads.

These effects can be seen in Figure 5.22 through a weak scaling study. For some load balancers, weak scaling runs tend to show an uptick towards higher node counts. Interestingly, though, overdecomposition factors up to 16 for Haswell or 64 for KNL actually show steady or even improved performance. Thus, at least for the benchmark here, very large overdecomposition factors derive more benefit from load balancing flexibility than they trade off in load balancing overhead.

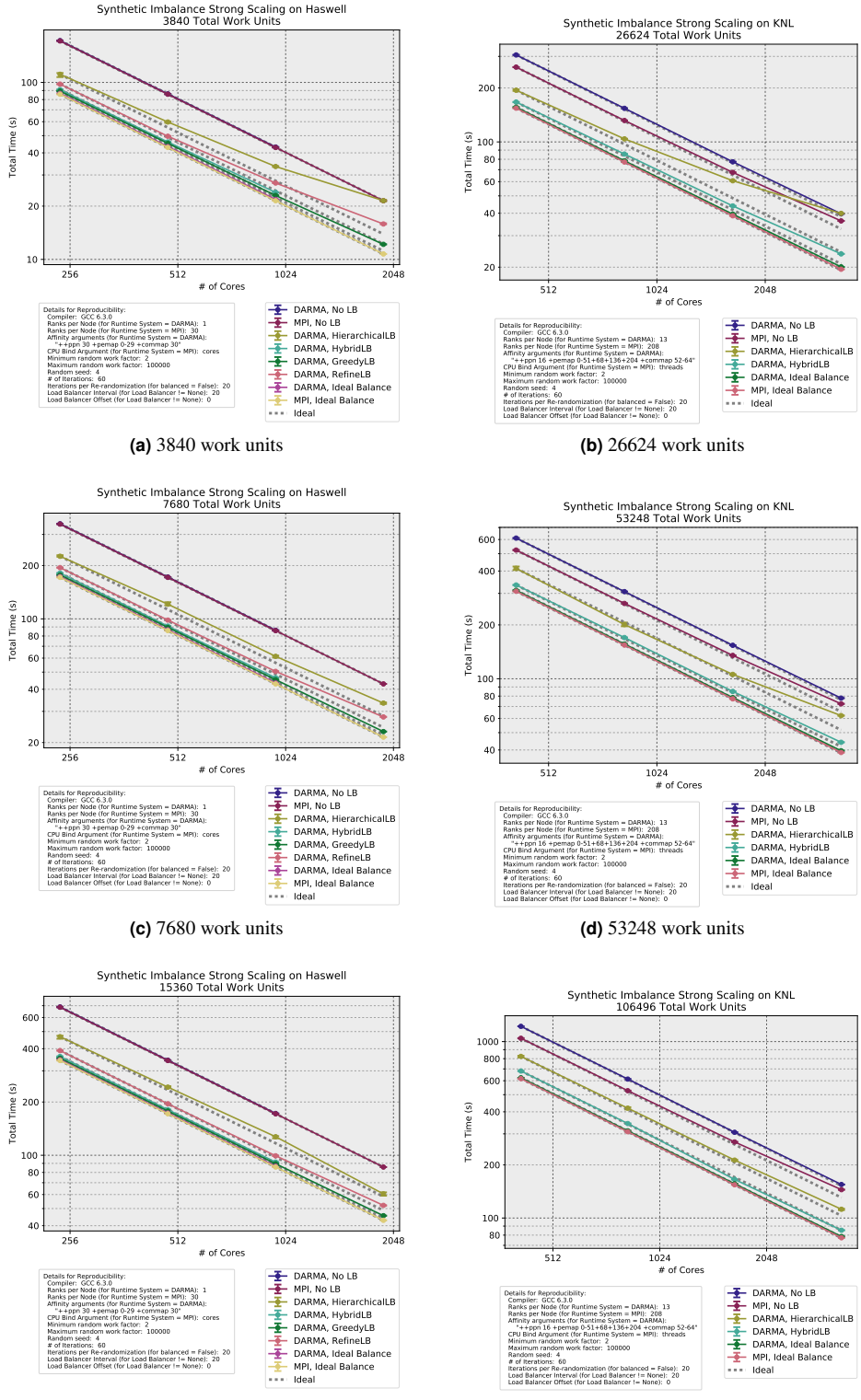
5.4.3.3 Trinity: 2048 nodes

Figure 5.23 show strong scaling results for Trinity up to 2K on the simulated imbalance benchmark. Load-balancing overheads are now more critical as scale increases and centralized load-balancers in-particular are no longer suitable. The Refine load balancer is no longer even included in the plots since the times are an order of magnitude larger than the other load balancers. At lower scales, an expensive but high-quality load balancer is favored since load-balancing is still relatively inexpensive. Generating the optimal solution is more important. At larger scales, a scalable load-balancer that generates sub-optimal load balance becomes favorable since any benefit from improved load-balance is easily canceled by higher load-balancing overheads.

The Hybrid load balancer achieves the best performance, but scaling levels off at the largest scales. Pushing to even larger scales is needed to understand whether the scaling continues, or, like the Refine balancer, load balancing overheads will continue to grow and overcome the benefits of an improved work distribution. The Distributed and Hierarchical balancers are expected to be more scalable, but their heuristics generate a non-optimal distribution. However, pushed to even larger scales and extrapolating Figure 5.23, these load balancers may actually become the best performers for 10K or 100K endpoints. Future work is therefore likely to focus on either improving the heuristics used in the Distributed and Hierarchical balancers or alleviating the scaling bottlenecks in Hybrid.

5.4.3.4 Productivity and Semantic Information Gain

Load balancing is not apparent in the DARMA user code other than a single keyword hint indicating to the runtime when load balancing should occur. The DARMA runtime handles 1) profiling and timing individual tasks to gather statistics for load balancing and 2) migrating data and tasks in response to load balancing decisions. `AccessHandle` abstracts the exact physical location of data and provides serialization hooks that allow the runtime to move data freely around the system. DARMA algorithms therefore require no code changes to support load balancing. This critically separates performance concerns (load balancing) from correctness concerns (the algorithm). While application-agnostic load balancers may lack application-specific information, agnostic load balancers can actually improve performance since we are reusing heavily optimized load-balancers rather than writing an *ad hoc* balancer for each application. It can further improve stability, implementing task migration once in a stable runtime core rather than re-engineering error-prone migration code for each application.



(a) 3840 work units

(b) 26624 work units

(c) 7680 work units

(d) 53248 work units

(e) 15360 work units

(f) 106496 work units

Figure 5.19: Strong scaling performance of simulated imbalance benchmark for MPI and DARMA-CHARM++ for increasing total work: (a) 3840 work units, (c) 7680 work units, and (e) 15360 work units up to 64 nodes on Mutrino for Haswell (2048 cores) and (b) 26624 work units, (d) 53248 work units, and (f) 106496 work units on KNL (4352 cores). Error bars show min and max values observed from multiple trials. Several different load balancing (LB) algorithms are shown.

LB Type	LB Name	Description	Benefits	Drawbacks
Centralized	**GreedyLB**	Heap-based, considers all tasks for redistribution	Provides high quality distribution	Not scalable, expensive in memory and space
Centralized	**RefineLB**	Heap-based, considers only tasks above threshold	Fast for centralized load balancer	Not scalable, quality might be low
Distributed, gossip-based	**DistributedLB**	Gossip-based, probabilistic transfer	Extremely fast, fully decentralized	Quality may be low
Distributed, tree-based	**HierarchicalLB**	Tree-based, hierarchical transfer	Fast, typically provides high quality	Greedy algorithm may not be aggressive
Distributed, group-based	**HybridLB**	Creates subgroups of processors and applies centralized	Can reuse centralized LB schemes	May be expensive and slow with large groups

Figure 5.20: List and descriptions of load balancers considered in the simulated imbalanced benchmark and particle-in-cell (PIC) proxy application.

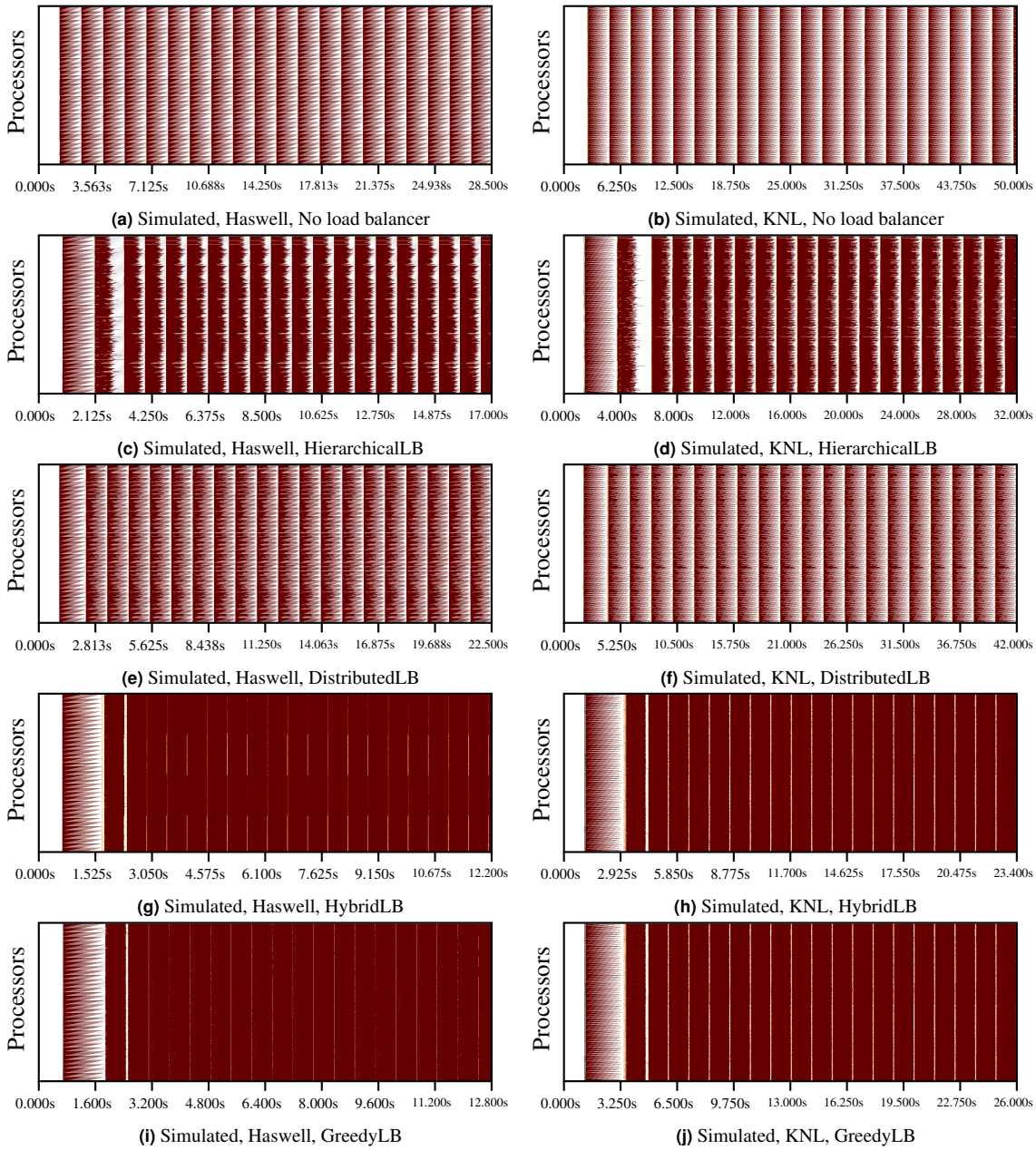
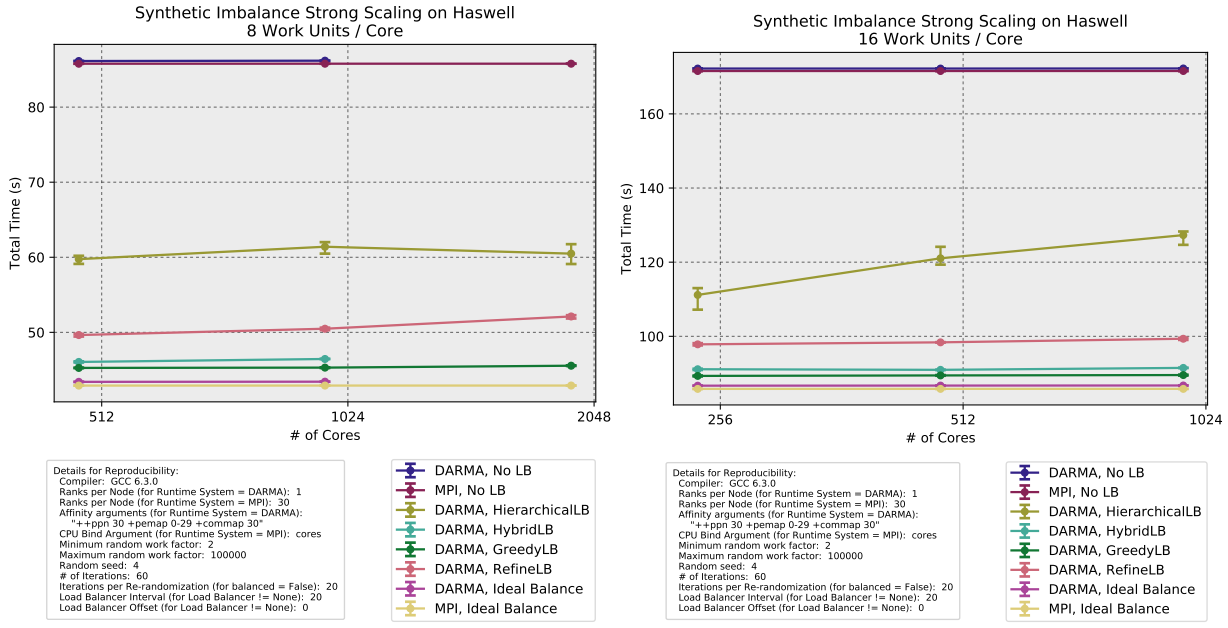


Figure 5.21: Execution timeline (projection) of different load balancers for DARMA-CHARM++ simulated imbalance benchmark on 64 nodes of Mutrino for different load balancers introduced in Figure 5.20. Time advances on the x-axis for individual threads of execution on the y-axis. Maroon shows compute while white shows idle time.



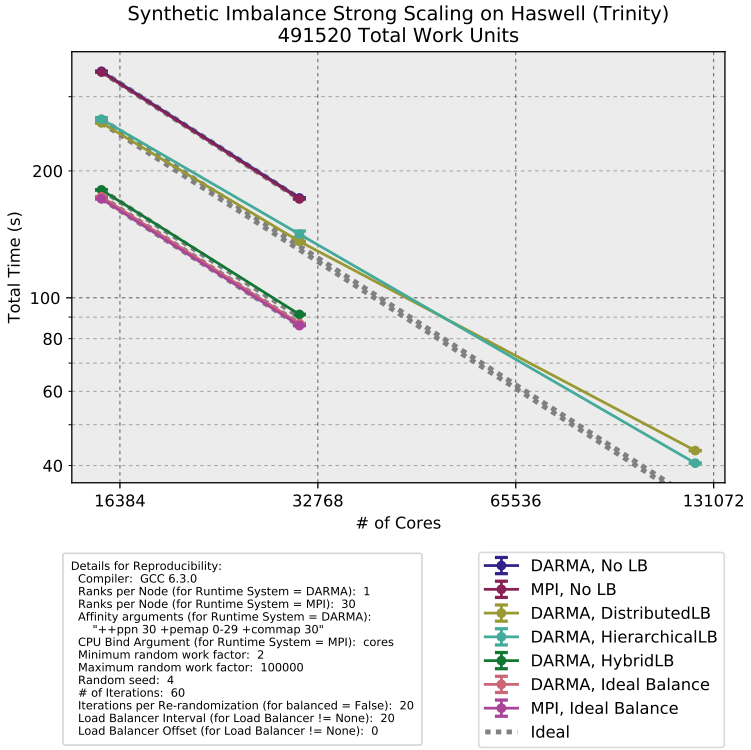
(a) 8 Tasks Per Core

(b) 16 Tasks Per Core

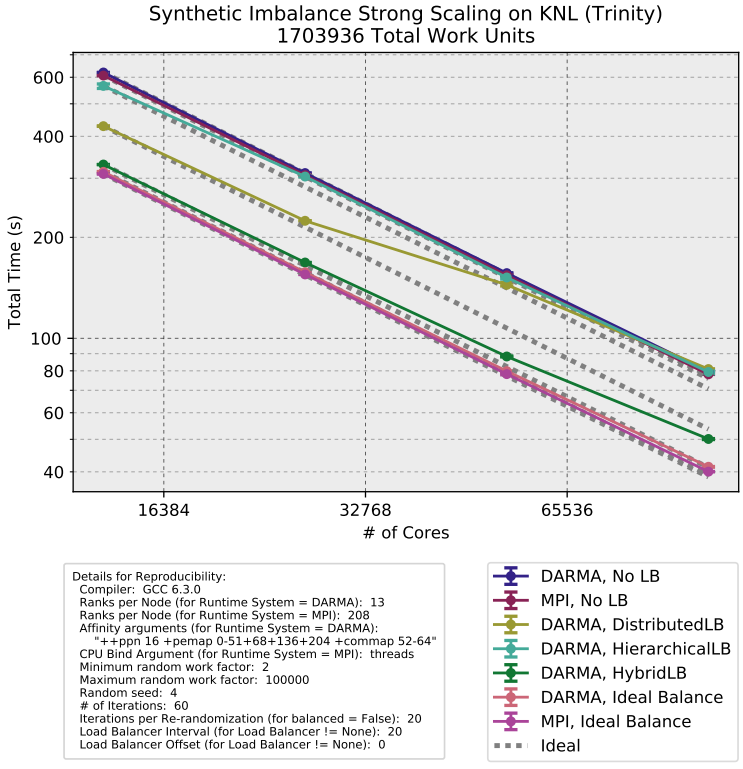
(c) 32 Tasks Per Core

(d) 64 Tasks Per Core

Figure 5.22: Weak scaling performance of simulated imbalance benchmark for MPI and DARMA-CHARM++ for increasing overdecomposition factors. Scaling is shown up to 64 nodes on Mutrino for Haswell (2048 cores) in (a) and (b) and for KNL (4352 cores) in (c) and (d). Error bars show min and max values observed from multiple trials. Several different load balancing (LB) algorithms are shown.



(a) Haswell



(b) KNL

Figure 5.23: Strong scaling performance of simulated imbalance benchmark for MPI and DARMA-CHARM++. Scaling is shown up to 2048 nodes on Trinity for (a) Haswell (65K cores) and (b) KNL (139K cores). Error bars show min and max values observed from multiple trials. Several different load balancing (LB) algorithms are shown. The overdecomposition factor is set to 4.

5.4.4 Common Trends Amongst Benchmarks

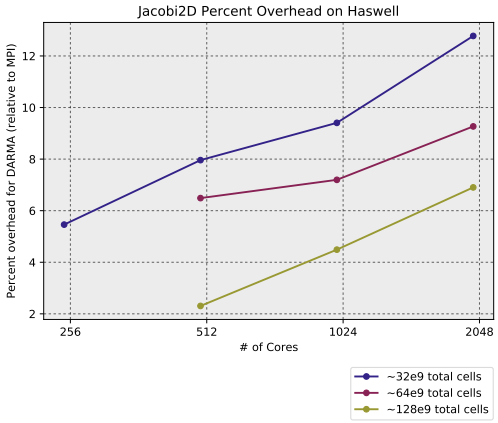
5.4.4.1 Overhead relative to MPI on Haswell

Figures 5.30 and 5.24 show clear trends on Haswell for the relative performance difference between MPI and DARMA. Larger problem sizes amortize DARMA scheduling overheads, lowering the relative overhead. The overheads tend to increase at larger node counts, which is again related to the extra task overhead of scheduling tasks over additional nodes. For the synthetic imbalance benchmark in particular and also Jacobi, the relative differences are quite small. For PIC, the differences are more significant.

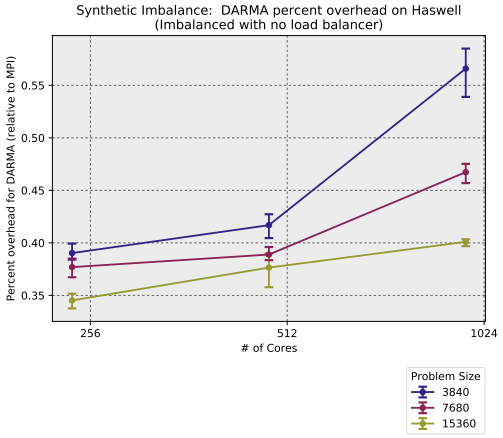
5.4.4.2 Overhead relative to MPI on KNL

Some of the trends for Haswell reverse for KNL as seen in Figures 5.28 and 5.25. For larger problem sizes, the trend remains the same and DARMA relative performance improves as heavier-weight tasks amortize scheduling overheads. For strong scaling, though, relative DARMA overhead actually decreases at larger scales. One hypothesis for this is the nature of KNL versus Haswell cores. The KNL is being used in an MPI-only manner, which requires all cores (and threads) to drive communication. DARMA, however, uses at most 13 communication threads for the entire KNL and runs fewer processes per node. Haswell cores also have better serial performance and are likely more efficient at performing system and communication work. These factors likely indicate that the use of dedicated asynchronous communication threads in DARMA make communication intrinsically more scalable. Future study is required here, particularly running MPI codes with OpenMP rather than MPI-only mode.

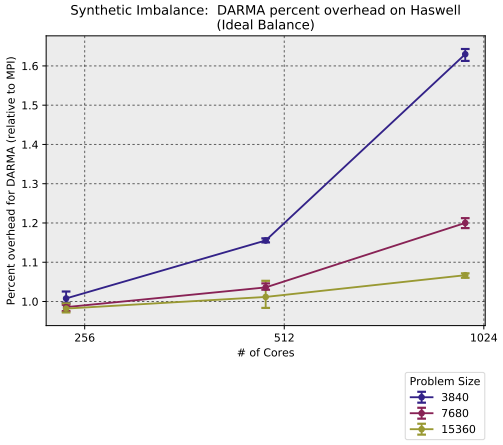
The downward trend in figures (a)-(c) also changes when load balancing is run. Figure 5.25 (c) shows gradually increasing overheads for larger node counts. This makes sense given that load balancing is the heaviest operation that DARMA performs - and becomes more expensive with scale.



(a) Jacobi

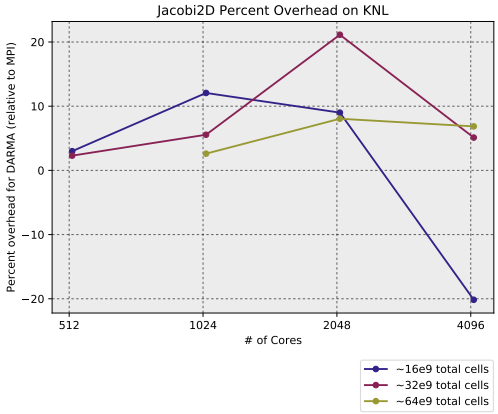


(b) Simulated No Balancing

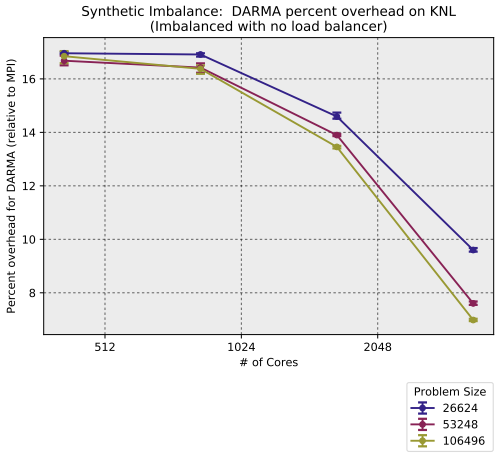


(c) Simulated With Balancing

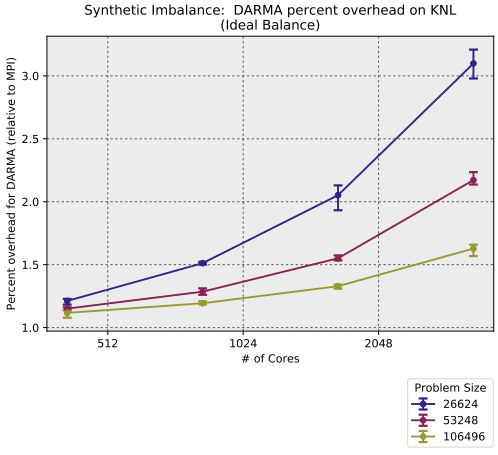
Figure 5.24: Percent overhead of DARMA relative to MPI. Scaling is shown up to 64 nodes on Mutrino for Haswell (2048 cores; 4096 threads). Note the different scales. Benchmarks shown are for (a) Jacobi and synthetic imbalance benchmarks (b) without load balancing and (c) with load balancing.



(a) Jacobi



(b) Simulated No Balancing



(c) Simulated With Balancing

Figure 5.25: Percent overhead of DARMA relative to MPI. Scaling is shown up to 64 nodes on Mutrino for KNL. Note the different scales. Benchmarks shown are for (a) Jacobi and synthetic imbalance benchmarks (b) without load balancing and (c) with load balancing.

5.5 Particle-in-Cell Proxy

5.5.1 Background and Relevance to Sandia's ATDM Program

Computer simulations of plasmas rely on kinetic or fluid descriptions of the underlying physics. In a fluid description of plasmas, moments of the distribution function are self-consistently integrated with the field equations, assuming approximate transport coefficients. Kinetic descriptions, in contrast, provide a self-consistent and fully kinetic representation of charged particle interactions with each other and electromagnetic fields. The primary two methods using kinetic descriptions are either solving plasma kinetic equations (Vlasov or Fokker-Planck equations, for example) or by "particle" simulations. There are three principal types of plasma simulation models: the particle-particle (PP) model, PIC (sometimes referred, also, as particle-mesh (PM)), and the particle-particle–particle-mesh (PPPM or P^3M) [33, 34]. In present study we will focus only on PIC method, which was successfully employed since early 1950s [35, 36].

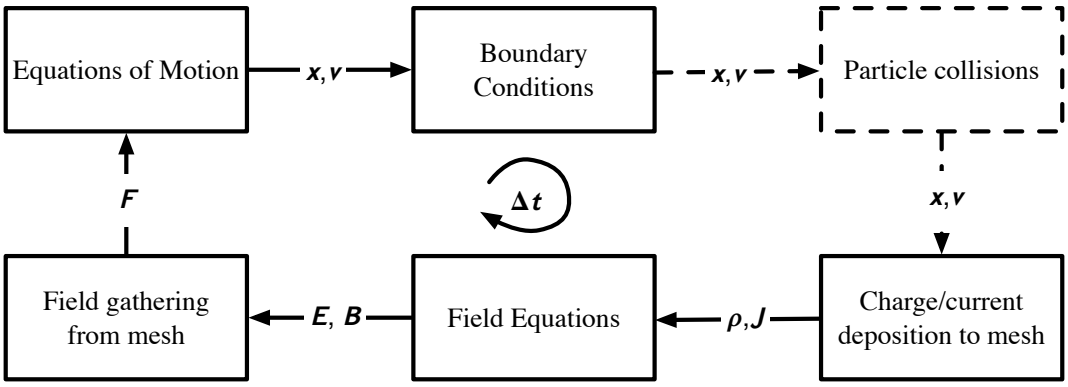


Figure 5.26: Scheme of the PIC simulation.

A simplified scheme of the PIC simulation is given in Figure 5.26, which in short is:

- solve equations of motion using force F computed during the previous time step and update particle position x and velocity v taking into the account boundary conditions, $\mathcal{O}(N_{\text{particles}})$,
- for collisional problems perform collisions and update (x, v), $\mathcal{O}(N_{\text{particles}})$,
- from (x, v) find charge ρ and current J (deposition from the particles to mesh), $\mathcal{O}(N_{\text{particles}})$,
- using (ρ, J) solve Maxwell's equations to obtain electric E and magnetic B fields, $\mathcal{O}(N_{\text{cells}})$,
- from (E, B) compute force F (field gathering from mesh to particles), $\mathcal{O}(N_{\text{particles}})$,
- start over.

Production-quality electrodynamics applications (like Sandia's EMPIRE project) based on PIC have very large code bases and are typically complicated and full-featured software products. Porting such applications to new computer architectures is a very challenging task, and consequently there have been multiple co-design efforts with PIC Miniapps. For example, in the scope of NERSC Exascale Science Applications Program (NESAP), an open source PIC library (a miniapp) PICSAR has been developed based on WARP code bases to prepare for the arrival of the supercomputer CORI phase II equipped with KNL nodes [37, 38] (performance details of this particular miniapp are included in the next Sections).

MINIPIC is Sandia's particle-in-cell miniapp, designed to solve the discrete Boltzmann's equation in an electrostatic field in an arbitrary domain with reflective walls. It is used as a tool within Sandia's ATDM program to prepare the EMPIRE code base for next generation platforms. MINIPIC has also been used

externally within the APEX procurement process as a benchmark for Crossroads/NERSC 9 [39]. MINIPIC has potentially difficult dynamic memory requirements, localized work (potentially with imbalances), and stochastic processes. The MINIPIC benchmark uses an unstructured hex- or tet-based mesh with a static partition used for a particle mesh. Particles are tracked to every cell crossing, packed and passed off to adjacent processors using MPI. The main code base uses Tpetra objects from the Trilinos mathematics library for matrix/vector operations. KOKKOS kernels are used on a node level to provide performance portability across architectures.

MINIPIC has been ported to the following three platforms: Pthreads on Intel Xeon, OpenMP on Intel Xeon Phi, and CUDA on Nvidia GPU. KOKKOS had significantly simplified the porting process, yet some machine specific tuning was required. Preliminary profiling showed the work breakdown to be roughly the following: particle move $\approx 60\%$, particle charge/current deposition to mesh $\approx 30\%$, field equations solve $\approx 1\%$ and field gathering from mesh $\approx 10\%$. The study also showed around 10 fold improvement in the particle move kernel on a structured mesh.

Given the number of external packages that MINIPIC relies on, a variant of MINIPIC, called SIMPLEPIC was developed, to serve as a testing ground for design and implementation of new DARMA-based PIC algorithms. From SIMPLEPIC, the algorithms can be ported into MINIPIC and, as KOKKOS and MPI interoperability are supported, eventually into the EMPIRE code base. SIMPLEPIC is a three-dimensional PIC move kernel (in Figure 5.26, the part where the equations of motion are solved) on a structured (regular) hex-based mesh with periodic or reflective boundary conditions. SIMPLEPIC can run in parallel either using MPI or DARMA. To date, the DARMA version of the move kernel has been ported into a simplified version of MINIPIC, comprising only move and collide kernels (see Figure 5.26). This version of MINIPIC has been further simplified to eliminate MPI, KOKKOS, and Trilinos (other than the Teuchos packages). Over the course of FY18, as interoperability with these capabilities is supported, this functionality will be added back into MINIPIC.

In the following Subsection 5.5.2 we discuss the general code flow of SIMPLEPIC. Both DARMA-CHARM++ and MPI implementations are made as simple as possible, i.e. no advanced MPI or DARMA-CHARM++ features were used. The goal was to reflect an "average" user's experience with both systems. In the Subsection 5.5.3 we show the performance of SIMPLEPIC on Mutrino and Trinity machines. The summary and future work are discussed in the Subsection 5.5.4.

5.5.2 Design of SIMPLEPIC

In SIMPLEPIC, the computational domain is a box composed of a $n_x \times n_y \times n_z$ grid. The computational grid is divided amongst $P_x \times P_y \times P_z$ patches. Each patch hold its own set of particles. The collection of particles on a given patch is called a *swarm*. The particle move kernel algorithm implemented in SIMPLEPIC is shown in Algorithm 1. The program starts from parsing the command line arguments, which are n_x, n_y, n_z, P_x, P_y, P_z and the input file location (line 1). The input file contains information including the maximum CFL (Courant-Friedrichs-Lewy) number as well as particle initial positions and velocities.

Once the inputs are read, the computational domain is decomposed and assigned to the units of computation (line 2). In the MPI implementation we require that the total number of patches be equal to the total number of MPI ranks, i.e. no *overdecomposition* is allowed. For the DARMA-CHARM++ implementation, the total number of patches can be specified independently of the number of processes or nodes. After patches are created, the swarm gets initialized on a patch (line 3). For every time step the following actions are

Algorithm 1 Code flow for a naïve implementation of PIC move kernel in SIMPLEPIC.

```
1:  Read input file and parse command line arguments
2:  Decompose domain into patches and assign them to units of computation
3:  For every patch initialize the swarm (particles on a given patch)
4:  For each patch and for each interface (6 in total) initialize an empty buffer for migrants
5:  for each time step do
6:      for each particle p in the swarm do
7:          Advance time for p until it reaches the patch interface or time expires
8:          if for p time is not expired then
9:              Add p in the appropriate migrants buffer
10:             Delete p from swarm
11:         end if
12:     end for
13:     Compute the total number of migrants in the domain
14:     while total number of migrants > 0 do
15:         For every interface exchange the migrants with adjacent patch's migrants
16:         For each interface (6 in total) initialize an empty buffer for migrants_temp
17:         for each interface do
18:             for each particle p in the migrants do
19:                 Advance time for p until it reaches the patch interface or time expires
20:                 if for p time is expired then
21:                     Add p in the swarm
22:                 else
23:                     Add p in the appropriate migrants_temp
24:                 end if
25:             end for
26:         end for
27:         Replace migrants with migrants_temp
28:         Compute the total number of migrants in the domain
29:     end while
30: end for
```

performed (lines 5-30): every particle advances in time (moves) until either the time (dt) expires or the particle reaches the patch interface (lines 6-12). In the latter case, the particle is removed from the swarm and added to the *migrants* buffer corresponding to that interface. Next, the total number of migrants (in all the buffers) in the domain is computed via allreduce (line 13).

If the total number of migrants is not zero, a while loop is launched, to perform so-called *micro-iterations*, until the total number of migrants is zero (lines 14-29). During the micro-iteration, the migrants on the patch interfaces will be exchanged (line 15). For a given patch, the newly arrived migrants (the particles) will be advanced in time (moved) until either they expire their remaining time, in which case they will be added to the swarm, or they arrive at another patch interface, in which case they will be placed in a temporary *migrants_temp* buffer (lines 17-26). Once all the migrants from all the interfaces are advanced, the migrants buffer will be replaced by the temporary migrants_temp buffer (line 27). The total number of migrants in the domain will be computed once again with an `allreduce` (line 28).

5.5.3 Performance of SIMPLEPIC on Target Architectures

To assess the performance of SIMPLEPIC, we have designed two types of PIC benchmark problems: *balanced* and *imbalanced*. We use the balanced PIC benchmark to assess the overheads of DARMA-CHARM++ with respect to MPI. While the imbalanced PIC benchmark is used to assess the benefits of overdecomposition and load balancing. A balanced PIC problem starts with an equal amount of particles (e.g. 5-30 particles) uniformly distributed in each cell. Every particle has the same constant velocity norm (equal to 1 in our studies), but the direction of velocity is randomly assigned (uniform distribution). For this study there are periodic boundary conditions on all the boundaries of computational domain.

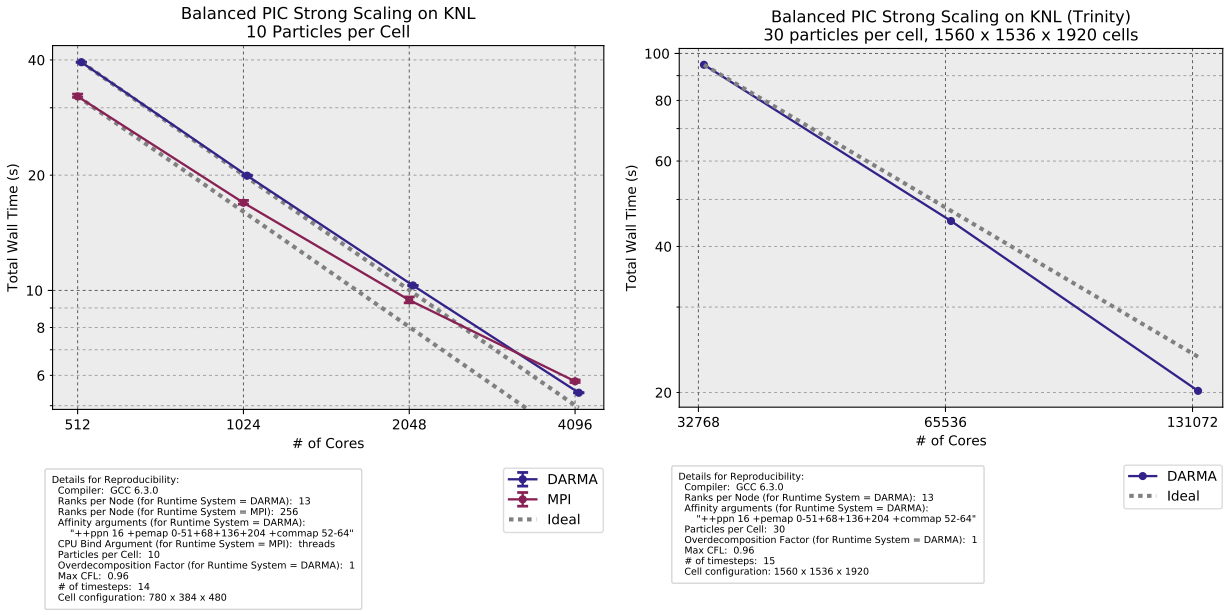
An imbalanced PIC problem starts by uniformly concentrating X% from a given total number of particles in Y% of the computational domain (for our benchmarks, X = 80% and Y = 20%). The remaining particles (1-X) are uniformly distributed in the remaining computational domain (1-Y), such that every cell has at least one particle. Every particle has the same constant velocity norm (equal to 1 in our studies), but the direction of velocity is randomly assigned (uniform distribution). We use periodic boundary conditions for imbalanced problems. For our studies, the imbalanced problems are constructed so that after 1000 iterations all particles will return to their initial positions. The problem therefore goes from imbalanced to balanced in 500 iterations and then back to imbalanced in the next 500 iterations. For the both balanced and imbalanced benchmarks the maximum CFL number is chosen to be 0.96. This ensures that particles will cross at most three computational cells during the time step.

Similar to the Jacobi benchmark discussed in Section 5.4.1.1, for SIMPLEPIC, we performed extensive studies on CPU binding and affinity for various overdecomposition factors. Our findings matched the results reported in Figure 5.4. On Haswell, SIMPLEPIC was run with hyperthreading turned off, while on KNL the best performance was achieved by using 4× hyperthreading per core in the 13× configuration.

Doerfler *et al.* reported that, for their PIC studies, the best performance is achieved when run no hyperthreading on Haswell (2 MPI tasks and 16 OpenMP threads) and with hyperthreading on KNL [38]. Doerfler *et al.* utilized a roofline analysis to investigate the performance envelope under which the PICSAR code base performance exists [38,40]. They reported on how tiling and vectorization improve the arithmetic intensity (total flops computed/total bytes transferred from DRAM) and increase the overall performance, reaching a higher memory bandwidth ceiling. In particular, the performance relative to the unoptimized code was improved by a factor of 4 for Haswell and 10.8 for KNL. PICSAR, even optimized, is still mainly memory bound and far from maximum CPU utilization.

5.5.3.1 Benchmark of Balanced PIC Problem

In Figure 5.27 the strong scaling of the balanced PIC problem is shown for the KNL architecture on Mutrino (a) and on Trinity (b). DARMA-CHARM++ version of SIMPLEPIC has excellent scalability on both machines. The scaling of DARMA-CHARM++ on Trinity is super-linear mostly due to cache effects and efficient utilization of high-bandwidth MCDRAM. Similar scaling trends are observed for the Jacobi-2D benchmark discussed in the Section 5.4.1. While DARMA-CHARM++ scales linearly and almost ideally for Mutrino, the MPI scaling is not optimal for higher core counts. The relative differences between MPI and DARMA-CHARM++ for various problem sizes on Mutrino, shown in the Figure 5.28, are between 20% and 25%. Due to the poor scaling (compared to the ideal scaling) of MPI, the relative difference decreases for a larger number of cores.



(a) 1.4B particles

(b) 138B particles

Figure 5.27: Strong scaling performance of balanced PIC problem for (a) MPI and DARMA-CHARM++ for 1.4B particles (30 particles per cell), shown up to 4K cores (64 nodes) on Mutrino KNL; (b) DARMA-CHARM++ for 138B particles, shown up to 131K cores (2K nodes) on Trinity KNL. In both Figures overdecomposition factor (ODF) is 1. Error bars show min and max values observed from multiple trials.

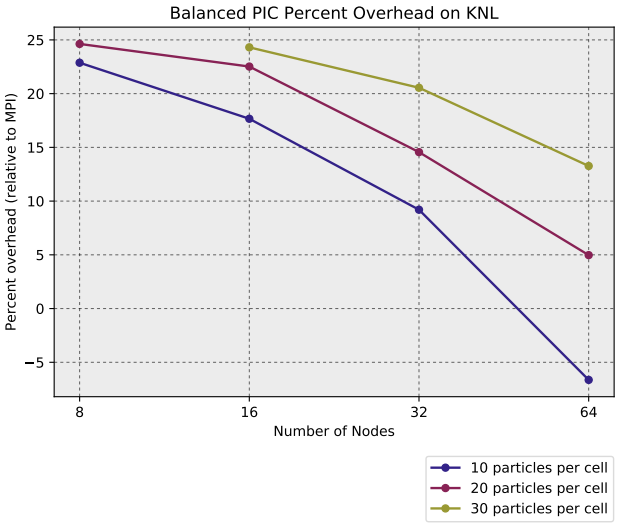
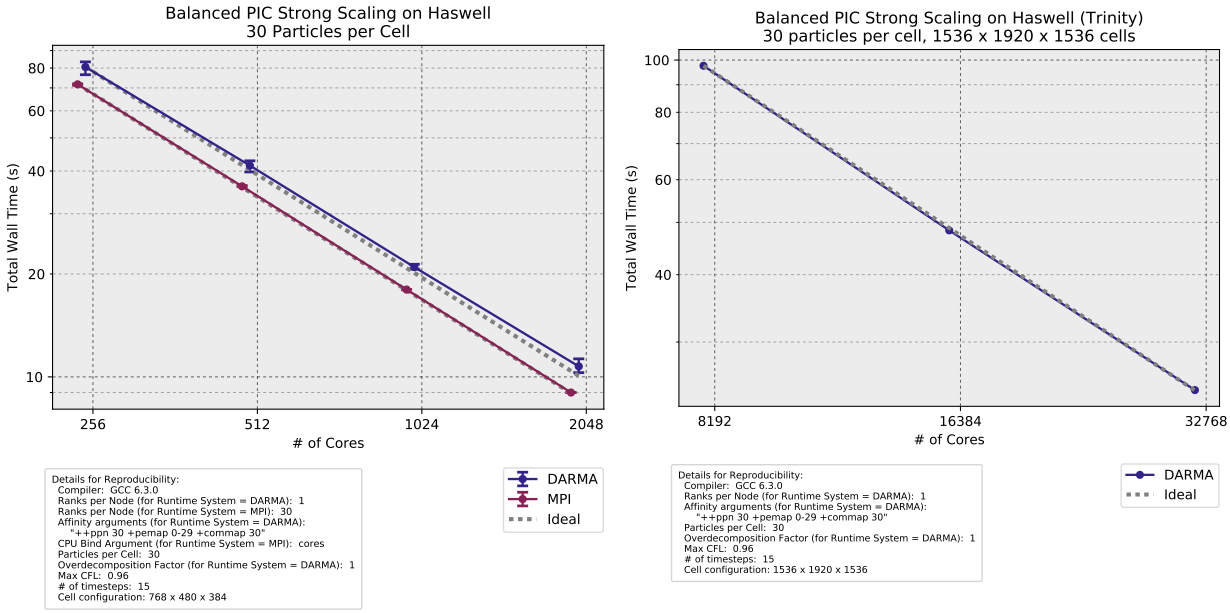


Figure 5.28: Percent overhead of DARMA-CHARM++ relative to MPI for different problem sizes as a function of core count. Scaling is shown up to 64 nodes (4K cores) on Mutrino for KNL.

In Figure 5.29 the strong scaling of the balanced PIC problem is shown for the Haswell architecture on Mutrino (a) and on Trinity (b). MPI scales ideally as one would expect in this case. DARMA-CHARM++ scales almost ideally on Mutrino and ideally on Trinity. The relative differences between MPI and DARMA-CHARM++ for various problem sizes on Mutrino, shown in Figure 5.30, are between 12% and 29%. The relative difference increases when the number of nodes is increasing. For a larger problem size (30 particles per cell) the relative difference is 50% lower than for a smaller problem size (10 particles

per cell) on 64 nodes. This difference can be potentially explained by the fact that for smaller problems the computation per iteration (the grain size) is not sufficient to amortize the overhead costs of DARMA-CHARM++ by better overlapping computation and communication.



(a) 4.2B particles

(b) 136B particles

Figure 5.29: Strong scaling performance of balanced PIC problem for (a) MPI and DARMA-CHARM++ for 4.2B particles (30 particles per cell), shown up to 2K cores (64 nodes) on Mutrino Haswell; (b) DARMA-CHARM++ for 136B particles, shown up to 32K cores (1K nodes) on Trinity Haswell. In both Figures the overdecomposition factor (ODF) is 1. Error bars show min and max values observed from multiple trials.

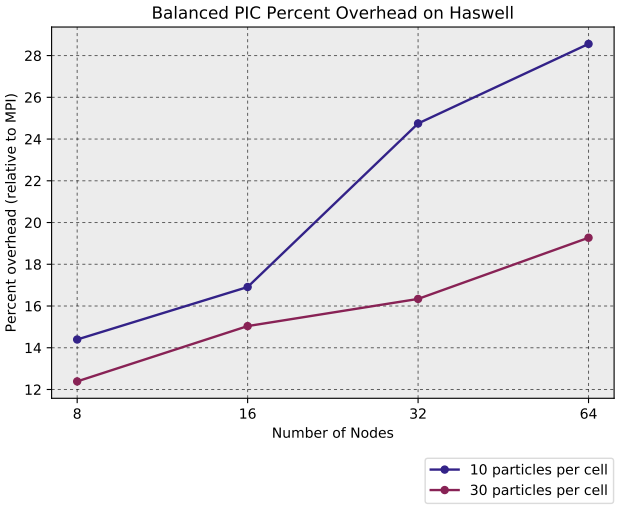
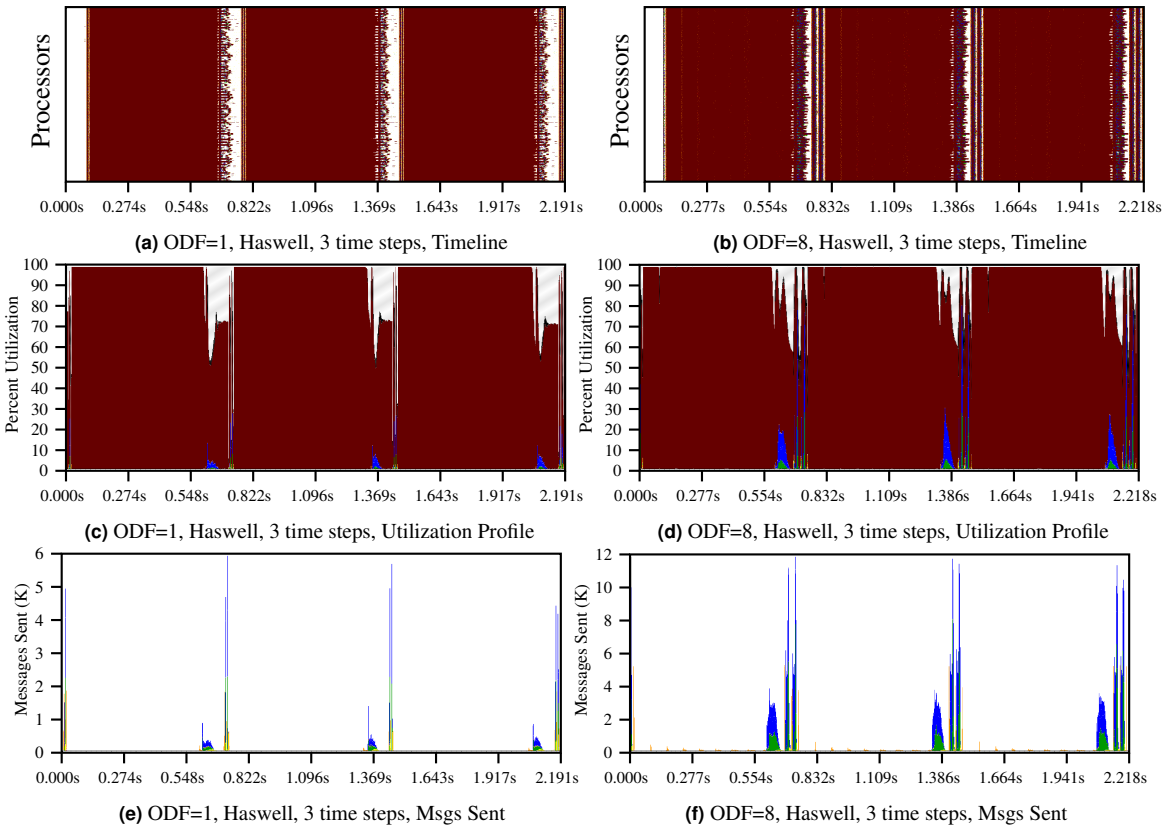


Figure 5.30: Percent overhead of DARMA-CHARM++ relative to MPI for different problem sizes as a function of core count. Scaling is shown up to 64 nodes (2K cores) on Mutrino for Haswell.

Detailed Performance Analysis for DARMA-CHARM++ In Figure 5.31 we show a timeline and CPU utilization views that incorporates a sample of the work individual threads are performing across time (white

is idle time, maroon is application work). We also show histograms for the messages sent over time for 3 iterations. On the right side, the timelines are shown for an overdecomposition factor of 1, while on the right the overdecomposition factor is 8. The timelines shown in Figure 5.31 are for Mutrino (Haswell) on 64 nodes, while analogous results are presented for Mutrino (KNL) on 64 nodes in Figure 5.32.



(a) ODF=1, Haswell, 3 time steps, Timeline

(b) ODF=8, Haswell, 3 time steps, Timeline

(c) ODF=1, Haswell, 3 time steps, Utilization Profile

(d) ODF=8, Haswell, 3 time steps, Utilization Profile

(e) ODF=1, Haswell, 3 time steps, Msgs Sent

(f) ODF=8, Haswell, 3 time steps, Msgs Sent

Figure 5.31: Timeline views (first row) and CPU utilizations (second row) for a sample of threads over time of the SIMPLEPIC (in DARMA-CHARM++, 4.2B particles) on 64 nodes of Mutrino (Haswell) for the overdecomposition levels of 1, shown on left column, and 8, shown on right column. In the timeline views, task execution is indicated in maroon while idle time is shown in white. The orange and yellow colors represent dependency management and collectives (`allreduce`) respectively.

A majority of SIMPLEPIC's work (shown in a maroon color) is spent on the first portion of the move kernel (see the lines 6-12 in the algorithm 1). After this block of work, the buffers of migrants are exchanged. This exchange is shown in rows 2 and 3 (in both Figures 5.31 and 5.32) using the color blue for sends and green for receives. As can be seen from the images in row 2, the send/receives are well overlapped with computation. With an overdecomposition factor of 8 (the right column), the send/receive is more spread in time.

In Figure 5.33 we show similar timelines and histograms for the last 2 micro-iterations. On the right side, the timelines are shown for the overdecomposition factor of 1, while on the right the overdecomposition factor is 8. The timelines, shown in Figure 5.33 are for Mutrino (Haswell) on 64 nodes, while a similar results are presented for the Mutrino (KNL) on 64 nodes in the Figure 5.34.

For the overdecomposition factor of 8, surface-to-volume ratios are increased in the domain decomposition. This introduces associated increases in the amount of time spent on `allreduce` (shown in yellow) and dependency management (shown in orange). However, since, these operations are asynchronous, they are

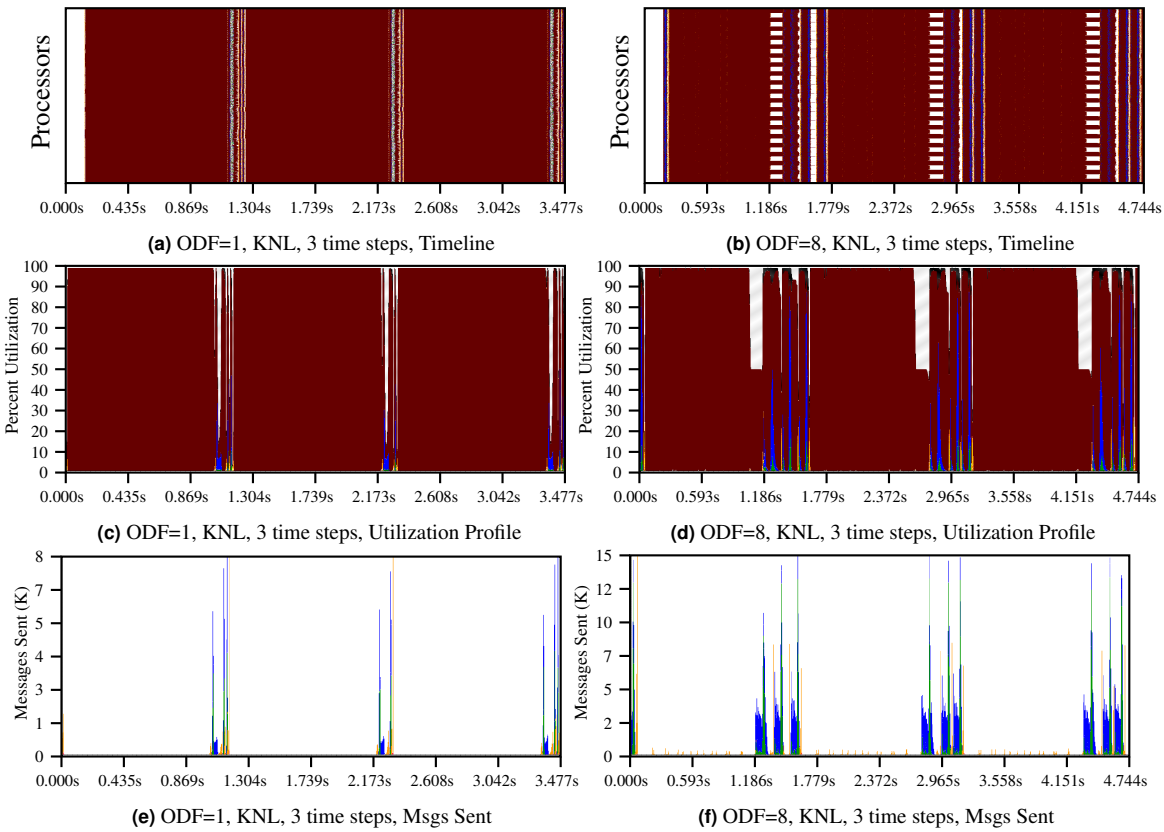


Figure 5.32: Timeline views (first row) and CPU utilizations (second row) for a sample of threads over time of the SIMPLEPIC (in DARMA-CHARM++, 4.3B particles) on 64 nodes of Mutrino (KNL) for the overdecomposition levels of 1, shown on left column, and 8, shown on right column. In the timeline views, task execution is indicated in maroon while idle time is shown in white. The orange and yellow colors represent dependency management and collectives (`allreduce`) respectively.

spread over time and are well overlapped with application computation. Of course, the more computation we can introduce (e.g. include collide kernel, start computing current) the impact of overdecomposition overheads decreases further.

Summary The balanced case study of SIMPLEPIC demonstrated excellent scalability of DARMA-CHARM++ on up to 131K cores on KNL and 32K cores on Haswell. The observed maximum overhead of DARMA-CHARM++ with respect to MPI depends on the problem size (the grain size). For example, when SIMPLEPIC was run on Mutrino (Haswell), shown in the Figure 5.30, the grain size is sufficiently large on 8 nodes and even larger when the 30 particles per cell were used. In this case, the overhead is only 12%. Since the grain size decreases with increased node count (strong scaling), the overhead of DARMA-CHARM++ increases with node count. The overhead increase is more dramatic when the grain size is small as seen in the case when only 10 particles per cell are used.

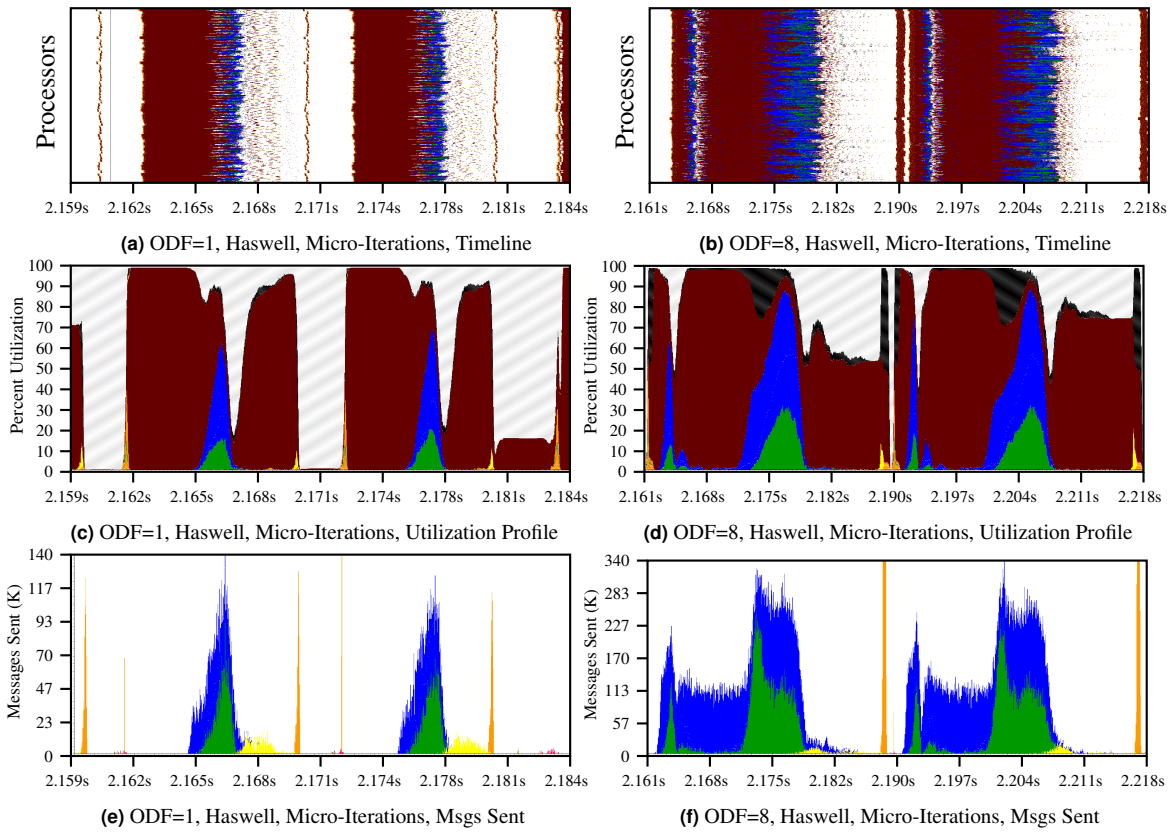


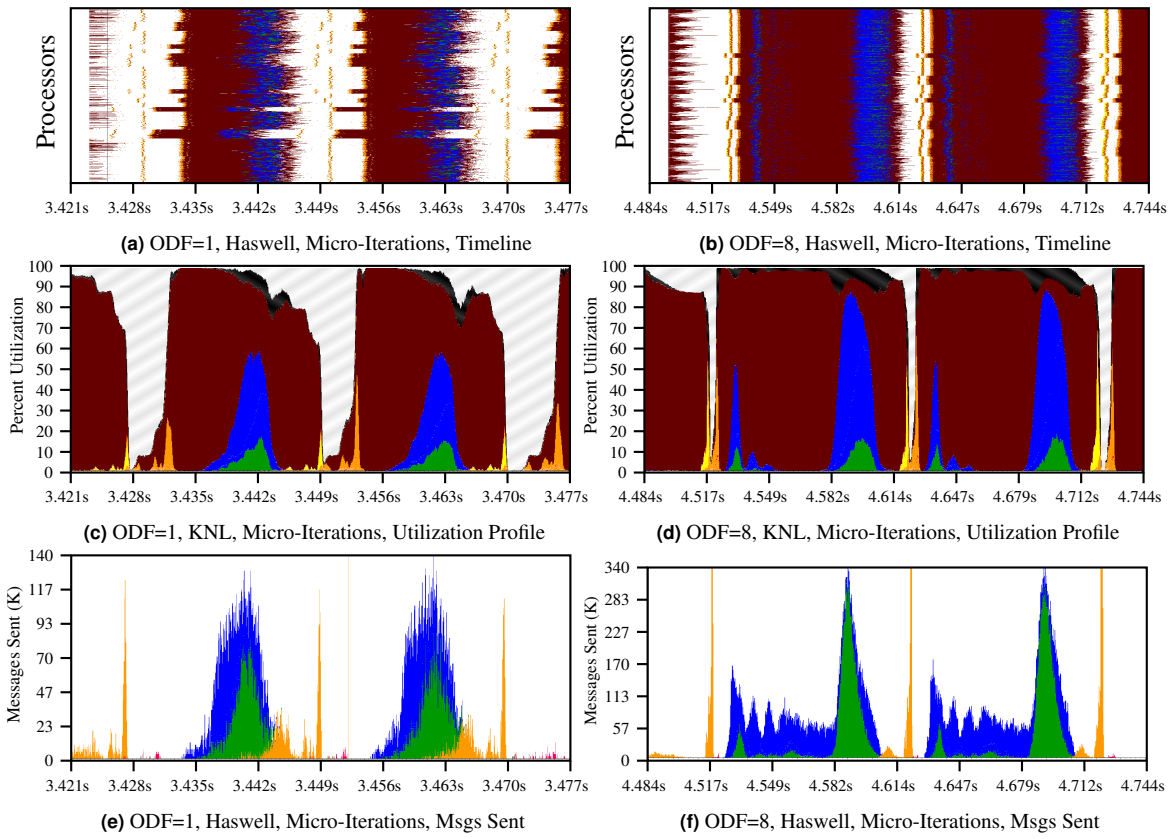
Figure 5.33: Timeline views (first row) and CPU utilizations (second row) for a sample of threads over time of the SIMPLEPIC (in DARMA-CHARM++, 4.2B particles) on 64 nodes of Mutrino (Haswell) for the overdecomposition levels of 1, shown on left column, and 8, shown on right column. In the timeline views, task execution is indicated in maroon while idle time is shown in white. The sizes of messages sent (blue) and received (green) over one iteration for varying levels of overdecomposition is shown in the third row. The CPU utilization during the micro-iterations (2 micro-iterations are performed) over one iteration for varying levels of overdecomposition is shown in the last row. The orange and yellow colors represent dependency management and collectives (`allreduce`) respectively.

5.5.3.2 Benchmark of Imbalanced PIC Problem

In Figure 5.35 the strong scaling of the imbalanced PIC problem is shown for the KNL architecture on Mutrino (a) and on Trinity (b). For Mutrino, we show the strong scaling results when Hybrid and Hierarchical load balancers were launched only once. At low core counts, a single load balancer launch provides 1.5× speedup. When the number of cores are increased, the speedup decreases. The factors contributing to this are the constant grain size and scalability of load balancing overheads. Similar trends are shown in the Figure 5.35 (b) for higher core counts on Trinity.

In Figure 5.36 we show the effect of changing the overdecomposition factor (or the grain size) on 64 nodes (Mutrino, KNL). Without load balancing, decreasing grain size will increase the overheads and performance will be decreased. On the other hand, when a load balancer is used, increasing overdecomposition factor will increase load balancing flexibility. In this case, performance for ODF 8 compared to ODF 1 is increased by 20% (for hybrid load balancer) and 10% (for hierarchical load balancer).

In Figure 5.37 (a) we show the strong scaling of the imbalanced PIC problem for the Haswell architecture



(a) ODF=1, Haswell, Micro-Iterations, Timeline

(b) ODF=8, Haswell, Micro-Iterations, Timeline

(c) ODF=1, KNL, Micro-Iterations, Utilization Profile

(d) ODF=8, KNL, Micro-Iterations, Utilization Profile

(e) ODF=1, Haswell, Micro-Iterations, Msgs Sent

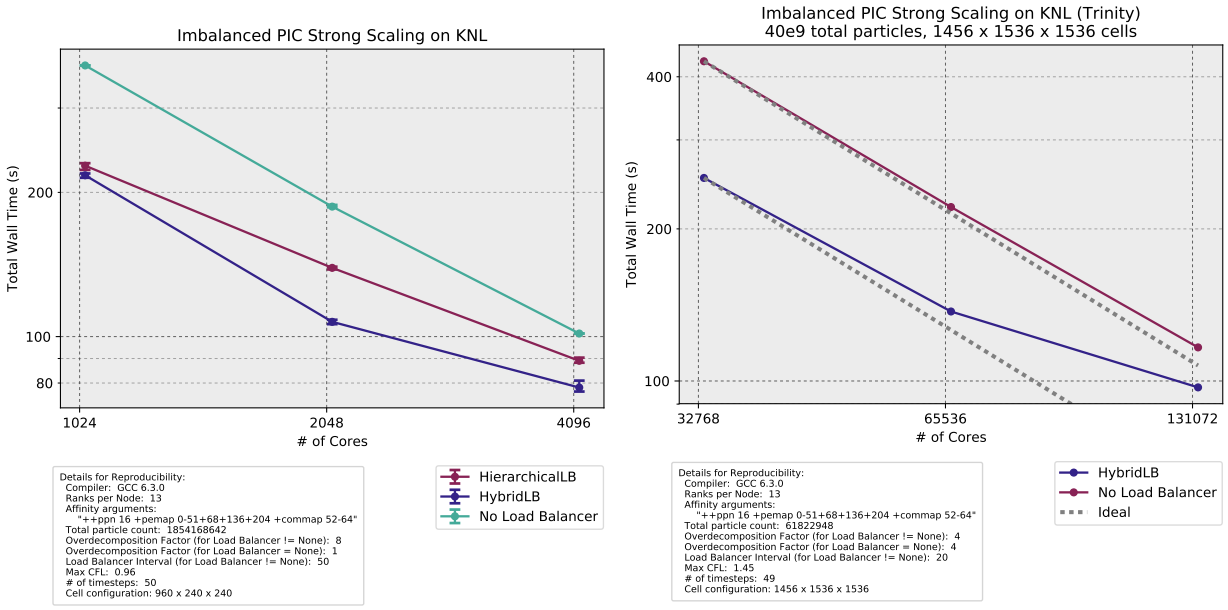
(f) ODF=8, Haswell, Micro-Iterations, Msgs Sent

Figure 5.34: Timeline views (first row) and CPU utilizations (second row) for a sample of threads over time of the SIMPLEPIC (in DARMA-CHARM++, 4.3B particles) on 64 nodes of Mutrino (KNL) for the overdecomposition levels of 1, shown on left column, and 8, shown on right column. In the timeline views, task execution is indicated in maroon while idle time is shown in white. The sizes of messages sent (blue) and received (green) over one iteration for varying levels of overdecomposition is shown in the third row. The CPU utilization during the micro-iterations (2 micro-iterations are performed) over one iteration for varying levels of overdecomposition is shown in the last row. The orange and yellow colors represent dependency management and collectives (`allreduce`) respectively.

on Mutrino. In this case, both load balancers scale linearly and significantly improve the performance. Such a strong difference between Figures 5.35 (a) and 5.37 (a) are related to the inherent architectural differences between KNL and Haswell (serial performance of individual cores and MCDRAM usage).

In Figure 5.37 (b) we show the effect of the frequency of the load balancer. When the total number of iterations are 50, the optimal frequency for load balancing is every 20 iterations. In this case, the performance is improved 2×. The difference between launching the load balancer every 10, 20 or 50 iterations is only a few percent. When the frequency is increased, the overhead associated with the load balancer becomes significant. Of course, the optimal frequency will depend on the specific problem and how quickly it becomes imbalanced.

Detailed Performance Analysis for DARMA-CHARM++ In Figure 5.38 we show a detailed timeline view on Mutrino (KNL) with the load balancer called only once (first row) and again with the load balancer called five times (second row). When the load balancer is called once, the CPU utilization is increased for the hybrid load balancer and, hence, the total execution time is decreased from 139s to 119s. Increasing the



(a) 1.8B particles

(b) 40B particles

Figure 5.35: Strong scaling performance of the imbalanced PIC problem for DARMA-CHARM++ with (a) 1.8B particles, shown up to 4K cores (64 nodes) on Mutrino KNL, hybrid and hierarchical load balancer are shown for ODF=8 together with no load balancer case with ODF=1; (b) 40B particles, shown up to 131K cores (2K nodes) on Trinity KNL, hybrid load balancer is shown for ODF=4 together with no load balancer case with ODF=4. Error bars show min and max values observed from multiple trials.

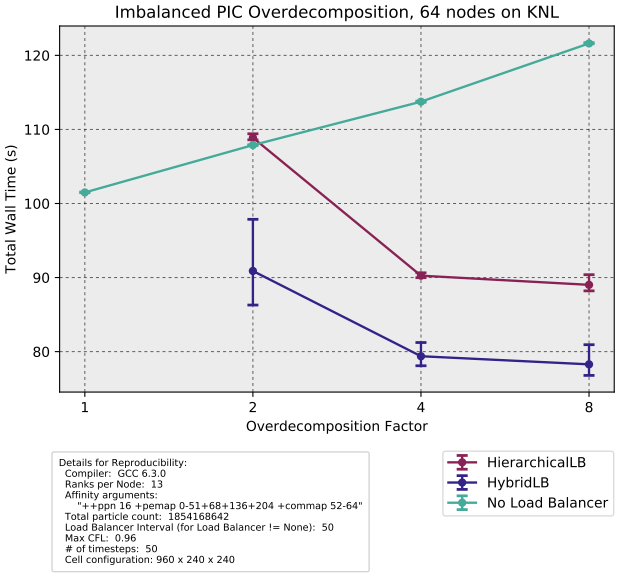
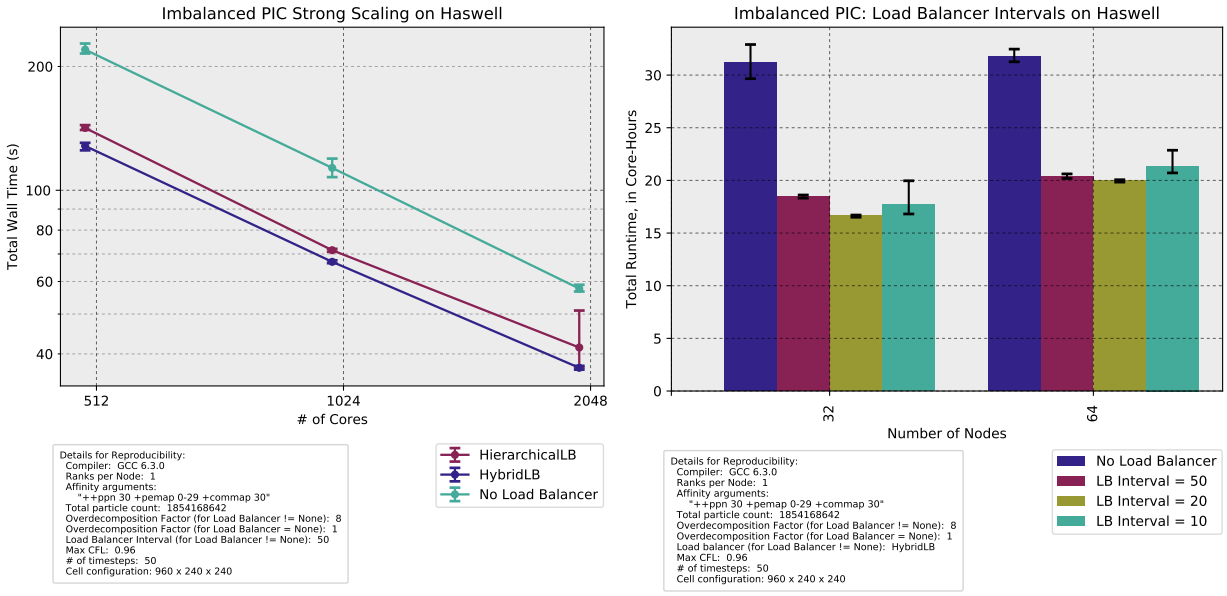


Figure 5.36: Total wall time for increasing overdecomposition factors. Scaling is shown up to 4K cores (64 nodes) on Mutrino for KNL. Hybrid and hierarchical load balancer are shown. Error bars show min and max values observed from multiple trials.

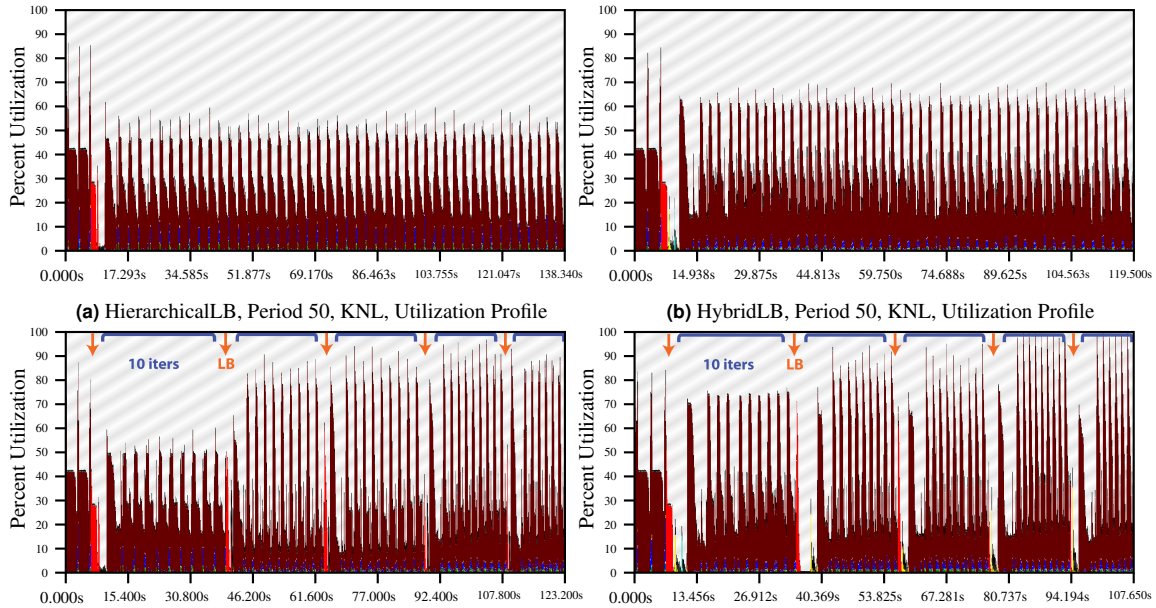
load balance launch frequency up to five improves the CPU utilization and the wall time.

We have investigated the effect of load balancers on long simulations in which the problem starts from a very imbalanced state and progresses to a fully balanced state within 500 iterations. In Figure 5.39 the timeline



(a) 1.8B particles (b) 1.8B particles

Figure 5.37: Strong scaling performance of balanced PIC problem for DARMA-CHARM++ with (a) 1.8B particles, hybrid and hierarchical load balancer are shown for ODF=8 together with no load balancer case with ODF=1; (b) The effect of the frequency of HybridLB load balancer call on performance for DARMA-CHARM++. Results are shown up to 64 nodes (2K cores) on Mutrino Haswell. Error bars show min and max values observed from multiple trials.



(a) HierarchicalLB, Period 50, KNL, Utilization Profile

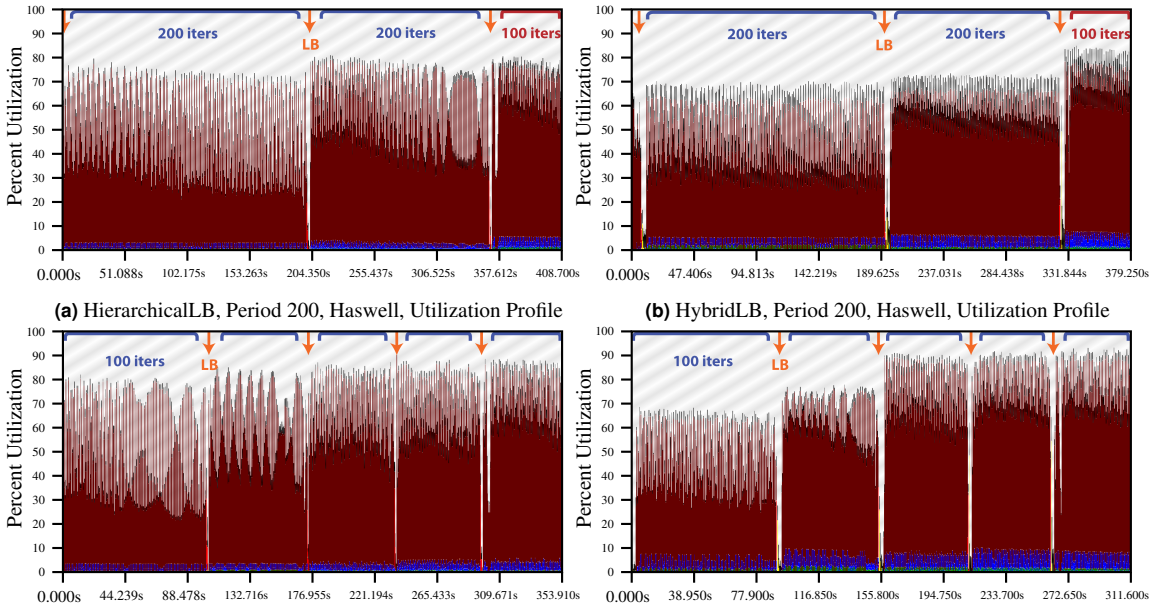
(b) HybridLB, Period 50, KNL, Utilization Profile

(c) HierarchicalLB, Period 10, KNL, Utilization Profile

(d) HybridLB, Period 10, KNL, Utilization Profile

Figure 5.38: CPU utilizations for a sample of threads over time of the SIMPLEPIC (in DARMA-CHARM++, 1.8B particles) on 4K cores (64 nodes) of Mutrino (KNL). On the left column the results with hierarchical load balancer are shown, while on right the results are obtained with hybrid load balancer. The CPU utilizations, where the load balancers were called only once (period is 50) and after every 10 iterations (period is 10), are shown in the first and second rows respectively.

of all 500 iterations are shown for Mutrino (Haswell) with the load balancer launched 3 times (first row) and 5 times (second row). The CPU utilization is higher and more uniform (balanced) when the Hybrid load balancer is used.



(a) HierarchicalLB, Period 200, Haswell, Utilization Profile

(b) HybridLB, Period 200, Haswell, Utilization Profile

(c) HierarchicalLB, Period 100, Haswell, Utilization Profile

(d) HybridLB, Period 100, Haswell, Utilization Profile

Figure 5.39: CPU utilizations for a sample of threads over time of the SIMPLEPIC (in DARMA-CHARM++, 1.8B particles) on 2K cores (64 nodes) of Mutrino (Haswell). On the left column the results with hierarchical load balancer are shown, while on right the results are obtained with hybrid load balancer. The CPU utilizations, where the load balancers were called after every 200 iterations (period is 200) and after every 100 iterations (period is 100), are shown in the first and second rows respectively.

Summary The imbalanced case study of SIMPLEPIC demonstrated that DARMA can leverage the load balancing capabilities provided by the CHARM++ backend. With load balancing, DARMA-CHARM++ showed promising strong scaling capabilities. The choice of load balancer, its frequency in the simulation, and the overdecomposition factor remain problem-specific parameters. DARMA-CHARM++ showed very predictable performance scalability on the Haswell architecture (see Figure 5.37 (a)), while on KNL, strong scaling performance degrades (see Figure 5.35 (a)).

5.5.4 Findings and Future Work

SIMPLEPIC was specifically designed to assess the performance and productivity of DARMA-CHARM++ for the PIC problem on Haswell and KNL architectures. For this purpose, two scenarios of PIC problems were designed: balanced and imbalanced. The goal of the PIC balanced problem was assessing the overheads of DARMA-CHARM++ compared to MPI, while the PIC imbalanced PIC problem assessed the benefits of load balancing capabilities.

Performance We have demonstrated that the balanced PIC problem strong scales on 131K KNL and 32K Haswell cores for DARMA-CHARM++. In the worst case (when using a grain size that is too small), we

Code	SLOCCount (lines of code)*
MPI	435
DARMA	592
Common Components	1061

* From David A. Wheeler tool that counts lines of code.

Table 5.3: Lines of code for SIMPLEPIC in MPI, DARMA, and the common components shared by both MPI and DARMA versions.

observe that DARMA-CHARM++ has a 29% overhead over MPI. However, we believe that the overheads should be between 10% and 15% for a properly sized simulation. The imbalanced PIC study showed more than 2× performance improvement with load balancing enabled, while strong scaling on up to 131K KNL cores.

Productivity Table 5.3 shows the number of lines of code for the SIMPLEPIC implementation in DARMA and MPI. Although, the DARMA implementation is slightly longer, DARMA-CHARM++ provides a seamless and productive way expressing the PIC algorithm. The abstractions like `create_concurrent_work(...)` and `AccessHandleCollection<T>` allow the application developer to implement the PIC algorithm without thinking about overdecomposition. The overdecomposition factor can be an input parameter to the simulation. Similarly, the load balancer and the frequency with which it is run are inputs to the simulation. The same application code will run with or without load balancers, as well as with any value of overdecomposition factor. Although the DARMA-CHARM++ backend is still maturing and hardening, there are productivity and intuitiveness advantages in DARMA implementation over MPI.

Future Work SIMPLEPIC will be extended to include a collide kernel. This will provide a significant work load, which will decrease the overhead of collective operations (like `allreduce`). Further, field solvers can be included to increase the complexity of the problem. In parallel to the efforts presented in the current report, the move kernel of MINIPIC was modified to leverage the advantages provided by DARMA. Over the course of the present work, various move algorithms were designed and are going to be implemented in SIMPLEPIC/MINIPIC next year. Crucial capability requirements going forward for MINIPIC in DARMA are KOKKOS and MPI interoperability (see the Sections 4.1 and 4.2).

5.6 Uncertainty quantification (UQ) Proxy

5.6.1 Background and Relevance to Sandia's ATDM Program

UQ broadly defines the science of quantitatively characterizing uncertainties in computational models and real world applications. Embedded analysis for sensitivities and UQ is a pivotal research and development capability for both of Sandia's ATDM applications, as it enables new design and decision support capabilities for nuclear weapons applications. UQ can be divided into two main categories, namely forward and inverse problems. Forward problems explore how uncertainties in input parameters of a target model/application affect target observables. Inverse problems deal with finding the parameters of a target model that best suit some data. In this work we focus on forward problems. The uncertainty of target parameters is expressed as probability density, which is then propagated through the forward problem, yielding probability densities for the outputs.

Figure 5.40 shows a very high-level overview of a typical forward UQ work flow. The plot should be read from bottom to top. The goal of a UQ analysis is to obtain statistics for target quantities of interests. The data from all the samples become available once all the forward problem solves are completed. These solves are independent from each other and can thus be run (in theory) simultaneously.

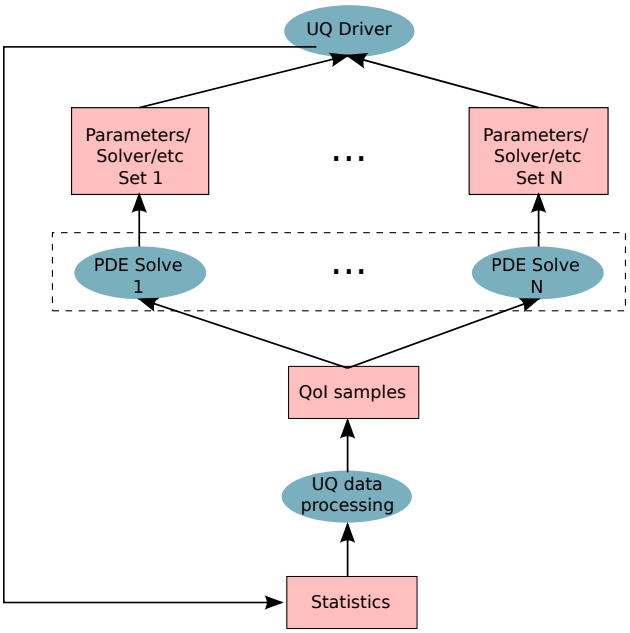


Figure 5.40: Draft of UQ Workflow: blue ovals represent tasks, pink boxes represents data.

5.6.2 Design of UQ Proxy

Some key features of typical UQ applications and their interplay with AMT models are:

- Forward problems are characterized by many independent samples, making them a natural fit for task-based models.

- Task re-usability can be a key property to leverage for sampling methods. At the level of forward problems solves one can exploit dependencies between tasks. For instance, a subset of these solves can use as initial condition the solution of other tasks if already available, and if not, use a "default" one. The reason for this is that it could potentially accelerate the convergence if the initial condition is good enough. This could be of substantial benefit for expensive forward simulations. This is a feature that we have not explored yet, but are planning to within the next fiscal year.
- Adaptivity: latest advances in UQ methods exploit more and more adaptive algorithms. The main idea behind adaptive algorithms is to schedule/drive the work based on the convergence of target metrics. One example of this class is shown below, namely adaptive Multi Level Monte Carlo. This field of applications is a good fit for dynamic work scheduling and lookahead techniques.

5.6.2.1 Monte Carlo (MC) Methods

Monte Carlo methods are one of the most common non-intrusive tools for tackling stochastic problems. These methods focus on finding the expected value $\mathbb{E}[Q]$, with Q being some quantify of interest (QoI). Q is typically a functional of the solution u of a target problem. For instance, the drag on an aircraft, the heat flux over a surface, the flow rate in channel flow, etc.

Since the "true" Q is not generally accessible, one relies on its approximation $Q_h = \mathcal{G}(u_h)$, where u_h denotes a discretized solution on a sufficiently fine spatial grid. For example, u_h can be obtained via a computational model. We assume that the expected value $\mathbb{E}[Q_h] \to \mathbb{E}[Q]$ as $h \to 0$.

Classical MC The classical Monte Carlo (MC) estimator for $\mathbb{E}[Q_h]$ is

$$\hat{Q}^{MC}_{h,N} := \frac{1}{N}\sum_{i=1}^{N} Q_h^{(i)}, \tag{5.1}$$

where $Q_h^{(i)}$ is the i^{th} samples of Q_h and N independent samples are computed. The estimator is denoted as $\hat{Q}^{MC}_{h,N}$, with the superscript indicating that it refers to the classical MC, and subscripts indicate that is obtained for a h-discretization and N samples. There are two sources of error in the above estimator: first, the use of Q_h to approximate Q, which depends on the (spatial) discretization used to solve the computational problem; second, the sampling error due to replacing the expected value by a finite sample average.

MLMC The main idea of multilevel Monte Carlo (MLMC) is to sample not just from one approximation Q_h of Q, but from several of them. Let $\{h_\ell\}_{l=0,\dots,L}$ be a sequence of meshes (approximations) of increasingly fine triangulations with L defining the finest level. The expectation at the finest level $\mathbb{E}[Q_L]$ can be written as

$$\mathbb{E}[Q_{h_L}] = \mathbb{E}[Q_{h_0}] + \sum_{\ell=1}^{L} \mathbb{E}[Q_{h_\ell} - Q_{h_{\ell-1}}] = \sum_{\ell=0}^{L} \mathbb{E}[Y_\ell] \tag{5.2}$$

where we have set $Y_\ell = Q_{h_\ell} - Q_{h_{\ell-1}}$ and $Y_0 := Q_{h_0}$. Eq. (5.2) says that expectation on the finest level is equal to the expectation on the coarsest level, plus a sum of corrections adding the difference in expectation between simulations on consecutive levels. The multilevel idea is to independently estimate each of these expectations such that the overall variance is minimized for a fixed computational cost. The work flow of the MLMC algorithm is summarized in Algorithm 2.

Algorithm 2 MLMC algorithm

Start with N_* samples on three levels $\ell = 0, 1, 2$.
converged = false
while !converged **do**
    evaluate extra samples needed on each level
    compute estimates for $\mathbb{V}_\ell$, $l = 0, ..., L$
    define optimal $N_\ell, l = 0, ..., L$
    test for weak convergence
    if all samples done and !converged **then**
        $L = L + 1$
        $N_L = N_*$
    end if
end while

It is important to emphasize that MLMC does not require the use of a geometric sequence of grids. The only assumption is that the accuracy and cost increase with level, and the variance of the correction term decreases with level. Obviously, problems involving meshes and sequences of grids are a suitable application. An example of a geometric sequence of levels is given in figure 5.41.

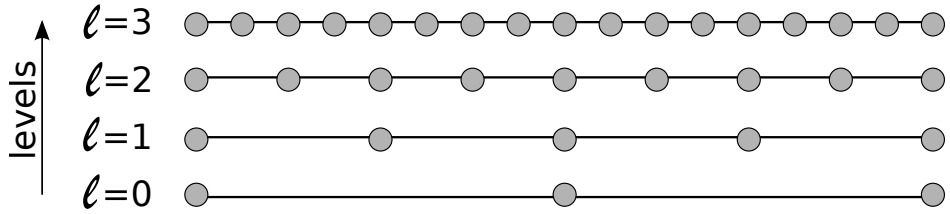


Figure 5.41: Example of geometric sequence of levels suitable for MLMC.

5.6.3 Performance of UQ Proxy on Target Architectures

In this section, we report some representative results obtained for a Monte Carlo and Multi Level Monte Carlo test.

5.6.3.1 Monte Carlo Example

As a test case for exploring MC in DARMA, we choose a linear 1D stochastic diffusion equation. The skeleton of the main function code is shown in Figure 5.42. The code takes three arguments from command line, namely the number of samples to execute for each DARMA index within a `create_concurrent_work` , the width of the `create_concurrent_work` , and an integer defining the grid size in power of 2. The core of the application is contained within the `Running` functor, whose code is shown in Figure 5.43. `create_work` is used to schedule a single PDE solve per task. However, this can be later relaxed when nested `create_concurrent_work` will be available. Each DARMA index deals with multiple solves. This is not the only solution possible, and it makes sense for cases where each PDE solve is small enough and the number of samples per index is such that the workload per index is substantial. However, if the computing load involved with each PDE solve is large enough, then it would be ideal to have a single solve per index. The

```
void darma_main_task(  std::vector<std::string> args)
{
  using vecD =  std::vector<double>;

  // command  line  arguments
  const size_t n =  std::atoi(args[1].c_str());     //# of samples  per  region
  const size_t Nreg =   std::atoi(args[2].c_str());     //# of regions
  const size_t ll =  std::atoi(args[3].c_str());     //2^l+1 is the grid

  // setup  grid  info
  const size_t gridSize = getGridPts(ll);
  const size_t numInnerPts = gridSize-2;
  const double dx = getSpacing(gridSize);

  // AHC  for  storing  samples ' results
  auto data = initial_access_collection<vecD>("Data", index_range=Range1D<size_t>(Nreg));

  // Launch  PDE  solves
  create_concurrent_work<Running>(n, dx, numInnerPts, data, index_range=Range1D<size_t>(Nreg));

  // Collect  results
  create_concurrent_work<Collect>(n, data, index_range=Range1D<size_t>(Nreg));
}
DARMA_REGISTER_TOP_LEVEL_FUNCTION(darma_main_task);
```

Figure 5.42: Main function for the Monte Carlo code test in DARMA.

current code allows to explore all regimes by evaluating the interplay between number/cost of each sample, and overhead of the API abstractions.

Figure 5.44 shows strong scaling results obtained on the Haswell partition of Mutrino for a MC test where we vary the number, n, of PDE solves per index within a `create_concurrent_work`, and fix the grid size of each PDE to be $4,194,305$ grid points. The full problem size is fixed to take a total number of samples $N = n * 2880$, where 2880 is total number of threads (96 nodes $\times 30$ threads/node). Each PDE is solved using a direct method to solve the corresponding discretized linear system. This implies that the cost of each solve is proportional to the grid size, which again is $4,194,305$ grid points. The plot shows excellent scaling, with some cache/memory effects surfacing when n becomes sufficiently large. Some interesting future analysis will be to replace the solver from a direct to iterative. This will pose challenges for cases where the parameters of the PDE can impact the time to convergence. This can happen because, for some problems, the linear system can be stiff for some sampled values of the parameters. Hence, strategies like load balancing, work stealing and speculative execution will play a key role.

```
struct Running {
   void operator()(
          Index1D<size_t> index,
          size_t n,  double dx, size_t numInnerPts,
          AccessHandleCollection<vecD, Range1D<size_t>> ahcdata)         const
   {
     const size_t me = index.value;
     const size_t cntxtSize = index.max_value + 1;
     //--------------------------------
     auto vecH =  std::vector<AccessHandle<  double>>(n);
     for (auto & it : vecH){
       it = initial_access<   double>();
     }
     // schedule  solves
     RandNumGen rng;
     for (size_t i = 0; i < n; ++i)
     {
       auto singleH = vecH[i];
       const double sourceRand = rng.drawStandardGaussian();
       create_work([=]{
         // solve  PDE  and  generate  QoI
         // ...
         // store  QoI
         singleH.set_value(...);
       });
     }
     auto myDataH = ahcdata[index].local_access();
     create_work([=]{
       myDataH->resize(n,0.0);
       for (size_t i = 0; i < n; ++i){
         myDataH.get_reference()[i] = vecH[i].get_value();
       }
     });
   }//op
};
```

Figure 5.43: Code for the `Running` functor used in the Monte Carlo test application.

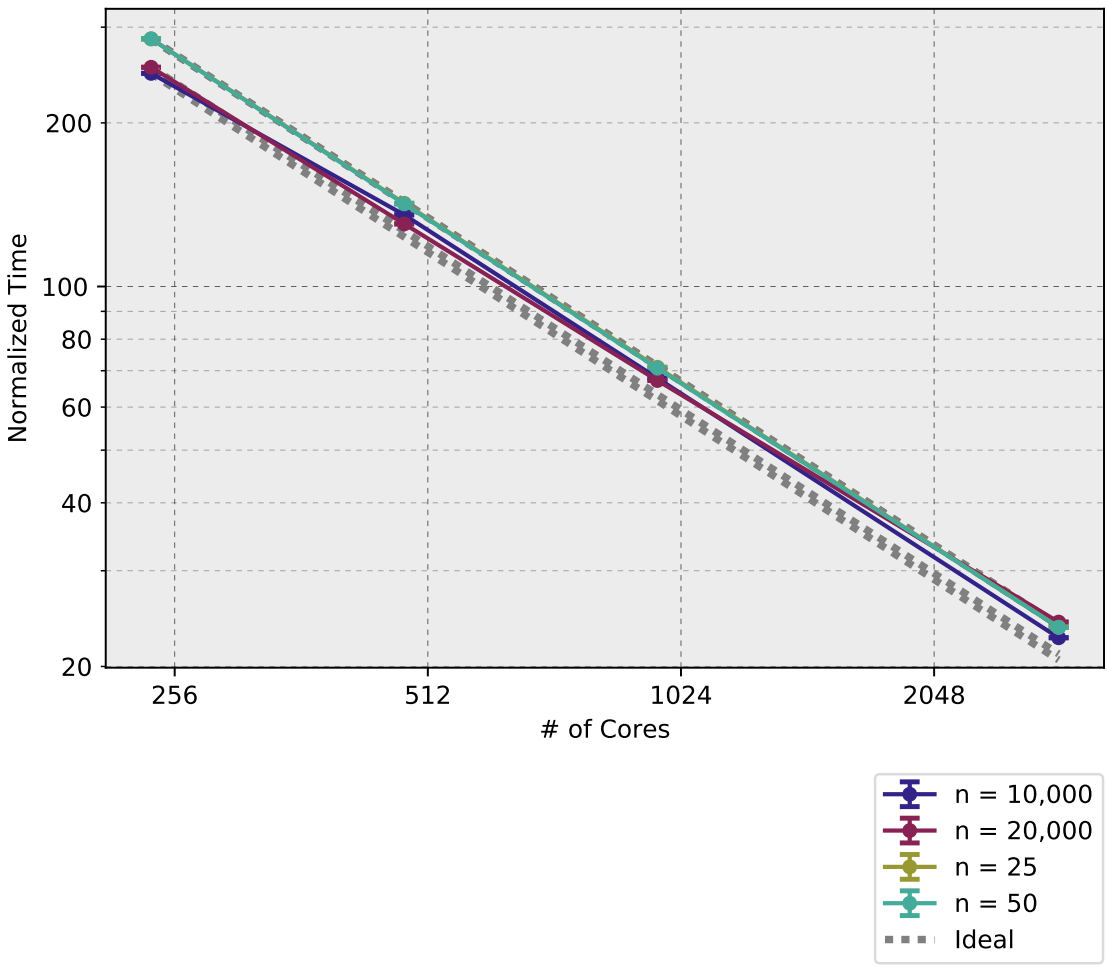


Figure 5.44: Strong scaling for a Monte Carlo test problem on Haswell for various number n of samples per DARMA index.

5.6.3.2 Multi Level Monte Carlo Example

As a test case for UQ in DARMA, we choose a MLMC solver for a 1D Stochastic PDE defined as follows:

$$\begin{cases} (1+\xi_1 x)\frac{\partial^2 u}{\partial x^2} + \xi_1 \frac{\partial u}{\partial x} = -50\xi_2^2, & \text{in } (0,1) \\ u(0) = 0 \text{and} \quad u(1) = 0, \end{cases} \tag{5.3}$$

where ξ_1 is a uniformly distributed random variable, and ξ_2 is a standard Gaussian random variable. Given these two uncertain parameters, the solution u is also a random variable. We are interested in the integral of the solution over the domain $Q = \int_0^1 u(x)dx$. The multilevel implementation involves a geometric sequence of grids, where at each level ℓ the grid cell has size $h_\ell = 1/2^\ell$, and we thus have $2^\ell + 1$ grid points. Specifically, for any level, the discrete problem is solved with second-order finite differences using a Thomas algorithm to solve the tridiagonal linear system. Since we are using a direct method for each PDE solve, the cost is proportional only to the grid size. The QoI is then computed using a trapezoidal rule.

```
1  void darma_main_task(  std::vector<std::string> args)
2  {
3    auto vLevelsH = initial_accesss<    std::vector<Level>>();
4    create_work<initialize>(vLevelsH, ...);
5
6    auto converged = initial_accesss<    bool>();
7    auto iter = initial_accesss<uint>();
8    create_work([=]{
9      converged.set_value(   false);
10     iter.set_value(1);
11   });
12
13   create_work_while([=]{
14     return converged.get_value()==    false && iter.get_value()<=maxIter;
15   }).do_([=]
16   {
17     // loop  over  level  and  schedule  /collect  samples
18     for (auto & lev : vLevelsH)
19       create_concurrent_work<runFunctor>(lev, ...);
20
21     // compute  stats , new  # of  samples , check  convergence
22     create_work<checkStats>(vLevelsH, converged, ...);
23     iter.get_reference()++;
24   });
25
26   // compute  estimator
27   create_work<MLestimator>(vLevelsH, ...);
28   create_work([=]{
29     cout << " ML Value = " << mlmcEst.get_value() << endl;
30   });
31 };
32 DARMA_REGISTER_TOP_LEVEL_FUNCTION(darma_main_task);
```

Figure 5.45: Skeletonized version of the main function for the MLMC test application.

For brevity, we do not report the full code but only show the skeleton version of the main function in Figure 5.45. For this test application, a class called `Level` is used to store all the information (e.g. cost, number of samples to run, metrics) about a level. A vector of levels is initialized first, and then passed around for work scheduling. The MLMC loop is run until convergence using the `create_work_while` API.

Within its scope, there is a main loop over the levels that spawns a `create_concurrent_work` for each level to schedule all the solves that need to be run for it. The `create_concurrent_work` API is used here similarly to how it was used for the MC application, namely by running multiple solves for each DARMA index. This is the first draft of the code, but not the optimal. The reason is that while for the coarsest levels the number of samples is large and all have a small execution time, for the finest levels the runtime can be large enough that each region within the `create_concurrent_work` would need to run a single solve rather than many of them. These features pose the need to differentiate how we schedule the samples based on the level. After all the solves are done for the current iteration, then the statistics are collected and metric evaluated.

Figure 5.46 show strong scaling results for KNL and Haswell. The problem for KNL is setup such that the width of the create concurrent region is 8320. This number was chosen such that it is divisible by the total threads running. The coarsest level has $2^{12}+1$ grid points, tolerance for convergence is set to 0.001, initial number of levels is 5 and the initial number of samples for each level is 8320. This translates into each DARMA index doing initially one sample per level during the first interaction. The problem for Haswell is setup such that the width of the create concurrent region is 7200, the coarsest level has $2^{12}+1$ grid points, tolerance for convergence is set to 0.001, initial number of levels is 5 and the initial number of samples for each level is 7200. This translates into each DARMA index doing initially one sample per level during the first interaction. The scaling is excellent for KNL, and a slightly off the ideal trend for Haswell. Since we were only able to do a single run, more analysis is needed to identify the causes behind this trend.

To build on top of this, several interesting things can be explored. First, tackling a more complex and challenging problem where the impact of the problem parameters is stronger leading to bigger variability and longer time to convergence. Second, changing the solver used for each PDE from direct to iterative. This situation will be interesting to evaluate within DARMA in terms of load balancing and speculative execution. One can envision using physical information to inform the backend of possible differences in the execution time. This information can then be taken into account by the backend to schedule and map the tasks more properly. Third, data re-usability can become important for these problems. One can envision leveraging the result from PDE samples that execute quickly to speed up the execution of similar and subsequent tasks. In this scenario, it is important to evaluate the compromise between data movement and execution time.

5.6.4 Findings and Feedback

The UQ proxy was defined to explore the benefits of an AMT approach in the context of embedded UQ analysis techniques. Two scenarios have been explored: Monte Carlo and Multi Level Monte Carlo.

Performance Scaling studies with the simple Monte Carlo case study show excellent scaling on the Haswell partition, with some cache/memory effects degrading performance when the number of PDE solves per index within a `create_concurrent_work` becomes large. For the MLMC case study we find excellent scaling on KNL, and scaling degradation on Haswell. Further study is required in this scenario to determine the cause of the performance degradation at larger scales.

Productivity In the following we present feedback regarding various aspects of the DARMA API, with regards to the UQ use cases.

`create_work` : Given the granularity requirements of the tasks that are "optimal" for being handled by the

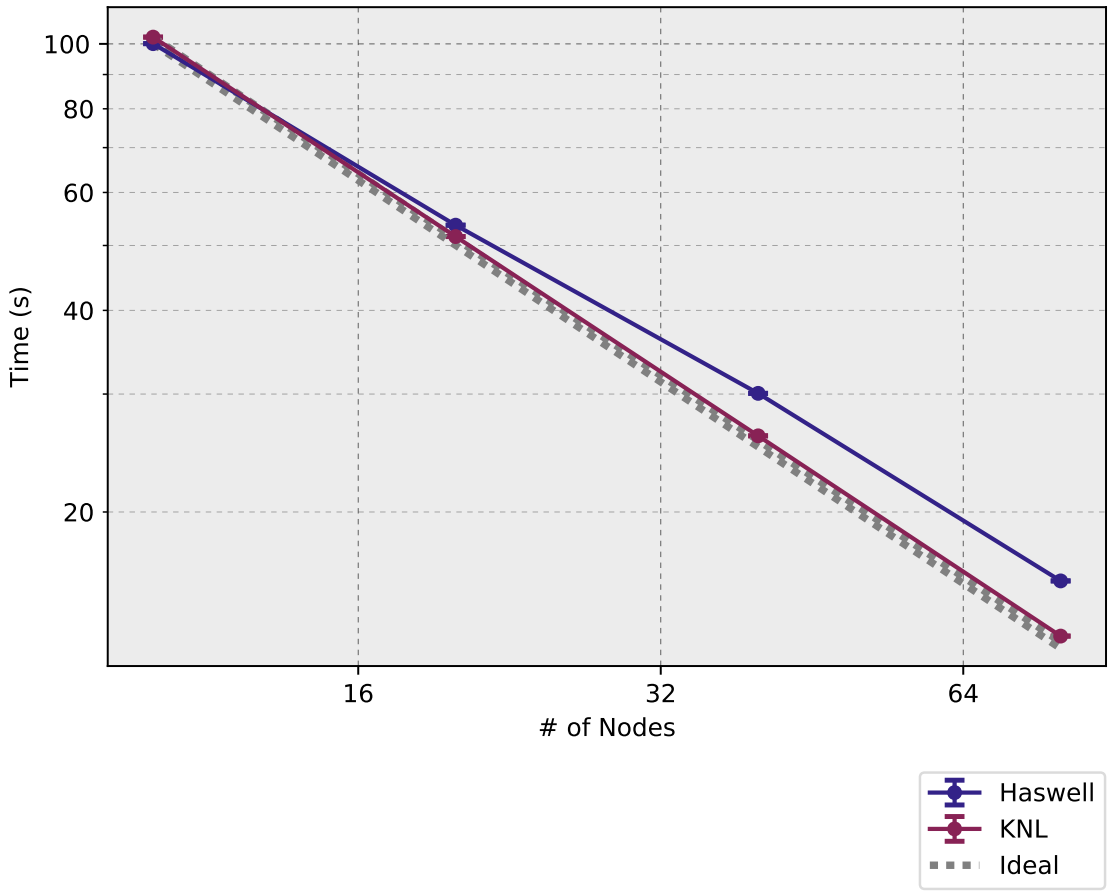


Figure 5.46: Strong scaling for MLMC test problem on both KNL and Haswell.

backend, this API has been used within the UQ development to schedule a single PDE solve. One of the main benefits of this API functionality from a UQ perspective is the ability to nest multiple `create_work` together. This benefits step-by-step approaches of UQ developers where users first write a draft of the app with a target granularity, for example one `create_work` for one PDE solve. Based on the performance of this approach one might then break down each sample into further nested tasks of finer granularity to achieve a better performance/efficiency. This feature is not only useful for UQ problems where many independent tasks are being scheduled, but also for writing/optimizing any application.

`create_concurrent_work` /`access_handle_collection` : For UQ purposes, this API is useful and intuitive for expressing distributed work. It provides a natural way to distribute tasks/sampling tasks without having to worry about the nature and types of each group of tasks and the level of heterogeneity/homogeneity. It is intuitive and its use is flexible because it is templated on the mapping/distribution of data to workers. This framework is thus ideal for any user that has specific needs of mapping. The interplay between this API and `create_work` is somewhat intuitive too. Within each execution index, one schedules work as needed. These scheduled tasks are then mapped in a certain way by the backend to run efficiently. The use of publishing and fetching across the indices is a great tool to express data exchange in an asynchronous way. One feature that is missing that would be useful is the concept of nested `create_concurrent_work` . This is already in the plan for future work and will be a useful addition to the API list.

The idea of the `access_handle_collection` is very useful. It provides a suitable abstraction for distribut-

ing data. The main feature of it is the flexibility of mapping the data according to a user-defined mapping structure. Currently, only Fortran provides an intrinsic data structure for distributed arrays. However, this is still within the context of CSP programming models. C++ does not have intrinsic functionalities for "native" parallel programming features, and typically users resort to MPI or other libraries like Trilinos to have access to these kind of distributed abstractions.

`access_handle` : Originally it was somewhat difficult and very verbose to use because of the need to provide a unique label for each piece of data created. This was very difficult to carry over and use especially for complex codes with lots of pieces. One can envision a large production code with multiple people working on it, where it would be extremely difficult to have a setting where labels need to be created uniquely. This obstacle has been removed, which makes the framework much more agile and flexible.

Future Work This year the UQ proxy application work was an "exceeds criteria" for the milestone and, thus, the analysis performed was not as in depth as for PIC or the benchmarks. Consequently, we will perform more in depth analysis with the UQ proxy use cases next fiscal year. The use cases will evolve and become more complex, in particular moving towards implicit solvers. Runtime system support for data-reusability and load balancing are key features that will be explored within our upcoming studies.

5.7 Multiscale Proxy

5.7.1 Background and Relevance to Sandia's ATDM Program

Computational solid mechanics comprises a significant fraction of the simulations carried out for Sandia's core mission of ensuring the safety and surety of nuclear weapons. The primary governing equation is the balance of linear momentum, applied in combination with constitutive laws that govern material behavior. Solutions are obtained using the finite element method on conformal, nonuniform meshes that deform over the course of the simulation (i.e., Lagrangian meshes). The simulation of events that are short in duration and dominated by dynamic effects is typically carried out using explicit time integration, while implicit dynamic or quasi-static approaches are applied to model slower events and those in which dynamic effects can be neglected. The ASC IC code for solid mechanics is *Sierra/SolidiMechanics*, which contains a rich feature set including a comprehensive material model library, a variety of finite element formulations, and the ability to model contact [41].

The ATDM multiscale technology demonstrator described in this section was developed to investigate the application of DARMA to an engineering analysis code that is representative of *Sierra* ASC IC codes. At its core, the multiscale technology demonstrator is a Lagrangian finite element code for the solution of dynamic mechanics problems on nonuniform meshes. A multiscale modeling strategy was implemented in addition to the standard, single-scale approach to evaluate DARMA for the management of high-cost, high-fidelity models that can lead to a large load imbalance. For the purpose of comparison, several versions of the multiscale technology demonstrator were developed in which different parallelization strategies were employed, including both a DARMA version and a pure MPI version. A serial version was also developed to provide a baseline with no overhead for parallel communication.

5.7.2 Multiscale Proxy Design

Mechanics problems solved by the multiscale technology demonstrator are governed by the well-known equations of motion, given in index notion by

$$\rho \ddot{u}_i = \sigma_{ij,j} + \mathrm{b}_i, \tag{5.4}$$

with displacement and traction boundary conditions given by

$$\begin{aligned} u_i &= g_i \qquad \text{on} \qquad \Gamma_{g_i} \\ \sigma_{ij} n_j &= h_i \qquad \text{on} \qquad \Gamma_{h_i}, \end{aligned}$$

and initial conditions given by

$$\dot{u}_i(x,t) = \dot{u}_{i_o}(x) \qquad \text{at} \qquad t = 0.$$

Here, u denotes displacement, $\dot{u}$ is velocity, $\ddot{u}$ is acceleration, $()_{,j}$ indicates the spatial derivative $\frac{\partial()}{\partial x_j}$, ρ is density, σ is stress, b is an external body force, Γ_g and Γ_h are the domain boundaries over which displacement and traction boundary conditions are applied, respectively, and $\dot{u}_o(x)$ are initial velocity values defined over the domain as a function of position, x.

The weak form of Equation 5.4 is solved by the technology demonstrator via the standard finite element method, which requires discretization in both time and space [42, 43]. Time integration may be achieved

using either implicit schemes for quasi-statics and implicit dynamics, or using explicit schemes for explicit dynamics. The version of the multiscale tech demonstrator that utilizes DARMA is restricted to explicit dynamics. This restriction eliminates the need for the solution of a global system of equations, which is required for implicit schemes.

The standard Velocity-Verlet approach is applied for explicit time integration. The simulation is advanced from time step n to time step $n+1$ by determining the nodal positions, $\mathbf{u}$, velocities, $\dot{\mathbf{u}}$, and accelerations, $\ddot{\mathbf{u}}$, as follows,

$$\begin{aligned}
\dot{u}_i^{n+\frac{1}{2}} &= \dot{u}_i^n + \frac{1}{2}\Delta t\, \ddot{u}_i^n \\
u_i^{n+1} &= u_i^n + \Delta t\, \dot{u}_i^{n+\frac{1}{2}} \\
\dot{u}^{n+1} &= \dot{u}_i^{n+\frac{1}{2}} + \frac{1}{2}\Delta t\, \ddot{u}_i^{n+1}.
\end{aligned}$$

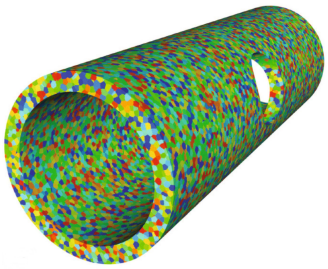
Advancing the simulation through time requires the calculation of the nodal accelerations, $\ddot{\mathbf{u}}$, via Equation 5.4. This calculation is nontrivial and comprises the vast majority of computation expense for the simulation.

Explicit time integration is conditionally stable, requiring that the time step Δt be no greater than the maximum stable time step, $\Delta t_{\max}$. The maximum stable time step is a function of the discretization (element size) and the material properties (wave speed). Due to the restriction on the maximum allowable time step, explicit time integration generally requires a large number of small time steps. Code execution for explicit dynamics is comprised mainly of evaluation of the internal force at each time step, which necessitates a vector reduction for calculation of the net force acting on nodes that lie on partition boundaries.

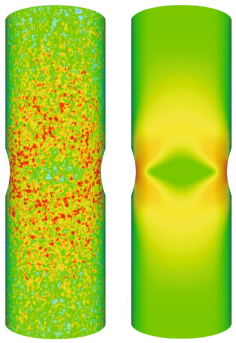
5.7.2.1 Multiscale capability

The resolution of small-scale heterogeneities in predictive simulations is challenging due largely to computational expense. The resolution of material variability at the mesoscale, which can play a critical role in the process of component failure, is generally computational intractable because the salient length scales are orders of magnitude smaller than the engineering length scale. Recent work by Bishop, et al., demonstrated the need to resolve fine-scale mechanisms [1, 2]. Figure 5.47 illustrates the effect of mesoscale variability on the stress field for a cylinder with a hole pulled in tension. Resolution of the crystalline grain structure reveals a stress field that is significantly less smooth than that predicted by a homogenized model.

Multiscale methods offer the ability to model the effects of fine-scale mechanisms at a fraction of the cost of direct numerical simulation. This is achieved by the multiscale technology demonstrator using the so-called FE^2 multiscale strategy [44–47]. In this approach, sub-models that resolve the fine scale are associated with material points in the macroscale model. The fine-scale model is provided by a representative volume element (RVE), as illustrated in Figure 5.48. Kinematic information determined at the macroscale (e.g., deformation gradient) provides boundary conditions for the lower-scale RVE model. The RVE model is then solved as an independent finite element problem, and the resulting stress field is used to determine a homogenized stress that is passed back to the macroscale model. The FE^2 approach was included in the multiscale technology demonstrator partially because it is naturally compatible with next-generation, heterogeneous computing platforms [48, 49]. DARMA has the potential to effectively manage the complex computational demands associated with solving both the macroscale model and a set of largely independent sub-models at the fine scale.

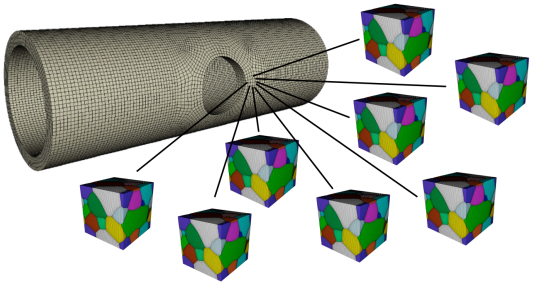


(a) Explicit modeling of a grain structure in an engineering component for direct numerical simulation.

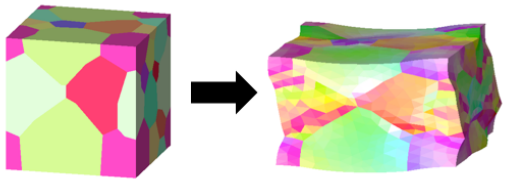


(b) Stress values resulting from a fully-resolved grain structure (left) and from the application of a homogenized constitutive model (right).

Figure 5.47: The effect of microstructure on material response motivates the development of multiscale methods [1, 2].



(a) Representative volume elements (RVEs) are associated with material points of interest in the macroscale model.



(b) Kinematics determined at the macroscale are imposed on an RVE in combination with periodic boundary conditions.

Figure 5.48: Illustration of the FE^2 multiscale method.

5.7.2.2 Software Implementation Strategy

The multiscale technology demonstrator was designed specifically to facilitate an investigation of DARMA and other emerging ATDM technologies. It was written from scratch in C++ and makes minimal use of third-party libraries. Data structures and the procedures that act on them follow a compartmentalized design such that DARMA can effectively optimize code performance, for example through work migration and asynchronous execution. DARMA requires that data structures be serializable. This is a significant design constraint, and while it was relatively straightforward to adhere to when writing new software, it would be much more difficult in the presence of legacy code and objects defined by third-party libraries. Further, the DARMA version of the technology demonstrator has no dependency on MPI. As of July 2017, DARMA does not support interoperability with application-level MPI, which would make current integration with current ASC IC codes, and many Trilinos components, difficult or impossible. Enabling simultaneous use of DARMA and application-level MPI is planned for FY18.

High-level pseudocode for the multiscale technology demonstrator is given in Algorithm 3. Each of the tasks in the pseudocode is called via a DARMA `create_concurrent_work` command. The data that the tasks act upon is managed using DARMA access handle collections. The access handle collections are listed in Table 5.4.

Algorithm 3 Code flow for the multiscale technology demonstrator. The routines listed were implemented as functors and executed in separate DARMA `create_concurrent_work` blocks, allowing for work migration and, where dependencies allow, asynchronous execution.

```
1:  InitializeDataManager
2:  ReadGenesisFiles
3:  IdentifyGloballySharedNodes                  ▷ communication among multiple ranks
4:  InitializeBoundaryConditionManager
5:  InitializeExodusOutput                       ▷ calls third-party library
6:  ComputeLumpedMass
7:  VectorReduction(LumpedMass)                  ▷ communication among multiple ranks
8:  ComputeCriticalTimeStep                      ▷ communication among multiple ranks
9:  ApplyInitialConditions
10: ApplyKinematicBC
11: ComputeDerivedElementData
12: ExodusWriteStep                              ▷ calls third-party library
13: for each time step do
14:     ExplicitTimeStep
15:         calls ApplyKinematicBC
16:         calls VectorReduction(InternalForce) ▷ communication among multiple ranks
17:     if output step then
18:         ComputeDerivedElementData
19:         ApplyKinematicBC
20:         ExodusWriteStep                      ▷ calls third-party library
21:     end if
22: end for
```

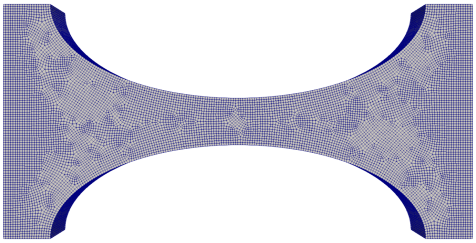
Table 5.4: Multiscale technology demonstrator classes that can be serialized for use with DARMA.

Class	Purpose
GenesisMesh	Input mesh data, file reading
ExodusOutput	Output data, file writing
DataManager	Node data, element data
Block	Material model, element calculations
DerivedElementData	Secondary data for output
BoundaryCondition Manager	Boundary conditions, node sets
BoundaryCondition	User-defined data for a boundary condition
LinearSolver	Vector, matrix, and (serial) solver for RVE sub-models

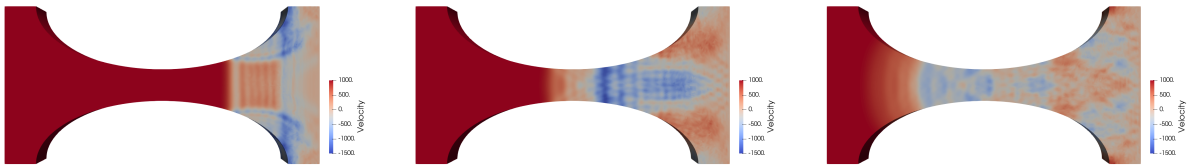
5.7.3 Preliminary Results

5.7.3.1 Single-Scale Simulation

A single-scale wave propagation simulation carried out with the technology demonstrator is presented in Figure 5.49. In this simulation, a notched plate is modeled with a Neohookean material model and given a uniform initial velocity in the horizontal direction. A fixed-displacement boundary condition is applied to the right-hand face, which approximates the effect of the plate impacting a rigid body and produces a compression wave. The finite element discretization of the plate contains roughly five million elements and is shown in Figure 5.49a. The predicted wave propagation through the body of the plate is given in Figures 5.49b-5.49d.



(a) Finite element discretization.



(b) Velocity at time = 5 μs.

(c) Velocity at time = 10 μs.

(d) Velocity at time = 15 μs.

Figure 5.49: Simulation of wave propagation in a notched plate.

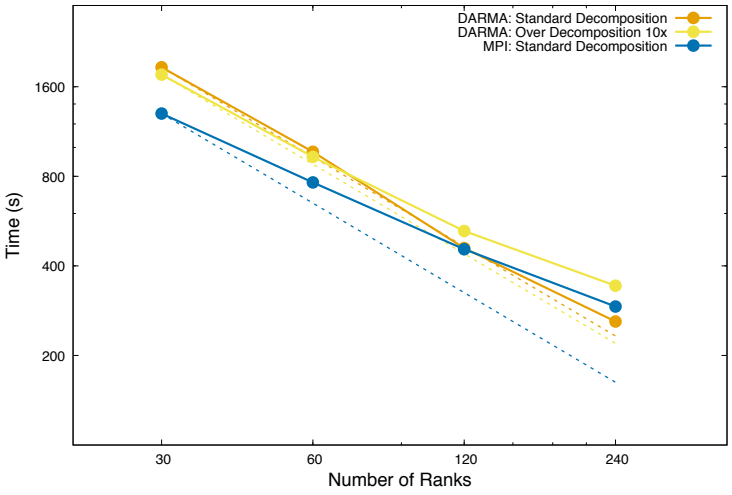


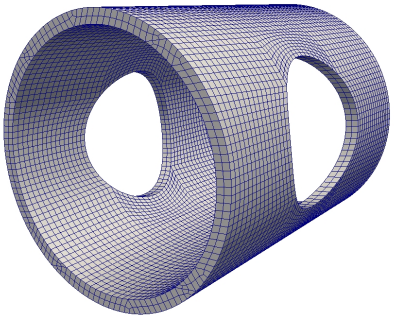
Figure 5.50: Preliminary strong scaling results for the simulation of wave propagation in a notched plate.

Preliminary timing data for solution of the single-scale wave propagation problem were obtained under three sets of conditions. The first two sets of data were obtained using the version of the technology demonstrator that utilizes DARMA. In one case, a standard parallel decomposition was used in which the number of mesh partitions and virtual ranks equals the number of physical computational ranks. In the second case, an over-decomposition strategy was used in which the number of mesh partitions and virtual ranks equals ten times the number of physical computational ranks. The third set of data was obtained using the version of the technology demonstrator that utilizes MPI for parallel communication (i.e., DARMA is not used). The timing data are presented in Figure 5.50. The DARMA version of the technology demonstrator shows good scaling and roughly equivalent performance to the MPI version, with a maximum relative difference in run time of 25%. Further, timing data for the standard-decomposition and over-decomposition cases are in close agreement for 30 and 60 ranks, with the over-decomposed case showing a modest slowdown for larger numbers of ranks.

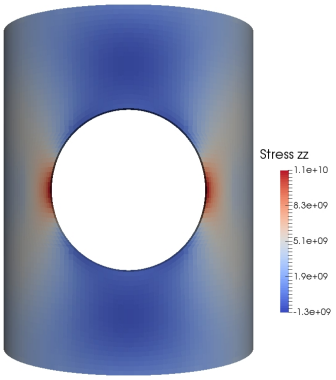
5.7.3.2 Multiscale Simulation with Load Balancing

A FE^2 multiscale simulation used to investigate the performance of the technology demonstrator with DARMA-enabled load balancing is presented in Figure 5.51. A cylinder with holes is loaded in tension in the longitudinal direction, resulting in stress concentrations in the vicinity of the holes. A multiscale constitutive model was applied in the high-stress regions, as illustrated in Figure 5.51c. A standard, geometry-based partitioning scheme results in significant load imbalance owning to the fact that the multiscale constitutive model is much more computationally expensive than the single-scale model. The degree of load imbalance depends on the nature of the RVE sub-problems associated with the material points in the multiscale domains. For the purpose of investigating DARMA in the presence of load imbalance, the macroscale domain was discretized with roughly twelve thousand elements and the RVE model was discretized with sixty elements (see Figures 5.51a and 5.51d). The microscale RVE model consists of a matrix phase and a fiber. While the RVE is not sufficiently resolved to be statistically representative of a two-phase material, it does provide sufficient computational complexity for an evaluation of DARMA in the presence of load imbalance.

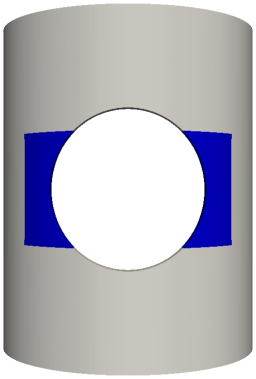
Timing data for a set of multiscale simulations are presented in Figure 5.52. As with the single-scale simu-



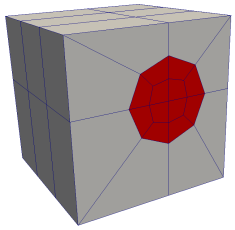
(a) Macroscale finite element discretization.



(b) Stress concentrations are present in the vicinity of the holes, motivating the use of a high-fidelity model in these regions.



(c) Multi-domain strategy; a high-fidelity multiscale model is applied in the highlighted subdomains.



(d) Microscale finite element discretization of a two-phase material (matrix and fiber) applied at each integration point in the multiscale domain.

Figure 5.51: Simulation of a cylinder with holes under tensile loading. The application of a multiscale model in the regions highlighted in (c) leads to a large computational load imbalance.

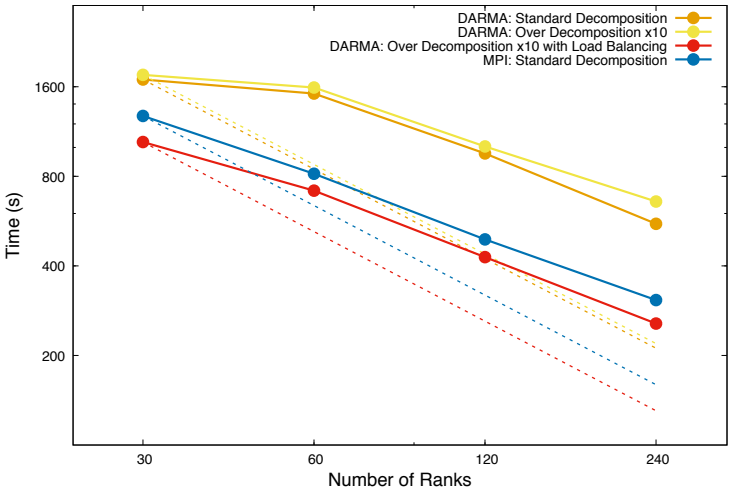


Figure 5.52: Preliminary strong scaling results for the simulation of cylinder with a hole under tensile loading. The simulation utilizes a computationally-expensive multiscale constitutive model in roughly 12 percent of the domain, creating a large load imbalance.

lation, the DARMA version of the technology demonstrator was exercised on a standard decomposition in which the number of parallel partitions equals the number of physical computational ranks, and on an over decomposition in which the number of parallel partitions equals ten times the number of physical ranks. The over-decomposed case was evaluated both with and without the load-balancing capabilities of DARMA. For the purpose of comparison, the standard decomposition was also evaluated with the MPI version of the technology demonstrator. The DARMA version of the code performed similarly for the standard decomposition and the over decomposition without load balancing. In these cases, scaling was poor and the overall run time was longer than that required by the MPI version of the code. The best performance, however, was achieved by the DARMA version of the code with load balancing enabled. Load balancing, in which a number of the over-decomposed partitions are redistributed to maximize the utilization of physical resources, greatly improved both scaling and absolute performance.

5.7.4 Findings and Feedback

The Multiscale proxy was developed to explore the benefits of an AMT approach in the context of Multiscale applications important to Sandia's ASC IC program. Similar to the UQ proxy, this was also an "exceeds" criteria for this milestone and will be the focus of more in-depth study next fiscal year, particularly with regards to performance studies. Nonetheless, preliminary findings from the application developer perspective, based on code development activities are captured herein.

Adoption of DARMA for the multiscale technology demonstrator required a shift in developer mindset and a revamping of conventional architecture for engineering analysis codes. Engineering codes are generally procedural in nature, owning largely to the fact that are an implementation of a solution strategy for an underlying mathematical model that is itself procedural. Moving toward an AMT runtime model is best achieved by conceptualizing the application software as a set of tasks with well-defined dependencies. This declarative approach facilitates a programming model in which tasks are defined by the application developer, and the specifics of their execution are a responsibility of the runtime. This has both significant advantages and significant costs. The primary advantages are improved runtime performance, for example

resulting from work migration and asynchronous execution, and the ability of the application developer to delegate many aspects of performance tuning to domain experts working within the bounds of the runtime code. The primary cost associated with adopting DARMA is the need to fully encapsulate application-level routines and data structures to facilitate AMT scheduling and task management. Further, the inherent "black box" nature of an AMT runtime complicates testing, debugging, and maintenance of application codes.

A listing of the most significant advantages and costs of adopting the DARMA runtime model for the multiscale technology demonstrator are summarized below:

Benefits

- Parallelization and task execution are managed by DARMA, potentially enabling software performance across a large number of diverse hardware platforms.
- Design and implementation of application code is carried out at the task level. This allows for a higher-level conceptualization of the code architecture and the delegation of many aspects of code performance to the runtime.
- Elements of code performance that are traditionally very difficult to achieve at the application level, e.g., work migration and asynchronous execution, are enabled automatically by adhering to the DARMA programming model.

Costs

- The task-based programming model requires encapsulation and serialization of data structures and procedures, which has a profound impact on code design.
- Debugging is complicated by increased potential for non-repeatable bugs and by the opaque "black box" nature of DARMA and the backends.
- The current (July 2017) version of DARMA has limited compatibility with application-level MPI and third-party libraries.
- Code development, debugging, performance tuning, and long-term maintenance requires close collaboration with DARMA domain experts.

This page intentionally left blank.

Chapter 6

Conclusions and Recommendations

This report presents a summary and in depth analysis of DARMA and a DARMA-compliant AMT software stack. Herein we focus our examination in three areas 1) performance and productivity gains in the context of proxy applications and benchmark use cases relevant to Sandia's ASC/ATDM program, 2) generality of DARMA's backend API, and 3) interoperability of DARMA with regards to node- and network-level libraries.

Performance and Productivity Performance experiments up to 2K nodes on Trinity demonstrate the scalability of the DARMA-CHARM++ backend. Although DARMA's deferred execution and task model imposes overheads over imperative MPI implementations (typically 10-20% for balanced use cases), the performance gains for irregular, load-imbalanced problems are significant. From a productivity perspective, DARMA's declarative backend specification enables three capabilities to be managed by the DARMA-CHARM++ backend: 1) communication/computation overlap, 2) tunable granularity in which data decomposition does not necessarily match the execution resources, and 3) load balancing. Based on application developer feedback stemming from Sandia's FY15 ASC/ATDM Level 2 Milestone #5325, DARMA's frontend enables a deferred execution model via sequential semantics. Specifically, the user writes code (annotated with DARMA abstractions for data and tasks) that can be reasoned about as though it were run sequentially. DARMA then inserts the necessary dependency capture hooks at compile-time to support run-time dependency capture and dependency analysis. Although productivity is difficult to measure quantitatively, application-developer feedback regarding frontend abstractions is positive, indicating the approach does facilitate transition from imperative to declarative programming styles.

Generality of Backend API By design, DARMA captures a declarative representation of an application that can map to a variety of runtime implementations. However, this year's milestone highlighted opportunities for componentization within both the frontend and backend that could result in improved scheduling and optimization strategies long-term. The current implementation of DARMA combines capture hooks, dependency capture, and dependency analysis into a single "translation layer" between a frontend API and backend API, implemented as a header-only C++ library. However, through our initial implementation effort, we see a natural decomposition into three layers and future development of the DARMA frontend will explore splitting this functionality more cleanly into distinct software components. Such a componentization strategy lends itself to supporting shifts in responsibilities between frontend and backend, allowing backends to receive type-specific information in the future, to enable, *e.g.*, serialization optimizations or type-optimized collectives. Our experiences this year have highlighted opportunities for backend componentization as well. In particular, as the DARMA-CHARM++ backend was built, we discovered that a few subsets of the backend interface had a very direct mapping to CHARM++, but others required significant implementation effort due to the semantic disparity between a heavily data-centric model (DARMA) and a

message-driven, concurrent object model (CHARM++). A primary example of this is collectives. A backend component that matches DARMA semantics for collectives, but is runtime-agnostic (*e.g.*, only requires implementing a `send` function) could bridge the gap between the frontend and other backends with similar semantic disparities. Such a component may even be universally applicable beyond DARMA, improving runtimes as the broader community develops best practices and collectively develops general libraries for HPC.

Interoperability Interoperability was one of the more crucial challenges identified over the course of this milestone. Although the challenge of interoperability between node-level and network-level libraries exists within current software stacks, the asynchrony of AMT runtimes raises additional interoperability challenges. For example, similar to other AMT frameworks, CHARM++ is centered around explicitly managed Pthreads. In particular, difficulties must be managed when using, *e.g.*, the OpenMP affinity layer for resource management and arbitration between the AMT runtime and node-level libraries. We made progress with the DARMA-ONNODE backend, demonstrating interoperability with KOKKOS via the OpenMP affinity layer. The distributed runtime case will play a central role in next year's research and development efforts.

Summary and Recommendations Together with the initial DARMA-compliant stacks that have been developed, the accumulated information in this report highlights the potential of the overall DARMA approach while clearly identifying focus areas for future work. These include productizing and hardening efforts such as productivity tools (*e.g.*, timers, performance profilers, debugging aides), tuning of critical overheads (which can be ameliorated in part via optimizations in collectives and certain task scheduling operations), general documentation, and testing. Focused research and development on interoperability with node-level libraries is crucial and will play a major role in next year's efforts. Although DARMA is geared towards dynamic task graphs and irregular problems, further study is required to delineate concretely between classes of applications that will and will not benefit a DARMA approach. We know that DARMA overheads can be ameliorated in part via optimizations in collectives and certain task scheduling operations. Furthermore, as backend runtimes mature, they will likely be better able to make use of the lookahead provided by the DARMA approach, benefiting both regular and irregular use cases. Lastly, through our design and implementation efforts this year, we have identified opportunities for componentization that would enable optimization strategies and, perhaps more importantly, increase the participatory nature of the DARMA architecture. Enabling and garnering broader community engagement would facilitate development of AMT best practices, which are ultimately required for vendor-supported solutions long-term.

Chapter 7

Appendix

7.1 Frontend Abstractions

The application-facing portion of DARMA (referred to as the *frontend* in this document) is designed with the primary purpose of addressing application developer needs and maintaining the necessary agility to adapt to those needs without requiring significant backend code changes. Most application developer needs can be categorized as *correctness* concerns or *performance* concerns, and the separation of these two categories of concerns is a fundamental design focus of the DARMA frontend.

Fundamental Design Principle

The DARMA frontend should create a natural and intuitive separation of correctness and performance concerns.

Whenever possible, application code addressing these concerns should be expressed with different syntax. The user should be able to write code that gives a well-defined result but is generic with respect to performance hints and directives (and this result should be independent of such hints and directives).

DARMA leverages this separation provide *performance portability* across a variety of current and future architectures.

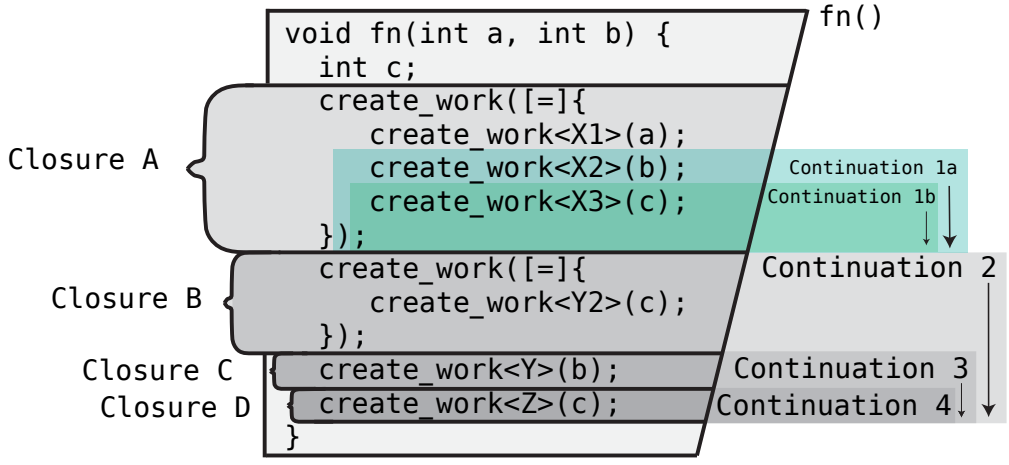


Figure 7.1: Closure and continuation.

7.1.1 Abstractions

A program in DARMA consists of a sequence of *asynchronously executable closures* (AECs), potentially nested to an arbitrary depth, and sets of *asynchronously accessible objects* (AAOs) that these closures *capture*. To understand this statement, we need to define some terms:

Definition

asynchronously executable closure (AEC)

An ordered set of operations using a set of asynchronously accessible objects with well-defined *access permissions* to each of these objects. The most common syntaxes for creating AECs in DARMA are `create_work` and `create_concurrent_work` .

Definition

asynchronously accessible object (AAO)

A representation of a piece of application data with well-defined relationships (usually strictly-non-aliasing) to data represented by other AAOs in the program. The most common syntaxes for creating AAOs in DARMA are `initial_access` and `initial_access_collection` (which, respectively, create `AccessHandle` and `AccessHandleCollection` objects).

Definition

capture

An AEC *captures* an AAO with a given set of permissions if the AEC requires those permissions to be available (usually, available in a manner consistent with program sequence) to begin executing.

The following code sample illustrates these terms using the DARMA C++ frontend implementation:

```
1  // create an AAO with that represents data of type int:
2  auto ah = initial_access<int>();
3  // create an AEC capturing the AAO created in the previous line:
4  create_work([=]{
5      ah.set_value(42);
6  });
```

These abstractions are *declarative* in that they state *what* the program needs to execute but not *how* it should be executed. In general, it is up to the backend to determine when and where to execute the work declared in a DARMA program. However, any conforming implementation of the DARMA backend API should yield the same result when executing a given program regardless of when, where, and how its component parts are executed. AAOs provide the fundamental abstraction for *data* expression in DARMA, and AECs provide the fundamental abstraction for units of *execution*.

7.1.1.1 Distributable AAOs and Concurrent AECs

It is not always feasible, scalable, or desirable to access all of the underlying data for a given AAO in one AEC execution. For instance, when expressing a stencil computation, the user may want to subdivide the problem into pieces that perform the same computation on different portions of a very large chunk of application data (one that would, for instance, only fit in distributed memory storage for a large enough program). DARMA allows the expression of AECs that are to be executed a given number of times with different (for now, disjoint) pieces of the data from some or all of its captured AAOs. Closures that support this behavior are called *concurrent asynchronously executable closures* (CAECs). In DARMA, CAECs are primarily created using `create_concurrent_work`, and objects that can be distributed across these closures are called *distributable AAOs* (DAAOs). Each of these pieces that can be executed disjointly is called a *CAEC element*, and the piece or pieces of the DAAOs it operates on are called DAAO elements. (Inside of the closure, each DAAO element is, in turn, represented by an AAO). With some notable exceptions (mostly involving *access permissions*, as described below), DAAOs and CAECs interact with other closures AAOs and CAECs in their surrounding scope as if they were not special. The following code snippet illustrates these concepts:

```
1  // create an DAAO with that provides a coarsened representation of
2  // 1000 objects of type int:
3  auto mycol = initial_access_collection<int>(index_range=Range1D<int>(1000));
4  // declare the functor that executes an element of a CAEC
5  struct MyFunc {
6     void operator()(Index1D< int> idx, int& data);
7  };
8  // create an CAEC that captures one DAAO element of mycol per CAEC element:
9  create_work<MyFunc>(ahc, index_range=Range1D<int>(1000));
```

7.1.1.2 Access Permissions

Definition

access permissions

A succinct description of the required state of and allowed operations on a piece of data represented by an AAO to be captured by a given asynchronously executable closure. Examples of permissions in DARMA syntax include `Read`, `Modify`, or `Commutative`.

Access permissions to data represented by an AAO at any given point in the program can be decomposed into *scheduling permissions* (that is, the permissions that can be captured by a closure at the given point in the program) and *immediate permissions* (that is, the allowed operations on the underlying data itself).

A list of existing or currently planned permissions follows.

- `None`
  - As immediate permissions: No access to the underlying data is available. No pointer to the underlying data is even assigned to AAO at that time. AECs that are not CAECs always capture

DAAOs with immediate permissions of `None` since DAAOs are meant to represent data that cannot be scalably accessed in a single AEC execution.

  - As scheduling permissions: No AECs may be created that capture this AAO. When combined with immediate permissions other than `None`, this represents an access pattern sometimes referred to as a *leaf* access, since it corresponds to a leaf in the task parentage DAG.

- `Read`
  - As immediate permissions: Read-only operations may be performed on the underlying data. The data represented by the AAO is guaranteed to not change for as long as Read immediate permissions are active. Similarly, the user is required to not modify the underlying data via an AAO with `Read` immediate permissions. The DARMA C++ frontend implementation cannot enforce this at compile-time since developers can always circumvent const rules with `mutable`, `const_cast`, or shallow `const` -ness of C++ objects. Applications that break this requirement will have undefined behavior and difficult to solve bugs.
  - As scheduling permissions: AECs capturing immediate and/or scheduling permissions less than or equal to (with respect to partial ordering of permissions, as described below) `Read` may be created for this AAO.
- `Modify`
  - As immediate permissions: The data represented by this AAO may be changed, and doing so will not create data races. The state of the data upon execution of the capturing AEC is consistent with the output of the nearest sequentially preceding closure or closures[1] with immediate permissions that allow mutation of the underlying data (the permissions collectively referred to as "mutating" that are currently defined are: `Modify`, `Write`, `Commutative`, `Relaxed`, and `Combine`). See discussion below for more detail on how permissions establish the dependency structure of the program.
  - As scheduling permissions: AECs capturing immediate permissions less than or equal to `Modify` immediate and scheduling permissions less than or equal to `Modify` may created for this AAO.
- `Commutative`
  - As immediate permissions: The data represented by this AAO may be changed, and doing so will not create data races. The state of the data upon execution of the capturing AEC is consistent with the output of the nearest sequentially preceding closure(s) *o*r any "sequentially equivalent" closures (i.e., other AECs that capture immediate permissions of `Commutative` on the same version — that is, the group of `Commutative` captures between the nearest lexically preceding `Read` or `Modify` captures of an `AAO` and the nearest lexically subsequent `Read` or `Modify` captures of that same AAO).
  - As scheduling permissions: AECs capturing immediate permissions less than or equal to `Commutative` immediate and scheduling permissions less than or equal to `Commutative` may created for this AAO. Children of capturing AECs with `Commutative` permissions commute with AECs in their parent's enclosing scope as if they were in that scope.
- `Relaxed`: Similar to `Commutative` permissions, except that changing the data represented by an AAO *m*ay create data races and thus should be done atomically via some other mechanism (e.g., atomics).

[1] In the case of the unordered mutating permissions (`Commutative`, `Relaxed`, and `Combine`), there can be more than one sequentially preceding closure, and the input state of the subsequent modify closure is consistent with the output as defined by those permissions.

- `Combine` : Similar to `Commutative` permissions, except that two AECs may execute on separate copies of the same preceding state and may later be combined using a user-defined "combine operator."

The relationships between permissions for which ordering is defined are as follows (these relationships establish a partial ordering on permissions):

- `None` $<$ `Read` $<$ `Modify`
- `None` $<$ `Write` $<$ `Modify`
- `None` $<$ `Commutative`
- `None` $<$ `Relaxed`
- `None` $<$ `Combine`

All other relationships (e.g., `Read` $<=>$ `Commutative`) are unordered, and thus statements such as "*A* permissions less than or equal to *B* permissions" will be false.

How permissions are used in practice The set of permissions captured on the set of AAOs used by an AEC are used by DARMA to determine the dependency structure of the program. For instance, an AEC capturing `None` scheduling permissions and `Read` immediate permissions on a given AAO establishes a dependency on all lexically preceding AECs (and their nested descendants) that capture scheduling or immediate permissions that could mutate the value (that is, those that capture `Modify` , `Write` , `Commutative` , `Relaxed` , or `Combine`) and establishes an *anti-dependency* on any lexically subsequent AECs that request immediate permissions that could modify the value. Nested AECs maintain a dependency structure consistent with their parents' sequence as well as any other AECs nested at the same scope. For instance, consider the following code snippet using the DARMA C++ frontend syntax:

```
1  auto ah = initial_access<  int>();
2  create_work([=]{
3    // <-- Inside AEC #1
4    ah.set_value(3);
5  });
6  create_work(schedule_only(ah), [=]{
7    // <-- Inside AEC #2
8    create_work([=]{
9        // <-- Inside AEC #3
10       ah.set_value(ah.get_value() * 5);
11   });
12 });
13 create_work(reads(ah), [=]{
14   // <-- Inside AEC #4
15   printf("%d", ah.get_value());
16 });
```

The snippet above should deterministically print `15` every type it executesAEC #1 captures `ah` with the $\{scheduling, immediate\}$ permissions tuple of `{Modify, Modify}` .[2] AEC #2 captures `{Modify, None}`

[2]For the rest of this document, permissions tuples will be written in this form. If a pair of permissions are given without any context, the first is assumed to describe the scheduling permissions and the second is assumed to describe the immediate permissions.

permissions on ah, and thus does not establish a dependency on AEC #1. The backend is free to execute it before, during, or after it executes AEC #1. However, since AEC #2 is (lexically) sequenced after AEC #1, its nested AECs must maintain consistency with the sequence of AEC #2 with respect to AECs #1 and #4. Thus, AEC #3 bears a *dependency* on AEC #1 and cannot execute until AEC #1 relinquishes all permissions on ah (either implicitly, via completion of the capturing AEC at the closing curly brace, or explicitly, via a user call to ah.release()). AEC #4 is sequenced after AEC #1 and #2, and thus cannot execute as long as those closures or any closures nested therein (in this case, AEC #3) release any mutating permissions (in this case, Modify), either scheduling or immediate. While the requirement that the predecessors of AEC #4 to release mutating immediate permissions is relatively clear (failure to do so would introduce data races), the requirement that the predecessors also release scheduling permissions is a bit more subtle. While an AEC with None immediate permissions, such as AEC #2 in the above example, can never introduce data races, it retains the ability to create AECs that must be sequenced before any subsequent AECs from outer scopes. Specifically, the full set of predecessors of AEC #4 is not known until AEC #2 releases the permission to schedule Modify AECs that capture ah, since it could, hypothetically, create another AEC after AEC #3 that would also have to finish before the execution of AEC #4. (The same is true of AEC #1 and AEC #3). Maintaining this sequential consistency is important for determinism of DARMA programs.

7.1.1.3 Tracking available access permissions of an AAO

Keeping track of the permissions available to a given AAO at any particular point in the program can be tricky. To simplify the process, DARMA provides several helpful invariants and guiding principles by which the user should be able to reason about the permissions available in a given context. To begin with, DARMA always honors the requested permissions for a given AEC.

Invariant

At the beginning of an AEC, the permissions available to an AAO will always be greater than or equal to the permissions captured by that AEC.

(Note that the "greater than" part of the above invariant only comes up in rare cases, such as the if clause in the create_work_if(...).then(...) AEC. Available permissions at the beginning of an AEC are almost always equal to those requested.) In the DARMA C++ frontend, it can be a bit tricky to determine what permissions an AEC is requesting, especially when using lambdas. In general, a create_work (and its analogs) in the DARMA C++ frontend requests the greatest possible immediate and scheduling permissions from the context in which the AEC is created. This is best illustrated via a few examples:

```
1  auto ah1 = initial_access<   int>(); // initial_access   creates  AAO  with  { Modify , None  } access
2  auto ah2 = initial_access<   int>();
3  create_work([=]{
4    // by default , this  AEC  captures   { Modify , Modify  } on ah1  and  ah2  since  no  explicit
5    // downgrades   are  given  and  since  Modify  scheduling   permissions   are  available   in  the
6    // enclosing   context
7    ah1.set_value(10);    // The  C++ interface   can't tell  that  set_value   is called  on  ah1
8                        // in this  AEC, so its  presence  has no effect  on the  capture
9                        // permissions   (only  the  presence   of ah1  itself )
10   *ah1 += ah2.get_value();     // Likewise , the  C++ interface   cannot  detect  that  ah2  is
11                            // only  read  and  not  modified , so it follows  the  same
12                            // rules  as ah1  for  determining   captured   permissions
13 });
14 create_work(reads(ah1), [=]{
```

```
  // An explicit permissions downgrade is given here, so \darma captures { Read, Read }
  // on ah1. ah2 isn't used anywhere in this AEC's scope, so it is not captured.
  printf("%d", ah1.get_value());
  create_work([=]{
    // Since no explicit permissions downgrades are given here, this nested AEC
    // captures the maximum permissions possible from the enclosing context. Since
    // the enclosing context has { Read, Read } access, the permissions captured
    // on ah1 here are { Read, Read }
    printf("%d", ah1.get_value());
  });
});

struct Func1 {
  void operator()(AccessHandle< int> my_h) const { }
};
// Since Func1 doesn't specify anything permissions downgrades or requirements via
// its parameters, ah1 is captured with the maximum available permissions in from
// the enclosing context : { Modify, Modify } (since the enclosing context has
// { Modify, None } permissions )
create_work<Func1>(ah1);

struct Func2 {
  void operator()(ReadAccessHandle< int> my_h) const { }
};
// Func2 specifically downgrades its permissions requirements by specifying the
// parameter as a ReadAccessHandle <int>, so ah2 will be captured with
// { Read, Read } permissions .
create_work<Func2>(ah2);

struct Func3 {
  void operator()(int& my_h) const { }
};
// Func3 gives a parameter that requests a dereference of the AAO passed to it,
// thus indicating it doesn't need scheduling permissions. It requests a
// non-const reference, so the immediate permissions must be Modify. Thus,
// this statement creates an AEC captures { None, Modify } on ah2.
create_work<Func3>(ah2);
```

Furthermore, in order to maintain the *concurrent continuation* guarantee (see Section 7.1.2), the creation of an AEC that captures certain permissions can result in the reduction of available immediate permissions in continuing context. For instance, in a context where `{ Modify, Modify }` permissions are available for a given AAO, the creation of an AEC that captures `{ Modify, Modify }` permissions will result in that AAO having only `{ Modify, None }` permissions in the continuing context. When determining the available permissions in a given continuing context, the following invariant is helpful:

Invariant

In the continuing context following the creation of a given AEC, DARMA will always retain the maximum available permissions possible that will allow the created AEC to run concurrently with the continuation without introducing data races.

This principle is perhaps best illustrated by example:

```
auto ah = initial_access< int>(); // initially has { Modify, None } permissions
create_work([=]{
```

```
  // This AEC captures { Modify, Modify } permissions on ah
  ah.set_value(42);

  create_work([=]{
    // The nested AEC also captures { Modify, Modify } permissions on ah. By
    // sequence, the initial state of ah in this AEC must be 42.
    ah.set_value(ah.get_value() * 2);
  );

  // The continuing context must relinquish immediate permissions on ah, because even having
  // Read permissions would result in a data race with the nested AEC above. Permissions
  // available on ah are now { Modify, None }. Any further immediate modifications
  // must be made in another nested AEC:
  create_work([=]{ ah.set_value(ah.get_value() + 73); });
});
create_work([=]{
  // Another { Modify, Modify } capture.
  *ah = *ah - 5;

  create_work(reads(ah), [=]{
    // This AEC captures { Read, Read } permissions on ah
    printf("%d\n", ah.get_value());
  });

  // The continuing context has { Modify, Read } permissions on ah, since modifying
  // the data represented by ah would introduce a data race with the above nested AEC,
  // but reading introduces no such race.
  printf("Same as: %d\n", ah.get_value());

});
```

Beyond these basic rules, a couple more invariants may be helpful:

Invariant

The scheduling permissions available on a given AAO captured by a given AEC will never be greater (in terms of the partial ordering on permissions) at any point in that closure or any nested AECs than they are at the beginning of that closure.

Invariant

The immediate permissions available on a given AAO captured by a given AEC will never be greater (in terms of the partial ordering on permissions) at any point in that closure (but *not* necessarily in any nested AECs) than they are at the beginning of that closure.

Note that users can (and should) always explicitly release permissions they do not intend to use. The reasoning behind a lot of the above rules and invariants may become a bit clearer after we discuss the invariants on the execution of AECs in Section 7.1.2 below.

7.1.2 Execution of AECs

Asynchronously executable closures enable the backend to perform various approaches to task-based execution of DARMA programs. They provide a rough granularity specification at which the backend can reasonably perform dependency analysis and apply various task scheduling algorithms and heuristics. The rules for how AECs operate within the programming model are designed around these purposes, starting with the most basic design principle:

Fundamental Design Principle

Mandatory blocking is bad.

Backends will have much more flexibility to adapt to the challenges posed by modern high performance hardware — including heterogeneity, non-uniformity, fault-tolerance, and other concerns — if the points at which the program waits for data or other prerequisites are clearly specified. Structures that mandate blocking at the user level up resources that could be utilized to make progress on other work in the system.

Runtime systems could, of course, mitigate some of this with context switching, but most scientific applications have well-defined points at which context switching is significantly less expensive than others (for example, in between timesteps or other iterations of some sort). Furthermore, a backend scheduling model that incorporates both tasking and context switching within tasks in response to idleness needs to be significantly more sophisticated than one that schedules individual, non-interruptible units. Since AECs in the DARMA programming model specify individual units at which it is reasonable for the backend to perform scheduling work, the introduction of structures or abstractions that require the program to block make it difficult for the backend to apply scheduling heuristics based on workload introspection. From the user's perspective, this approach is a positive because it is much easier to avoid and detect deadlock if execution requirements are data-specific and correspond to well-defined, contiguous contexts in the source code. For all of these reasons and more, it is helpful for the DARMA programming model to provide the following invariant:

Invariant

The backend guarantees that any AEC will execute as if its execution were uninterrupted.

Note that AECs descended from a given AEC are *n*ot included in this uninterrupted execution guarantee. An immediate corollary to this invariant is that user code must not block indefinitely inside of any AEC.

Perhaps the most important invariant relating to AEC execution in DARMA is called the *concurrent continuation* guarantee:

Invariant

The programming model guarantees that it is safe to execute the continuation of a given AEC before, during, or after the execution of the AEC itself. This is called the *concurrent continuation* guarantee.

The need for this follows from the previous invariant, allowing the backend to work with non-blocking units of execution. It is useful for the user to keep this invariant in mind when reasoning about the permissions

available at any point in the source code. If maintaining permissions on some AAO in the continuing context following a given AEC could cause data races, those permissions are dropped in the continuation as described in Section 7.1.1.3

7.2 Backend Abstractions

7.2.1 The `Use` Life Cycle

In DARMA, `Use` objects are short-lived, frontend-defined, immutable objects that describe a particular set of permissions at a particular point in the program for a particular piece of data.

Invariant

Between the time `Runtime::register_use()` is called and the time `Runtime::release_use()` returns, the translation layer will not mutate the `Use` object with respect to the values returned by the methods of `Use` (or any of the methods accessible to a subclass of `Use` exposed in a particular part of the interface). This includes the address of the `Use`.

For a given `Use u`, the lifetime of `u` begins with a call to `Runtime::register_use()` with `&u` as an argument (exposed via a pointer to the subclass `UsePendingRegistration`). The subclass `UsePendingRegistration` exposes several methods to the backend that aren't available for the rest of the `Use`'s lifetime, including descriptions of the `Use`'s relationship to other `Use`s (via `Flow`s and `AntiFlow`s) and hooks that allow the backend to set the `Flow` and `AntiFlow` objects associated with the `Use` for the remainder of its lifetime. Thus, for all of the other places that `u` is accessible to the backend, `get_in_flow()` will return the `Flow` object set by the backend with `set_in_flow()` during registration, `get_anti_in_flow()` will return the `AntiFlow` object set by the backend with `set_anti_in_flow()` during registration, etc. The backend should use information from the `FlowRelationship` objects (returned by `get_in_flow_relationship()`, `get_anti_in_flow_relationship()`, etc.) associated with the `UsePendingRegistration` to determine what flow and anti-flow objects it should set for the `Use`. `FlowRelationship`s are frontend-provided, transient objects that describe how the backend should connect its implementation of `Flow` and `AntiFlow` object for a particular `Use` to other `Use`s (and their `Flow`s/`AntiFlow`s) in the system. The backend is free to assign any object it wants to the `Use` via these `UsePendingRegistration::set_*_flow()` methods; however there are some assumptions that we'll make in this document for linguistic brevity. When the result of `FlowRelationship::description()` is the value `Same` or `SameCollection`, we'll speak as if they are the same object (though they need not be, most implementations will probably do this). When the result of `FlowRelationship::description()` is `Insignificant` (or `InsignificantCollection`), we'll speak as if it's not there. (All of these values for `FlowRelationship::description()` are static data members of the `FlowRelationship` class).

The life cycle of a `Use` has up to three phases. First, *every* `Use` is registered using `Runtime::register_use()`. After that, the middle phase of a `Use`'s life cycle can vary significantly. Some `Use`s are part of the iterable object returned by `Task::get_dependencies()`. Other `Use` objects are simply for reference counting purposes and represent the retention of permissions immediately after construction or in a continuation. This second type of `Use` will not have a middle phase of its life cycle — only a registration phase and a release phase. Finally, there are some special `Use`s that are registered and then ownership of the `Use` is transferred to the backend via methods like `Runtime::publish_use` or `Runtime::allreduce_use`. These

third types of uses do not have a release phase, since the responsibility for release has been transferred to the backend. All other `Use`s, though, have a release phase, in which `Runtime::release_use()` is called and after which it is not valid to call any methods on the `Use` object.

7.2.2 `Flow`s and `AntiFlow`s

A `Flow` is intended to correspond (roughly) to a generation of a piece of data. All `Use`s that require (consume) a given generation of that data (or the permission to schedule to it) will have the same flow as an `in` flow. All tasks that produce a given generation of the data (or that could schedule tasks that do) will return the same `out` flow. (Usually, there is only one producer of a given generation of data, but for certain permissions — `Commutative`, `Relax`, and `Reduce` — there can be multiple producers). An `AntiFlow` expresses an anti-dependency relationship in an analogous manner — `Use` objects that access the same data with the understanding that it will not be modified will have the same `anti-out` flow, and the corresponding `Use` objects that modify that data and need to wait for these "anti-producers" to release permissions will have that same `AntiFlow` as their `anti-in` flow.

While all of this sounds extremely complicated, `Use`-`Flow` semantics actually provides a number of very useful invariants to help deal with these situations.

Invariant

As long as the in-flow (or any flow that aliases it) of a dependency `Use` of a closure that captures immediate permissions greater than `None` is the out-flow of any `Use` that has been registered but not yet released, the capturing closure of that dependency `Use` cannot be executed. *As soon as* this is no longer the case, the capturing closure will be no longer constrained by the in-flow of that `Use` for the remainder of its lifetime.

Definition

Dependency Use

A `Use` that establishes an dependency for its capturing closure via its in-flow. Whether or not a `Use` is a dependency `Use` can be queried with the instance method `Use::is_dependency()`.

(Note that the dependency established by an dependency `Use` may be trivially satisfied at the time that the capturing `Task` is registered, depending on DAG expansion and execution order).

A similar invariant exists for `AntiFlow`s:

Invariant

As long as the anti-in-flow (or any anti-flow that aliases it) of an anti-dependency `Use` of a closure is the anti-out-flow of any `Use` that has been registered but not yet released *or* is the anti-in flow of any non-anti-dependency `Use` that has been registered but not yet released, the capturing closure of that dependency `Use` cannot be executed. *As soon as* this is no longer the case, the capturing closure will be no longer constrained by the anti-in-flow of that `Use` for the remainder of its lifetime.

Definition

Anti-dependency Use

A `Use` that establishes an anti-dependency for its capturing closure via its anti-in-flow. Whether or not a `Use` is an anti-dependency `Use` can be queried with the instance method `Use::is_anti_dependency()` .

(Note that the anti-dependency established by an anti-dependency `Use` may be trivially satisfied at the time that the capturing `Task` is registered, depending on DAG expansion and execution order).

7.2.3 `Handle` s and `SerializationManager`

Modern programming languages have extremely complex and flexible approaches to abstracting and representing data. In general, the most natural representations of the application's data (for instance, in C++ these representations are usually objects) can have several properties that are difficult for task-based runtime systems to deal with:

1. The underlying data that an object represents can be *contiguous* or *disjoint* in memory.[3]
2. Objects can represent data of the same size over their entire lifetime or can represent variable amounts of data at different points in their lifetimes.
3. Two objects may represent some or all of the same underlying data. For two objects `a` and `b`, there are several ways they can be related in this respect (we will use the general term *aliasing* for this general category of concerns):
   (a) `a` and `b` can represent the exact same data in memory.
   (b) `a` can represent a strict subset of the memory `b` represents.
   (c) A subset of the data represented by `a` can be the same as a subset of the data represented by `b`.

   Additionally, the nature of or constraints on the relationship between `a` and `b` can be static or can change over the lifetime of the two objects, and the ownership of the underlying data (or subsets thereof) can be associated with `a`, with `b`, or shared between the two.
4. Objects can reference each other in cycles (without cycles, this concern is identical to issue 3b).

The implications of these properties of data for runtime systems can vary significantly, and thus the mechanisms for and degrees to which DARMA addresses these concerns also vary. DARMA separates these concerns into abstractions (mostly) accessible through `Handle` objects, which are frontend-defined objects that abstract information about a piece of application data that is common to all parts of a given chain of `Flow` s and `Use` s.

[3] This, of course, assumes the relatively ubiquitous representation of memory as byte-addressable with a uniform, monotonically increasing (almost always integral) representation of an address. Under this assumption, the data for an object is contiguous if its size is exactly equal to the distance from its smallest addressable byte to its largest, inclusive. Much more could be said, formally speaking, about this and other concepts in this section, but we feel most of our readers have a fairly reasonable understanding of what these terms mean. Clarifications of these or related terms can often be found in the specification of the host language (e.g., C++ for the DARMA C++ frontend, etc.)

Issue 1 mostly affects the requirements for communication between disjoint memory spaces,[4] since, for instance, most networking protocols operate on contiguous chunks of memory. In the DARMA backend API, the transformation between (potentially) disjoint and contiguous representations of the data is handled by `SerializationManager` objects, which are frontend-defined objects that express attributes of the data in memory for a given `Handle`. `SerializationManager` objects in DARMA always assume that the user-level abstraction for some object's data has at least one statically sized, contiguous piece of data through which the rest of the object's data may be accessed, which we call the object's *metadata*. (In C++, this is a safe assumption since all objects have a well-defined *object representation* that is contiguous in memory and statically sized.) The size of this metadata block for a particular object can be obtained using the instance method `SerializationManager::get_metadata_size()` (which, in C++ for an object of type `T`, returns exactly `sizeof(T)`). To perform the transformation of a `Handle`'s data into a contiguous representation, the backend first queries the size of the contiguous representation using the instance method `SerializationManager::get_packed_data_size()` with a pointer to the object's metadata block. Through this mechanism, the size of the total data represented by an object in DARMA is allowed to change over its lifetime, addressing issue 2. If an object's data or portions of its data are already contiguous, the translation layer will express this to the backend via callbacks to a `SerialiationPolicy` object (backend-provided) passed to the various methods of `SerializationManager`, thus enabling optimizations such as zero-copy transfer of data. To continue the transformation to a contiguous representation, the backend allocates a buffer of the appropriate size (returned by `SerializationManager::get_packed_data_size()`) and calls the instance method `SerializationManager::pack_data()`. Upon return, the buffer passed as an argument contains a contiguous representation (also called a "packed" representation) of the data. To reverse the process and transform the contiguous representation back to a disjoint representation usable by the application, the backend calls the instance method `SerializationManager::unpack_data()` with a buffer containing the packed data and a metadata block allocated to the appropriate size.

As an aside, some portion of the information presented by `SerializationManager` can be determined statically (for instance, from the static type of an object). To simplify the backend development experience, the DARMA API presents all of this information to the backend via dynamic hooks, but future iterations may express more information statically to help improve performance.

Currently, DARMA simplifies the concerns associated with issues 3 and 4 by disallowing their expression in application code. For instance, backends do not have to deal with issue 3 because DARMA requires that the data underlying any two `Handle` instances cannot overlap. We recognize that this is unlikely to be a viable long-term strategy, and we are investigating means of expressing the relaxation of this requirement. It is likely that the first extensions will address issues 3a and 3b, with extensions to address issue 3c in the slightly more distant future. We currently have no plans to address issue 4 (most distributed programming models do not address this concern).

7.2.4 `Task`s

With the heavy emphasis on a data-centric concurrency model in DARMA, the `Task` abstraction, represented by a frontend-provided object, becomes a relatively light-weight concept. A `Task` in DARMA simply groups a set of `Use`s, expressing the `Task`'s dependencies and anti-dependencies, with a function to be run (using the instance method `Task::run()`) at some finite time after those dependencies and anti-dependencies are satisfied.

[4] That is, roughly speaking, disjoint domains of addressability.

In general, Task s are meant to be well-isolated contexts within which the user can rely on relatively normal behavior from the host language. In C++ (and likely for other host languages we plan to explore in the future), this places some fairly significant restrictions on what the runtime is allowed to do with a task once its run() method has been invoked (and before it returns; also referred to as "while the Task is running"). Most importantly, the Task basically cannot be moved to a different memory space once it has started running.[5]

7.2.5 UseCollection s and TaskCollection s

To allow for scalable implementation of distributed dependency analysis, DARMA provides a mechanism for coarsening of data effect expression. Closures in DARMA with some or all of their data effect annotations coarsened (as well as the expression of their execution requirements) are expressed with TaskCollection s objects passed to Runtime::register_task_collection() . Each TaskCollection has a size() method that indicates the number of Task s that it is a coarsened representation of. The runtime is required to create and execute size() of these Task s (in the memory space where they are to be executed) by calling TaskCollection::create_task_for_index() with each integer in the range [0, size()) as an argument. Some of the dependencies to a TaskCollection may be Use objects that manage a UseCollection (indicated by a return value of true from Use::manages_collection()), which is a coarsened representation of the dependency structure of the user's data. Like TaskCollection s, UseCollection s have a size() , and frontend may request indexed refinements of the coarse-grained Use 's Flow s and AntiFlow s with indices in the range [0, UseCollection::size()) . A UseCollection object may be *mapped* (indicated by the return value of UseCollection::is_mapped()) over the TaskCollection that depends on it. A mapping of a UseCollection is a compact representation of the indices for the Task s created for the TaskCollection that the Use is a dependency of that will require the permissions specified in the Use on a given index of the UseCollection . Because of scalability concerns, a UseCollection that is not mapped to a TaskCollection will never request immediate permissions other than None on the data.[6] There are further restrictions to prevent other capture modes that don't make sense with the sequential semantic requirements of the capture permissions. For instance, since Modify permissions imply the maintainance of program sequence (a requirement enforced by the translation layer via Flow s and AntiFlow s), and since the Task s in the refinement of a TaskCollection are unsequenced with respect to each other, id does not make sense for an unmapped UseCollection to request Modify permissions, and any mapping of a UseCollection that requests Modify permissions (either scheduling or immediate) must establish a disjoint partition (no index of a UseCollection may map to more than one index of the corresponding TaskCollection).

These interactions are perhaps best illustrated with an example. Consider a TaskCollection (call it t) registered via Runtime::register_task_collection() . t->size() returns 5, so the backend is required to run Task s for all indices in the range [0,5) . The iterable returned by t->get_dependencies() contains two Use objects, u1 and u2. Both u1->manages_collection() and u2->manages_collection() return true . u1 expresses the capture of {Modify, Modify} permissions on some Handle , and u2 expresses teh capture of {Read, None} permissions on some other handle. u1->get_managed_collection() and u2->get_managed_collection() return pointers to the UseCollection objects uc1 and uc2 , respectively. uc1->is_mapped() returns true , and uc1->size() returns 10. The runtime may then call uc1

[5] We say "basically" here because if the backend can ensure that the effects of mid-Task migration are undetectable by the application, the migration is allowed. However, this is very difficult even just within the C++ frontend, and was we generalize further to other host languages, it will become even more difficult for runtimes to provide this sort of guarantee.

[6] This may change to include Combine permissions once the details of that mode are more hardened.

->task_index_for(i) , where i is any integer in the range [0,10) . If uc1->task_index_for(0) and uc1->task_for_index(1) both return 0, and uc1->local_indices_for(0) returns an iterable containing 0 and 1. The runtime can then expect that t->create_task_for_index(0) (call the return value t_0) will result in registration of two Use s, uc1_0 and uc1_1 , that will be dependencies of the resulting Task . The registration of uc1_1 , for instance, will have an in FlowRelationship that indicates an IndexedLocal relationship to the in Flow of uc1 (with uc1_1->get_in_flow_relationship()->index() returning 1), and analogously for the other FlowRelationship s that aren't Insignificant . Since uc2 is unmapped, there may be any number of dependencies for t_0 that have flows with Indexed relationships to uc2 , but all of them must request {Read, None} permissions. The execution of t_0 may create tasks that request, for instance, {Read, Read} permissions on any of these, and for it may be important for the backend to migrate these tasks (as discussed below) to the physical location of the data abstracted by the given index of uc2 .

7.2.6 Task Migration

Most Task s and all TaskCollection s in DARMA can be migrated by the backend from the memory space on which they were created. (Task s that cannot be migrated will return false for Task::is_migratable () .) The basic migration procedure is similar for both Task s and TaskCollection s. The runtime initiates the process by inquiring about the size of the contiguous buffer needed to pack up the Task or TaskCollection (using the get_packed_size() method of either object) and then calls the pack() method with a buffer of the required size. During the frontend's execution of these methods, it will call Runtime::get_packed_flow_size() and Runtime::pack_flow() (and their anti_flow analogs) for each Flow and AntiFlow in the Task 's dependencies, allowing the backend to pack up its data for that Task . The unpack procedure proceeds similarly on the remote side, except that the unpack process will call Runtime::reregister_migrated_use() to re-initiate the Use life-cycle on the remote end.

7.2.7 Publication and Collectives

When a user wants to interact with data not mapped to a given index of a TaskCollection , they must do so explicitly via publication or collectives. Collectives can also be performed over UseCollection s that are not captured by a TaskCollection , as long as the call to the collective operation happens outside the context of a TaskCollection closure.

7.3 Future Work

7.3.1 Improvements for Interfacing with Mature Applications

As of the writing of this report, the DARMA framework is principally useful for rapid prototyping of applications built from scratch. It is critical for the continued viability of DARMA that we (1) provide better abstractions for customization and modification of both translation layer and backend functionality for use as applications written in DARMA mature and (2) provide better abstractions for interfacing with existing code in mature applications. The following are the details of some of the plans for several important

improvements and extensions to the current frontend API that will help to address these concerns moving forward.

7.3.1.1 Generalization of and Specialization Hooks for AccessHandle

As alluded to in Section 7.1, we view `AccessHandle` and `AccessHandleCollection` as specific instances of a more general entity called an *asynchronously accessible object* (AAO). The aspects of this more general entity that the current implementations of `AccessHandle` and `AccessHandleCollection` specialize could, ideally, be customized by advanced application developers to create an AAO abstraction that better fits their application needs. The specific mechanism for providing such hooks would utilize C++ template specialization to allow developers to describe the semantics of capturing their custom AAO. For instance, advanced app developers may wish to customize the type of the dereferenced AAO or specify which types (when given as parameters to functors) indicate which captures (for example, for `AccessHandle<T>`, a parameter of type `T&` indicates a capture with scheduling permissions of `None` and immediate permissions of `Modify` [7]).

7.3.1.2 Interfacing with Unmanaged Data

At their root, AAOs like `AccessHandle` can be thought of as concurrent control flow abstractions that happen to be tied to a particular piece of data. By default, the approach in the DARMA programming model has been to manage that data in the backend so that it can be allocated, copied, and moved in accordance with the needs of the scheduler. But under some restrictions to the scheduler, these control flow abstractions can be similarly used to restrict concurrent access to any arbitrary piece of data, even ones created by non-DARMA parts of an application. In the near future, we plan to provide an interface that allows users to do exactly this, which should allow for a significant increase in interoperability and should provide a reasonable stepwise path towards integration of DARMA into existing applications.

7.3.1.3 DARMA Regions

Another promising path forward for progressive integration of DARMA into existing applications involves an abstraction we call DARMA regions. A DARMA region would be a fully strict[8] containment scope before which DARMA operations are not allowed and after which all work created in the scope is guaranteed to have completed. Initially, these regions would likely be only for on-node concurrency management, since efficient, lightweight startup and shutdown of the distributed DARMA infrastructure is not reasonable at this time. However, we have several ideas we would like to pursue with respect to stepwise integration of DARMA regions into the distributed logic of an application. One possibility is that DARMA regions could return a waitable entity (for instance, a future) on which users could predicate other parts of the application (including, potentially, other DARMA regions). For interaction with communicating sequential processes (CSP) programming models, this waitable entity could provide a mechanism to send itself outside the confines of DARMA, given an callable that tells it how to send and receive an arbitrary collection of bits under the runtime system used by the application outside of the DARMA region. Once communicated, the user

[7] Unless the argument captured has scheduling permissions of `Commutative`, `Relaxed`, or `Combine`, in which case those permissions are captured instead.

[8] at least at first; there are plans to relax the strictness of DARMA regions

could predicate other DARMA regions on remote nodes on these objects, thus progressively building up a task-based description of their program that could eventually be transitioned into full adoption of DARMA.

This page intentionally left blank.

Glossary

actor model An actor model covers both aspects of programming and execution models. In the actor model, applications are decomposed across objects called actors rather than processes or threads (MPI ranks). The actor model shares similarities with active messages. Actors send messages to other actors, but beyond simply exchanging data they can invoke remote procedure calls to create remote work or even spawn new actors. The actor model mixes aspects of single-program multiple-data (SPMD) in that many actors are usually created for a data-parallel decomposition. It also mixes aspects of fork-join in that actor messages can "fork" new parallel work; the forks and joins, however, do not conform to any strict parent-child structure since usually any actor can send messages to any other actor. In the Charm++ implementation of the actor model, the actors are chares and are migratable between processes. 28

AMT See AMT model. 23, 27, 28, 34, 155

AMT model Asynchronous many-task (AMT) is a categorization of programming and execution models that break from the dominant CSP or SPMD models. Different asynchronous many-task runtime system (AMT RTS) implementations can share a common AMT model. An AMT programming model decomposes applications into small, transferable units of work (many tasks) with associated inputs (dependencies or data blocks) rather than simply decomposing at the process level (MPI ranks). An AMT execution model can be viewed as the coarse-grained, distributed memory analog of instruction-level parallelism, extending the concepts of data prefetching, out-of-order task execution based on dependency analysis, and even branch prediction (speculative execution). Rather than executing in a well-defined order, tasks execute when inputs become available. An AMT model aims to leverage all available task and pipeline parallelism, rather just relying on basic data parallelism for concurrency. The term asynchronous encompasses the idea that 1) processes (threads) can diverge to different tasks, rather than executing in the same order; and 2) concurrency is maximized (minimum synchronization) by leveraging multiple forms of parallelism. The term many-task encompasses the idea that the application is decomposed into many *transferable* or *migratable* units of work, to enable the overlap of communication and computation as well as asynchronous load balancing strategies. 27, 155

AMT RTS A runtime system based on AMT concepts. An AMT RTS provides a specific implementation of an AMT model. 155

anti-dependency See Write-After-Read. 38, *Glossary:* Write-After-Read

API An application programmer interface (API) is set of functions and tools provided by a library developer to allow an application programmer to interact with a specific piece of software or allow a developer to utilize prebuilt functionality. 28, 43

ASC The Advanced Simulation and Computing (ASC) Program supports the Department of Energy's National Nuclear Security Administration (NNSA) Defense Programs' shift in emphasis from test-based confidence to simulation-based confidence. Under ASC, computer simulation capabilities are developed to analyze and predict the performance, safety, and reliability of nuclear weapons and to certify

their functionality. ASC integrates the work of three Defense programs laboratories (Los Alamos National Laboratory, Lawrence Livermore National Laboratory, and Sandia National Laboratories) and university researchers nationally into a coordinated program administered by NNSA. 23, 25, 28, 156

ASD Architecture and Software Development. 25

ATDM This ASC program includes laboratory code and computer engineering and science projects that pursue long-term simulation and computing goals relevant to the broad national security missions of the National Nuclear Security Administration. 25, 26, 28

CDA Code Development and Applications. 25

CFD computational fluid dynamics. 28

chare The basic unit of computational work within the Charm++ framework. Chares are essentially C++ objects that contain methods that carry out computations on an objects data asynchronously from the method's invocation. 158

CSP CSP (communicating sequential processes) is the most popular concurrency model for science and engineering applications, often being synonymous with SPMD. CSP covers execution models where a usually fixed number of independent workers operate in parallel, occasionally synchronizing and exchanging data through inter-process communication. Workers are *disjoint processes*, operating in separate address spaces. This also makes it generally synonymous with message-passing in which data exchanges between parallel workers are copy-on-read, creating disjoint data parallelism. The term sequential is historical and CSP is generally applied even to cases in which each "sequential process" is composed of multiple parallel workers (usually threads).. 26, 27, 155–157

CSSE Computational Systems and Software Engineering. 25

DAG A directed acyclic graph (DAG) is a directed graph with no cycles. This type of data representation is common form for representing dependencies. 34

data flow dependency A data dependency where a set of tasks or instructions require a certain sequence to complete without causing race conditions. Data-flow dependency types include Write-After-Read, Read-After-Write and Write-After-Write. 38

declarative A style of programming that focuses on using statements to define what a program should accomplish rather than how it should accomplish the desired result. 26–28, 43, 157

DSL Domain specific Languages (DSL) are a subset of programming languages that have been specialized to a particular application domain. Typically, DSL code focuses on what a programmer wants to happen with respect to their application and leaves the runtime system to determine how the application is executed. 28

EMPIRE General purpose open source code (ElectroMagnetic Plasma In Radiation Environments). 101, 102

execution model A parallel execution model specifies how an application creates and manages concurrency. This covers, e.g., CSP (communicating sequential processes), strict fork-join, or event-based execution. These classifications distinguish whether many parallel workers begin simultaneously

(CSP) and synchronize to reduce concurrency or if a single top-level worker forks new tasks to increase concurrency. These classifications also distinguish how parallel hazards (WAR, Read-After-Write (RAW), Write-After-Write (WAW)) are managed either through synchronization, atomics, conservative execution, or idempotent execution. In many cases, the programming model and execution model are closely tied and therefore not distinguished. The non-specific term parallel model can be applied. In other cases, the way execution is managed is decoupled from the programming model in runtime systems with declarative programming models like Legion or Uintah. The execution model is implemented in the runtime system. 26, 27, 155, 158, 159

fork-join A model of concurrent execution in which child tasks are forked off a parent task. When child tasks complete, they synchronize with join partners to signal execution is complete. Fully strict execution requires join edges be from parent to child while terminally strict requires child tasks to join with grandparent or other ancestor tasks. This style of execution contrasts with SPMD in which there are many parallel sibling tasks running, but they did not fork from a common parent and do not join with ancestor tasks. 58, 155, 156

FOUS Facility Operations and User Support. 25

GNI General Network Interface. On Cray systems (Gemini and Aries), provides the lower-level network API with datagram and RDMA operations.. 27

IC Integrated codes. 25

imperative A style of programming where statements change the state of a program to produce a specific result. This contrasts to declarative programming that focuses on defining the desired result without specifying how the result is to be accomplished. 26–28

interoperability The ability of different software components to effectively and efficiently exchange information and share resources.. 57

KNL Intel's 2nd-generation many-integrated core (MIC) architecture with high multi-threaded performance. 157

libfabrics Emerging standard for diverse network interfaces to provide the lower-level network API with datagram and RDMA operations. Supported by Cray, Intel, Mellanox and others.. 27

MCDRAM High-bandwidth memory used on Knight's Landing systems to improve performance relative to regular DRAM. 66, 69

Miniapp Small, self-contained programs that embody essential performance characteristics of key applications. 101

MPI Message Passing Interface. 26–28, 155, 158, 159

NESAP NERSC Exascale Science Applications Program. 101

NGP next generation platform. 25

NGS next generation simulation. 25

NNSA National Nuclear Security Administration. 158

PAMI Parallel Active Message Interface. On IBM systems (Blue Gene) provides the lower-level network API with datagram and RDMA operations.. 27

PEM Physics and Engineering Models. 25

PIC A technique used to solve wide range of partial deferential equations and very popular method for computer simulations of plasmas. 158, 159

PICSAR A high performance Fortran/Python particle-in-cell (PIC) library, based on WARP targeting MIC architectures (Particle-In-Cell Scalable Application Resource). 101, 104

procedural A style of programming where developers define step by step instructions to complete a given function/task. A procedural program has a clearly defined structure with statements ordered specifically to define program behavior. 28

programming model A parallel programming model is an abstract view of a machine and set of first-class constructs for expressing algorithms. The programming model focuses on how problems are decomposed and expressed. In MPI, programs are decomposed based on MPI ranks that coordinate via messages. This programming model can be termed SPMD, decomposing the problem into disjoint (non-conflicting) data regions. Charm++ decomposes problems via migratable objects called chares that coordinate via remote procedure calls (entry methods). Legion decomposes problems in a data-centric way with logical regions. All parallel coordination is implicitly expressed via data dependencies. The parallel programming model covers how an application *expresses* concurrency. In many cases, the execution model and programming model are closely tied and therefore not distinguished. In these cases the non-specific term parallel model can be applied. 26, 155, 157, 159

Proxy application Parametrizable applications that can be calibrated to mimic the performance of a large scale application, then used as a proxy for the large scale application. 63

PSAAP-II The primary goal of the National Nuclear Security Administration (NNSA)'s Predictive Science Academic Alliance Program (PSAAP) is to establish validated, large-scale, multidisciplinary, simulation-based "Predictive Science" as a major academic and applied research program. The Program Statement lays out the goals for a multiyear program as follow-on to the present ASC Alliance program. This "Predictive Science" is the application of verified and validated computational simulations to predict properties and dynamics of complex systems. This process is potentially applicable to a variety of applications, from nuclear weapons effects to efficient manufacturing, global economics, to a basic understanding of the universe. Each of these simulations requires the integration of a diverse set of disciplines; each discipline in its own right is an important component of many applications. Success requires both software and algorithmic frameworks for integrating models and code from multiple disciplines into a single application and significant disciplinary strength and depth to make that integration effective. 27

RAW Read-After-Write. 157, 158

Read-After-Write Read after write (RAW) is a standard data dependency (or potential hazard) where one instruction or task requires, as an input, a data value that is computed by some other instruction or task. 156–158

remote procedure invocation See RPC. 28, *Glossary:* RPC

runtime system A parallel runtime system primarily implements portions of an execution model, managing how and where concurrency is managed and created. Runtime systems therefore control the order in which parallel work (decomposed and expressed via the programming model) is actually performed and executed. Runtime systems can range greatly in complexity. A runtime could only provide point-to-point message-passing, for which the runtime only manages message order and tag matching. A full MPI implementation automatically manages collectives and global synchronization mechanisms. Legion handles not only data movement but task placement and out-of-order task execution, handling almost all aspects of execution in the runtime. Generally, parallel execution requires managing task placement, data placement, concurrency creation, concurrency managed, task ordering, and data movement. A runtime comprises all aspects of parallel execution that are not explicitly managed by the application. 26, 27, 157

SPMD The term single-program multiple-data (SPMD) refers to a parallel programming model where the same tasks are carried out by multiple processing units but operate on different sets of input data. This is the most common form of parallelization and often involves multithreading on a single compute node and/or distributed computing using MPI communication. 155–158

SSP Stockpile Stewardship Program. 25

task stealing See work stealing. *Glossary:* work stealing

transport layer The system layer responsible for delivering data to distributed application processes.. 27

UQ uncertainty quantification. 25, 115

V&V Verification and Validation. 25

WAR Write-After-Read. 38, 157, 159

WARP Open source PIC code designed to simulate charged particle beams and laser-matter interactions. 101, 158

WAW Write-After-Write. 157, 159

Write-After-Read Write after read (WAR), also known as an anti-dependency, is a potential data hazard where a task or instruction has required input(s) that are later changed. An anti-dependency can be removed at instruction-level through register renaming or a task-level through copy-on-read or copy-on-write. 38, 155, 156, 159

Write-After-Write Write after write (WAW), also known as an output dependency, is a potential data hazard where data dependence is only written (not read) by two or more tasks. In a sequential execution, the value of the data will be well defined, but in a parallel execution, the value is determined by the execution order of the tasks writing the value. 156, 157, 159

This page intentionally left blank.

References

[1] Joeseph E. Bishop, John M. Emery, Richard V. Field, Christopher R. Weinberger, and David J. Littlewood. Direct numerical simulations in solid mechanics for understanding the macroscale effects of microscale material variability. *Computer Methods in Applied Mechanics and Engineering*, 287:262–289, 2015.

[2] Joseph E. Bishop, John M. Emery, Corbett C. Battaile, David J. Littlewood, and Andrew J. Baines. Direct numerical simulations in solid mechanics for quantifying the macroscale effects of microstructure and material model-form error. *JOM*, 68(5):1427–1445, 2016.

[3] Rick Stevens, Andrew White, Sudip Dosanjh, Al Geist, Brent Gorda, Kathy Yelick, John Morrison, Horst Simon, John Shalf, Jeff Nichols, and Mark Seager. Architectures and technology for extreme scale computing. Technical report, U. S. Department of Energy, 2009.

[4] Sean Ahern, Arie Shoshani, Kwan-Liu Ma, Alok Choudhary, Terence Critchlow, Scott Klasky, Valerio Pascucci, Jim Ahrens, E. Wes Bethel, Hank Childs, Jian Huang, Ken Joy, Quincey Koziol, Gerald Lofstead, Jeremy S Meredith, Kenneth Moreland, George Ostrouchov, Michael Papka, Venkatram Vishwanath, Matthew Wolf, Nicholas Wright, and Kensheng Wu. *Scientific Discovery at the Exascale, a Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization*. 2011.

[5] Predictive Science Academic Alliance Program (PSAAP II). `http://www.sandia.gov/psaap` .

[6] P An, A Jula, S Rus, S Saunders, T Smith, G Tanase, N Thomas, N Amato, and L Rauchwerger. Stapl: an adaptive, generic parallel c++ library. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, pages 193–208, 2003.

[7] M Bauer, S Treichler, E Slaughter, and A Aiken. Legion: expressing locality and independence with logical regions. In *SC '12: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2012.

[8] L V Kale and S Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *OOPSLA 1993: 8th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–108, 1993.

[9] John Davison De St Germain, Steven G. Parker, Christopher R. Johnson, and John McCorquodale. Uintah: a massively parallel problem solving environment. 2000.

[10] Edward A. Luke. Loci: A deductive framework for graph-based algorithms. In Satoshi Matsuoka, R. R. Oldehoeft, and Marydell Tholburn, editors, *Computing in Object-Oriented Parallel Environments (3rd ISCOPE'99)*, volume 1732 of *Lecture Notes in Computer Science (LNCS)*, pages 142–153. Springer-Verlag (New York), San Francisco, California, USA, December 1999.

[11] HPX3. `http://stellar.cct.lsu.edu/projects/hpx/` .

[12] HPX5. `https://hpx.crest.iu.edu` .

[13] Tim Mattson, Romain Cledat, Zoran Budimlic, Vincent Cave, Sanjay Chatterjee, Bala Seshasayee, Rob van der Wijngaart, and Vivek Sarkar. OCR: The Open Community Runtime Interface. Technical report, June 2015.

[14] G Bosilca, A Bouteiller, A Danalis, M Faverge, T Herault, and J J Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computer Science and Engineering*, 15:36–45, 2013.

[15] Howard Pritchard, Igor Gorodetsky, and Darius Buntinas. A ugni-based mpich2 nemesis network module for the cray xe. In *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, EuroMPI'11, pages 110–119, Berlin, Heidelberg, 2011. Springer-Verlag.

[16] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. Pami: A parallel active message interface for the blue gene/q supercomputer. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 763–773, May 2012.

[17] S. E. Choi, H. Pritchard, J. Shimek, J. Swaro, Z. Tiffany, and B. Turrubiates. An implementation of ofi libfabric in support of multithreaded pgas solutions. In *2015 9th International Conference on Partitioned Global Address Space Programming Models*, pages 59–69, Sept 2015.

[18] Janine Bennett, Robert Clay, Gavin Baker, Marc Gamell, David Hollman, Samuel Knight, Hemanth Kolla, Gregory Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke, Matt Bettencourt, Steve Bova, Ken Franko, Paul Lin, Ryan Grant, Si Hammond, Stephen Olivier, Laxmikant Kale, Nikhil Jain, Eric Mikida, Alex Aiken, Mike Bauer, Wonchan Lee, Elliott Slaughter, Sean Treichler, Martin Berzins, Todd Harman, Alan Humphrey, John Schmidt, Dan Sunderland, Pat McCormick, Samuel Gutierrez, Martin Schulz, Abhinav Bhatele, David Boehme, Peer-Timo Bremer, and Todd Gamblin. ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms. Technical Report SAND2015-8312, Sandia National Laboratories, Albuquerque, NM, August 2015.

[19] Kenneth Franko. MiniAero: A performance portable mini-application for compressible fluid dynamics. In preparation.

[20] Kenneth J. Franko, Travis C. Fisher, Paul Lin, and Steven W. Bova. CFD for next generation hardware: Experiences with proxy applications. In *Proceedings of the 22nd AIAA Computational Fluid Dynamics Conference, June 22–26, 2015, Dallas, TX*. AIAA 2015–3053.

[21] Mantevo. `http://mantevo.org/` .

[22] Justin Luitjens and Martin Berzins. Improving the performance of Uintah: A large-scale adaptive meshing computational framework. In *Proc. 24th IEEE International Symposium on Parallel and Distributed Processing (24th IPDPS'10)*, pages 1–10, Atlanta, Georgia, USA, April 2010.

[23] James C. Phillips, Yanhua Sun, Nikhil Jain, Eric J. Bohm, and Laxmikant V. Kale. Mapping to Irregular Torus Topologies and Other Techniques for Petascale Biomolecular Simulation. In *Proceedings of ACM/IEEE SC 2014*, New Orleans, Louisiana, November 2014.

[24] Harshitha Menon, Lukasz Wesolowski, Gengbin Zheng, Pritish Jetley, Laxmikant Kale, Thomas Quinn, and Fabio Governato. Adaptive techniques for clustered n-body cosmological simulations. *Computational Astrophysics and Cosmology*, 2(1):1–16, 2015.

[25] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[26] Wikipedia: Producer-Consumer problem. https://en.wikipedia.org/wiki/Producerconsumer_problem .

[27] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: an event-based low-level runtime for distributed memory architectures. In José Nelson Amaral and Josep Torrellas, editors, *PACT*, pages 263–276. ACM, 2014.

[28] The Common Component Architecture Forum. http://www.cca-forum.org/groups/index.html .

[29] Stephen L. Olivier, Kevin T. Pedretti, and Ronald B. Brightwell. Software Requirements for ATDM On-Node Resource Management. Technical Report SAND2016-6357, Sandia National Laboratories, Albuquerque, NM, June 2016.

[30] Charmworks. http://charmplusplus.com/about.html .

[31] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furment o, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. IEEE, 2010.

[32] Trilinos. https://trilinos.org .

[33] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation*. 1991.

[34] R. W. Hockney and J. W. Eastwood. *Computer simulation using particles*. 1988.

[35] O. Buneman. Dissipation of currents in ionized media. *Phys. Rev.*, 115:503–517, 1959.

[36] J. Dawson. One-Dimensional Plasma Model. *Physics of Fluids*, 5:445–459, 1962.

[37] NESAP Web page. http://nersc.gov/users/computational-systems/cori/nesap, 2017.

[38] Douglas Doerfler, Jack Deslippe, Samuel Williams, Leonid Oliker, Brandon Cook, Thorsten Kurth, Mathieu Lobet, Tareq Malas, Jean-Luc Vay, and Henri Vincenti. *Applying the Roofline Performance Model to the Intel Xeon Phi Knights Landing Processor*, pages 339–353. Springer International Publishing, Cham, 2016.

[39] APEX Procurement Benchmark Distribution. http://www.nersc.gov/research-and-development/apex/apex-benchmarks/ .

[40] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[41] SIERRA Solid Mechanics Team. Sierra/SolidMechanics 4.44 user's guide. SAND Report 2017-4016, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2017.

[42] T. J. R. Hughes. *The finite element method: linear static and dynamic finite element analysis*. Englewood Cliffs, N.J. : Prentice-Hall, 1987.

[43] T. Belytschko, W. K. Liu, and B. Moran. *Nonlinear finite elements for continua and structures*. John Wiley & Sons, Ltd., Chichester, England, 2000.

[44] M.G.D. Geers, V.G. Kouznetsova, and W.A.M. Brekelmans. Multi-scale computational homogenization: Trends and challenges. *Journal of Computational and Applied Mathematics*, 234:2175–2182, 2010.

[45] K. Pham, V.G. Kouznetsova, and Geers M.G.D. Transient computational homogenization for heterogeneous materials under dynamic excitation. *Journal of the Mechanics and Physicas of Solids*, 61:2125–2146, 2013.

[46] C. Miehe and C.G. Bayreuther. On multiscale FE analyses of heterogeneous structures: From homogenization to multigrid solvers. *International Journal for Numerical Methods in Engineering*, 71:1135–1180, 2007.

[47] S. Saeb, P. Steinmann, and A. Javili. Aspects of computational homogenization at finite deformations: A unifying review from Reuss' to Voigt's bound. *Applied Mechanics Reviews*, 68, 20016.

[48] M. Mosby and K. Matouš. Hierarchically parallel coupled finite strain multiscale solver for modeling heterogenerous layers. *International Journal for Numerical Methods in Engineering*, 102:748–765, 2015.

[49] M. Mosby and K. Matouš. Computational homogenization at extreme scales. *Extreme Mechanics Letters*, 6:68–74, 2016.

DISTRIBUTION:

1 MS 0899 Technical Library, 8944 (electronic copy)

This page intentionally left blank.


Sandia National Laboratories