SANDIA REPORT

SAND2015-8312
Unlimited Release
Printed September 2015

ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms

Janine Bennett (PI), Robert Clay (PM), Gavin Baker, Marc Gamell, David Hollman, Samuel Knight, Hemanth Kolla, Gregory Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke (DHARMA Programming Model and Runtime System Research),
Matt Bettencourt, Steve Bova, Ken Franko, Paul Lin (Applications),
Ryan Grant, Si Hammond, Stephen Olivier (Performance Analysis)
Sandia National Laboratories

Laxmikant Kale, Nikhil Jain, Eric Mikida (Charm++)
University of Illinois, Urbana Champaign

Alex Aiken, Mike Bauer, Wonchan Lee, Elliott Slaughter, Sean Treichler (Legion)
Stanford University

Martin Berzins, Todd Harman, Alan Humphrey, John Schmidt, Dan Sunderland (Uintah)
University of Utah

Pat McCormick and Samuel Gutierrez (Tools)
Los Alamos National Laboratory

Martin Schulz, Abhinav Bhatele, David Boehme, Peer-Timo Bremer, Todd Gamblin (Tools)
Lawrence Livermore National Laboratory

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

Sandia National Laboratories

SAND2015-8312
Unlimited Release
Printed September 2015

ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platforms

Janine Bennett (PI), Robert Clay (PM), Gavin Baker, Marc Gamell, David Hollman, Samuel Knight, Hemanth Kolla, Gregory Sjaardema, Nicole Slattengren, Keita Teranishi, Jeremiah Wilke (DHARMA Programming Model and Runtime System Research),
Matt Bettencourt, Steve Bova, Ken Franko, Paul Lin (Applications),
Ryan Grant, Si Hammond, Stephen Olivier (Performance Analysis)
Sandia National Laboratories

Laxmikant Kale, Nikhil Jain, Eric Mikida (Charm++)
University of Illinois, Urbana Champaign

Alex Aiken, Mike Bauer, Wonchan Lee, Elliott Slaughter, Sean Treichler (Legion)
Stanford University

Martin Berzins, Todd Harman, Alan Humphrey, John Schmidt, Dan Sunderland (Uintah)
University of Utah

Pat McCormick and Samuel Gutierrez (Tools)
Los Alamos National Laboratory

Martin Schulz, Abhinav Bhatele, David Boehme, Peer-Timo Bremer, Todd Gamblin (Tools)
Lawrence Livermore National Laboratory

Abstract

This report provides in-depth information and analysis to help create a technical road map for developing next-generation programming models and runtime systems that support Advanced Simulation and Computing (ASC) workload requirements. The focus herein is on asynchronous many-task (AMT) model and runtime systems, which are of great interest in the context of "exascale" computing, as they hold the promise to address key issues associated with future extreme-scale computer architectures. This report includes a thorough qualitative and quantitative examination of three best-of-class AMT runtime systems—Charm++, Legion, and Uintah, all of which are in use as part of the ASC Predictive Science Academic Alliance Program II (PSAAP-II) Centers. The studies focus on each of the runtimes' *programmability*, *performance*, and *mutability*. Through the experiments and analysis presented, several overarching findings emerge. From a performance perspective, AMT runtimes show tremendous potential for addressing extreme-scale challenges. Empirical studies show an AMT runtime can mitigate performance heterogeneity inherent to the machine itself and that Message Passing Interface (MPI) and AMT runtimes perform comparably under balanced conditions. From a programmability and mutability perspective however, none of the runtimes in this study are currently ready for use in developing production-ready Sandia ASC applications. The report concludes by recommending a co-design path forward, wherein application, programming model, and runtime system developers work together to define requirements and solutions. Such a requirements-driven co-design approach benefits the high-performance computing (HPC) community as a whole, with widespread community engagement mitigating risk for both application developers and runtime system developers.

Acknowledgment

This work was supported by the U. S. Department of Energy (DOE) National Nuclear Security Administration (NNSA) ASC program and the DOE Office of Advanced Scientific Computing Research. Sandia National Laboratories (SNL) is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the DOE NNSA under contract DE-AC04-94AL85000. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We thank Dr. Karen Pao for granting the compute allocation at NERSC. We thank Ben Santos for his assistance with the performance runs on the Los Alamos National Laboratory (LANL) Cielo system. We thank Dr. Peter Strazdins for useful feedback on this report. We also thank Dr. Rob Van der Wijngaart, Dr. Tim Mattson, Dr. Abdullah Kayi for interesting and useful conversations in regards to this study.

Contents

List of Figures

Executive Summary

This report presents a qualitative and quantitative examination of three best-of-class asynchronous many-task (AMT) runtime systems—Charm++ [6], Legion [7], and Uintah [8], all of which are in use as part of the Advanced Simulation and Computing (ASC) Predictive Science Academic Alliance Program II (PSAAP-II) Centers. The primary aim of this report is to provide information to help create a technical road map for developing next-generation programming models and runtime systems that support ASC workload requirements. The focus herein is on AMT models and runtime systems, which are of great interest in the context of "exascale" computing, as they hold the promise to address key issues associated with future extreme-scale computer architectures.

Extreme-scale architectures will combine multiple new memory and compute architectures with dynamic power/performance, increasing both the complexity and heterogeneity of future machines. Furthermore, the significant increase in machine concurrency and the anticipated decrease in overall machine reliability motivate the need to efficiently distribute application workloads in a fault tolerant manner. Taken together, these changes present serious challenges to current ASC application codes. In particular, the procedural and imperative nature of Message Passing Interface (MPI)-based applications requires the management of machine performance heterogeneity, fault tolerance and increasingly complex workflows at the *application-level*. AMT models and associated runtime systems are a leading alternative to current practice that promise to mitigate exascale challenges at the *runtime system-level*, sheltering the application developer from the complexities introduced by future architectures.

Although the asynchronous many-task runtime system (AMT RTS) research community is very active [6–11] a comprehensive comparison of existing runtime systems is lacking. This milestone research seeks to address this gap by thoroughly examining three AMT RTS as alternatives to current practice in the context of ASC workloads. The runtimes selected for this study cover a spectrum of low-level flexibility to domain-specific expression. 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 demonstrates the potential optimization of a domain-specific runtime. MiniAero[1] was used as a basis for this study, and its functionality was implemented using each of the Charm++, Legion, and Uintah runtimes (replacing MPI for all inter- and intra-processor communication). Using these implementations, the three runtimes are each 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 exascale challenges?

Mutability: What is the ease of adopting this runtime and modifying it to suit ASC/ATDM needs?

The analysis regarding programmability and mutability is largely subjective; the associated measures may vary over time, across laboratories, and individual application areas. Although this report summarizes the work of a large number of contributors (from various institutions and runtime system research efforts), the subjective analysis contained herein reflects the opinions and conclusions drawn only by the Sandia DHARMA programming model and runtime system research team[2]. As such, unless otherwise specified, first-person pronouns such as "we" and "us" refer to this core team. Although subjective, these opinions and analysis evaluate the principles and practices of a runtime system, providing a mechanism for the DHARMA team to avoid straw man arguments that would declare an AMT RTS "better" or "worse" based on isolated comparisons of performance studies.

Through the experiments and analysis presented in this report, several overarching findings emerge. From a performance perspective, AMT runtimes show tremendous potential for addressing extreme-scale challenges. Empirical

[1]MiniAero is a three-dimensional, unstructured, finite volume, computational fluid dynamics mini application. It is representative of a part of the computational requirements for Sandia's ASC/Advanced Technology Development and Mitigation (ATDM) re-entry application.

[2]i.e., the subjective analysis may not be representative of the individual runtime teams' opinions

studies show an AMT RTS can mitigate performance heterogeneity inherent to the machine itself[3] and that MPI and AMT runtimes perform comparably under balanced conditions. From a programmability and mutability perspective however, none of the runtimes are currently ready for use in developing production-ready Sandia ASC applications. Legion is still relatively immature and undergoing rapid development and feature addition. Uintah is targeted at Cartesian structured mesh applications, but the majority of the Sandia ASC applications use unstructured or hybrid meshes. Charm++ will require additional effort, with new abstractions as well as improved component implementations, to realize its full potential. Note that in different domains, each of the AMT runtimes have been used for production-level applications.

Each of the runtimes make trade-offs between higher-level constructs and low-level flexibility to strike their own balance of code performance, correctness, and programmer productivity. Consequently, these trade-offs affect aspects of how and where concurrency is created and managed. Charm++ falls on one side of the spectrum with the management of data and concurrent data accesses falling largely to the application developer. This provides tremendous flexibility, but also adds complexity in a number of application settings. At the other end of the spectrum is Legion, where the runtime assumes as much control as possible of concurrency creation and management. For performance reasons, there are application use cases that are not well suited to this extreme, and the Legion team has begun to introduce mechanisms to relinquish control to the application in some settings.

The findings in this report suggest that there is a critical design issue facing runtime development. Namely, should there be a single execution style for the runtime, forcing applications to accommodate and adapt; or should the runtime accommodate and adapt to several execution styles suited to many applications? A third option could involve developing several runtimes, each optimized for different application workloads. The community requires a significantly more comprehensive understanding of the interplay between the various AMT concurrency management schemes and their associated performance and productivity impacts (across a variety of applications and architectures) to make a confident decision regarding this design issue that will serve long term interests.

The DHARMA team believes this comprehensive understanding can be achieved via a concerted co-design effort between application, programming model, and runtime developers centered on common concepts and vocabulary for discussing requirements. Such a co-design approach allows for ASC application workload requirements to directly impact the design decisions of any programming model and runtime system that is adopted. Although there are many possible ways for the application, programming model, and runtime system developers to co-design solutions, we recommend a path forward in which application requirements are clearly articulated in terms of programming model and runtime system features. The current co-design approach of applications providing terse algorithmic descriptions along with MPI baseline mini-applications is useful but does not suffice. Instead, we believe developers from a representative set of application areas should work closely with programming models teams to co-design a community adopted AMT programming model specification. This specification would provide 1) a concrete application programmer interface (API) to facilitate the gathering of application requirements, and 2) an effective means for communicating those requirements to the AMT community. Development of this specification is already underway for the DHARMA team and is a key component of our technical road map.

In order for this approach to be successful, the AMT RTS community must establish a common vocabulary for expressing application requirements. A shared vocabulary for common concurrency concepts is a critical prerequisite to establishing shared best practices. Not only does a common vocabulary and programming model specification facilitate co-design interactions across a broad class of application areas, it provides a mechanism for current AMT research efforts to compare and contrast their results in a more rigorous manner. Key to the overall success of this approach, is the adoption or buy-in from representative AMT RTS teams.

We believe a requirements-driven co-design approach benefits the high-performance computing (HPC) community as a whole, and that widespread community engagement mitigates risk for both application developers and runtime system developers and vendors. Application developers need only write their applications to a single API—that they can directly shape. Application developers further benefit from this approach as it greatly simplifies the process of assessing various AMT runtime implementations. In particular, it enables them to rapidly switch between implementations on various architectures based on performance and other considerations. From the perspective of the AMT RTS teams, this approach greatly facilitates the transition to and the adoption of AMT technologies, helping the AMT RTS teams ensure a potential long term user base for their runtime systems.

[3]Although the experiments in this report are with static workloads, there are other studies that show the AMT RTS can mitigate performance heterogeneity inherent in the application [12–14].

Chapter 1

Introduction

1.1 Motivation: Exascale Drivers

Sandia science and engineering codes must adapt to rapidly developing extreme-scale computer architectures to ensure efficient and productive use of future high-performance computing (HPC) machines. Relative to current practice, both the hardware and future algorithms will be characterized by dynamic behavior and a lack of uniformity.

Heterogeneous Machine Architectures Future architectures will combine multiple accelerators, multi-level memories, potential scratchpad and processing in memory, high bandwidth optical interconnects, and possibly even system-on-chip fabrics. Figure 1.1 is an abstract machine model of an exascale node from [1] that illustrates some of these complexities. As a result of these architectural changes, overall system concurrency may increase by a factor of 40,000-400,000, as shown in Figure 1.2 from [2, 3], with energy constraints leading to power capping or even near-threshold voltage (NTV) [15] architectures, producing highly non-uniform node-level performance.

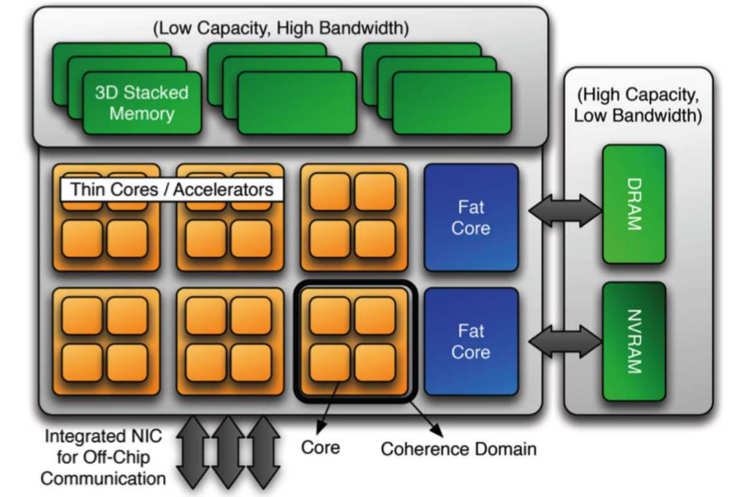

(Low Capacity, High Bandwidth)
3D Stacked Memory
(High Capacity, Low Bandwidth)
Thin Cores / Accelerators
Fat Core
Fat Core
DRAM
NVRAM
Integrated NIC for Off-Chip Communication
Core
Coherence Domain

Figure 1.1: Abstract machine model of a projected exascale node architecture as presented in [1].

Dynamic Workloads Seeking to exploit all available performance, the national laboratories are investing in algorithms that exhibit highly variable computational loads and a mixture of inherent task- and data-parallelism (including electromagnetic particle in cell, molecular dynamics, and structural mechanics contact applications). Along with increased (and distinct forms) of concurrency, Input/Output (I/O) constraints will increasingly limit performance due to widening compute and I/O performance disparities (see Table 1.2). In an effort to mitigate against the widening discrepancy between compute and I/O capabilities on future machines, the laboratories are also developing increasingly complex and dynamic workflows that include *in-situ* analysis and multi-physics coupling of codes. The design space of dynamic applications, *in-situ* analysis, and multi-physics coupling demands new runtime system solutions to maximize programmer productivity and code performance.

System Parameter	2011	2018		Factor Change
System Peak	2 Pf/s	1 Ef/s		**500**
Power	6 MW	$\leq$ 20 MW		3
System Memory	0.3 PB	32-64 PB		100-200
Total Concurrency	225K	1B$\times$ 10	1B $\times$ 100	**40000-400000**
Node Performance	125 GF	1TF	10 TF	8-80
Node Concurrency	12	1000	10000	83-830
Network Bandwidth	1.5 GB/s	100 GB/s	1000 GB/s	66-660
System Size (nodes)	18700	1000000	100000	50-500
I/O Capacity	15 PB	30-100 PB		**20-67**
I/O Bandwidth	0.2 TB/s	20-60 TB/s		**10-30**

Figure 1.2: Expected exascale architecture parameters for the design of two "swim lanes" of very different design choices [2,3]. Note the drastic difference between expected improvements in I/O and compute capacities in both swim lanes.

Figure 1.3 illustrates four regimes captured by the cross product of machine performance and workload characteristics. A large portion of Advanced Simulation and Computing (ASC) workloads have historically assumed static homogeneous machine performance, with dynamic parallelism requirements stemming solely from the workload. As we move to next generation platforms, we are entering a regime where both the workloads and the machine performance characteristics are increasingly dynamic in nature.

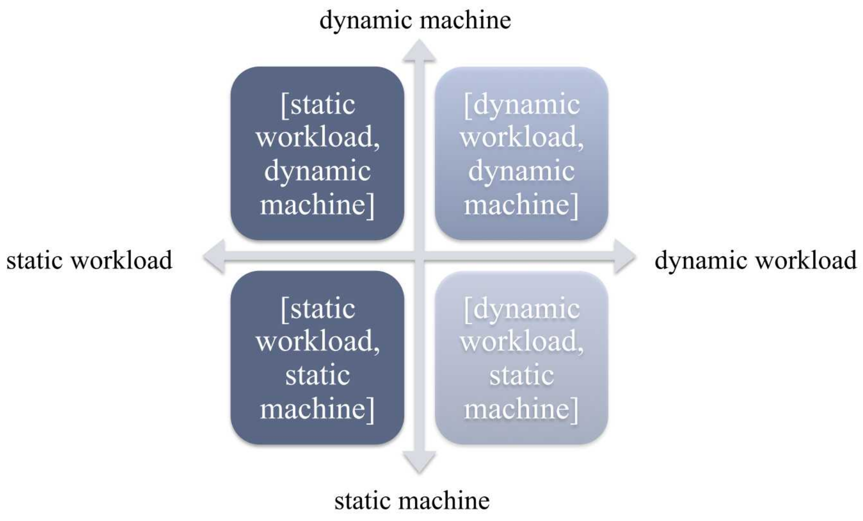

dynamic machine
[static workload, dynamic machine]
[dynamic workload, dynamic machine]
static workload
dynamic workload
[static workload, static machine]
[dynamic workload, static machine]
static machine

Figure 1.3: Four regimes are defined by the cross product of machine performance and workload characteristics.

1.2 Background and Terminology

This Advanced Technology Development and Mitigation (ATDM) Level 2 milestone lays the groundwork necessary for Sandia to develop a technical roadmap in the context of next generation programming models, execution models, and runtime systems. The HPC community often uses these and other terms interchangeably, which can result in confusion at times. This section introduces and defines some of the terminology that will be used throughout this report. We begin with a discussion of concurrency—this is often referred to generically without concern for where it comes from. *Data parallelism* involves carrying out a single task and/or instruction on different segments of data across many computational units. The terms single-instruction, multiple-data (SIMD) and single-program multiple-data (SPMD) describe different instantiations of data parallelism. SIMD refers to a type of instruction level parallelism where an individual instruction is synchronously executed on different segments of data and is best illustrated by vector processing on a central processing unit (CPU) or Many Integrated Core Architecture (MIC). In SPMD the same tasks are carried out by multiple processing units but operate on different sets of input data. Examples of this are multithreading on a single compute node and/or distributed computing using Message Passing Interface (MPI) communication. *Task parallelism* focuses on completing multiple tasks simultaneously over different computational units. These tasks may operate on the same segment of data or many different datasets. In particular, task parallelism can occur when non-conflicting tasks operate on the same data, usually because they only require read-only access. At a process level, this is a form of *multiple-program multiple-data (MPMD)*. *Pipeline parallelism* is achieved by breaking up a task into a sequence of individual sub-tasks, each of which represents a stage whose execution can be overlapped. Pipeline parallelism is most often associated with data movement operations, overlapping data fetches with computational work to hide latency and minimize gaps in the task pipeline. This can occur for both on-node and remote memory fetches.

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 invocations (entry methods). Legion decomposes problems in a data-centric way with logical regions. In Legion, parallel coordination is implicitly expressed via data dependencies.

The parallel programming model provides the mechanisms for an application to *express* concurrency. Programming models are often characterized according to their style, for example *imperative*, *declarative*, *procedural*, or *functional*. In an imperative style of programming, statements explicitly change the state of a program to produce a specific result. The programmer explicitly expresses how an operation is to be performed. This contrasts to declarative programming in which the programmer expresses or defines the desired result without specifying how the result is to be achieved. In a procedural programming model 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. A functional programming model on the other hand, is a style of programming that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. Other programming styles or paradigms exist, and a full characterization is beyond the scope of this report, see [16] for additional details. A *programming language* is a syntax and code constructs for implementing one or more programming models. For example, the C++ programming language supports both functional and procedural imperative programming models. A domain specific language (DSL) is a programming language that has a number of abstractions in place that have been specialized to a particular application domain.

A parallel *execution model* specifies how an application creates and manages concurrency. Examples of various execution models include *communicating sequential processes (CSP)*, strict *fork-join*, the *actor model*, and *event-based models*. CSP is the most popular concurrency model for science and engineering applications, often being synonymous with SPMD. Fork-join is 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. Actor and event-based models cover aspects from both programming and execution models. In an actor model, applications are decomposed across objects called actors rather than processes or threads (MPI ranks). Actors send messages to other actors, but beyond simply exchanging data, they can make remote procedure invocations to create remote work or even spawn new actors. The actor model mixes aspects of 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 an event-based model an application is expressed and managed as a set of events with precedence constraints, often taking the form of a directed graph of event dependencies. The different execution model classifications distinguish how concurrency is created. For example, in CSP, many parallel workers begin simultaneously and synchronize to reduce concurrency, whereas in fork-join a single top-level worker forks new tasks to increase concurrency. The execution model classifications also distinguish how parallel data access hazards are managed. We note that in imperative styles of programming, the programming model and execution model are closely tied and therefore not distinguished. The non-specific term *parallel model* can be applied in these settings. A declarative programming style decouples the execution model from the programming model.

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, parallel hazards, task ordering, and data movement. A runtime comprises all aspects of parallel execution that are not explicitly managed by the application. We borrow the terms high-level runtime (HLR) and low-level runtime (LLR) from Legion to distinguish implicit and explicit runtime behavior. A high-level runtime is generally any aspect of the runtime system that implicitly creates concurrency via higher-level logic based on what is expressed via the application programming model. High-level runtimes generally involve data, task, and machine models expressed in a declarative fashion through which the runtime reasons about application concurrency. This implicit creation of concurrency differs from the LLR, which only executes operations explicitly specified. The LLR is only responsible for ensuring that data movement and task scheduling operations satisfy explicit precedence constraints. The terms runtime system, runtime, and RTS are often used interchangeably, and all of these variants will be used throughout this report. We often refer to generically to the RTS as a whole and generally do not distinguish the HLR and LLR portions.

An *asynchronous many-task (AMT) model* 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 asynchronous many-task (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.

1.3 Motivation and Approach

As was already mentioned, ASC workloads have historically assumed static homogeneous, system performance, with dynamic parallelism requirements stemming solely from the work load. Consequently, ASC codes typically follow the CSP programming model using MPI. We have seen over the years that MPI is highly flexible and adaptable as evidenced by the great progress already being made by MPI+X [17]. However, the procedural and imperative nature of current programming models and runtime systems will require the management of system performance heterogeneity, fault tolerance, and increasingly complex workflows at the *application-level*. AMT models and associated runtime systems are a leading alternative to the traditional approach that promise to mitigate extreme-scale challenges at the *runtime system-level*, sheltering the application developer from the complexities introduced by future architectures. AMT RTS present an opportunity for applications and hardware to interact through a flexible, intrinsically dynamic runtime and programming environment. Numerous AMT RTS such as Cilk [18] or later versions of OpenMP [19] have demonstrated the performance improvements achievable through dynamic many-task parallelism at the node-level. However, the dynamic algorithms and hardware challenges outlined above cross distributed-memory boundaries and therefore demand a machine-level runtime solution.

This ATDM Level 2 milestone lays the groundwork necessary for Sandia to develop a technical roadmap in the context of next generation programming and execution models. This study focuses on AMT RTS, which have a very active research community [6–11]. However, while many of these runtime systems may have underlying concepts that are similar, a comprehensive comparison of both their programming and execution models is lacking. This milestone research seeks to address this gap by thoroughly examining three AMT RTS as alternatives to current practice in the context of ASC workloads.

1.3.1 AMT Runtimes

In the summer of FY14, a number of AMT RTS were considered for this study. The three exemplar runtimes chosen cover a spectrum of low-level flexibility to domain-specific expression: Charm++ [6], Legion [7], and Uintah [8]. These runtimes were selected because 1) they provide three very different implementations, application programmer interface (API)s, and abstractions, 2) each has demonstrated results on science applications at scale, and 3) their teams were incredibly responsive and committed to providing the feedback and engagement required for this study to be successful. The following is a brief description of the team structure and history of each runtime.

Charm++ Charm++ is an actor model with low-level flexibility, replacing message passing with remote procedure invocations. The Charm++ effort is the most mature of the runtimes studied, with the first Charm++ papers published in 1990. Prof. Laxmikant Kale leads this effort out of the Parallel Programming Laboratory at the University of Illinois, Urbana Champaign. Nearly a hundred people have contributed to Charm++ over the course of the last 20 years, with many of these contributors now members at various U. S. Department of Energy (DOE) laboratories. The current research group comprises approximately 20 people who are actively engaged in maintaining and extending Charm++ with a focus on its various frameworks, including adaptive mesh refinement (AMR), the unstructured meshing framework, and the parallel state space search engine.

Legion Legion is a data-centric programming model with higher-level constructs, representing a strong shift from the procedural style of MPI and Charm++ to a highly declarative program expression. The Legion effort began in 2011 with two researchers at Stanford, growing out of earlier work on the data-centric Sequoia [20] language. It has since grown to a team of at least ten active developers. Members of Professor Alex Aiken's research group still represent the bulk of the team, but they are now joined by contributors at Los Alamos National Laboratory, NVIDIA Research, UC Santa Cruz, and the University of Utah. The focus of the current developers is primarily on programming model research—it is expected that additional developers will be added to help with tasks that are more "engineering" than research. In addition to the research efforts actively coordinated by the team at Stanford, several independent research efforts involving Legion are under way, including teams at Northwestern and Carnegie Mellon University.

Uintah Although throughout this report Uintah is referred to as a runtime, it is in fact a true *framework*, with a number of components, libraries, and additional abstractions provided to support a specific application domain; the components include a scheduler and abstract task graph representing the runtime. While not a true DSL, it demonstrates the potential optimization of a domain-specific runtime. The Uintah effort began in 1998 with an initial design by Steve Parker and has been in continuous evolution since then at the University of Utah. It is currently led by Prof. Martin Berzins within the Scientific Computing and Imaging (SCI) Institute. The development team comprises a mix of application and computer scientists. The design decisions and development for Uintah has always been driven by their application and computer hardware needs, with a goal of solving challenging engineering applications at the appropriate resolution and hence hardware scales. As a side effect some of the application codes have not needed to change as they have been moved from 600 to 600K cores. A large part of their current research is focused on exploiting accelerator performance on heterogeneous architectures.

1.3.2 MiniAero

Sandia's ATDM program is focused on two application drivers: re-entry and electromagnetic particle-in-cell. MiniAero [21, 22] is used as basis for this study. MiniAero is a compressible Navier-Stokes, three-dimensional, unstructured mesh, finite volume, explicit, computational fluid dynamics (CFD) mini application that is representative of a part of the computational requirements for the re-entry application. There is a baseline implementation of MiniAero available online at [23] that is approximately 3800 lines of C++ code, using MPI+Kokkos [24]. MiniAero solves the Compressible Navier-Stokes equations using Runga-Kutta fourth-order time marching and provides options for 1st or 2nd order spatial discretization of inviscid fluxes (employing Roe's approximate Riemann solver). The boundary conditions include supersonic inflow, supersonic outflow, and tangent flow.

Task-DAG representation Most many-task runtime schedulers work with a directed acyclic graph (DAG), often referred to as a task-DAG that encodes all the precedence constraints in a program. Each node in the DAG represents computational work. A basic task graph for MiniAero is shown in Figure 1.4. Precedence constraints are expressed via directed edges in the graph, indicating which tasks must complete before other tasks can begin. If there is no edge (precedence constraint) between two tasks then they can safely run in parallel. In the convention here, the task execution flows down. In general, fewer precedence constraints lead to a wider task-DAG which has more concurrency to exploit. The task graph in Figure 1.4 is concise description of the algorithm, showing only computational work. In Uintah and in the Realm runtime in Legion, the term "operations graph" is more appropriate since data movement and copy operations are also included as events with precursors (the operations graph only exists implicitly in Charm++.

For simplicity, only a single step of a Runga-Kutta time integration is shown in Figure 1.4, representing a single residual computation. A set of fluxes is computed on the faces of a hexahedral cell. These face-centered fluxes are computed based on cell-centered quantities.

The Compressible Navier-Stokes equations give equations for the Mass, Momentum, and Energy:

$$\begin{aligned}
\frac{\partial \rho}{\partial t} + \frac{\partial \rho u_j}{\partial x_j} &= 0 && \text{Mass}\\
\frac{\partial \rho u_i}{\partial t} + \frac{\partial}{\partial x_j}(\rho u_i u_j + P\delta_{ij}) &= \frac{\partial \tau_{ij}}{\partial x_j} && \text{Momentum}\\
\frac{\partial \rho E}{\partial t} + \frac{\partial \rho u_j H}{\partial x_j} &= -\frac{\partial q_j}{\partial x_j} + \frac{\partial u_j \tau_{ij}}{\partial x_j} && \text{Energy}
\end{aligned}$$

The mass, momentum vector, and energy are represented as a five-component solution vector $\vec{U}$ which is updated from a set of residuals:

$$\begin{aligned}
\frac{d\vec{U}}{dt} &= R(\vec{U}) \qquad \vec{U}(t_0) = \vec{U}_0\\
\vec{U}_{n+1} &= \vec{U}_n + \frac{\Delta t}{6}(\vec{k}_1 + 2\vec{k}_2 + 2\vec{k}_3 + \vec{k}_4)\\
\vec{k}_1 &= R(\vec{U}_n)\\
\vec{k}_2 &= R(\vec{U}_n + \tfrac{\Delta t}{2}\vec{k}_1)\\
\vec{k}_3 &= R(\vec{U}_n + \tfrac{\Delta t}{2}\vec{k}_2)\\
\vec{k}_4 &= R(\vec{U}_n + \Delta t\vec{k}_3)
\end{aligned}$$

where Δt is the time step and R is a residual computation. Each of the quantities involved is a large, data-parallel vector with each entry being a cell-averaged value. As cell-centered values are updated, face-centered fluxes are recomputed and used in the residual computation. The data-flow graph in Figure 1.5 shows the dependencies used in each task, building a residual from the face-centered fluxes. The fluxes are five-component vector quantities (mass, momentum vector, and energy). For the viscous and 2nd-order terms, a matrix of spatial gradients is required consisting of the x, y, and z components of each of the mass, momentum vector, and energy quantities. Once the residual is computed, it can be summed into the original values to obtain new momentum, mass, and energy.

In its most basic form, the task-DAG for MiniAero is fairly narrow. It is possible to give the task graph more breadth by computing the mass, momentum, and energy in separate tasks. However, all quantities are coupled in the residual computations so it actually provides no practical benefit to divide the tasks (although in other applications this form

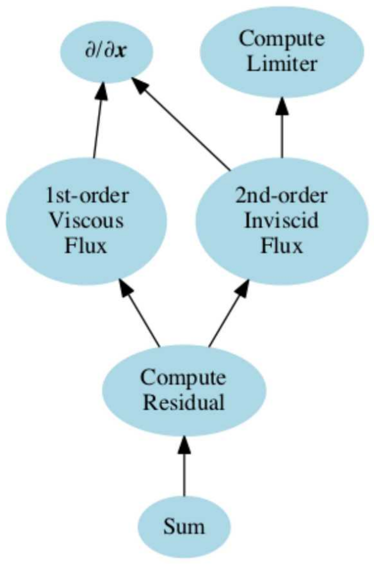


Figure 1.4: Task graph for MiniAero finite volume, explicit aerodynamics code using 2nd-order inviscid/1st-order viscous terms. Very little breadth is available in the graph to exploit task-level concurrency. Arrow direction indicates task depends on precursor.

of task parallelism may provide benefits [25]). Concurrency will be driven almost exclusively by data parallelism for large problems. A data parallel task graph for MiniAero can be achieved by encapsulating the flux, limiter, and gradient tasks into a single "residual" task. Figure 1.6 shows the first few Runga-Kutta steps for a problem with 3-way data parallelism, splitting the solution into three vector chunks. Thus, as is the case with many ghost-exchange computations, after the exchange phase with many cross-dependencies, the residual and update tasks proceed down independent branches. Thus, given data parallelism, MiniAero actually exhibits some depth-wise parallelism. This data parallelism can be discovered automatically by an AMT runtime system given the data dependencies and restrictions of each task. The studies in Section 3.3 explore how varying degrees of data parallelism (e.g., different levels of overdecomposition) enable overlap of communication and computation and effective load balancing.

To summarize, MiniAero has an inherently narrow task graph and is, at its core a very static application, amenable to a traditional SPMD implementation. In spite of this, it presents an interesting use case for this study. This report comprehensively assesses Charm++, Legion, and Uintah in the context of the first column of the quad chart in Figure 1.3, shown in dark blue. Given the static workload of MiniAero, the performance studies have designed to test 1) whether or not the AMT runtimes perform comparably to the baseline MPI implementation on machines with homogeneous performance, and 2) whether or not the runtime systems can mitigate against performance heterogeneity in the machine when it exists. It is noted that the static nature of the underlying MiniAero algorithm does not impact the assessment of the other runtime system performance measures in this study (i.e., fault tolerance and dynamic workflows).

1.3.3 Milestone Implementation Details

The functionality of MiniAero was implemented using each of the Charm++, Legion, and Uintah runtimes (replacing MPI for all inter and intra-processor communication). Using these implementations, the three runtimes are each 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 exascale challenges?

Mutability: What is the ease of adopting this runtime and modifying it to suit ASC/ATDM needs?

Additional details regarding the evaluation strategy in each of these areas is included in the following chapters.

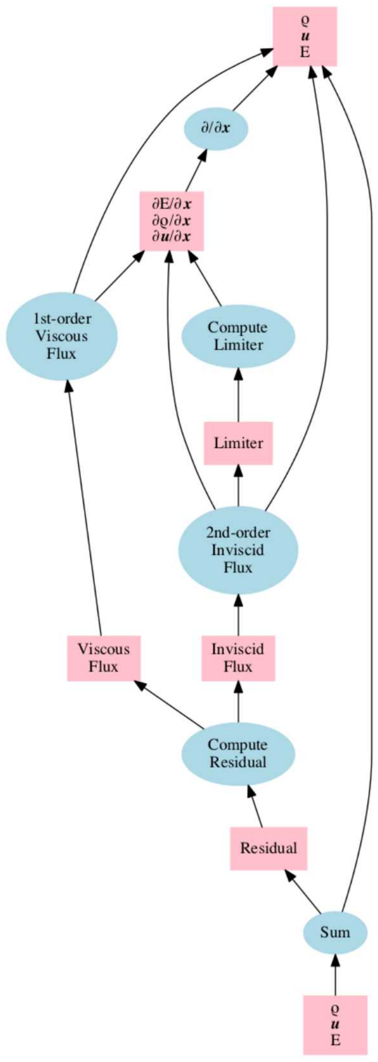


Figure 1.5: Data flow dependency graph for MiniAero finite volume, explicit aerodynamics code using 2nd-order inviscid/1st-order viscous terms. Momentum, energy, and mass are not treated as separate quantities. Here blue, oval nodes represent tasks and pink, rectangular nodes represent data. Arrow direction indicates task depends on precursor.

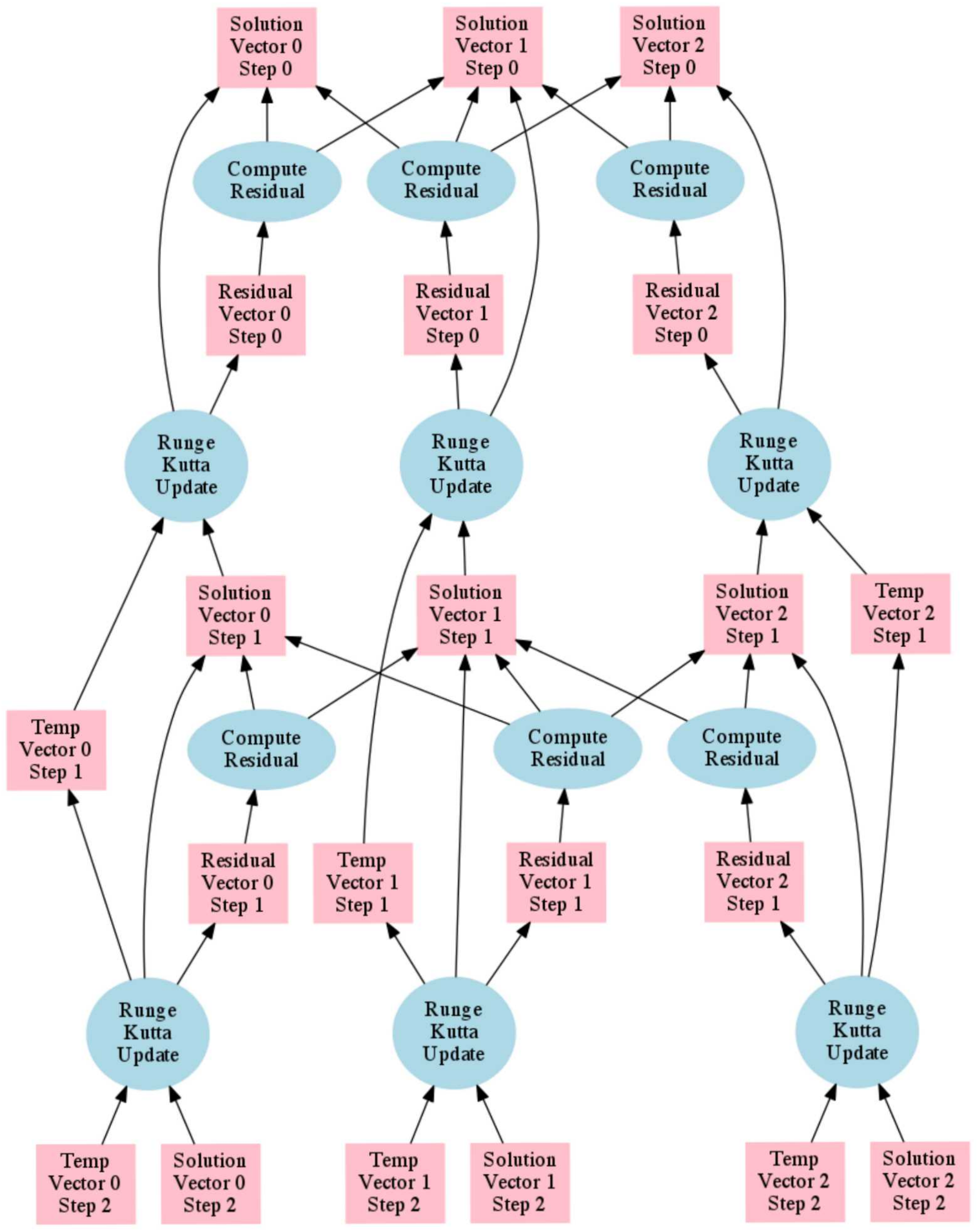


Figure 1.6: Data flow dependency graph for MiniAero finite volume, explicit aerodynamics code. Momentum, energy, and mass are treated as three different quantities. Blue, oval nodes represent tasks and pink, rectangular nodes represent data.

From a philosophical perspective, this milestone was approached as a unique opportunity to engage the AMT RTS community in a dialogue towards best practices, with an eye towards eventual standards. As such, experiment design and execution have been rigorous to ensure reproducibility of results. Furthermore, each of the MiniAero implementations will be made available at `http://mantevo.org` (pending the Sandia legal release process).

The large author list on this report reflects the broad AMT RTS, application, and tools community involvement the DHARMA programming model and runtime system research team at Sandia was fortunate to engage for this study. In particular, the individual runtime teams provided tremendous support throughout the course of this study. To ensure the capabilities of each runtime were properly leveraged, a series of coding bootcamps were held. The Uintah bootcamp was November 10–12, 2014 at the University of Utah, the Legion bootcamp was December 4–5, 2014 at Stanford, and the Charm++ bootcamp was March 9–12 at Sandia, CA. The core DHARMA research team doing the implementation and analysis comprised a mix of application and computer scientists. Application scientists from other ATDM application areas, including thermo-mechanical and electromagnetic particle in cell, attended each of the bootcamps. While most of the analysis in this report is based on the DHARMA team's experiences with MiniAero, the results reflect perspectives and feedback from these other application areas as well.

While there is substantial summary information and empirical results in this report, it is important to note that the analysis regarding programmability and mutability is largely subjective; the associated measures may vary over time, across laboratories, and individuals application areas. Given the large author list (from varied institutions), note that the subjective analysis contained herein reflects the opinions and conclusions drawn by the DHARMA team only (i.e., the subjective opinions may not be representative of the individual runtime teams themselves). This subjective analysis is a critical component of this report however, for by evaluating the principles and practices of a runtime system, the DHARMA team seeks to avoid straw man arguments that would declare an AMT RTS "better" or "worse" based on isolated comparisons of performance studies.

The next three chapters summarize milestone findings on programmability, performance, and mutability respectively. Finally, the DHARMA team presents conclusions and recommendations going forward for Sandia's technical roadmap in the context of next generation programming model, execution model, and runtime systems.

Chapter 2

Programmability

2.1 Approach for Measuring Programmability

As we assess the programmability of each runtime, we seek to answer the following question:

Does this runtime enable the efficient expression of ASC/ATDM workloads?

In this chapter we describe the key design decisions, abstractions, and controls provided by each runtime. We highlight the mechanisms by which the runtime enables performance portability and discuss how the runtime system maturity affects application development. After highlighting current research efforts for each runtime system, we provide a comparative analysis across the three runtimes by summarizing the responses to a set of subjective questions regarding the programming experience. We comment on the implementation times and the learning curves for the three runtimes, and then conclude with a summary of the state of the art in debugging and performance analysis tools for AMT runtimes. We discuss common tools used in the HPC community, the tools provided by the runtime themselves, and highlight the challenges and future research directions for the tools community in the context of AMT runtimes.

2.2 Charm++ Programmability

2.2.1 Key Design Decisions

Charm++ is an AMT runtime system fundamentally designed around the migratable-objects programming model [26] and the actor execution model [27]. The actor execution model differs subtly from the CSP model [28] of MPI: with CSP, a worker is typically considered to be active until it reaches a synchronization or communication point, whereas in the actor model, workers ("actors") are considered *inactive* until they receive a message. Given the subtlety of this difference, it is not surprising that the experience of programming in Charm++ resembles that of MPI more than any of the other runtime systems we studied.

The migratable objects in Charm++ are known as "chares," and there is essentially a one-to-one mapping of chares to actors. At the most basic level, a chare is just a C++ object, and many of the fundamental principles of C++ objects involving encapsulation of data and functionality also apply to chares. As such, the primary supported data concurrency model is copy-on-read. With some exceptions and nuances, each chare acts only on its own data. Any other data needed must, in general, be copied via Parameter Marshalling. At any given time, at most one non-preemptible unit of work associated with a given chare may be executing, though units of work from different chares may be executing concurrently. Thus, data safety is guaranteed by ensuring *a priori* that no more than one unit of work may be executing on a given piece of data at a given time. In this execution model, ordering dependencies on data are not distinguished from execution ordering dependencies, and still must be handled manually, as discussed below.

2.2.2 Abstractions and Controls

The basic unit of parallel computation in Charm++ is the chare. According to the Charm++ manual, "A Charm++ computation consists of a large number of chares distributed on available processors of the machine, and interacting with each other via asynchronous method invocations." [29] These methods of a chare object that may be invoked

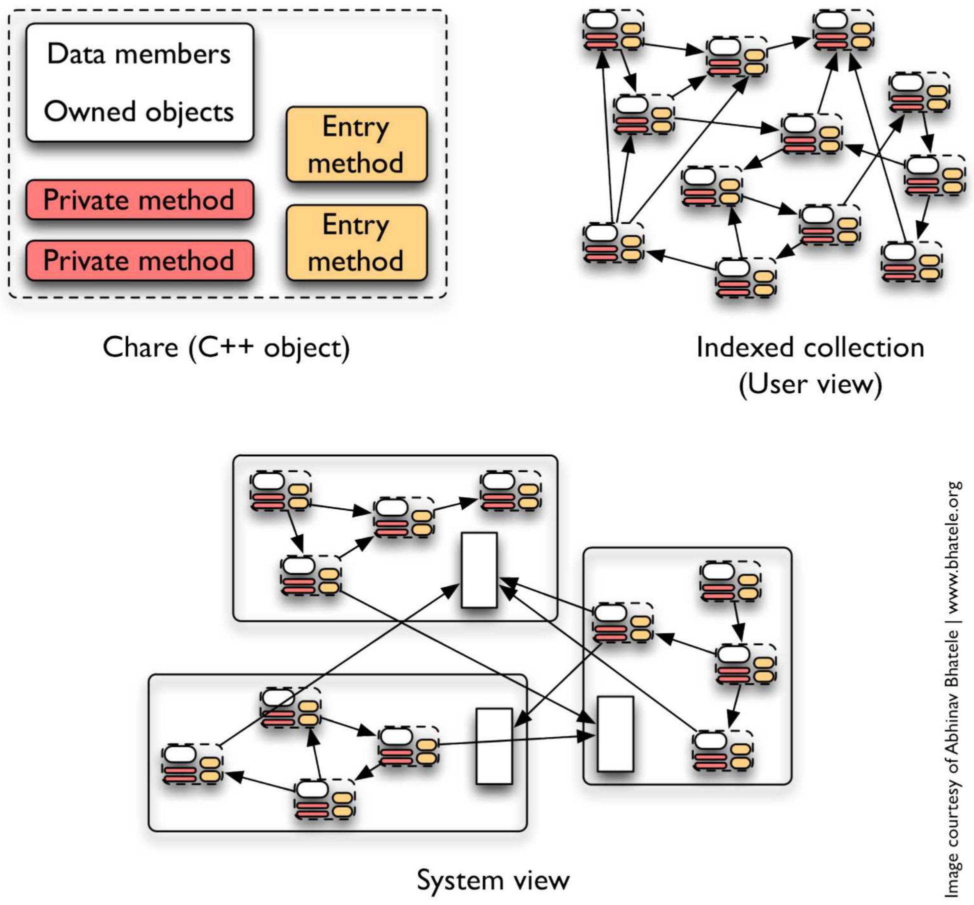


Figure 2.1: This image (courtesy of Abhinav Bhatele) illustrates key Charm++ abstractions. Chares are the basic unit of parallel work in Charm++. Chares are C++ objects with entry methods that can be invoked remotely by other chares. The user expresses parallelism via interacting collections of chares, without requiring awareness regarding their physical layout on the machine. The Charm++ runtime system is introspective and migrates chares around the machine to optimize performance.

remotely are known as “entry” methods. Entry method invocation is performed asynchronously, in keeping with the non-preemptible nature of work units in Charm++. Asynchronous entry method invocation on a remote chare is essentially equivalent to remote procedure calls (RPC) or active message passing. The parameters to a remote method invocation are automatically marshalled on the sender side (serialized into a packed buffer) and unmarshalled by the recipient. For user-defined objects, Charm++ provides a serialization interface known as the “PUP” (for Pack-UnPack) framework, by which users can specify how the runtime system should marshall an object when needed. Figure 2.1 illustrates some of the key abstractions in the Charm++ runtime system.

The specification by the programmer of parallel workflow in Charm++ is primarily done using a specialized “charm interface” mini-language (written in files with the extension “.ci” and thus referred to as ci files). The runtime system cross-compiles these ci files into C++ code, which a standard C++ compiler can then compile into the main executable. Programmers declare most of the programming model constructs in these files, including chares and entry methods on those chares, so that the runtime can correctly associate remote entry method invocation messages with the proper C++ implementation of that method. Beyond this basic usage, the ci file syntax allows the programmer to write event-driven code, in which execution flow proceeds based on the satisfaction of certain preconditions, expressed using the `when` construct. The preconditions in these `when` constructs are specified with one or more entry-method-like signatures, and they are satisfied by asynchronous invocation, either remotely or locally, of these methods. Thus, the `entry` keyword serves two purposes: 1) declaring functions that can be invoked remotely and allocating the necessary runtime metadata, and 2) declaring preconditions for the execution of one or more blocks of code in the ci file. Both uses are strictly orthogonal. A precondition `entry` method cannot have custom user code to be executed upon remote method invocation, though the sender-side syntax and implementation is the same in both cases. This dual usage was confusing to our team, at least initially.

The ci file specification is currently not compatible with all the rich template metaprogramming features in C++, creating significant hurdles for porting codes of interest to Sandia. Because the ci file is cross-compiled without any input from the actual C++ compiler, writing generic, templated entry methods and chares is awkward in some cases and not possible in others. This limitation stems from subtleties in generating unique and consistent indices for each entry method, which requires fully specified C++ types, not generic template types. In simple cases this limitation can be overcome by explicit user template specialization, but complicated cases with many specializations are infeasible. We have attempted workarounds that convert compile-time polymorphism into runtime polymorphism via type-casting that are compatible with the ci framework, but such an approach is awkward, may sacrifice compile-time optimizations, and is not tenable long-term. In general, the severity of the ci file limitation will depend heavily on application requirements. The Charm++ developers are cognizant of this limitation and are working to resolve it.

Unsurprisingly, the similarities to the MPI execution models made Charm++ a comfortable programming model for team members who have developed intuition for MPI. However, the lack of data-driven semantics in the programming model comes at a significant cost, and many of these drawbacks are similar to those inherent in the MPI programming model. Data-flow dependencies and control-flow anti-dependencies must be explicitly specified using `when` constructs and explicitly satisfied using messages. Charm++ has no built-in concept of a data block. In fact, there is no distinction at all in Charm++ between execution flow and data flow, so the runtime has no way of making scheduling decisions based on data locality. Though Charm++ provides a robust runtime system for the user to give both compile-time and run-time hints about scheduling priorities, the determination of these priorities must be done entirely at the user level and without any data state or access permissions feedback from the runtime. This is less challenging than it sounds because the Charm++ execution model does not allow multiple tasks on the same Chare to execute concurrently (i.e., priorities need only hint at a reasonable linear schedule rather than provide information about the cost of concurrent execution), but the lack of data-based scheduling hints is nonetheless a drawback of the Charm++ design. Given its core copy-on-read design, Charm++ also has minimal support for zero-copy transfers of large data blocks. Any permissions or exclusivity associated with this data must be managed entirely at the user level, essentially breaking the core Charm++ programming model.

Nevertheless, the Charm++ programming model still offers advantages over MPI for performance and resilience. The pervasive use of Parameter Marshalling in Charm++ essentially requires the user to specify a correct serialization of every actor and object used in the application. This serialization can just as easily be used to migrate or checkpoint computations, and the Charm++ developers have exploited this feature with significant success. The Charm++ runtime system supports checkpoint-restart resilience in both in-memory (partner-based) and on-disk forms. The ability to serialize and migrate entire actors also allows Charm++ to perform relatively transparent load balancing, requiring only a user hook to specify when a chare is migratable. This can all proceed transparently to the user because algorithms are tied to chare indices rather than physical nodes. Additionally failure recovery can proceed transparently by redistributing the actors on the failed node and updating the chare to physical node mapping. By not rigorously binding chares to physical nodes, Charm++ allows a many-to-one mapping of actors to processing elements for overdecomposition. Though this resource "virtualization" features in many AMT runtimes, Charm++ demonstrates that many extreme-scale computing challenges might be resolved with a more evolutionary shift from CSP to the actor model.

Charm++ supports an extensive and extremely modular interface for both centralized and distributed load balancing. Though Charm++ ostensibly supports both synchronous and continuous load balancing, we only used the synchronous version in our MiniAero implementation. The Charm++ stable release contains dozens of load balancers, and the development branch has many more. Almost all of these load balancers are based on communication and computation work-based introspection via timers and other mechanisms, though Charm++ does provide a mechanism for any chare to feed its own load information to the load balancer through the `UserSetLBLoad()` virtual method. Charm++ also exposes an API for users to write their own load balancers, which have access to all of the same introspection and timing information that the built-in load balancers have. Advanced users can use this API to integrate physics-based load balancing with collected timing and communication information.

Another feature often proposed in AMT RTS motifs is code collaboration and code coupling by strictly partitioning work into well-defined, modular tasks. The basic unit of collaborative modularity in Charm++ (beyond the relatively primitive `module` construct) is the chare. This design decision has positive and negative implications. In most data-driven runtime systems, task-level specification of data requirements and permissions enable significantly greater opacity in code collaboration. However, the lack of data-flow semantics in Charm++ means that any potential simultaneous data access involving, for instance, Read-After-Write or Write-After-Read dependencies must be explicitly managed by user agreement or through user-level constructs. The Charm++ programming model encourages the user to consider individual chares to be disjoint memory spaces (almost like separate processes, since they may be mapped as such), insofar as data members of a chare class are owned by the chare. Thus, opaque collaboration between separate chares is straightforward in Charm++, as long as the user passes data between chares by value. Opaque, simultaneous access to the same data is not supported by the runtime even at the chare level. Nonetheless, user-level management of dependencies and anti-dependencies in Charm++ is not difficult using the event driven interface, given some agreed upon set of data usage event constructs. Charm++ is therefore no different from MPI in that applications must explicitly yield control to *in-situ* analysis tools or explicitly copy data into staging areas for in-transit analysis to avoid race conditions between coupled codes.

2.2.3 Performance Portability

Charm++ is primarily a distributed memory runtime system. As with the "MPI+X" motif, this means that any node-level parallelism management (and, as such, the performance portability thereof) is entirely user level, and could be handled by other libraries specializing in performance-portable shared-memory parallelism, such as Kokkos [24]. However, this approach also means there is little opportunity for information sharing between the distributed memory layer and the shared memory layer for purposes such as data layout, fine-granularity load balancing, or cache reuse when making scheduling decisions. As noted above, much of this information would be not be useful to the Charm++ scheduler anyway because of the lack of data model and data-flow semantics. The inability at present to express hierarchical parallelism in the execution model, even if not required in the short term, would potentially require significant updating of application code to accommodate future architectures.

The developers of Charm++ are aware of this shortcoming and are taking steps to ease the transition to hierarchical parallel platforms. The Charm++ runtime system can be built in a hybrid shared-memory/distributed-memory mode that assigns work within a multi-threaded shared-memory process on node and uses the usual mechanisms for communication between nodes. The runtime provides hooks by which messages can be passed between chares within the same shared-memory process without serialization and deserialization, but any access permissions or synchronizations on this shared memory data must be managed at the user level. Additionally, the Charm++ manual [29] includes a section about "loop-level parallelism" under experimental features (though it will no longer be listed as experimental in the next release) that introduces the *CkLoop* add-on library for Charm++, presumably aimed at being a portable Charm++ conduit to an OpenMP-like interface. Similarly, Converse [30] (the communication layer on which Charm++ is built, see Chapter 4.2.1) provides some basic thread primitives and synchronization mechanisms. However, neither of these latter two features are part of the typical Charm++ programming process, and their development and maintenance reflect this fact.

Charm++ can support some performance portability libraries better than other runtime systems due to its runtime-independent data structures. Once the C++ template issues are resolved, performance portability libraries such as RAJA [31] and Kokkos can be incorporated into an application that uses the Charm++ runtime. Charm++ applications therefore gain flexibility in the choice and interchangeability of on-node data structures and parallelism management relative to runtimes with rigorous data models like Legion. However, this flexibility comes at the cost of the runtime being unable to make intelligent scheduling decisions based on data locality, since it has no model of the data. For instance, it can not automatically schedule two tasks that need read-only access to the same data structure to run concurrently, since the core Charm++ RTS has no internal concept of either "read-only" or a "data structure."

2.2.4 Maturity

Perhaps the greatest strength of Charm++ relative to the other runtimes is its maturity. The roots of Charm++ date back to the development of the Chare Kernel Parallel Programming Language and System [32] as early as 1990. Since then, Charm++ development has proceeded with an emphasis on requirements driven development; the runtime developers have collaborators in numerous scientific application fields, including molecular dynamics [33], computational cosmology [34, 35], and quantum chemistry [36]. Most of these collaborators maintain production-level codes based on Charm++ in their respective fields. As such, the performance of Charm++ has been highly tuned and vetted, at least as it pertains to their collaborators' software.

In some ways however, the maturity of Charm++ may actually negatively affect the runtime. To put the age of Charm++ in perspective, the original Charm Kernel paper predates the first MPI standard by four years. The age of the Charm++ code base limits its adaptability and relevance with respect to many new hardware paradigms, particularly with respect to emerging manycore architectures. Nevertheless, the stability of the Charm++ runtime was almost entirely a positive contributor to the experience of porting MiniAero. The lack of more widespread adoption of Charm++ could be attributed to the relatively static and homogeneous nature of HPC machines up until now. As dynamic task parallelism and system performance become more predominant, languages like Charm++ that dynamically manage parallelism in the runtime could gain momentum relative to languages like MPI that must manage dynamic parallelism entirely at the user-level.

2.2.5 Current Research Efforts

A significant portion of the Charm++ development team's research is devoted to providing support for their application collaborators. However, the Charm++ development team is based in an active research group that has recently published new work areas such as load balancing, [37] fault tolerance, [38–40] and power management. [41] Much of this research has been or is being implemented as Charm++ experimental features.

2.2.6 MiniAero Port

The process of converting our MPI baseline implementation of MiniAero into a runtime-specific implementation was the simplest in Charm++. As a first approximation, the process started by converting `MPI_Send()` (and the like) calls to entry method invocations and `MPI_Recv()` calls into the entry methods themselves. This simple approach allowed us to stand up an initial working implementation very quickly. Because this process often breaks up local variable contexts, a direct application of this simplistic approach initially involved promoting a large number of local variables to class member variables, which our team felt led to a lot of unnecessary clutter.

As we progressed through the porting process, however, the direct one-for-one replacement of MPI calls became more untenable, forcing us to rewrite the outline of our code in a more nuanced, event-driven structure. A common pattern throughout our code for MiniAero-Charm++ involved the definition of an entry method for each unit of work (e.g., `compute_internal_face_fluxes()`) and a corresponding precondition-style entry method for use in a later `when` clause (see Section 2.2.1) postfixed with `*_done()` (e.g., `compute_internal_face_fluxes_done()`

that is invoked at the end of the work. The readability of the execution-driven control-flow specification in the ci file is clarified significantly by using naming conventions like this.

Figure 2.2 shows the ci file code for the bulk of the main RK4 inner loop. This particular section of code demonstrates several of the design conventions, patterns, and difficulties we encountered over the course of the MiniAero-Charm++ porting process, in addition to the `*_done()` convention discussed above. Beginning in line 6, the pattern used for conditional execution of a part of the program's DAG is shown. With the `*_done()` convention, the lack of execution of a given piece of work can be signalled by calling the `*_done()` version of the entry method (in the `else`) instead of the method itself. The `overlap` construct in line 19 is an example of confusion arising from the dissonance between the Charm++ programming model syntax and the Charm++ execution model — as written in this section of the code, this construct actually does nothing. Charm++'s `overlap` construct allows the *preconditions* of several sections of code to be satisfied in any order. While this is certainly true of the five `serial` code segments within this block (which is why we left the code this way), none of those segments have preconditions (`when` statements) of their own (other than the ones common to all five given in line 16 and following). Thus, these five segments do not execute out-of-order. More notably, they definitely do *not* execute in parallel ("overlap") as the name would seem to imply — recall that Charm++'s execution model only allows one entry from a given chare to execute at a given time. Similarly, the `serial` keyword led to a lot of confusion; if an entry method containing a `when` clause is called from within a `serial` block, the block may not execute in serial with subsequent `serial` blocks as expected. A concrete example of this confusion is discussed below. The `serial` keyword only means that the code in the subsequent block is C++ code.

Perhaps the most complex portion of the Charm++ port was the generalization of the communication section. The baseline MiniAero version had one function that handled all ghost cell communications. This sort of encapsulation proved a little more difficult in the MiniAero-Charm++ case because sends are expressed as remote method invocations and receives are expressed as entry methods or `when` preconditions. Figure 2.3 shows the entry method definitions in the ci file that generalize the ghost communication in MiniAero-Charm++, and Figure 2.4 shows an example usage of this pattern. Several independent types of ghost exchanges can be overlapped, since they are sent to different locations in memory. In the baseline, the destination for incoming data was passed into the communication routine via a pointer. However, in the Charm++ case, this cannot be done, since the receive invocation is done *remotely* and local pointers are not meaningful in that context. Furthermore, in Charm++, all arguments to entry methods are passed as copy-on-read (i.e., serialize-on-invocation, particularly in the case of asynchronous invocation), and passing local pointers in this fashion deviates from the runtime's fundamental data model. Thus, we used a "tag" for each type of ghost communication, and parameterized the send and receive functions to handle the proper sources and destinations for the communicated data based on the tag. The process is further complicated by the fact that one of the source/destination data structures (the gradients) has a different static type from the others. Because of this, the ghost communication most prominently exposed the shortfalls of Charm++'s poor template support. The `forall` construct in line 27 of Figure 2.3 allows the receive `when` statements for each neighbor to be satisfied in any order, similar to an `overlap`. The `forall` in the send (line 4) also allows the sends to execute out of order, but like the `overlap` construct in Figure 2.2 discussed above, actually does nothing more than simple iteration. In fact, the whole `send_ghost_data()` method should just be a regular C++ member function rather than an SDAG entry method. An SDAG method without any `when` clauses inside it is pointless (it will work, but there is no need to do it that way). Furthermore, the call to `done_sending_ghosts()` in line 15 is unnecessary and confusing; message sends happen immediately in Charm++ and do not require notification of completion. We have included this in our code examples as yet another example of how the Charm++ ci syntax led to some misconceptions on our team about how Charm++ operates. Receives are posted asynchronously, however, so the chare does not stop execution to wait for the receives in, for instance, line 6 of Figure 2.4 until the `when` construct in line 27 (note that multiple conditions on a single `when` can be satisfied in any order), which is triggered by the entry method invocation in line 37 of Figure 2.3.

Another difficulty in reading and maintaining Charm++ code is illustrated in Figure 2.4. It is not immediately clear what the control flow in this code snippet is since some of this code expresses normal C++ method calls and some expresses asynchronous entry method invocations on the local chare. In this figure, line 3 is a normal C++ method invocation, and it executes immediately as expected. However, lines 6 and 9 are entry methods, and the code in these lines expresses an asynchronous send of a message telling a chare (in this case, the local one) to *enqueue* an invocation of that entry method. Thus, the `recv_all_ghost_data()` entry method will not run until line 21, since the Charm++ execution model specifies that an entry method's context is non-preemptible (except when explicitly specified by a `when` statement). However, the next line of code, line 12, is a regular method invocation, and happens

```
entry [local] void do_stage() {

  serial "start_do_stage" {

    //Compute Gradients and Limiters, if necessary
    if(options_.second_order_space || options_.viscous){
      // entry method invocation (non-blocking)
      compute_gradients_and_limiters();
    }
    else {
      // nothing to do, so this is already done
      compute_gradients_and_limiters_done();
    }
  }

  when
    zero_cell_fluxes_done(),
    compute_gradients_and_limiters_done()
  overlap {
    serial "face_fluxes" {
      compute_internal_face_fluxes();
    }
    serial "extrapolated_bc_face_fluxes" {
      compute_extrapolated_bc_face_fluxes();
    }
    serial "tangent_bc_face_fluxes" {
      compute_tangent_bc_face_fluxes();
    }
    serial "noslip_bc_face_fluxes" {
      compute_noslip_bc_face_fluxes();
    }
    serial "inflow_bc_face_fluxes" {
      compute_inflow_bc_face_fluxes();
    }
  }

  // wait for all the bc fluxes to finish
  forall [tag] (BoundaryConditionType_MIN : BoundaryConditionType_MAX, 1) {
    when compute_bc_face_fluxes_done[tag](int tag) serial { }
  }

  // ...and wait for the internal face fluxes to finish
  when compute_internal_face_fluxes_done() serial "apply_fluxes" {

    // entry method invocation (non-blocking).
    // this calls stageFinished() when it is done, returning to the main solve()
    apply_fluxes_and_update();

  } // end serial block

}; //  end do_stage
```

Figure 2.2: Code from the ci file specification in MiniAero-Charm++ that contains much of the execution flow for a single RK4 stage in the solver. See discussion in text.

```
entry [local] void send_ghost_data(ghost_data_message_tag_type tag) {

  if(mesh_data_.send_local_ids.size() > 0) {
    forall [dest_counter] (0:mesh_data_.send_local_ids.size()-1,1) {
      serial "ghost_data_send" {
        std::map<int, std::vector<int> >::const_iterator spot =
            mesh_data_.send_local_ids.begin();
        std::advance(spot, dest_counter);
        do_ghost_send(tag, spot->first);
      }
    }
  }

  serial "finish_ghost_data_send" {
    this->thisProxy[this->thisIndex].done_sending_ghosts((int)tag);
  }

};



entry void receive_all_ghost_data(
    ghost_data_message_tag_type tag
) {

  if(mesh_data_.recv_local_ids.size() > 0) {
    forall [msg_counter] (0:mesh_data_.recv_local_ids.size()-1, 1) {
      when receive_ghost_data_message[tag](ghost_data_message_tag_type,
          int sender, int ndata, double data[ndata]
      ) serial "ghost_data_recv" {
        do_ghost_recv(tag, sender, data);
      }
    }
  }

  serial "finish_ghost_data_recv" {
    this->thisProxy[this->thisIndex].done_receiving_ghosts((int)tag);
  }

};
```

Figure 2.3: Code from the ci file specification in MiniAero-Charm++ that illustrates the communication encapsulation pattern. See discussion in text.

```
serial {
  // Compute the gradients
  green_gauss_gradient.compute_gradients(sol_temp_vec, gradients);

  // Post the receives of gradient data
  receive_all_ghost_data(gradient_tag);

  // Now share our ghosted gradient
  send_ghost_data(gradient_tag);

  // Compute the limiter mins and maxes
  stencil_limiter.compute_min_max(sol_temp_vec);

  // communicate our mins...
  receive_all_ghost_data(stencil_min_tag);
  send_ghost_data(stencil_min_tag);

  // and our maxes...
  receive_all_ghost_data(stencil_max_tag);
  send_ghost_data(stencil_max_tag);
}

// Wait for messages to be received before computing the limiters...

// Now compute limiters
when
  done_receiving_ghosts[gradient_tag](int),
  done_receiving_ghosts[stencil_min_tag](int),
  done_receiving_ghosts[stencil_max_tag](int)
serial {
  // ... compute limiters, etc...
}
```

Figure 2.4: Code from the ci file specification in MiniAero-Charm++ that shows an example usage of the communication encapsulation pattern. See discussion in text.

immediately. As discussed above, the use of the `serial` keyword in the Charm++ ci language for this context further confuses the matter.

From the code samples given in this section, it would be easy to assume that much of the code for the Charm++ port is written in the ci file pseudo-language. However, that is not the case. Most of the broad, *overview* code is written in this file, but that is a relatively small portion of the actual code. Most of the *detail* code is written in normal C++. The ci file contains only the control flow of the program from a broad perspective. The Charm++ developers strongly recommend that the physics code that makes up the actual methods invoked here (for instance, line 3 of Figure 2.4) should *not* be in the ci file, but rather should remain in ordinary C or C++, which significantly reduces the amount of code rewriting required.

2.3 Legion Programmability

2.3.1 Key Design Decisions

Legion is a data-centric programming model for writing high-performance applications for distributed heterogeneous architectures. Its highly declarative program expression is a strong shift from the procedural style of MPI and Charm++. Legion programs comprise tasks that operate on logical regions, which simply name collections of objects. When writing a task, the programmer explicitly declares the properties of the data that will be operated on by the task. This includes the data's type, organization (e.g., array of structs, struct of arrays), privileges (e.g., read-only, read-write, write-only, reduction), partitioning, and coherence. The runtime system leverages these data properties to issue data movement operations as needed, removing this burden from the developer. Task dependencies can be inferred from the data properties allowing the runtime to determine when tasks can be executed, including reordering tasks and executing them in parallel. Furthermore, the separation of the logical and physical representation of data enables the runtime to, for example, create multiple copies of read-only data to maximize parallelism as appropriate. Legion aims to decouple the specification of a program from its optimization via its mapping interface, which gives developers control over the details of data placement and task execution. While this data-driven approach is extremely powerful for extracting parallelism, the trade-off is that all inputs and outputs for a task must be declared *a priori*. In cases involving data-dependent or dynamically sparse execution, not enough information can always be expressed *a priori* for the Legion runtime to extract useful amounts of parallelism. Legion currently provides some mechanisms and has other proposed solutions for addressing these issues. The ability of Legion to handle these dynamic applications is a core part of the discussion to follow, and will be a critical issue facing the adoption of Legion for these types of applications.

The following principles have driven the design and implementation of Legion:

User control of decomposition: While Legion, in contrast to Charm++ and MPI, can automatically manage task preconditions, it is similar to Charm++ and MPI in requiring the user to specify the data decomposition into logical regions. Similarly the choice of how to decompose algorithms into tasks is also the responsibility of the application developer. MPI (other than MPI types) and Charm++ do not provide a data model, leaving data management at the application level. Legion provides a relational data model for expressing task decomposition that supports multiple views or decompositions of the same data.

Handle irregularity: Legions aims to support dynamic decision making at runtime to handle irregularities. This includes the ability to make dynamic decisions regarding how data is partitioned, where data and tasks are placed, and - although not yet realized - responses to hard and soft-errors.

Hybrid programming model: Tasks in Legion are functional with controlled side effects on logical regions as described by the task's data properties. However the tasks themselves consist of traditional imperative code. This coarse-grained functional programming model enables Legion's distributed scheduling algorithm to reason about the ordering of tasks, while still supporting the imperative coding style within tasks that is familiar to most application developers.

Deferred execution: All runtime calls in Legion are deferred, which means that they can be launched asynchronously and Legion is responsible for computing the necessary dependencies; not performing operations until it is safe to do so. This is only possible because Legion understands the structure of program data for deferring data movement operations and because Legion can reason about task's side-effects on logical regions.

Provide mechanism but not policy: Legion is designed to give programmers control over the policy of how an application is executed, while still automating any operations which can be inferred from the given policy. For example, Legion provides control over where tasks and data are run, but the runtime automatically infers the necessary copies and data movement operations to conform to the specified privilege and coherence annotations on the logical regions arguments to each task. Default implementations exists for many tools like the Mapper, accelerating the path to a debug implementation. A wide and performant set of default libraries is currently lacking, but more and better default implementations are planned and others will likely be developed as the project matures.

Decouple correctness from performance: In conjunction with the previous design principle, Legion ensures that policy decisions never impact the correctness of an application. Specifically, policy decisions about how to map applications should be limited to the mapping interface and should only impact performance, allowing applications to customize mappings to particular architectures without needing to be concerned with affecting correctness. This is useful from a performance portability perspective, as a common specification can be mapped to multiple machine types. The Legion runtime provides a default mapper, and the mapping interface is intentionally extensible to support both custom mappers, and the creation of mapping tools for building custom mappers—for it will never be possible for a default mapper to perform an optimal mapping for all applications and all machine architectures. In practice, the decoupling of performance optimization and application code has not always been limited to the mapper interface, e.g., explicit ghosting can be used to improve performance (which changes the application code).

Legion is designed for two classes of users: advanced application developers and DSL and library authors. Advanced application developers include programmers who traditionally have used combinations of MPI [42], GASNet [43], Pthreads [44], OpenCL [45], and/or Compute Unified Device Architecture (CUDA) [46] to develop their applications and always re-write applications from scratch for maximum performance on each new architecture. These programmers will find that Legion provides support for managing multiple functionally-equivalent variants of a task on different processor kinds, but it does not help with implementing the variants themselves. DSL and library authors are tool writers who develop high-level productivity languages and libraries that support separate implementations for every target architecture for maximum performance. For both user bases, Legion provides a common runtime system for implementing applications which can achieve portable performance across a range of architectures. The target classes of users also dictates that *productivity* in Legion will always be a second-class design constraint behind *performance*. Instead Legion is designed to be extensible and to support higher-level productivity languages and libraries (for example Regent [47]) or even the composition of multiple DSLs.

2.3.2 Abstractions and Controls

Figure 2.5 shows the architecture of the Legion programming system. Applications targeting Legion have the option of either being written in the compiled Legion language (Regent) or written directly to the Legion C++ runtime interface. Applications written to the compiled Legion language are translated to the C++ runtime API by their source-to-source Legion compiler. The Legion high-level runtime system implements the Legion programming model and supports all the necessary API calls for writing Legion applications. The high-level runtime system sits on top of a low-level runtime interface. The low-level interface is designed to provide portability to the entire Legion runtime system by providing primitives which can be implemented on a wide range of architectures. There are currently two implementations of the interface: a shared-memory-only version which is useful for prototyping and debugging, and a high-performance version which can run on large heterogeneous clusters. Note that the low-level interface also defines the machine object which provides the interface to the mapper for understanding the underlying architecture. Extensions to the low-level interface which support plug-and-play modules are currently in the early stages of development.

The following describes the key abstractions and controls of Legion programming model and runtime system.

Logical Regions Logical regions are the fundamental abstraction used for describing program data in Legion applications. Logical regions describe collections of data and do not imply any placement or layout in the memory hierarchy, providing some flexibility for decoupling the specification of an application from its mapping to a target architecture. Legion enforces its own relational model for data. Each logical region is described by an index space of rows (either unstructured pointers or structured 1D, 2D, or 3D arrays) and a field space of columns. Unlike other

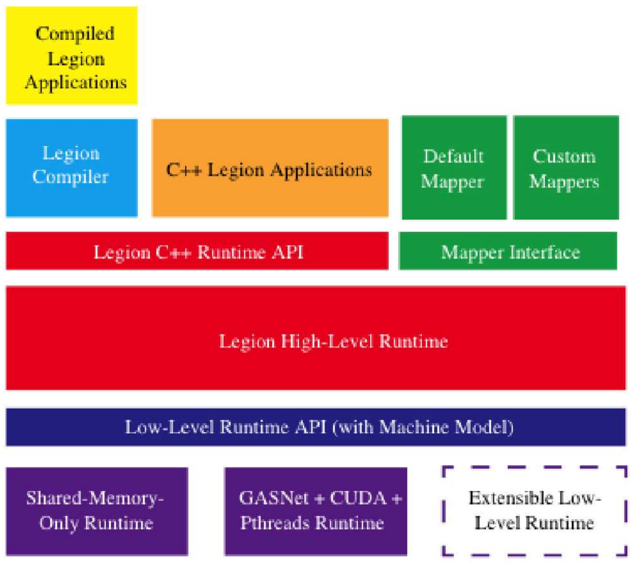

Compiled Legion Applications
Legion Compiler
C++ Legion Applications
Default Mapper
Custom Mappers
Legion C++ Runtime API
Mapper Interface
Legion High-Level Runtime
Low-Level Runtime API (with Machine Model)
Shared-Memory-Only Runtime
GASNet + CUDA + Pthreads Runtime
Extensible Low-Level Runtime

Figure 2.5: Legion Architecture

relational models, Legion logical regions can be arbitrarily partitioned into index subspaces or sliced on their field space. This relational model expresses detailed data dependencies to the Legion runtime dependency analysis, aiding detection of concurrent tasks. However, it complicates (if not precludes) incorporating custom data structures or performance portable libraries such as Kokkos [24].

Tasks A task is the fundamental unit of control in Legion, the meaning of which depends on the processor type:

- For a CPU, it is a single thread,
- For a graphics processor unit (GPU), it is a host function on the CPU with an attached CUDA context,
- For an "OpenMP" processor, it is multiple threads on a CPU,
- For an "OpenGL" processor, it is a host function on the CPU with an attached graphics context.

Multiple variants of the same task can be specified for different processors and the mapper can be configured to decide at runtime which task variant to execute. To amortize the scheduling overhead, tasks should be relatively coarse-grained with optional fine-grained parallelism within tasks. In most cases, the task should not communicate or stop for anything once it is started. All synchronization and communication happens at task boundaries.[1]

Tasks are issued in program order and every possible program execution is guaranteed to be indistinguishable from serial execution if exclusive coherence is used. If coherence is relaxed for one or more regions, data accesses to those may not be serializable, but the programmer has the ability (and responsibility) to enforce whatever semantics they require. Tasks specify their *region requirements*, which comprise the fields, privileges and coherency requirements of a task. Privileges specify the side-effects the task will have on a logical region. Coherence specifies what other tasks can do with the task's logical regions (if anything) while it is running. Existing imperative code can be wrapped inside of a task, and the dependencies between tasks described by their region requirements are used by the Legion runtime to determine execution ordering. Whenever two tasks are non-interfering, accessing either disjoint regions, different fields of the same region, or the same fields with compatible permissions (e.g., both tasks only read the field or only perform the same reduction to the field), Legion allows those tasks to run in parallel. Wherever two tasks interfere, Legion inserts the appropriate synchronization and copy operations to ensure that the data dependence is handled properly.

Legion distinguishes a *Single* task which is similar to a single function call, and an *Index Space* task which is similar to a potentially nested `for` loop around a function call with the restriction that each invocation be independent. If the Index Space is explicitly declared as disjoint, the runtime can reduce the associated dynamic runtime analysis cost compared to the alternative of expressing each index space task as a single task. Legion tasks are permitted to return

[1] With certain exceptions in advanced cases.

values. When a task call is performed in Legion it does not block. This is an essential component of the Legion deferred execution model. The asynchronous nature of tasks in Legion necessitates that a place-holder object be used to represent the return values from a task—this value is a *future*. In the case of index space task launches, a *future map* can be returned which contains a single future entry for each point in the index space of tasks launched—or the map can even be reduced to a single value.

Tree of tasks Every Legion program executes as a *tree of tasks* with a top-level task spawning sub-tasks which can recursively spawn further sub-tasks. A root task is initially executed on some processor in the machine and this task can launch an arbitrary number of sub-tasks. Each sub-task can in turn launch its own sub-tasks. There is no limit to the depth of the task tree in a Legion program execution. A sub-task can only access regions (or sub-regions) that its parent task could access; furthermore, the sub-task can only have permissions on a region compatible with the parent's permissions. As mentioned above, tasks must *a priori* declare all data they will operate on. Because of the tree structure, parent tasks must actually *a priori* declare any data that their children will operate on.

While the tree of tasks approach is different from the traditional SPMD programming model for targeting supercomputers, Legion provides some mechanisms for replicating the SPMD programming model. In many cases (e.g., stencil codes) tasks operate on overlapping data, creating region conflicts that would prevent them running concurrently. SPMD can be implemented at the application level by explicit ghosting, in which extra ghost entries are included in a logical region. Before the task runs, region-to-region copy operations are issued to update the ghost data. Through explicit ghosting, a disjoint index space is created which can be mapped in an SPMD fashion. Region requirements can also be declared in relaxed coherence modes (e.g., SIMULTANEOUS) that instructs the scheduler to ignore data conflicts. The application must then explicitly manage synchronization between overlapping (conflicting) regions.

This restriction of *a priori* data and privilege declarations poses a challenge to highly dynamic engineering codes, such as electromagnetic particle in cell. Potentials solutions include inline mappings of particle buffers with relaxed coherence modes for directly writing (migrating) particles between task particle buffers, but difficulties may remain in achieving efficient quiescence detection and limiting inter-task synchronization. Legion also allows dynamically allocating *new* logical regions within a child task and returning it to the parent, which better supports the dynamic particle creation and migration in PIC codes. The performance bottleneck currently is that a child task cannot pass new logical regions to sibling tasks, requiring new tasks to be created and scheduled (serially) through the parent. Solutions have been proposed for delegating and distributing task creation to child tasks to avoid the serial bottleneck. In this case code is semantically equivalent to all tasks being scheduled by the parent, but, given certain safety guarantees, tasks can directly schedule sibling tasks. The exact implementation has not been finalized, but the potential performance of such a solution merits further exploration.

Mapping Interface and Machine Model The Mapper interface in Legion is the primary mechanism by which Legion applications can directly control how they are mapped onto target hardware. Mapping is the process of selecting a processor to run each task and a memory (and data layout) for each logical region. Mappers are special C++ objects that are built on top of the Legion mapping interface which is queried by the high-level runtime system to make all mapping decisions when executing a Legion program. Applications can be developed with the default Legion mapper, but will usually require an application-specific mapper for higher performance. Mapper objects have access to a singleton object called the machine object. The machine object is an interface for introspecting the underlying hardware on which the application is executing. Currently, the machine object is static and does not change during the duration of an application, but in the future it may dynamically be modified to reflect the changing state of hardware.

Runtime Execution Model The Legion execution model is *Implicit Parallelism with Explicit Serial Semantics.* The code executed by the task is explicitly serial, but one or more tasks can be executing simultaneously under control of the runtime. Task execution in Legion is pipelined. In general, a task must complete a pipeline stage before it passes to the next stage. If a given stage stalls for any reason, that task and any task that depends on it also stalls. All runtime calls in Legion are deferred which means that they can be launched asynchronously and Legion is responsible for computing the necessary dependencies and not performing operations until it is safe to do so. The essence of a deferred execution model is the promise that all operations are asynchronous and it is the responsibility of the Legion implementation to maintain the sequential program order semantics of an application. By guaranteeing that Legion is fully asynchronous, Legion applications can launch many outstanding sub-tasks and other operations, allowing a Legion implementation to automatically discover as much parallelism as possible and hide communication latency.

Load Balancing The Mapper interface is where load balancing and work stealing are implemented. Legion supports several different features for allowing mappers to customize load balance at runtime. These features can be categorized into two areas: support for load balancing *within* a node and load balancing *between* nodes.

To support load balancing *within* a node, Legion permits mappers to create *processor groups*. A processor group is effectively a name for a task queue that is shared between a set of processors. Tasks that are mapped to a processor group are placed in the task queue for the processor group. As soon as any processor in the processor group is available for executing a task, it will pull a task off the queue and execute it. Using processor groups, mappers can easily load balance task execution across a set of processors.

There are two possible mechanisms for performing load balancing between nodes: one based on a pull methodology and one based on a push methodology. Task stealing can be used to *pull* work between nodes. To support task stealing, as part of every scheduler invocation, the Legion runtime invokes the target task `steal` mapper call, which queries each mapper to see if it would like to target any other processors in the machine for task stealing. The mapper is free to target any subset of processors (or none at all). If steal requests are made, the runtime sends the necessary messages to the same kind of mappers on the remote node. When these requests arrive, they trigger an invocation of the `permit` task steal mapper call.

In Legion, tasks are not stolen automatically. Mappers that own tasks must explicitly permit them to be stolen. The reason for this is that most other task stealing mechanisms operate in shared memory environments, and there is minimal data movement as a result of stealing. In Legion, however, stealing primarily occurs *between* nodes, and the cost of moving data is much higher. Legion therefore gives the owning mapper the prerogative to reject steal requests. If a request is approved, Legion can also pick the tasks to be stolen based on region affinities.

The receiving node of a steal request is only permitted to allow tasks currently in its ready-queue to be stolen. Tasks that have been mapped onto a processor are not eligible for stealing. Since the default mapper has no information about the structure of an application, stealing is not enabled in normal conditions but can be enabled with a command line flag. When enabled, the default mappers randomly choose a processor from which to attempt to steal; this avoids stampedes where all mappers attempt to steal from the same processor at the same time.

The other Legion approach to performing between-node load balancing is to implement a push-based execution model where mappers coordinate to balance load. The mapper interface provides a mechanism for mappers to send messages to other mappers of the same kind on other processors. A message consists of a pointer to an untyped buffer and a size of the number of bytes to copy. The runtime makes a copy of this buffer and transmits it to the target node. On the target node the runtime invokes the message handler mapper call handle message. Mappers are permitted to send messages from inside of any mapper call including the message handler mapper call.

Using messages, mappers of the same kind can orchestrate dynamic load balancing patterns that can be re-used for long epochs of application execution. For example, in an adaptive mesh refinement code, a custom mapper implementation could have mappers communicate the load of tasks that they receive after each refinement. Based on load information for different processors, each mapper can independently compute a load balancing scheme and determine where to send all the tasks for which it was initially responsible. The mappers can memoize this result and re-use it until a new refinement occurs or an old refinement is deleted. The granularity at which load balancing schemes are computed will vary with the size of the machine and the amount of work being generated, but these kinds of performance considerations are explicitly left to the mapper by design.

While knowledge about how tasks and other operations are mapped is a necessary condition for understanding the performance of an application, it is not sufficient. Mappers also need access to profiling information to understand the performance implications of mapping results. Mappers can ask for profiling options including the execution time of a task, total execution time of a task and its children, and hardware instruction and memory counters such as loads and stores issued and cache hits/misses. By coupling these data with the mapping decision results reported by the mapper calls, mappers can infer the performance effects that different decisions might have on performance. Profiling information closes the loop in the mapping process, giving mappers the ability to drive future mapping decisions based on the performance of previous mapping results.

While performance is one metric by which mappers might choose to make mapping decisions, another is resource utilization. For example, a mapper might opt to omit a specific memory from a ranking for a region requirement in the map task mapping call because the memory is nearly full, and the mapper knows that the space needs to be reserved for a later task. Discerning such information requires mappers to query state information about different kinds of resources in the machine. To make this possible, as part of the mapper interface the runtime provides calls for mappers to inquire about the state of different resources.

Another area where the mapper can receive feedback about the state of an application is through the dynamic data flow graph. Legion makes available the input and output dependencies for all operations, allowing the mapper to explore the dynamic dataflow graph. The runtime also provides a mechanism for mappers to query which operations have already been mapped and which ones are still un-mapped. By combining access to the shape of the graph along with profiling information, mappers can infer critical paths that are important for assigning priorities. Furthermore, by monitoring the location of the wavefront of mapped tasks within the graph, mappers can determine how far ahead of actual application execution mapping is occurring, thereby giving the mapper the feedback necessary to accurately manage deferred execution.

Legion has been designed with all of the hooks in place for the design and development of powerful and performant application and architecture-specific load balancing mappers. As the runtime system matures, the Legion developers will likely provide a collection of default load-balancing mappers that can be specialized to suit a specific application's needs for a given architecture. However, currently mapping, and therefore load balancing is the responsibility of individual application developers.

2.3.3 A Note About SPMD Applications in Legion

A Legion application is expressed as a hierarchy of tasks, ideally using a mixture of data and task parallelism. An initial *top-level task* is executed at runtime startup and this task launches subtasks, which may be distributed around the machine and launch further subtasks. The Legion privilege system is carefully designed to allow dependency analysis performed within sibling tasks (i.e., for their subtasks) to be done in parallel, but the "apparently sequential" semantics provided by Legion requires that the dependency analysis between all sibling child tasks be performed within the parent task's context. If a task has many children, especially short-running ones, the runtime overhead can become a performance bottleneck. If the task in question is the top-level task, this bottleneck becomes a scalability problem.

Handling this very common application pattern efficiently is required of any runtime to meet ASC/ATDM application requirements. A "bulk synchronous" application (i.e., one in which each task is a "for all elements ..."), if ported to Legion in the intuitive way, will have the worst possible task hierarchy, with the top-level task being the immediate parent of every other task in the application. For some applications, there might be a comparable hierarchical (e.g., divide and conquer) formulation possible (perhaps with the bulk synchronous version having been chosen as a better fit for MPI), and one could argue that the Legion version should be rewritten in the hierarchical way. However, there are other applications for which there is no reasonable alternative to the bulk synchronous expression. It is essential that Legion provide a way to achieve scalable performance for these applications, ideally while also maintaining the productivity and portability advantages offered by the Legion programming model.

The underlying mechanism for this is a transformation that will be familiar—replace a single task that has too much work with N copies of the task, each doing $1/N$ of the work. This is very similar to MPI's SPMD execution model, but there are several important differences. First, while the execution model may look similar to MPI, the data model is based on logical regions and tasks with privileges rather than the explicit message passing of MPI. Legion provides *relaxed coherence* modes that allow two tasks with conflicting privileges on the same logical region to run concurrently, and *phase barriers* to allow those tasks to establish the necessary inter-task data dependencies that allow the Legion runtime to correctly build task graphs that can be executed in the usual asynchronous fashion.

Another major difference is that the SPMD-ification of a Legion task does not impact the application mapper's freedom to make placement decisions for subtasks and/or instances of logical regions used by those subtasks. A child task launched by an SPMD copy of a parent task on a given node may be executed on a different node, permitting the same load-balancing decisions as would be possible without the SPMD transformation.

For applications that require it, Legion developers are pursuing possible ways to SPMD-ify a bottleneck task (or tasks).

The first is a manual programmer effort, which has been shown to yield excellent scalability for a reasonable amount of programmer effort. There is also a prototype for an automatic SPMD transformation pass in Regent, and it is hoped that a similar transformation can be implemented in the Legion runtime for applications using the C++ API. Regardless of the transformation technique used, the programmer is encouraged to write a “naïve” (i.e., non-SPMD) implementation first.

Manual SPMD-ification The process of manually applying a SPMD transformation to a Legion task is, unsurprisingly, isomorphic to how one would port a single-node implementation of a task to MPI. First, the programmer must decide how the application’s data will be decomposed. As discussed above, this decomposition is purely logical in the Legion case (the mapper will still control in which memory data is actually allocated), a clear notion of “ownership” of the data is just as important for Legion SPMD tasks as it is for an MPI implementation. Second, the programmer must understand where in the code data will cross these decomposition boundaries. These become phase barrier operations for Legion and `MPI_send/recv` calls for MPI. The Legion implementation requires some additional effort to set up the necessary phase barriers ahead of time, but also avoids considerable headaches by not having to find computation to “overlap” with the communication in order to hide communication latency. A third step that is necessary in the MPI case, but not in the Legion case, is the conversion between “global” and “local” coordinates or indices when data is transferred. The Legion data model allows all tasks to use the same coordinate or index for a given piece of data. A performance analysis for the manual SPMD transformation of S3D is discussed in Figure 3.7.

Automatic SPMD-ification Although the manual SPMD-ification of a Legion application’s top-level task is clearly effective, it is not without drawbacks. The most obvious is the programmer time required to implement the transformation and maintain it if the required decomposition and/or the application’s communication pattern changes. The second is that the resulting code deviates from the “Legion style” of coding in that the machine-agnostic “functional description” of the application becomes entangled with code that is specific to how it is being mapped to a given machine (or class of machines). Ideally, the programmer would write the machine-agnostic version and SPMD-related changes could be applied by a compiler or runtime transformation. Similar transformations have been explored for MPI in the past with very limited success - due primarily to the challenge of identifying those places in which data crosses the decomposition boundaries. However, identification of the communication pattern in the corresponding Legion application is made possible by the Legion data model. The compiler’s (or runtime’s) analysis of the data dependencies between tasks within a decomposition boundary will naturally discover the dependencies that cross boundaries, identifying precisely the tasks for which phase barriers operations must be inserted. Initial performance results are explored with Regent in Figure 3.7b.

2.3.4 Performance Portability

The Legion design enables performance portability through the mapper interface. An individual task is executed serially, and therefore can be implemented without knowing or caring about parallel execution issues. Legion guarantees that mapping decisions only impact performance and are orthogonal to correctness which simplifies tuning of Legion applications and enables easy porting to different architectures. The separation of concerns enables Legion to perform the important task of managing concurrency and data movement. However, the burden of achieving high-performance is still fundamentally at the application-level through the mapper. Since mapping decisions do not affect application correctness, it may be possible for programmers to separately tune mappers for different code regions without introducing bugs.

The Mapper interface is highly flexible; however, it can require knowledge of the low-level details of the Legion runtime system and the low-level details of the machine architecture(s) on which the application will run to fully reason about the optimal mapping decisions. Note that this is not a criticism of the approach; it is very helpful to have a single, documented and accessible API for making these decisions which are required for all task-based runtime systems.

2.3.5 Maturity

Legion is a relative newcomer in the task-programming runtime systems. The project began in 2011 and the first version was released in 2012, although many of the concepts grew out earlier work on the Sequoia [20] language. It has been used on a few large projects including S3D [48] on which Legion demonstrated very good scalability results. Development continues on improving the quality and capability of the runtime system and higher-level Domain-Specific Languages which can hide the low-level Legion interface and make it easier for application developers to use.

The emphasis in Legion is on providing a common runtime system which can achieve portable performance across a range of architectures; developer productivity in Legion is a second-class design constraint. However, developer productivity is being addressed via the design of higher-level languages such as Regent and Terra which accept a higher-level syntax and output the low-level Legion runtime code. The immaturity of Legion is both a benefit and a cost. The benefits include being able to influence the design and implementation of the language; the costs include dealing with sometimes major API changes and missing functionality.

2.3.6 Current Research Efforts

The Legion programming model is very much a work in progress. All parts of the Legion programming model are under active development. Some of the efforts are focused on ways to improve performance, but many are also providing new features or capabilities to the programming model. Several of these "coming soon" features are described below. (Each is being worked on by different developers, so the order in which they are listed implies neither prioritization nor expected delivery order.)

Regent Regent [47] is a higher-level, but still general-purpose, language that is compiled into C++ that uses the Legion API. In addition to improving productivity by automatically generating the often-verbose Legion API calls (see Figures 2.6 and 2.7 for an example), the Regent compiler implements the Legion type system [49]. This provides two major benefits. First, Regent is able to detect (at compile-time) application bugs in which data is accessed outside the specified logical regions or in the wrong way (e.g., writing instead of reading). In an asynchronous runtime, such application bugs generally result non-deterministic data corruption, which can be very difficult to debug at run time. Second, the Regent compiler is able to use the Legion type information to perform several valuable optimizations, exposing additional parallelism and reducing Legion runtime overhead.

We note that some of the optimizations that were applied to the Legion implementation of MiniAero were developed and prototyped initially in Regent. The current MiniAero mapper is the same as the Mapper used in the Regent MiniAero implementation. Regent is not C++, so porting an existing C++ application to use Regent would involve translating from C++ to Regent; however, Regent applications can interact with C++ routines, so a complete port of an application into Regent is not required.

Mapping Language Regent provides a more productive path for writing the functional description of the application, but a custom mapper for a Regent application is still written using the Legion C++ API. Another effort is looking at techniques similar to Regent to implement a higher-level "mapping language" companion. Again, the most obvious benefit is an improvement in productivity - it is expected that mappers for many existing Legion applications could be written in 10's of lines of higher-level code. However, it is expected that this mapping language will provide new capabilities as well, such as the ability to automatically generate "variants" of a task for different processor types or memory layouts, or use Regent's static knowledge of the shape of the region tree to simplify load-balancing decisions in the mapper.

Dependent Partitioning Legion's current partitioning API has the benefit of being maximally expressive (i.e., the programmer can describe ANY subset of elements), but achieves this by treating *colorings* as opaque objects that must be generated entirely by application code and then "thrown over the wall" to the Legion runtime. This represents a productivity loss for many common partitioning patterns but more critically, prevents the computation of a partitioning from benefiting from distributed and deferred execution like normal Legion tasks.

```
IndexSpace is_point = runtime->create_index_space(ctx, conf.np);
FieldSpace fs_point = runtime->create_field_space(ctx);
FieldAllocator fa = runtime->create_field_allocator(ctx, fs_point);
fa->allocate_field(sizeof(double), PX0_X);
fa->allocate_field(sizeof(double), PX0_Y);
fa->allocate_field(sizeof(double), PX_X);
fa->allocate_field(sizeof(double), PX_Y);
fa->allocate_field(sizeof(double), PU0_X);
fa->allocate_field(sizeof(double), PU0_Y);
fa->allocate_field(sizeof(double), PU_X);
fa->allocate_field(sizeof(double), PU_Y);
fa->allocate_field(sizeof(double), PF_X);
fa->allocate_field(sizeof(double), PF_Y);
fa->allocate_field(sizeof(double), PMASWT);
LogicalRegion points = runtime->create_logical_region(ctx, is_point, fs_point);
... (additional code not shown here)
runtime->unmap_region(ctx, pr_points_all_private);
Domain domain = Domain::from_rect<1>(
  Rect<1>(Point<1>(0), Point<1>(conf.npieces - 1)));
IndexLauncher launcher(ADV_POS_FULL, domain,
                       TaskArgument(), ArgumentMap());
launcher.add_region_requirement(
  RegionRequirement(points_all_private_p, 0 /* projection */,
                    READ_ONLY, EXCLUSIVE, points_all_private));
launcher.add_field(0, PX0_X);
launcher.add_field(0, PX0_Y);
launcher.add_field(0, PU0_X);
launcher.add_field(0, PU0_Y);
launcher.add_field(0, PF_X);
launcher.add_field(0, PF_Y);
launcher.add_field(0, PMASWT);
launcher.add_region_requirement(
  RegionRequirement(points_all_private_p, 0 /* projection */,
                    READ_WRITE, EXCLUSIVE, points_all_private));
launcher.add_field(1, PX_X);
launcher.add_field(1, PX_Y);
launcher.add_field(1, PU_X);
launcher.add_field(1, PU_Y);
launcher.add_future(dt);
runtime->execute_index_space(ctx, launcher);
```

Figure 2.6: Example Code from PENNANT — C++ Implementation

```
fspace point {
  px0 : vec2,          -- point coordinates, start of cycle
  px :  vec2,          -- point coordinates, end of cycle
  pu0 : vec2,          -- point velocity, start of cycle
  pu :  vec2,          -- point velocity, end of cycle
  pf :  vec2,          -- point force
  pmaswt : double,     -- point mass
}
var rp_all = region(ispace(ptr, conf.np), point)
... (additional code not shown here)
for i = 0, conf.npieces do
  adv_pos_full(points_all_private_p[i], dt)
end
```

Figure 2.7: Example Code from PENNANT — Regent Implementation

The newer *dependent partitioning* API [50] eliminates colorings entirely, and instead allows computation of partitions based on other partitions and/or data stored in logical regions (which may be distributed across the memory hierarchy). These computations are performed as operations in Realm's task graph, allowing them to automatically overlap with other tasks. For example, you could compute a new partition for improving load balancing while still making simulation progress with the previous partition.

Storage Support Legion already supports the use of "disk storage" (i.e., anything exposed as a filesystem) as a type of memory in which instances of logical regions can be placed, but the lifetime of any data stored to disk in this manner ends with the application process and the data format is Realm-specific. In order to read and write "persistent" data from disk (i.e., data that exists before and/or after the Legion application is executed), the application must explicitly perform disk I/O in an application task, mapping the data from the format on disk into the Legion data model. The inefficiencies that result from this approach are being addressed by adding a notion of *external resources* to Legion. An external resource can be *attached* to a logical region (similar to how one can `mmap` a file into the virtual address space of a POSIX process). This makes the connection to the Legion data model explicit and allows the Legion runtime (under the direction of the application mapper, as always) to manage the actual data movement and any necessary layout transformations. Furthermore, by bringing persistent data into the Legion programming model, an application mapper can direct Legion to perform some of the computation on newly available execution resources, such as the embedded processors on storage systems.

Processor and Interconnect Support Realm's machine model abstraction allows the Legion runtime to be completely agnostic to the number and type of processors in a node or the kind of high-speed interconnect used to connect the nodes. However, for performance reasons, the Realm implementation generally wants to be aware of the details of the exact architecture and therefore needs to be updated for new processor and/or network architectures[2]. In particular, work is planned to add support for Knights Landing-based systems as soon as those are made available.

General Polishing and Productization Thus far, the primary users of the Legion programming model have been the researchers themselves. In order to make the programming model useful and stable for application developers, some effort has been put into documentation, tutorials, regression testing, and bug fixing, but this is area in which additional staffing is planned. It is expected that such staffing will be more of the *engineering* than *researcher* variety, but the research team will need to devote some time to this as well.

2.3.7 MiniAero Port

In the Legion implementation of MiniAero, we converted the existing Kokkos functors into tasks. Each functor was examined to determine its data requirements which were translated into *Region Requirements*. The core *physics* code of the functor remained virtually unchanged; the data access code was changed to use the Legion Accessor methods. The specification of `RegionRequirements` for each task coupled with the related accessing of the field data on the Regions inside the task implementations results in several additional lines of code that may not be required in other programming models. Much of the code was common boilerplate so the verbosity might be hidden or abstracted away via wrapper classes or higher-level languages. This is a known issue and Legion documentation continually emphasizes that performance is more important than productivity and that productivity issues should be solved by independent libraries or programming environment tools, such as Regent [47].

Figure 2.8 shows an example of the code used to specify the launching of a task including the region requirement specifications. The corresponding task implementation for this task launch is shown in Figure 2.9. The main physics of the task are shown on lines 39 to 60 and this portion looks very similar, or even the same, as the corresponding code in the original MiniAero implementation. The code preceding those lines is setting up the data accessors.

Overall, approximately 50% of the lines of code for the Legion MiniAero implementation of the *timestep code* (that code not dealing with problem setup and mesh generation) were *boiler-plate* code dealing with region requirements,

[2]Realm's use of GASNet simplifies adding support for a new network architecture (assuming GASNet supports it), but does not eliminate the need for performance tuning.

```
IndexLauncher launcher(MiniAero::boundary_face_gradient_TID,
                        meshdata.domain, TaskArgument(), local_args);

RegionRequirement cw0(meshdata.cell_logical_partition,
                      identity_mapping, READ_WRITE, EXCLUSIVE,
                      meshdata.cell_logical_region);
cw0.add_field(gradient);
launcher.add_region_requirement(cw0);

RegionRequirement cr(meshdata.cell_logical_partition,
                     identity_mapping, READ_ONLY, EXCLUSIVE,
                     meshdata.cell_logical_region);
cr.add_field(solution_var);
cr.add_field(MeshData::volume);
launcher.add_region_requirement(cr);

RegionRequirement rof(meshdata.face_logical_partition,
                      i, READ_ONLY, EXCLUSIVE,
                      meshdata.face_logical_region);
rof.add_field(MeshData::face_cell_conn);
rof.add_field(MeshData::a_vec);
launcher.add_region_requirement(rof);

runtime->execute_index_space(ctx, launcher);
```

Figure 2.8: Legion Task Launch Initialization

field accessors, and other non-physics-related setup. This overhead can be alleviated or even completely eliminated through the use of library abstractions or through a higher-level language that generates the low-level Legion code. As an example the latter, the Regent implementation of MiniAero used 30% fewer lines of code than the Legion implementation[3]. The boiler plate code is fairly separable from the physics code except for the case of the Field Accessors in the task implementation code where both types of code exist in the same function; although even in this case, the physics code is easily identifiable.

2.4 Uintah Programmability

2.4.1 Key Design Decisions

The Uintah runtime system facilitates the numerical solution of partial differential equations (PDEs) on structured meshes with AMR capability. Uintah is designed with a focus on insulating the application developer from architectures: Application code should run "unchanged" from the view of the application developer from 600 to 600K cores. Application developers describe their algorithm as a task graph in C++. While Uintah provides options for common linear solvers—conjugate gradient (CG) implemented in Uintah or an interface to the solvers from *hypre* [51] and *petsc* [52] libraries—the explicit solver for MiniAero was implemented as a collection of tasks with dependencies. The runtime supports out of order execution of tasks, work stealing, and overlap of communication and computation, and execution of tasks on cores or accelerators. The central design philosophy of Uintah can best be described as "achieving parallel slackness through multiplicity of tasks and out of order execution". The runtime focuses on maximizing parallel slackness and overlapping communication and computation by having a sufficiently rich mix of tasks on a per node basis which allows the scheduler to keep all the resources on a heterogeneous node sufficiently busy. This is achieved by having multiple task graphs executed on each node by the node-level scheduler which handles off-node communication transparently in a manner that avoids blocking or waiting for messages.

[3]This includes counting the mesh generation code which is approximately half of the lines of code. Eliminating this from the comparison should make the Regent implementation of the time-step loop code have almost the same lines of code as the baseline MiniAero implementation.

```
static void boundary_face_gradient_task(const Task* task,
                                        const std::vector<PhysicalRegion> &region,
                                        Context ctx, HighLevelRuntime *runtime)
{
  int cell_region = 1;
  int face_region = 2;

  typedef Accessor::RegionAccessor<AT, FaceCellConnect> FaceCellConnAcc;
  typedef Accessor::RegionAccessor<AT, SolutionFieldType> SolutionAcc;

  FieldID soln_fid = task->regions[cell_region].instance_fields[0];
  SolutionAcc solution =
     get_accessor<AT,SolutionFieldType>(region[cell_region], soln_fid);

  typedef Accessor::RegionAccessor<AT, double> RealAccessor;
  RealAccessor volume =
     get_accessor<AT,double>(region[cell_region], MeshData::volume);

  typedef Accessor::RegionAccessor<AT, GradientFieldType> GradientAcc;
  GradientAcc cell_gradient =
           get_accessor<AT,GradientFieldType>(region[0], MeshData::cell_gradients);

  typedef Accessor::RegionAccessor<AT, Vector3D> Vector3DAcc;
  Vector3DAcc a_vec =
     get_accessor<AT,Vector3D>(region[face_region], MeshData::a_vec);

  FaceCellConnAcc face_cell_conn =
     get_accessor<AT,FaceCellConnect>(region[face_region], MeshData::face_cell_conn);

  IndexSpace face_subspace =
     region[face_region].get_logical_region().get_index_space();
  CachedIndexIterator it(runtime, ctx, face_subspace, true);

  while (it.has_next()){
    size_t count = 0;
    ptr_t start = it.next_span(count);
    for (size_t i= start.value; i < start.value+count; i++) {
      ptr_t faceptr(i);
      FaceCellConnect conn = face_cell_conn.read(faceptr);

      SolutionFieldType conservatives = solution.read(conn.left);
      SolutionFieldType primitives;
      ComputePrimitives(conservatives, primitives);

      GradientFieldType gradient;
      double cell_volume = volume.read(conn.left);
      Vector3D face_normal = a_vec.read(faceptr);
      face_normal[0] = -face_normal[0];
      face_normal[1] = -face_normal[1];
      face_normal[2] = -face_normal[2];

      for(int icomp = 0; icomp < 5; ++icomp) {
        for(int idir = 0; idir < 3; ++idir) {
          gradient(icomp,idir) = primitives[icomp]*face_normal[idir] / cell_volume;
         }
       }

    GradientFieldType grad = cell_gradient.read(conn.left);
    grad += gradient;
    cell_gradient.write(conn.left, grad);
    }
  }
}
```

Figure 2.9: Legion Task Execution

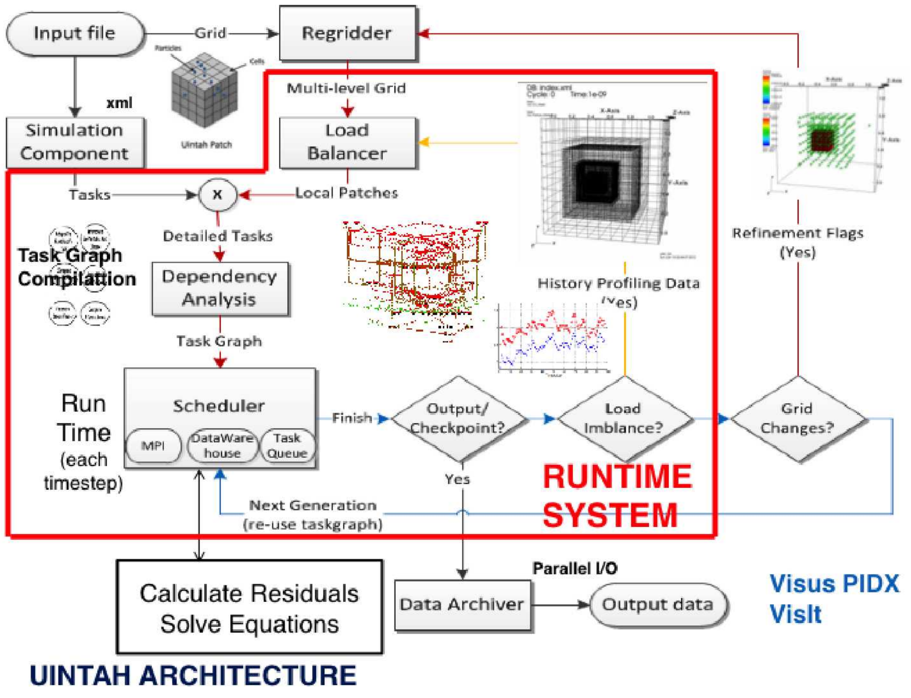


Figure 2.10: A schematic overview of the Uintah software architecture, as available at [4].

2.4.2 Abstractions and Controls

Patch-based Structured Domain Decomposition The first level of parallelism in Uintah is data parallelism using a "patch" based domain decomposition. User written code is required to express a set of tasks operating on a patch of cells. By having multiple patches on each node, the per-patch task graph is replicated multiple times on each node resulting in the parallel slackness. The patch based domain decomposition restricts Uintah to problems with structured meshes only. This might seem a very stringent restriction for certain applications, but for applications with structured meshes it offers significant advantages since no coding effort is required, nor significant computational time spent, to set up the mesh. In other words the mesh and its decomposition is specified entirely implicitly through the input file and runtime parameters that specify the size of the domain, the global number of cells and the number of patches to decompose into. Furthermore, no user code is required to determine patch connectivity as the runtime infers this implicitly and manages it for the user transparently. However, the structured mesh restriction does not necessarily translate to a restriction of keeping the geometry simple but rather Uintah provides the capability to handle complex geometries which can be constructed using operations (union, intersection and difference) on a base set of simple solid geometry objects (boxes, cylinders, spheres and cones). Nonetheless, the structure mesh restriction is a major limitation for ASC workloads. For the purposes of comparison in this study, we chose to do a structured mesh implementation of MiniAero in Uintah, and we took care to not include the cost of mesh generation/setup while comparing the performance across the runtimes.

Tasks In Uintah a task is effectively a user-written C++ method that specifies a computational kernel. Tasks are required to be written like serial code operating on a generic patch without any explicit MPI calls or threading directives. This ensures the separation of user code from the underlying parallelism. The Uintah task scheduler compiles all of the tasks based on specified variable dependencies into a task-graph with edges in the graph denoting dependencies. Internal dependencies i.e., dependencies between patches on the same processor "imply a necessary order" while external dependencies i.e., dependencies between patches on different processors "specify required communication". The task scheduler also identifies and combines external dependencies from the same source, or to the same destination, and coalesces messages accordingly. However, the granularity of a task is restricted to that specified in the `scheduleTimeAdvance` method. In other words tasks specified in this method cannot create further sub-tasks. The body of the task itself contains a pointer to a function that implements the actual computation along with the data requirements and outputs from this function.

Data warehouse In Uintah each task specifies the inputs it requires and outputs it generates. The tasks do not explicitly define communications but rather only specify what inputs they need. The creation, storage and retrieval of the actual data is performed through transactions with an abstract layer called the data warehouse which is specific to each patch. At all times two versions of the data warehouse are made available to the user specified tasks: an "old" data warehouse which is the repository for data from the previous time step and a "new" data warehouse which is for the current time step. The user tasks (for each patch) specify which variables they need as input and what they will generate as output and whether it is from/to the new or the old data warehouses. The only restriction placed is that outputs cannot be written to the old data warehouse. At the end of each time step the current old data warehouse is over written by the current new data warehouse and an empty new data warehouse is created, which is effectively the garbage collection mechanism. Based purely on the expressed input/output dependencies Uintah infers the task graph and exposes any inherent task parallelism for the scheduler to exploit. Tasks are also allowed to request ghost cells of arbitrary thickness. Uintah automatically translates the ghost cell requirements to dependencies from neighboring patches and schedules the necessary MPI communication completely transparently. In our implementation of MiniAero, care was required in specifying the right requirements of variables from the correct versions of the data warehouses since the task graph is implied fully by the specified dependencies. This is typical when one goes from a bulk synchronous programming model to an asynchronous one since the latter forces a programmer to reason about data dependencies, in this case across tasks and time steps, very carefully.

Schedulers Considerable effort was spent over the last two decades on refining the node-level task schedulers in Uintah [53]. In its earliest form, a static MPI based scheduler was employed which places one MPI process on each core. This approach represents only data parallelism, with the task graphs being identical between the cores and the order of execution static and deterministic. Even cores on the same node having access to the same shared memory communicate through MPI messages, which results in a lot of unnecessary memory usage, in addition to the execution model being bulk synchronous. This was improved in the subsequent dynamic MPI scheduler, which works with a two-stage task queue execution model. Each CPU has an internal ready and an external ready task queue. Tasks whose required inputs from local tasks have become available are placed in the internal ready queue; while tasks whose inputs from external tasks have become available (when the corresponding messages have been received) are placed on the external ready queue ready for execution. This allows dynamic out of order execution and greatly reduces MPI wait times. Nonetheless, the disadvantages of accessing data from shared memory only through MPI messages, and resulting extra memory storage, remain.

In the next improvement, optimal access to shared memory on node was adopted in a "threaded" MPI scheduler. In this execution model one MPI process is placed on each node and, using Pthreads, multiple worker threads are created across the shared memory cores, all orchestrated by a single control thread. All threads on node have shared access to the same data warehouse and task queues which were redesigned to ensure thread safety. The control thread processes MPI receives, manages task queues and assigns ready tasks to worker threads, while the MPI sends are posted by the worker threads directly. This is currently the default scheduler in Uintah but it requires `MPI_THREAD_MULTIPLE` support. The threaded MPI scheduler has also been adapted for hybrid CPU-GPU architectures. In this setting the scheduler handles the data copies between the CPU and GPU transparently by maintaining two extra task queues (in addition to the internal and external ready queues for the CPU): one for initially ready GPU tasks and the second for tasks whose host-device or device-host data copies are pending. The control of all task queues is still handled by the solitary control thread. Currently development efforts are focused on a "unified" scheduler in which the single control thread is dispensed with and all worker threads are independent and have shared access to the data warehouse and task queues, implemented using novel lock-free data structures. Each thread assigns work to itself by querying the shared external ready queue and processes its own MPI sends and receives. This execution model is expected to avoid situations where the dedicated single control thread in the threaded scheduler is under-utilized. GPU support for the "unified" scheduler is also currently under active development.

In essence the current default scheduler as well as the schedulers actively being developed subscribe to the "MPI+X" paradigm. Our experiments with the threaded schedulers exposed certain implementation aspects that gave unexpected results. In particular, when spawning multiple MPI processes on a single node (as might be desirable if there are multiple non-uniform memory access (NUMA) domains on a node and one MPI process per NUMA domain) the speedups were below expected. This turned out to be due to an implementation quirk as will be elaborated in section 3.2.

Component-based architecture The design of Uintah software is component-based where each component encapsulates an aspect of the application code that can be developed independent of the others. Even the scheduler is viewed as one component whose job is to extract the parallelism inherent in the other components that define the computational work. The component-based design facilitates the separate development of simulation algorithms, physics models, boundary conditions and runtime infrastructure. This makes it very easy to set up a rudimentary implementation of any application and adding refinements incrementally from thereon. The user written application code is required to be written as a derived class inheriting from two Uintah base classes: `UintahParallelComponent` and `SimulationInterface`. At a minimum, implementations need to be provided for four virtual methods. These include: `problemSetup`, `scheduleInitialize`, `scheduleComputeStableTimestep`, and lastly, `scheduleTimeAdvance`. The purpose of the `problemSetup` method, as the name suggests, is the basic problem setup including the computational domain, the grid and its decomposition into patches, time step size, length of time for the simulation and specifics for periodically storing the simulation data. The set of variables that need to be defined at each grid cell, and an initialization of these variables are specified in the `scheduleInitialize` method. The computation of the time step in the simulation is specified in the `scheduleComputeStableTimestep` method, which might be required for scenarios where the time step size needs to adapt to ensure numerical stability (e.g., based on a Courant-Friedrichs-Lewy condition). The implementation of the actual algorithm for the numerical solution of the PDEs is specified in the `scheduleTimeAdvance` method. The set of tasks listed in this method are executed for every time step. Most of the coding effort goes into the setup of this method and care is required in correctly specifying the tasks, their input dependencies, the results generated and whether these transactions are with the old or the new data warehouses.

Load Balancing Patches are assigned to processors according to a space filling curve. Patch assignments can be updated between time steps at a user-specified frequency via Uintah's load balancing scheme. In their approach patch costs are computed using a combined profiling plus forecasting scheme. As described in detail in [54], their timing model associates weights with the various aspects that contribute to the overall runtime cost for a patch: the patch size, its particle load, and historical patch compute time, which is where system performance heterogeneity is accounted for. When load balancing, Uintah sets up a list of equations that it solves at runtime to estimate the model weights that determine the relative importance paid to each of these quantities. We found our load-balancing experiments with MiniAero hit a use case that had not been stress-tested by the Uintah team previously: a static workload (i.e., patches have an equal number of grid points, and no particles) with system performance heterogeneity (see Section 3.3 for experiment details). Although the Uintah runtime does account for system performance heterogeneity, in their use cases patch size and particle loads typically dominate the costs from a load-balancing perspective, and their runtime is tuned accordingly. We found that static workloads on systems with heterogeneous performance hit a degenerate case in their forecasting model implementation. Consequently, even when turned on, load balancing had no effect in our experiments. The Uintah team is making the necessary edits to their runtime and tuning their model to account for static application workloads on a system with performance heterogeneity.

2.4.3 Performance Portability

As mentioned above, Uintah is focused on insulating the application developer from the underlying architecture. This has resulted in modular runtime components that can take advantage of differing architectures simply by swapping a specific runtime component during compilation. The scheduler is an example of a component that has evolved with new architecture developments to provide better performance without major refactoring of application codes. The trend in multithreading resulted in the development of a threaded MPI scheduler while the development of a new unified scheduler was preceded by the prevalence of GPU's being utilized through computational kernels within many user codes. Threading performance was improved simply by adding a new scheduler while the addition of the GPU data warehouse and unified scheduler to provide explicit support for GPU offload necessitated refactoring of user applications. It is the decoupled nature of application codes from internal runtime components that has resulted in a modular runtime system that makes performance portability easier. Users can continue to work on extending their application codes without needing to understand the underlying runtime or architecture. This modularity is a key component of Uintah's performance portability but has the added side effect of requiring the runtime system developers to keep runtime components up to date and constantly develop with new architectures in mind.

The performance portability of Uintah in the context of heterogeneous architectures has been clearly demonstrated even though some of the runtime features that were utilized within these demonstrations are somewhat less mature. On both IBM and Cray HPC machines a small subset of application codes have been shown to provide good scalability and performance while utilizing up to 700k cores. Furthermore, the Unified scheduler has recently scaled a multi-level GPU implementation of their Reverse Monte Carlo Ray Tracing (RMCRT) radiation code to 16,384 GPUs, following their CPU-based RMCRT calculation shown in [55]. While current GPU data warehouse support is not entirely free of runtime system execution bugs and some fundamental design limitations remain, these are both very impressive efforts. Changes in architecture have been and continue to be reflected in the development of new components from the Uintah researchers, as is evidenced by their current work integrating with Kokkos to support automatic GPU parallelization and improved node-level memory management.

Overall, the Uintah team has a positive track record of developing their runtime system with upcoming architectures in mind. For example, from it's inception, the GPU-enabled Unified scheduler has been designed for an arbitrary number of GPUs per node [56], not just single GPU nodes. Furthermore, multiple MICs have also been considered [57, 58]. The team originally ran to machine capacity on both NFS Keeneland systems, each with 3 GPUs per node, and they regularly run on their own local 2-GPU/node cluster at the SCI Institute. The Unified Scheduler design is quite extensible, and they found that supporting more than 1 GPU per node was relatively straight forward due to task-level parallelism and the multi-stage task queue architecture built into the scheduler. Lastly, the Uintah team tends to be early users on most of the big machines (e.g., Titan, Jaguar) and they are actively planning and designing for future architectures. Within their role on the Predictive Science Academic Alliance Program II (PSAAP-II) centers, they aim to run on both Coral machines in 2018–2019 timeframe.

2.4.4 Maturity

Compared to perhaps most other task-based runtimes, Uintah has been very application driven right from its inception. The Uintah software suite was conceived at the Center for the Simulation of Accidental Fires and Explosions (C-SAFE) at the University of Utah in 1997. Consequently, Uintah's focus has always been on a set of target applications that involve chemically reacting gaseous flows in various settings: explosive devices placed in pool fires, industrial combustion systems (flares, oxy-coal combustion systems, gasifiers). Within this setting Uintah has demonstrated efficient implementations for a slew of multi-physics including fluid-dynamics, fluid-structure interactions, all modes of fluid-solid heat transfer (conduction, convection and radiation), chemical reactions and turbulence modelling. The runtime system software has packaged many of these physics based implementations as components that are available off-the-shelf for any relevant application but with a new target problem. For the set of target problems catered to, Uintah has demonstrated scalability on all major petascale platforms as mentioned in the previous subsection. That being said, the assumption of structured meshes, and the lack of support for unstructured meshes might make it unsuitable for quite a few applications.

2.4.5 Current Research Efforts

The current research efforts of Uintah are focused on the following areas:

- **Fault-Tolerance**: Efforts are afoot to leverage MPI-ULFM [59] to respond to hard faults. They are also working on task replication using AMR for duplicating tasks at a coarse level with marginal overhead, see Section 3.4.4.
- **GPU device utilization**: While the design is mostly in place to make the runtime handle GPUs transparently, some of the features are not fully functional yet and are actively being debugged.
- **Kokkos support** There is active development in interfacing Uintah with Kokkos. The team is working off a fork of Kokkos to relax some of the constraints in Kokkos and make it compatible with the node-level execution model of Uintah, see Section 4.4.2.

- **A self-aware runtime system**: Beyond dealing with faults at exascale, the team is working on making their runtime be self-aware and possibly compute within a specified energy budget. The plan is to measure the power consumption of components in the same way as CPU time is presently being monitored for load balancing. At present the Uintah runtime system uses a time series feedback loop to monitor the differences between predicted and actual CPU time. If similar information on energy usage will be available in the future it can also be monitored using the same kind of approach. If cores automatically reduce clock speeds, then it will be possible to use the present approach to load balance the calculation. If a core or node is not performing or is consuming too much power, then tasks can be moved elsewhere or delayed. Again this requires the nodal runtime system to monitor energy consumption while considering the total energy budget. Such a process must be self-adapting and so must build up information about the energy consumption in actual use. More difficult challenges arise from the energy expended in storage and in communications. In the area of communications, the Uintah team proposes to use location aware task-placement so as to ensure that the energy associated with communication is minimized, by making use of applications knowledge. When cores or memory slow down to conserve energy, their measurement-based load balancer will detect and mitigate the resulting load balance accordingly. Local performance variance will be handled automatically by node-level scheduling.

2.4.6 MiniAero Port

The Uintah implementation of MiniAero involved, at the highest level, fleshing out the `scheduleTimeAdvance` method that establishes the node-level task graph that is repeatedly executed for each time step. Decisions regarding the right level of task granularity are reflected in the body of this method. By and large, the same level of task granularity as in the baseline version was adopted, as is evident in Figure 2.11, with each task encapsulating the individual physics kernels for computing inviscid fluxes (both first and second order), viscous fluxes, gradients, stencil limiters and residual. This then leads to the set of individual tasks whose implementation needs to be provided by the user. Unlike the other runtimes, since the mesh is structured and constructed entirely implicitly from the problem parameters specified in the input files, no code was required for the mesh generation part. However, Uintah distinguishes between cell-centered variables and face-centered variables and the full list of variables of either type that will be used in the simulation (and need storing in the data warehouse) need to be declared upfront in the constructor of the specific simulation interface class. In this case this is illustrated in Figure 2.12 which shows one example of a cell-centered variable (`rho_CClabel`) and a face-centered variable (`flux_mass_FCXlabel`) declared in the constructor of the class `MiniAero`.

As mentioned earlier, each task body contains primarily two aspects: the input/output dependencies for the task (variables associated with the correct version of the data warehouses) and a pointer to a function which has the actual task implementation. This can be illustrated with the example of the task that computes the primitive variables from the conserved variables in MiniAero. Figures 2.13 and 2.14 show the bodies of the task `schedPrimitives`, which contains the pointer to the actual function `Primitives`, respectively. The keyword `requires` denotes an input dependency which in this case is the conserved variables from the current time step and hence is fetched from the new data warehouse denoted by the keyword `NewDW`, while the keyword `updates` denotes an output generated from this task, in this case all the primitive variables, which results in these variables being overwritten in the new data warehouse. A combined input/output dependency for variables that need to be modified in place can be specified with the keyword `modifies` which by default are for the new data warehouse. In this manner the bodies of all the tasks are largely just expressing the dependencies and little else. The body of the actual function, in this case `MiniAero::Primitives` contains the implementation. Even here, there is boiler-plate code that reflects the dependencies (keywords such as `new_dw->get`, `new_dw->allocateAndPut` and `new_dw->getModifiable`) that appears first followed by the actual computation itself which is expressed inside a `for` loop that iterates over all the cells of the current patch. Uintah provides transparent handles to the iterators over all the patches that might be within a node, and all the cells that are on each patch, thus hiding the physical data completely from the user. In this manner, the tasks doing the actual computation for the physics kernels could reuse a lot of the Kokkos functors from the baseline version. Accordingly, the portion of the code that does not deal with the actual computation but rather requires specifying the task bodies and dependencies was comparable, in terms of number of lines, to the portion of the code containing the actual computation. However, the amount of time taken to write the two portions was significantly different with the former taking much less compared to the latter. In total, a rudimentary implementation of MiniAero in Uintah was completed by the Sandia team working with the Uintah team in a little over two days.

```
void MiniAero::scheduleTimeAdvance(const LevelP& level,
                                   SchedulerP& sched)
{
  for(int k=0; k<d_RKSteps; k++ ){

    if(d_viscousFlow || d_secondOrder){
      schedGradients(level, sched, k);
    }

    if(d_secondOrder){
      schedLimiters(level, sched, k);
      schedSecondOrderFaceFlux(level, sched, k);
      schedSecondOrderDissipativeFaceFlux(level, sched, k);
    }
    else{
      schedCellCenteredFlux(level, sched, k);
      schedFaceCenteredFlux(level, sched, k);
      schedDissipativeFaceFlux(level, sched, k);
    }

    if(d_viscousFlow){
      schedViscousFaceFlux(level, sched, k);
    }
    schedUpdateResidual(level, sched, k);
    schedUpdate_RK_State(level, sched, k);
    schedPrimitives(level,sched, k);
  }

  schedUpdateSolution( level, sched);
  schedPrimitives(level,sched, d_RKSteps);
}
```

Figure 2.11: Uintah `scheduleTimeAdvance` method with a coarse view of the taskgraph

```
MiniAero::MiniAero(const ProcessorGroup* myworld)
    : UintahParallelComponent(myworld)
  rho_CClabel          = VarLabel::create("density",
                                  CCVariable<double>::getTypeDescription());
  flux_mass_FCXlabel = VarLabel::create("faceX_flux_mass",
                                  SFCXVariable<double>::getTypeDescription());
}
```

Figure 2.12: Constructor of the derived simulation interface class `MiniAero` illustrating the types of Uintah variables.

```
void MiniAero::schedPrimitives(const LevelP& level,
                                SchedulerP& sched,
                                const int RK_step)
{
  Task* task = scinew Task("MiniAero::Primitives", this,
                           &MiniAero::Primitives, RK_step);

  task->requires(Task::NewDW, conserved_label,Ghost::None);

  if (RK_step == 0 ){
    task->computes(rho_CClabel);
    task->computes(vel_CClabel);
    task->computes(press_CClabel);
    task->computes(temp_CClabel);

    if(d_viscousFlow){
      task->computes(viscosityLabel);
    }
  } else {
    task->modifies(rho_CClabel);
    task->modifies(vel_CClabel);
    task->modifies(press_CClabel);
    task->modifies(temp_CClabel);

    if(d_viscousFlow){
      task->modifies(viscosityLabel);
    }
  }

  sched->addTask(task, level->eachPatch(), sharedState_->allMaterials());
}
```

Figure 2.13: Body of the task that computes the primitive variables in Uintah port of MiniAero.

```
void MiniAero::Primitives(const ProcessorGroup* /*pg*/,
                          const PatchSubset* patches,
                          const MaterialSubset* /*matls*/,
                          DataWarehouse* old_dw,
                          DataWarehouse* new_dw,
                          const int RK_step)
{

  for(int p=0;p<patches->size();p++){
    const Patch* patch = patches->get(p);

    // Requires...
    constCCVariable<Vector5> conserved;
    Ghost::GhostType  gn  = Ghost::None;
    new_dw->get( conserved,  conserved_label, 0, patch, gn, 0 );

    // Provides...
    CCVariable<double> rho_CC, pressure_CC, Temp_CC, viscosity;
    CCVariable<Vector> vel_CC;

    if ( RK_step == 0 || RK_step == d_RKSteps) {
      new_dw->allocateAndPut( rho_CC,      rho_CClabel,       0, patch );
      new_dw->allocateAndPut( pressure_CC, press_CClabel,     0, patch );
      new_dw->allocateAndPut( Temp_CC,     temp_CClabel,      0, patch );
      new_dw->allocateAndPut( vel_CC,      vel_CClabel,       0, patch );
      if(d_viscousFlow){
        new_dw->allocateAndPut( viscosity,  viscosityLabel, 0, patch );
      }
    } else {
      new_dw->getModifiable( rho_CC,      rho_CClabel,       0, patch );
      new_dw->getModifiable( pressure_CC, press_CClabel,     0, patch );
      new_dw->getModifiable( Temp_CC,     temp_CClabel,      0, patch );
      new_dw->getModifiable( vel_CC,      vel_CClabel,       0, patch );
      if(d_viscousFlow){
        new_dw->getModifiable( viscosity,  viscosityLabel, 0, patch );
      }
    }

    for(CellIterator iter = patch->getCellIterator(); !iter.done(); iter++) {
      IntVector c = *iter;
      rho_CC[c]  = conserved[c].rho;

      vel_CC[c].x( conserved[c].momX/rho_CC[c] );
      vel_CC[c].y( conserved[c].momY/rho_CC[c] );
      vel_CC[c].z( conserved[c].momZ/rho_CC[c] );

      Temp_CC[c] = (d_gamma-1.)/d_R*(conserved[c].eng/conserved[c].rho
                   - 0.5*vel_CC[c].length2());

      pressure_CC[c] = rho_CC[c] * d_R * Temp_CC[c];
    }
  }
}
```

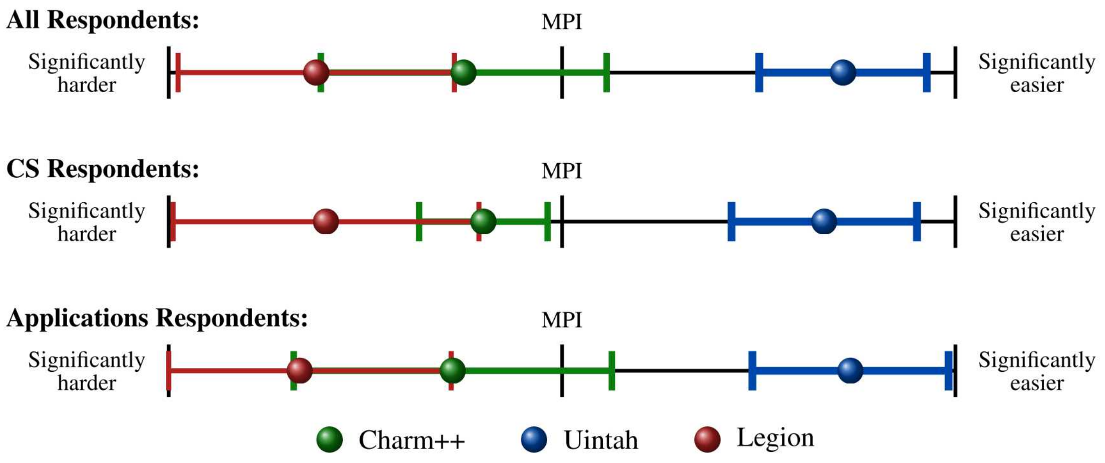
Figure 2.14: Body of the task the function encapsulating the actual computation of primitive variables in MiniAero.

2.5 Comparative Analysis

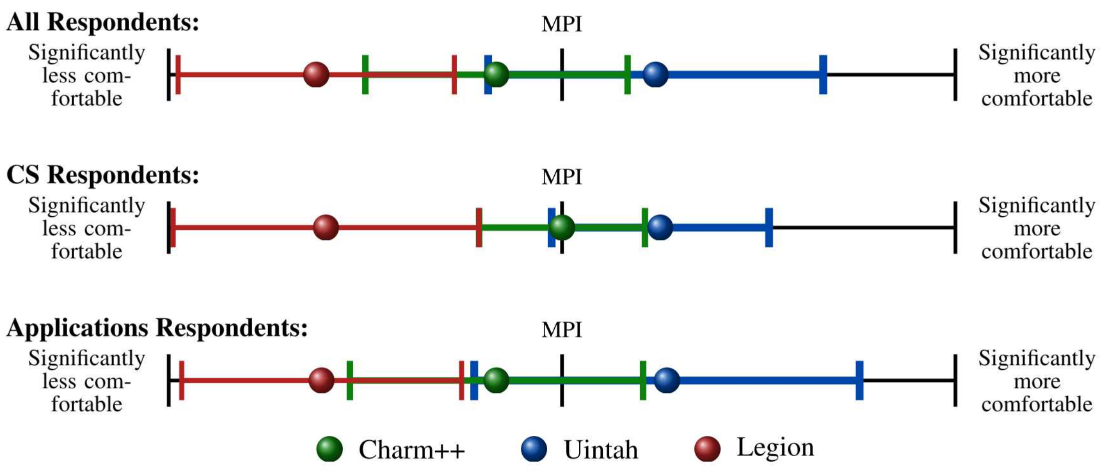
In this section we compare and contrast the three runtimes across a number of subjective programmability measures. We posed questions to the attendees of each bootcamp. They were asked to provide a response on a seven-point scale, *relative to MPI*. For each question below, a low score indicates a response "significantly worse than MPI", a mid-range score indicates "comparable to MPI", and a high score indicates a response "significantly better than MPI". For each question, we show overall responses, along with the responses separated according to respondents with an applications (Apps) background versus respondents with a computer science (CS) background.

The declarative programming style of both Legion and Uintah led to high ratings with regards to the separation of of machine-specific code from domain-specific code. Furthermore, all respondents agreed that the DSL-like nature of Uintah led to its simple learning curve, and familiarity. Although both Uintah and Legion provide a declarative programming style that differs from the imperative style of Charm++ and MPI, the APIs and abstractions put in place by the Uintah team made a very big difference with regards to learning curve and familiarity (when compared to Legion). Charm++ rated slightly worse than MPI in both learning curve and familiarity due to the subtle shift from the CSP to actor execution model, combined with the template issues with Charm++'s use of the `ci` files. In terms of verbosity and readability of code, Uintah again fared best, with Legion scoring the lowest due to the fact that it is intended to be a layer for DSL and library authors, rather than the typical application developer. In general it was deemed that AMT code maintenance would be comparable to or better than MPI, with Uintah ranking higher than the rest. This is not surprising given one of Uintah's primary goals is to have application code remain unchanged from 600 to 600K cores. Application and CS developers had different rankings with regards to the runtimes' ability to express parallelism. CS respondents ranked Uintah highest and Legion the lowest, whereas the Apps developers ranked Legion the highest and Uintah the lowest. Uintah's data warehouse motivated the respondents to rate it the highest from the perspective of workflow management. The respondents felt all of the runtimes could have better documentation, although Charm++ has much more complete documentation than either of the other two runtimes. From a debugging perspective, Uintah provides an extensive set of options for outputting debug statements that respondents felt were very useful in the debugging process. However, Legion has the infrastructure in place to provide compile-time checks and so it fared better in this regard. Lastly, when it came to data structure flexibility, Uintah's limitation to structured meshes was a major concern. Legion's data model, while extremely useful with regards to automatically extracting parallelism, does impose some rigidity in comparison to Charm++, whose lack of data model and PUP interface makes it very flexible with regards to data structures.

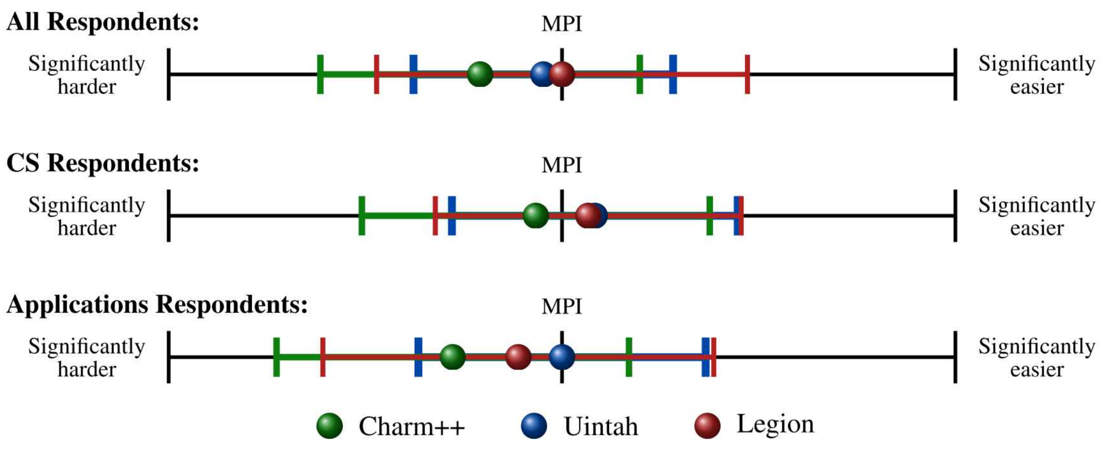
Simplicity and Learning curve: How easy is it to get started expressing scientific concepts in the runtime's programming model?



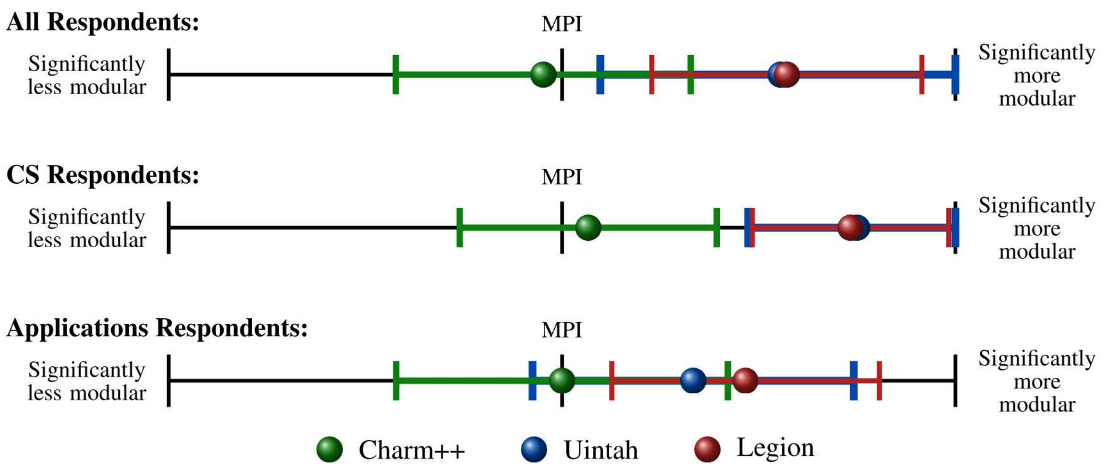
Familiarity: How similar does the runtime paradigm/philosophy feel to what you are used to? How comfortable are you coding in the runtime's programming model?



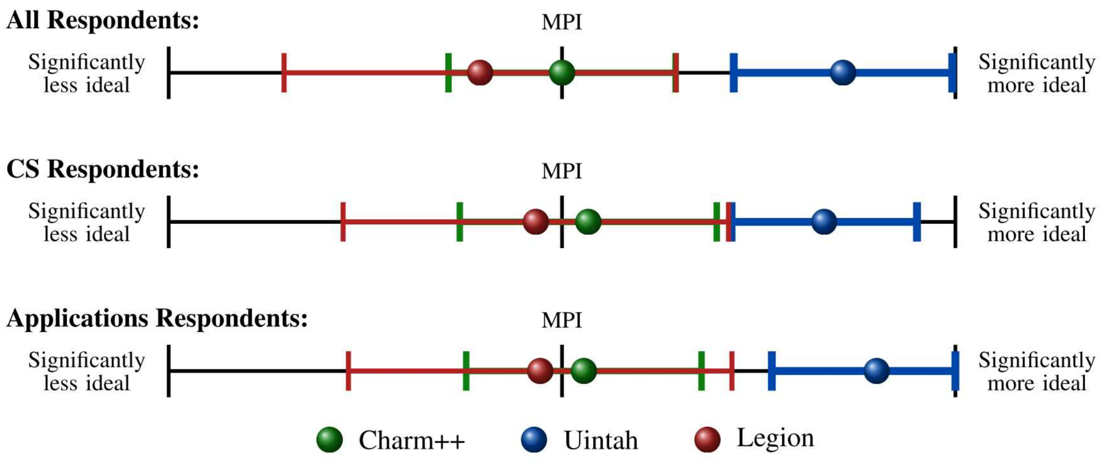
Correctness Transparency: How easy is it to ascertain and develop confidence that your program is getting the right answer consistently and across a wide variety of hardware?



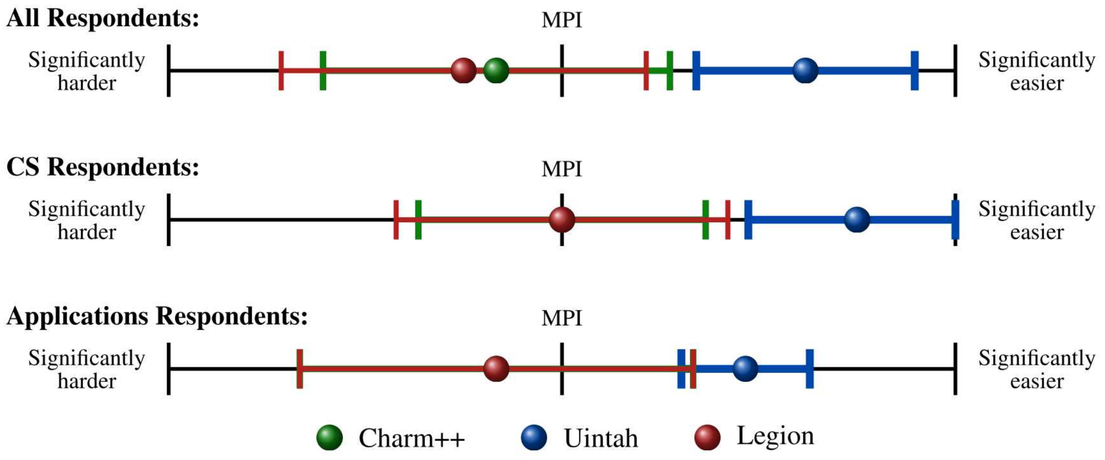
Abstractions: How separate is the domain-specific code from the machine-specific code, and how modular or swappable is the connection between the two?



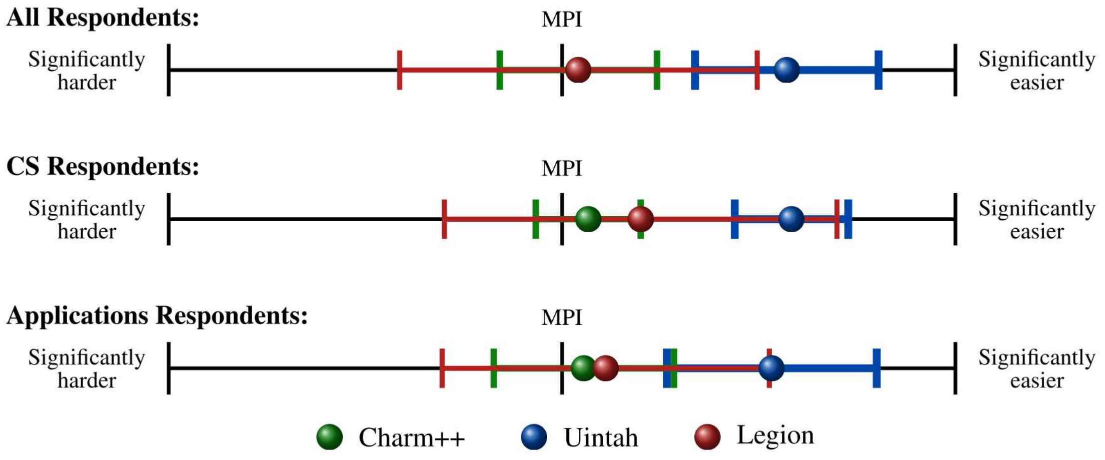
Verbosity: How close is the application code to an ideal, "concepts-only" pseudo-code-like specification of the problem?



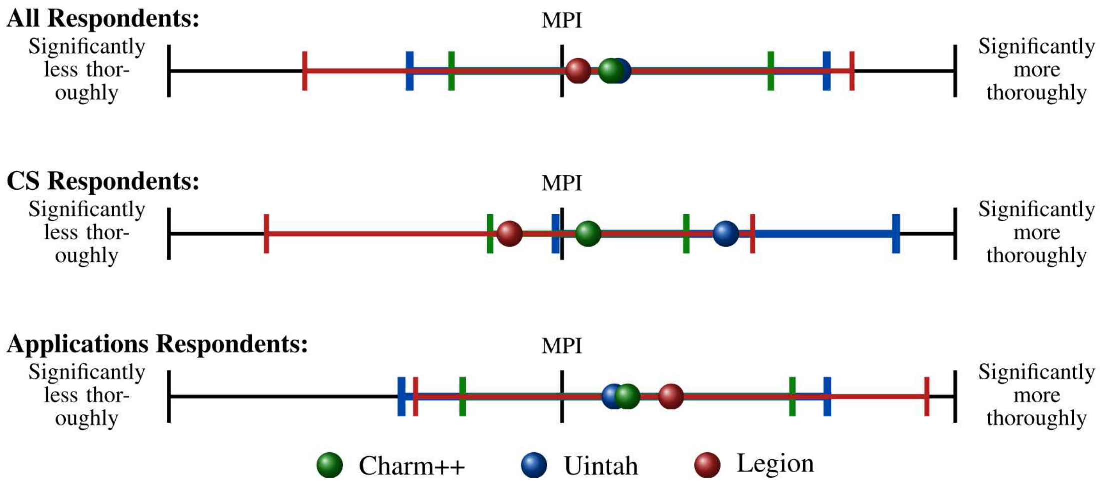
Readability: How easy is it to understand code written by someone else using the runtime's programming model?



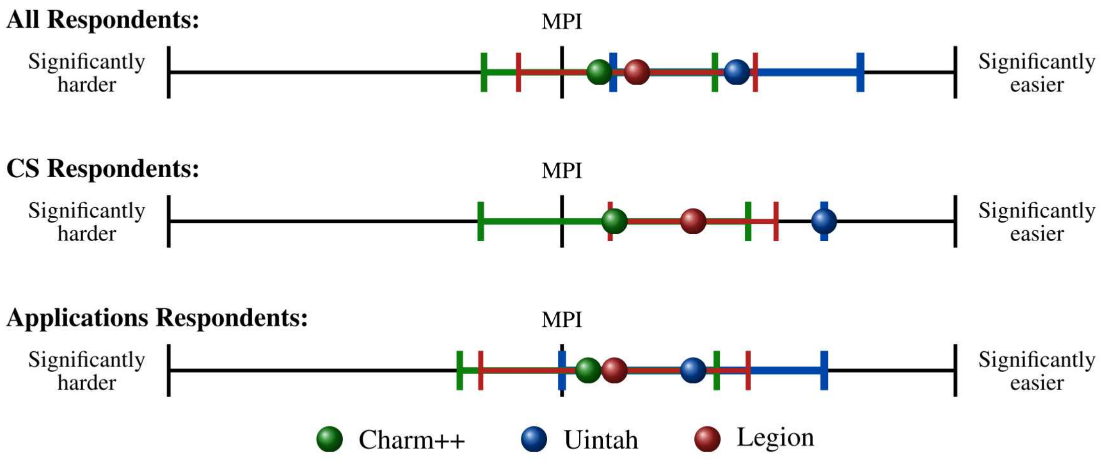
Maintainability of User Code: How easy is it to maintain code written using the runtime's programming model?



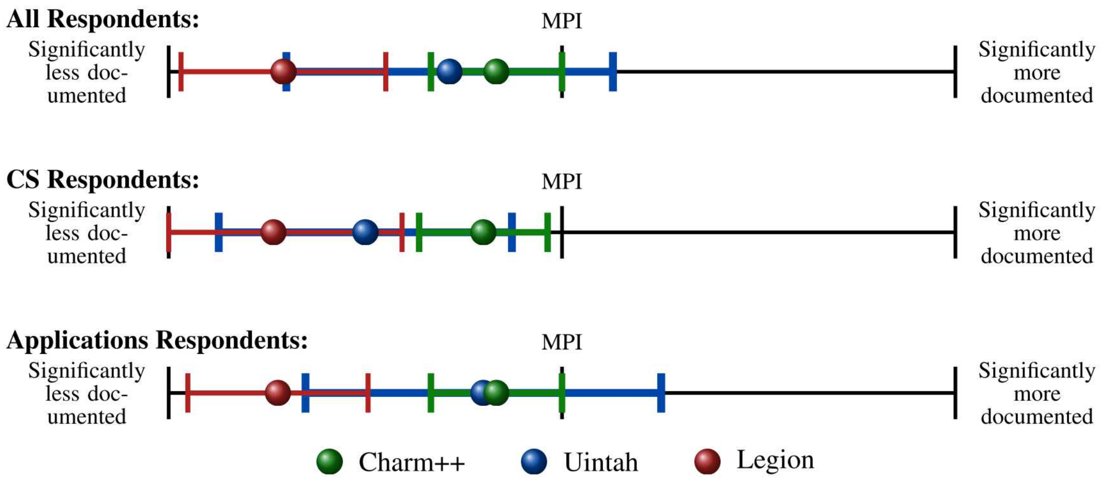
Expressivity: How thoroughly can all of the parallel aspects of a domain-specific problem be expressed in the runtime's programming model, whether or not these aspects are utilized by the runtime?



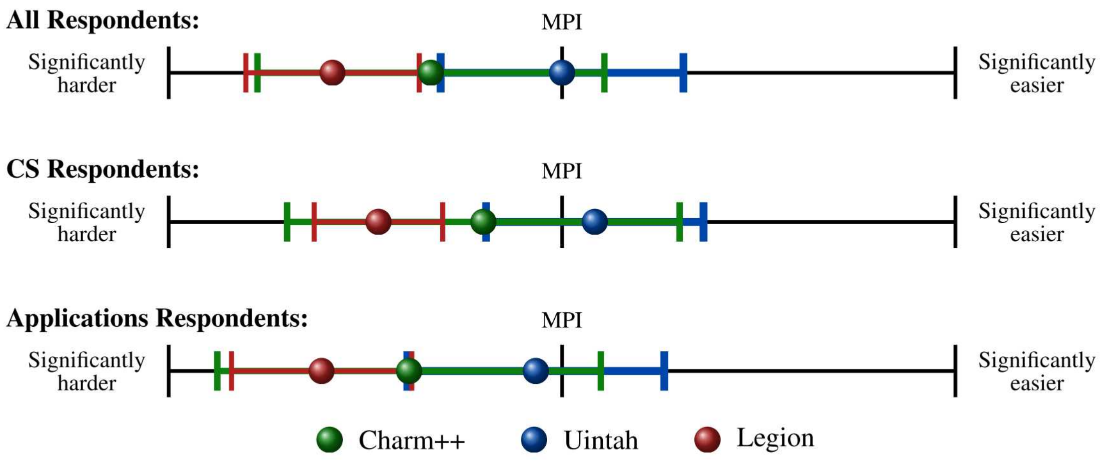
Workflow Management: To what extent does the runtime make code coupling, I/O, and analysis (both *in-situ* or post-processing) easy?



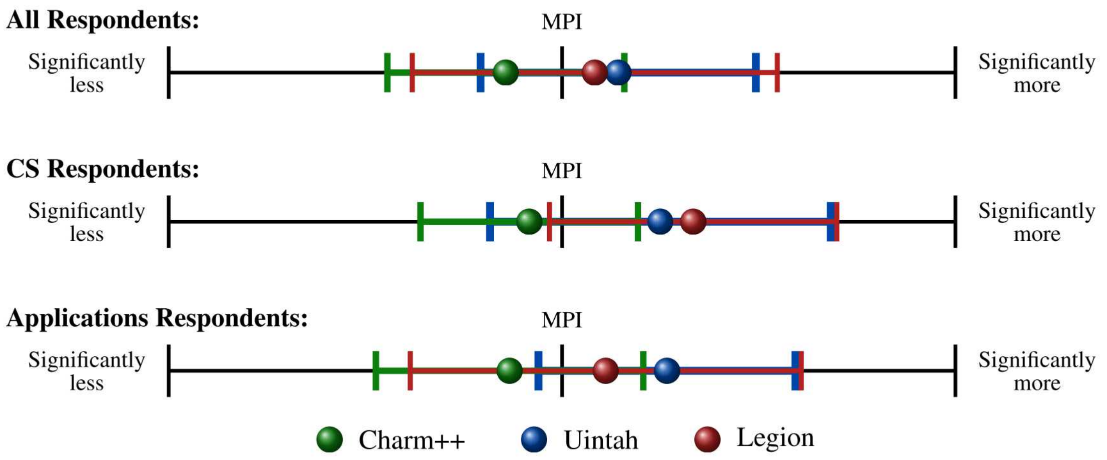
Documentation: How well are the available features documented?



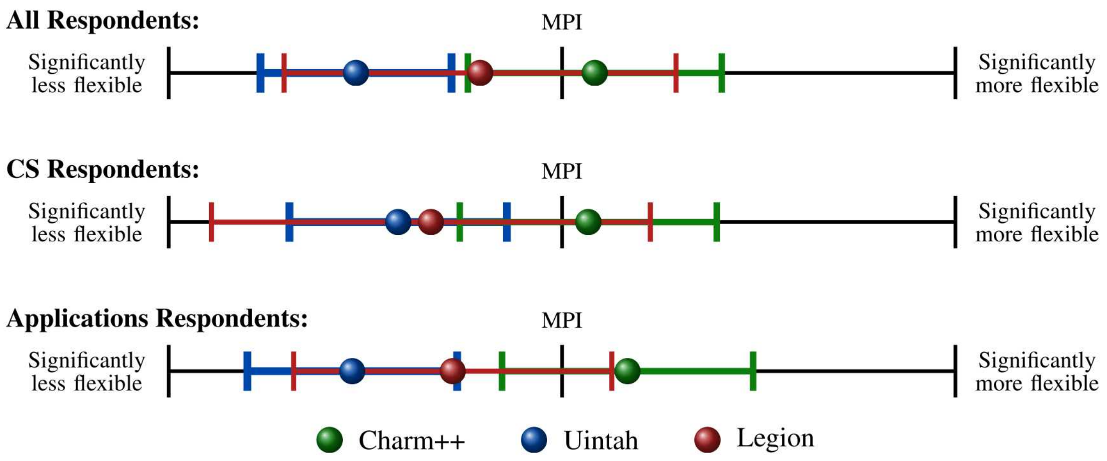
DebugAbility: Once a program is observed to crash, how easy is it to fix the problem given the tools provided by the runtime or commercial tools?



Compile Time Error Detectability: To what extent does the programming model allow the compiler and runtime to detect errors at compile time?



Data Structure Flexibility: Are users able to define data structures of arbitrary complexity?



2.6 Learning Curve and Implementation Timelines

2.6.1 Charm++

As already discussed, the actor execution model in Charm++ can be structured to match intrinsically data-parallel codes, creating a natural migration path in which MPI ranks are encapsulated in chares. The initial port of MiniAero occurred over a 4-day working group in direct collaboration with two Charm++ developers. The coding was essentially complete after this session, producing a functional application and even testing some fault-tolerance features. Much of the time spent during the working group was in dealing with the C++ template incompatibility issue discussed above, which complicated a direct port of the Kokkos MiniAero version. Given a non-template C++ code, the port likely would have been completed in less than two days. Since the core code was completed, various optimizations and bug fixes have been introduced over the last 6 months.

2.6.2 Legion

The time to port MiniAero to the Legion runtime was easily the longest of the three runtimes, which supports the Legion design constraint that productivity will be sacrificed for performance. In contrast to the Charm++ port which used much of the existing MiniAero mesh generation code and the Uintah port which provides its own mesh generation infrastructure, in the the Legion port, we completely rewrote the mesh generation portion to fit more closely with the Legion structure and data model. This—in conjunction with learning the Legion API and data model, and experimenting with coding styles—took about one to two weeks; the programming time could probably be reduced to about a day for a knowledgeable Legion developer.

At that time, porting physics routines into the Legion task launch structure began. As discussed in Section 2.3.7, this consisted of identifying the field/data privileges (read-only, read-write, write-only, reduction) and creating both the task-launch routines and then the wrappers inside the tasks for accessing the field data. Although there was some rewriting and improving code modularity during this time, the main porting took approximately two weeks until we were successfully running the MiniAero regression tests on shared-memory machines.

We then continued improving the code structure at the same time that we investigated optimizations. We used the Legion profiling and debugging tools (See discussion on page 65) as a guide to what optimizations were needed and how well they performed. The default Mapper was shown to not be sufficient for use with MiniAero, so we implemented a MiniAero-specific version with the help of the Legion developers. After some effort (involving direct guidance from Legion developers) this resulted in a speedup comparable to MPI and Charm++ that reflected better task overlap in the profile output.

We also investigated other optimizations including improving the field accessors, removing unneeded reduction operations, restructuring of task routines, and data structure improvements. Some of these were driven by output from VTune and other profiling tools. Some additional time was used in adapting the code to changes in the Legion API since we were using the development version of the runtime instead of the more stable release version, in order to make use of optimizations and bug fixes provided by the Legion developers.

We were also investigating distributed memory execution issues during this time until it was decided that the underlying runtime issue would not be solved in time for us to do meaningful benchmarking of the Legion implementation in this study.

Overall, work has been done on the Legion port at various levels of effort from about the end of January 2015 until late July 2015; this includes periods of near full-time effort along with long periods of little to no effort. With the knowledge gained in the initial port, we could definitely reduce the time in a future port of similar applications, but as stated many times before, the Legion API and data model value performance over productivity. Although we were not able to verify the performance claim on distributed memory machines; we were able to verify the productivity claim. The other claim that higher-level tools can provide productivity was also verified in the Regent port of MiniAero done by researchers at Stanford. They did their port in a very short period of time, and the results of their work provided us with many examples of where we could optimize our low-level Legion port.

2.6.3 Uintah

As mentioned above, an initial, first-order, non-viscous, proof of concept serial port of MiniAero to Uintah was essentially complete after a two-day working group with Uintah developers. The full version of the code was finished and tested with several additional weeks of part time effort. Various load balancing optimizations and performance tweaks have been tested since then. Debugging of the load balancers has been the primary time consumer, with an initial correct implementation representing only a small portion of the time spent on Uintah.

2.7 Tools Support

Parallel and distributed development tools play a critical role in the development and upkeep of parallel scientific applications. Tools provide the means for developers to analyze application behavior running across a set of distributed resources in order to verify bug-free operation and correctness and to assess performance, resource utilization, and other metrics such as power and energy. The vast majority of tools may be categorized in terms of their primary functionality (i.e., debugging, correctness checks, or performance analysis), instrumentation methodology (i.e., dynamic vs. static, source-code vs. binary, transparent vs. explicit), measurement methodology (e.g., event tracing, event sampling), and analysis interactivity (i.e., online vs. offline). Debuggers and correctness tools aid in identifying application correctness issues, whereas performance tools aid in identifying performance-related issues.

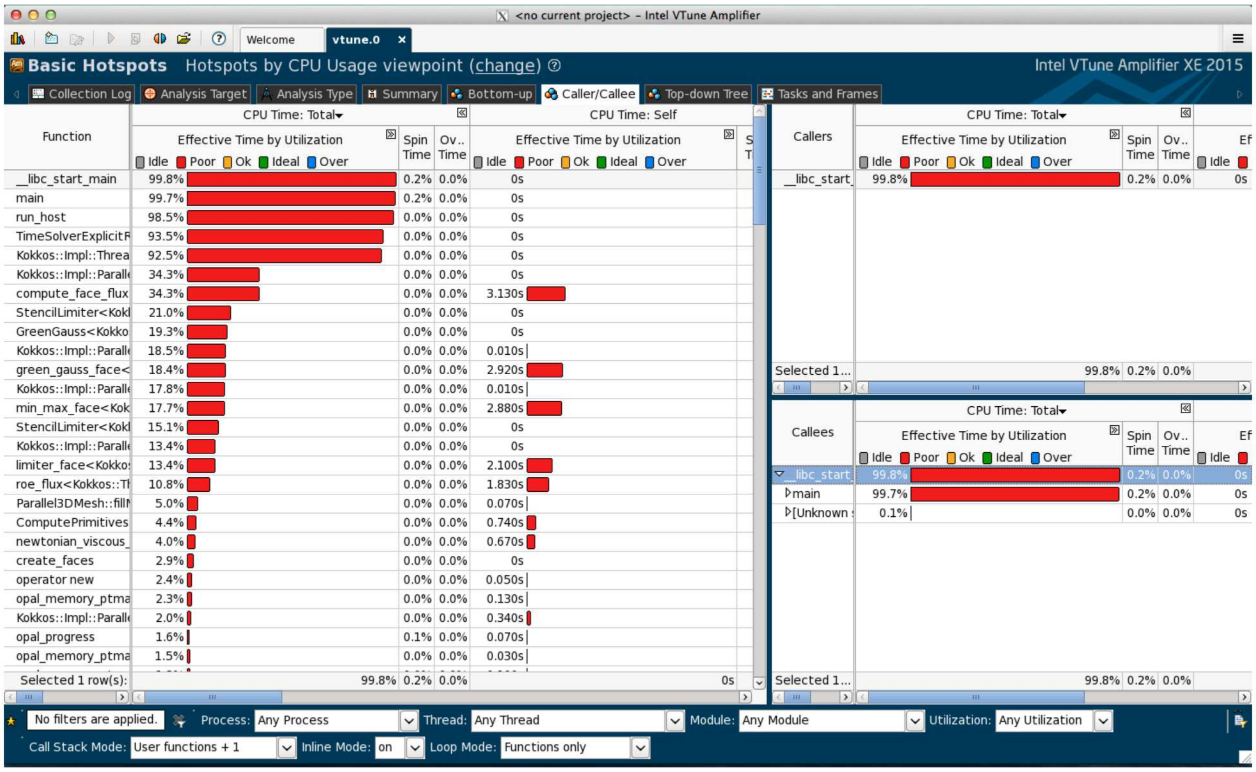
Tools of both types make use of a wide variety of instrumentation methodologies that can range from transparent instrumentation at the binary or link level to explicit source code annotations. This naturally has a significant impact on the type of information that can be collected and the impact the tool has on the overall application workflow. Closely connected to the instrumentation methodology is the measurement methodology: event tracing gathers data by activating a set of instrumentation probes at every event associated with the trace (e.g., using function interpositioning), whereas sampling-based measurements are typically interrupt-driven and only provide a statistical view of application behavior (e.g., program counter sampling). Online analysis tools are interactive and allow for user-driven steering of analyses throughout an application's lifetime. By contrast, offline analysis tools are not meant to be driven by an end-user during an application's execution. That is, the tool is started with the application; runs alongside the application until application termination; and tool data are written, post-processed, and then analyzed.

We first present a survey of existing tools that support HPC application development, and our experiences with them during our AMT runtime study. This is followed by a discussion of the tool support provided directly by Charm++, Legion, and Uintah. We conclude with a discussion of interesting research challenges that AMT runtimes introduce in the development of HPC tools.

2.7.1 General Tools

We begin with a discussion of four tools applicable to general HPC application development—PAPI [60,61], VTune [62], CrayPAT [63] and Open|SpeedShop [64], the major HPC programming tools installed in the HPC systems at DOE laboratories. We also discuss recent research efforts in this area, which (while not production tools, yet) can provide novel insights into application performance. Note that we do not include TAU [65, 66], which is very comparable to CrayPAT, due to its limited availability on the HPC systems at Sandia.

PAPI PAPI [60, 61] has been the de-facto standard software used to access the performance counter hardware across different CPU families. The PAPI is platform agnostic, and allows the users to introspect the performance of a program by delineating the source code via a few API calls (initialize, read counters, and reset counters). The usage is very similar to inserting wall clock functions to measure the execution time for the events of interest. Owing to its simplicity and interoperability, PAPI has been a major building block for more sophisticated performance analysis and profiling tools for both event-tracing and sampling methods. Today, PAPI has been integrated with major performance tools including TAU [65], Open|SpeedShop [64], HPCToolkit [67]. Also, PAPI has been integrated into some AMT runtime systems such as Charm++ and Uintah to obtain the performance counter information.



<no current project> – Intel VTune Amplifier
Welcome
vtune.0
Basic Hotspots Hotspots by CPU Usage viewpoint (change)
Intel VTune Amplifier XE 2015
Collection Log
Analysis Target
Analysis Type
Summary
Bottom-up
Caller/Callee
Top-down Tree
Tasks and Frames

Function	CPU Time: Total – Effective Time by Utilization (⬜ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	CPU Time: Total – Spin Time	CPU Time: Total – Ov.. Time	CPU Time: Self – Effective Time by Utilization (⬜ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)
__libc_start_main	99.8%	0.2%	0.0%	0s
main	99.7%	0.2%	0.0%	0s
run_host	98.5%	0.0%	0.0%	0s
TimeSolverExplicitR	93.5%	0.0%	0.0%	0s
Kokkos::Impl::Threa	92.5%	0.0%	0.0%	0s
Kokkos::Impl::Paralle	34.3%	0.0%	0.0%	0s
compute_face_flux	34.3%	0.0%	0.0%	3.130s
StencilLimiter<Kokl	21.0%	0.0%	0.0%	0s
GreenGauss<Kokko	19.3%	0.0%	0.0%	0s
Kokkos::Impl::Paralle	18.5%	0.0%	0.0%	0.010s
green_gauss_face<	18.4%	0.0%	0.0%	2.920s
Kokkos::Impl::Paralle	17.8%	0.0%	0.0%	0.010s
min_max_face<Kok	17.7%	0.0%	0.0%	2.880s
StencilLimiter<Kokl	15.1%	0.0%	0.0%	0s
Kokkos::Impl::Paralle	13.4%	0.0%	0.0%	0s
limiter_face<Kokko	13.4%	0.0%	0.0%	2.100s
roe_flux<Kokkos::Tl	10.8%	0.0%	0.0%	1.830s
Parallel3DMesh::fillN	5.0%	0.0%	0.0%	0.070s
ComputePrimitives	4.4%	0.0%	0.0%	0.740s
newtonian_viscous_	4.0%	0.0%	0.0%	0.670s
create_faces	2.9%	0.0%	0.0%	0s
operator new	2.4%	0.0%	0.0%	0.050s
opal_memory_ptma	2.3%	0.0%	0.0%	0.130s
Kokkos::Impl::Paralle	2.0%	0.0%	0.0%	0.340s
opal_progress	1.6%	0.1%	0.0%	0.070s
opal_memory_ptma	1.5%	0.0%	0.0%	0.030s
Selected 1 row(s):	99.8%	0.2%	0.0%	0s

Callers	CPU Time: Total – Effective Time by Utilization (⬜ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Spin Time	Ov.. Time	Ef (⬜ Idle 🟥)
__libc_start_	99.8%	0.2%	0.0%	0s
Selected 1...	99.8%	0.2%	0.0%	

Callees	CPU Time: Total – Effective Time by Utilization (⬜ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Spin Time	Ov.. Time	Ef (⬜ Idle 🟥)
▽ __libc_start_	99.8%	0.2%	0.0%	0s
▷main	99.7%	0.2%	0.0%	0s
▷[Unknown :	0.1%	0.0%	0.0%	0s
Selected 1...	99.8%	0.2%	0.0%	

No filters are applied. Process: Any Process Thread: Any Thread Module: Any Module Utilization: Any Utilization

Call Stack Mode: User functions + 1 Inline Mode: on Loop Mode: Functions only

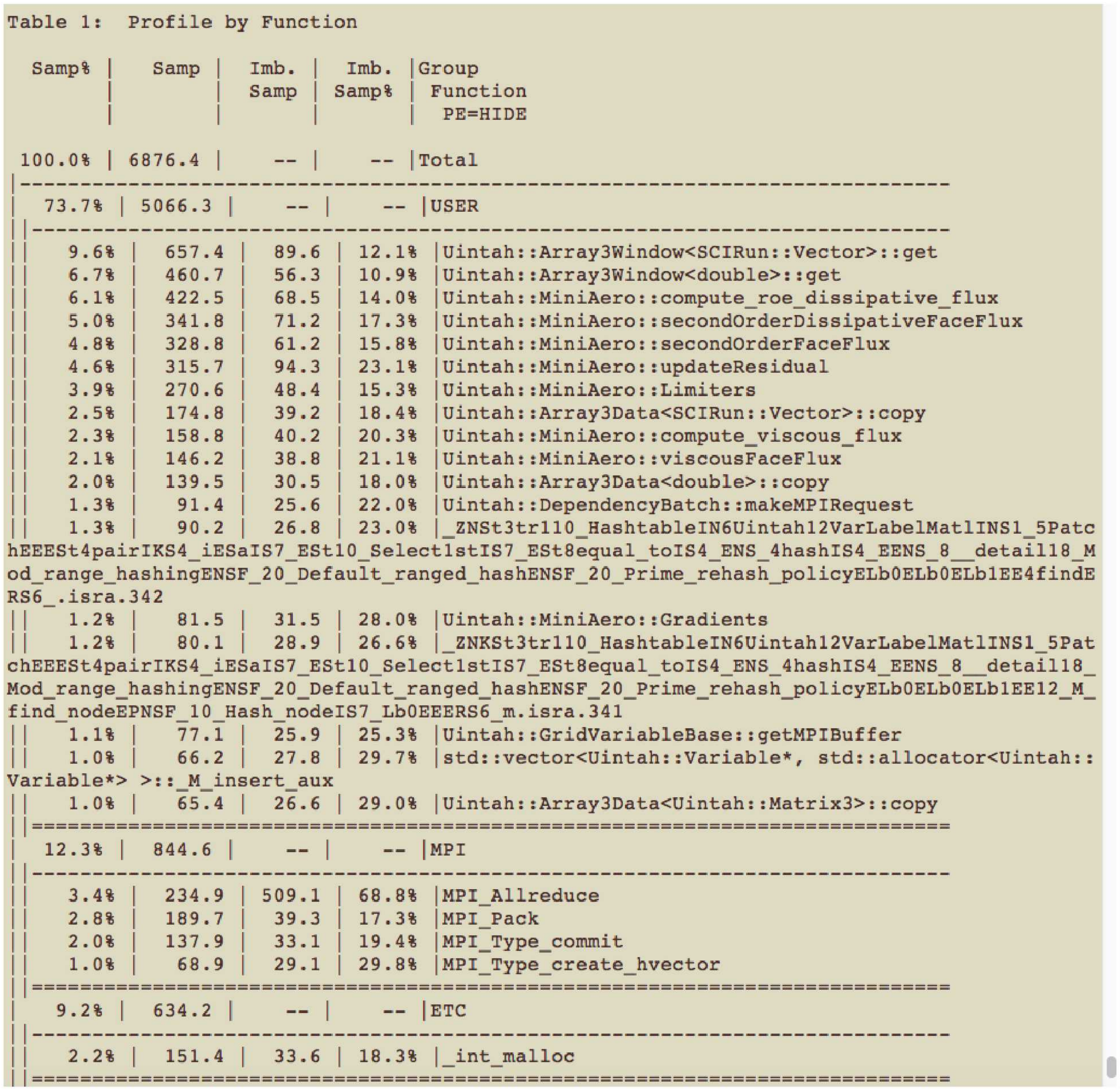
Figure 2.15: Original version of MiniAero analyzed through VTune

VTune Amplifier VTune^{TM}Amplifier (VTune) [62] is a performance profiler tool developed by Intel, targeted at Intel and AMD X86 architecture. VTune provides performance profiling capabilities with GUIs to facilitate performance assessment of a program across a range of granularities, including procedure, loop, and event. VTune employs a sampling approach for performance profiling and does not require any re-linking or code modification for instrumentation. One of the strengths of VTune is its support for a variety of programming languages; C, C++, Fortran, in addition to other languages not popular in HPC applications such as C# and Java.

VTune is claimed to be applicable for major parallel programming models, including threads, OpenMP and MPI. However, our experience indicates that its applicability is severely limited for distributed memory programming due to a lack of scalable data aggregation. With the default sampling options, VTune creates a large performance data file for each process, which can reach 10 Gbytes just for 10 seconds of program execution. This inflation of the file sizes prohibits the performance measurements at large process counts, saturating the bandwidth of the global I/O subsystem and network. In our experiments on Shepard testbed system at Sandia, VTune caused a significant slowdown in the execution time of MiniAero (10 times or more), generating meaningless performance data. We note that the sampling frequency can be lowered to ensure that program execution remains unaffected by the I/O activities. However, one would need to increase the duration of the simulation runs dramatically (to 3 hours or more) to guarantee the integrity of the performance statistics. For these reasons, we decided to exclude VTune from further investigations.

Cray Performance Measurement and Analysis Tools (CrayPAT) Cray Performance Measurement and Analysis Tools (CrayPAT) [63] is a suite of off-line performance profiling and analysis tools pre-installed in Cray platforms. Like VTune, CrayPat provides a large variety of capabilities, including sampling and tracing for the parallel programming model supported by Cray's proprietary programming environment (MPI, UPC, CoArray Fortran, and SHMEM). CrayPAT aggregates all performance data stored in memory of the compute nodes through MRNet [68], which provides a tree overlay network to coordinate the data traffic across compute nodes reducing the performance impact on the global file I/O.

For the preparation of performance profiling, CrayPAT requires re-building an executable to instrument probes through



```
Table 1:  Profile by Function

  Samp% |   Samp |  Imb. |  Imb. |Group
        |        |  Samp | Samp% | Function
        |        |       |       |  PE=HIDE

 100.0% | 6876.4 |    -- |    -- |Total
|------------------------------------------------------------------------------
|  73.7% | 5066.3 |    -- |    -- |USER
||-----------------------------------------------------------------------------
||   9.6% |  657.4 |  89.6 | 12.1% |Uintah::Array3Window<SCIRun::Vector>::get
||   6.7% |  460.7 |  56.3 | 10.9% |Uintah::Array3Window<double>::get
||   6.1% |  422.5 |  68.5 | 14.0% |Uintah::MiniAero::compute_roe_dissipative_flux
||   5.0% |  341.8 |  71.2 | 17.3% |Uintah::MiniAero::secondOrderDissipativeFaceFlux
||   4.8% |  328.8 |  61.2 | 15.8% |Uintah::MiniAero::secondOrderFaceFlux
||   4.6% |  315.7 |  94.3 | 23.1% |Uintah::MiniAero::updateResidual
||   3.9% |  270.6 |  48.4 | 15.3% |Uintah::MiniAero::Limiters
||   2.5% |  174.8 |  39.2 | 18.4% |Uintah::Array3Data<SCIRun::Vector>::copy
||   2.3% |  158.8 |  40.2 | 20.3% |Uintah::MiniAero::compute_viscous_flux
||   2.1% |  146.2 |  38.8 | 21.1% |Uintah::MiniAero::viscousFaceFlux
||   2.0% |  139.5 |  30.5 | 18.0% |Uintah::Array3Data<double>::copy
||   1.3% |   91.4 |  25.6 | 22.0% |Uintah::DependencyBatch::makeMPIRequest
||   1.3% |   90.2 |  26.8 | 23.0% |_ZNSt3tr110_HashtableIN6Uintah12VarLabelMatlINS1_5Patc
hEEESt4pairIKS4_iESaIS7_ESt10_Select1stIS7_ESt8equal_toIS4_ENS_4hashIS4_EENS_8__detail18_M
od_range_hashingENSF_20_Default_ranged_hashENSF_20_Prime_rehash_policyELb0ELb0ELb1EE4findE
RS6_.isra.342
||   1.2% |   81.5 |  31.5 | 28.0% |Uintah::MiniAero::Gradients
||   1.2% |   80.1 |  28.9 | 26.6% |_ZNKSt3tr110_HashtableIN6Uintah12VarLabelMatlINS1_5Pat
chEEESt4pairIKS4_iESaIS7_ESt10_Select1stIS7_ESt8equal_toIS4_ENS_4hashIS4_EENS_8__detail18_
Mod_range_hashingENSF_20_Default_ranged_hashENSF_20_Prime_rehash_policyELb0ELb0ELb1EE12_M_
find_nodeEPNSF_10_Hash_nodeIS7_Lb0EEERS6_m.isra.341
||   1.1% |   77.1 |  25.9 | 25.3% |Uintah::GridVariableBase::getMPIBuffer
||   1.0% |   66.2 |  27.8 | 29.7% |std::vector<Uintah::Variable*, std::allocator<Uintah::
Variable*> >::_M_insert_aux
||   1.0% |   65.4 |  26.6 | 29.0% |Uintah::Array3Data<Uintah::Matrix3>::copy
||=============================================================================
|  12.3% |  844.6 |    -- |    -- |MPI
||-----------------------------------------------------------------------------
||   3.4% |  234.9 | 509.1 | 68.8% |MPI_Allreduce
||   2.8% |  189.7 |  39.3 | 17.3% |MPI_Pack
||   2.0% |  137.9 |  33.1 | 19.4% |MPI_Type_commit
||   1.0% |   68.9 |  29.1 | 29.8% |MPI_Type_create_hvector
||=============================================================================
|   9.2% |  634.2 |    -- |    -- |ETC
||-----------------------------------------------------------------------------
||   2.2% |  151.4 |  33.6 | 18.3% |_int_malloc
||=============================================================================
```

Figure 2.16: CUI output from CrayPat. The table indicates the performance of MiniAero implemented with Uintah

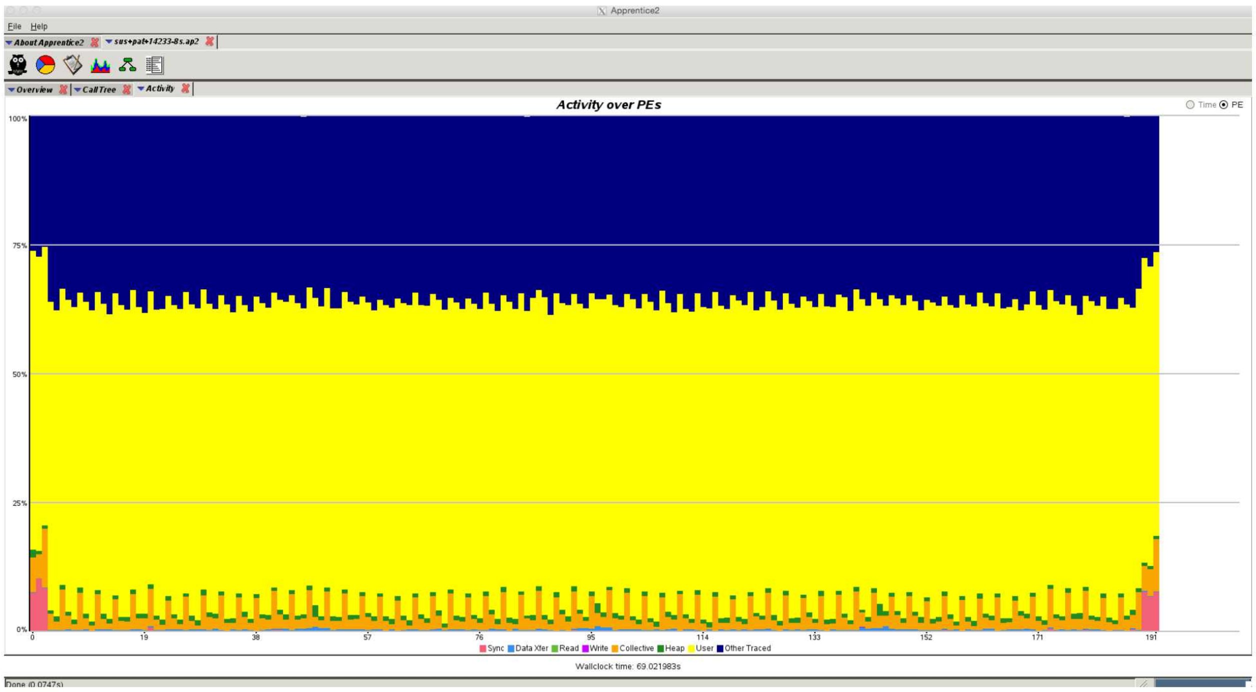


Figure 2.17: GUI of CrayPat, presenting the performance profile per process.

`pat_build` with many options to control the target of performance profiling including the preset events subject to tracing (mpi, blas/lapack, I/O, etc) and the time intervals for profiling. In addition to the command line options, CrayPAT offers APIs to manually delineate the events of interests. For reading profile data, CrayPAT provides CUI and GUI to meet different needs. The CUI offers numerous options to filter the data, generated in a tabulated formula with a very short turnaround time (see figure 2.16. The GUI, used via X11 client, enables intuitive understanding of the performance of parallel program execution, including bar chart for load balancing and a pie chart for dominant (time consuming) routines and events (See Figure 2.17). Among these visualization features, the communication tracing would be the most helpful for the users to understand how computation and communication are coordinated by AMT runtime systems. However, this capability is only enabled at the expense of large disk space and processing overhead. Note that our experiment with Uintah indicates that `pat_report` spent more than 10 minutes to process the tracing data files of a program execution for 10 seconds on 96 processes [4].

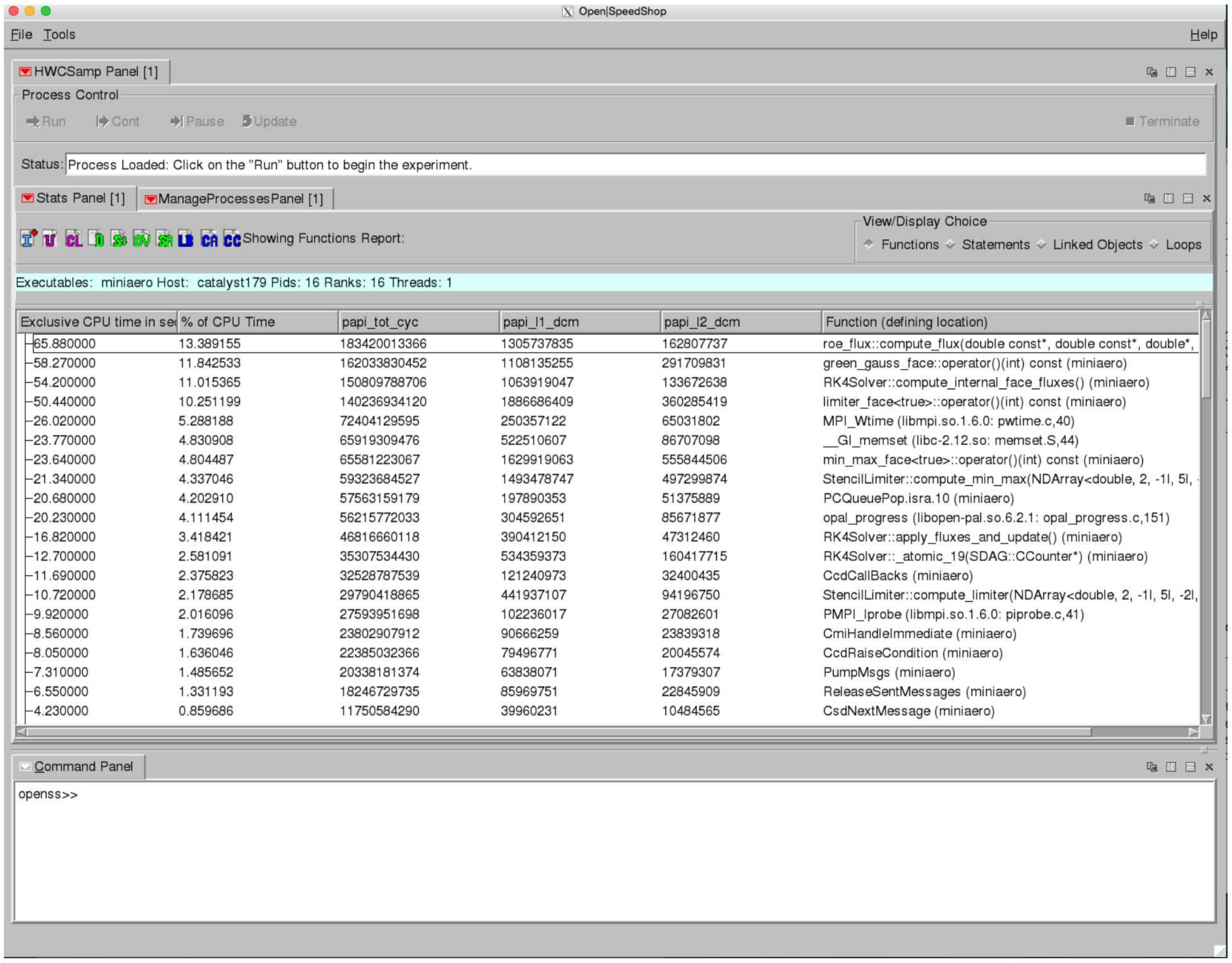
Open|SpeedShop Open|SpeedShop (OSS) [64] is a multi-institution effort to develop a suite of off-line performance analysis tools targeted at large scale HPC systems. OSS covers a broad range of performance profiling and analysis capabilities listed below:

- Program Counter Sampling
- Support for Callstack Analysis
- Hardware Performance Counter Sampling and Threshold based
- MPI Lightweight Profiling and Tracing
- I/O Lightweight Profiling and Tracing
- Floating Point Exception Analysis
- Memory Trace Analysis
- POSIX Thread Trace Analysis

The wide coverage of OSS makes it somewhat comparable to TAU and CrayPAT. Like the other two, OSS heavily leverages open source software, including PAPI [60] for performance counter and MRNet [68] for scalable management of the performance statistics data. While CrayPAT (and TAU) requires users to rebuild the program executable to

[4]Other communication tracing tools such as Jumpshot [69] and Vampir [70] are more efficient than CrayPAT

instrument performance profiling, OSS does not require any rebuilding to deploy the tool. OSS works as a job launch command, encapsulating the original job launch command lines subject to profiling. OSS is platform-independent, supporting generic PC clusters, ARM, IBM Power, IBM Blue Gene series and Cray platforms.


Open|SpeedShop
File Tools Help
HWCSamp Panel [1]
Process Control
Run Cont Pause Update Terminate

Status: Process Loaded: Click on the "Run" button to begin the experiment.

Stats Panel [1] ManageProcessesPanel [1]
Showing Functions Report:
View/Display Choice
Functions Statements Linked Objects Loops

Executables: miniaero Host: catalyst179 Pids: 16 Ranks: 16 Threads: 1

Exclusive CPU time in se	% of CPU Time	papi_tot_cyc	papi_l1_dcm	papi_l2_dcm	Function (defining location)
65.880000	13.389155	183420013366	1305737835	162807737	roe_flux::compute_flux(double const*, double const*, double*,
58.270000	11.842533	162033830452	1108135255	291709831	green_gauss_face::operator()(int) const (miniaero)
54.200000	11.015365	150809788706	1063919047	133672638	RK4Solver::compute_internal_face_fluxes() (miniaero)
50.440000	10.251199	140236934120	1886686409	360285419	limiter_face<true>::operator()(int) const (miniaero)
26.020000	5.288188	72404129595	250357122	65031802	MPI_Wtime (libmpi.so.1.6.0: pwtime.c,40)
23.770000	4.830908	65919309476	522510607	86707098	__GI_memset (libc-2.12.so: memset.S,44)
23.640000	4.804487	65581223067	1629919063	555844506	min_max_face<true>::operator()(int) const (miniaero)
21.340000	4.337046	59323684527	1493478747	497299874	StencilLimiter::compute_min_max(NDArray<double, 2, -1l, 5l,
20.680000	4.202910	57563159179	197890353	51375889	PCQueuePop.isra.10 (miniaero)
20.230000	4.111454	56215772033	304592651	85671877	opal_progress (libopen-pal.so.6.2.1: opal_progress.c,151)
16.820000	3.418421	46816660118	390412150	47312460	RK4Solver::apply_fluxes_and_update() (miniaero)
12.700000	2.581091	35307534430	534359373	160417715	RK4Solver::_atomic_19(SDAG::CCounter*) (miniaero)
11.690000	2.375823	32528787539	121240973	32400435	CcdCallBacks (miniaero)
10.720000	2.178685	29790418865	441937107	94196750	StencilLimiter::compute_limiter(NDArray<double, 2, -1l, 5l, -2l,
9.920000	2.016096	27593951698	102236017	27082601	PMPI_Iprobe (libmpi.so.1.6.0: piprobe.c,41)
8.560000	1.739696	23802907912	90666259	23839318	CmiHandleImmediate (miniaero)
8.050000	1.636046	22385032366	79496771	20045574	CcdRaiseCondition (miniaero)
7.310000	1.485652	20338181374	63838071	17379307	PumpMsgs (miniaero)
6.550000	1.331193	18246729735	85969751	22845909	ReleaseSentMessages (miniaero)
4.230000	0.859686	11750584290	39960231	10484565	CsdNextMessage (miniaero)

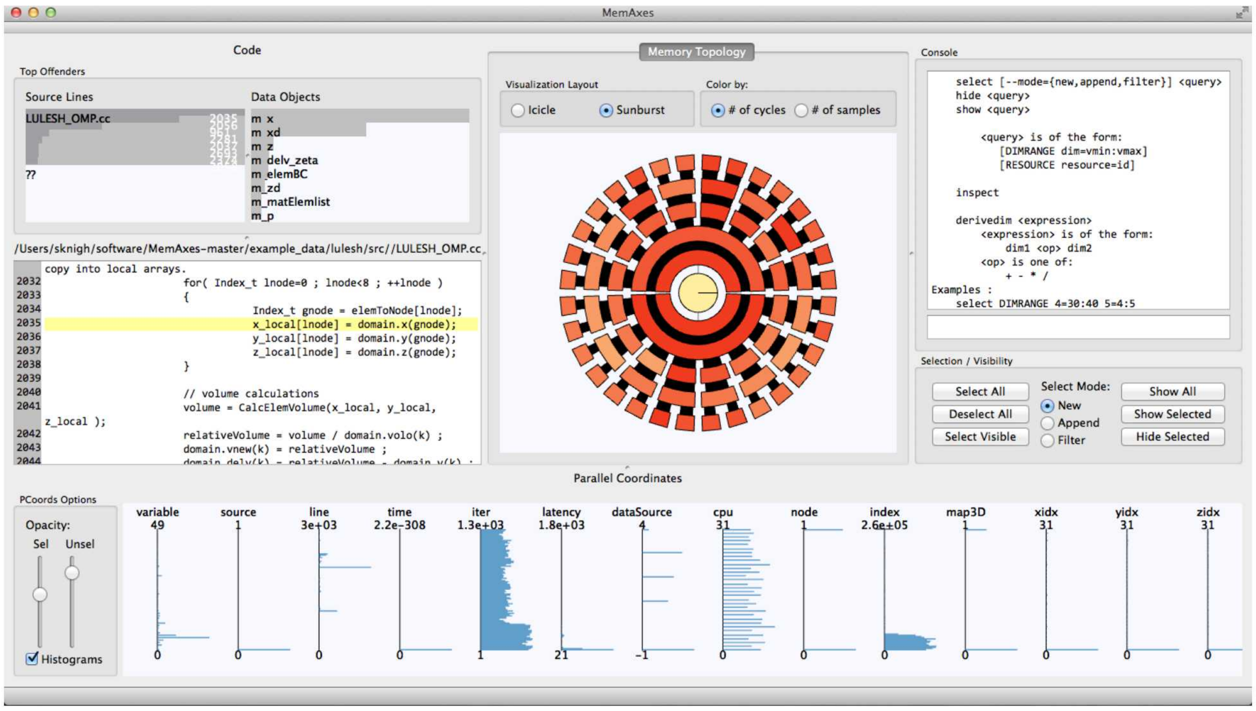
```
openss>>
```

Figure 2.18: GUI of Open|SpeedShop

We have tested OSS on two clusters: Lawrence Livermore National Laboratory (LLNL)'s Catalyst and Sandia National Laboratories (SNL)'s Chama. For all three AMT runtime systems of our interest, OSS is able to produce reports of the hardware counter information and execution time for tasks and threading underneath the runtime as shown in Figure 2.18. The visualization capability for charts and parallel performance tracing is not as expressive as CrayPAT (and TAU). However, the platform independence and quick instrumentation are the major strength of OSS.

The major weakness of OSS is its installation process, which is ideally as simple as a few configure script executions. However, we have observed that extensive knowledge of OSS itself, compilers and system-specific runtime libraries is required for successful installation. Although OSS provides an extensive installation guide with platform specific installation examples, the document is slightly outdated to meet all the subtleties in the software and environment for the target systems. This situation should improve with the currently ongoing integration into the Spack installer [71], which will be publicly available in the coming months.

Lawrence Livermore National Laboratory Performance Analysis Suite LLNL leads R&D activities in HPC programming tools to support debugging and performance optimization for future extreme scale HPC systems. Much of their performance tools research focuses on developing novel visualizations to assist the programmer in reasoning about their application's system resources utilization. Some of their recent efforts focus on automatic correlation



```
copy into local arrays.
            for( Index_t lnode=0 ; lnode<8 ; ++lnode )
            {
                    Index_t gnode = elemToNode[lnode];
                    x_local[lnode] = domain.x(gnode);
                    y_local[lnode] = domain.y(gnode);
                    z_local[lnode] = domain.z(gnode);
            }

            // volume calculations
            volume = CalcElemVolume(x_local, y_local,
z_local );
            relativeVolume = volume / domain.volo(k) ;
            domain.vnew(k) = relativeVolume ;
            domain.delv(k) = relativeVolume - domain.v(k) ;
```

```
    select [--mode={new,append,filter}] <query>
    hide <query>
    show <query>

        <query> is of the form:
           [DIMRANGE dim=vmin:vmax]
           [RESOURCE resource=id]

    inspect

    derivedim <expression>
        <expression> is of the form:
            dim1 <op> dim2
        <op> is one of:
            + - * /
Examples :
    select DIMRANGE 4=30:40 5=4:5
```

Figure 2.19: The circular diagram of MemAxes to indicate hotspots of memory access.

between performance and application program context, memory efficiency and optimization, and advanced timeline visualizations for trace data. MemAxes [72] is a visualization tool for on-node memory traffic, allowing the user to visualize a program's memory access patterns through a graphical user interface (GUI). It leverages Mitos [73], a memory tracing tool developed at LLNL, for collecting memory performance data. The circular diagram as shown in Figure 2.19 illustrates memory accesses on a multicore CPU node and can be used to help the user optimize application performance. For the AMT runtimes, it is anticipated that MemAxes can provide hints regarding optimal data and task granularity settings for an application via a few test runs on small node counts.

Ravel [5] provides a novel mapping of events onto a logical timeline, that facilitates detection and analysis of patterns in communication traces. Time information is reintroduced into the logical timelines using novel metrics such as lateness and is shown as a color in the trace. A sample screenshot of Ravel can be seen in Figure 2.20. Originally developed for MPI traces, Ravel has recently been extended for Charm++ to investigate its use for AMT approaches. This is ongoing research, however, and hence for now is not included in the work reported in this document.

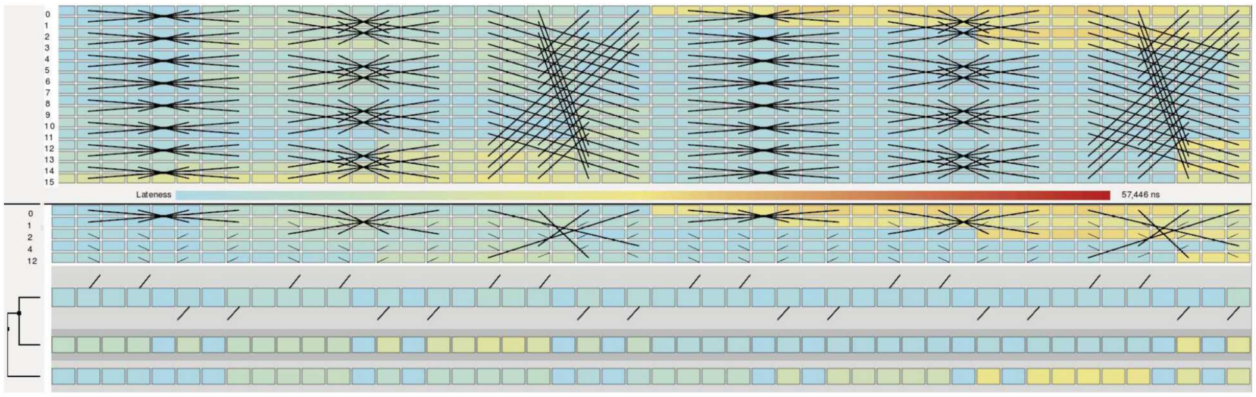


Figure 2.20: Logical timeline and clustered logical timeline views from Ravel [5].

Caliper [74] is an ongoing research effort aimed at correlating performance data with the program source for better holistic understanding of application performance. Conventionally, system profiling tools have provided a number of metrics, each of which is associated with a specific feature of HPC systems. Despite their usefulness, the deluge of multiple performance metrics makes it hard to extract useful information for performance optimization, leaving users to hypothesize the performance model for the given performance profile data and underlying program source. Caliper facilitates this process through (1) a flexible data model that can efficiently represent arbitrary performance-related data, and (2) a library that transparently combines performance metrics and program context information provided by source-code annotations and automatic measurement modules. Measurement modules (e.g., for PAPI and Mitos) and source-code annotations in different program components are independent of each other and can be combined in an arbitrary fashion. With this composite approach, it is easy to create powerful measurement solutions that facilitate the correlation of performance data across the software stack. Caliper supports multi-threaded applications in a programming model agnostic way. Data from different processes (e.g., of an MPI program) can be combined in a post-mortem step. More work is needed to integrate Caliper data into downstream data analytics stacks and visualization tools such as MemAxes.

2.7.2 AMT Runtime-Provided Tools

We note the vast majority of today's AMT runtime systems analysis tools are offline and therefore require post-mortem data aggregation and analysis. Across the tools provided by Charm++, Legion, and Uintah, we find two prevailing views into the execution of asynchronous task-based applications. The first view is a task execution timeline where tasks are represented in terms of their start time, duration, and location (i.e., the processing unit that executed the task's code). The second representation takes on the form of a DAG $G = (V, E)$ where $v \in V$ represent computational tasks and $e \in E$ represent dependencies between them. In the following we present an overview of each of the runtime system provided tools.

Charm++ Charm++ provides a suite of performance tools and a debugger, which compensates for its incompatibility with external programming and performance analysis tools. There are two major tools for Charm++, which are distributed separately. The first tool is Projections [75], which is a suite of performance analysis tools, providing very robust timeline analysis, load balance analysis, and memory allocation profiles for multiple processes. The second tool is the Charm++ debugger, which enables both parallel and sequential debugging with extensive GUI support.

Tracing is turned on by default in a Charm++ build. Only if one specifies the `--with-production` option, the `--enable-tracing` option needs to be specified. Only a relink (and not recompilation) is required to enable tracing in an application (-tracemode projections needs to be added to the link line). The probes for tracing are automatically instrumented by the specialized `charmc`, but it is possible for the users to manually insert the performance profiler function calls in the application program source to control the tracing and registering of events. All the APIs are comparable to those of proprietary tools like CrayPAT. The performance logs are text files or their gzipped versions depending on whether `libz` was found when building Charm++. We did not observe any significant performance degradation for data generation, indicating there is a built-in data aggregation capability.

Projections is distributed as source code (git repo) or in binary form: `http://charm.cs.uiuc.edu/software`. The Projections Java class object allows users to interactively steer the performance data on their local machines, with less delay than would be caused by remote access through an X11client. In our experience, the Projections response was almost instantaneous, whereas with X11client tools such as CrayPAT [63] and Open|SpeedShop (OSS) [64], a few seconds to minutes could be required to change the view. The most useful feature of Projections is the timeline view of program execution as shown in Figure 2.21, indicating idle time, method execution and runtime overhead for the given time in each process. In particular, the horizontal bars on each PE represents individual entry method executions (of individual chares) on each process. By hovering over the bar, one can view the chare it was executed for, the time the entry method executed for, and various other information. More importantly, by clicking those bars, one can view the message that lead to invocation of that entry method. Other views in projections (such as time profile, usage profile, histograms, etc) also provide information on execution time of entry methods and idle time on processes.

The major drawback of Projections is a lack of capability to visualize the trace with respect to **chares**. Charm++'s tracing framework does record individual chares and their current processor (PE) at all times, and hence tracks object migrations in that sense. However, load balancing (including migrations) is considered a runtime activity and most of the runtime is not instrumented so as not to clutter the application traces with runtime events. The current version of Projections groups multiple chares executed in a same process, and visualizes them as a single horizontal stacked bar per process. The image and the numbers provided by individual method (chare) invocations provide very useful information regarding load-balancing and idle time to help performance tuning, but it does not help the user infer how individual chares are coordinated and interact with one another. Recent work [76] presents a new framework to organize event traces of parallel programs written in Charm++, reorganizing trace data so that the user can more easily explore and analyze its logical structure.

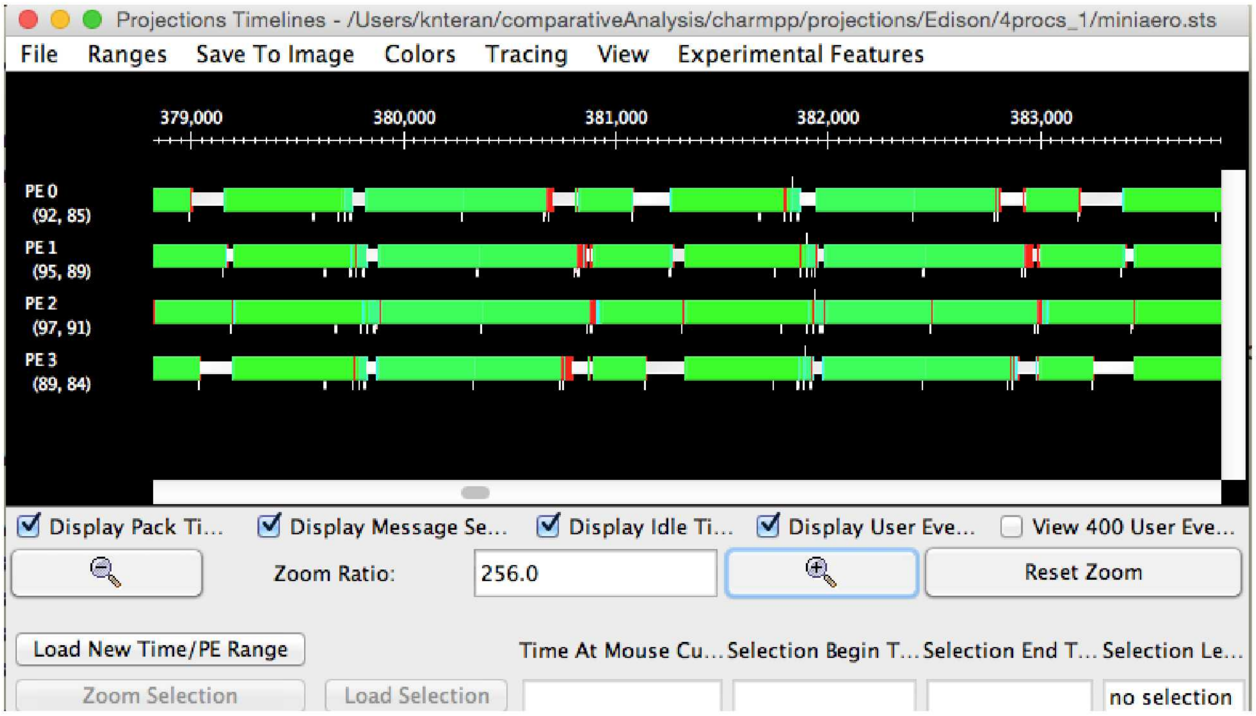


Figure 2.21: Timeline chart of 4 process execution presented by Projections.

We generally found the process of debugging code in Charm++ to be quite straightforward. It is hard for us to comment in great detail on the strengths and weaknesses of the Charm++ debugging process based on our experience with MiniAero because we did not use the provided Charm++ tools much in this process. Much of our initial difficulties arose because of confusion about the programming model (as is often the case when working with a new programming model or runtime system). After clearing up these initial issues, not much additional debugging was necessary. At the basic level, Charm++ allows the user to give the `++debug` flag to the `charmrun` command that launches Charm++ programs. This launches a `gdb` instance for each worker in a separate terminal window. However, our experience with Charm++ debugging tools only scrapes the surface of those available. Charm++ provides a Java-based graphical user interface, called Charm Debug, specifically designed for debugging Charm++ programs. Other nice debugging features, none of which we used, include message queue randomization to help detect race conditions, message interception and logging, and even dynamic code injection with Python.

Legion There are several tools and options for debugging, profiling, and tuning an application for high performance provided by Legion. The programming model itself provides information that the runtime and tools can leverage to facilitate debugging. Compiling the Legion application in debug mode enables many checks that can uncover problems with the application. These runtime checks include warning and/or error messages when the application breaks the assumptions of the runtime system. These include:

- Disjointness checks which verify that an index partitioning that claims to be disjoint actually is disjoint;
- Privilege checks which verify that all memory accesses abide by the privileges (Read-Only, Read-Write, and Write-Discard) stated in the task's region requirements; and

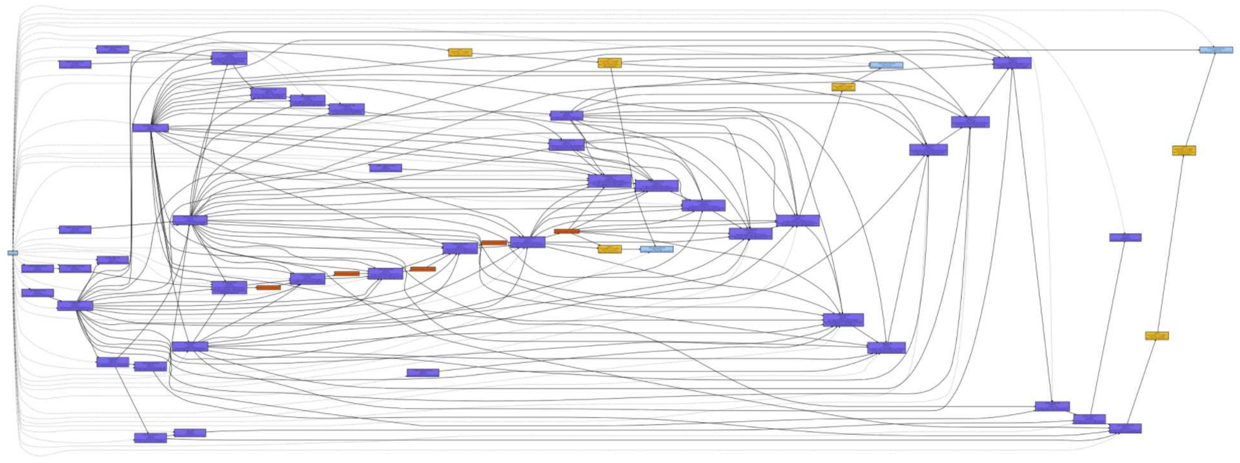


Figure 2.22: Legion Spy output showing event graph for single Runga-Kutta iteration for MiniAero

- Bounds checks which verify that all memory accesses fall within the logical region's bounds.

Legion also provides a very useful logging infrastructure which supports multiple levels and categories of logging output. The logger is used for both legion runtime debugging and application debugging and can be selectively compiled out in release mode to avoid any potential performance penalty.

Legion provides a graphical tool called Legion Spy which is a python post-processing debugging tool that can check correctness of both a Legion application and the Legion runtime. Legion Spy can perform several analyses of the output from an application run. The checks include logical dependence, physical dependence, data races, and data flow. Legion Spy also provides graphical output including event graphs and instance graphs. An example of an event graph is shown in Figure 2.22 and a zoomed in view of a portion of that graph is shown in Figure 2.23. The graph shows the operations and their dependencies. Each box represents an operation and an edge corresponds to explicit dependence between two operations. The operations are annotated with physical instances that are accessed and the access privilege and coherence mode for the fields. These graphs help detect sub-optimal decisions by the mapper and can be used to check logical and physical data flows.

Another graphical tool is the Legion Prof profiler. This also is a python post-processing tool that parses the output from an application run and creates a graph showing time lines of execution illustrating which tasks ran on which processors at which time. When the plots are rendered in a web browser, hovering the mouse over a task will show additional information about the task. Figure 2.24 shows output for a single MiniAero timestep. The four repetitive groups show fourth-order Runga-Kutta iterations. Legion Prof will also output runtime statistics including how long each processor was active, how many instances were created on each memory, how often a task was invoked, and how long it was running.

Very extensive diagnostics are available. Legion Prof tracks three kinds of low-level resources: processors, channels, and memories. For processors, it tracks all tasks run by the low-level runtime including both application tasks and runtime meta-tasks. For a channel (a directed pair of memories), it tracks all copies issued from the source memory to the destination memory. Finally, for individual memories, it tracks all instances allocated in that memory. Legion Prof can visualize all of these in the same plot, or selected subsets (e.g., just processors, or just memories, or processors and channels, etc.).

Legion Prof also tracks what caused something to happen. For all meta-tasks, channel copies, and instance creations, Legion Prof will report which application task caused that resource to be consumed, giving some insight into why certain things are occurring.

These tools were very useful and all of them were used during the development of the Legion implementation of MiniAero. The Legion Prof and Legion Spy output made it very clear that the default Legion Mapper was not correctly mapping the tasks. With some minor changes to the mapper behavior, the performance of MiniAero was vastly improved. The logging infrastructure was used extensively to verify that the correct values were being calculated in

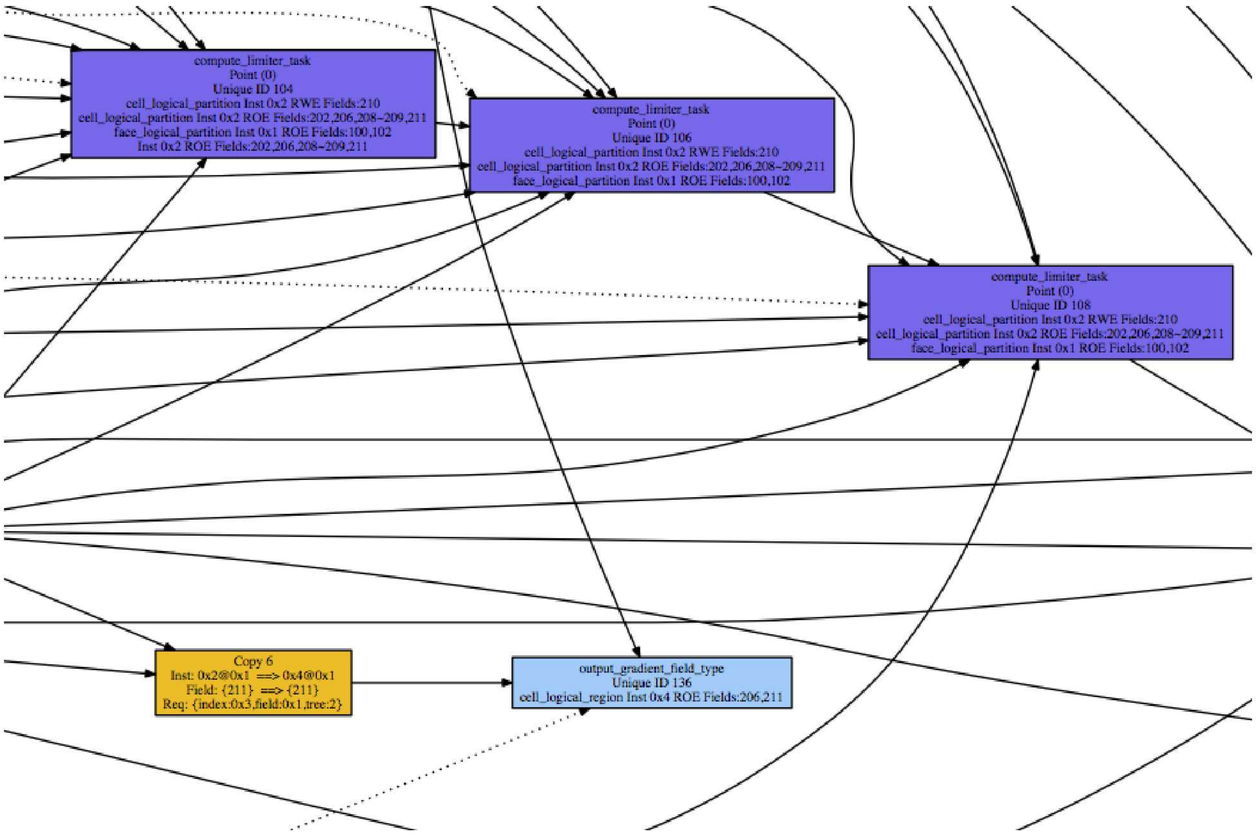

compute_limiter_task
Point (0)
Unique ID 104
cell_logical_partition Inst 0x2 RWE Fields:210
cell_logical_partition Inst 0x2 ROE Fields:202,206,208-209,211
face_logical_partition Inst 0x1 ROE Fields:100,102
Inst 0x2 ROE Fields:202,206,208-209,211
compute_limiter_task
Point (0)
Unique ID 106
cell_logical_partition Inst 0x2 RWE Fields:210
cell_logical_partition Inst 0x2 ROE Fields:202,206,208-209,211
face_logical_partition Inst 0x1 ROE Fields:100,102
compute_limiter_task
Point (0)
Unique ID 108
cell_logical_partition Inst 0x2 RWE Fields:210
cell_logical_partition Inst 0x2 ROE Fields:202,206,208-209,211
face_logical_partition Inst 0x1 ROE Fields:100,102
Copy 6
Inst: 0x2@0x1 ==> 0x4@0x1
Field: {211} ==> {211}
Req: {index:0x3,field:0x1,tree:2}
output_gradient_field_type
Unique ID 136
cell_logical_region Inst 0x4 ROE Fields:206,211

Figure 2.23: Legion Spy output showing event graph for single Runga-Kutta iteration for MiniAero. Zoomed view showing task information including partition and field properties.

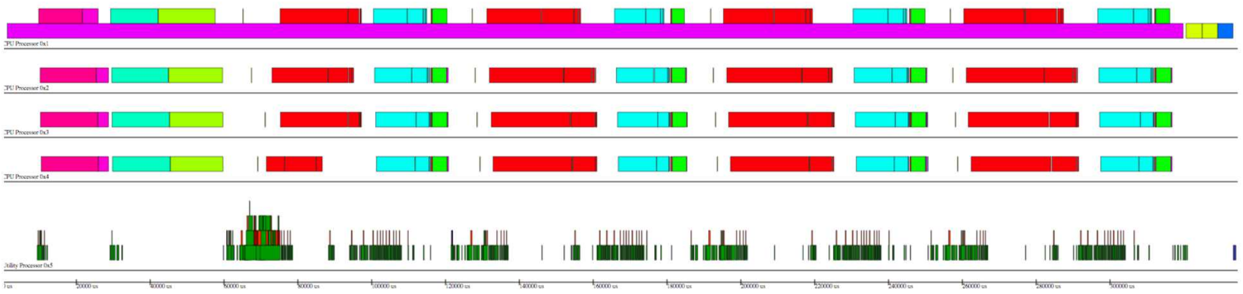
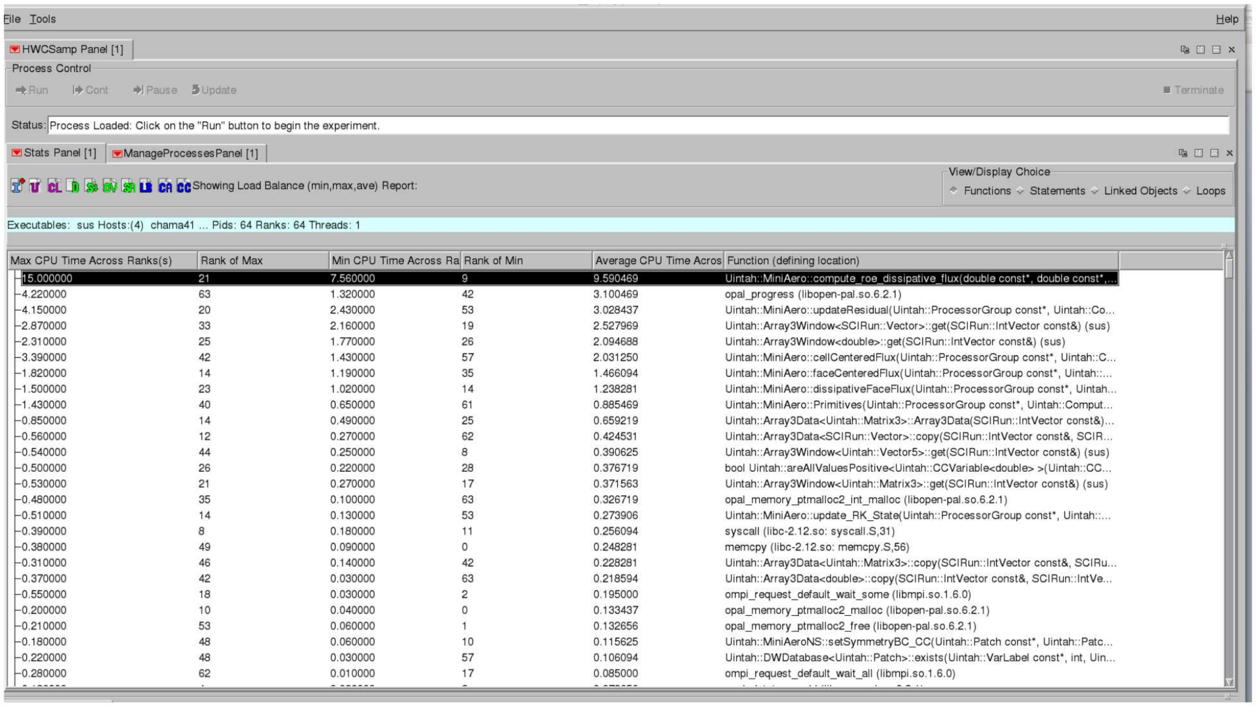


Figure 2.24: Legion Prof output showing timeline for a single MiniAero timestep on 4 processes. In “live” graph, hovering the mouse over a task shows additional information.

the various MiniAero tasks and that the correct data structures were being constructed.

The Legion developers have plans to improve the tools and actually made a few useful improvements over the course of this study which made the profiling tool much easier to use.



Status: Process Loaded: Click on the "Run" button to begin the experiment.

Showing Load Balance (min,max,ave) Report:

Executables: sus Hosts:(4) chama41 ... Pids: 64 Ranks: 64 Threads: 1

Max CPU Time Across Ranks(s)	Rank of Max	Min CPU Time Across Ra	Rank of Min	Average CPU Time Acros	Function (defining location)
15.000000	21	7.560000	9	9.590469	Uintah::MiniAero::compute_roe_dissipative_flux(double const*, double const*,...
4.220000	63	1.320000	42	3.100469	opal_progress (libopen-pal.so.6.2.1)
4.150000	20	2.430000	53	3.028437	Uintah::MiniAero::updateResidual(Uintah::ProcessorGroup const*, Uintah::Co...
2.870000	33	2.160000	19	2.527969	Uintah::Array3Window<SCIRun::Vector>::get(SCIRun::IntVector const&) (sus)
2.310000	25	1.770000	26	2.094688	Uintah::Array3Window<double>::get(SCIRun::IntVector const&) (sus)
3.390000	42	1.430000	57	2.031250	Uintah::MiniAero::cellCenteredFlux(Uintah::ProcessorGroup const*, Uintah::C...
1.820000	14	1.190000	35	1.466094	Uintah::MiniAero::faceCenteredFlux(Uintah::ProcessorGroup const*, Uintah::...
1.500000	23	1.020000	14	1.238281	Uintah::MiniAero::dissipativeFaceFlux(Uintah::ProcessorGroup const*, Uintah...
1.430000	40	0.650000	61	0.885469	Uintah::MiniAero::Primitives(Uintah::ProcessorGroup const*, Uintah::Comput...
0.850000	14	0.490000	25	0.659219	Uintah::Array3Data<Uintah::Matrix3>::Array3Data(SCIRun::IntVector const&)...
0.560000	12	0.270000	62	0.424531	Uintah::Array3Data<SCIRun::Vector>::copy(SCIRun::IntVector const&, SCIR...
0.540000	44	0.250000	8	0.390625	Uintah::Array3Window<Uintah::Vector5>::get(SCIRun::IntVector const&) (sus)
0.500000	26	0.220000	28	0.376719	bool Uintah::areAllValuesPositive<Uintah::CCVariable<double> >(Uintah::CC...
0.530000	21	0.270000	17	0.371563	Uintah::Array3Window<Uintah::Matrix3>::get(SCIRun::IntVector const&) (sus)
0.480000	35	0.100000	63	0.326719	opal_memory_ptmalloc2_int_malloc (libopen-pal.so.6.2.1)
0.510000	14	0.130000	53	0.273906	Uintah::MiniAero::update_RK_State(Uintah::ProcessorGroup const*, Uintah::...
0.390000	8	0.180000	11	0.256094	syscall (libc-2.12.so: syscall.S,31)
0.380000	49	0.090000	0	0.248281	memcpy (libc-2.12.so: memcpy.S,56)
0.310000	46	0.140000	42	0.228281	Uintah::Array3Data<Uintah::Matrix3>::copy(SCIRun::IntVector const&, SCIRu...
0.370000	42	0.030000	63	0.218594	Uintah::Array3Data<double>::copy(SCIRun::IntVector const&, SCIRun::IntVe...
0.550000	18	0.030000	2	0.195000	ompi_request_default_wait_some (libmpi.so.1.6.0)
0.200000	10	0.040000	0	0.133437	opal_memory_ptmalloc2_malloc (libopen-pal.so.6.2.1)
0.210000	53	0.060000	1	0.132656	opal_memory_ptmalloc2_free (libopen-pal.so.6.2.1)
0.180000	48	0.060000	10	0.115625	Uintah::MiniAeroNS::setSymmetryBC_CC(Uintah::Patch const*, Uintah::Patc...
0.220000	48	0.030000	57	0.106094	Uintah::DWDatabase<Uintah::Patch>::exists(Uintah::VarLabel const*, int, Uin...
0.280000	62	0.010000	17	0.085000	ompi_request_default_wait_all (libmpi.so.1.6.0)

Figure 2.25: Performance report of Uintah from Open|SpeedShop.

Uintah The tools provided by Uintah focus on analysis of simulation data, which helps the users to check the correctness of the simulation as well as capturing insights from the simulated data. Serving as an integrity checking mechanism, Uintah has a tool called `puda` for browsing the contents of the Uintah Data Archive (.uda) files, which represent the data registered in the data warehouse. Uintah support of performance analysis is minimal; it provides a build option to enable PAPI to report the performance counter numbers per time step, and runtime options to report the timing data for individual tasks. Uintah also provides an experimental implementation to enable `gperftools` [77] to automate the instrumentation of performance measurement [5], but we found that it is not supporting the latest versions of these tools. Despite the lack of internal and embedded tool support, Uintah applications are compatible with many tools designed for MPI programming. As discussed above, both CrayPAT and OSS can measure the performance of Uintah and capture the data associated with the major tasks and MPI calls (see Figure 2.25 for OSS). Tracing communication patterns is possible with CrayPAT as presented in Figure 2.26, but it requires a large amount of disk space and long turnaround time to visualize the data, even for short program execution as mentioned above.

For debugging, the whole Uintah runtime can be seen as a big C++ class, which again makes it amenable to existing debugging tools, both in sequential and parallel settings such as `gdb` and TotalView [79].

Uintah's dependence on MPI allows effective reuse of existing tools, which may attract many application developers. However, there is no tool that fully leverages the task and data abstractions of Uintah for quantifying the performance at the task-graph level. For example, one could tune the granularity of tasks in addition to the granularity of the patch size (over-decomposition) in an attempt to derive more parallelism. Without the tool support, this performance tuning process would devolve into a repetition of code modification and experimentation. Therefore, high-level performance analysis tools would be very beneficial, preventing the users from such repetitions with some performance guidance.

[5]In the past, Uintah team also attempted to integrate TAU [78] by manual instrumentation of TAU API calls in user's source.

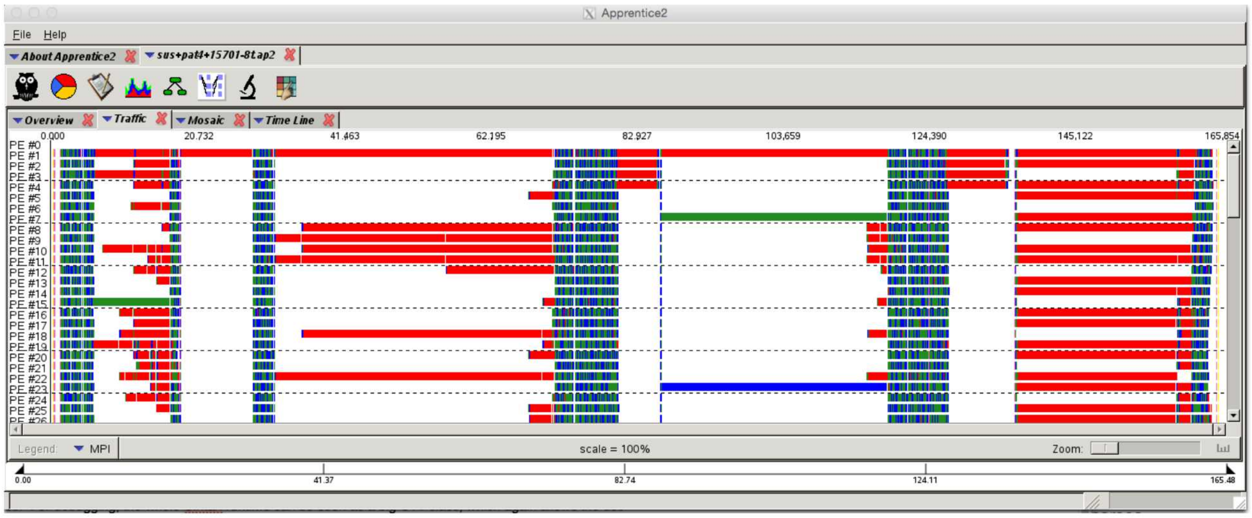


Figure 2.26: Timeline chart of MiniAero-Uintah using CrayPAT.

AMT Research Challenges

The overarching AMT tools research challenges stem from the need to develop best practices and eventual standards. In our study, we noted that the general HPC tools we surveyed are very useful when developing applications for AMT runtime systems. However, because they are designed for MPI+X, they are lacking capabilities in (i) capturing AMT communication patterns that employ non-traditional use of the transport layer and (ii) workflow extraction from task-parallel program execution. These issues will soon be addressed by the tool community in the context of MPI+X, resulting from the extensive support of task-parallelism in the new OpenMP-4.0 standard and its runtime API (OMPT) intended for the access to the internal OpenMP runtime information from external tools [80]. The new tools will enable the users to analyze the performance and dependencies of *OpenMP tasks* in node level. Likewise, this methodology can benefit the AMT runtime systems in our study, but extra efforts would be necessary to understand how to express tasks, data and workflow of each specific AMT runtime system in order to integrate with these tools. The emergence of the next generation performance tools at LLNL may fill the semantic gap between general-purpose tools and the execution patterns of AMT runtime systems. Albeit, these tools are still in an early research stage, and need further investigation to assess their feasibility.

The deficiencies in general purpose tools from an AMT perspective is compensated by a variety of runtime system-specific tools. Charm++ and Legion provide a good suite of performance tools and debugging assistance. In particular, the Charm++ team has published many papers to address the scalability of performance measurement and debugging assistance [75, 76, 81–84], and the results have been integrated into their tools. Legion has its own performance profiling layer in its modular runtime design to prepare for and facilitate runtime specific tool support. Among the three runtime systems, Uintah is the most compatible with the existing tools owing to the use of MPI as its transport layer. However, it is not convincing that the existing tools would suffice in all settings, because of Uintah's departure from the traditional SPMD programming model. In particular, many of their schedulers introduce non-deterministic task execution in addition to heterogeneous computing.

We note the development of runtime-specific tools is an important precursor to general purpose tool support. However, the tools and AMT runtime communities need to work together to address the following issues as we move towards best practices and eventual standards with regards to AMT runtimes and tools:

- We need a common set of tool abstractions and interfaces for AMT runtimes. This includes the necessary support for instrumentation, mapping of abstractions to system information and *vice versa*, and trace collection and trace control.
- Currently the community lacks a good statistical analysis environment for performance data. Building upon the abstractions and interfaces described in the first bullet, we need to develop automated and user-friendly downstream data analysis and aggregation tools to help the user glean insight into their programs' behavior.

- We need to further investigate the deficiencies that exist with today's tool chains that prevent their effective use in the analysis of AMT runtime systems and the applications they support.
- Beyond aggregated statistics we believe understanding and optimizing the performance of AMT runtime systems will rely heavily on processing massive task graphs to find critical sections, understand the impacts of non-deterministic scheduling, and highlight potential improvements. We need to develop new tools to handle system wide task graphs ideally using a combination of automatic processing and human assisted exploration.
- There will need to be research to assess how analysis utilities can be designed to maximize code and tool reuse while providing an adequate level of model/runtime/domain awareness.
- New views into the execution of applications that interoperate with external runtimes (e.g., MPI) need to be developed.

Chapter 3

Performance

3.1 Approach to Performance Analysis

As we assess the performance of each runtime, we seek to answer the following question:

> *How performant is this runtime for ASC/ATDM workloads on current platforms and how well suited is this runtime to address exascale challenges?*

As was discussed in Chapter 1, MiniAero is essentially a load-balanced algorithm, with static and regular data requirements and a relatively narrow task graph. However, it is still a very interesting use case from a performance analysis perspective. As was indicated in Figure 1.3, there are basically four regimes that can be considered: the cross product of [static workloads, dynamic workloads] and [static (homogeneous) machines, dynamic (heterogeneous) machines]. In this chapter we present the results of experiments for the two of these regimes that can be explored using MiniAero. First we explore whether or not AMT runtimes perform comparably to the baseline MPI implementation on machines with homogeneous performance. We next examine whether the runtime systems can mitigate performance heterogeneity in the machine through a set of artificially induced load imbalance experiments. We note that in this report we do not present results on any heterogeneous machines that include GPUs, as we did not have the time to develop the additional kernels for each runtime. We follow our empirical studies by examining each runtime's ability, from a performance perspective, to address some of the underlying challenges expected at exascale, namely fault tolerance, and complex workflows. We conclude this chapter with a comparative analysis based on survey results, analogous to that performed in Chapter 2.5, this time focused on performance issues.

3.1.1 Machines

Cielo Cielo is a Cray XE6 machine consisting of $8,944$ compute nodes, each comprising two 2.4 GHz AMD eight core Opteron Magny-Cours processors. Each Magny-Cours processor is divided into two memory regions, called NUMA nodes, each consisting of four processor cores and 8 GBytes of DDR3 1333 MHz memory. Each core has a dedicated 64 kByte L1 data cache, a 64 kByte L1 instruction cache, and a 512 kByte L2 data cache, and the cores within a NUMA node share a 6 MByte L3 cache (of which 5 MBytes are user available). Figure 3.1 shows two compute nodes connected to the Gemini network.

Each compute node runs Cray Linux Environment (CLE), a light weight propriety Linux operating system. All compute nodes are connected by Cray's Gemini network with $16 \times 12 \times 24$ (xyz) 3D torus topology, achieving $6.57 \times 4.38 \times 4.38$ TB peak aggregate bandwidth in each direction. The entire machine provides more than 1.374 petaflops (PF) of peak performance, and 268 terabytes (TB) of memory. Cielo is one of the large capability machines funded by the National Nuclear Security Administration (NNSA) for the ASC Program. It is operated by the Advanced Computing at Extreme Scale (ACES) program, a Los Alamos National Laboratory (LANL) and SNL Partnership. In the 2015–2016 timeframe, ACES will install Trinity, a next generation computing machine, as a replacement for Cielo.

Shepard Shepard is one the testbeds within Sandia's Heterogeneous Advanced Architecture Platform (HAAP) machine. Shepard is a 36 node cluster with Dual Intel Xeon Haswell E5-2698 v3 processors at 2.30 GHz, with 16 cores each. There are 2 NUMA regions per node with 64 GB of memory each. The interconnect is a Mellanox FDR Infiniband and the operating system is Red Hat 6.5. The processors are comparable to Trinity's first delivery and the

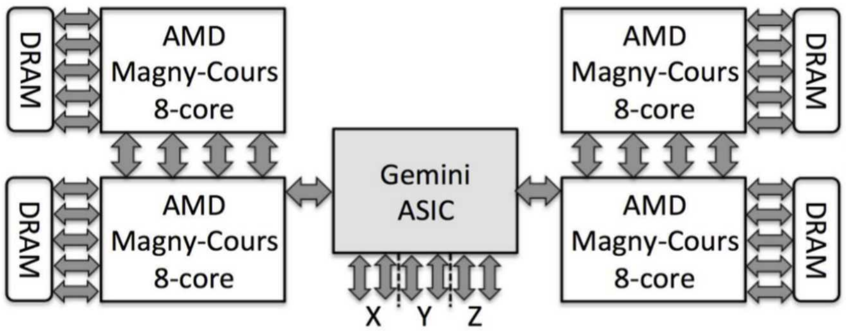


Figure 3.1: Cielo Compute Node Architecture (Two Nodes)

machine has a custom-designed power monitoring capability using PowerInsight V2 [85].

3.1.2 MiniAero Analysis Model

MiniAero is modeling a viscous flow over a flat plate. The computational domain is rectangular with the following boundary conditions applied to each of the six faces:

Left	Inflow	Right	Extrapolate
Bottom	No Slip	Top	Extrapolate
Back	Tangent	Front	Tangent

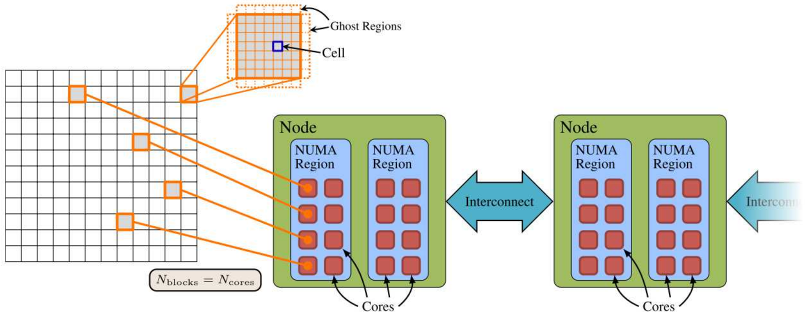
The physics of the problem uses 2$^{\mathrm{nd}}$-order spatial integration with viscous flux.

Mesh The number of cells in each of the three coordinate directions is specified as is the number of "blocks" or "patches" in which to group or decompose the cells in each direction. Typically, one or more of the blocks is a computational unit. For example, in the MPI implementation, each processor rank "owns" the cells and faces in a block. When overdecomposition is used, each block is further subdivided.

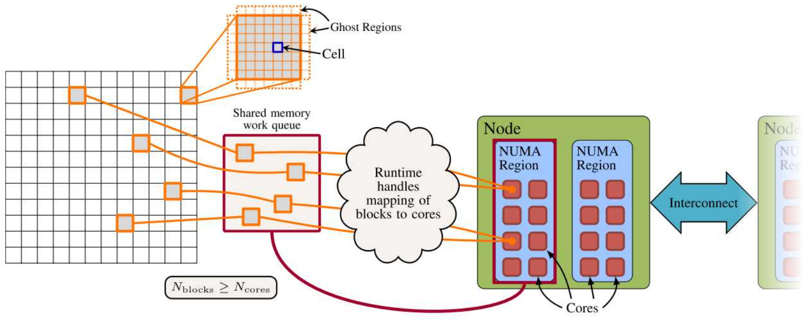
As an example, if the mesh size is specified as $128 \times 32 \times 512$ with a processor decomposition of $4 \times 1 \times 16$, then there would be $2,097,152$ cells and 64 blocks. Each block would consist of a cube with 32 cells on a side for a total of $32,768$ cells per block. For an overdecomposition of 2, this block would be subdivided into 8 subblocks ($2 \times 2 \times 2$) with 16 cells on a side.

3.1.3 Comparing Runtime System Resource Mapping

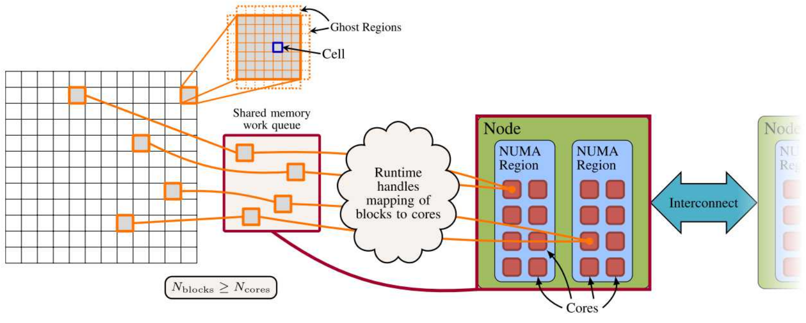
Figure 3.2 illustrates roughly how the different runtimes map data to cores. The mesh is decomposed into blocks that contain a subset of the mesh cells. In the MPI allocation shown in Figure 3.2a, the mesh is decomposed into the same number of blocks as there are cores, with a single block being mapped per core. Each of the other runtimes support overdecomposition, where the number of blocks is greater than the number of cores. Figure 3.2b shows the Charm++ SMP mapping, where the runtime handles mapping of blocks to cores. The overdecomposed blocks (`chares`) are mapped to cores, and those within a NUMA region belong to a shared memory work queue; currently, though, the runtime creates a fixed mapping of chares within this work queue to cores and those bindings persist until a chare is migrated during synchronous load balancing. The Uintah runtime, shown in Figure 3.2c is similar to Charm++ SMP; overdecomposition of blocks (now `patches`) is supported, and those within a node belong to a shared memory work queue in the threaded Uintah scheduler. Figure 3.2d illustrates the mapping for Legion. In the Legion runtime, the mesh fields belong to a logical region which is partitioned (again overdecomposition is supported). Using the Mapping interface, the runtime has tremendous flexibility in mapping data to resources.



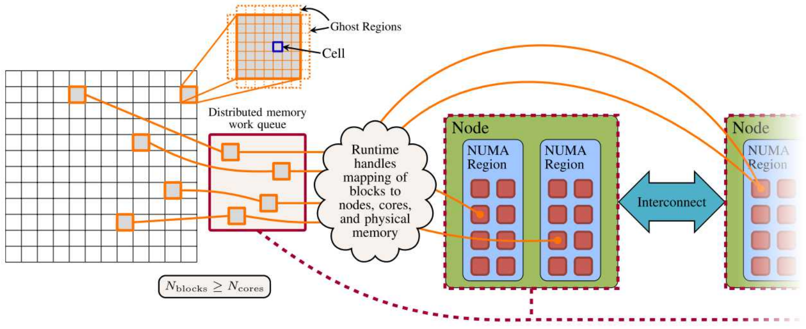
(a) MPI



(b) Charm++ SMP



(c) Uintah Threaded


Ghost Regions
Cell
Distributed memory work queue
Runtime handles mapping of blocks to nodes, cores, and physical memory
Node
NUMA Region
NUMA Region
Interconnect
Node
NUMA Region
$N_{\text{blocks}} \geq N_{\text{cores}}$

(d) Legion

Figure 3.2: Runtime system resource mapping

3.1.4 Caveats

MPI Baseline Implementation It is worth noting that the baseline MPI implementation of MiniAero on the Mantevo site was found to have performance issues. A major contributor to these problems was staff turnover. Another source of these issues was the relatively naïve use of the Kokkos library, leading to poor compute intensity and a large number of cycles per instruction. In addition, the baseline application had not been tested at the scales we used in our experiments, and when run on thousands of nodes, new bugs appeared. An unscalable implementation of the mesh setup as well as 32-bit integer overflow issues led to the need for *ad hoc* debugging during dedicated testing time.

In light of the early issues with the use of Kokkos in the baseline implementation, combined with the difficulty of leveraging Kokkos uniformly within the runtimes, our team decided early in the testing process to run our scaling tests with Kokkos in threaded mode, but with only one thread — thus emulating essentially a flat MPI baseline.

The interpretation of the results shown here for our MPI runs must be done carefully. The purpose of this exercise is not to present the fastest possible MPI implementation of MiniAero and compare that to various runtime implementations. Just about everything done by the runtimes can be implemented in MPI, given enough effort. Indeed, both Uintah and Charm++ can be compiled to use MPI as a transport layer. The implementation used in this study for the baseline (and for all runtimes, for that matter) is meant to represent the use of the *typical, supported programming model* of MPI to implement MiniAero. We recognize that the definition of "typical" here can vary wildly, and these data should be interpreted with that in mind.

Legion As was mentioned in Section 2.3.3, there are some current issues with the implementation of the Legion runtime that its developers are in the midst of addressing. In particular, they are currently making the changes to the runtime necessary to support the automatic SPMD-ification of a Legion application developer's top-level task. As they are making these changes, there is bug that they are resolving in the runtime that causes serialization issues between nodes. At the time that we were doing our scaling studies, neither the automatic SPMD-ification nor the runtime system bug were resolved, and so we will not present results in this chapter for MiniAero Legion. Instead, we include results for a Regent implementation of MiniAero, S3D [48], and PENNANT [86] in Section 3.2.

Uintah The Uintah implementation of MiniAero uses the internal uniform Cartesian structured grid format, while the other implementations use unstructured grids. Although the performance numbers here do *NOT* include the time to set up the mesh, the Uintah mesh storage results in regular and predictable memory access patterns which allow for data access optimizations and efficiencies which are not possible in the mesh storage required to store a fully unstructured mesh representation. The data we present in this chapter should be interpreted with that in mind.

MPI, Charm++, and Legion The three MiniAero implementations in MPI, Charm++, and Legion were unable to handle meshes with 2^{31} or more cells in our performance comparison runs on Cielo due to various issues. In the MPI and Charm++ implementations, the underlying runtimes and the application code have no problem handling meshes of this size, but the applications crashed during execution due to sloppiness in the original implementation (using integer widths too small to represent global indices). The issue was quickly remedied in both the versions of the code; however another problem with a non-scalable mesh setup implementation in the MPI version prevented us from running larger tests within the allotted time frame. The latter issue has also been fixed in the MPI version. Both issues arose from a fundamental miscommunication between our programming models team and our applications collaborators on the purpose and scalability requirements of mini-apps in general, which we are working to resolve.

The Legion runtime currently has a (known) limitation that the size of an IndexSpace must be less than 2^{31}. This limitation is currently being addressed and should be eliminated in the near future. The MiniAero mesh generation implementation will also need a few modifications to support the large meshes.

Because of these limitations, we limited the weak scaling runs to use a maximum 262K cells/node which would give a mesh size of 2^{31} cells when run on 8192 nodes which is the largest machine we used; the strong scaling runs used a maximum of 1.07 billion cells (2^{30} cells). Ideally, we would have liked to use much larger cell counts in both types of scaling, but the smaller mesh sizes did allow us to highlight runtime overheads much more prominently, which could be quite relevant in applications with less compute intensity. The Uintah implementation did not have these limitations, though it should be noted that it was most negatively affected by this issue because the Uintah runtime has a strongly requirements-driven design process, and their users typically perform runs with meshes having a much larger number of cells per node than those in our performance tests.

3.2 Performance Analysis on Homogeneous Machines

Here we explore whether or not the various runtimes perform comparably to the baseline MPI implementation of MiniAero on machines with relatively homogeneous performance. MiniAero is a load-balanced algorithm with regular data requirements and a narrow task graph. Outside of overdecomposition, there is very little task or pipeline parallelism to exploit. In this set of experiments we therefore focus on "baseline" AMT results, demonstrating whether in the load-balanced SPMD case, performance is still comparable to MPI. Experiments are shown for both Cielo and Shepard.

3.2.1 Weak and Strong Scaling on Cielo

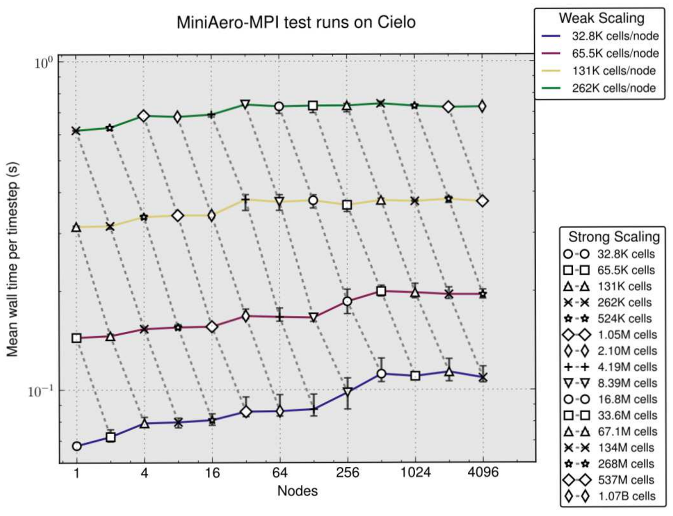
We performed a set of weak and strong scaling studies on Cielo, see Figure 3.3. In each case, the number of nodes was varied across powers of two, with problem sizes held constant (strong scaling, dotted lines) and varied to maintain a constant number of cells per node (weak scaling, solid lines). In all cases, an attempt was made to hold other parameters — such as compilation flags, runtime arguments, overdecomposition — constant at their respective optimal values for each of the implementations. Optimal values were determined based on a suite of smaller-scale experiments on Cielo. No load balancing was used in the Charm++ and Uintah runs. The Charm++ runs had an overdecomposition level of 2, whereas the Uintah did not have any overdecomposition. Overall, we see that performance is very comparable with the MPI baseline. Indeed, the Charm++ implementation demonstrates slightly better scaling behavior, particular at higher numbers of processing elements.

In Figure 3.3c, we show results for the Uintah dynamic MPI scheduler, where we see a depreciation in weak scaling. The Uintah team helped us to investigate this by repeating similar scale runs on Titan [87], which indicated an increase in local communication time at large node counts (> 500). This exposed a possibility of improving the implementation of tasks that require ghost cell communication. The Uintah team also pointed out that their performance and scalability efforts have always focused on computational workloads with average timesteps sizes on the order of seconds as opposed to sub-second ranges. In this realm, Uintah's runtime environment does well with good strong and weak scalability. This aspect stands out clearly in the scaling results from the Titan runs, shown in Figure 3.4 where, for the three problem sizes studied, the depreciation in strong scaling is observed once the average timesteps sizes go below the 1 second mark. These results suggest that the task granularity, as a consequence of the physics inherent in MiniAero and the problem sizes chosen, is too small for Uintah to sustain performance beyond a certain scale, something that was always a possibility when working with a mini app.

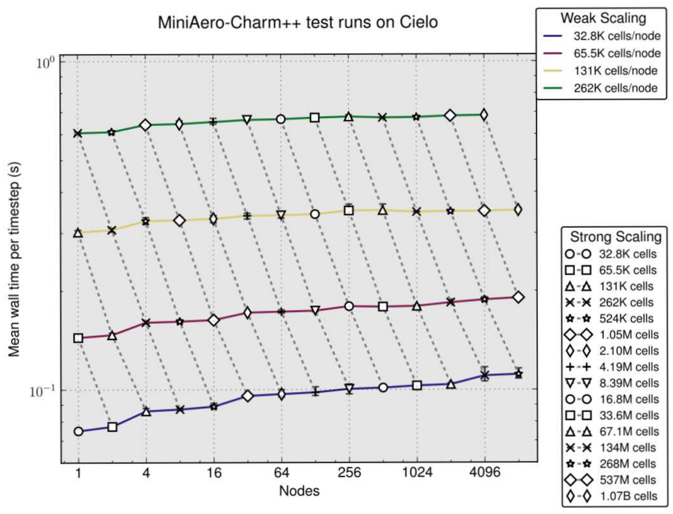
We also generated results for the threaded scheduler on Cielo, however there we ran into additional performance issues. For each instance of the threaded scheduler, there are $n-1$ worker threads created with `threadID` $1 \ldots n-1$. The main thread of execution is held by `threadID` 0 and CPU affinity is set based on `threadID` (e.g. `threadID` 1 is pinned to core 1, etc.). When we ran four processes, each with four threads (four instances of the threaded scheduler), cores $0-3$ were effectively quadruple booked while other cores remained completely idle. The Uintah team is doing work in a branch now to handle this more intelligently and they have seen performance improvements on an IBM Blue Gene/Q when running 2 processes per node, each with 16 threads, as these CPUs are dual issue, 4-way multi-threaded.

3.2.2 Time to Solution vs CPU Frequency for Varying Problem Sizes

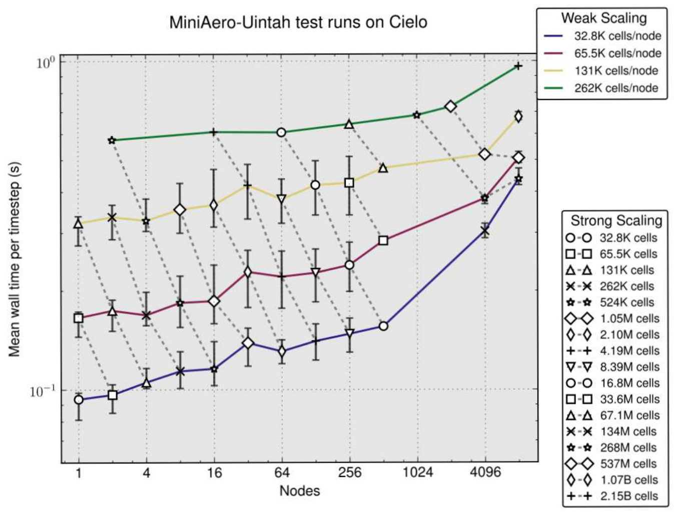
In Figure 3.5 we plot the results of an experiment where, for three problem sizes, we examined the wall time in seconds vs. frequency (on the left) and cells per second vs. frequency (on the right). These experiments were performed using 16 nodes on Shepard, and all nodes used are set to the same frequency using the `cpu_freq` command, resulting in homogeneous machine performance for each run. The three problem sizes include 8K points per node, 0.5M points per node, 4M points per node. We show two overdecomposition levels (4 and 8), and run the MiniAero implementations for 15 timesteps. Frequencies range from 1200 to 2300 MHz, increases of 100 MHz. We used the Charm++ `SMP` scheduler and the `RefineLB` load balancer, run at each timestep. For Uintah, we used the `MPI` scheduler and the



(a)



(b)



(c)

Figure 3.3: Weak and strong scaling results on Cielo for MiniAero-MPI, for MiniAero-Charm++, and for MiniAero-Uintah.

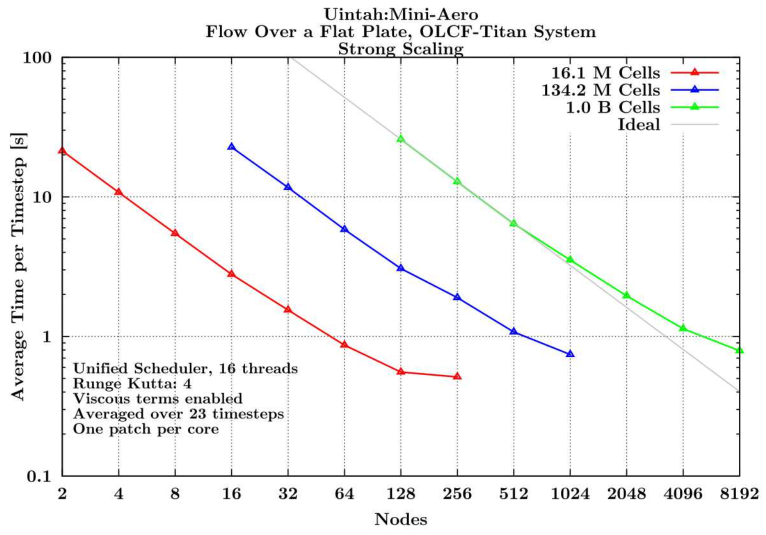


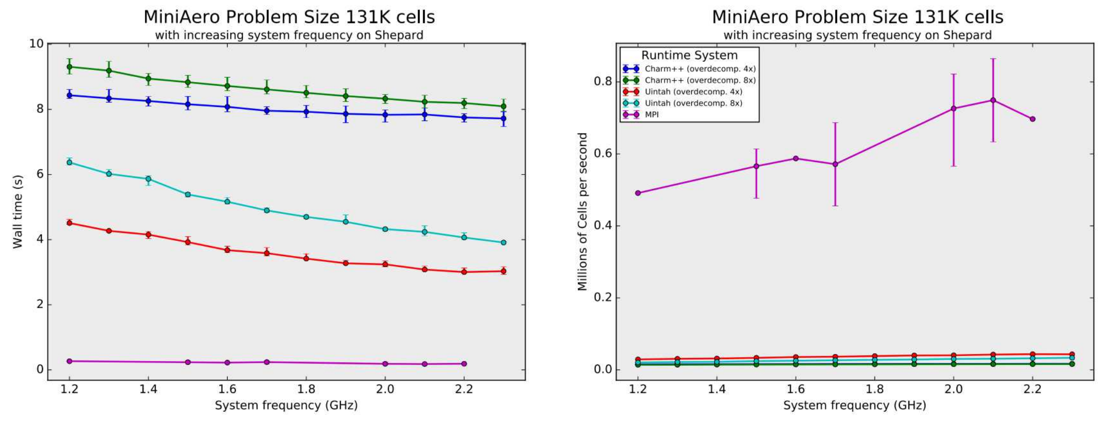
Figure 3.4: Strong scaling results on Titan for MiniAero-Uintah for three problem sizes.

`MostMessages-ModelLS` load balancer at each timestep. We note that although the load balancers were enabled in these experiments, aside from the overhead of running them, they served little effect as both the machine and the application were very static in nature. We also note that the largest problem size our MPI baseline runs on Shepard failed due to its memory issues in the mesh set up phase, as we performed these experiments prior to resolving some of the memory issues mentioned in Section 3.1.4. However, we plan to rerun the MPI baseline experiments for the large problem size now that this issue has been addressed. In these experiments we find that the AMT runtimes perform comparably to the MPI implementation. In particular MPI outperforms with small problem sizes per node. There is a cross-over point however where at larger problem sizes the AMTs outperform the MPI implementation.

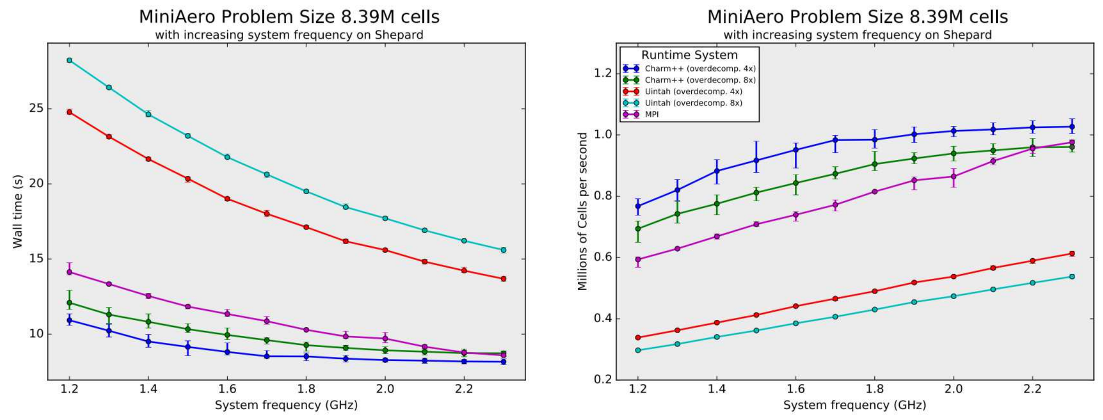
3.2.3 Runtime-Provided Scaling Examples

Charm++ Charm++ has been used to develop a variety of scientific applications. Several of these applications, such as NAMD [13], OpenAtom [88], ChaNGa [14], and Episimdemics [89], scale to hundreds of thousands of cores and are routinely run at that scale by scientists on present day supercomputers. NAMD won the Gordon Bell award in 2002, while Charm++ itself won the HPC Challenge Class 2 award in 2011. Here, we focus on ChaNGa, a complex application that exercises several features of the Charm++ runtime system to obtain scalable performance despite its irregularity and dynamic nature. ChaNGa is a parallel n-body+SPH cosmological code widely used for the simulation of astrophysical systems. Cosmology research based on ChaNGa includes studying the structure and formation of galaxies using a higher resolution simulation, which results in load imbalance due to highly non-uniform distribution of particles in the simulated systems. A wide variation in mass densities can also result in particles having dynamical times that vary by a large factor. Hierarchical time-stepping schemes address this by using different time-scales for each particle [class]; this reduces the operation count significantly but creates load balancing challenges, as load varies continuously across phases of computation. ChaNGa relies on key features of Charm++, such as over-decomposition, asynchronous message driven execution, prioritized messages and dynamic load balancing, to handle all these variations. Additionally, it uses automatic checkpoint restart capabilities to restart a simulation from a checkpoint. ChaNGa demonstrates 93% parallel efficiency for strong scaling simulations of 12 and 24 billion particle systems on 512K cores of Blue Waters as shown in Figure 3.6a. For a 2 billion particle clustered dataset, the single-stepping run has a high parallel efficiency of 80%. Despite its lower parallel efficiency, the multi-stepping simulation is 2 to 3 times faster than single-stepping (Figure 3.6b).

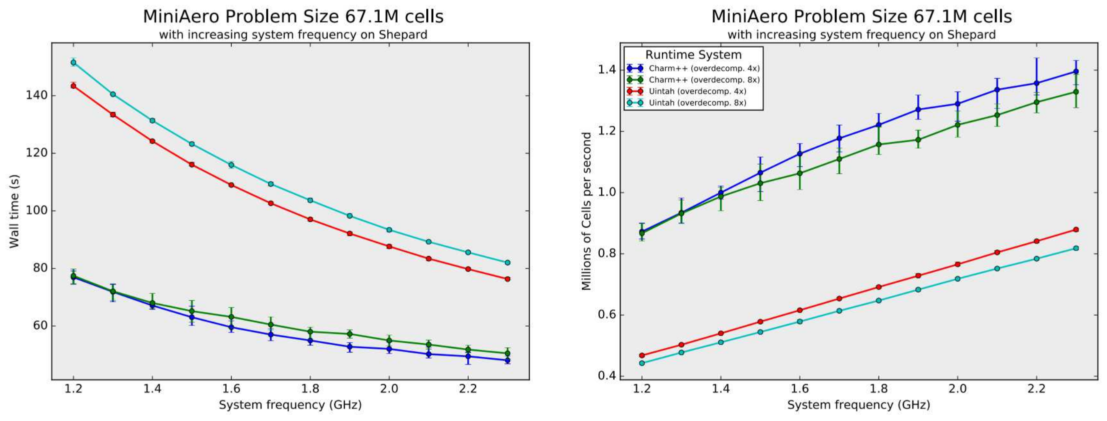
Legion An example of the benefits of the manual SPMD transformation can be seen in the Legion port of S3D [48]. S3D simulates combustion on a large regular grid with ample data parallelism, while the use of high-order stencils for derivative calculations effectively prevents the use of a hierarchical decomposition of the computation. The Legion



(a) Small Problem Size – 131K cells

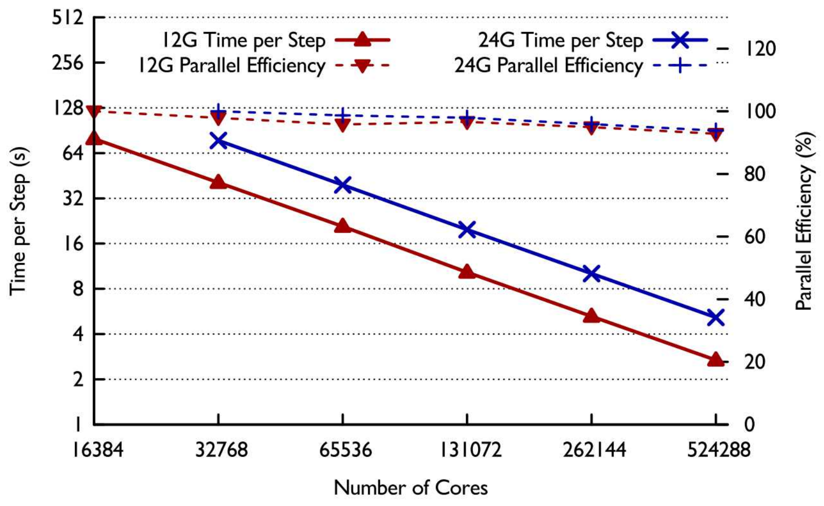


(b) Medium Problem Size – 8.39M cells

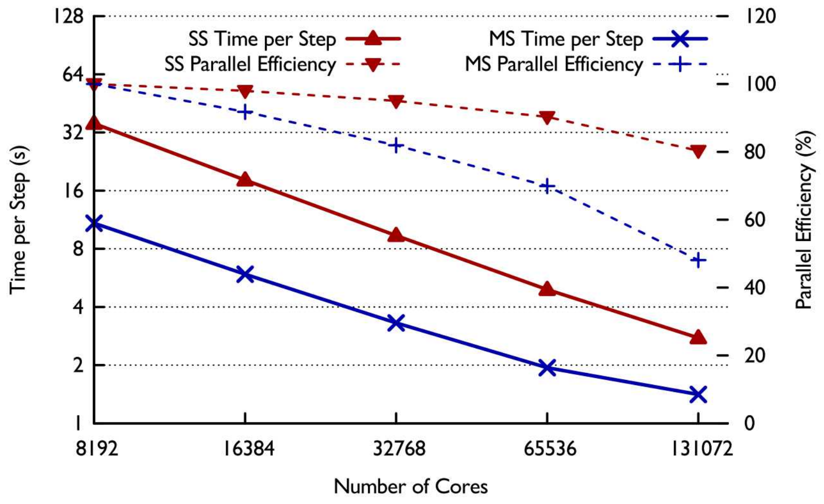


(c) Large Problem Size – 67.1M cells

Figure 3.5: Experiments varying the machine frequency for various problem sizes. We see that the AMT runtimes perform comparably to the MPI implementation. In particular MPI outperforms with small problem sizes per node. There is a cross-over point however where at larger problem sizes the AMTs outperform the MPI implementation.

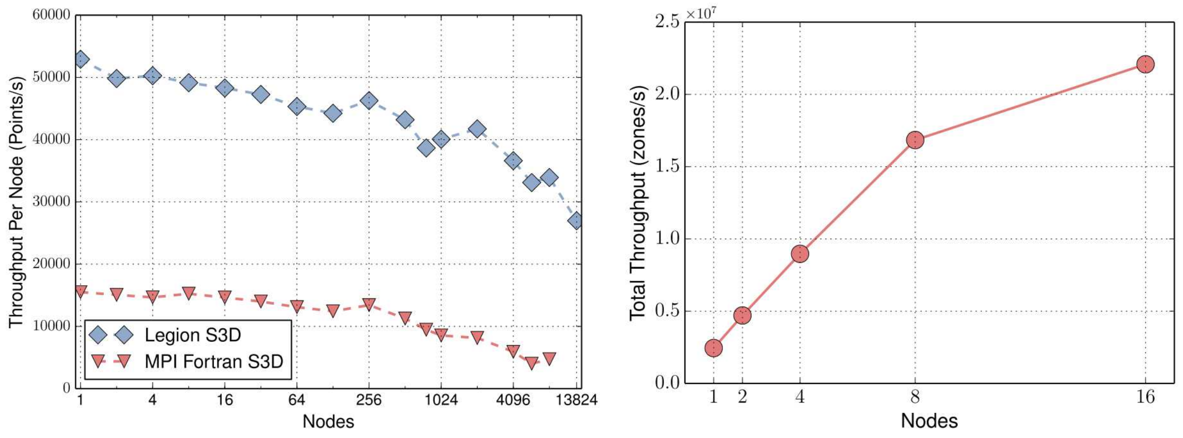


(a) Time per step and parallel efficiency



(b) Performance comparison of single stepping (SS) and multi stepping (MS)

Figure 3.6: Performance results for ChaNGa on Blue Waters. In Figure 3.6a results are shown for simulations with 12 and 24 billion particles. Both the cases scale well achieving a parallel efficiency of 93%. Figure 3.6b shows time per step and parallel efficiency for a simulation with 2 billion particles.



(a) Weak Scaling of S3D on Titan

(b) Strong Scaling of SPMD-transformed PENNANT

Figure 3.7: Examples of Weak and Strong Scaling Results for S3D and Pennant

port was initially done in the naïve way, which was much better for debugging at small scale and for making sure that all of the tasks and privileges were set up correctly. It also ensured that the most benefit was made of Legion's asynchronous execution within a "rank", resulting in single-node performance that was more than twice the reference GPU-accelerated implementation [25]. Unfortunately, S3D uses many small tasks (hundreds per second per node) and launching those all from a single copy of the top-level task quickly showed up as the bottleneck. Using the same grid-based decomposition and communication pattern as the original MPI implementation, the SPMD-ification of S3D's top-level task required only a few days of work, turning an application that showed negative weak scaling above 16 nodes to one that scales nearly twice as well as the MPI reference at 8192 nodes (i.e., $131,072$ cores) on Titan (see Figure 3.7).

The automatic SPMD-ification optimization is currently being implemented in the Regent compiler, but once the necessary analysis and transformations are verified, an effort to implement them within the Legion runtime will commence. Although the Regent effort is not yet complete, Figure 3.7b shows some promising results. A Regent version of the PENNANT [86] mini-app was analyzed by hand (using the proposed compiler algorithm) and then transformed based on that analysis, resulting in near-linear strong scaling at small node counts. (The test case used is small enough that there was almost no scaling before the transformation.)

Uintah A case that best illustrates the scalability of Uintah is the simulation of a complex multi-material problem that captures the fluid-structure interaction using the so called MPMICE method. MPM stands for "material point method" according to which solid objects are discretized into particles (material points) each of which contains all state data (position, mass, volume, velocity) corresponding to the material portion they represent. On the other hand, ICE stands for Implicit, Continuous Fluid, Eulerian, and as the name suggests represents the physics of the fluids on a continuous Eulerian mesh. When combined MPMICE denotes a particle-in-cell method with the particles representing the solid material and cells representing the fluid material. Such problems are inherently load imbalanced. In addition adaptive mesh refinement adds further dynamic load imbalance to MPMICE problems. A full MPMICE-AMR simulation stresses all the features of Uintah. Figure 3.8 shows the scalability results from a Uintah MPM-AMR-ICE simulation on three leading DOE petascale platforms: Titan (Cray XK7), Mira (BlueGene Q) and Blue Waters (Cray XK7). This simulation contained 1.2 million mesh patches, 4 billion mesh cells and 29 billion particles [4].

3.3 Mitigating Machine Performance Heterogeneity

Here we explore the ability of each runtime to mitigate machine performance heterogeneity. As was discussed in the introduction, there are numerous sources of performance heterogeneity on future architectures, including accelerators,

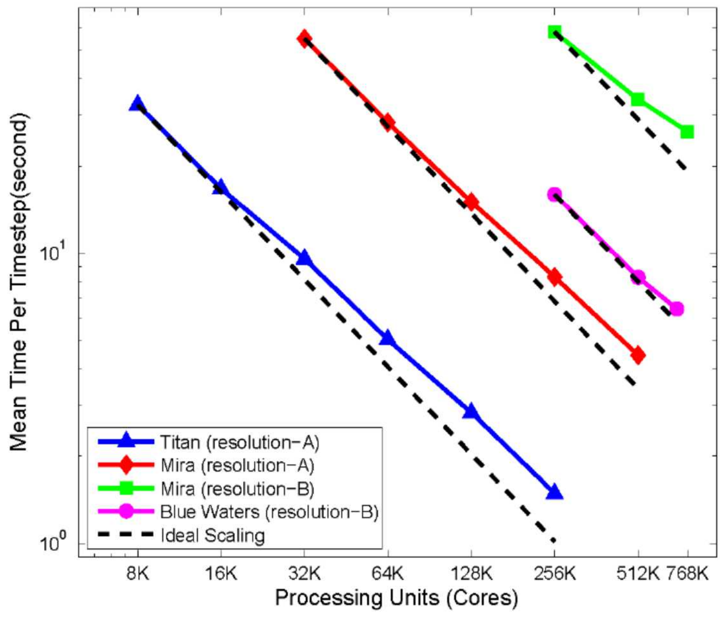


Figure 3.8: Strong scaling results of a Uintah MPM-AMR-ICE simulation on three platforms

responses to transient failures, thermal throttling, and general system noise. Each of the runtimes provides various mechanisms to facilitate load balancing or work stealing, as summarized in their respective subsections in Chapter 2.

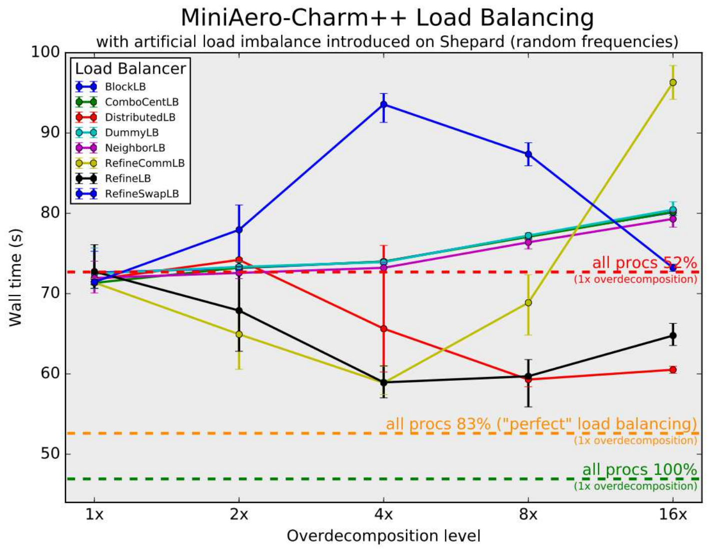
Because we were unable to run the distributed implementation of MiniAero Legion, we currently do not have results for these experiments. However, the mapping functionality to support the load balancing required for our experiments is ready and we look forward to performing this suite of experiments for Legion once the runtime issue is resolved.

In the case of Uintah, their load balancing capability employs a forecasting timing model summarized in Chapter 2 and described in detail in [54]. This timing model associates weights with the various aspects that contribute to the overall runtime cost for a patch. The Uintah runtime is domain-specific, and has support for AMR grids as well as particles. The forecasting model sets up a list of equations that it solves at runtime to estimate the model weights that determine the relative importance paid to local patch size, the number of particles associated with a patch, and the patch's historical execution time. Our use case of a static load (equal sized patches and no particles), but heterogeneous cores had not previously been tested by the Uintah team. Although the Uintah runtime does account for machine performance heterogeneity, application specific needs typically dominate the costs from a load-balancing perspective. Our use case hit a degenerate edge case in their forecasting model implementation. Consequently, even when turned on, load balancing has no effect for our use case. The Uintah team is making the necessary edits to their code and tuning their forecasting model to account for our use case. We look forward to performing this suite of experiments for Uintah once their development and tuning is complete.

Working with the Charm++ implementation of MiniAero, we performed a suite of experiments on Shepard in which we introduced artificial load imbalance using the `cpu_freq` utility to set the frequency of the machine's nodes to different frequencies. In all experiments we used the `SMP` scheduler. The experiments ran for 15 time steps with load balancing performed at each time step. All experiments used 16 of the 36 Shepard nodes (there were issues with the machine's resources and allocations that precluded us from using the entire machine).

In Figure 3.9a we show a comparison of the performance across all Charm++ load balancers for the configuration shown in Figure 3.9b. In this image we see that `RefineLB` performs best for our use case. In the remaining plots in this section we include results for `RefineLB`, `DummyLB`, and `NeighborLB` only. Note that `DummyLB` essentially does not perform load balancing at all, however it does introduce the synchronization point that is required for coordinated load balancing and is useful when comparing load balancing overheads.

In Figure 3.10a and Figure 3.10b we show the results of the load balancing options for two configurations. In the



(a) Random Frequencies for All Load Balancing Options

100%	52%	69%	52%	100%	100%	86%	100%
69%	100%	100%	52%	100%	52%	100%	86%

(b) Random Frequencies System Configuration

Figure 3.9: Performance comparison across different Charm++ load balancers running each time step. The artificial imbalance introduced on 16 nodes of Shepard is depicted in (b). A problem size of 4.1 million grid points per node was used, running a total of 15 time steps. Each point represents the average of four trials.

x-axis of these plots we vary the overdecomposition levels from 1 to 32. In Figure 3.10a only one node is set to a low frequency and the rest are set at the maximum frequency, as depicted in Figure 3.10c. This scenario could represent the case where e.g., a failure is about to occur in a single node, a thermal hotspot is slowing it down, etc. There are two horizontal lines shown. The top (red) line indicates the time that it takes for MiniAero to run when all the nodes are set to the lowest frequency possible (i.e., 52%) with no overdecomposition (i.e., overdecomposition is equal to one). No load balancer was running to generate this timing in order to avoid the overheads and synchronization costs of the load balancer itself. This line represents a lower bound on the performance we could expect on this machine configuration. Similarly, the bottom (green) line indicates the best case scenario in which all nodes are running at the maximum frequency. In order to determine each of the points for this plot, as well as the horizontal lines, a minimum of 5 trials were run and averaged. We can observe that by using `DummyLB` (i.e., essentially, not balancing the load), we obtain a time to solution that is close to the worst case scenario. This is because even though 15 of the nodes operate at high frequency, the node that operates at low frequency serves as a bottleneck. However, when using `RefineLB`, the results get close to the ideal homogeneous machine performance case. With an overdecomposition level of 4 the simulation achieves its best performance, and with overdecomposition higher than that, runtime overheads and higher surface-to-volume ratios on the chares cancel out the advantages of the increased granularity. In Figure 3.10b we observe a similar experiment, but with a semi-random frequency settings. As depicted in Figure 3.10d the machine is set such that:

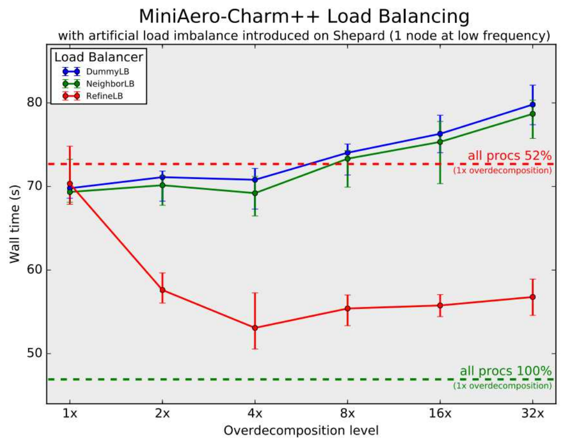
- 50% of the nodes (8) run at 100% of the maximum frequency,
- 25% of the nodes (4) run at 52% of the maximum frequency,
- 12.5% of the nodes (2) run at 86% of the maximum frequency, and
- the remaining 12.5% of the nodes (2) run at 69% of the maximum frequency.

If we average the frequency settings for all the nodes ($0.5 \times 100 + 0.25 \times 52 + 0.125 \times 86 + 0.125 \times 69 \approx 83$, note that the error is due to decimal rounding of 52, 69 and 86), we obtain a frequency corresponding to 83% of the maximum frequency. The horizontal orange dashed line plots the results of running all the nodes at 83%. This is the optimal case that a perfect theoretical load balancer could ever achieve. Again, we observe that with no overdecomposition, there is no benefit from load balancing (because there is only one chare per processor). Similarly, we can conclude that an overdecomposition of four chares per processing unit offers the best performance gain when using the `RefineLB` load balancer.

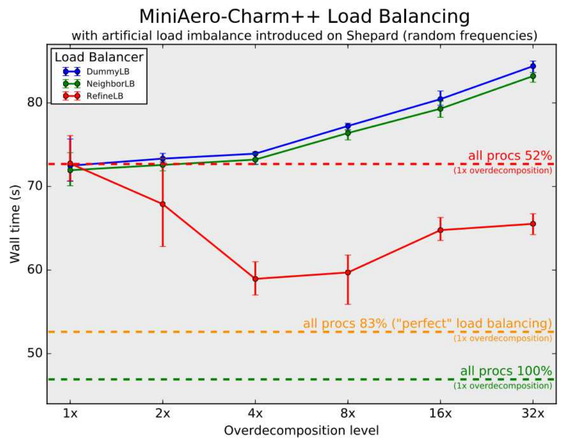
In Figure 3.11 we show the results of another experiment. In this image, the x-axis indicates the number of nodes that have been slowed from maximum frequency down to 52% of maximum, and the y-axis plots the corresponding time to solution for the simulation. We show results for three different load balancers and an overdecomposition level of four (4x). We see how with the `DummyLB`, a really small imbalance (i.e., just one node) quickly shoots up the total end-to-end simulation time. When running `RefineLB` we can observe that much better performance is achieved, because the load balancer redistributes the work among the nodes operating at higher frequencies. We note that could not find any plausible explanation for why `NeighborLB` performs slightly better than `DummyLB` when there is no imbalance (i.e., at the x-axis points of 0 and 16).

Shortly after we finished these experiments, we learned that the Charm++ team has developed load balancers with parameters and inputs specifically tuned to address machine heterogeneity. For the load balancers to utilize this information, the `+LBTestPESpeed` run-time command-line flag must be given, which we did not do in these experiments (this flag is not documented anywhere in the Charm++ manual; we learned this from direct discussion with the developers). Not all of the load balancers even utilize this information, but nonetheless we would expect better performance for some of the load balancers with this additional metric enabled. System heterogeneity is an active area of research in the Charm++ development team, and they anticipate that most of Charm++'s load balancers will handle machine heterogeneity explicitly in the near future.

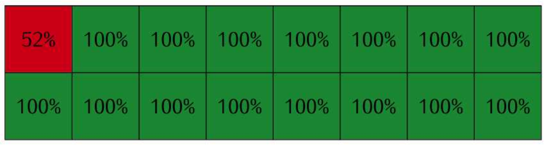
In summary, these experiments on Shepard clearly demonstrate that AMT runtimes can mitigate machine performance heterogeneity at the runtime system level when there is static machine load imbalance. In addition to these studies, our team has begun a suite of experiments to examine AMT runtimes' ability to mitigate *dynamic* machine performance heterogeneity, such as that which occurs under a power-capping regime. For this next set of experiments we are using Mutrino, one of the Application Regression Testbeds (ARTs) that was delivered to Sandia as part of the upcoming Trinity contract. Our team is in the initial stages of these experiments, but will continue them as future work during the course of the next fiscal year.



(a) Single Node Imbalance Plot

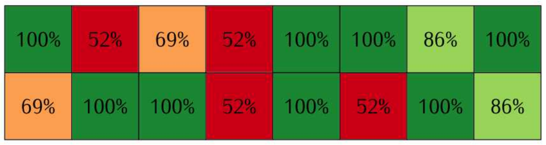


(b) Random Frequencies Plot



52%	100%	100%	100%	100%	100%	100%	100%
100%	100%	100%	100%	100%	100%	100%	100%

(c) Single Node Imbalance System Configuration



100%	52%	69%	52%	100%	100%	86%	100%
69%	100%	100%	52%	100%	52%	100%	86%

(d) Random Frequencies System Configuration

Figure 3.10: Plots (a) and (b) and their corresponding machine configurations (c) and (d) for the load imbalance experiments on Shepard. In (c) a configuration is shown where one node is set to 52% of maximum frequency. In (d), there are irregular frequency drops observed across the machine.

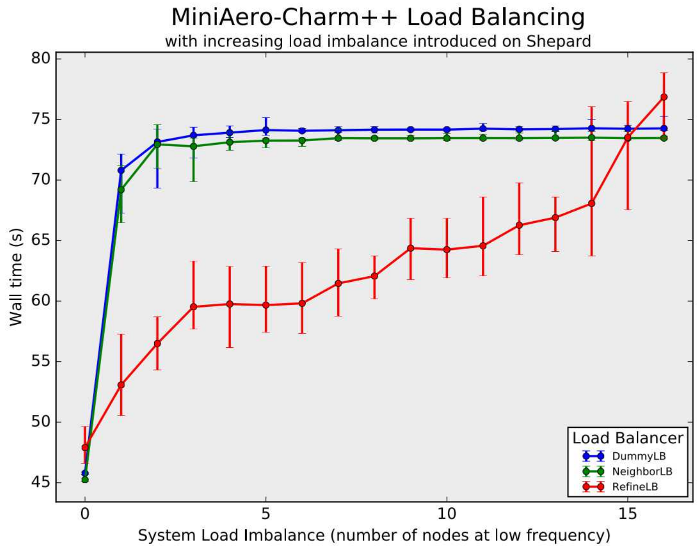


Figure 3.11: System Load Imbalance experiment using Charm++ on Shepard. The x-axis indicates the number of nodes that have been slowed from maximum frequency down to 52% of maximum. The y-axis indicates the time to solution for the three load balancers in this study.

3.4 Fault Containment and Recovery

3.4.1 Extreme-Scale Challenge

At extreme scale the significant increase in the complexity of hardware and software stacks is expected to dramatically increase the occurrence rates of errors impacting application progress and correctness [90]. While detecting, containing and correcting faults in the user level application code is hard enough in a bulk synchronous programming model, it can be significantly harder in an asynchronous many task programming model. By definition, in an AMT programming model no assumptions can be made about what order tasks get executed or where a certain piece of data is physically located. Furthermore, depending on how much of the specifics of the mapping of tasks and data to physical resources are exposed by the runtime, explicit management of faults and fault recovery in the user code might range from seamlessly easy to next to impossible.

A key issue here is the ownership of the responsibilities for fault recovery. Should fault detection, containment and recovery be transparent to the user and handled entirely by the runtime? Or, conversely, should the fault recovery strategies and implementation be left entirely to the user with the runtime providing necessary features to express these? It is instructive to consider two current projects that are aiming to make MPI fault tolerant: FA-MPI [91] and ULFM [59]. In both of these the primary aim is to maintain the MPI environment through failure of one or more processes. However, the bookkeeping to revoke communicators, reassign ranks and recover any lost data or messages is left entirely to the user. This gives complete control to the user in deciding how to respond to failures that would otherwise globally halt execution. This is achievable in the bulk synchronous model using a combination of strategies, but require effort by the user to implement one or more of the strategies. Alternatively middleware that can handle faults transparently can be built on top of these libraries which application users can use off-the-shelf (i.e., LFLR [92], Fenix [93] and Falanx [94]).

In an AMT environment if the programming model and the underlying execution model are completely independent, this becomes harder to implement in the user-level code. However, since the runtime systems do the global bookkeeping for tasks and data anyway, including where they are physically mapped to at any given instant, implementing common fault recovery methods should in theory be easier to implement at the runtime level rather than at the user level. The potential drawback of having the runtime handle everything, however, is that it might become impractical or impossible to design for all possible modes of faults and all possible methods of recovery. Certain users might require fault recovery strategies that are more optimal for their case than what the runtime system is able to provide. Consider two immediate actions required when a node or core fails: (i) the load has to be re-balanced, and (ii) the data lost has to be recovered. In a "non-shrinking" resources scenario, where the lost node/core can be replaced by a spare, the repartitioning does not have to happen globally with the spare resource simply taking over. In a "shrinking" scenario, where the failed node/core is not replaced, the partitioning has to be redone by all the remaining resources. Likewise, for recovering lost data, in the "roll-back" strategy, all processes roll back to the last stable checkpoint and then continue forward—a global response. Alternatively, in a "roll-forward" strategy, only the tasks required to recover the lost data can be replayed which should affect only a small subset of the processes. A combined non-shrinking roll-back strategy is case independent and can be incorporated into the runtime. On the other hand a shrinking roll-forward strategy is very case specific and can not be generalized at the runtime level.

In the remainder of this section we present the current support and the research plans for each of the runtimes in our study with respect to fault tolerance.

3.4.2 Charm++

Of all the runtimes covered in this study Charm++ appears to have explored fault tolerance and recovery the most. By leveraging the concept of migratable objects that is central to their model the Charm++ team has designed various protocols for automatic recovery from failures [95]:

- The simplest protocol involves periodic checkpoint to disk and an automatic restart from a failure on a different number of processors that read the last checkpoint from disk. The protocol requires synchronization for the checkpointing.
- In-memory checkpoint where duplicate copies of checkpoint data are stored at separate locations. Upon failure

the protocol triggers a restart by reading the in-memory checkpoint and continuing execution on the remaining available processors and any performance penalty due to a reduced number of processors is amortized by load balancing schemes. This checkpointing can also happen on local non-volatile memory (e.g., Solid-State Disk (SSD)).

- A sender-based pessimistic message logging protocol in which, prior to sending the message, the sender sends a ticket request to the receiver. The ticket reply from the receiver logs the sequence in which the receiver processes the message. The sender sends the actual message upon receiving the ticket reply. When restarting from a failure the objects pertaining to the restarting processor are spread across the other processors which reduces the restart time compared to conventional message logging approaches.
- A proactive protocol which relies on the availability of schemes that can predict an impending failure. When a warning is received that a node is about to fail the runtime immediately migrates the objects on that node to other locations and also changes the runtime for seamless continuation of message delivery.
- An automatic checkpoint restart protocol [96] which provides recovery from both hard and soft errors. The protocol divides the resources into two equal partitions and performs application replication i.e., executing the program simultaneously on the two partitions with identical checkpointing frequency. During a checkpoint, every node in one partition sends a copy of its checkpoint data to a "buddy" in the other partition, which compares that copy of the data with its own local copy. If a mismatch is found, both the node replicas are rolled back to the last stable checkpoint and repeat execution, which protects against soft errors. For hard failures spare nodes replace failed nodes and receive a copy of the checkpoint data from the designated "buddy" in the other partition to make forward progress.

3.4.3 Legion

While currently there are no supported fault recovery mechanisms in Legion, the team is actively working on adding some features. In existing programming models, finer-grained approaches to resilience can be difficult to implement and maintain. One must determine which computations were impacted by corrupted data and then which other computations need to be performed again to restore the corrupted data. This effort is even more difficult if a fault is not detected immediately. In contrast, Legion's hierarchical data model, and its use of tasks and privileges allows the runtime to perform these analyses automatically. A convenient side-effect of the deferred execution model in Legion is a memory location will not be reused until necessary, so it will often be the case that the input data for the tasks that need to be rerun is still available. If the data is not available, the Legion runtime can work backwards within the task graph or upwards in the task hierarchy as needed to find data from which computation can be restarted. This effort will still benefit from programmer input regarding which logical regions are good candidates for checkpointing, but the use of the Legion mapping API allows the programmers involvement to be limited to policy directives (e.g. "after this task is finished, make a copy of its output logical region in this other memory (or on disk)") and the Legion runtime can take care of all the actual implementation and hiding of communication latency, just as it does with any other data movement.

3.4.4 Uintah

Uintah currently has no fault tolerance features but have secured NSF funding to work on this topic. They are actively working on user system monitors for hard fault-tolerance. They also propose to use Task replication at a coarse level using AMR, providing a sort of double redundancy. A task is replicated at a coarse level at 1/8 of the cost. If the parent task fails then interpolation is used to reconstruct the parent task data. Being built on top of MPI, they propose to use MPI-ULFM [59]. It is probably safe to speculate that, being an MPI+X model, Uintah will depend on fault-tolerance features becoming mature in the MPI standard as well as compatible node-level libraries. This should allow the runtime to respond to both hard faults (node/core failures) as well as soft faults.

3.5 Complex Workflows

3.5.1 Extreme-Scale Challenge

The computing power at extreme scale will enable application scientists to simulate broader regimes and scales of the physics inherent to their problems than currently feasible. While this was always inevitable, there are important challenges and opportunities at this juncture. With power constraints being more important than ever the double imperatives of reducing power usage and reducing data movement will require novel workflows that will be increasingly more complex. For instance many applications currently have the approach of either performing the simulation first, storing state to persistent memory and then performing the analysis *a posteriori*, or, conversely performing a bulk of the analysis fully *in-situ*. This represents a trade-off between a large data footprint and movement versus a large computational overhead. For instance, anecdotal evidence suggests that the Sandia electro-magnetics code EMPIRE spends about 20% of computational time for *in-situ* analysis and I/O while the combustion code S3D generates a few hundred terabytes of data each simulation.

Neither of these overheads might be acceptable at extreme scale. Simulations, while generating massively increased volumes of data, must organize and analyze this data in a manner that does not add to the computational overhead inordinately while simultaneously reducing the data footprint and movement. This will require weaving of complex workflows that bring together a varied set of codes and algorithms that cover both the physics and data analytics. While such coming together currently happens in an offline collaboration mode, disparate codes will have to be possibly more tightly coupled online at extreme scale.

Complex workflows present unique challenges that are more amenable to AMT runtimes. Different pieces of the coupled codes will likely have different spatio-temporal characteristics in terms of compute intensity, degree of parallelism, data access patterns and the task and data dependencies between each other. It is axiomatic that a bulk synchronous programming model will be severely limiting for designing efficient workflows. However, even in AMT runtimes the benefits might not be automatic. Since the workflows are likely to be increasingly domain specific, both on the physics as well as the analytics side, it might be difficult to generalize the decisions that guide the mapping and scheduling of a workflow to the underlying computational resources. At the other extreme, leaving the decisions entirely on the shoulders of the application programmer might make a runtime less preferable.

3.5.2 Yet Another Mini-App Experiment: the MiniAnalysis API

To better understand how the runtimes in our study handle complex workflows, we wanted to design an experiment in the same spirit as the porting of MiniAero. In that pursuit, we created the MiniAnalysis API, which was designed to stress many of the performance-relevant aspects of *in situ* analysis. Unlike typical mini-apps, however, MiniAnalysis was also designed as a proxy for the typical code-coupling experience. The interface was designed to simulate the externally facing portion of a production analysis library and the process of interfacing user code with such a library. It is extremely generic and relies heavily on C++ templates to interact with user code via static polymorphism. (We use the term "user" here loosely to indicate the developer interacting with MiniAnalysis). The goal of the experiment was to try to implement a connection to the MiniAnalysis API in each of our MiniAero implementations *without* changing any of the internals of MiniAnalysis, thus simulating a similar experience with a production analysis or visualization package.

Most code coupling endeavors revolve around two fundamental points of interaction:

Data Format Translation: external libraries want to interact with user data through their own structures and abstractions. The API needs to provide the hooks for the external library to capture or inspect user data on its own terms, agnostic of user data structures

External Workload Invocation, or Workflow Insertion The external library's API needs to provide hooks for the user to invoke the actual work to be performed on the user's data

For the former of these, the MiniAnalysis API requires users to provide an implementation of one basic abstraction: `DataReader`. The user's implementation of a `DataReader` essentially tells MiniAnalysis how to iterate over the user's data *in situ*. This implementation must be a C++ class templated on two parameters: a type `Container` and an enumerated type `DataAccessOrder`. The concrete `Container` template parameter is provided to the MiniAnalysis interface by the user when making a MiniAnalysis `Analyzer`, whereupon the interface internally instantiates the user's `DataReader` with a `null` pointer to a `Container` instance. Then when the user wishes to "analyze" data, he or she provides the `Analyzer` with a reference to the `Container` and invokes the `Analyzer::analyze_data()` instance method. The `analyze_data()` method provides the second of these fundamental interaction points for MiniAnalysis. The `Container` can be an arbitrary user-level or runtime-level construct that the user-provided `DataReader` implementation knows how to iterate over. Indeed, in our three use cases, the `Container` was: a user-level N-dimensional array object (MiniAero-Charm++), a wrapper to Uintah `PatchSubset` and `DataWarehouse` pointers, and a wrapper to a Legion `PhysicalRegion`. The user's implementation of a `DataReader` is required to provide three methods: `has_next()`, `next()`, and `reset()`, all of which tell MiniAnalysis how to iterate over the data to be analyzed. The user's implementation of each of these three methods should comply with the `DataAccessOrder` given as the second `DataReader` template parameter. The allowed values of the `DataAccessOrder` enumeration are `Sequential`, `SequentialJumps`, `Random`, and `RandomStrides`. Given data N to iterate over (expressed as a fraction f of the `Container`'s data, with the constraint that $0 < f \leq 1$), the user's `next()` and `has_next()` implementations should:

- iterate sequentially over N contiguous values of the data (`Sequential`),
- iterate sequentially over the full range of the data, visiting only N values with a constant stride (`SequentialJumps`),
- visit N random data points in random order, accessing each data point no more than once (`Random`), or
- iterate sequentially over the full range of the data, visiting only N values with random strides between each value (`RandomStrides`).

These latter two modes present another challenge to the integration process: that of a non-trivial state that an external library might want to maintain privately. In our use cases, it was deemed too inefficient to generate a random visitation order for each call to `analyze_data()`, and the `DataReader` implementations had to keep track of a visitation index array generated during the simulation setup phase. As discussed below, this proved to be an interesting challenge when working within some of our runtimes' programming models.

The second key element of the MiniAnalysis interfacing process is the integration of the analysis workload into the application's workflow. Practically speaking, this involves creating an instance of the MiniAnalysis `Analyzer` class during the simulation setup phase, giving this instance access to reference to the `Container` object containing the data to be analyzed when it becomes available, and calling `Analyzer::analyze_data()` in the appropriate place to allow the analysis to run *in situ* and out of the critical path of the main computation. In our use case, we further imposed the requirement that it must be *possible* for the runtime to run the analysis task without copying any of the data to be analyzed (though in the case of Legion, the runtime's programming model dictates that the data *may* be copied before the task is run if the scheduler or mapper deems it helpful for performance reasons). Implicit to this portion of the interfacing process is the need to ensure that the data under analysis is in the expected state before the analysis begins and is not modified during the analysis task. In our case for MiniAero, we required that the analysis be performed every timestep on the previous timestep's solution vector (which must persist during the current timestep due to the nature of the RK4 algorithm). With this preliminary study, we primarily targeted the use case where the application developer might want to do some small, *in situ* analyses every iteration out of the critical path with some extra compute resources that might otherwise be idle. Such smaller analyses could be used, for instance, to trigger larger analyses on a variable timestep interval based on some results from the smaller analyses.

One critical aspect of *in situ* analysis that was omitted from the MiniAnalysis study due to time constraints was that of communicating analysis tasks. For many *in situ* analysis workloads, the degree to which the programming model facilitates this communication and the efficiency with which the runtime implements it are critical, and we would like to follow up on this aspect in future work. In addition to significant time constraints, this omission was partially due to the difficulty in expressing a generalized API for analysis communication and a generalized backend for simulating common analysis communication patterns. The former of these challenges was particularly difficult with respect to expressing communication without resorting to a message-passing paradigm. Most existing, production-level analysis APIs that support communication would likely not "ask" the application how to communicate through an abstract interface anyway; instead, such communication is likely to be handled by the library itself, almost certainly using MPI. In that sense, the communication problem reduces to one of how well the runtime can interact with third-party MPI applications and libraries, which is discussed elsewhere in this document (see, for instance, Chapter 4).

3.5.3 Charm++

Owing partially to the lack of programming model support for data-driven semantics (which comes with the significant downsides discussed in Section 2.2.2), the `DataReader` implementation portion of the MiniAero-Charm++ interfacing process was the easiest of the runtimes in our study. The MiniAero-Charm++ implementation stores the data to be analyzed in a user-defined N-dimensional-array-like object (designed to mimic a `Kokkos::View` object to maximize code reuse from the MPI baseline MiniAero implementation), of which the Charm++ runtime has no knowledge or interaction beyond a basic serialization interface. Thus, the process of creating a `DataReader` for the MiniAero-Charm++ data was about as difficult as implementing one for a `double*`. As "users," we had easy access to the internals of the user-level data structure to be iterated over, and implementing the data access patterns required by MiniAnalysis was very straightforward.

The other portion of the interfacing process, insertion of the analysis workload into the application's workflow, was also relatively straightforward in MiniAero-Charm++, though the process of doing so leads to several interesting observations. The lack of permission-dependent data-flow semantics in the Charm++ programming model means that data access ordering for an out-of-critical-path analysis task must be managed explicitly through execution-driven event-based semantics. While this was relatively trivial in MiniAero (and it could be argued that it actually resulted in fewer lines of code than the analogous portion of the other implementations), it would likely be much harder or even prohibitive in a more complicated application. The maintainability of the code is also negatively affected by this design pattern, since new code must also be aware of these explicit execution order constraints when accessing the analyzed data. Moreover, the errors arising from mistakes in this implementation pattern are the worst kind possible: errors at the time the program is executed that result from race conditions and may go unnoticed for years until the right set of conditions (or even, for instance, updates to the runtime's scheduler) cause ordering violations to occur.

Another interesting aspect of the process with Charm++ was the requirement that it be possible for the analysis task to run in parallel and out of the critical path. Because the migratable-objects programming model couples data and work on that data into one "actor," and because these actors are restricted such that they (typically) only run one task on that data (recall from Section 2.2.2 that Charm++'s primary data safety model is based on disjoint data), allowing two tasks to operate on the same data simultaneously necessarily requires deviation from Charm++'s primary supported programming model. To allow the analysis tasks to run out-of-critical-path and simultaneously to the main workflow, the analysis work must be done by a different actor that is co-located with (i.e., having access to the same shared memory as) the main workflow actor. These actors then exchange a message containing a pointer to the data and manage access order restrictions explicitly using `when` dependencies in the ci file. It is possible to ensure two chares (Charm++'s actors) are co-located using the `nodegroup` construct in Charm++'s ci file interface (and this is what we did), but the use of constructs that are explicitly tied to physical computing resources makes the code start to look a lot like MPI and makes it harder to efficiently take advantage of the load-balancing and resilience features available in Charm++. It is also important to note here that the event-based, execution-driven semantics of Charm++'s ci file syntax (specifically, the so-called "structured dagger") make it easy to conflate order-independent execution and concurrent execution: while the user can express the execution-dependent portion of a task-DAG with concurrently executable tasks in the structured dagger, the inability to express the data-flow information means that these tasks cannot be executed concurrently, only order-independently.

Existing demonstration of complex workflow coupling Charm++ does provide the functionality for external library chare arrays to be "bound" to analogously indexed portions of user chare arrays, such that two chares with the same array index in separate arrays are always migrated together and thus always have local access to each other's members. This capability has been leveraged to create the LiveViz library, [97] which facilitates *in situ* analysis in Charm++. The Charm++ developers have utilized LiveViz to build an *in situ* performance monitoring and visualization tool. [98] This tool gathers utilization statistics about the running program on every processor, efficiently compresses the utilization data, and merges the compressed data in a reduction from all the processors. During execution, the performance monitoring tool is just another module which runs alongside the application (like other Charm++ functionalities do), without affecting the application significantly. It can send messages that are independent of the messages being sent by the rest of the parallel program, and are handled gracefully by the RTS. A visualization client, run on a workstation, can connect to the performance monitoring tool and retrieve continuously updating utilization profiles from the parallel program. We did not investigate using bound chare arrays or the LiveViz library to implement our interface to MiniAnalysis due to time constraints.

3.5.4 Uintah

The basic story for the other two runtimes in our study is relatively similar: contrary to the experience with Charm++, the data-flow semantics available in both Uintah and Legion make it much easier to express the concurrency characteristics of the analysis workload, but the need to interact directly with the runtimes' data structures makes it harder to implement the `DataReader` portion of the interface. Uintah's data model involves a construct called a `DataWarehouse`, with fields identified using a `CCVariable`. The `DataWarehouse` is decomposed into pieces of type `Patch`. A set of data to be analyzed by a MiniAnalysis task can be uniquely identified by a `DataWarehouse`, a `Variable`, and a `Patch` or list of patches. Thus, the `Container` for the MiniAnalysis `DataReader` implementation in MiniAero-Uintah is essentially a struct of (pointers to) these instances of these three classes.

The more difficult part of implementing the `DataReader` in MiniAero-Uintah was ensuring that inner loops (i.e., the `next()` and `has_next()` methods) could be inlined and compiled efficiently. While the Uintah data structures do provide access to the raw data pointers, the programming model primarily encourages the use of the `CellIterator` object. Values are obtained by calling the square bracket operator on a `Variable` object with a `CellIterator` instance. All of the steps in this process seem to be carefully defined as inline methods, though we did notice that the `CellIterator::operator+=(int)` was simply implemented as multiple consecutive increments. We did not attempt to implement a version of the `DataReader` that utilizes raw pointers to the data, though we suspect it would not be that much harder if certain assumptions about the consistency data layout are made. Unlike Legion, where the programming model clearly encourages the user to write domain code that is independent of data layout, it is unclear whether or not Uintah's provision of access to raw data indicates that the user can expect the runtime will not change its data layout pattern at some point in the future. Nonetheless, the basic programming model of Uintah is much more amenable to complex data-driven workflows than that of Charm++.

3.5.5 Legion

Like Uintah, the permissions-dependent data-flow semantics in Legion made the out-of-critical-path integration of the analysis workloads relatively trivial (albeit verbose). The actual iteration portion of the `DataReader` object was not that difficult either. Similar to the Uintah implementation, an `IndexIterator` can be constructed for the data in a given `PhysicalRegion` of memory, and a particular field of that data can be selected based on a given `FieldID`. The Legion iterator has the additional ability to retrieve contiguous blocks of memory with one iterator method call, which we found to be particularly useful in writing efficient code.

The major difficulties encountered in coupling MiniAnalysis to MiniAero-Legion involved maintaining the state of the `Analyzer` objects across tasks that can be mapped to any physical resource and thus must store all of their state in Legion data structures. For relatively simple plain old data objects, Legion's data model works quite well for this (though it is unclear how much overhead would be required if the data did not map to an indexing concept and thus must be stored in a logical region of size 1 per node). However, for complex data types that contain both stack- and heap-allocated resources, this is much more difficult. In general, one cannot assume that external library constructs will be POD objects, and thus an exploration of maintaining state of non-POD objects in this sort of situation is critical. In our case, the `Analyzer` in the MiniAnalysis library is not a POD object because it contains the pure virtual method `analyze_data()`. (In the MiniAnalysis library, the purpose of this virtual method can be thought of as the interface point between static and dynamic polymorphism, allowing the user to take advantage of inline static polymorphism in the inner loop through a single virtual method call on a polymorphic base class pointer). Thus, within the primary programming model of Legion, the `Analyzer` objects must be deserialized from a Legion `PhysicalRegion` at the beginning of each analysis task and re-serialized at the end of each analysis task. Clearly, this is unacceptable for our use case, as discussed in Section 3.5.2. It is also not necessarily feasible, since the MiniAnalysis `Analyzers` contain some members for which it is not clear how to serialize (in particular, C++ standard library random number generators). We are thus required to circumvent the primary supported programming model.

One way to deal with this problem, as suggested by the Legion developers, would be to store a pointer to an Analyzer object in each shared memory space and store that pointer in a `LogicalRegion` with an `IndexSpace` of size one per shared memory space. After the pointer is initially stored, we could manually add a special case to our application specific `Mapper` that ensures that pointer gets mapped to the same shared memory space every time, and thus is valid for the application to use. (This would work because the MiniAnalysis `Analyzer` objects do not necessarily need to process data from the same index subspace of the solver data every time, since the library is supposed to be simulating a reduction or random sampling on the data. If `Analyzer` objects had to work on the same portion of the `IndexSpace` every time, the problem would be in some sense easier, because the `Analyzer` objects would either have to be serializeable or risk impeding dynamic load balancing capabilities.)

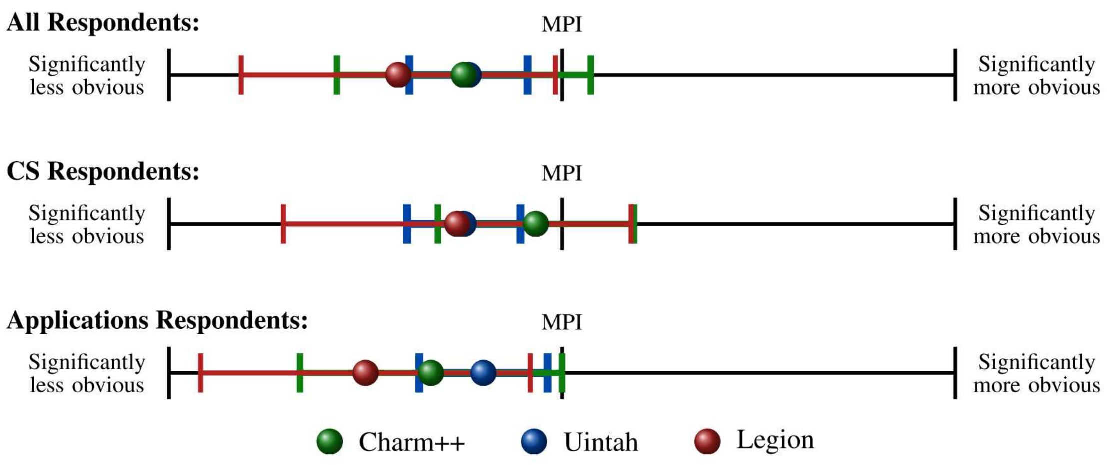
We found this solution to be unsatisfactory, and thus we never carried out a full implementation. Rather, we completed an implementation that only works for the `Sequential` and `SequentialJumps` data access orders, and solves the problem of state by reconstructing an `Analyzer` object each time an analysis task is called (this reconstruction has a relatively small overhead in the `Sequential` and `SequentialJumps` cases). This latter implementation, however, required that we pass to each analysis task an `AnalysisOptions` setup struct. `AnalysisOptions` contain all of the information for transforming the user's input file into an `Analyzer` object, thus allowing access to all of its statically polymorphic variants at runtime. This object is also not POD, so we had to write serialization functions for it. Unlike Charm++, however, Legion contains no framework for serialization and no helper functions for serializing standard library constructs. Thus, in our mini-app, we had to write serializers for `std::map` and `std::vector`, which were used by `AnalysisOptions` or its members, in addition to `AnalysisOptions` itself and several of its helper classes. The whole process was a bit frustrating compared to the other two runtimes. In theory, Legion's data model should be the most efficient and most general for code coupling, but in practice the lack of maturity of the runtime impeded our ability to code up a satisfactory implementation.

3.6 Comparative Analysis

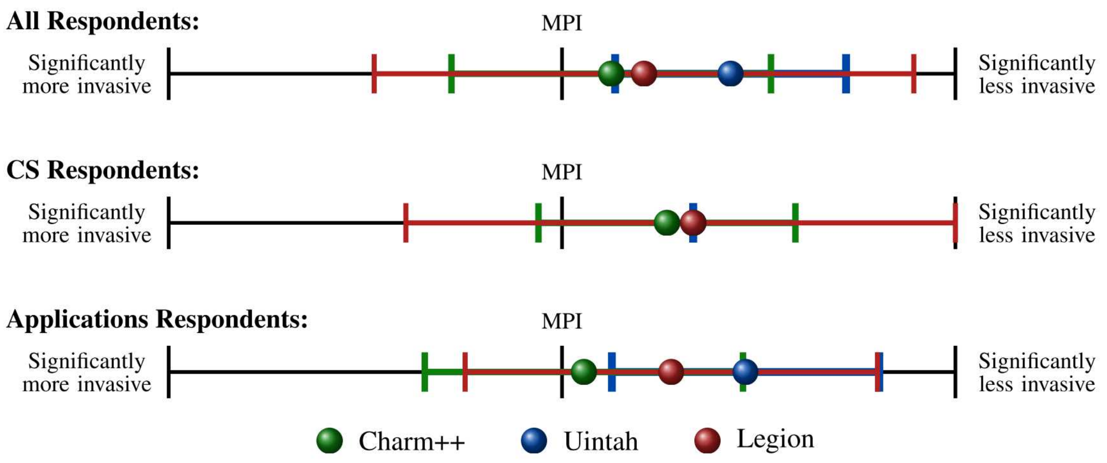
In this section we compare and contrast the three runtimes across a number of subjective performance measures. Similar to the approach in Section 2.5, we posed questions to the attendees of each bootcamp. Attendees were asked to provide a response on a seven-point scale, *relative to MPI*. For each question below, a low score indicates a response "significantly worse than MPI", a mid-range score indicates "comparable to MPI", and a high score indicates a response "significantly better than MPI". For each question, we show overall responses, along with the responses separated according to respondents with an applications (Apps) background versus respondents with a computer science (CS) background.

All respondents felt that, compared with MPI, it was less obvious when an AMT runtime code segment would be expensive, with Legion being the least obvious of the runtimes. This response is interesting given the Mapper interface that Legion provides, which grants the user control over how data and tasks are mapped onto the machine. All respondents agreed the AMT runtimes provided more control via tunable "knobs" than MPI, and that performance optimization was less invasive into user-level code. Although Uintah's runtime provided tools were largely text-based, all respondents agreed that the data provided greatly facilitated in assessing and optimizing performance. In terms of both the potential and availability of fault tolerance and mitigation of machine performance heterogeneity, Charm++ fared better than the other runtimes. With regards to support for applications with dynamic data requirements, the respondents felt both Charm++ and Uintah provided more support than MPI. Respondents rated Legion lower in this regard due to the requirement to specify all data requirements upfront. Respondents viewed both Charm++ and Uintah favorably in the context of performance on bulk synchronous algorithms. Based on the SPMD-ification issues discussed in Section 2.3.3, the current implementation of Legion fared worse than MPI in this regard. However, it was noted that it will be interesting to revisit this question upon the Legion team's completion of the automatic SPMD-ification research.

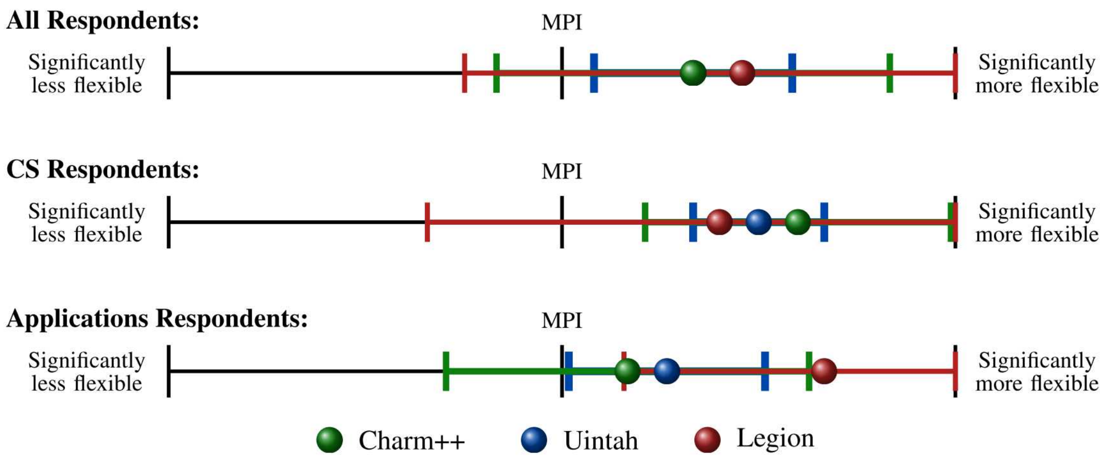
Performance Transparency: How obvious is it when a particular construct or segment of code will be expensive (e.g. trigger data movement, use lots of memory, bad cache use)?



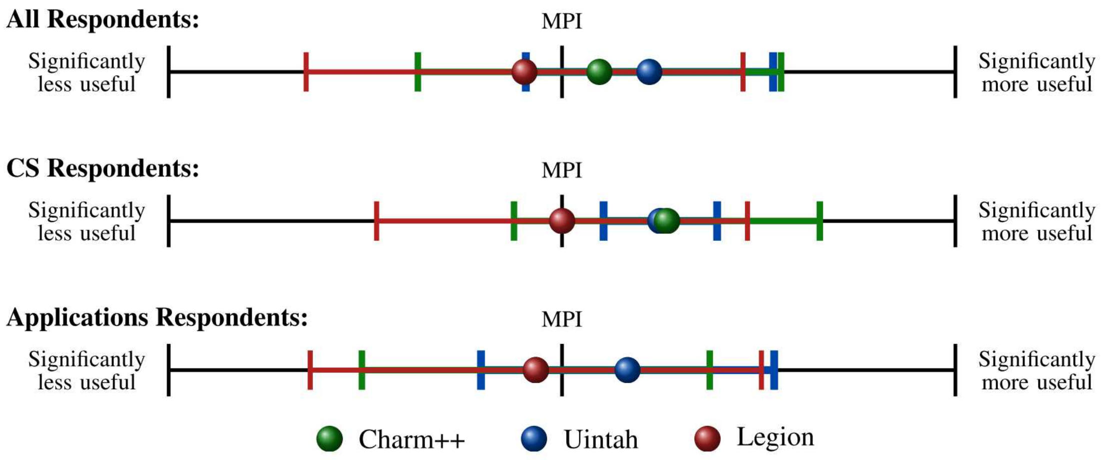
Performance Optimization: How invasive is performance optimization in user-level code?



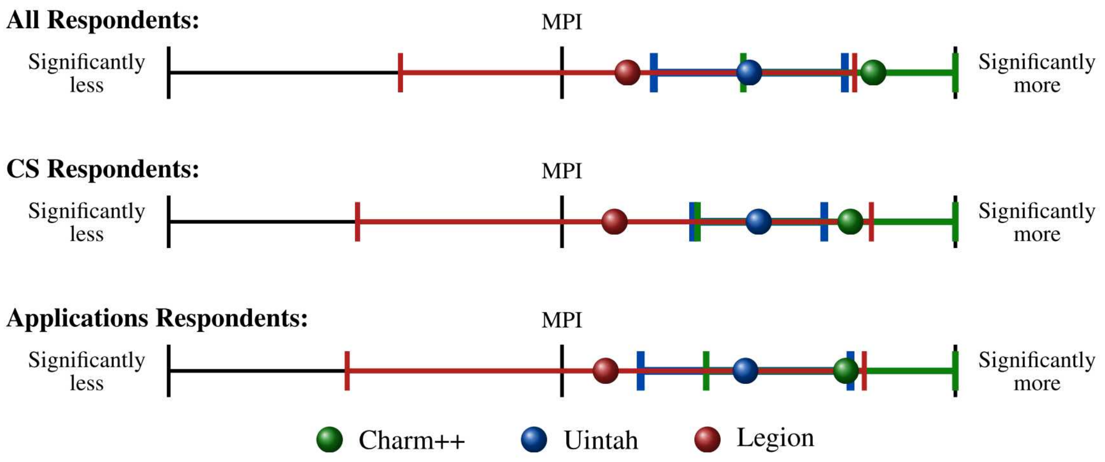
Performance Flexibility: How many “knobs” does the runtime system expose, and to what extent is this flexibility documented and supported?



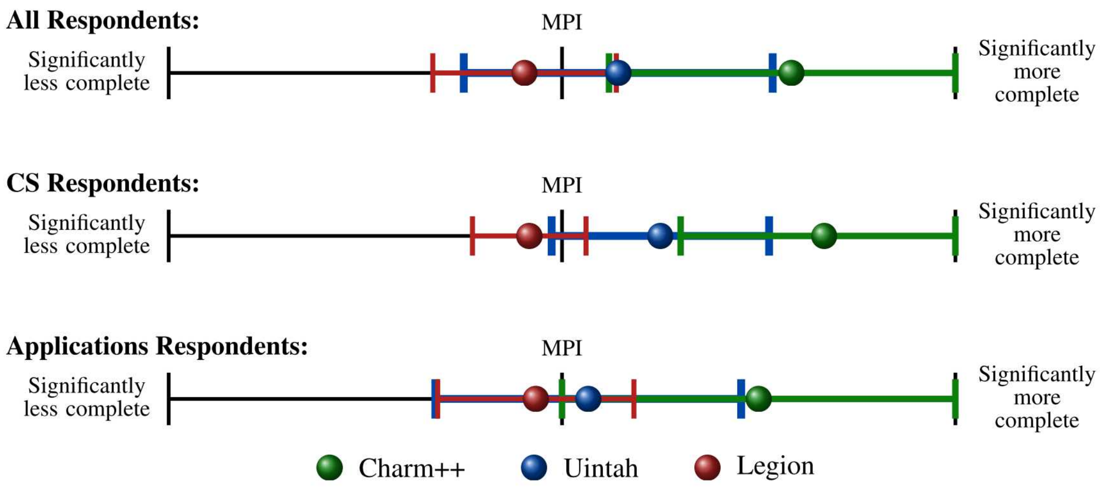
Performance Assessment: How useful are the runtime system tools in helping assess and optimize performance?



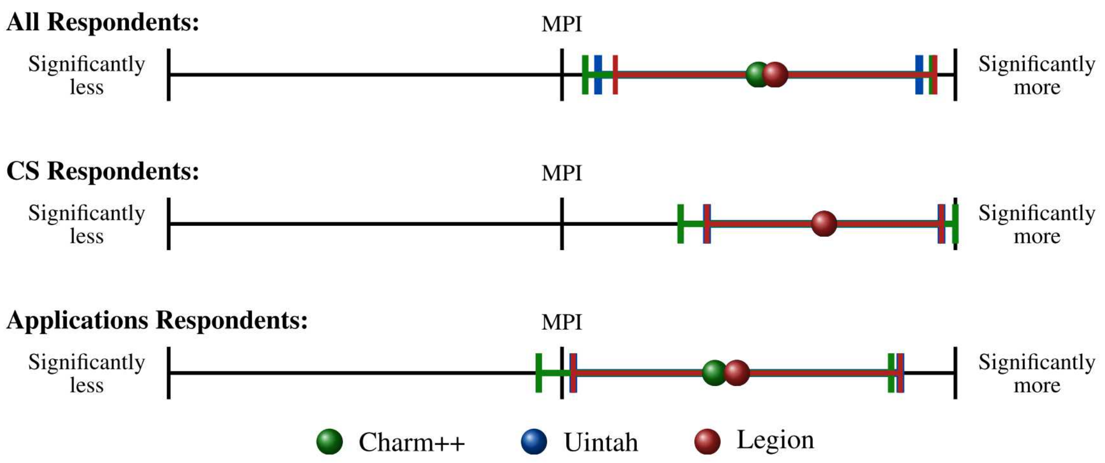
Fault-tolerance Potential: To what extent does the programming model facilitate fault tolerance with minimal modification to the user's code?



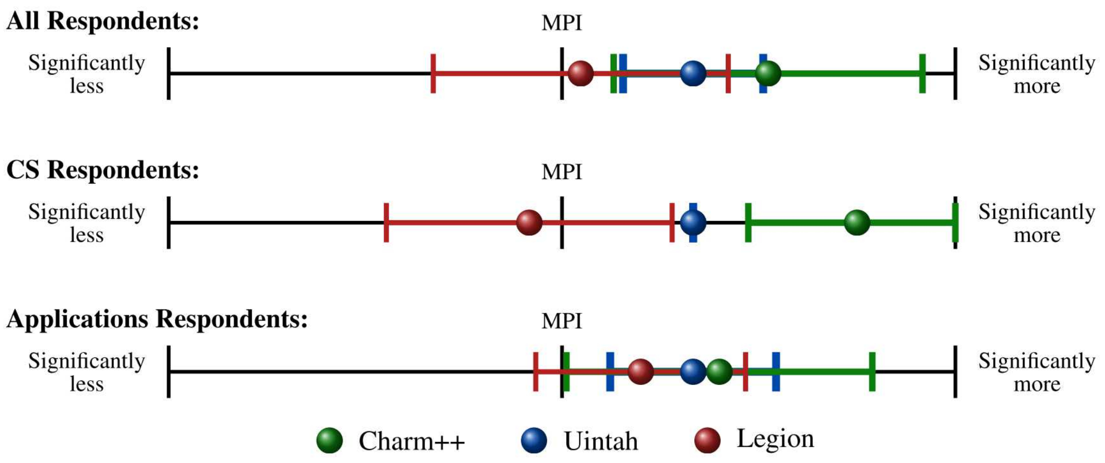
Fault-tolerance Implementation Availability: To what extent have the runtime developers leveraged this fault-tolerance potential in the runtime implementation?



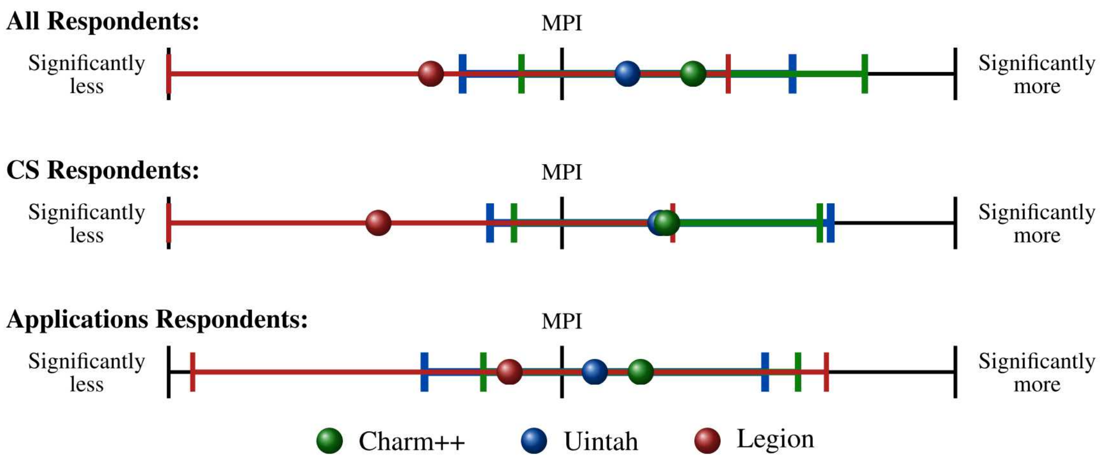
System Performance Heterogeneity Potential: To what extent does this runtime facilitate mitigation of machine performance heterogeneity with minimal modification to the user code?



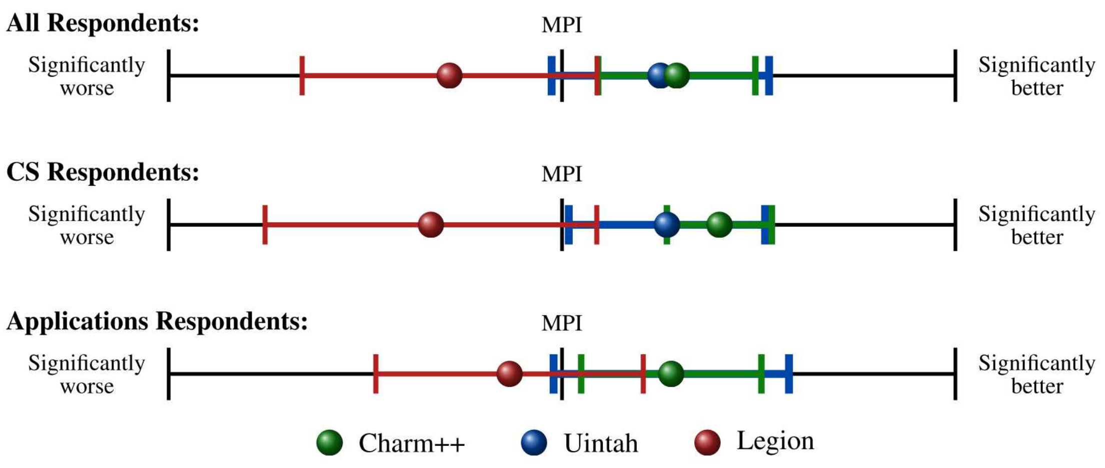
System Performance Heterogeneity Availability: To what extent have the runtime developers leveraged this ability to mitigate machine performance heterogeneity potential in the runtime implementation?



Dynamic Application Performance: To what extent does this runtime support applications with dynamic data requirements?



Bulk-Synchronous Performance: What is the ability of this runtime to be performant for SPMD, bulk synchronous algorithms (i.e., how comparable is performance to MPI on balanced applications running on homogeneous machine)?



Chapter 4

Mutability

4.1 Approach for Measuring Mutability

As we assess the mutability of each runtime, we seek to answer the following question:

What is the ease of adopting this runtime and modifying it to suit ASC/ATDM needs?

In this chapter we provide an assessment of the modularity of each runtime, and assess its interoperability with other languages and libraries (including node-level libraries, such as Kokkos). We conclude this chapter with a comparative analysis based on survey results, analogous to that performed in Sections 2.5 and 3.6, this time focused on mutability issues.

4.2 Charm++ Mutability

4.2.1 Modularity

At its core, Charm++ has two basic layers in the software stack: the main Charm++ layer that the user interacts with (discussed in many other parts of this document) and the Converse [99] portability layer. According to its developers, the Converse framework was designed for quick development of multiparadigm parallel applications, languages, and frameworks. It is designed with the goal that any parallel runtime or library built on top of Converse should be "almost as efficient as a native implementation on each particular machine." The framework provides generic abstractions and machine specific implementations for communication, threading, memory allocation, and many other basic functions. In particular for communication, this means that programmers can call generic Converse communication functions and those calls will be translated into uGNI, DCMF, PAMI, IB verbs, or even TCP/UDP communication functions depending on the machine the code is being compiled on. Converse's communication layer can also simply be built on top of MPI. Converse is quite mature — the original version dates back to 1996 — and, in our experience, very stable. In our performance testing, we built and ran MiniAero-Charm++ on a variety of testbeds, several of which were so new and experimental that the cluster's recommended MPI implementation was not entirely stable. Even in these cases, MiniAero-Charm++ ran at least as consistently and was roughly as performant as the baseline MiniAero version, and in some cases the Charm++ implementation was more stable. While it is clear that the Converse framework is not designed for domain science application developers to write code directly to, a flexible base layer like Converse is ideal for use in AMT RTS development. It is unclear how separable Converse is from Charm++, being integrated into the Charm++ repository and build system, for which potential drawbacks have been discussed in Section 4.2.2. These minor issues could likely be resolved if there was ever a need to isolate Converse for separate use.

4.2.2 Interoperability With Other Languages and Libraries

Aside from the cross-compiled "charm interface" language, Charm++ is written entirely in C++ and expects user code written in C++. The interoperability with other languages is thus tied to C++'s interoperability with other languages, which is relatively well-established. Beyond this basic layer, though, the Charm++ toolchain makes it difficult to connect other libraries and languages to Charm++ application code. Because of the need to cross-compile the ci files in the build process, Charm++ provides a compiler wrapper that builds the cross-compiled header files and links in the necessary runtime libraries automatically. This wrapper calls the user's C++ compiler underneath, and while the user can pass additional flags of their own to the compiler, using an additional layer of compiler wrappers, like NVIDIA's `nvcc` wrapper for CUDA, can be frustrating. Since the Charm++ compiler wrapper is generated when Charm++ itself is compiled, another wrapper like `nvcc` either needs to be used to compile Charm++ itself or needs to be hacked in to the files generated by the Charm++ compilation process. Further complicating matters, Charm++ uses a custom-modified fork of GNU autotools for the compilation of the runtime itself. This is convenient when one needs to compile on one of the machines that the developers of Charm++ have compiled on before, because the best settings and compiler flags are baked right in to the compilation process, but it makes it difficult to customize the build process for custom use cases and interfaces with other libraries and languages.

On the other hand, as discussed in Section 2.2, Charm++ does not have a data model that requires runtime-managed data structures. Thus, when it comes to interaction with libraries like Kokkos, the experience was significantly easier than with the other AMT runtimes. Kokkos is allowed to manage its own data structures without interference from Charm++, provided the actors responsible for the Kokkos data are not migrated. Migration of Kokkos `View` objects seemed to be an issue at first, but because Charm++'s programming model only allows the movement of data via serialization and deserialization, the interface to allow Charm++ to migrate Kokkos `View` objects was written entirely without modification or understanding of Kokkos internals. If the runtime relied heavily on data access permissions and zero-copy semantics, this simple integration would not have been possible. Nevertheless, the relationship between Kokkos and Charm++ in our first implementation of MiniAero-Charm (later implementations removed Kokkos in the interest of reducing the number of variables in the performance studies) would be better described as "coexistence" than "interoperability." No information useful for the purposes of scheduling, cache reuse, or other optimizations was passed between Kokkos and Charm++, and there is not really a user-level interface in either package for doing so. This information exchange would be a feature of an ideal AMT RTS.

Despite the toolchain issues discussed above, third-party MPI-based libraries are straightforward to integrate with Charm++. Charm++ additionally has a custom version of MPI built on top of the actor model called AMPI (for adaptive MPI), which brings some of the advantages of the migratable-objects programming model to MPI-based applications. This, however, may require extensive refactoring of the original MPI code since global variables are not allowed in AMPI. The interaction between MPI and AMT runtimes is an active topic of research in the Charm++ development team, [100] and this support for smooth integration of existing MPI code is a key strength of the Charm++ runtime system.

4.3 Legion Mutability

4.3.1 Modularity

The Legion software stack was designed from the start with modularity in mind. The overall runtime system is split into "modules" as shown in Figure 4.1, with well-defined interfaces between them. The initial motivation for this was to allow multiple research efforts to progress independently, and in their experience has worked well, allowing different researchers to design and iterate on different modules with little interference. The one area where conflicts regularly occur is within Realm [101] which is the low-level runtime in Legion. Although Realm was also designed with modularity in mind, the effort to formalize the internal interfaces has only started recently.

The modular design allows a user of the Legion runtime system to decide for themselves which pieces they wish to use. For example, a compiler for a DSL could choose to generate Regent code, or C++ code that uses the Legion C++ API, or even code that directly targets the Realm API. Conversely, a user could decide to use just the upper parts of the stack, allowing the exploration of different design choices in a "low-level" runtime while keeping support for existing Legion and/or Regent applications.

One of the benefits of Legion's modular design is that it is possible to use some of the modules without having to use everything. The low-level runtime, Realm [101], is one such module. Realm is an event-based runtime system for heterogeneous, distributed memory machines. It is fully asynchronous — all AMT RTS actions are non-blocking. Realm supports spawning computations, moving data, and reservations, which are a synchronization primitive. Realm is currently the foundation for Legion, but could also be used as the foundation for a new runtime system, or existing

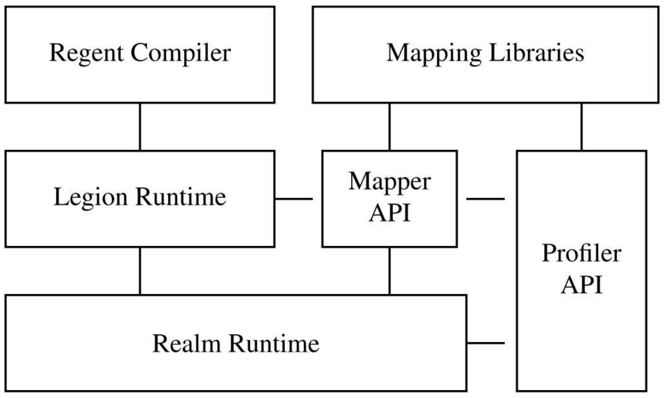


Figure 4.1: Legion Module Block Diagram

MPI-based applications could be written directly on top of Realm; especially if the nature of the application makes it difficult to implement in Legion. The simple primitives of the Realm API are expressive, but also capable of abstracting many different hardware architectures. Combined with automatic latency hiding, the Realm API provides a natural layer of abstraction for the development of portable higher-level runtime systems targeting both current and future architectures.

Legion source code is available via github. The repository contains the runtime system implementation, debugging and profiling tools, several examples that are described in an online tutorial `http://Legion.stanford.edu/tutorial`, several sample Legion applications, and an experimental compiler for the Legion programming language. Legion is licensed under the Apache software license, version 2.0 and should run on any hardware on which the following dependencies are satisfied:

- A POSIX-compliant operating system such as Linux or Mac OSX,
- A C++98 compiler. It has been tested with GCC, Clang, Intel, and PGI C++ compilers. It has also been tested with C++11 features enabled.
- GNU Make,
- Pthreads,
- Python 2.7 for debugging and profiling.

For high-performance environments, in clusters, or with GPUs, the following are also required:

- CUDA, version 5.0 or later for use with NVIDIA GPUs.
- GASNet, for use with clusters.

4.3.2 Interoperability with Other Languages and Libraries

The current implementation of the Legion runtime uses C++98, although later versions of C++ can also be used. Legion applications can be programmed in any language which can interact with a C++-based library. It is easiest for the application to also be written in C++, but C and Fortran should also be compatible with some extra effort.

MPI A Legion-based application can inter-operate with MPI, which helps porting an existing MPI-based code to Legion since the code can continue to execute during the transition when it uses both MPI and Legion. The application is started as an MPI application using, for example `mpiexec` or `aprun`. A Legion-only application would use GASNet to start a process on each node. A call to the Legion library triggers the initialization of the Legion data structures on each node, which means that Legion is initialized within the same process as MPI. When Legion is initialized, the runtime triggers the execution of the top-level Legion task. In a non-MPI coupled application, this task would immediately start the execution of the application; in a coupled MPI-Legion application, this task launches a sub-task on every processor which synchronizes with the MPI process. These sub-tasks synchronize with the MPI thread in each process and determine the mapping from Legion processors which is needed to ensure proper interoperation with MPI. Additional sub-tasks are then launched to distribute the mapping result to each of the mappers in the application.

After determining and distributing the mapping of Legion processors to MPI ranks, the top-level task launches a sub-task for each of the MPI ranks. These tasks are long-running tasks responsible for synchronizing the MPI ranks. Legion can manage these tasks such that they will have the correct memory coherence and launching characteristics to ensure that the MPI tasks are executed at the same time.

This is an effective approach which allows for a portion of the application to be ported to Legion without requiring all code to be transitioned; however, the down-side of this approach is that either all of the computational resources of the machine are being used for performing Legion work or MPI work exclusively with no overlap possible. A better, more automatic approach for dealing with coupled MPI-Legion applications is planned for the future.

This approach was used very successfully in the Legion implementation of S3D [48]. The approach is described in detail in Section 11 of Mike Bauer's PhD. thesis [102], with performance results being shown in Section 11.3. In summary, instead of porting all 200K lines of S3D into Legion, the above method was used to port the right-hand-side function (RHSF) to Legion, leaving the remaining start-up/tear-down code as well as the Runga-Kutta loop in the original Fortran. This function represents between 95%-97% of the execution for a time step in S3D. This is a good example showing an evolutionary path for applications to be ported into Legion without requiring all code to be ported.

Node-level libraries Legion should be able to interact with node-level libraries such as OpenMP or pthreads since the Mapper interface gives the application developer total control over the mapping of tasks. The Mapper could ensure that only a single task was mapped per node and then the task itself could use the complete resources of the node.

Legion has support for GPU tasks with the minor restriction that a GPU task must be a "leaf" task, which is a task that does not call other tasks. The Legion function `register_hybrid_variants` is used to register a CPU and GPU variant of a task. The Mapper object is responsible for determining which variant should be executed. At the time a GPU task is executed, the physical region for the task will be located in the memory that the mapper requested (framebuffer or zero-copy). The user simply needs to launch the corresponding GPU kernel and does not have to be concerned with memory movement. GPU support uses CUDA and the normal `nvcc` CUDA compiler. Legion provides its own implementation of the CUDA API which gives it control over the set of API calls that can be done inside of Legion GPU tasks. This also gives the Legion runtime the ability to track the launching of GPU kernels. Legion has accessors for physical regions. For GPU execution, a SOA (Struct-of-Arrays) accessor is specified which guarantees that all global loads and stores in the GPU kernels will be coalesced.

Kokkos integration currently poses difficulties for integrating with the Legion runtime since features provided by Kokkos intersect with features provided by Legion, particularly data allocation, data layout, and thread allocation. Kokkos provides a template metaprogramming API that automatically transforms and optimizes code to different architectures by changing template View parameters. Kokkos therefore provides both a C++ programming API and a backend implementation. Initial solutions to the integration problem have been proposed that would map the Kokkos API to a Legion implementation through a Kokkos view parameter customized for Legion accessors. This would overcome the initial hurdle of allowing code written in Kokkos to be reused within Legion. Such a solution, however, preserves the API but does not leverage much of the existing code within Kokkos. The Kokkos code that achieves performance portable memory allocation/layout would be replaced by Legion code.

More generally, Kokkos integration illustrates the broader issue of custom data structures. Legion currently imposes its relational data model to enable the runtime to optimize data movement and layout transformations. The runtime is aware of the internal structure of the logical region. In some use cases, applications may not want or require Legion to automatically perform data transformations, instead treating logical regions as simply data blobs. Legion could still provide concurrency management and much of the data movement. More control would need to be granted at the application level over data layout, similar to relaxed coherency modes where control over concurrency is granted in isolation to specific tasks. Such an approach seems feasible, but would require significant extensions to the existing mapper interface. Continued collaboration for a broad set of use cases is required to assess whether (1) applications with custom data structures can be adapted to the Legion data model to ensure the runtime remains aware of logical regions' internal structure or (2) whether enough use cases would justify a "relaxed data model" for certain tasks.

4.4 Uintah Mutability

4.4.1 Modularity

The modularity of Uintah is one of the most compelling features of the runtime system. Functionally, Uintah is divided into member classes and includes the following individual components: a scheduler, a load balancer, a grid component, data warehouses, optional analytics/visualization tools and the core user application. Components may be switched out independently of other aspects of the AMT RTS. The scheduler is a good example of a component that has developed over time into separate versions to take advantage of changing hardware architecture (threading and new GPU hardware). With regards to threading, this development was done independently of any applications and resulted in improved performance without significant refactoring of application codes. A developer could create new analytics tools, separate versions of application code or simply adjust which scheduler is used, by changing compilation flags that are passed to the configuration scripts. The runtime system's software stack includes very few dependencies, allowing portability to many different system architectures. The core software dependencies include pthreads, MPI (for distributed applications), and C++. There are few hardware constraints with the exception of application codes that take advantage of GPU device support. Underlying GPU support is provided by inclusion of the vendor specific C/C++ language extension CUDA. CUDA is only supported by GPU's designed and developed by NVIDIA. However, most current HPC architectures utilize NVIDIA GPU's so this fact does not provide a significant roadblock to the use of this Uintah feature. In the future, it is likely that the developers will refactor relevant Uintah components to utilize new co-processor architectures. In summary, with the exception of CUDA, Uintah has few hardware constraints and has high modularity between its components.

The modularity and abstractions inherent to the design of Uintah provide a high level of mutability within the runtime and result in easy portability of application codes *to the runtime*. Transferring an existing code from Uintah presents a developer with a bit more of a challenge because of the high degree of abstraction with regards to data management, communication and task scheduling. Essentially, an existing application code that needs to be transferred to another runtime system or implemented using baseline MPI would require a complete rewrite of the application code with the exception of the computational kernels. The runtime system provides the infrastructure for communication and data transfer, so this functionality would need to be replicated when changing runtime systems. The computational kernels themselves, that take the form of component member functions, could be easily ported to any runtime that is written in C++, but the developer would be required to setup data transfers, task scheduling and synchronization. The kernels themselves could be maintained as serialized tasks within any new runtime or be refactored using a threading directive like OpenMP to include additional parallelization. The ease of setup for a structured mesh application within Uintah necessitates that users be separated from some of the lower level details of the runtime execution code but results in the users having to replicate that functionality if they wish to port their code to a different AMT RTS.

Extracting any of the Uintah AMT RTS components (consisting of the scheduler, load balancer, task graph compiler, dependency analysis, and data warehouse) would be difficult due to their interdependence on the other components and the strong assumption that the data layout is based on a Cartesian structured grid computational mesh.

4.4.2 Interoperability With Other Languages and Libraries

The Uintah AMT RTS is implemented in C++ with recently added support for the C++11 standard. As discussed in the programmability section, user developed applications are placed within individual components, while computational kernels are defined as private member functions of those component classes. Each class is written in C++ and compiled along with other components required by the runtime system into an executable. Passing any number of configure flags to a GNU autotools configuration script generates a makefile that will compile the Uintah runtime system and all included components based on the user's specifications. A user can choose what components are included, which compilers are utilized and what libraries are linked using these configuration flags. Integration of Uintah into a custom user application can best be summarized as developing an additional class for an existing C++ application. Therefore, interoperability with other programming languages is almost entirely dependent on the requirements of the C++ language itself.

While interoperability depends mostly upon the C++ language, support for various node-level libraries is co-dependent on compiler support and compatibility with the abstractions provided by the Uintah runtime. The two main abstractions that affect node-level functionality of common libraries are the concurrency and data models. Uintah's concurrency model defines how tasks operating on a node's set of patches are distributed and its data model handles communication of relevant patch data between worker nodes. These abstractions remove management of machine-dependent concerns related to threading and memory from user applications but introduce some limitations in the compatibility with some third party libraries. As an asynchronous runtime system, Uintah supports any node-level library that does not assume a centralized memory or threading model. Libraries that require specific memory layout patterns or utilize internal threading (via pthreads or OpenMP), including some third party math libraries, are not explicitly supported in a performance portable manner. Any node-level libraries that do not fall under the category of utilizing these abstraction layers would be implicitly supported and support would depend on the compiler chosen.

Kokkos provides an interesting case study for the requirements and limitations of Uintah's programming model. The baseline Kokkos implementation generates a single execution space and therefore assumes one master thread. As previously stated, global threading and memory management are abstracted away from user applications and because Kokkos depends on this master threading model, it is currently incompatible with Uintah. Researchers at the University of Utah have demonstrated how Kokkos might work with a data parallel task by forking an existing version of Kokkos. Their work involved modifying the execution space internals in Kokkos to allow for multiple disjoint instances of that execution space. In theory this allows initialization of a distinct execution space on each Uintah worker thread. To test this implementation researchers created a simple Poisson unit test using a Kokkos functor to demonstrate a data parallel task. This branch allows the execution of Kokkos algorithms including `parallel_for`, `parallel_reduce` and `parallel_scan` within the context of a Uintah task. However, the data warehouse and grid variables need to be modified before Kokkos views and other data structures can be utilized. It is important to note that the Kokkos development team is currently working on refactoring their execution space model which will allow for compatibility with run time systems like Uintah.

GPU support within Uintah provides some insights into the current level of functionality for some fringe runtime features. The explicitly supported GPU threading model that Uintah operates over is NVIDIA's CUDA C/C++ language extension. Uintah's unified scheduler provides a unique set of queues for tasks that are scheduled to the GPU and CPU on an individual node. Additionally, a data abstraction layer is provided for GPU devices that is an extension of the existing `DataWarehouse` and aptly named the `GPUDataWarehouse`. The combined use of the unified scheduler and the GPUDataWarehouse allows the runtime system to overlap communication with computation on the GPU. To manage GPU tasks, CUDA streams are utilized along with automatic transfer of mesh data between the device and host. Programming CUDA kernels is simpler using Uintah's built-in `GPUDataWarehouse` because users do not have to explicitly allocate device memory or copy data between the host and device. Instead, the `GPUDataWarehouse` is aware of the data dependencies of device enabled tasks and automatically schedules the required data transfers. One key feature that limits the potential performance of GPU-enabled tasks is the lack of persistence of data within device memory. Instead of transferring modified data only, all variables from a set of patches must be transferred before and after a computational kernel is executed. While GPU support seems well designed, the immaturity of these features result in some bugs inherent to the runtime system. Despite its current maturity and functionality limitations, it is important to note that Uintah has the most fully fledged GPU device support of any of the runtime systemswe have tested. Uintah developers are currently working on revamping GPU support to include in-memory data persistence and flush out quirks their GPU support.

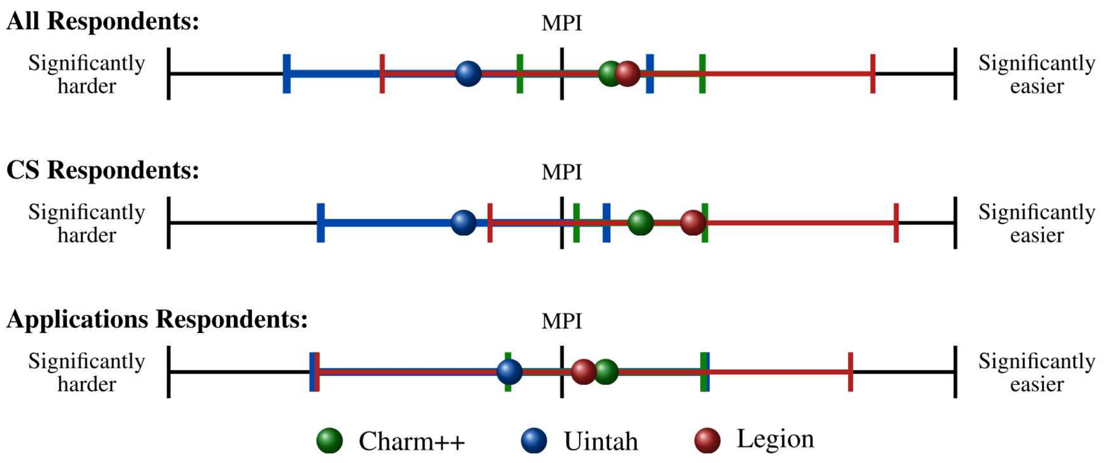
Uintah uses MPI as its transport layer and thus very easily interfaces with MPI-based libraries, applications, and solver packages, as is seen by its interface to the *hypre* [51] and the *petsc* [52] libraries.

4.5 Comparative Analysis

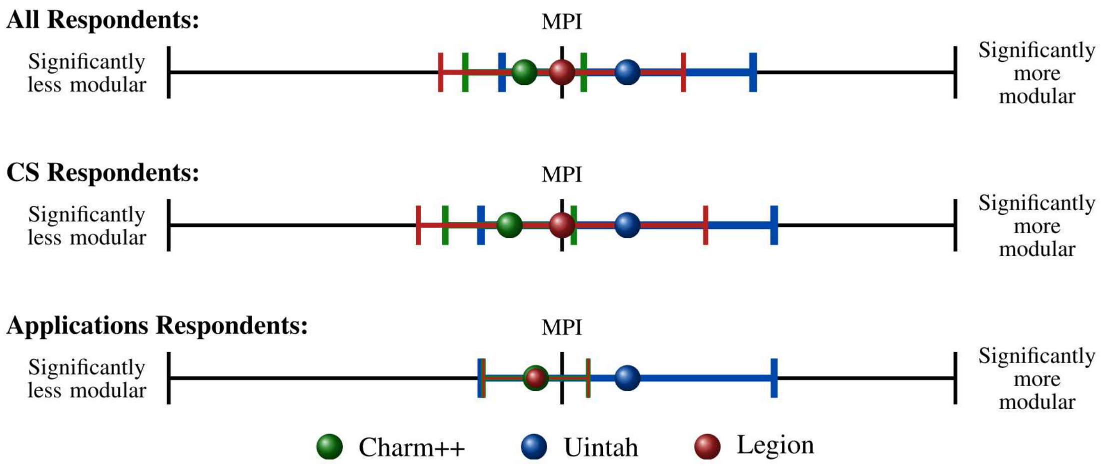
In this section we compare and contrast the three runtimes across a number of subjective mutability measures. Similar to the approach in Sections 2.5 and 3.6, we posed questions to the attendees of each bootcamp. Attendees were asked to provide a response on a seven-point scale, *relative to MPI*. For each question below, a low score indicates a response "significantly worse than MPI", a mid-range score indicates "comparable to MPI", and a high score indicates a response "significantly better than MPI". For each question, we show overall responses, along with the responses separated according to respondents with an applications (Apps) background versus respondents with a computer science (CS) background.

Not surprisingly, all respondents found a strong correlation between the maturity of the runtimes and their implementation completeness. From a design and specification perspective, the CS respondents slightly favored Uintah over Legion, with the Apps respondents, favoring Legion slightly over Uintah. All respondents agreed that Uintah was the least mutable, in large part because of its focus on structures meshes. However, the CS respondents slightly favored Legion over Charm++, with the Apps respondents favoring Charm++ as the most mutable of the runtimes tested.

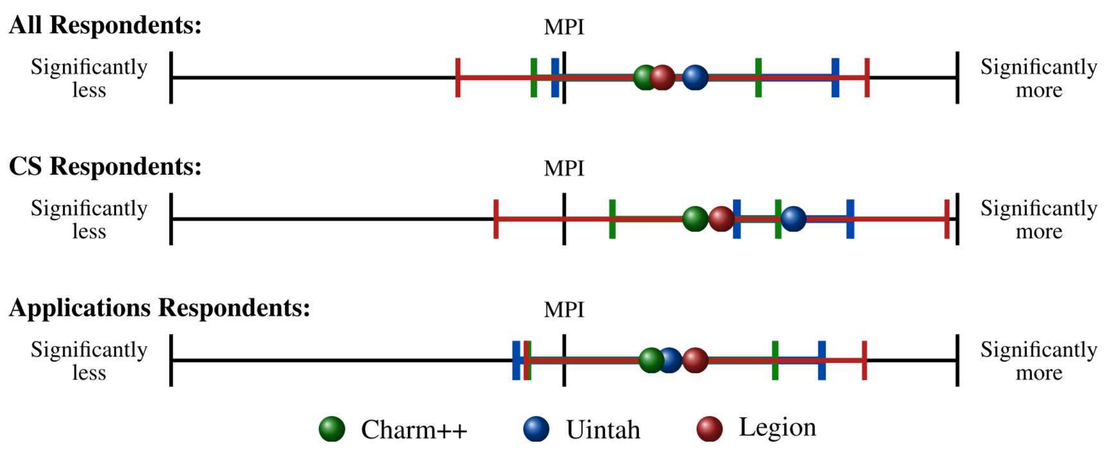
Mutability of Runtime: How much and how easily can the runtime itself be modified to suit the needs of Sandia (either on our own or with support from the runtime developers)?



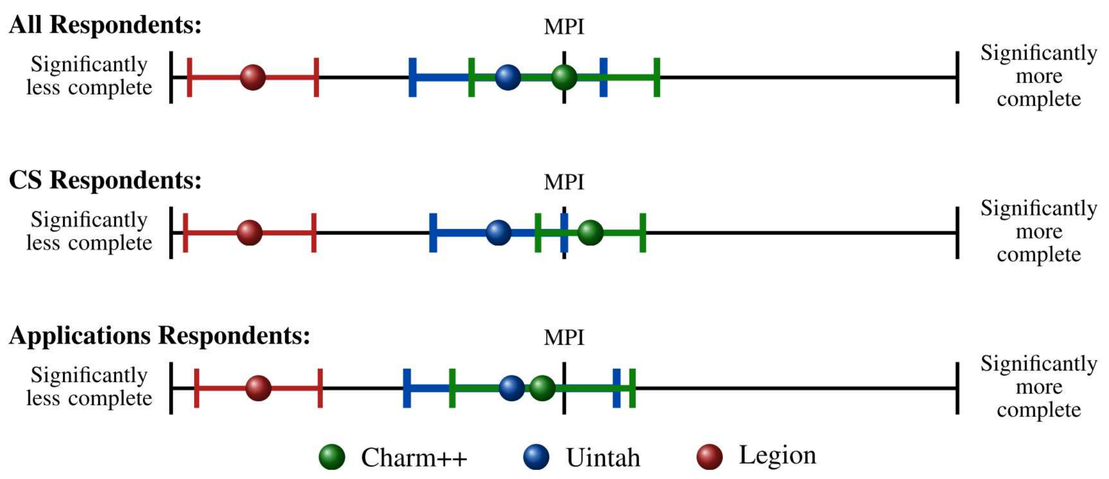
Modularity: How much do developers of separate portions of the runtime code need to communicate?



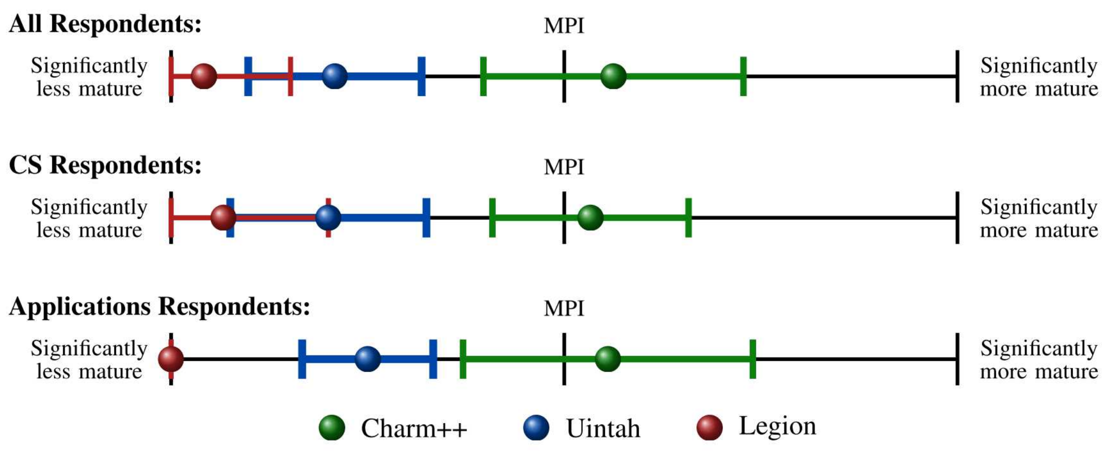
Design and Specification: How well thought out is the runtime's programming model and application programming interface specification?



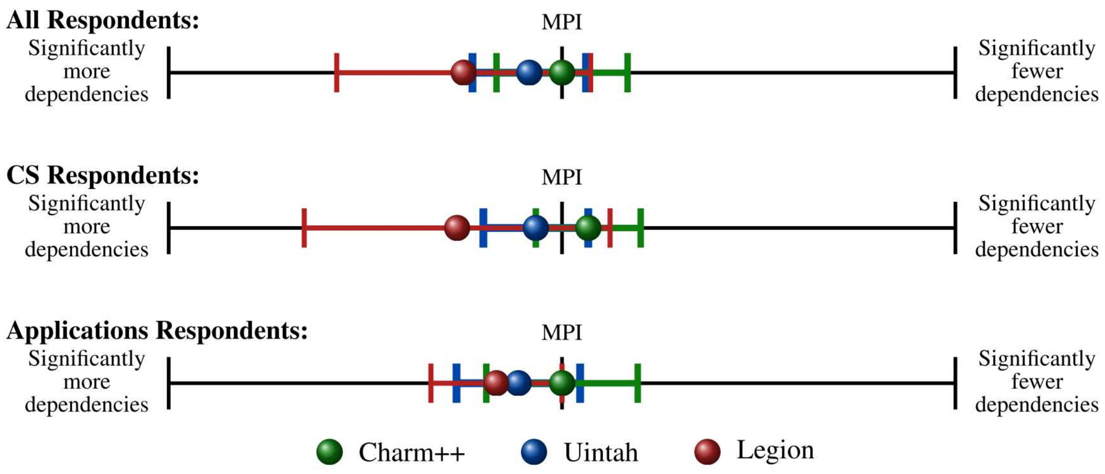
Implementation Completeness: How much of the programming model's specification is implemented and working at a production level?



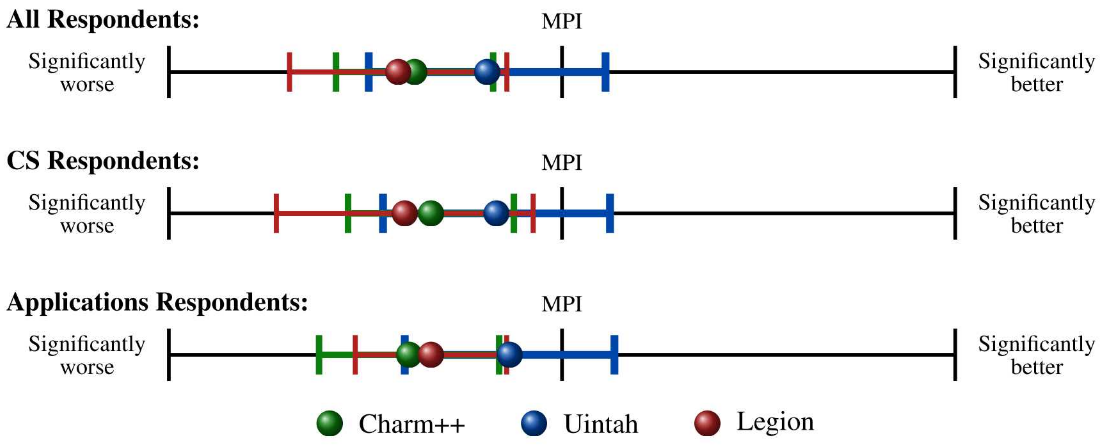
Maturity: How long has the runtime been around?



Dependency Complexity: How many dependencies on external libraries does the runtime have (software layers for network, node level thread/memory management).



Interoperability: How well does the runtime interact with existing libraries and programming models?



Chapter 5

Conclusions and Recommendations

This report presents a qualitative and quantitative examination of three best-of-class AMT runtime systems —Charm++, Legion, and Uintah, all of which are in use as part of the ASC PSAAP-II Centers. The primary aim of this report is to provide information to help create a technical road map for developing next-generation programming models and runtime systems that support ASC workload requirements. However, it is Sandia's hope that the analysis presented here serves as a catalyst to the AMT RTS community to begin working towards establishing best practices, with an eye towards eventual standards.

Each runtime in this study is evaluated with respect to three main criteria: programmability (Chapter 2), performance (Chapter 3), and mutability (Chapter 4). We reiterate here that programmability and mutability are somewhat subjective; their measures may vary over time, across laboratories, and individual application areas. Although there is a large author list on this report, the subjective opinions here reflect the views of the DHARMA team (and may not represent the views of the individual runtime teams). The evaluation of the three AMT runtimes highlights the trade-offs they have made between low-level flexibility, higher-level constructs, and domain-specific optimizations. These trade-offs are significant as they directly affect aspects of how and where concurrency is created and managed.

At one end of the spectrum lies Charm++, a highly flexible actor model. While Charm++ provides tools for managing data transfers and control flow, it essentially leaves the management of data and concurrent data accesses to the application. Charm++ has a number of strengths, with one of the most prominent strengths being its maturity. It provides a stable and feature-rich set of capabilities to help manage fault tolerance, load balancing, data transfers, and program control flow. Charm++ supports flexible communication patterns and (relatively) fine-grained dynamic parallelism via its highly portable, high-performance, asynchronous transport layer. Furthermore, it provides an extensive serialization framework to facilitate distributed memory transfers of user-defined data structures, whether for explicit transfer between objects performing work or for implicit migration to facilitate load balancing and resilience.

The flexibility of Charm++ comes at the cost of increased application-level responsibility. For example, while it provides mechanisms to facilitate control flow[1], handling data race conditions and ensuring correct data-flow is entirely up to the application. Furthermore, zero-copy data exchanges require potentially awkward application-level constructs to manage any concurrent data accesses. This shortcoming is tightly coupled to the lack of a data model that could inform the scheduler of task dependencies, granting it the ability to identify and manage additional concurrency at the runtime-system level. In addition to these issues, the current limited support for template metaprogramming due to the limitations of its control flow metalanguage[2] is an issue that hinders its widespread adoption.

Legion lies at the other end of the spectrum from Charm++: it is a relatively new (less than 5 years old) data-centric model that pushes as much data and concurrency management as possible into the runtime. This automatic handling of data movement and concurrency is its key strength, with task conflicts automatically detected, allowing the runtime to derive and manage the maximum concurrency possible expressed in the application. Legion's separation of the correctness of an application's implementation from its mapping to a target architecture is another key strength. The interface that accomplishes this separation also provides a central location for dynamic load-balancing and, potentially, fault tolerance to be managed. Legion provides a flexible relational data model that is particularly useful for expressing data ordering transformations (e.g., struct-of-arrays to array-of-structs) or index/field subsets. This has proven very useful in applications like S3D with abundant task parallelism based on field subsets.

Legion, being relatively less mature, has many features that are not fully implemented or stress tested. A notable example was the incompleteness of the map locally vs. map remotely features, which led to very poor performance

[1] e.g, the `when` construct

[2] ci files

in the distributed memory version of MiniAero. However, a more fundamental issue will potentially stem from its design choice trading off flexibility for runtime control. Transferring more responsibility for concurrency and data management into the runtime enables optimizations in certain cases, but consequently provides less flexibility and generality. In some cases, Legion nicely provides mechanisms for transferring more control back to the application, restoring some of the lost flexibility. Consider, for example, the discussion in Section 2.3.3 regarding the Legion tree-of-tasks structure not mapping naturally to support SPMD applications. The runtime *does* currently provide mechanisms to address this issue, granting the application developer the ability to take control via explicit ghosting or relaxed coherence modes. These relaxed modes for transferring concurrency control to the application are region-specific, thus their use can be isolated from other parts of the application. The Legion team also has plans for automatic SPMD transformations (see Section 2.3.3), but these capabilities are not yet complete.

In another use case however, the overheads introduced by the data model currently outweigh its associated benefits. In order for the runtime to automatically derive parallelism and manage all data transfers, all data dependencies must be expressed for every task *a priori*. Furthermore, child tasks can only operate on a subset of the data that parent tasks request, requiring a parent task to know all data the child tasks will need. If a child task detects extra data dependencies while computing, the entire task subtree must be closed and new data dependencies added. This is a notable issue for dynamic applications, such as particle-in-cell problems, where tasks cannot know where data will end up until the kernel starts running. We note the overheads introduced by the data model for supporting dynamically sized arrays and/or additional data are not present in other, lighter-weight runtime systems without a data model. Some workarounds to make dynamic data requirements scalable have been proposed (Section 2.3) wherein compiler tools or the runtime system would automatically transform explicitly serial code (parent schedules all tasks) to implicitly parallel code (scheduling distributed to children tasks), but implementations are not yet complete. Lastly, we note that interfacing with Kokkos or other libraries that manage data layout and threading would require 1) shifting Kokkos functionality into the Legion runtime or 2) making significant changes to the Legion mapper to support custom data structures (Section 4.3.2). Together, these use cases highlight that there are other places within the Legion runtime where mechanisms to relax the model by relinquishing control to the application and/or third-party libraries could be beneficial (both technically and to facilitate adoption).

Uintah's task and data management is similar to Legion's, but is fundamentally domain specific. Uintah thereby trades generality for domain-specific optimizations in managing tasks and dependencies. For the specific target applications (numerical PDEs with structured, uniform Cartesian meshes) Uintah almost seems like a DSL, making it very easy to have an initial implementation of the application. Furthermore, the Uintah API and the user-written portion of the code is very lightweight, intuitive, and easy to read and maintain. There are implementations available for a slew of multi-physics and solver algorithms that can be utilized off-the-shelf. The primary issue facing wide spread adoption of Uintah is that its target applications are too narrow for Uintah to be general purpose. It faces similar challenges to Legion when adding support for Kokkos and other node-level libraries that manage either data layout or threading.

Through the experiments and analysis presented in this report, several overarching findings emerge. From a performance perspective, AMT runtimes show tremendous potential for addressing extreme-scale challenges. Empirical studies show an AMT RTS can mitigate performance heterogeneity inherent to the machine itself[3] and that MPI and AMT runtimes perform comparably under balanced conditions. From a programmability and mutability perspective however, none of the runtimes are currently ready for use in developing production-ready Sandia ASC applications. Legion is still relatively immature and undergoing rapid development and feature addition. Uintah is targeted at Cartesian structured mesh applications, but the majority of the Sandia ASC applications use unstructured or hybrid meshes. Charm++ will require additional effort, with new abstractions as well as improved component implementations, to realize its full potential. Note that in different domains, each of the AMT runtimes have been used for production-level applications.

The runtimes studied herein each make trade-offs between higher-level constructs and low-level flexibility to strike their own balance of code performance, correctness, and programmer productivity. The trade-offs made affect aspects of how and where concurrency is created and managed. Charm++ falls on one side of the spectrum with the management of data and concurrent data accesses falling largely to the application developer. This provides tremendous flexibility, but also adds complexity in a number of application settings. At the other end of the spectrum is Legion, where the runtime assumes as much control as possible of concurrency creation and management. For performance

[3]The experiments in this report are with static workloads, however there are other studies that show the AMT RTS can mitigate performance heterogeneity inherent in the application [12–14].

reasons, there are application use cases that are not well suited to this extreme, and the Legion team has begun to introduce mechanisms to relinquish control to the application in some settings.

Based on these findings, the DHARMA team provides the following conclusions and recommendations as Sandia, and more broadly the ASC program, develops a technical roadmap for next-generation programming models and runtime systems. The findings in this report suggest that there is a critical design issue facing runtime development. Namely, should there be a single execution style for the runtime, forcing applications to accommodate and adapt, or should the runtime accommodate and adapt to several execution styles suited to many applications? A third option could involve developing several runtimes, each optimized for different application workloads and machine architectures. The community requires a significantly more comprehensive understanding of the interplay between the various AMT concurrency management schemes and their associated performance and productivity impacts (across a variety of applications and architectures) to make a confident decision regarding this design issue that will serve long term interests.

We believe this comprehensive understanding can be achieved via a concerted co-design effort between application, programming model, and runtime developers centered on common concepts and vocabulary for discussing requirements. Such a co-design approach allows for ASC application workload requirements to directly impact the design decisions of any programming model and runtime system that is adopted. We note that there are many possible ways for the application, programming model, and runtime system developers to co-design solutions. We conclude here by recommending a path forward that we believe has merit. First, we believe that co-design interactions will be greatly facilitated if application requirements are clearly articulated in terms of programming model and runtime system features. The current co-design approach of applications providing terse algorithmic descriptions along with MPI baseline mini-applications is useful but does not suffice.

We believe the best design path going forward involves the development of a runtime that can accommodate and couple a diverse range of application execution patterns, as this approach avoids the re-engineering of application-specific *ad hoc* solutions. Towards this end, developers from a representative set of application areas should work closely with programming model teams to co-design community-adopted AMT programming model abstractions that meet a set of application-driven requirements. Practically, this process could start with a programming model specification that includes an API that serves as a concrete starting point for gathering and communicating application requirements. The DHARMA team is currently working on an initial draft of such an API in collaboration with application and runtime development teams.

While a programming model specification compatible with diverse workloads seems daunting, we believe this goal is reasonable given an earnest effort within the AMT community towards best practices and eventual standards. The first step towards the development of best practices requires consensus among the AMT community regarding the vocabulary that captures the AMT design space. A common vocabulary allows requirements to be meaningfully expressed in an implementation-agnostic manner, enabling co-design interactions between many different application and runtime teams. Common vocabulary and parallel programming model abstractions within the AMT runtime community are a critical prerequisite to the establishment of best practices—and can only be achieved with the adoption or buy-in from a number of AMT RTS teams. Towards this end, the DHARMA team is working on a draft document with a proposed vocabulary and classification scheme, which we are developing based on a broad survey of existing runtimes. Our hope is that this document serves as a starting point for the dialogue and debate required for the AMT community come to consensus on a common vocabulary.

Overall, we believe this requirements-driven co-design approach benefits the HPC community as a whole, and that widespread community engagement mitigates risk for both application developers and runtime system developers and vendors. Application developers need only write their applications to a single API—that they can directly shape. Application developers further benefit from this approach as it greatly simplifies the process of comparing various AMT runtime implementations. In particular, it enables them to rapidly switch between implementations on various architectures based on performance and other considerations. From the perspective of the AMT RTS teams, this approach greatly facilitates the transition to and the adoption of AMT technologies, helping the AMT RTS teams ensure a potential long term user base for their runtime systems.

The Sandia DHARMA team is currently working collaboratively to develop the aforementioned API, and also to develop a technical road map for the implementation of an AMT RTS that satisfies this API for Sandia's ASC/ATDM program. Working first drafts of both are planned to be complete in Q1 of FY16. Interested parties from the AMT RTS community are invited to contact either of the first two authors of this report regarding participating in this effort.

Glossary

ACES Advanced Computing at Extreme Scale. 71

active message passing An Active message is a messaging object capable of performing processing on its own. It is a lightweight messaging protocol used to optimize network communications with an emphasis on reducing latency by removing software overheads associated with buffering and providing applications with direct user-level access to the network hardware. This contrasts with traditional computer-based messaging systems in which messages are passive entities with no processing power. 25

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 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. 15, 17, 23, 26, 107

ALU arithmetic logic unit. 116

AMR adaptive mesh refinement. 17, 42, 47, 81

AMT See AMT model. 3, 9, 11, 12, 16, 19, 23, 26, 52, 58, 61–64, 69–71, 75, 77, 78, 83, 85, 87, 91, 98, 107–109, 111

AMT model Asynchronous many-task (AMT) is a categorization of programming and execution models that break from the dominant CSP or SPMD models. Different 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. 3, 11, 16, 111

AMT RTS A runtime system based on AMT concepts. An AMT RTS provides a specific implementation of an AMT model. 11, 12, 16, 17, 22, 26, 97, 98, 101, 107–109, 111

anti-dependency See Write-After-Read. 25, 26, *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. 12, 17, 26, 33, 38, 39, 41, 52, 57, 58, 61, 64, 68, 69, 86–88, 98–100, 108, 109

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. 3, 4, 11, 12, 14, 16, 17, 19, 23, 37, 44, 71, 97, 107–109, 112

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. 11, 14, 16, 17, 19, 22, 23, 37, 71, 97

AVX Advanced Vector Extensions. 117

bulk synchronous The bulk synchronous model of parallel computation (BSP) is defined as the combination of three attributes: 1) A number of components, each performing processing and/or memory functions; 2) A router that delivers messages point to point between pairs of components; and 3) Facilities for synchronizing all or a subset of the components at regular intervals of L time units where L is the periodicity parameter. A computation consists of a sequence of supersteps. In each superstep, each component is allocated a task consisting of some combination of local computation steps, message transmissions and (implicitly) message arrivals from other components. After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components. If it has, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep. See Reference [28] and [103] for more details. 37, 45, 85, 87, 91, 95, 112

CFD computational fluid dynamics. 17

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. 8, 15, 23–26, 28, 57, 64, 65, 72, 83, 89, 115

CLE Cray Linux Environment. 71

coherence An input parameter within the Legion runtime that determines the types of manipulations one function can do to another function's logical region. 32–35, 37, 66

CPU central processing unit. 15, 34, 45, 47, 48, 58, 63, 75, 100, 102, 112, 115, 117

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).. 15, 16, 23, 26, 52, 111, 113

CUDA Compute Unified Device Architecture. 33, 34, 98–102

DAG A directed acyclic graph (DAG) is a directed graph with no cycles. This type of data representation is common form for representing dependencies. 18, 28, 64, 117

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. 8, 20, 21, 25

data parallelism A type of parallelism that involves carrying out a single task and/or instruction on different segments of data across many computational units. Data parallelism is best illustrated by vector processing or SIMD operations on CPUs and MICs or typical bulk synchronous parallel applications. 14, 15, 19, 44, 77, 117

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. 11, 15–17, 32, 52, 113, 114

DHARMA The DHARMA (Distributed asyncHronous Adaptive and Resilient Models for Applications) research team at Sandia is focused on next generation programming models, execution models, and runtime systems research. 11, 12, 22, 107, 109

DMA direct memory access. 116

DOE U. S. Department of Energy. 4, 17, 58, 80, 113

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. 11, 15, 17, 33, 52, 98, 108

event-based The term event-based covers both programming models and execution models in which an application is expressed and managed as a set of events with precedence constraints, often taking the form of a directed graph of event dependencies. 15, 89, 98

exascale Exascale computing refers to computing systems capable of at least one exaFLOPS, or a billion billion (10^{18}) calculations per second. Such capacity represents a thousandfold increase over the first petascale computer that came into operation in 2008. The DOE is planning to develop and deliver capable exascale computing systems by 2023-24. These systems are expected to have a one-hundred to one-thousand-fold increase in sustained performance over today's computing capabilities, capabilities critical to enabling the next-generation computing for national security, science, engineering, and large-scale data analytics. Leadership in HPC and large-scale data analytics will advance national competitiveness in a wide array of strategic sectors. An integrated government-industry-academia approach to the development of hardware, system software, and applications software, will be required to overcome the barriers of power efficiency, massive parallelism, and programmability to attain maximum benefit from exascale computers. 3, 11, 13, 19, 48, 71

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 (Write-After-Read (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. 14–17, 22, 23, 25, 26, 28, 35–37, 45, 47, 52, 57, 85, 86, 111, 113, 116

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. 15, 111, 113

functional A style of programming that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. 15, 32, 39, 115

GPU graphics processor unit. 34, 45–47, 71, 80, 99–102

HAAP Heterogeneous Advanced Architecture Platform. 71

HLR A high-level runtime is generally any aspect of the runtime system that implicitly creates concurrency via higher-level logic based on what is expressed via the application programming model. High-level runtimes generally involve data, task, and machine models expressed in a declarative fashion through which the runtime reasons about application concurrency. This implicit creation of concurrency differs from the low-level runtime (LLR), which only executes operations explicitly specified. Legion and Uintah both implement extensive HLRs while Charm++ has very little implicit behavior. 16, 114

HPC high-performance computing. 3, 12–14, 23, 27, 47, 58, 59, 61, 62, 64, 69, 77, 101, 109, 113

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. 11, 15, 16, 32, 34, 52, 115

in-situ *In-situ* analysis involves analyzing data *on site* or *in place* where it was generated, in contrast to in-transit which first migrates data to another physical location. 13, 26, 55, 87, 114

in-transit In-transit analysis is a method for performing analysis on an applications raw computational data while the application is running by offloading the simulation data to a set of processing units allocated for data analytics. Typically, this method involves more network communication and requires a balance between the compute hardware running the application and analysis but allows an application to resume its computations faster. This contrasts with *in-situ* analysis that operates on data in-place. 114

LANL Los Alamos National Laboratory. 4, 71

LLNL Lawrence Livermore National Laboratory. 62, 63, 69

LLR A low-level runtime is generally any aspect of a runtime system that manages explicitly specified data movement and task scheduling operations. There is very little implicit behavior. The runtime is only responsible for ensuring that events and operations satisfy explicit precedence constraints. This contrasts with a high-level runtime (HLR) that implicitly creates parallelism from a declarative program, converting higher-level program logic into explicit operations in the LLR. 16, 114

load balancing Load balancing distributes workloads across multiple computing resources. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource. Using multiple components with load balancing instead of a single component may increase reliability and availability through redundancy. 19, 25–27, 36, 37, 41, 46, 48, 58, 61, 64, 72, 75, 81, 86, 91, 107

logical region A collection of objects operated on by a task. Various parameters define the behavior of logical region when operated on by a given task including privilege, coherence and behavior. 15, 32–35, 37, 39, 41, 66, 72, 86, 90

MIC Intel Many Integrated Core Architecture or Intel MIC is a coprocessor computer architecture developed by Intel incorporating earlier work on the Larrabee many core architecture, the Teraflops Research Chip multicore chip research project, and the Intel Single-chip Cloud Computer multicore microprocessor. Prototype products codenamed Knights Ferry were announced and released to developers in 2010. The Knights Corner product was announced in 2011 and uses a 22 nm process. A second generation product codenamed Knights Landing using a 14 nm process was announced in June 2013. Xeon Phi is the brand name used for all products based on the Many Integrated Core architecture. 15, 47, 112

MPI Message Passing Interface. 3, 9, 11, 12, 15–17, 19, 23, 25–27, 32, 33, 37, 38, 44–46, 52, 57, 59, 63, 64, 68–72, 74, 75, 77, 78, 80, 85, 86, 88, 89, 91, 95, 97–102, 108, 109, 111, 114–117

MPI+X A hybrid programming model combining MPI and another parallel programming model in the same application. The combination may be mixed in the same source or combinations of components or routines, each of which is written in a single parallel programming model. MPI+Threads or MPI+OpenMP are the most common hybrid models involving MPI. MPI describes the parallelism between processes (with separate memory address spaces) and the "X" typically provides parallelism within a process (typically with a shared-memory model). 16, 26, 45, 69, 86, 108

MPMD The term multiple-program multiple-data (MPMD) refers to a parallel programming model where tasks operate on disjoint data like SPMD, but are not constrained to perform the same tasks. 15

multi-level memory A hybrid memory system that integrates multiple types of memory components with different sizes, bandwidths, and access methods. There may be two or more levels with each level composed of a different memory technology, such as NVRAM, DRAM, 3D Stacked, or other memory technologies. This is an extension of the L1, L2, and L3 cache memory systems of current CPU architectures. As a result, future application analysis must account for complexities created by these multi-level memory systems with or without coherency. Despite the increased complexity, the performance benefits of such a system should greatly outweigh the additional burden in programming brought by multi-level memory. For instance, the amount of data movement will be reduced both for cache memory and scratch space resulting in reduced energy consumption and greater performance [1]. 13

NERSC National Energy Research Scientific Computing Center. 4

NNSA National Nuclear Security Administration. 4, 71, 116

NTV near-threshold voltage. 13

NUMA non-uniform memory access. 46, 71, 72, 116

Parameter Marshalling The process of transferring an object within Charm++ that is required for a read/write. 23, 25

patch A unit of data within a structured mesh involved with discretizing workloads into data-parallel segments. Data segments takes the form of cells that can contain particles and/or member data. As the basic unit of parallel work, Uintah uses computations in the form of tasks over a single patch to express parallelism. 44–46, 48, 68, 72, 80, 81, 90, 102

PDE partial differential equation. 42, 108

pipeline parallelism Pipeline parallelism is achieved by breaking up a task into a sequence of individual sub-tasks, each of which represents a stage whose execution can be overlapped. 15, 75

POD In C++, POD stands for Plain Old Data—that is, a class or struct without constructors, destructors and virtual members functions and all data members of the class are also POD. 90, 91

privilege A Legion parameter that defines the side-effects a task will have on a given logical region. These side-effects could include read, write or reduction permissions. 32–35, 37, 65, 66, 80, 86

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. 11, 15–17, 32, 115

processing in memory Processing in memory (PIM) is the concept of placing computation capabilities directly in memory. The PIM approach can reduce the latency and energy consumption associated with moving data back-and-forth through the cache and memory hierarchy, as well as greatly increasing memory bandwidth by sidestepping the conventional memory-package pin-count limitations. 13

programming language A programming language is a syntax and code constructs for implementing one or more programming models. For example, the C++ programming language supports both functional and procedural imperative programming models. 15, 99, 101

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. 3, 11, 12, 14–17, 22, 23, 25, 26, 28, 32, 33, 35, 37, 39, 41, 45, 52–56, 59, 64, 65, 69, 74, 85–90, 93, 98, 102, 104, 105, 107, 109, 111, 113–115, 117

PSAAP-II The primary goal of the 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. 3, 11, 47, 107

PUP The Pack-UnPack framework (PUP) is a serialization interface within Charm++ that allows programmers to tell the runtime how objects are marshaled when required. 25, 52

RAW Read-After-Write. 113, 116

RDMA Remote direct memory access (RDMA) is a direct memory access from the memory of one computer into that of another without involving either one's operating system. This permits high-throughput, low-latency networking, which is especially useful in massively parallel computing. 118

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. 26, 112, 113, 116

remote procedure invocation See RPC. 11, 15, 17, *Glossary:* RPC

RMCRT Reverse Monte Carlo Ray Tracing. 47

RPC Remote Procedure Call (RPC) is a protocol that one program can use to request a service from a program located in another computer in a network without having to understand network details. RPC uses the client/server model. 25

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. 3, 8, 11, 12, 14, 16, 17, 19, 22–27, 32, 33, 35, 37–39, 42, 47, 48, 58, 61, 62, 64, 65, 69–71, 74, 83, 85, 92, 93, 98, 99, 101, 102, 107–109, 113, 114

scratchpad Scratchpad memory, also known as scratchpad, scratchpad RAM or local store in computer terminology, is a high-speed internal memory used for temporary storage of calculations, data, and other work in progress. In reference to a microprocessor, scratchpad refers to a special high-speed memory circuit used to hold small items of data for rapid retrieval. It can be considered similar to the L1 cache in that it is the next closest memory to the arithmetic logic unit (ALU) after the internal registers, with explicit instructions to move data to and from main memory, often using direct memory access (DMA)-based data transfer. In contrast to a system that uses caches, a system with scratchpads is a system with NUMA latencies, because the memory access latencies to the different scratchpads and the main memory vary. Another difference from a system that employs caches is that a scratchpad commonly does not contain a copy of data that is also stored in the main memory. 13

SIMD The term single-instruction multiple-data (SIMD) refers to a type of instruction level parallelism where an individual instruction is synchronously executed on different segments of data. This type of data parallelism is best illustrated by vector processing. 14, 15, 112, 117

SNL Sandia National Laboratories. 4, 62, 71

SoC A system on a chip or system on chip (SoC or SOC) is an integrated circuit (IC) that integrates all components of a computer or other electronic system into a single chip. It may contain digital, analog, mixed-signal, and often radio-frequency functions–all on a single chip substrate. The System on Chip approach enables HPC chip designers to include features they need, and exclude features that are not required in a manner that is not feasible with today's commodity board-level computing system design. SoC integration is able to further reduce power, increase integration density, and improve reliability. It also enables designers to minimize off-chip I/O by integrating peripheral functions, such as network interfaces and memory controllers by integrating the components onto a single chip. 13

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. 5, 14–16, 19, 35, 37, 38, 75, 77, 108, 111–113, 115

SSE Streaming SIMD Extensions. 117

task parallelism A type of parallelism that focuses on completing multiple tasks simultaneously over different computational units. These tasks may operate on the same segment of data or many different datasets. 15, 27, 37, 45

task stealing See work stealing. 36, *Glossary:* work stealing

task-DAG A use of a directed acyclic graph (DAG) that represents tasks as nodes and directed lines as dependencies of a particular task/data segment. These graphs have no cycles, they do not represent iteration within a program. 18, 89

vector processing A vector processing is performed by a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data called vectors, compared to scalar processors, whose instructions operate on single data items. Vector processing can greatly improve performance on certain workloads, notably numerical simulation and similar tasks. Vector machines appeared in the early 1970s and dominated supercomputer design through the 1970s into the 1990s, notably the various Cray platforms. As of 2015 most commodity CPUs implement architectures that feature instructions for a form of vector processing on multiple (vectorized) data sets, typically known as SIMD. Common examples include MMX, Streaming SIMD Extensions (SSE), AltiVec and Advanced Vector Extensions (AVX). 15, 112, 117

WAR Write-After-Read. 113, 117

WAW Write-After-Write. 113, 117

work stealing The act of one computational unit (thread/process), which has completed it's workload, taking some task/job from another computational unit. This is a basic method of distributed load balancing. 36, 42, 81

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. 26, 111–113, 117

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. 112, 113, 117

zero-copy Zero-copy transfers are data transfers that occur directly from send to receive location without any additional buffering. Data is put immediately on the wire on the sender side and stored immediately in the final receive buffer off the wire on the receiver side. This usually leverages remote direct memory access (RDMA) operations on pinned memory. 25, 98, 100, 107

References

[1] J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. T. Stark, D. Unat, and N. J. Wright, “Abstract machine models and proxy architectures for exascale computing,” in *Co-HPC@SC*. IEEE, 2014, pp. 25–32. [Online]. Available: http://dl.acm.org/citation.cfm?id=2689669

[2] R. Stevens, A. White, S. Dosanjh, A. Geist, B. Gorda, K. Yelick, J. Morrison, H. Simon, J. Shalf, J. Nichols, and M. Seager, “Architectures and technology for extreme scale computing,” U. S. Department of Energy, Tech. Rep., 2009. [Online]. Available: http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Arch_tech_grand_challenges_report.pdf

[3] S. Ahern, A. Shoshani, K.-L. Ma, A. Choudhary, T. Critchlow, S. Klasky, V. Pascucci, J. Ahrens, E. W. Bethel, H. Childs, J. Huang, K. Joy, Q. Koziol, G. Lofstead, J. S. Meredith, K. Moreland, G. Ostrouchov, M. Papka, V. Vishwanath, M. Wolf, N. Wright, and K. Wu, *Scientific Discovery at the Exascale, a Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization*, 2011. [Online]. Available: http://science.energy.gov/~/media/ascr/pdf/program-documents/docs/Exascale-ASCR-Analysis.pdf

[4] Scientific Computing and Imaging Institute, The University of Utah. Overview of the uintah software. [Online]. Available: http://ccmscwiki.sci.utah.edu/w/images/e/eb/UINTAH_OVERVIEW.pdf

[5] K. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, “Combing the communication hairball: Visualizing parallel execution traces using logical time,” *Visualization and Computer Graphics, IEEE Transactions on*, vol. 20, no. 12, pp. 2349–2358, Dec 2014.

[6] 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*, 1993, pp. 91–108.

[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*, 2012, pp. 1–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2389086

[8] J. D. D. S. Germain, S. G. Parker, C. R. Johnson, and J. McCorquodale, “Uintah: a massively parallel problem solving environment,” 2000. [Online]. Available: http://content.lib.utah.edu/u?/ir-main,29551

[9] 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*, 2003, pp. 193–208. [Online]. Available: https://parasol.tamu.edu/publications/download.php?file_id=663

[10] T. Heller, H. Kaiser, and K. Iglberger, “Application of the parallex execution model to stencil-based problems,” *Comput. Sci.*, vol. 28, pp. 253–261, 2013.

[11] E. A. Luke, “Loci: A deductive framework for graph-based algorithms,” in *Computing in Object-Oriented Parallel Environments (3rd ISCOPE’99)*, ser. Lecture Notes in Computer Science (LNCS), S. Matsuoka, R. R. Oldehoeft, and M. Tholburn, Eds. San Francisco, California, USA: Springer-Verlag (New York), Dec. 1999, vol. 1732, pp. 142–153.

[12] J. Luitjens and M. 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)*, Atlanta, Georgia, USA, Apr. 2010, pp. 1–10.

[13] J. C. Phillips, Y. Sun, N. Jain, E. J. Bohm, and L. 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.

[14] H. Menon, L. Wesolowski, G. Zheng, P. Jetley, L. Kale, T. Quinn, and F. Governato, “Adaptive techniques for clustered n-body cosmological simulations,” *Computational Astrophysics and Cosmology*, vol. 2, no. 1, pp. 1–16, 2015.

[15] H. Kaul, M. Anders, S. Hsu, A. Agarwal, R. Krishnamurthy, and S. Borkar, “Near-threshold voltage (NTV) design—opportunities and challenges,” in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, June 2012, pp. 1149–1154.

[16] P. V. Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*. Cambridge, MA, USA: MIT Press, 2004.

[17] W. Gropp, “MPI at exascale: Challenges for data structures and algorithms,” in *Proceedings of the 16th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 3–3. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_3

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” *SIGPLAN Notices*, vol. 30, pp. 207–216, 1995.

[19] L. Dagum and R. Menon, “OpenMP: an industry standard API for shared-memory programming,” *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[20] K. Fatahalian *et al.*, “Sequoia: Programming the memory hierarchy,” in *Supercomputing (SC)*, November 2006.

[21] K. Franko, “MiniAero: A performance portable mini-application for compressible fluid dynamics,” in preparation.

[22] K. J. Franko, T. C. Fisher, P. Lin, and S. 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.

[23] “Mantevo.” [Online]. Available: http://mantevo.org/

[24] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202 – 3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731514001257

[25] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Structure slicing: Extending logical regions with fields,” in *Supercomputing (SC)*, 2014. [Online]. Available: http://legion.stanford.edu/pdfs/legion-fields.pdf

[26] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, “Parallel programming with migratable objects: Charm++ in practice,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC ’14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 647–658. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.58

[27] C. Hewitt, P. Bishop, and R. Steiger, “A universal modular actor formalism for artificial intelligence,” in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, ser. IJCAI’73. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=1624775.1624804

[28] L. G. Valiant, “A bridging model for parallel computation,” *Commun. ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990. [Online]. Available: http://doi.acm.org/10.1145/79173.79181

[29] "The Charm++ parallel programming system manual," accessed: 2015-08-23. [Online]. Available: http://charm.cs.illinois.edu/manuals/html/charm++/manual-1p.html

[30] "Converse programming manual," accessed: 2015-08-23. [Online]. Available: http://charm.cs.illinois.edu/manuals/html/converse/manual-1p.html

[31] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," LLNL, Tech. Rep. 782261, September 2014. [Online]. Available: https://e-reports-ext.llnl.gov/pdf/782261.pdf

[32] L. Kale, "The Chare Kernel parallel programming language and system," in *Proceedings of the International Conference on Parallel Processing*, vol. II, Aug. 1990, pp. 17–25.

[33] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *J. Comput. Chem.*, vol. 26, no. 16, pp. 1781–1802, 2005. [Online]. Available: http://dx.doi.org/10.1002/jcc.20289

[34] F. Gioachin, P. Jetley, C. L. Mendes, L. V. Kale, and T. R. Quinn, "Toward petascale cosmological simulations with changa," Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. 07-08, 2007.

[35] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[36] R. V. Vadali, Y. Shi, S. Kumar, L. V. Kale, M. E. Tuckerman, and G. J. Martyna, "Scalable fine-grained parallelization of plane-wave-based ab initio molecular dynamics for large supercomputers," *Journal of Computational Chemistry*, vol. 25, no. 16, pp. 2006–2022, Oct. 2004.

[37] H. Menon and L. Kalé, "A distributed dynamic load balancer for iterative applications," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '13. New York, NY, USA: ACM, 2013, pp. 15:1–15:11. [Online]. Available: http://doi.acm.org/10.1145/2503210.2503284

[38] E. Meneses, X. Ni, G. Zheng, C. Mendes, and L. Kale, "Using migratable objects to enhance fault tolerance schemes in supercomputers," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 7, pp. 2061–2074, July 2015.

[39] E. Meneses and L. V. Kale, "A Fault-Tolerance Protocol for Parallel Applications with Communication Imbalance," in *Proceedings of the 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Santa Catarina, Brazil, October 2015.

[40] E. M. Jonathan Lifflander, H. Menon, P. Miller, S. Krishnamoorthy, and L. Kale, "Scalable Replay with Partial-Order Dependencies for Message-Logging Fault Tolerance," in *Proceedings of IEEE Cluster 2014*, Madrid, Spain, September 2014.

[41] Y. Sun, J. Lifflander, and L. V. Kale, "PICS: A Performance-Analysis-Based Introspective Control System to Steer Parallel Applications," in *Proceedings of 4th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2014*, Munich, Germany, June 2014.

[42] MPI Forum, "MPI: A message-passing interface standard: Version 2.1," 2008.

[43] D. Bonachea, "GASNet specification, V1.1," 2002. [Online]. Available: http://gasnet.lbl.gov/CSD-02-1207.pdf

[44] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads Programming*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[45] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010. [Online]. Available: http://dx.doi.org/10.1109/MCSE.2010.69

[46] J. Nickolls, I. Buck, M. Garland, and K. Skadron, “Scalable parallel programming with CUDA,” *Queue*, vol. 6, pp. 40–53, 2008.

[47] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken, “Regent: A high-productivity programming language for hpc with logical regions,” in *Supercomputing (SC)*, 2015. [Online]. Available: http://legion.stanford.edu/pdfs/regent2015.pdf

[48] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo, “Terascale direct numerical simulations of turbulent combustion using S3D,” *Computational Science and Discovery*, p. 015001, 2009.

[49] S. Treichler, M. Bauer, and A. Aiken, “Language support for dynamic, hierarchical data partitioning,” in *Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013. [Online]. Available: http://legion.stanford.edu/pdfs/oopsla2013.pdf

[50] “Legion bootcamp: Data model,” 2014. [Online]. Available: http://legion.stanford.edu/pdfs/bootcamp/03_regions.pdf

[51] R. Falgout, T. Kolevr, J. Schroder, P. Vassilevski, U. M. Yang, A. Baker, C. Baldwin, E. Chow, N. Elliott, V. E. Henson, E. Hill, D. Hysom, J. Jones, M. Lambert, B. Lee, J. Painter, C. Tong, T. Treadway, and D. Walker. hypre Web page. [Online]. Available: http://computation.llnl.gov/project/linear_solvers/software.php

[52] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, and H. Zhang, “PETSc Web page,” http://www.mcs.anl.gov/petsc, 2015. [Online]. Available: http://www.mcs.anl.gov/petsc

[53] Q. Meng, A. Humphrey, and M. Berzins, “The Uintah framework: A unified heterogeneous task scheduling and runtime system,” in *Supercomputing (SC)*, 2012.

[54] J. Luitjens and M. 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)*, Atlanta, Georgia, USA, Apr. 2010, pp. 1–10.

[55] A. Humphrey, T. Harman, M. Berzins, and P. Smith, “A scalable algorithm for radiative heat transfer using reverse monte carlo ray tracing,” in *High Performance Computing*, ser. Lecture Notes in Computer Science, J. M. Kunkel and T. Ludwig, Eds. Springer International Publishing, 2015, vol. 9137, pp. 212–230. [Online]. Available: http://www.sci.utah.edu/publications/Hum2015a/humphrey_isc2015.pdf

[56] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, “Radiation modeling using the uintah heterogeneous cpu/gpu runtime system,” in *Proceedings of the first conference of the Extreme Science and Engineering Discovery Environment (XSEDE’12)*, no. 4, 2012, pp. 4:1–4:8. [Online]. Available: http://www.sci.utah.edu/publications/humphrey12/Humphrey_uintah_xsede12_gpu.pdf

[57] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, “Preliminary experiences with the uintah framework on intel xeon phi and stampede,” in *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery (XSEDE 2013)*, 2013, pp. 48:1–48:8. [Online]. Available: http://www.sci.utah.edu/publications/meng13/Meng_XSEDE2013.pdf

[58] J. K. Holmen, A. Humphrey, and M. Berzins, *Exploring Use of the Reserved Core*, 2015, vol. 2.

[59] MPI Forum’s Fault Tolerance working group. User Level Failure Mitigation. [Online]. Available: http://fault-tolerance.org/ulfm/

[60] University of Tennessee, Knoxville. Performance application programming interface. [Online]. Available: http://icl.cs.utk.edu/papi

[61] P. J. Mucci, S. Browne, C. Deane, and G. Ho, “PAPI: a portable interface to hardware performance counters,” in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[62] Intel Corporation. Intel® VTune™Amplifier 2015. [Online]. Available: https://software.intel.com/en-us/intel-vtune-amplifier-xe

[63] Cray Inc. Cray Performance Measurement and Analysis Tools. [Online]. Available: http://docs.cray.com

[64] Krell Institute. Open|SpeedShop overview. [Online]. Available: https://openspeedshop.org

[65] S. S. Shende and A. D. Malony, "The Tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 2006. [Online]. Available: http://dx.doi.org/10.1177/1094342006064482

[66] ParaTools, Inc. TAU Performance System. [Online]. Available: http://www.paratools.com/Products

[67] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.1553

[68] P. C. Roth, D. C. Arnold, and B. P. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 21–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050172

[69] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *High Performance Computing Applications*, vol. 13, no. 2, pp. 277–288, Fall 1999.

[70] GWT-TUD GmbH. Vampir 8.5. [Online]. Available: http://vampir.eu

[71] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The Spack package manager: Bringing order to hpc software chaos," to appear in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '15. Piscataway, NJ, USA: IEEE Press, 2015.

[72] A. Giménez, T. Gamblin, B. Rountree, A. Bhatele, I. Jusufi, P.-T. Bremer, and B. Hamann, "Dissecting on-node memory access performance: A semantic approach," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 166–176. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.19

[73] A. Giménez. Mitos. [Online]. Available: https://github.com/scalability-llnl/Mitos

[74] M. Schulz, A. Bhatele, D. Böhme, P.-T. Bremer, T. Gamblin, A. Gimenez, and K. Isaacs, "A Flexible Data Model to Support Multi-domain Performance Analysis," in *Tools for High Performance Computing 2014*, C. Niethammer, J. Gracia, A. Kaünpfer, M. M. Resch, and W. E. Nagel, Eds. Springer International Publishing, 2015, pp. 211–229. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16012-2_10

[75] L. Kalé and A. Sinha, "Projections: A preliminary performance tool for charm," in *Parallel Systems Fair, International Parallel Processing Symposium*, Newport Beach, CA, April 1993, pp. 108–114.

[76] K. Isaacs, A. Bhatele, J. Lifflander, D. Böhme, T. Gamblin, M. Schulz, B. Hamann, and P.-T. Bremer, "Recovering logical structure from charm++ event traces," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015.

[77] gperftools. [Online]. Available: https://github.com/gperftools/gperftools

[78] J. Davison de St. Germain, A. Morris, S. G. Parker, A. D. Malony, and S. Shende, "Performance analysis integration in the Uintah software development cycle," *International Journal of Parallel Programming*, vol. 31, no. 1, pp. 35–53, 2003. [Online]. Available: http://dx.doi.org/10.1023/A%3A1021786219770

[79] RogueWave Software. Totalview debugger. [Online]. Available: http://www.roguewave.com/products-services/totalview

[80] A. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT: An OpenMP tools application programming interface for performance analysis," in *OpenMP in the Era of Low Power Devices and Accelerators*, ser. Lecture Notes in Computer Science, A. P. Rendell, B. M. Chapman, and M. S. Müller, Eds. Springer Berlin Heidelberg, 2013, vol. 8122, pp. 171–185. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40698-0_13

[81] S. Biersdorff, C. W. Lee, A. D. Malony, and L. V. Kale, "Integrated Performance Views in Charm ++: Projections Meets TAU," in *Proceedings of The 38th International Conference on Parallel Processing (ICPP)*, Vienna, Austria, September 2009, pp. 140–147.

[82] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using projections performance analysis tool," in *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, vol. 22, no. 3, February 2006, pp. 347–358.

[83] R. Jyothi, O. S. Lawlor, and L. V. Kale, "Debugging support for Charm++," in *PADTAD Workshop for IPDPS 2004*. IEEE Press, 2004, p. 294.

[84] F. Gioachin, G. Zheng, and L. V. Kalé, "Debugging Large Scale Applications in a Virtualized Environment," in *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*, no. 10-11, Houston, TX (USA), October 2010.

[85] J. Laros, P. Pokorny, and D. DeBonis, "PowerInsight - a commodity power measurement capability," in *Green Computing Conference (IGCC), 2013 International*, June 2013, pp. 1–6.

[86] C. R. Ferenbaugh, "PENNANT: An unstructured mesh mini-app for advanced architecture research," *Concurrency and Computation: Practice and Experience*, 2014.

[87] Titan supercomputer at oak ridge leadership computing facility. [Online]. Available: https://www.olcf.ornl.gov/titan/

[88] "OpenAtom," http://charm.cs.uiuc.edu/OpenAtom/research.shtml.

[89] J.-S. Yeom, A. Bhatele, K. R. Bisset, E. Bohm, A. Gupta, L. V. Kale, M. Marathe, D. S. Nikolopoulos, M. Schulz, and L. Wesolowski, "Overcoming the scalability challenges of epidemic simulations on blue waters," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '14. IEEE Computer Society, May 2014.

[90] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing failures in exascale computing," *International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[91] A. Hassani, A. Skjellum, and R. Brightwell, "Design and evaluation of FA-MPI, a transactional resilience scheme for non-blocking mpi," in *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, June 2014, pp. 750–755.

[92] K. Teranishi and M. A. Heroux, "Toward local failure local recovery resilience model using MPI-ULFM," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 51:51–51:56. [Online]. Available: http://doi.acm.org/10.1145/2642769.2642774

[93] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 895–906. [Online]. Available: http://dx.doi.org/10.1109/SC.2014.78

[94] T. Ikegami, H. Nakada, A. Takefusa, R. Takano, and Y. Tanaka. The Falanx project. [Online]. Available: https://sites.google.com/site/spfalanx/

[95] Parallel Programming Laboratory, University of Illinois Urbana-Champaign. Fault Tolerance Support in Charm++. [Online]. Available: http://charm.cs.illinois.edu/research/ft

[96] X. Ni, E. Meneses, N. Jain, and L. V. Kale, "ACR: Automatic Checkpoint/Restart for Soft and Hard Error Protection," *SC-SuperComputing*, 2013.

[97] "Charm++ tutorial - liveviz," accessed: 2015-09-15. [Online]. Available: https://charm.cs.illinois.edu/tutorial/LiveViz.htm

[98] I. Dooley, C. W. Lee, and L. Kale, "Continuous performance monitoring for large-scale parallel applications," in *16th annual IEEE International Conference on High Performance Computing (HiPC 2009)*, December 2009.

[99] L. V. Kale, M. Bhandarkar, R. Brunner, and J. Yelon, "Multiparadigm, Multilingual Interoperability: Experience with Converse," in *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, ser. Lecture Notes in Computer Science, March 1998.

[100] N. Jain, A. Bhatele, J.-S. Yeom, M. F. Adams, F. Miniati, C. Mei, and L. V. Kale, "Charm++ & MPI: Combining the best of both worlds," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '15. IEEE Computer Society, May 2015.

[101] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *PACT 2014: 23rd International Conference on Parallel Architectures and Compilation*, 2014, pp. 263–276.

[102] M. Bauer, "Legion: Programming distributed heterogeneous architectures with logical regions," Ph.D. dissertation, Stanford University, 2014. [Online]. Available: http://legion.stanford.edu/pdfs/bauer_thesis.pdf

[103] Bulk synchronous parallel. [Online]. Available: https://en.wikipedia.org/wiki/Bulk_synchronous_parallel

DISTRIBUTION:

,

1	MS 0899	Technical Library, 8944 (electronic copy)

Sandia National Laboratories