ENHANCING ASYNCHRONOUS MANY-TASK RUNTIME SYSTEMS FOR NEXT-GENERATION ARCHITECTURES AND EXASCALE SUPERCOMPUTERS

by

Damodar Sahasrabudhe

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2021

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Damodar Sahasrabudhe**

has been approved by the following supervisory committee members:

Martin Berzins ,	Chair(s)	**10/27/2021** Date Approved
Mary W. Hall ,	Member	**11/4/2021** Date Approved
Hari Sundar ,	Member	**10/28/2021** Date Approved
Robert Michael Kirby II ,	Member	**11/1/2021** Date Approved
Sivasankaran Rajamanickam ,	Member	**10/27/2021** Date Approved

by **Mary W. Hall** , Chair/Dean of

the Department/College/School of **Computing**

and by **David B. Kieda** , Dean of The Graduate School.

ABSTRACT

Exascale supercomputers capable of computing 10^{18} double-precision floating point operations per second are expected to be operational around 2022/23. The complexity and diversity of the proposed exascale machines pose new challenges for the software applications, namely, 1) implementing efficient data management; 2) having programming systems to exploit locality and multimillion parallelism; 3) developing efficient algorithms to leverage new architectures; 4) ensuring resiliency; and 5) improving scientific productivity on diverse architectures. Due to data-driven scheduling and asynchronous execution, Asynchronous Many-Task (AMT) runtime systems show promise to handle these exascale challenges.

One such AMT, the Uintah Computational Framework, maintains two distinct layers for the application and underlying runtime infrastructure. This distinction allows Uintah users to concentrate on application and the Uintah infrastructure handles communication, data coherency, multithreading, and architecture-specific complexities.

This dissertation addresses some of the exascale challenges and also integrates the individual solutions under the single umbrella of Uintah. The resiliency approach handles node failure faster than the traditional checkpointing method and helps to address challenge (4). A potential solution for challenges (2) and (3) can be the new asynchronous scheduler designed for the Sunway Taihulight supercomputer that shows the benefits of asynchronous execution. The novel portable Single Instruction Multiple Data (SIMD) primitive provides a prospective approach to handle (2) and (5), which achieves near-ideal vectorization on Central Processing Units (CPUs) along with Graphics Processing Unit (GPU) portability provided by the CUDA back end. The newly developed threading model using MPI endpoints shows performance improvements over the MPI-everywhere version, which can be one of the solutions to tackle challenges (2) and (3). Finally, this work enhances the heterogeneous scheduler, contributes to the ongoing portability drive, and successfully runs a simulation using *portable* AMT tasks on thousands of CPUs and GPUs. These

enhancements are important to answer challenges (2), (3), and (5). As a result, this research takes Uintah closer to exascale readiness. Using Uintah as an example, this work demonstrates how AMTs, third-party libraries, and applications can be enhanced to benefit from the next-generation architectures.

Dedicated to my parents.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank the Uintah team, Scientific Computing and Imaging (SCI) Institute, and fellow researchers/students for their help, specifically, John Holmen, Bradley Peterson, Alan Humphrey, John Schmidt, Jeremy Thornock, Daniel Sunderland, Eric Phipps, Rohit Zambre, Ashok Jallepalli, T. A. J. Ouermi, Will Usher, and Harshitha Parnandi. Special thanks to John Holmen and Anupama Goparaju, without whose help this work would have taken much longer. I owe a debt of gratitude to my adviser, Prof. Martin Berzins, for his guidance, support, and the highest research standards. I am thankful to all the committee members for all I have learned from them. I am very much grateful to Aniruddha Marathe, who inspired me to pursue the Ph.D. I am also thankful to Christine Pickett for helping improve my writing skills. I am profoundly indebted to Matrusri Dandi Vimalamma Garu, Prof. Ramesh Goel, and his wife for the spiritual guidance and Almighty for the blessings and grace. Finally, I thank my family, extended family, and friends for their support and patience. Special thanks to Makarand Kelkar, Bhushana Karandikar, Swati Kelkar, Saumitra Kholgade, all Kelkars, and all Goparajus for looking after my family in my absence.

Funding Acknowledgements:

I am extremely grateful to all the funding agencies and the computational resource providers. This work was funded by: 1) National Science Foundation under grant number 1337145; 2) The Department of Energy, National Nuclear Security Administration, under award number DE-NA0002375; 3) Sandia National Laboratories; and 4) Intel Parallel Computing Centers. This work used the computational resources at: 1) SCI Institute, University of Utah; 2) Center for High Performance Computing, University of Utah; 3) Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under contract DE-AC02-06CH11357; 4) Lawrence Livermore National Laboratory; 5) Sandia National Laboratories; and 6) National Supercomputing Center in Wuxi.

CHAPTER 1

INTRODUCTION

Exascale supercomputers that can carry out at least 10^{18} double-precision floating point operations per second are expected to be operational in 2022/23. The size, complexity, and diversity of the computer architectures anticipated in the exascale and postexascale era pose new challenges for the software applications running on these machines. The software running on the exascale supercomputers needs to effectively manage/utilize increased concurrency, deep memory hierarchies and data management, heterogeneous components with varying performance; Single Instruction Multiple Data (SIMD) architectures/Graphics Processing Units (GPUs)/accelerators; diversity in architectures that are being rapidly developed; and resiliency for potentially increasing numbers of faults. These challenges are discussed in more detail in Section 1.1. Although such challenges can also be met by heroic programming efforts, the process can be made easier. One easier approach is to use **Asynchronous Many-Task (AMT) runtime systems** (abbreviated as AMT systems or just AMTs throughout this work) to help manage the increased concurrency, deep memory hierarchies, and heterogeneity. AMTs allow application developers to divide the parallel code into smaller tasks and specify the dependencies between tasks to control the data flow and execution path. These tasks are executed by AMT runtime on different architectures and multiple compute nodes. Based on the capabilities of the AMT used, AMTs can take care of data movement and exploit the concurrency up to varying capacities, which not only increases the effective utilization of nodes, but also takes the burden of application developers. However, a great deal of work needs to be done to prepare AMT infrastructure to overcome the exascale challenges.

This dissertation provides some key steps needed to solve/help solve the challenges in utilizing upcoming architectures. This work uses Uintah Computational Framework (UCF) [40], an open-source AMT, to demonstrate different techniques required to port AMTs

to next-generation architectures and specifically to exascale systems.

1.1 Exascale Challenges

The Department of Energy (DOE) report [132] identified the top 10 challenges (not rank-ordered by importance) in building and using an exascale supercomputer.

1) Energy efficiency: Limit the power consumption to 20 MW.
2) Interconnect technology: If the high performance and energy efficient network is not present, then the exascale system would be like a data center with the millions of individual computers rather than a supercomputer [132].
3) Memory technology: Minimizing data movement and making memories more energy efficient are critical factors.
4) Scalable system software: System software needs to handle the growing scale of new systems, along with overall power management and the resilience of the thousands of nodes.
5) Programming systems: More expressive programming models are needed to simplify developer efforts and benefit from the locality, multimillion or even perhaps billion-way parallelism, resilience.
6) Data management: The system needs to effectively manage the explosively increasing amount and complexity of the data generated by experiments and simulations.
7) Exascale algorithms: Refactoring of legacy codes is needed to run them efficiently on massively parallel architectures. Such refactoring may involve leveraging multi-core architectures, communication-avoiding algorithms, synchronization-minimizing algorithms, adaptive load balancing, efficient scheduling, memory management for heterogeneity and scale, energy-efficient algorithms, etc.
8) Algorithms for discovery, design, and decision: New methods are needed to efficiently carry out uncertainty quantification and optimization on complex multiphysics problems.
9) Resilience and correctness: Exascale supercomputers may have more faults than the current generation machines, and both the machines and applications need the ability to handle and recover from the increased number of faults to maintain accuracy.

10) Scientific productivity: New tools are needed to improve the productivity of the scientists using the exascale machines.

1.2 Motivation

The first two of the three planned US exascale supercomputers are expected to be operational by 2022/23: DOE Aurora [1] will have Intel GPUs and DOE Frontier [2] will use AMD GPUs. The Japanese Fugaku supercomputer based on the ARM architecture is now operational and currently the fastest machine in the world. Among the operational systems, six of the top 10 supercomputers in the current (June 2021) Top500 list [3] use Nvidia GPUs, whereas the number 10 machine, NSF Frontera [4], is built using Intel CPUs. Some open questions need to be answered to run AMTs on these current and emerging machines efficiently. This research focuses particularly on Uintah AMT [40] (described in more detail in Chapter 2). Uintah must answer exascale challenges mentioned in Section 1.1, specifically those challenges where applications/algorithms need innovative refactoring to run on the exascale machines and next-generation architectures.

Humphrey et al. developed a preliminary task execution model for Nvidia GPUs within Uintah [103, 104, 139]. John Holmen [89–91] and Bradley Peterson [150] introduced the Performance Portability Layer (PPL) along with an intermediate portability layer in Uintah. The aim of introducing the PPL is to be able to execute the same single-source C++ "portable" code on diverse architectures, including CPUs, GPUs/accelerators, and manycore systems, and achieve performance close to that of the native programming model without rewriting or fine-tuning code on every platform. Peterson and Holmen also enhanced Humphrey's task execution model and developed different task schedulers to efficiently run Uintah tasks on manycore platforms, multicore CPUs, and GPUs [88–90, 141, 153, 154, 179]. Peterson introduced the ability to asynchronously launch multiple GPU kernels in parallel [154], which improved the GPU utilization. These efforts take an additional step to solve the exascale challenges such as handling heterogeneity, exploiting ever-increasing parallelism, productivity, data management and movement, and efficient scheduling.

However, some problems remain unsolved: resiliency infrastructure for AMT, portable infrastructure for effective vectorization, more efficient algorithms, and programming models to exploit the emerging hardware, bridging some gaps in the heterogeneous infras-

tructure to fully support asynchronous execution of hundreds of portable heterogeneous tasks, integration of third-party libraries, etc. At the same time, solving these problems individually is not enough to achieve exascale performance. The individual solutions to exascale challenges spanning a breadth of the technological areas must be integrated within a single application in a smooth, conflict-free, and performant manner, just like the multiple pieces of a jigsaw puzzle fitting together to form a single image. Answering these questions becomes nontrivial because the Uintah codebase:

1) consists of 1-2 million lines of complex code;
2) has hundreds of pre-existing tasks in application code;
3) executes tasks out of order, which makes it difficult to detect and reproduce defects;
4) supports multiple levels of parallelism such as multiple nodes threads, vectorization, GPUs mixed with task parallelism and data parallelism, which causes numerous race conditions at different levels;
5) demands that enhancements maintain accuracy for all combinatorially increasing mixes of various application codes (e.g., components, tasks, models), infrastructure (e.g., task schedulers, portability support, task graph compilation, etc.), destination platforms (CPUs, GPUs, manycore architectures, SIMD execution), multiprocess/-multinode execution, multiple third-party libraries, and different runtime configurations;
6) needs to have backward compatibility for any solution to work with the legacy code;
7) must maintain a divide between application code and infrastructure code;
8) is under active development with many contributors; and
9) has a pre-existing userbase to support.

1.3 Thesis Statement

This research aims to help move Uintah closer to the ultimate goal of running on the next-generation architectures and exascale supercomputers. As a next step toward the exascale readiness, this work develops from scratch: 1) resiliency infrastructure (Chapter 3); 2) a new asynchronous task scheduler (Chapter 4); 3) a novel portable and efficient vectorization primitive (Chapter 5); and 4) a state-of-the-art threading model for multicore and manycore

architectures (Chapter 6). This research also enhances the existing heterogeneous task schedulers, related infrastructure, and integration of third-party libraries (Chapter 7). These developments provide potential solutions to address some aspects of the following exascale challenges:

1) Exascale algorithms: The new asynchronous task scheduler, the new threading model for many core architectures, improvements in heterogeneous task schedulers, etc., help minimize synchronization and efficiently use intranode parallelism.
2) Data management: The improvements in the heterogeneous task scheduler help manage data movement between CPUs and GPUs.
3) Programming systems: This work provides new easy-to-use capabilities to exploit parallelism to its full potential. Examples include the portable vectorization primitive and the new threading model.
4) Resilience and correctness: The resiliency research demonstrates a faster solution capable of gracefully handling the node level failures without traditional checkpointing.
5) Scientific productivity: Uintah itself hides details such as architectural complexities, task scheduling, load balancing, etc., from the application developers. It also provides a Performance Portability Layer (PPL) using Kokkos that allows users to write portable code. This work introduces a new portable vectorization primitive and enhances the existing portability framework within Uintah. Both of these contributions can save the architecture-specific application development and tuning efforts for the developers.

This work achieves another goal of integrating the individual solutions to ensure smooth, conflict-free, and performant interactions among them, just like the different cogs in the same machine working together. The techniques demonstrated using Uintah can be easily adopted by other AMTs. Some of the research can also be used by third-party libraries and applications to improve their performance.

This work prioritizes the topics needed in the immediate future and helps reach production readiness for them, e.g., new threading models, portability and task scheduling, etc. On the other hand, there is not enough clarity on some challenges regarding how and when these problems might become significant. Hence, current research has developed new

solutions but has not invested more efforts to integrate them into production code at full scale, e.g., resiliency research.

The success of this work advances Uintah toward the ultimate goal of achieving the exascale runs.

1.4 Unique Contributions

The unique contributions of this work in moving Uintah closer to the exascale readiness are listed below, along with the associated publications :

1) Developing a new "resiliency" component within Uintah to detect node failures and recover from them without using traditional checkpointing-and-restart [162].

2) Demonstrating a novel "asynchronous" task scheduler to help run Uintah to the Sunway Taihulight supercomputer – the first AMT to do so [192].

3) Developing a prototype of a portable SIMD primitive in Kokkos to achieve efficient vectorization within portable code [165]. This prototype was the first SIMD primitive with a CUDA back end.

4) Implementing a new threading model based on MPI endpoints for performance improvements on manycore architectures – the first full-scale implementation of a real-life application that demonstrated the usefulness of MPI endpoints by running the code on 256 nodes of Intel Knight Landing [163, 164, 195].

5) Enhancing the existing heterogeneous task scheduler and infrastructure and interoperability mechanisms to 1) avoid excessive synchronizations while gathering halo cells; 2) ensure support for all types of task dependencies (which was missing earlier); 3) port the third-party library tasks; 4) improve handoffs between Uintah and third-party libraries; and 5) fix race conditions. These modifications, coupled with the earlier implementations of PPL and heterogeneous task scheduler by other Uintah developers, allowed **Uintah to run hundreds of fully portable tasks successfully at the scale of 6,144 Nvidia V100 GPUs and 3,072 IBM Power 9 CPUs on DOE Summit, and 4096 nodes of Intel Cascade Lake on TACC Frontera without rewriting any code – *the first AMT to achieve the feat*** [1] [93]

[1] It must be noted that this work extended the earlier contributions by Alan Humphrey, Daniel Sunderland,

1.5 Asynchronous Many-Task Runtime Systems

AMT is an asynchronous, adaptive, out-of-order, data-flow-based programming model. The application developers and domain scientists can write the code divided into smaller "tasks" using AMT's application programming interfaces (APIs), where each task "requires" or "computes" or "modifies" some data. The AMT executes tasks based on the availability of data and compute resources, and tries to minimize synchronization. The data access pattern of "require/compute/modify" leads to the formation of "dependencies" - "read-only," "write-only," and "read-write," respectively, between tasks, e.g., a simple matrix multiplication task (C=AxB) needs two inputs matrices A and B and generates the output matrix C. A and B form read-only dependencies, whereas C forms a write-only dependency for the task. The multiplication task can be executed only after both inputs, A and B, are available. However, tasks to generate A and B can be executed in any order (assuming A and B do not have other dependencies). AMTs can also exploit increase node-level parallelism through overdecomposition of an application into many tasks while also managing low-level system details necessary for efficient resource utilization behind-the-scenes. The "task scheduler" in AMT takes care of such data-flow-based out-of-order task execution based on the availability of data. In a multithreaded environment, AMTs can execute multiple ready tasks at a time in parallel. AMTs also assist in interprocess / internode communication up to varying capacities, e.g., gathering of "halo cells," i.e., neighboring cells, for stencil-like computations. The advantage of AMTs is that such communication can possibly happen in the background while other "ready" tasks are getting executed, hence the name "asynchronous." The adaptive nature of task execution also takes care of load balancing. AMTs may also provide easy abstractions to hide architectural details and effectively use different levels of parallelism such as multithreading, GPUs, manycore machines, vectorization, etc. Because of such abstractions and the dynamic, adaptive, and asynchronous nature of AMTs, they are becoming a favorite among the exascale research community.

Examples of AMTs include Charm++ [114], HPX [113], Legion [33], PaRSEC [44], StarPU [30], and Uintah [40]. More details of Uintah are presented in Chapter 2. The

---

Bradley Peterson, John Holmen, Derek Harris, Jeremy Thornock, and Jebin Elias with the same goal, and everyone deserves the credit for these successful runs.

comparison between different AMTs can be found in [105, 150]

1.6 Target Problem

The University of Utah's Carbon Capture Multidisciplinary Simulation Center (CCMSC) was a member of the Department of Energy (DOE)/National Nuclear Security Administration's (NNSA's) Predictive Science Academic Alliance Program (PSAAP) II initiative, supporting the large-scale simulations used for the design and evaluation of an existing 1000 MWe ultra-supercritical clean coal boiler developed by Alstom (GE) Power. At the 1 mm grid resolution, the boiler simulation runs across approximately 9 x 10^{12} cells. The simulation has three main components: 1) The "Arches" component implements Large Eddy Simulation (LES), which solves the mass, momentum, and energy conservation equations for the gas and solid phases of combustion on a three-dimensional finite volume mesh [173, 174]; 2) The Hypre Solve component solves the "pressure equation" with billions of variables (generated by Arches) using Hypre linear solver [71, 72]; and 3) The Reverse Monte Carlo Ray Tracing (RMCRT)-based radiation model simulates the heat transfer through radiation inside the boiler [105, 106, 178]. More details of these components are presented in Chapter 2. The shear scale and complexity of the problem make it worthy of running the simulation at exascale.

1.7 Target Architectures and Programming Models

The primary target machines for these simulations are DOE Aurora and DOE Frontier, which are built using Intel and AMD GPUs, respectively. It is also desirable to run the simulations on DOE Summit, which is built using IBM Power 9 processors and Nvidia V100 GPUs, and NSF Frontera based on Intel Cascadelake CPUs. Given the increasing success of Fugaku, it will be worth targeting ARM architectures in the future.

Multiple programming models are available to execute on different architectures. As a strategic decision, Kokkos [68, 182] was chosen as the preferred programming model for the "X" component of the MPI+X model and to develop the performance portable application code in Uintah. Using Kokkos is expected to help in writing "performance portable" code. Here, "portable" in "performance portable code" means single-source C++ code written using Performance Portability Layers (PPLs) such as Kokkos and Kokkos can map user code to the appropriate back end depending on the platform. "Performance"

indicates achieving performance close to that achieved by the native programming model of the target architecture, *without architecture-specific development tuning in the application code*. Thus, Kokkos provided an easier way to tackle diverse target architecture and improve productivity.

1.8 Dissertation Organization

This dissertation is organized as: Chapter 2 overviews Uintah. The newly introduced resiliency infrastructure [162] is presented in Chapter 3. Chapter 4 talks about the new asynchronous task scheduler in Uintah and how Uintah was ported to the Sunway Taihulight supercomputer [192]. Chapter 5 shows the novel portable SIMD primitive and improved vectorization [165]. Research about the new threading model based on MPI endpoints and resulting performance improvements [163, 164, 195] are covered in Chapter 6. Chapter 7 presents enhancements in the heterogeneous task schedulers, related infrastructure, and interoperability of the third-party libraries [93]. Conclusions, future work, and lessons learned are captured in Chapter 9.

CHAPTER 2

UINTAH COMPUTATIONAL FRAMEWORK

The Uintah Computational Framework [40] is a massively parallel, distributed, Asynchronous Many-Task (AMT) runtime system with Adaptive Mesh Refinement (AMR) support. Uintah is designed to solve complex partial differential equations (PDEs) involved in multiphysics problems such as combustion simulations and fluid interactions. Uintah's philosophy makes a clear distinction between application development and the underlying AMT infrastructure and allows independent development of both. Uintah has different infrastructure components/modules such as dependency analysis, task graph compilation, several types of task schedulers, MPI communication, multithreading, Performance Portability, GPU execution support, load balancing, data archiving, and datawarehouse. The components/modules used for multiphysics simulations include Material Point Method (MPM), Arches, Implicit Continuous-fluid Eulerian (ICE), MPM-ICE, Wasatch, Reverse Monte Carlo Ray Tracing (RMCRT), Hypre pressure solve, etc. The clear distinction between the AMT infrastructure and the application development allows the development of both by the respective experts. It frees the application developers from manual domain decomposition, gathering of halo regions, multithreading, handling host-device transfers, and writing GPU specific code.

Originally developed at University of Utah's Center for the Simulation of Accidental Fires and Explosions (C-SAFE), Uintah-based simulations of next generation coal boilers have been successfully ported to different architectures, including heterogeneous architectures and have scaled up to 96K, 262K, and 512K cores on the NSF Stampede, DOE Titan, and DOE Mira, respectively [40, 140]. For portable standalone use of Kokkos::OpenMP, good strong-scaling has been shown to 1,728 Intel Knights Landing processors on the NSF Stampede 2 system [90]. For portable standalone use of Kokkos::CUDA, good strong-scaling has been shown to 64 NVIDIA K20X GPUs on the DOE Titan system [154].

2.1 Structured Grid and Variables

Uintah support for Adaptive Mesh Refinement (AMR) as described in detail in [133]. Uintah employs a structured grid of cubic mesh cells grouped into patches. The grid can be divided into "levels" of patches. With each level, patches can be refined to a higher resolution or can be coarsened back to a lower resolution. Uintah provides five different types of "simulation variables," which are basically C++ classes that provide simple interfaces to create, access/iterate, help MPI communication, and update these variables. This research used only one type of variable, called "the grid variable." A grid variable is always associated with a patch and holds values corresponding to every cell within a patch. More information about the different variable types and their usage can be found in [38, 80].

2.2 Uintah AMT Design

Fig. 2.1 shows an abstract view of the Uintah AMT runtime system and has five logical layers.

1) Application components layer: The topmost layer contains the application compo-

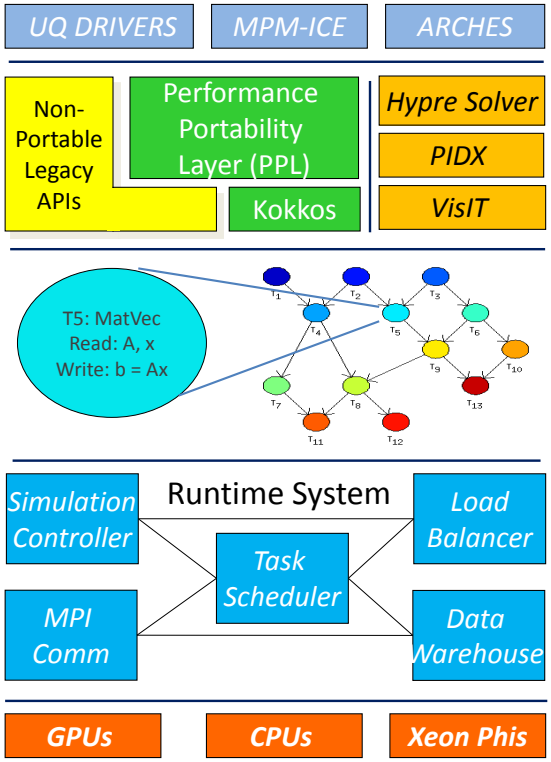


Fig. 2.1: Uintah software architecture.

nents, which may interact with each other.

2) API layer: The application components are developed using any of the three APIs: Nonportable legacy Uintah APIs, Performance Portability Layer (PPL) [92,93,150,154] built on top of Kokkos [68,182], and third-party libraries such as Hypre linear solver [71,72]. The Performance Portability Layer also supports legacy Uintah APIs so that users get an option to write serial tasks and incrementally promote those to OpenMP or GPUs.

3) Task graph layer: Each application component is written as a group of tasks, and each task is a small unit of work that reads/computes/modifies a number of patch-based simulation variables (known as datawarehouse variables) [80]. Tasks can access datawarehouse variables in three access patterns: read-only (i.e., the task reads a variable created earlier), write-only (i.e., the task creates a new variable), and read-write (i.e., the task updates the created earlier). Uintah provides APIs for the application developers to create tasks and specify these patterns along with the number of halo cells (or ghost cells) required by the task. Uintah then compiles the task and variable information into a task graph, a directed acyclic graph (DAG), where a task forms a node, and dependencies form the edges. Based on the variable access pattern, three types of dependencies are created: "requires" dependency for read-only access, "computes" dependency for write-only access, and "modifies" dependency for read-write access. The task graph compilation phase ensures that the task that "computes" a variable is always placed *before* the task that "requires" it. Based on the halo cells requirements given by users, the compilation phase also generates metadata and additional dependencies needed for automated MPI communication. More information about the creation of tasks, variable dependencies, and task graph compilation can be found in [80,105].

4) Runtime system layer: The runtime system is the heart of Uintah. It contains different components such as 1) simulation controller, which generates the task graph and controls the main simulation; 2) load balancer, which provides different algorithms to do load balancing between different MPI ranks; 3) datawarehouse, which maintains all the simulation variables known as datawarehouse variables; 4) MPI communication

component, which manages the automated exchange of MPI messages; and 5) task scheduler, which consumes the task graph and controls the task execution and data movement. All these components of the runtime system interact with each other to run the simulation.

5) Hardware layer: Finally, tasks executed by the scheduler are run on the underlying hardware based on the availability of hardware and the interfaces used by the application developer (e.g., Kokkos or a third-party library). The target platforms could be CPUs, GPUs, and many core architectures such as Intel Xeon Phis.

The rest of the chapter gives a brief introduction to the key components used/modified during this work.

2.3 Datawarehouse

The "datawarehouse" object manages all the simulation variables across tasks. It stores metadata for every simulation variable as a key-value pair and provides simple interfaces to allocate, update, and destroy the variables. The variables can be uniquely identified based on the variable name, patch id, and material id. Other useful metadata include a pointer to actual data, the type of the simulation variable, size in bytes, low and high indexes of the variable in three-dimensional Cartesian coordinates, padding, and access stride [150]. The datawarehouse can distinguish data in different timesteps. The *Old* datawarehouse holds the data calculated in the previous timestep, and the *new* datawarehouse stores the data of the current timestep.

2.4 Task Scheduler

Uintah's task scheduler reads the DAG, interacts with the datawarehouse, uses MPI to exchange the halo region, tracks various tasks and dependencies, transfers host-device data, and executes tasks as and when dependencies are satisfied. As dependencies are satisfied, many tasks can become ready for execution and are run by multiple threads in parallel on CPU or GPU, while communication for other tasks progresses in the background overlapping computation with communication. Such an execution model gives Uintah its "Asynchronous Many Task" nature. Uintah adopted an MPI+X hybrid parallelism approach using the foundational MPI+PThreads task scheduler [138] to overcome memory

footprint limitations on the NSF Kraken and DOE Jaguar systems. Iterative efforts since have targeted extensions in three key areas. 1) support for heterogeneous systems that resulted in an MPI + PThreads + CUDA task scheduler [103, 104, 139, 151, 152, 155]; 2) support for manycore systems was provided using an MPI + Kokkos::OpenMP task scheduler [88, 141]; and 3) portability support for CPU and GPU tasks was added using by a heterogeneous MPI+PThreads+Kokkos::CUDA task scheduler [89, 90, 153, 154, 179]. Fig. 2.2 shows the basic per-MPI process infrastructure forming Uintah's heterogeneous task schedulers.

The work presented here most closely relates to the centrally located "CPU Core" and "GPU" ovals, which correspond conceptually to individual task executors. A task executor corresponds to the specific compute resources (e.g., cores) used to execute task executor logic discussed in a subsequent section. A task executor can be a single thread or can be a

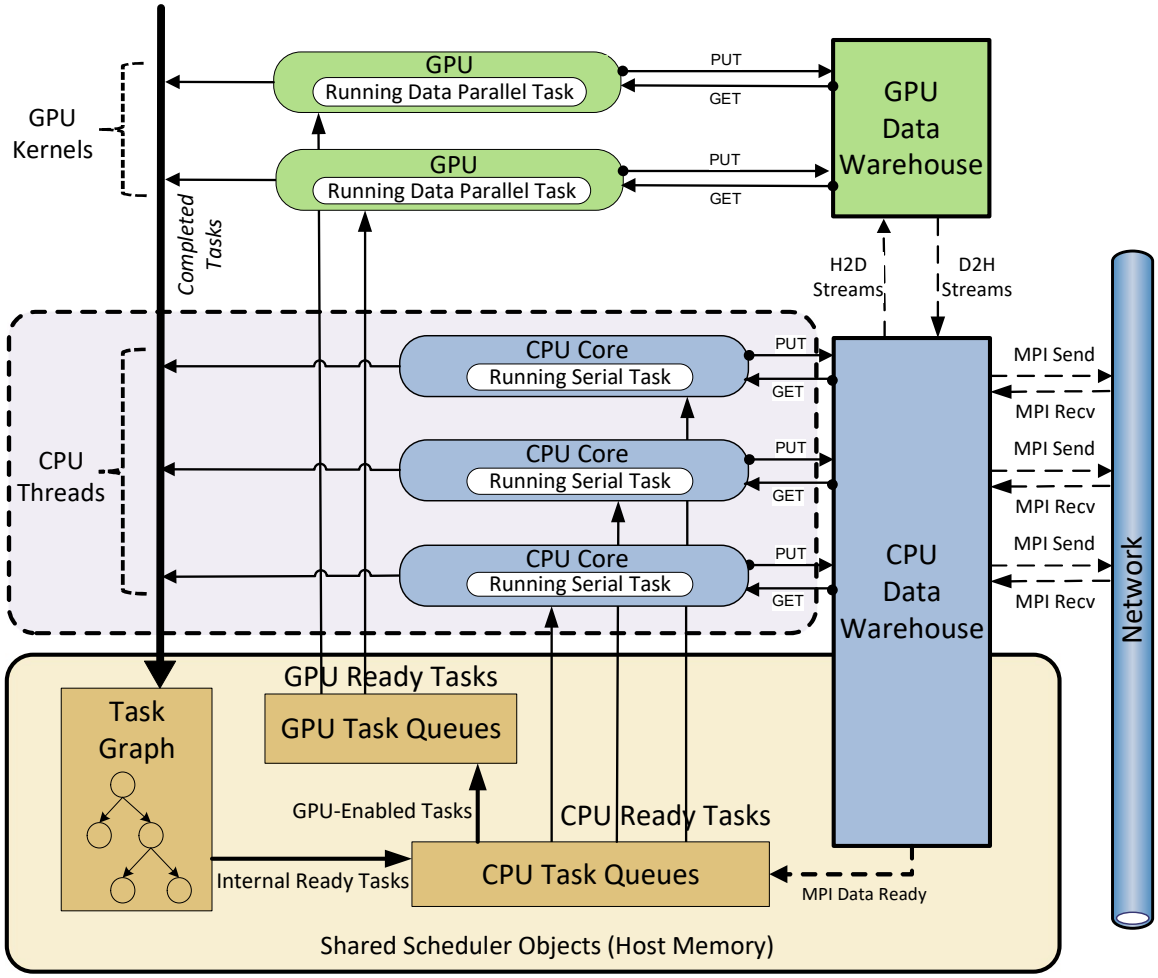


Fig. 2.2: Uintah's multithreaded MPI task scheduler [93] (Reprinted with permission from ACM).

team of threads that may or may not use GPUs. More details on Uintah's task schedulers can be found in a scheduler survey [139]. Uintah targets the use of one MPI process per GPU for heterogeneous systems.

2.5 Performance Portability and Kokkos

Many new computer architectures are being developed to potentially improve floating point performance, such as those being developed for exascale computing. Intel Xe GPUs, ARM v8.2-A processors, Nvidia, and AMD GPUs are just a few examples of diverse hardware architectures being developed. [2, 5, 177]. Multiple performance portability frameworks are being developed to avoid architecture-specific tuning of programs for every new architecture. Some of such standards/libraries include Kokkos [68, 182], OpenCL [145], Intel OneAPI [6], SYCL [7], RAJA [96], and OCCA [136]. Such portability frameworks provide uniform APIs to shield a programmer from architectural details and provide a new performant back end for every new architecture to achieve performance portability.

The Kokkos C++ library [68, 182] is an open-source C++ programming model for writing single-source performance portable code optimized for a diverse set of major HPC systems. The Kokkos Performance Portability Layer (PPL) provides core abstractions for portable parallel execution patterns (e.g., *parallel_for*, *parallel_reduce*, *parallel_scan*) and portable data structures (e.g., Kokkos Views). This effort has since expanded into the Kokkos C++ Performance Portability Programming EcoSystem, which additionally offers a broad set of solutions for science and engineering applications (e.g., Kokkos-aware math kernels). Originally developed at Sandia National Laboratories, the team has also expanded with dedicated developers at Argonne National Laboratory, Lawrence Berkeley National Laboratory, Los Alamos National Laboratory, Oak Ridge National Laboratory, and the Swiss National Supercomputing Centre. More details on Kokkos can be found on the Kokkos GitHub [8, 9].

Uintah is an early adopter of Kokkos, with developers from both teams collaborating directly as a part of the University of Utah Carbon Capture Multidisciplinary Simulation Center's (CCMSC) participation in the DOE/NNSA's Predictive Science Academic Alliance Program (PSAAP) II initiative. In recent years, this adoption has proven worthwhile in easing Uintah's transition from manycore systems (e.g., the NSF Stampede 2 system) to

multisocket, multidevice heterogeneous systems (e.g., the DOE Lassen system) given the already familiar programming model. Uintah adopted Kokkos through an intermediate layer [89,91]. The intermediate layer will make it easier to support the current and future interfaces to underlying programming models, hide low-level details from application developers, and allow infrastructure developers to tune the interfaces behind-the-scenes in a single location instead of making far-reaching changes across application code [90]. Uintah's Kokkos-related activities prior to this work or carried out in parallel by the other researchers are itemized in [92,150]. Kokkos back ends to OpenMP and CUDA are referred to throughout this work as Kokkos::OpenMP and Kokkos::CUDA, respectively.

2.6 Arches

The Arches component of Uintah simulates turbulent reacting flows with participating media radiation and is used in the predictive boiler simulations at CCMSC. Arches implements Large Eddy Simulation (LES), which solves the mass, momentum, and energy conservation equations for the gas and solid phases of combustion on a three-dimensional finite volume mesh [173,174]. Heat, mass, and momentum transport in the reacting flows are modeled using a low mach number (M <0.3) variable density formulation [173]. Arches time integrates these discretized equations using a strong stability-preserving second- or third-order Runge-Kutta method [78]. The low mach pressure projection requires a solution of a sparse linear system at each timestep using the Hypre linear solver package [71,72].

2.7 Hypre Pressure Solve

Developed at the Lawrence Livermore National Laboratory, the Hypre library provides high-performance preconditioners and solvers for the solution of large sparse linear systems on massively parallel computers. [71,72].

Arches initially leveraged the Discrete Ordinates [159] and P1 approximation [123] methods to solve the radiation heat transfer equation. However, the pressure equation formulated as a linear system is solved using Hypre and takes a significant portion of the compute time. Schmidt [168] attempted to improve the weak scaling of Hypre for the Arches/Uintah specific equations. He used a nonsymmetric multigrid red black Gauss Seidel preconditioner to the conjugate gradient (i.e., Preconditioned Conjugate Gradient PCG) method. Instead of solving the equations at every level of the multigrid, results were

interpolated to the "skipped" levels. As a result, the amount of work is reduced and the solve time is improved.

This work attempted to improve Hypre performance for the latest manycore architectures.

2.8 RMCRT

Work done by Sun [178] and Hunsaker [106] has shown that Monte Carlo ray tracing methods are potentially more efficient than the Discrete Ordinates Method used earlier. The Reverse Monte Carlo Ray Tracing (RMCRT)-based radiation model was developed as a standalone module and can be used within Uintah's simulation components [103]. The RMCRT module simulates the radiation occurring inside the boiler. The radiation is the primary mode of heat transfer. The RMCRT is parallelized by patch-based domain decomposition and by replicating the global information on each node using MPI [102]. Such a model needs global all-to-all MPI communication and is not scalable. The scalability and performance of the model were improved by the AMR technique. AMR reduces the communication volume and computational complexity [102]. The RMCRT kernel was ported to the manycore architectures such as GPUs and Intel's Knights Landing to improve performance further [102, 104].

The CCMSC's predictive boiler simulation employees 209.5 million to 13.5 billion rays and is an opportunity to exploit massive parallelism. More information about Uintah's RMCRT models can be found at [102–104].

CHAPTER 3

EXPLORING NODE FAILURE RESILIENCY FOR AMTS WITHOUT CHECKPOINTING

The move to a new generation of supercomputers with peak performance at exascale and beyond over the next few years presents significant challenges. Chips are expected to contain 100x more processing elements than those currently used, which may increase the processing components from approximately 500,000 in today's systems to a substantial fraction of one billion by 202X [10]. The exascale committee report predicted that the time required for an automatic or the application-level checkpoint/restart would exceed the mean time to failure (MTTF) of a full system [10]. As a result, the traditional recovery technique of checkpointing and recomputing from the last checkpoint may prove problematic in the development of exascale computing in the near future. Although the problem is mitigated for the exascale machines, it may arise again for the post-exascale machines [36]. In any case, with the potential for increased failures due to faults, a development path is needed to handle any potential resilience issues. At present, exascale systems running millions of cores are expected to experience numerous faults every day. This potential challenge has spurred research in the area of resiliency. For example, [55, 56] describe some resiliency approaches based on the types of problems that occur. These problems may be categorized as soft failures due to bit flips or hard failures due to the core, node, or communications failures.

This challenge makes Algorithm-Based Fault Tolerance (ABFT) a more attractive option. With ABFT, data lost during failure can be reconstructed at runtime because programmers are aware of the algorithms and functionality of the particular problem being solved. Since recovery occurs at runtime without using checkpointing (and hence without disk access), ABFT has the potential to be faster than traditional checkpoint/recovery.

Given the asynchronous task-based operations of AMT runtime systems, a fault

tolerance algorithm can also potentially be written and executed as a set of tasks. The approach can minimize the wait time of the fault detection and recovery tasks by running them with other ready tasks.

This chapter intends to explore an ABFT resiliency approach for Uintah AMT by addressing the future work challenge posed by Dubey [67]: using lower fidelity solutions on surviving compute nodes to rebuild a solution for failed nodes. Uintah supports Adaptive Mesh Refinement (AMR) and provides built-in tasks for coarsening/refining patches as required by ABFT. This chapter presents the following contributions toward making Uintah AMT resilient:

- An ABFT solution is constructed, that performs faster than the traditional checkpointing method. The new solution works as follows:
  - MPI Ranks take advantage of AMR and exchange coarse patches.
  - Any node failures are detected using User-Level Failure Mitigation (ULFM) [41], a modified version of OpenMPI that supports failure detection.
  - Surviving ranks are brought back to a stable state, and the orphan tasks are redistributed to surviving ranks.
  - Lost patches are recovered using interpolation, and the normal execution of tasks is continued.
- An accurate physics-constrained interpolation for recovery is used to reconstruct the solution. Evidence going back to [39] suggests that for problems involving integration forward in time, new values should be calculated with sufficient accuracy such that interpolation errors do not pollute the remainder of the time integration. The approach taken here is to extend Dubey's work on recovering from node failure by using a simple form of the limited ENO interpolation (LENO) scheme suggested by Berzins [37] that preserves the positivity and boundedness of the solution. This preservation of the physical bounds on the solution values is important in many real-life engineering applications, such as combustion [130] and weather forecasting [131].
- A novel advection-reaction type problem is developed and used in one- and three-dimensional cases to show the importance of using interpolation that respects physical bounds on the solution. A three-dimensional version of Burgers' equation is used to

show the need for high-accuracy interpolation.

All these contributions were earlier published in [162].

3.1 Implementation of Resilience in Uintah

This implementation of resilience in Uintah introduces a systematic way to detect and handle failures, as shown in Fig. 3.1 and Algorithm 1. Statements highlighted in Algorithm 1 indicate the changes made to achieve resiliency. The algorithm also provides a short description of existing steps for task graph compilation, dependency creation, automated MPI message generation, load balancing, task scheduling, execution, etc. More details on the existing task execution model of Uintah can be found in Chapter 2 and references [104, 153, 155].

ULFM was chosen as the resilient MPI implementation because of two important features: ULFM provides an abstraction to "revoke" MPI calls, and it can be used by surviving processes to revoke any pending calls made to the failed rank. Revoking pending MPI calls avoids the deadlock of processes that are waiting on the failed rank. ULFM also provides another abstraction to "shrink" the global communication handle. This abstraction

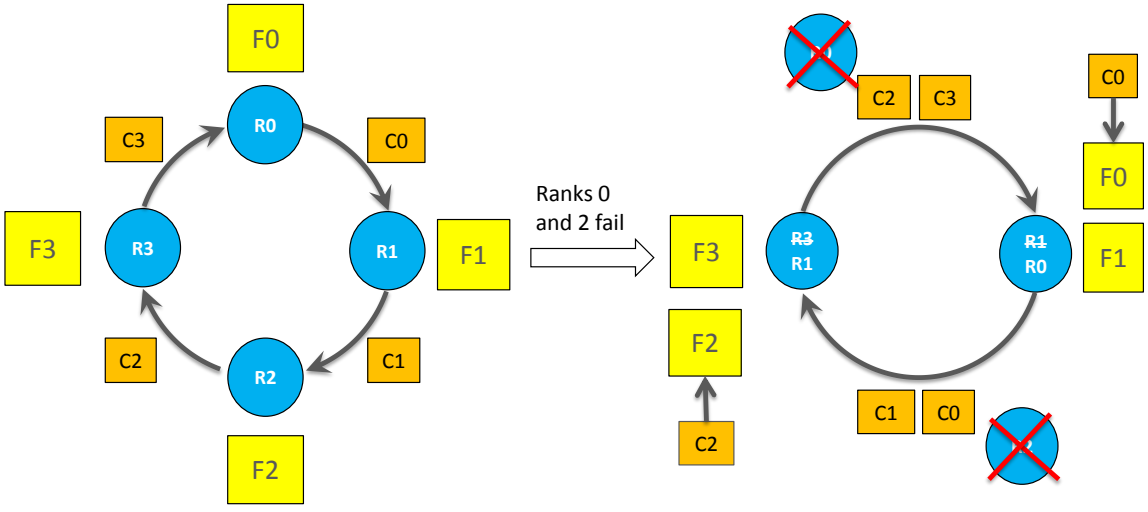


Fig. 3.1: Node failures and patch recovery for MPI ranks R0, R1, R2, and R3, with fine patches F0, F1, F2, and F3 and coarse patches C0, C1, C2, and C3.

1: Initialize MPI world. **Set error handler to static ErrorHandler method in new class Resiliency.**
2: Read input file (.ups file) to get the problem specification.
3: Create instances of classes Scheduler, Load Balancer, Data Archiver, execution component depending on input file, and call the Simulation Controller.
4: **Create an instance of Resiliency class, and save pointers to Processor Group (which holds information about MPI world), Simulation Controller, Scheduler, Load Balancer, Data Archiver and execution component to the Resiliency instance.**
5: **Create new tasks: 1. to coarsen patches with a dependency on timeAdvance task. 2. A dummy task with a dependency on coarsen task.**
6: Call "schedule initialize" and "schedule timestep" methods of the components provided by users of Uintah. These methods create and add tasks to the Scheduler instance.
7: Create Task Graphs (DAG) for all the tasks and their dependencies.
8: Determine neighborhood processors.
9: Add extra tasks to send data to dependent processes. Create extra tasks for neighboring processors. These tasks are not executed on the current processor but are used for creating dependencies.
10: Assign processing resources for each task.
11: Call EXECUTE()
12: Finalize MPI.
13:
14: **procedure** EXECUTE
15: **for** i = 1 to number of timesteps **do**
16: Advance Data Warehouse from previous timestep to current timestep.
17: **for** every task in task graph **do**
18: Post MPI Receive messages for dependencies.
19: Execute the task (i.e., the function pointer provided by the users in Scheduler timestep function)
20: Post MPI Send messages for dependencies.
21:
22: **procedure ERRORHANDLER**
23: **Use ULFM APIs MPIX_Comm_failure_ack, MPIX_Comm_agree, and MPIX_Comm_failure_get_acked to find out failed ranks, and create a consistent picture of failed ranks globally.**
24: **Call ULFM API MPIX_Comm_shrink to get new global MPI communicator handle, which excludes failed ranks.**
25: **Set newRank = oldRank**
26: **for every failed rank do**
27: **newRank = newRank $>$ failedRank ? newRank-1 : newRank**
28: **Call MPI_Comm_split to assign ranks as per newRank.**
29: **Update comm instances stored in class ProcessorGroup and add error handler for new communicator.**
30: **Clear existing communication queues.**
31: **Reassign 'execution rank' to patches using same logic of newRank.**

Algorithm 1: Resilient Uintah algorithm.

32: **for every task graph do**
33: **Call load balancer method to recreate neighborhood, because updating ranks and 'patch execution ranks' changes neighbors.**
34: **Reassign task resources as per patches.**
35: **Clear dependencies of every task within a task graph.**
36: **Merge any reductions/output tasks.**
37: **Create/update Send Old Data tasks to accommodate new neighbors.**
38: **Create new tasks corresponding to neighbors on every rank, which is necessary to create dependencies used in MPI communication.**
39: **Allocate memory for failed patches.**
40: **for every task graph do**
41: **Create dependencies.**
42: **Assign MPI message tags.**
43: **Recompute local tasks.**
44: **Schedule and execute an interpolation task for failed patches using subscheduler.**
45: **Call EXECUTE(): This continues execution of timesteps.**

Algorithm 1: Continued.

removes failed ranks from the set of active ranks and returns a new global communication handle containing surviving processes that can be used for recovery and postcrash execution. The following steps illustrate the resiliency process:

- **Patch Coarsening**: A new task to generate coarse patches is created and scheduled. This task utilizes the existing AMR capabilities of Uintah with a refinement ratio of 1:2. Thus, for every eight cells of a fine patch in three dimensions, there is one cell on a coarse patch. The refinement ratio can be changed depending on accuracy and performance requirements; however, both coarsening and interpolation tasks should use the same ratio. At the moment, coarsening is done by simply discarding alternate values in all three dimensions. For the coarsening task, a "required" dependency is added from the "timeAdvance" task, which is the main task used to implement the timestepping algorithm within Uintah. Because of task dependencies, Uintah schedules the coarsening task after timeAdvance and feeds the fine patches computed by timeAdvance as input to the coarsening task. Fig. 3.1 shows an MPI world with four ranks. Ranks R0 through R3 compute fine patches F0 through F3 and then compute coarse patches C0 through C3, respectively.
- **Coarse Patch Exchange**: To exchange coarse patches, each rank creates an empty dummy task and adds a dependency on a coarse patch computed by the $(n-1)^{th}$

rank. The dummy task does not compute anything, but because of the dependency on the $(n-1)^{th}$ rank, Uintah generates MPI messages to exchange patches. Thus, each process with rank n sends its coarse patches to the $(n+1)^{th}$ rank and receives coarse patches from the $(n-1)^{th}$ rank in a circular fashion, as shown in Fig. 3.1. This logic has two aspects: if two consecutive ranks fail, then both the fine and the coarse mesh patches are lost, causing an irreversible loss; or all tasks of failed rank get assigned to one neighboring rank, creating a severe load imbalance. However, the patch exchange logic is flexible and can be easily updated to send coarse patches to more than one rank. In the future, better load balancing can be achieved by using Uintah's dynamic load balancer to distribute orphan tasks evenly across survivor ranks. Coarsening and exchange tasks can also be scheduled at a fixed interval of timesteps rather than scheduling them for every timestep.

- **Determine the Faulty Nodes / Ranks**: MPI rank failure can be detected by changing the default MPI error handler from MPI_ERRORS_ARE_FATAL to a custom error handler in the new "Resiliency" class. This "Resiliency" class is instantiated after instantiating all other infrastructure classes and saves pointers to instances of Scheduler, Load Balancer, Simulation Controller, Application State, and MPI World. The error handler can then access these instance pointers to get the latest "state" of the simulation, and the framework can restore the context after an exception is caught.

  ULFM APIs (MPIX_Comm_failure_ack, MPIX_Comm_agree and MPIX_Comm_failure_get_acked) are used to determine failed ranks, create a uniform picture of failed ranks across all surviving ranks, create the new global communicator excluding failed ranks and invoke recovery routines.

- **Reassignment of Ranks and Patches**: When the n^{th} rank fails, the $(n+1)^{th}$ rank takes over patches of the n^{th} rank, and the rank of the process is also updated from $(n+1)$ to n. Subsequently, each rank greater than the failed rank is decremented by one. When multiple ranks fail, the same logic is iterated for every failed rank. The patch to rank assignment also uses the same logic. This process is shown in Fig. 3.1 where ranks R0 and R2 fail. Rank R1 then becomes the new rank R0 and rank R3 becomes the new rank R1. The orphan fine patches F0 and F2 are now allocated to ranks R0 and R1 (old ranks R1 and R3). When the ranks are updated, the patch to rank assignment

for patches F1 and F3 is also updated to R0 and R1, respectively. Algorithm 1 calls MPIX_Comm_split to ensure ranks are reassigned following the same reassignment strategy.

- **Updating the Task Graph**: The recovery process creates new tasks and/or changes the ownership of existing tasks, except for the reduction tasks that belonged to failed ranks. The reduction tasks are skipped to avoid additional calls to MPI_Reduce.

  New send data tasks are created on behalf of new neighbors to ensure correct dependency creation. All existing dependencies are then deleted and recreated to reflect new tasks and ranks for the automated MPI message generation.

- **Data Recovery**: Finally, the error handler reallocates memory for the adopted fine patches and schedules an interpolation task to generate fine patches from coarse patches received earlier. A subscheduler is used to schedule the interpolation task, because it is a one-time task and is not repeated for subsequent timesteps. The subscheduler creates a new instance of the task graph containing only interpolation tasks and avoids modifying the main task graph, which gets executed every timestep. Any MPI communication needed for halo exchange is automatically handled by the subscheduler utilizing the Uintah infrastructure. As shown in Fig. 3.1, new ranks R0 and R1 rebuild fine patches F0 and F2 using coarse patches C0 and C2. Uintah provides three types of variables - Node Centered (with data points at eight vertices of a cubic cell), Cell Centered (with data points at the center of a cubic cell), and Face Centered (with data points at centers of six faces of a cubic cell). In three space dimensions, the algorithm uses Tensor Product interpolation, which is a standard approach described in [76]. The application of this idea to the Newton polynomial approach described in the next section is an adaptive form of the algorithm that dates back to Narumi [146]. The challenge of addressing cell-vertex and cell-centered meshes requires a complex but straightforward application of the underlying Tensor Product approach. At each stage of this Tensor Product interpolation, boundedness in the solution is enforced for all intermediate values. The general interpolation approach is similar to the way that coarse-fine interfaces are treated in many adaptive mesh codes [134, 135].

- **Continued Execution**: Once the failed patches are recovered, execution continues in the usual fashion with the revised (reduced) MPI world. Fig. 3.1 shows that ranks R0 and R1 will continue executing tasks on patches F0, F1 and F2, F3, respectively. Rank R0 will also have coarse patches C2 and C3, and rank R1 will have coarse patches C0 and C1.

  The disadvantage of using the reduced communicator instead of spawning new ranks is the increase in the execution time for subsequent timesteps due to fewer MPI processes and nodes.

  Recreating instances of all components in newly spawned ranks and bringing those instances to the state of the current timestep is more involved than using the surviving ranks to take over patches with the existing state. This approach can be considered in the future.

The existing design keeps recovering from failures until only one rank remains at the end, provided that certain conditions hold, which requires that the maximum number of *simultaneous* failures is limited to half the ranks. In addition, the pattern of patch exchange dictates that two neighboring ranks should not fail at the same time; otherwise, the system cannot recover from a failure. Failures (which may or may not be concurrent) can occur one after another until only one rank remains, e.g., in Fig. 3.1, the failure of ranks R0 and R2 leads to renaming of the surviving ranks (R1 and R3) to R0 and R1, and they end up owning patches F0, F1, and F2, F3, respectively. If rank R0 fails now, R1 will be renamed as R0, and R0 will take over all the patches F0 through F3.

3.2 Interpolation Methods

One of the important factors in choosing an interpolation method is its accuracy. In real-life numerical simulations, it is essential to maintain physical properties such as positivity of the solution and to avoid the introduction of any spurious overshoots or undershoots. Weather forecasting [131] and combustion problems [130] are two such problems in which nonphysical solution values may not match with the underlying physics. Previous work [39] has demonstrated a need for more accurate methods than linear interpolation and suggested that cubic interpolation is important if the underlying discretization method is first or second order.

The difficulty with using cubic interpolation is possible overshoots and undershoots, i.e., interpolated values going above or below the limit of physically possible values. These overshoots and undershoots can cause errors by violating the given set of laws governing the system. For example, chemical concentrations cannot be negative. ENO methods and their extensions such as WENO [84, 171] provide a polynomial interpolant that preserves positivity in many but not all cases.

A simple approach inspired by Limited ENO (LENO) [37] is used to curtail overshoots and undershoots: allow only those divided differences that are sufficiently small. Alternatively, some of the other approaches considered in [37] might be adopted, such as the use of cubic splines.

The starting point is to use ENO interpolation schemes [37, 84] that employ the Newton divided difference form of the interpolating polynomial. For example, the cubic interpolation between known data points $U[x_i]$ and $U[x_{i+1}]$ to calculate $U[x]$ is given by

$$U[x] = U[x_i] + (x - x_i)U[x_i, x_{i+1}] + T3 + T4$$

where $U[x_i, x_{i+1}] = \frac{U[x_{i+1}]-U[x_i]}{x_{i+1}-x_i}$ and the terms $T3$ and $T4$ are products of a "multiplier" π and a "divided difference" δ: $T = \pi.\delta$. More terms may be added to obtain still higher order interpolation. The recursive formula for calculating the multiplier and divided difference for subsequent terms is $\pi_k = (x - x_i)(x - x_{i+1})...(x - x_{i+k-1})$ and $\delta_k = U[x_i, x_{i+1}, ..., x_{i+k}]$ where

$$\delta_k = \frac{U[x_{i+1}, x_{i+2}, ..., x_{i+k}] - U[x_i, x_{i+1}, ..., x_{i+k-1}]}{x_{i+1} - x_i}.$$

The third term ($T3$) can be calculated using either $U[x_{i-1}]$ or $U[x_{i+2}]$, and δ will depend on what the next interpolation points are chosen to be. Using x_{i-1} yields δ as

$$\delta = U[x_{i-1}, x_i, x_{i+1}] = \frac{U[x_i, x_{i+1}] - U[x_{i-1}, x_i]}{x_{i+1} - x_{i-1}}$$

and using x_{i+2} for δ yields

$$\delta = U[x_i, x_{i+1}, x_{i+2}] = \frac{U[x_{i+1}, x_{i+2}] - U[x_i, x_{i+1}]}{x_{i+2} - x_i}.$$

The ENO method [171] chooses the value of δ with the smallest absolute value, i.e., $min(|U[x_{i-1}, x_i, x_{i+1}]|, |U[x_i, x_{i+1}, x_{i+2}]|)$ to calculate T3. In the same way, subsequent terms can be computed by first calculating δ for points on either side, and picking the δ with the

smallest absolute value. π is calculated using terms picked up in the previous δ. ENO methods work well for many problems, but they may prove troublesome for problems bounded by physics (or any domain-specific) constraints. Berzins [37] used an adaptive ENO method that imposes conditions on the ratios of divided differences and limited the polynomial order if those conditions are violated. A simpler approach used here is to calculate the linear, quadratic, and cubic interpolation terms for the desired point, and then to pick the highest order value that satisfies the physical requirements of the solution. These requirements may include that the interpolated values are positive and bounded above, or that the interpolated values lie between the two nearest coarse mesh values [37]. In the worst case, linear interpolation is used. For example, when using an even mesh in one space dimension, the linear, left quadratic and cubic interpolants that approximate the solution values at $x_{i+1/2}$, as denoted by $u^{linear}_{i+1/2}, u^{quadratic}_{i+1/2}, u^{cubic}_{i+1/2}$, are then given by

$$u^{linear}_{i+1/2} = \frac{1}{2}\left(u_i + u_{i+1}\right), \tag{3.1}$$

$$u^{quadratic}_{i+1/2} = \frac{1}{8}\left(-u_{i-1} + 6u_i + 3u_{i+1}\right), and \tag{3.2}$$

$$u^{cubic}_{i+1/2} = \frac{1}{16}\left(-u_{i-1} + 9u_i + 9u_{i+1} - u_{i+2}\right). \tag{3.3}$$

It is straightforward to see that if the condition $u_{i-1} + u_{i+2} > 9(u_i + u_{i+1})$ holds, then $u^{cubic}_{i+1/2}$ will be negative, and either the quadratic or linear values should be used. For the quadratic case, if $u_{i-1} > 6u_i + 3u_{i+1}$, then the linear polynomial must be used. In effect, this approach employs a subset of the possible ENO interpolants while still ensuring that the interpolant is bounded. In the experiments that follow, the accuracy of linear, cubic ENO and Limited ENO interpolation methods will be shown. The extension to three space dimensions is achieved using a standard tensor product approach described in the previous section.

3.3 Experiments and Results

The computational experiments described below in Sections 6.1 and 6.2 were designed to compare the accuracies of three interpolation methods, linear, cubic ENO, and cubic LENO, for reconstructing fine patches from coarse patches. Section 6.3 describes further experiments that compare the performance of the ABFT approach (Algorithm 1) against the performance of checkpointing/restarts.

3.3.1 A One-Dimensional Advection-Reaction Test Problem

A simple but challenging test problem that illustrates some of the challenges with the interpolation methods described above is given by the advection reaction equation

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 40(c-1)u(1-u), \quad where\ (x,t) = [0,2]x(0,1.5]. \tag{3.4}$$

The left boundary condition and the initial condition are given by

$$u(x,t) = 0.5(1 - tanh(20(x - ct) - 4.0)). \tag{3.5}$$

The discretization method used is the explicit flux-limited scheme with forward Euler integration [27]. The source term in this problem is similar to that in [27, 130]. Solution values outside the range $[0,1]$ will cause the source term to change sign, with potentially catastrophic results, particularly if the parameter c is not close to one. At every 10 timesteps, failure is modeled by the fine mesh solution at every other fine mesh point being replaced by an interpolated value from the coarse mesh. This model represents every fine mesh patch failing at every 10 timesteps and is a particularly stringent test of the need for appropriate spatial interpolation routines. In all cases, the error is the difference between the computed solution and the exact solution. Table 3.1 shows errors for the advection reaction problem using an even mesh of 1601 points. Different values of the constant c are used. When c is some distance from one, the numerical wave solution starts to lag behind the true wave solution. In all cases, the L1 error norm is calculated at time $t = 1.5$.

Four schemes are shown: linear interpolation, cubic interpolation, limited interpolation, and none, which is the standard scheme without using any interpolation. For the limited interpolation, the accuracy is close to that of the underlying numerical scheme. For values of $c \leq 0.7$, the overshoots introduced by cubic interpolation cause the solution to become unstable and the calculation to fail. Using linear interpolation surprisingly improves

Table 3.1: One-dimensional advection reaction equation L1 error norms CFL = 0.0125.

c	1.0	0.9	0.8	0.7	0.6
Linear	8.5e-3	9.5e-3	8.0e-3	5.0e-3	1.0e-1
Cubic	1.8e-4	1.8e-3	4.0e-2	NaN	Nan
Limited	1.8e-4	1.8e-3	4.3e-2	1.0e-1	1.7e-1
None	1.9e-4	1.9e-3	4.3e-2	1.0e-1	1.7e-1

accuracy in some cases, most probably as it adds extra diffusion due to the interpolation error.

3.3.2 Three-Dimensional Test Problems

Experiments to test the interpolation accuracy for three-dimensional problems were conducted on a single node with two Sandy Bridge processors (Intel(R) Xeon(R) CPU E5-2680), each with 8 cores and 90GB of RAM. ULFM version 2.0 was first built using GCC compiler version 6.1.0. Uintah was compiled using MPI wrappers provided by ULFM. Both builds used optimization flag -O3. ULFM was built by disabling the support for MPI_THREAD_MULTIPLE. Uintah was compiled with a flag -fopenmp to provide OpenMP support. OpenMP pragmas are used in both the timestepping code to simulate Burgers' equation and the Advection Reaction equation. They are also used in the interpolation routines to improve performance through data parallelism.

Simulations were run for different combinations of mesh points (ranging from 12^3 to 96^3), different numbers of ranks (4 to 64 by oversubscribing cores), different numbers of sequential failures (from 1 to 5), and different timesteps at which failure is induced. At any given time, half the ranks (odd or even) are killed simultaneously by raising the signal sigkill, and the remaining ranks take over execution. Errors in four cases - timestepping without failure, linear interpolation, ENO, and LENO - were measured and compared against the exact solutions.

Two problems were used to evaluate the accuracies of interpolants. The first is a Burgers' equation problem that illustrates the accuracy differences from the different interpolation approaches. The second problem is a three-dimensional version of the one-dimensional advection reaction that suffers from catastrophic failures when unconstrained cubic interpolation is used.

3.3.2.1 Burgers' Equation

The three-dimensional model Burgers' equation used here is equivalent to many of the equations in the Uintah applications in terms of its computational structure and reflects the types of PDEs in many different applications in areas such as fluid mechanics, molecular acoustics, gas dynamics, and traffic flow. The viscous Burgers' equation in one-dimensional for $u = \phi(x, t)$ is given by

$$\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = v\frac{\partial^2 \phi}{\partial x^2} \tag{3.6}$$

where v is the viscosity of the medium. The function $\phi(x,t)$ used for an exact solution for a given timestep at a given location is given by

$$\phi(x,t) = \frac{0.1 + 0.5 * e^{\frac{d-f}{v}} + e^{\frac{d-g}{v}}}{1 + e^{\frac{d-f}{v}} + e^{\frac{d-g}{v}}}$$

where d, f, and g are calculated as

$a = 0.05 * (x - 0.05 + 4.95 * t)$, $b = 0.25 * (x - 0.5 + 0.750 * t)$, $c = 0.5 * (x - 0.375)$ and $d = min(a, b, c)$.

if d == a then f = b and g = c

else if d == b then f = a and g = c

else if d == c then f = a and g = b.

The one-dimensional Burgers' equation can be extended to three-dimensional for $u = \phi(x,t)\phi(y,t)\phi(z,t)$ as

$$\frac{\partial u}{\partial t} + \phi(x,t)\frac{\partial u}{\partial x} + \phi(y,t)\frac{\partial u}{\partial y} + \phi(z,t)\frac{\partial u}{\partial z} = \nu(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}). \tag{3.7}$$

The initial and boundary conditions are given by

$$u(x,y,z,t) = \phi(x,t)\phi(y,t)\phi(z,t).$$

The main kernel for solving discretized Burgers' equation in Uintah is presented in Algorithm 2. To calculate on a patch, the kernel requires exactly one extra layer of ghost cells.

Fig. 3.2 shows the results of using the different interpolation methods for the Burgers' equation for viscosity $\nu = 0.01$ and domain sizes of 24^3 and 96^3 meshes, respectively.

1: **for** every cell `(i, j, k)` **in** current patch **do**
2: $u\_dudx = \phi(i * dx, dt) * (u_{i-1,j,k} - u_{i,j,k})/dx$
3: $u\_dudy = \phi(j * dy, dt) * (u_{i,j-1,k} - u_{i,j,k})/dy$;
4: $u\_dudz = \phi(k * dz, dt) * (u_{i,j,k-1} - u_{i,j,k})/dz$;
5: $d2udx2 = (-2 * u_{i,j,k} + u_{i-1,j,k} + u_{i+1,j,k})/(dx^2)$;
6: $d2udy2 = (-2 * u_{i,j,k} + u_{i,j-1,k} + u_{i,j+1,k})/(dy^2)$;
7: $d2udz2 = (-2 * u_{i,j,k} + u_{i,j,k-1} + u_{i,j,k+1})/(dz^2)$;
8: $du = -((u\_dudx + u\_dudy + u\_dudz) + \nu * (d2udx2 + d2udy2 + d2udz2))$;
9: $u^{new}_{i,j,k} = u_{i,j,k} + dt * du$;

Algorithm 2: Pseudo code for the Burgers' kernel.

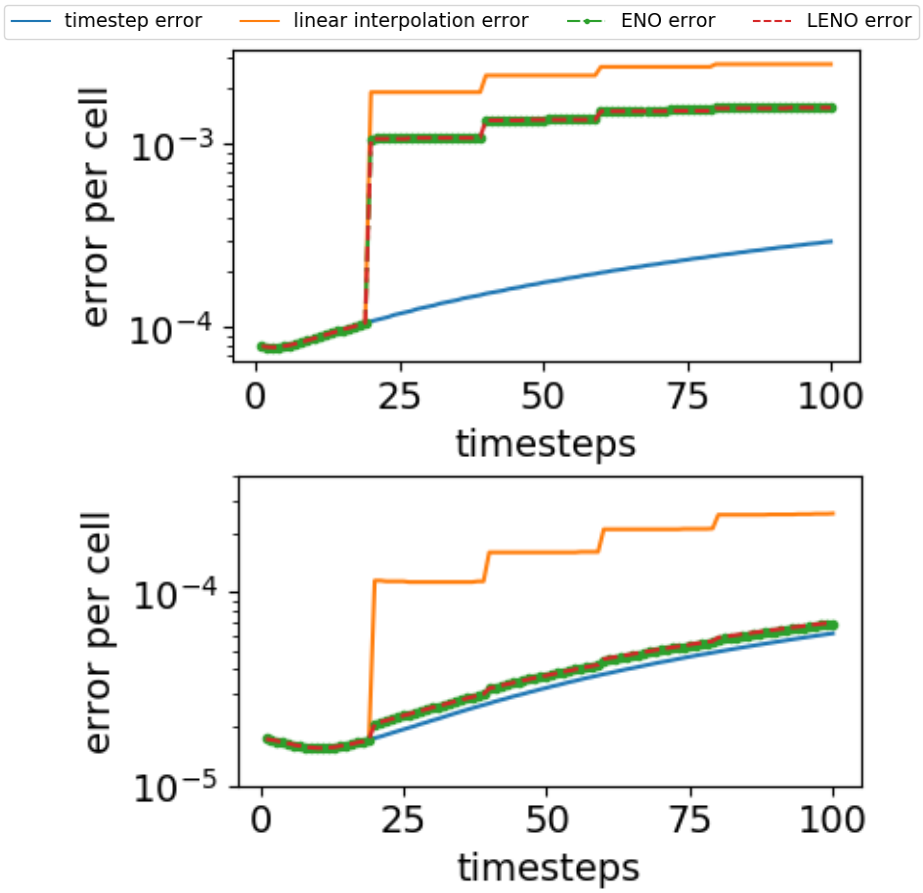


Fig. 3.2: Interpolation accuracy for Burgers' equation with viscosity of 0.01 on 24 and 86 cubed meshes.

The experiment spawned 32 MPI ranks and ran for 100 timesteps, with half the ranks failing simultaneously at the 20^{th}, 40^{th}, 60^{th}, and 80^{th} timesteps, respectively. Thus, only two ranks remained at the end. As the number of ranks was reduced, CPU cores remained unused. The sudden increase in the error indicates the time at which failure occurred. As the domain resolution increases, the accuracy of the ENO and LENO interpolants improves faster than that of linear interpolation. For the 24^3 domain, the ENO interpolation accuracy was 1.7x times better than that of linear interpolation, but for the 96^3 domain, ENO interpolation had 3.7x times better accuracy than linear interpolation. Although the ENO method performed better than linear interpolation, it caused overshoots in the numerical solution. The numbers of overshoots for the cases shown in Fig. 3.2 were 17,112 and 25,475, respectively. However, the impact of overshoots is to some extent compensated for by the small value of the timestep and is not significantly reflected in the error.

Using the LENO interpolant eliminates these overshoots by discarding the terms causing overshoots. The LENO approach marginally affects the accuracy. In the case of the 24^3 domain, LENO improved the accuracy over ENO by 2% but degraded the accuracy by 2% for the 96^3 domain. Similar results were obtained for a three-dimensional heat problem, but are omitted for reasons of brevity.

Table 3.2 compares errors of the numerical solution using various interpolation schemes, namely linear interpolation, ENO and LENO to the exact solution, whereas "None" represents the error without any failures/interpolation schemes.

3.3.2.2 Three-Dimensional Advection-Reaction Equation

The advection-reaction equation used in the one-dimensional test can be extended to three space dimensions as

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} + \frac{\partial u}{\partial z} = 40(c-1)u(1-u). \tag{3.8}$$

Table 3.2: Three-dimensional advection reaction equation errors compared to the exact solution.

c	1	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
None	2.40E-02	1.08E-01	2.13E-01	2.77E-01	2.81E-01	2.31E-01	1.55E-01	8.44E-02	3.61E-02	1.01E-02
Linear	2.44E-02	1.13E-01	2.26E-01	2.94E-01	2.98E-01	2.44E-01	1.62E-01	8.73E-02	3.67E-02	1.01E-02
ENO	2.41E-02	1.08E-01	2.14E-01	2.77E-01	NaN	NaN	NaN	NaN	3.66E-02	1.02E-02
LENO	2.41E-02	1.08E-01	2.15E-01	2.82E-01	2.88E-01	2.41E-01	1.56E-01	8.44E-02	3.65E-02	1.02E-02

The boundary conditions at $x = 0, y = 0$, and $z = 0$ and the initial condition are given by

$$u(x, y, z, t) = \phi((x + y + z)/3, t) \tag{3.9}$$

where $\phi(x, t)$ is calculated using $u(x, t)$ in Equation 3.5, and $(x, y, z, t) = [0, 2]x[0, 2]x[0, 2]x(0, 1.5]$. In this case, 32 MPI ranks were used, with half of the ranks crashing at the 100^{th}, 200^{th}, 300^{th}, and 400^{th} timesteps. A domain size of 24^3 mesh points is used for $(x, y, z) \in [0, 2]x[0, 2]x[0, 2]$. For this equation, overshoots and undershoots caused by using the ENO method lead to infinite values of the solution for $0.3 \leq c \leq 0.6$. However, LENO methods avoid such errors, which confirms that the accuracy results for the one-dimensional problem translate into three space dimensions. Similar results were obtained for the Leveque-Yee model problem [130] in one and three spatial dimensions.

3.3.3 Experiments to Measure Scalability

Preliminary strong scaling experiments were conducted to compare the performance of the new ABFT approach with checkpointing using 128 nodes of the Quartz cluster at the Lawrence Livermore National Laboratory (LLNL) [11]. Uintah has its own checkpointing and restart functionality. Users can choose a checkpointing interval and the type of I/O process among (i) I/O per process, (ii) I/O per N processes (i.e., one process collects output from N processes and performs I/O), and (iii) using the PIDX [125, 127, 128] high-performance I/O library. For a small node count, I/O per process performs as well as PIDX on the Lustre file system, which is used by the Quartz cluster [126]. For the modest size node counts used in this study, Uintah's built-in I/O per process is used for checkpoint and restore and performs well in comparison to more sophisticated approaches such as PIDX [125, 127, 128].

Each node of Quartz is equipped with 36 cores of the Intel Xeon E5-2695 v4 processor(s) and 128 GB RAM. ULFM and Uintah were compiled in the same way as described in the accuracy experiments using GCC 6.1.0. Experiments were conducted using the three-dimensional Burgers' equation with a fine mesh of size 256^3 points and a coarse mesh of 128^3 points (i.e., the refinement ratio of two). Both grids were divided into 4096 patches (total 8192 patches). Thus, the fine patch size was 16^3, and the coarse patch size was 8^3.

3.3.3.1 Failure Model

Strong scaling starting from two nodes up to 128 nodes was carried out with one rank per node. During every run, half of the ranks (odd or even) raise "sigkill" and simultaneously crash. Soon after the crash, MPI detects an error, and the custom error handler kicks in. Unlike the model used to verify the accuracy, failures occur at only one instance, i.e., at the 20^{th} timestep. Thus, for every crash, half of the data (i.e., 2048 patches) is interpolated by the surviving ranks. Although this is not a realistic failure model with half of the nodes failing at the same time, the point of considering this worst-case scenario was to demonstrate the effectiveness of this approach with extreme failure rates. Different studies have been conducted in the past to categorize failure rates and types [60, 81, 137]. For example, Meneses [137] shows that only 29% of the software failures during 2014 affected more than four nodes, and only 2% of the hardware failures affected four or more nodes on the Titan supercomputer. However, the "mean time between failures" (MTBF) of Titan is around 27 hours [137]. On the other hand, the DARPA ExaScale Computing Study [35] suggests that the MTBF at exascale could be as frequent as 35-39 minutes. This 45-fold increase in the failure rate indicates the possibility of multiple instances of failures within the lifetime of a job. Some of these failure instances might have concurrent node failures as well. Thus, our failure model assumes that half of the nodes crash simultaneously to prepare Uintah for the worst-case scenario. Fig. 3.3 shows the overhead incurred by the resiliency tasks during normal timesteps. The "not resilient" plot shows time per timestep when resiliency is turned off. The "resilient" plot shows wall time per timestep for the resilient approach and includes time for tasks to generate coarse patches and perform the

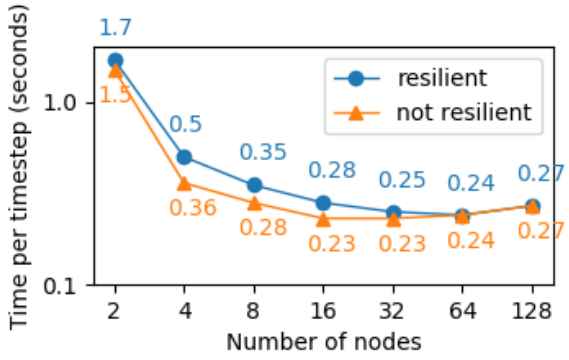


Fig. 3.3: Overhead of resiliency on LLNL Quartz cluster.

coarse patch exchange between neighbors. The plot does not include the recovery phase, which is analyzed later. The difference between the two plots shows that the overhead of resilience is around 38% for four nodes when each node processes 1024 patches. As the number of nodes increases, the number of patches per node decreases, resulting in less overhead. Increasing the node count to 64 and 128, the number of patches per node decreased to 64 and 32, respectively, with the resultant overhead decreasing to less than 10 milliseconds. Typically, Uintah's domain decomposition strategy usually assigns one to two patches per core. For Quartz's 36 core nodes, using this workload of 32 or 64 patches per node matches Uintah's domain decomposition scheme for realistic simulations.

3.3.3.2 Performance Evaluation

To analyze the performance of resilience, measurements are made of 1) the time to coarsen patches; 2) MPI communications during regular timesteps; and 3) the time to detect and recover from an error and interpolate lost data during the recovery timestep. Similar experiments were repeated using checkpoint/restarts instead of ABFT resilience. The time spent in IO tasks during checkpointing and in recovery was also measured. Again, the MPI communication time was also measured for the timesteps in which checkpointing does not take place. These timing values were also used to deduce the per-patch overhead in both methods and to measure the communication overheads due to coarsening, patch exchange, and the per-patch recovery time. Three metrics were used to compare the performance of ABFT-based resiliency with checkpointing within Uintah.

1) Per patch per timestep time for backup and recovery: For ABFT-based resiliency, this metric is a sum of the computation and communication wait times needed to execute coarsening tasks per timestep, the time ULFM takes to detect a set of failed ranks (in the case of simultaneous failures), and the subsequent recovery and interpolation operations. For the checkpointing approach, this metric is the sum of the time taken by one checkpoint operation (equivalent to one coarsening and exchange task of ABFT during one timestep) and the time taken for one recovery from the checkpoint. Coarse patches are exchanged at every timestep, whereas checkpointing takes place after every four timesteps. Both these frequencies can be adjusted depending upon the problem to be solved. For a fair comparison, only one checkpoint-restart was

timed and compared against one coarsening + exchange and ABFT recovery. This approach allows for a comparison of the raw execution times. Fig. 3.4(a) shows that ABFT-based solution performed 3.4x faster than checkpointing on two nodes, and the gap increased with the number of nodes. ABFT performance was almost linear up to 64 nodes, after which it starts to degrade. However, checkpointing performance degrades as the number of nodes increases. At 128 nodes, the performance boost from ABFT becomes 20X.

2) MPI overhead exchange of coarse patches: In the ABFT approach, the coarsening and exchange tasks execute more frequently than recovery. These tasks should have minimal overhead. The coarse task did not need any communication and was observed to be scalable from 125 ms for two nodes to 1.8 ms for 128 nodes (not shown in the chart for simplicity). However, exchanging coarse patches adds a small amount of overhead during MPI communication, as shown in Fig. 3.4(b). The difference between the MPI wait time of Uintah when coarse patches are exchanged, and the MPI wait time when coarse patches are not calculated and not exchanged,

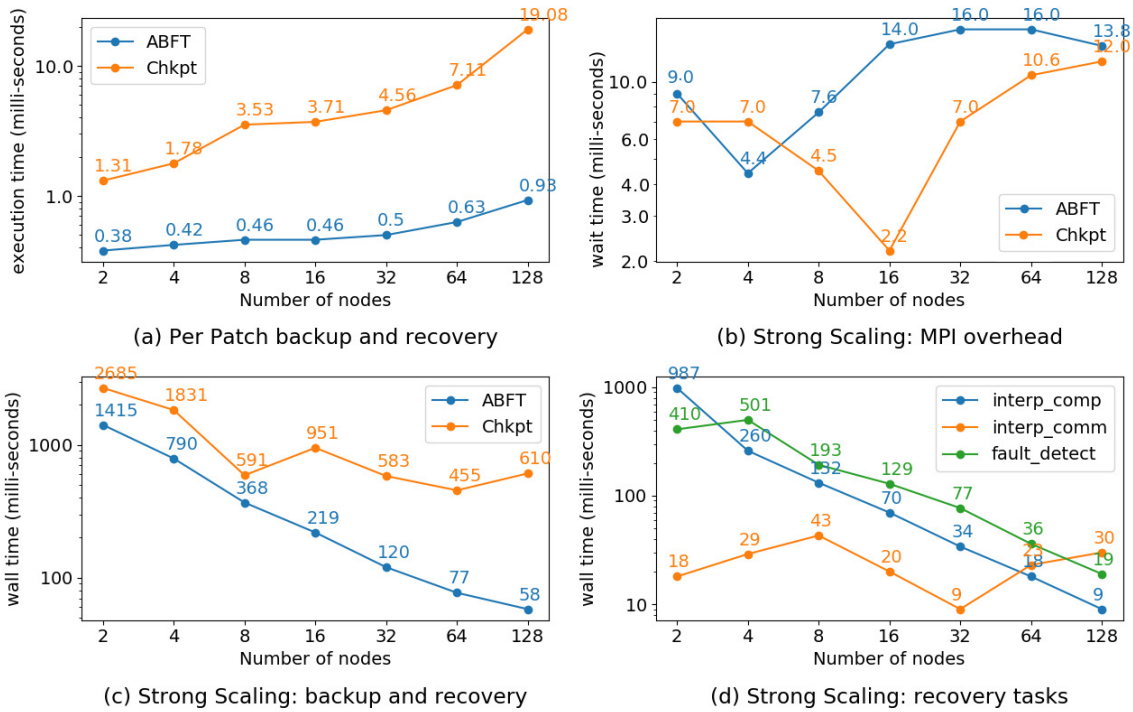


Fig. 3.4: Performance comparison on LLNL Quartz cluster.

shows the MPI overhead due to the coarsening tasks.

Uintah is designed to overlap computation and communication. However, if coarse patches are exchanged in an overlapped fashion, there is a risk of failure before the patch exchange is completed, and the recovery will become impossible. To avoid this situation, no other tasks are allowed to execute before coarse patch exchange is completed during every timestep. A communication overhead is visible in Fig. 3.4(b), even for eight nodes where each rank has enough computation to effectively hide this exchange. As the number of nodes increases, the number of coarse patches exchanged per node decreases, and the overhead decreases from 700% for 16 nodes to 10% for 128 nodes. This overhead can be further reduced by introducing a timestep lag during the exchange, and thus the recovery routine will have to execute one extra timestep. At 128 nodes, communication starts dominating computation (for the entire calculation and not just coarsening tasks). The effect can be seen in Fig.s 3.4(a) and (c) as the number of nodes increases to 128. This strong scaling pattern of coarsening computations and MPI overheads concurs with Fig. 3.3, where the resiliency overhead decreases as the number of nodes increases.

3) Strong scaling of backup and recovery: This metric compares the wall clock time of similar operations involved in the per patch metric, but now the wall clock time to coarsen/recover or checkpoint/recover all patches is measured. This metric demonstrates strong scaling of both approaches. Fig. 3.4(c) shows that ABFT scales better than checkpointing. There is one seeming contradiction between Fig. 3.4(a) and Fig. 3.4(c). The speed-up in Fig. 3.4(a) is twice that of the speed-up shown in Fig. 3.4(c). The reason for the apparent paradox is the number of ranks is halved after the recovery from a failure. In other words, for 128 nodes, one cannot expect the wall clock time to be 0.93 (per patch time) * 4096 (number of patches) / 128 (number of nodes) = 29.76 milliseconds. After the failure, only 64 nodes execute the program and, hence, the expected wall time will be 0.93 (per patch time) * 4096 (number of patches) / 64 (number of nodes) = 59 milliseconds, which is close to the observed value of 58 milliseconds. Experiments also demonstrated that, in the case of multiple simultaneous failures, the wall clock time to recover from one failure remains the same as the wall clock time to recover from 64 failures, as recovery takes place in

parallel. Fig. 3.4(d) gives a breakdown of the wall clock time in three categories - error detection + state recovery, interpolation, and the MPI wait time for halo exchange required by interpolation tasks. Due to the use of OpenMP loops, interpolation tasks execute faster than error detection + state recovery, which is a single-threaded task.

3.3.3.3 Communication Overhead and Scaling

The communication time, including communication needed for the simulation and the overhead of coarsening tasks, is presented in Fig. 3.4(b). The figure shows how communication time scales at the beginning and fails later as communication becomes more dominant. For the checkpointing-based approach, communication scales better than ABFT because checkpointing spends a longer time in IO, which dominates MPI internode communication. In either case, communication is performed locally, and the overall scaling (Fig. 3.4(c)) depends on the balance between computation and communication and how well AMT can hide communication. However, at the end of the strong scaling, both approaches spend nearly the same in communication, which automatically offsets the overhead of the coarse patch exchange. The overhead of coarse patch exchange can be asymptotically computed as follows. Considering one n^3 patch on each rank, the simulation needs to exchange n^2 cells with each neighbor, thus amounting to maximum of 12 MPI messages (six sends and six receives). The current coarsening strategy exchanges a coarse patch of size $n^3/8$ using two MPI messages (one send and one receive) with one neighbor. Hence, the overhead in terms of the number of messages becomes $2/12 \times 100 \approx 17\%$. The overhead of message size is $(n^3/8)/(6n^2) = n/48$, i.e., the coarse patch exchange will not dominate the communication required by the simulation for $n < 48$. The current value $n = 16$ falls well within the limit and causes negligible overhead. These limits on the size of n can be made more flexible in the future by 1) using a larger refinement ratio instead of two; 2) coarsening at an interval of a few timesteps instead of every timestep to amortize the overhead, in which case the simulation has to be rolled back to the last interval on failure; 3) combining the extra MPI messages for coarse patches with other simulation messages; and 4) compressing the coarse patches exchange.

3.3.3.4 Comparison to High-Performance Checkpointing

PIDX is a high-performance I/O library built for high-performance computing, and performs better than the state-of-the-art I/O libraries [125, 127, 128]. Uintah with PIDX was able to utilize 80% of the theoretical disk bandwidth on MIRA [126]. Compared to naive per process I/O, PIDX achieved a speed-up of 2x to 10x as the number of processes increased from 8192 to 262,890 [126]. However, the speed-up on Edison was limited to 0.7x to 1.3x as the number of cores increased from 1024 to 8192. This difference in performance occurs because Edison's Lustre filesystem performs better while handling large numbers of files compared to MIRA's GPFS file system [126]. Charm++ has a resiliency capability using in-memory checkpointing on the "buddy" process, which provided about 100x speed-up over checkpointing [199]. Zheng [198] further optimized the performance that resulted in checkpointing and restart finishing in a few milliseconds. This performance is as good as the ABFT performance achieved by resilient Uintah. Dong [64] was able to reduce the checkpointing overhead to $2-3\%$ using nonvolatile RAM (NVRAM) for checkpointing. In a similar work, Kannan [117] reported checkpointing overhead to 6% of the total run time.

In spite of these advances, checkpointing has it own challenges. Its performance also depends on environmental factors using shared resources. Moody [143] experienced an increase from 3.5 minutes (for 4096 processes on 512 nodes) to 1.5 hours (for 8192 processes on 1024 nodes) for checkpointing. The drastic slowdown was due to the load from other jobs and not the increased number of ranks [143]. The MTBF increased from 1.5 hours to 1.5 days when the Scalable Checkpoint/Restart (SCR) library was used instead of traditional checkpointing on the Atlas cluster at the Lawrence Livermore National Laboratory [143]. Nevertheless, it will be interesting to compare the performance of Resilient Uintah with these techniques at scale.

3.4 Limitations of ABFT+AMR+ULFM Approach

The approach of combining ABFT, AMR, and ULFM to achieve resiliency may not work in a few cases in which checkpointing will work. If the accuracy requirements do not allow coarsening and interpolating the grid, then the fine grid itself has to be exchanged among ranks, which may consume excessive memory and network bandwidth. Alternatively, the fine mesh can be compressed to reduce the size but would need more computing power.

Another scenario is when all the ranks with a backup of a particular patch fail, and none of the survivor ranks can recover the failed patch. A more severe version of this problem will be when the majority of the ranks fail due to network outage, or from the failure of the entire rack of compute nodes, etc. In such cases, checkpointing and restart will still work.

3.5 Conclusion and Possible Future Work

The combination of ULFM, AMR, and interpolation used here has been shown to be a faster recovery method than using the standard approach of checkpointing. The recovery routines were able to recover from repeated simultaneous failures until only one rank remained. Furthermore, recovery is treated as just another task that requires only a load-balancing step, which makes this approach more flexible without placing an extra burden on the runtime system. The combination of AMT and ABFT approaches has the potential to completely avoid delays and interruptions because survivor ranks can always continue the execution of other tasks while recovery is in progress.

Using ULFM to detect failed ranks and using interpolation for patch recovery provides a lightweight approach to ensure that Uintah is more resilient in the face of future generations of architectures with significantly higher MTBF.

The results presented here demonstrate the effectiveness of using limited ENO over linear interpolation and ENO methods to recover data lost by node failures. The Limited ENO method provides physically meaningful solution values that do not cause difficulties with associated physics routines. The combination of AMR with physically appropriate interpolation methods along with fault-tolerant ULFM and simple recovery tasks provides a faster resiliency mechanism for AMTs on future architectures.

Many opportunities exist to optimize the current implementation and compare it against other resiliency approaches.

The dynamic load-balancing capability of AMTs can minimize any load imbalance created by the recovery process. Overlapping the communication and computation while exchanging coarse patches will significantly reduce the resiliency overhead - especially at a smaller node count. An alternative to AMR and interpolation is the compression and duplication of data to neighboring nodes, which can be transferred and uncompressed to recover from failures. The ABFT approach has the advantage of strong scaling and

communication only to neighboring nodes. Hence, it will be interesting to compare the performance of ABFT at a large scale with the latest checkpointing methods that use nonvolatile RAM and in-RAM.

Future work may explore an option to allocate extra "reserve" nodes and ranks at the beginning, and to manipulate the MPI communicator to maintain a fixed number of "active" ranks. On failure, reserve ranks can replace failed ranks. The challenge for this approach will be passing the active state of the simulation to the reserve ranks. Finally, increasing the number of nodes for the exchange of coarse patches from the current design of a single node to two or more nodes will allow a greater level of fault tolerance and flexibility.

CHAPTER 4

PORTING AMT TO NONCONVENTIONAL ARCHITECTURES: EXPERIENCE AND LESSONS LEARNED

As discussed in Chapter 1, AMTs show a promise to run efficiently on exascale and post-exascale systems. Hence, it becomes important to experiment and understand the porting strategies, efforts required to port different components of AMT and the user code, the efficiency of the ported code, and the productivity of the entire exercise to port the AMT systems to a new, unusual architecture. This chapter presents a case study of running the Uintah AMT to the Sunway TaihuLight supercomputer [74] as described in [192]. This work shows how an asynchronous Sunway-specific task scheduler, based on MPI and athreads (a native threading library for Sunway), makes it easy to port the AMT runtime, and how individual task-code for a model structured-grid fluid-flow problem can be refactored. The research was carried out in collaboration with Zhang Yang.

The Sunway TaihuLight was the world's fastest supercomputer at the time and currently holds the fourth position in the June 2021 version of the top-500 list [12]. Many real-word applications were able to run at substantial fractions of peak performance on Sunway, e.g., the three applications selected as Gordon Bell Award finalists in 2016 [160, 191, 197]. However, these performance levels were obtained through extensive and intensive rewriting and tuning of the code at a level that may not be possible or affordable for general application codes. On the other hand, as discussed earlier in Chapter 1, the AMT runtimes can easily overlap computation and communication, take advantage of deep memory hierarchies, offload kernels asynchronously – all with minimal development efforts from the application developers, and allows running efficiently on the novel architectures. The decoupling of infrastructure and the simulation code in AMTs allows the independent development of both layers. The methodology can save a huge amount of porting efforts

not only for the Sunway architecture, but also for the next generation supercomputers and different unexplored architectures. Hence, it becomes essential to study the adoption techniques for AMTs on Sunway. Uintah is the first AMT runtime system ported to the Sunway TaihuLight supercomputer.

4.1 Challenges Arising from the Sunway Architecture

The central challenge in running the Uintah Computation Framework on Sunway is to devise an appropriate scheduler and write tasks to take advantage of the unique Sunway architecture. The 100PF Sunway TaihuLight features a special architecture consisting of 41K nodes with SW26010 processors, where each processor contains four Core-Groups (CGs). Each CG is made up of one Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs) sharing the same main memory as described below. Each CPE is equipped with a small user-controlled scratchpad memory instead of data caches. Thus, the most important factors to extract performance from the Sunway architecture are offloading the tasks to CPEs, vectorization of offloaded code, using the scratch-pad memory, and maximizing the CPE utilization by minimizing the wait time involved in the communication.

The main contribution of the current author is to develop an Asynchronous Task Scheduler, where the main thread *asynchronously* "offloads" the ready tasks into a task queue, and keeps checking for MPI messages by the time worker threads execute the ready tasks. In the next step, Zhang Yang replaced the worker threads based on C++ std::threads with 'athreads', the native threading model of Sunway, and used Sunway-specific intrinsics for vectorization. The experiments carried out using asynchronous offloading showed the potential to maximize the CPE utilization and also effectively overlap computation and communication.

4.2 A Model Fluid-Flow Problem

Burgers' equation, one of the hello world problems in Uintah, is used as a model problem for porting to the Sunway architecture. Burgers' equation was described earlier in Section 3.3.2.1. It resembles typical Uintah applications, which gather the ghost cells from the neighboring ranks.

4.2.1 Performance Characteristics

The Burgers' kernel combines a low-order stencil structure with complex coefficient evaluations, typically seen in scientific and engineering applications. The low order of the stencil prevents excessive data reuse optimizations. This complexity reflects some of the stencils of real applications in Uintah. With the presence of exponentials in the kernel (see Algorithm 2), it is not possible to count the floating point operations of the kernel directly. Instead, Zhang Yang carried out experiments with varying problem sizes, and the precise hardware counters on SW26010 were used to measure floating point operations. Table 4.1 displays the Flops per cell of the model problem are around 311, 215 of which are contributed by the exponentials. Given the 16 byte memory access required per cell as indicated by Algorithm 2, the arithmetic intensity of the kernel is approximately 19.4 Flop/Byte, and is still memory-bounded compared to that of the SW26010 processor.

4.3 Architectural Features of Sunway TaihuLight

Sunway TaihuLight has a theoretical peak performance at 125.4 Pflop/s and HPL benchmark performance at 93 Pflop/s. Like other supercomputers nowadays, Sunway employs an MPP (Massively Parallel Processing) architecture. The system is composed of 40,960 SW26010 processors connected with a proprietary high-speed network. The architectural features relevant to this work are described here. Some important system parameters of Sunway are summarized in Table 4.2, and more details can be found in [13,74].

Table 4.1: Flop per cell for the model problem.

Problem Size	Total Cells	Total Flops	Flops per Cell
16x16x512	17339400	5179553014	299
16x32x512	34412040	10399936968	302
32x32x512	68294664	20881857690	306
32x64x512	136059912	41845438269	308
64x64x512	271065096	83854642144	309
64x128x512	541075464	167873049894	310
128x128x512	1080045576	336073950828	311

Table 4.2: Major system parameters of Sunway TaihuLight.

Item	Description
Node architecture	1 SW26010 processor
Node cores	4 MPEs + 256 CPEs, 260 cores
Node memory	32GB, 4*128bit DDR3-2133
Node performance	3.06 Tflop/s
Interconnect Bandwidth	Bidirectional P2P 16 GB/s
Interconnect Latency	around 1 μs

4.3.1 Architectural Features of the SW26010 Processor

As shown in Fig. 4.1, the SW26010 processor is composed of four identical core-groups (the CGs) connected by an on-chip interconnect network. Each CG is a heterogeneous computing unit consisting of one 128bit-wide DDR3-2133 memory controller (the MC), one Management Processing Element (the MPE), and a cluster of 64 Computing Processing Units (the CPEs). The usual and recommended practice is to use these CGs as separate computing nodes; thus, 'CG' and 'computing node' are used interchangeably in this chapter.

The most significant architectural feature of SW26010 is the "shared-memory MPE+CPE heterogeneous architecture." The MPE and CPEs share the same memory space connected to the memory controller, and can communicate directly via the main memory, which enables lightweight kernel offloading to the CPEs. Although the MPE and CPEs have different Instruction Set Architectures (ISAs), both are based on a RISC architecture, 64 bit, SIMD, and out-of-order microarchitecture. So both can run the user's application. The performance of the MPE is 23.2 Gflop/s, and that is 742.4 Gflop/s for the cluster of CPEs. Since the MPE contributes only 3% of the aggregated performance, little is sacrificed if the MPE is excluded

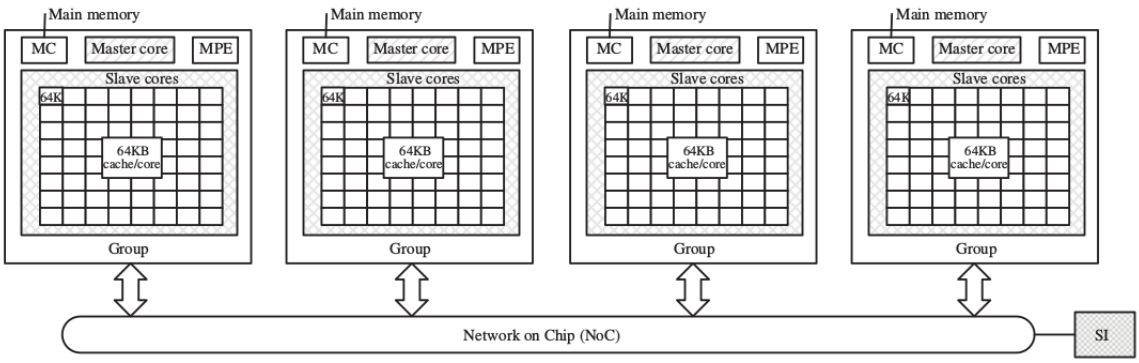


Fig. 4.1: Sunway SW26010 architecture [74] (Reprinted with permission from Springer Nature).

from intensive floating point computing. This architecture is suitable for asynchronous many-task runtimes since overlapping communications (as well as other runtime-related tasks) and computations with a low-overhead offloading interface is possible. SW26010 also provides instructions for CPEs to atomically increment a 4 or 8 byte location in the main memory, which helps the MPE to monitor the progress of the CPEs with little overhead.

Another interesting feature of the SW26010 is the introduction of user-controlled scratchpad memory. Whereas the MPE is equipped with 32KB/256KB hardware-controlled L1/L2 data cache, the CPEs are cacheless. Instead, an on-chip 64KB scratchpad memory, dubbed "Local Data Memory" (the LDM), is attached to each CPE. The CPE can use both the main memory and the LDM in computing, but the LDM enjoys a higher bandwidth and lower latency. To get the best performance, the application has to move data explicitly between the LDM and the main memory and use only the LDM as working memory. To assist the move, SW26010 provides DMA mechanisms that can move data between the LDM and the main memory asynchronously.

Manually overlapping communications with computations, together with explicitly managing the transfer between the LDM and the main memory, requires significant coding effort for general applications, and even more effort is required to make good use of these features. The expense of this motivates us to explore alternative porting options based on AMT runtimes such as Uintah which are a natural fit for the asynchronous nature of the operations needed on Sunway.

4.3.2 The MPI+athread Programming Interface

Sunway supports MPI for internode communications. On a single computing node, OpenACC is supported to allow the offload of computations to the cluster of CPEs (see [14]). However, the Sunway OpenACC interface does not expose all the features of SW26010, and the current implementation does not support OpenACC runtime functions such as `acc_async_test`. For this reason, a low-level atheads interface is used here.

The use of athreads provides a low-level offloading interface for the cluster of CPEs. Conceptually, an athread is a lightweight thread binding to one CPE. The athread library provides mechanisms for the MPE to create a group of athreads running a given function. APIs are provided for the offloaded function to transfer data asynchronously between the

LDM and the main memory through DMA operations, as well as to manage the environment on CPEs. athread provides enough freedom for the runtime system to take full control over the data transfer, the execution, and the scheduling.

4.4 Adapting Uintah to Sunway TaihuLight

This section describes how Uintah was adapted to Sunway TaihuLight.

4.4.1 Building Uintah on Sunway

Zhang Yang built Uintah on Sunway using Sunway's GNU compiler for C++, and compiled C code for the CPE using Sunway's native compiler.

4.4.2 Design of an Asynchronous Sunway-Specific Scheduler

To best use the Sunway's architecture, a new asynchronous task scheduler was introduced in Uintah. The scheduler was designed to facilitate:

- Adaption of the shared-memory MPI+CPE heterogeneous architecture and reduction in the scheduling overhead;
- Improved kernel performance with the per-CPE scratchpad memory.

The current author did not have access to Sunway. Hence, the new multithreaded asynchronous scheduler was first developed in an offline mode for Intel Knights Landing (KNL) node using C++ std::threads. The single-threaded "Dynamic MPI scheduler" [142] was used as a baseline for the new scheduler. As shown in Fig. 4.2, scheduling logic involves the following steps:

1) The main thread picks up the "ready" tasks, the tasks whose internal and external dependencies are satisfied. Internal dependencies correspond to the earlier tasks in DAG running on the same node that feed input to the current task. External dependencies are composed of the tasks running on the different ranks, and the data are brought in by MPI receives.
2) The scheduler maintains a lock-free task queue using atomics. The main thread pushes the ready tasks into this task queue.

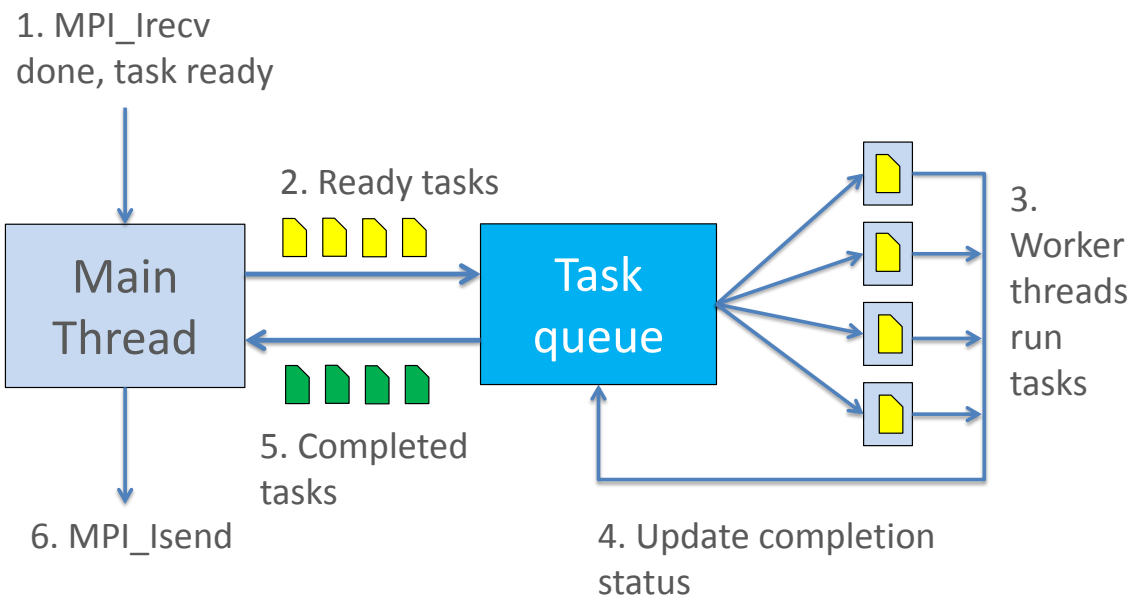


Fig. 4.2: Uintah's asynchronous scheduler using std::threads.

3) The worker threads spawned by the scheduler keep monitoring the task queue, and steal and execute the ready tasks as soon as the threads are free. The lock-free constructs using atomics keep the queuing overhead minimal.

4) The worker threads update the task status in the queue once the execution is completed.

5) The main thread detects the completed status, and removes the task from the queue.

6) Finally, MPI messages are sent out, if any of the tasks on the neighboring ranks are dependent on the result of the completed tasks.

While the worker threads are busy running tasks, the main thread keeps looping over steps 1, 2, 5, and 6. The scheduler "simulates" offloading behavior of the Sunway and provides the needed asynchrony. The asynchronous execution effectively overlaps communication with computation, keeps the worker threads busy with enough work, and minimizes the waiting time in MPI. The existing components, such as task graph creation, load balancing, automatic generation of MPI messages, etc., were reused by the new scheduler, and worked smoothly without any changes. One shortcoming of the Sunway software stack was the CPEs supported C and Fortran, but not C++. Therefore, the Burgers' kernel was rewritten in C.

In the next phase of development, Zhang Yang replaced the std::threads with "athreads," a native threading library for Sunway used to offload the kernels to CPEs, and the whole AMT just worked on Sunway. He also did Sunway specific optimizations: using Local Data Memory (LDM) and SIMD intrinsics for vectorization. Thus, on Sunway the entire management such as task graph generation, load balancing, etc., task scheduling, and MPI communication handled by the main thread runs on the MPE, and the kernels are asynchronously offloaded to the CPEs, as shown in Fig. 4.3.

4.5 Performance Evaluations

This section validates our effort porting Uintah and evaluates the performance of the Burgers' model fluid-flow simulation on Sunway TaihuLight by measuring the strong scalability and effectiveness of the asynchronous scheduler. A more detailed comparison between different performance optimizations and floating point efficiency can be found in [192]. As the current author did not have access to Sunway, all the experiments were run on Sunway by Zhang Yang.

4.5.1 Experimental Settings

The Burgers' equation is discretized on a rectangular grid and is run for 10 timesteps for performance evaluation purposes. The grid is partitioned into equally sized patches for

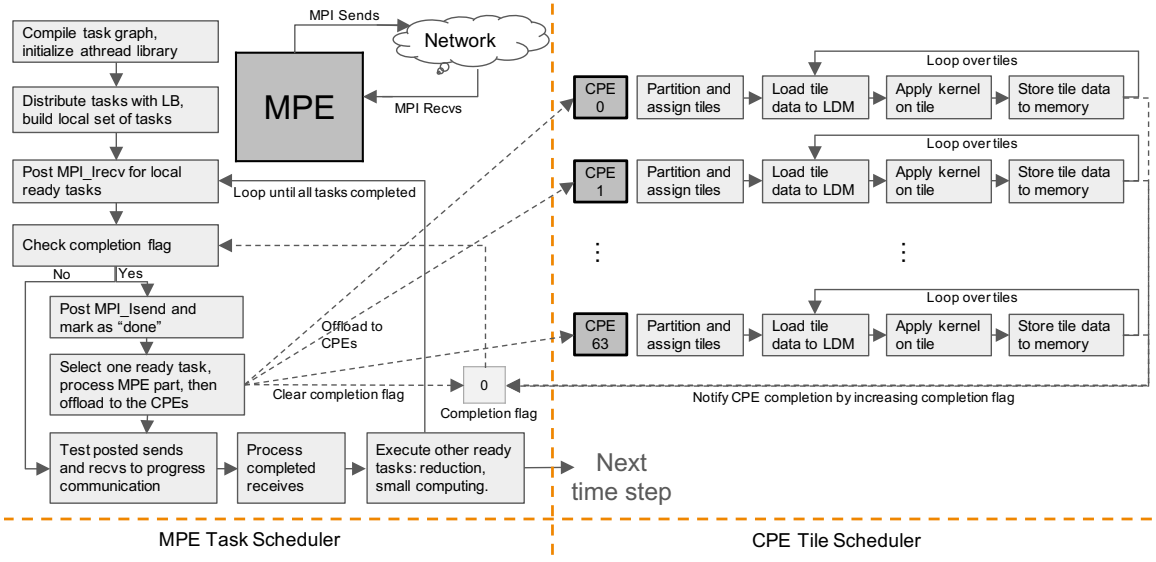


Fig. 4.3: Uintah's asynchronous scheduler for Sunway TaihuLight.

parallelization. One patch is scheduled for execution on one CG at a time. As only access to the Sunway experimental queue at the moment was available, experiments are limited from 1 to 128 CGs (8320 cores). The grid is partitioned into 128 patches with a fixed 8x8x2 patch layout, i.e., eight patches along the x and the y axis, and two patches along the z axis. A full set of patch sizes is chosen to represent typical cases in the following way: starting from the smallest possible patch, double the size in a round-robin way among the x and y axes each time, until a patch size exceeds the data exceeds the memory limit of one CG. As the tile size used is 16x16x8, and 64 CPEs per CG are used, the smallest patch is 16x16x512. The detailed problem settings are presented in Table 4.3. For each problem in Table 4.3, a strong scalability experiment is run from the smallest possible number of CGs to 128 CGs. The problem size 64x64x512 crashes with memory allocation errors when using 1 CG, so more than 1 CG is used in such cases (stared in the table). For each case in the experiments, different variants of the scheduler and optimization combination as defined in Table 4.4 are run. Every variant is compiled with the `-fPIC -O3 -DNDEBUG` flags to ensure performance. To mitigate the instabilities in the machine, each case is repeated multiple times, and the best result is selected. The wall time per timestep in each experimental case is used as the performance indicator.

Table 4.3: Problem settings in the evaluations.

Problem	Patch Size	Grid Size	Mem	Min
16x16x512	16x16x512	128x128x1024	256MB	1CG
16x32x512	16x32x512	128x256x1024	512MB	1CG
32x32x512	32x32x512	256x256x1024	1GB	1CG
32x64x512	32x64x512	256x512x1024	2GB	1CG
64x64x512*	64x64x512	512x512x1024	4GB	2CGs
64x128x512*	64x128x512	512x1024x1024	8GB	4CGs
128x128x512*	128x128x512	1024x1024x1024	16GB	8CGs

Table 4.4: Experimental variants in the evaluations.

Variant	Scheduler Mode	Tiling	Vectorization
host.sync	MPE-only	No	No
acc.sync	synchronous MPE+CPE	Yes	No
acc_simd.sync	synchronous MPE+CPE	Yes	Yes
acc.async	asynchronous MPE+CPE	Yes	No
acc_simd.async	asynchronous MPE+CPE	Yes	Yes

4.5.2 Strong Scalability

In evaluating the strong scalability of this simulation, the 'host.sync' variant is excluded since it uses only the MPE. The wall times of different variants on different problems are shown in Fig. 4.4, which indicates that this simulation has good strong scalability on all problems sizes, with both the synchronous and the asynchronous scheduler. Strong scalability persists with the vectorized kernel for which the computing time is reduced by half, which suggests that the async scheduler handles the communication and other tasks in a scalable manner.

4.5.3 Effectiveness of the Asynchronous Scheduler

The effectiveness of the asynchronous scheduler is calculated using the GFlop improvement of the scheduler running in asynchronous mode over synchronous mode as $(T_{sync} - T_{async})/T_{async}$.

The performance improvements on the nonvectorized and vectorized variants are

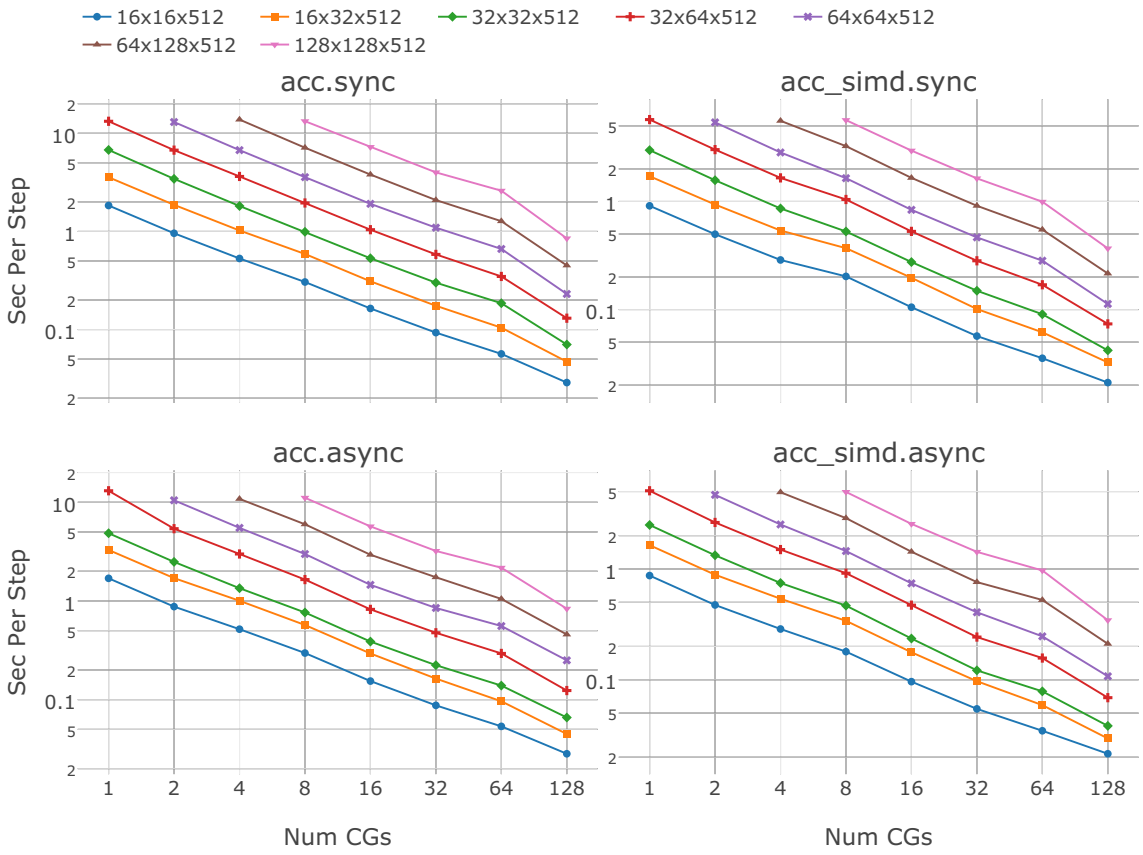


Fig. 4.4: Wall time of strong scaling different problems.

presented, respectively, in Table 4.5 and 4.6. The asynchronous mode is a clear winner, as it outperforms the synchronous mode in almost all cases, with an average improvement of 13.5%. The improvement varies for different problems. Medium-sized problems such as 32x32x512 and 32x64x512 show the biggest improvements. The best improvement is 39.3% for the nonvectorized kernel and 22.8% for the vectorized kernel. The green line in Fig. 4.4 shows strong scaling for the patch size 32x32x512, and the comparison between sync and async modes shows the improvement in the execution time. Even with only one CG, performance improvements are still observed. Smaller improvements are seen with the vectorized kernel than the nonvectorized kernel for most of the cases. The results also show that in the 128-CG runs, the asynchronous scheduler downgrades the performance in three cases. The cause of this anomaly is under investigation.

4.6 Sunway TaihuLight vs Mira vs Stampede2

Table 4.7 shows time per timestep in seconds for Burgers' equation on Sunway TaihuLight, DOE Mira, and TACC Stampede2 supercomputers using raw timings. Strong scaling experiments were conducted on 8, 16, and 32 nodes of Mira and Stampede2 with problem size of 2048 x 2048 x 1024. However, this problem did not fit on eight nodes of Sunway, and the nodes ran out of memory. Hence, the problem size was reduced by four times to

Table 4.5: Performance improvements of the asynchronous nonvectorized kernel.

Num CGs	1	2	4	8	16	32	64	128
16x16x512	8.3%	9.3%	2.5%	2.9%	6.4%	6.4%	5.4%	1.9%
16x32x512	8.5%	9.5%	1.8%	3.2%	5.7%	6.9%	8.6%	4.8%
32x32x512	39.3%	38.0%	34.9%	29.7%	37.3%	34.5%	33.7%	6.8%
32x64x512	1.4%	24.7%	21.4%	18.4%	26.7%	22.5%	18.3%	5.6%
64x64x512	-	24.4%	22.4%	19.5%	31.1%	28.6%	19.0%	-7.9%
64x128x512	-	-	27.5%	19.2%	29.1%	20.0%	21.3%	-1.5%
128x128x512	-	-	-	19.5%	27.3%	24.5%	19.8%	1.7%

Table 4.6: Performance improvement of the asynchronous vectorized kernel.

Num CGs	1	2	4	8	16	32	64	128
16x16x512	4.5%	5.1%	0.1%	13.0%	9.3%	4.2%	2.4%	-1.7%
16x32x512	4.4%	5.5%	0.0%	8.5%	11.0%	4.2%	4.4%	9.9%
32x32x512	19.5%	18.2%	15.3%	13.0%	16.7%	22.8%	15.2%	9.4%
32x64x512	12.5%	14.6%	10.3%	13.9%	12.1%	16.3%	7.9%	6.9%
64x64x512	-	15.4%	12.6%	13.1%	12.8%	14.4%	14.8%	4.8%
64x128x512	-	-	13.0%	12.9%	15.5%	19.8%	4.5%	2.1%
128x128x512	-	-	-	14.0%	15.7%	14.4%	2.5%	6.6%

Table 4.7: Problem settings in the evaluations.

Problem Size:	2048 x 2048 x 1024	2048 x 2048 x 1024	1024 x 1024 x 1024
Number of Nodes	DOE Mira	TACC Stampede2	Sunway TaihuLight
8	5.8	1.8	1.5
16	2.9	1.4	1
32	1.5	0.7	0.4

1024^3. Nevertheless, the data points give a rare opportunity for cross-system comparison of Sunway. To account for a difference in the problem sizes, timings on Mira and Stampede2 are divided by four, and then these weighted timings are compared with the Sunway results. Fig. 4.5 shows this weighted comparison. Mira, with IBM Blue Gene/Q having node level peak performance of 200 GFlops/s [144], performed marginally better than Sunway, whose peak node performance is 3 TFlops/s. Similarly, Stampede2 having Intel KNL with a peak performance of 3 TFlops/s (the same as that of Sunway), performed 2x to 3x faster. The performance gap is because of a few possible shortcomings on Sunway. The Sunway processor does not have instructions to compute exponential values, and exponentials are emulated in the software. Although all the tasks were offloaded asynchronously to CPEs, copying of data from main memory into LDM was done synchronously within every task and did not allow overlapping of data transfer with computation. Asynchronous copying

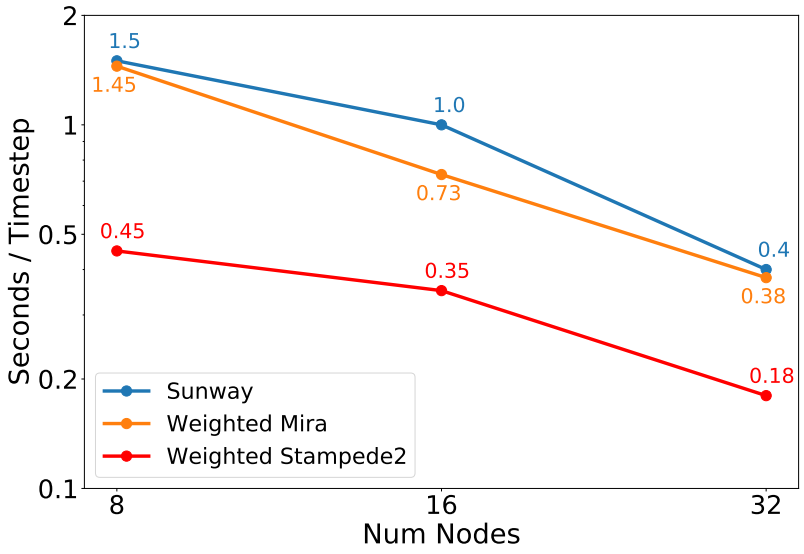


Fig. 4.5: Comparing Sunway with Mira and Stampede2.

of data into LDM can result into faster execution. Another possible factor could be the difference in Flop to memory bandwidth ratio. The ratio is 3.06 TFlops/s / 136.51 GB/s = 22.4 Flops/byte for Sunway, but 204.8 GFlops/s / 42.6 GB/s = 4.8 Flops/byte for IBM Blue Gene/Q, and 3.06 TFlops/s / 450 GB/s = 6.8 Flops/byte for Intel KNL. A larger Flops to byte ratio limits performance of the memory hungry stencil-like operations. More profiling is needed to gather the evidence for the last argument.

4.7 Related Work

This research has built heavily upon previous research with Uintah on other modern architectures such as Stampede. Some of the scaling experiments run on Stampede clearly showed the need for more sophisticated scheduling and task distribution mechanisms to utilize architectural features best and strike a balance between computation and communication to hide the communication latency [95]. A number of AMT runtime systems such as StarPU, Legion, and Charm++ exist and can scale up to hundreds of thousands of cores. However, as per the best knowledge of the author, Uintah is the first AMT system tailored to take advantage of the Sunway TaihuLight architecture.

4.8 Conclusion and Future Work

As shown by Uintah's port to Sunway, an AMT runtime system may be ported to the next-generation supercomputers - even those having diverse and unusual architectures - by 1) updating the build system; 2) using an asynchronous task scheduler based on MPI and native threading/offloading model; and 3) and with modifying task code tailored to the architecture.

The main advantage of AMT's design was the decoupling between tasks, and the infrastructure allowed the development of the asynchronous task scheduler, which made the infrastructure ready for the new architecture with minimal effort. However, huge efforts will be needed to port and tune *all* the tasks using architecture-specific programming models. The experience of porting Uintah to Sunway underlined the need to adopt a Performance Portability Layer (PPL) in the infrastructure and the user tasks. A PPL such as Kokkos [68, 182] would avoid rewriting 100s of tasks for every new upcoming architecture.

CHAPTER 5

PORTABLE SIMD PRIMITIVE FOR EFFICIENT VECTORIZATION ON HETEROGENEOUS ARCHITECTURES

This chapter presents the "portable SIMD primitive," which can be used to write portable code and achieve efficient vectorization on CPUs without affecting GPU performance. The new primitive has proven to be particularly helpful for complex kernels where autovectorization performs poorly. The research was earlier published in [165].

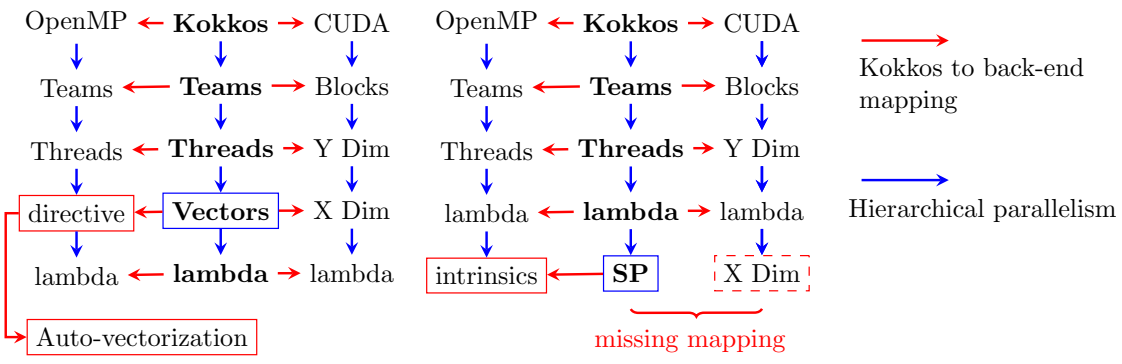
Different computer architectures are being developed to potentially improve floating point performance, such as those being developed for exascale. For example, Intel Haswell, Knights Landing (KNL), and Skylake processors support vector processing with a vector length of 512 bits. ARMv8.2-A processors have a vector length of 2048 bits [177]. Nvidia, Intel, and AMD GPUs may be part of upcoming supercomputers [2, 5]. Multiple performance portability frameworks are being developed to avoid architecture-specific tuning of programs for every new architecture. Such portability frameworks as Kokkos [68, 182] and RAJA [96] provide uniform APIs to shield a programmer from architectural details and provide a new performant back end for every new architecture to achieve the performance portability. The Kokkos [68, 182] library achieves performance portability across CPUs and GPUs through the use of C++ template meta-programming.

Many pre-exascale and proposed exascale CPU and manycore architectures increasingly rely on Vector Processing Units (VPUs) to provide faster performance. VPUs are designed with Single Instruction Multiple Data (SIMD) capabilities (vector capabilities) that execute a single instruction on multiple data elements of an array in parallel. SIMD constructs can enhance the performance by amortizing the costs of instruction fetch, decode, and memory reads/writes [69]. The process of converting a scalar code (which processes one element at a time) into a vector code (which can handle multiple elements of an array in parallel)

is known as the "vectorization" or "SIMD transformation." Thus, effective vectorization becomes very important for any performance portability tool, including Kokkos, to extract the best possible performance on CPUs.

Another important class of supercomputers uses GPUs as accelerators (e.g., Summit, Sierra). The Single Instruction Multiple Threads (SIMT) execution model of NVIDIA'S CUDA divides iterations of a data-parallel kernel among multiple CUDA blocks and threads. A warp, a group of 32 CUDA threads, runs in the SIMD mode *similar* to the VPU (an exception: the latest Volta GPUs allow out-of-sync execution of warp threads). Any portable solution to vectorization should allow both styles of vectorization without considerable effort from application programmers. Furthermore, it is essential to distinguish between the physical vector length (PVL) in the hardware and the logical vector length (LVL) as needed by the application usage. Fig. 5.1 (a) shows how Kokkos' uniform APIs, Team, Thread, and Vector [68, 182], provide three levels of parallelism, and how they are mapped to CPUs and Nvidia GPUs. At the third level, user-provided C++11 lambda is called, and loop indexes are passed to the lambda.

On Nvidia GPUS, the CUDA threads can be arranged in a three-dimensional grid. Each thread is identified by a triplet of ids in three dimensions that are accessed using "threadId.<x or y or z>." Consider the number of teams, threads, and vectors requested by a user are L, T, and V, respectively. In this case, GPUs, "L" Kokkos Teams are mapped to "L" CUDA blocks. The CUDA block id is mapped to the Kokkos team id. Each CUDA block is of the size VxTx1. The CUDA threads within a block can be logically divided among



(a) Existing Vectorization (b) If SIMD primitive (SP) used

Fig. 5.1: Kokkos APIs and mapping to CPU and CUDA.

T partitions of size V (not to be confused with CUDA-provided Cooperative Groups). Each partition is assigned a unique threadIdx.y ranging from 0 to T-1. Kokkos maps these partitions to Kokkos Threads and the Kokkos thread id to CUDA threadIdx.y. The threads within a partition are assigned a unique threadIdx.x ranging from 0 to V-1. Kokkos Vectors get mapped to V threads within each partition. On a CPU, the Kokkos Teams and Kokkos Threads are mapped to OpenMP thread teams and OpenMP threads. Using the Kokkos Vector augments the user code with the compiler directives, which helps the compiler in autovectorization. This Kokkos design enables efficient SIMT execution on GPUs. However, successful automatic vectorization on CPUs depends on a lack of loop dependencies, minimal execution path divergence in the code, and a countable number of iterations, i.e., the number of iterations should be known before the loop begins execution [15]. Traditionally, compilers autovectorize loops that meet these criteria but fail to autovectorize outer loops or codes having complex control flows, such as nested if conditions or break statements. This problem can be addressed by using SIMD primitive libraries that encapsulate architecture-specific intrinsic data types and operators to achieve explicit vectorization without compromising portability across CPUs. Several such libraries exist for CPUs [118, 122, 157, 188]. However, using SIMD primitive libraries would break the portability model, as shown in Fig. 5.1 (b). Instead of calling the Kokkos-provided **Vector**, programmers directly call the **lambda** from a **Thread**, and in turn invoke any SIMD primitive libraries, which would map user data types and functions to platform-specific intrinsics. This explicit vectorization can generate more efficient code where compilers do a poor job. However, to the best of our knowledge, no portable SIMD primitive library provides a GPU back end, except OpenCL, which supports vector data types on all devices [145]. Using such primitive libraries with Kokkos, however, leads to compilation errors due to missing CUDA back end for the primitive. As a result, programmers are forced to make a compromise - either achieve portability at the cost of nonoptimal CPU performance through the compiler autovectorization or achieve the optimal CPU performance using SIMD primitives, but maintain a separate version of code for GPU without using SIMD primitive, thereby compromising portability. Maintaining a different code for GPUs defeats the purpose of using Kokkos, i.e., "performance portability." The problem also defeats the notion of "portable tasks in AMT" as developed by Peterson [150].

To remedy the situation, this work makes the following contributions:

- **Heterogeneous Performance Portability**: The primary contribution of this work is to add a new CUDA back end to the existing SIMD primitive in Kokkos and make the SIMD primitive portable across heterogeneous platforms with Nvidia GPUs, for the first time. (More back ends can be added to Kokkos and to the primitive to support a wider range of heterogeneous platforms.) The CUDA back end is developed with the exact front-end interfaces as those built for the CPU back end. Using these uniform interfaces, the application programmers can now achieve efficient vectorization on the CPU without maintaining a separate GPU version of the code, which was not possible before. Thus, the new SIMD primitive provides GPU portability and requires only a few hundred lines of new code for the GPU back end.

  Using the new portable SIMD primitive gives a speed-up up to 7.8x on Intel KNL and 2.2x on Cavium ThunderX2 (ARMv8.1) for kernels that are hard to autovectorize. A comparison of the primitive with existing SIMD code (either autovectorized CPU code or equivalent CUDA code) shows no overhead due to the primitive. The portable primitive provides explicit vectorization capabilities, without the need to maintain a separate GPU code. As the outer loop may now be easily vectorized using the new primitive, more efficient code can be generated than autovectorization of the inner loop.

- **Logical Vector Length (LVL)**: Another feature of the new primitive is the Logical Vector Length (LVL). Application developers can pass the desired vector length as a template parameter (LVL) without considering the underlying physical vector length. The LVL can be used to write codes agnostic of physical vector length (PVL), as explained in Section 5.1. Vectorizing the outer loop coupled with the LVL automatically introduces the "unroll and jam [48]" transformations, without any burden on programmers. These transformations can exploit instruction-level parallelism and data locality to provide speed-ups up to 3x on KNL and 1.6x on CUDA than the autovectorized / SIMT code.

- **Easy Adoption**: Introducing the portable SIMD type needs less than a 10% change in the user code. Once the primitive is introduced, the code can be explicitly vectorized

on CPUS and also ported to GPUs without any further changes. Furthermore, it is easily possible extend portable interfaces by Peterson within Uintah to minimize the user code changes to even less than 5%.

- **Applicability to use cases**: The new portable SIMD data type supports a wide variety of computational science use cases, such as PDE assembly for complex applications, two-dimensional convolution, batched linear algebra, and ensemble sparse matrix-vector multiplication, as will be shown below.

5.1 Portable SIMD Primitive

5.1.1 Design

The portable SIMD primitive developed here sits on top of Kokkos, which provides basic performance portability across a range of architectures. Fig. 5.2, 5.3, and 5.4 present pseudo code of the portable SIMD primitive. The code can be divided into three parts:

- **Common declarations**: Fig. 5.2 shows some common declarations used to achieve portability. The PVL macro definition derives platform-specific vector length, i.e., physical vector length (PVL). "simd_cpu" and "simd_gpu" are forward declarations

```
// PVL: physical vector length
// LVL: logical vector length
// EL: element per vector lane

#define PVL ... //detect architecture-specific PVL

using namespace std;

// advanced declarations
template<typename T, int LVL, int EL=LVL/PVL>
struct simd_cpu; //for cpu

template<typename T, int LVL, int EL=LVL/PVL>
struct simd_gpu; //for gpu

template<typename T, int LVL, int EL=LVL/PVL>
struct gpu_temp;

// conditional aliases for Primitive and Temp
template<typename exe_space, typename T, int LVL>
using simd = typename conditional<
  is_same<exe_space, OpenMP>::value,
  simd_cpu<T, LVL>, simd_gpu<T, LVL> >::type;

template <typename exe_space, typename T, int LVL>
using Portable_Temp = typename std::conditional<
  is_same<exe_space, OpenMP>::value,
  simd_cpu<T, LVL>, gpu_temp<T, LVL>>::type;
```

Fig. 5.2: Common declarations used in SIMD primitive.

```
template<int LVL, int EL>
struct simd_cpu<double, LVL, EL>{

  __m512d _d[EL];  // knl instrinsic for 8 doubles

  Portable_Temp<exe_space, double, LVL> operator+ (const simd &x){
    Portable_Temp<exe_space, double, LVL> y;
#pragma unroll(EL)
    for(int i=0; i<EL; i++)
       y._d[i] = _mm512_add_pd( _d[i], x._d[i]);
    return y;
  }
  //more operators and overloads ...
};
```

Fig. 5.3: SIMD primitive: KNL specialization for double.

```
template <typename T, int LVL, int EL>
struct gpu_temp{
    T a[EL];
    //more operators and overloads ...
};

template<typename T, int LVL, int EL>
struct simd{

  T _d[LVL];

  Portable_Temp<exe_space, T, LVL> operator+ (const simd &x){
    Portable_Temp<exe_space, T, LVL> y;
#pragma unroll(EL)
    for(int i=0; i<EL; i++){
        int tid = i * blockDim.x + threadIdx.x;
        y._d[i] = _d[tid] + x._d[tid]
    }
    return y;
  }
  //more operators and overloads ...
};
```

Fig. 5.4: SIMD primitive: CUDA definition.

for CPU and CUDA primitives, respectively. They need a data type and the logical vector length (LVL) as the template parameters. An alias "simd" is created using std::conditional, which assigns "simd_cpu" to "simd" if the targeted architecture (or the execution space in the Kokkos nomenclature) is OpenMP and "simd_gpu" if the execution space is CUDA. The simd template expands into the respective definitions at compile time depending upon the execution space. As a result, both execution spaces can be used simultaneously, thus giving portable and heterogeneous execution. The "Portable_Temp" alias and "gpu_temp" type are used as a return type and are explained later.

- **CPU back end**: The CPU back ends containing architecture-specific SIMD intrinsics are developed for Intel's KNL and Cavium ThunderX2. Template specializations are used to create different definitions specific to a data type, as shown in Fig. 5.3 (which is a specialization for double on KNL). Overloaded operators invoke architecture-specific intrinsics to facilitate standard arithmetic operations, math library functions, if_else condition. The new primitive can support bitwise permutation operations such as shuffle and has been verified with some preliminary experiments. One such example of an overloaded operator is shown in Fig. 5.3. The operator+ calls the intrinsic function "_mm512_add_pd," which performs the addition of eight doubles stored in the intrinsic data type __mm512 in a simd manner. The return data type of the operator+ is "Portable_Temp." When the execution space is OpenMP, Portable_Temp is set to "simd_cpu" itself, which simply returns an intrinsic data type wrapped in the primitive. The KNL specific back end from Kokkos::Batched::Vector is reused, and the functionalities of the LVL and alias definition based on the execution space are added on top of it. A new back end was added for ThunderX2 using ARMv8.1 intrinsics.
- **Logical Vector Length**: Users can pass the desired vector length as the template parameter "LVL." The LVL iterations are evenly distributed among the physical vector lanes by the primitive. As shown in operator+ (Fig. 5.3), each vector lane iterates over EL iterations, where "EL=LVL/PVL," e.g., if PVL=8 and LVL=16, then EL=2, i.e., each vector lane will process two elements. Thus, LVL allows users to write vector length agnostic code. In use cases, such as the two-dimensional convolution kernel presented in this work later, using LVL improved performance up to 3x.

- **CUDA back end and Portable_Temp**: Finally, a new CUDA back end is added with the same front-end APIs as used in the CPU back end, making the primitive portable, as shown in Fig. 5.4. The common front-end APIs present a unified user interface across heterogeneous platforms, which allows users to maintain a single portable version of the code and yet achieve effective vectorization. The portability of the primitive avoids the development of two different versions as required prior to this work. The common front-end APIs include structures "simd" and "Portable_Temp," declared in Fig. 5.2, along with their member functions. Whenever a programmer switches to the CUDA execution space, "simd" alias refers to "simd_gpu" and expands into a CUDA definition of the SIMD primitive. To emulate the CPU execution model of SIMD processing, the GPU back end contains an array of the "logical vector length" number of elements (double _d [LVL]). These elements are divided among the PVL number of CUDA threads along the x dimension. (The PVL is autodetected based on a platform.) CUDA assigns unique threadIdx.x to each thread ranging from 0 to PVL-1. Each CUDA thread within operator+ (Fig. 5.4) adds different elements the array _d indexed by "tid = i * blockDim.x + threadIdx.x." (In this case, blockDim.x represents the number CUDA threads along x dimension, which is set to PVL.) Together, the PVL number of CUDA threads processes a chunk of PVL number of elements in a SIMT manner. Each CUDA thread executes EL number of iterations (loop variable i). Thus, the primitive processes LVL=PVL*EL number of elements within array _d. Offsetting by threadIdx.x allows coalesced access and improves the memory bandwidth utilization.

  However, the CUDA back end needed an additional development of gpu_temp to be used as a return type. Consider a temporary variable of a type "SIMD" used in the CPU code. The declaration is executed by the scalar CPU thread, and the elements of the variable are automatically divided among CPU vector lanes by the intrinsic function. Thus, each vector lane is assigned with only "EL=LVL/PVL" number of elements. However, when used inside a CUDA kernel, each CUDA thread, i.e., each vector lane, declares its own instance of the SIMD variable. Each instance contains LVL elements and results in allocating PVLxLVL elements. The problem can be fixed by setting the alias Portable_Temp to the type "gpu_temp." "gpu_temp" holds only EL

elements - exactly those needed by the vector lane. Thus, the total number of elements is still LVL. As a result, the CUDA implementation of the SIMD primitive needs combinations of operands: (SIMD, SIMD), (SIMD, Portable_Temp), (Portable_Temp, SIMD) and (Portable_Temp, Portable_Temp).

Two alternatives to avoid Portable_Temp were considered. The PVL can be set to 1 (or EL). One can even use CUDA-supported vector types such as float2 and float4. Both options will solve the return type problem mentioned earlier as each vector lane processes 1 / 2 / 4 elements and returns the same number of elements as opposed to elements getting shared by vector lanes. Using CUDA vector types can slightly improve the bandwidth utilization due to vectorized load and store. CUDA, however, lacks any vectorized instructions for floating point operations, and the computations on these vector types get serialized. Thus, using either of these options will remove the third level of parallelism (i.e., Kokkos Vector). Hence, the Portable_Temp construct was chosen.

5.1.2 Example Usage

Fig. 5.5 shows an example of vectorization using the portable SIMD primitive and Kokkos, but without showing Kokkos-specific details. Kokkos View is a portable data structure used to allocate two arrays, A and B. Elements of A are added into each element of B until B reaches 1. The scalar code (add_scalar function) does not get autovectorized due to a dependency between if(B[i]<1.0) condition and addition. (Of course, adding #pragma simd or interchanging loops helps in this example, but may not always work.) The add_vector function, a vectorized version of add_scalar, shows how the SIMD primitive can vectorize the outer loop. Array A is cast from double to simd<double>, the number of iterations of the outer loop is factored by the LVL, and the "if" condition is replaced by an if_else operator. The statement calls four overloaded operators, namely, <, +, if_else and =. Vectorizing across the outer loop works because the outer loop iterations are not dependent on each other. If the LVL is increased to 2*PVL, the loop gets unrolled by a factor of two, and each vector lane processes on two iterations consecutively. As the main computations usually take place in the innermost loop, the unrolled outer loop automatically gets jammed with the inner loop. Users can simply set LVL=*n*xPVL, and the primitive unrolls the outer

```
using namespace Kokkos;
typedef View<double*> dView;
void add_scalar(dView &A, dView&B, int n){
 parallel_for(..., [&](team_member t){//team loop
  parallel_for(..., [&](int tid){//thread loop
  //calculate "start" and "end" for the thread
  for(int i=start; i<end; i++)
   for(int j=0; j<n; j++)
    if(B[i] < 1.0)
     B[i] += A[j];
  });
 });
}

typedef simd<exe_space, double, SIMD_LVL> Double;
typedef Kokkos::View<Double*, KernelSpace> SimdView;

void add_vector(dView &A, dView&B_s, int n){
 SimdView B(reinterpret_cast<Double *>(B_s.data()));
 parallel_for(..., [&](team_member t){//team loop
  parallel_for(..., [&](int tid){//thread loop
  //calculate "start" and "end" for the thread
  for(int i=start; i<end/SIMD_LVL; i++)
   for(int j=0; j<n; j++)
    B[i] = if_else( (B[i]<1.0), (B[i]+A[j]), B[i]);
  });
 });
}
```

Fig. 5.5: Example usage of the SIMD primitive: Conditional addition of arrays without (top) and with SIMD primitive.

loop by a factor of n. Because the iterations of the outer loop are independent of each other, the transformation can exploit instruction-level parallelism.

5.2 Experiments

5.2.1 Experimental Platforms

A node of Intel KNL with 64 cores, 16GB of High Bandwidth Memory (or MCDRAM) configured in flat quadrant mode, and 192GB RAM was used to test the CPU version. Each KNL core consists of two VPUs with a vector length of 512 bits. Thus, using the double-precision floating point numbers allows a vector length of 8. The codes were compiled with the Intel compiler suite 2018 with the optimization flags -O3 -xMIC-AVX512 -std=c++11 -fopenmp.

Tests were also run on a single node of the Astra cluster at Sandia National Laboratories. Each Astra node provides 128 GB of high bandwidth memory and two Cavium ThunderX2 CN99xx processors with 28 cores each. Cavium ThunderX2 is an ARMv8.1-based processor with a vector length of 128 bits. It can execute two double-precision operations with a single SIMD instruction. The GNU 7.2.0 compiler suite was used to compile applications with the

flags -O3 -std=c++11 -mtune=thunderx2t99 -mcpu=thunderx2t99 -fopenmp.

The NVIDIA P100 GPU with Compute Capability 6.0, 3584 CUDA cores, 16GB of High Bandwidth Memory and 48 KB of shared memory per SM was used to test the GPU performance. The applications were compiled using gcc v4.9.2 and nvcc (from CUDA v 9.1) with the optimization flags -O3 -std=c++11 –expt-extended-lambda –expt-relaxed-constexpr.

5.2.2 Use Cases and Experimental Setup

The primary aim of the portability libraries such as Kokkos is to enable "performance portable" programming. The portable code gets compiled and executed on the heterogeneous platforms without making any platform-specific changes and also performs as well as the native implementations (such as using raw CUDA or using vector intrinsics). However, Kokkos or the SIMD primitive is not a magic construct to provide an extra performance boost. When the baseline itself is efficiently vectorized or has an efficient CUDA implementation, using Kokkos or the primitive can provide portability, but will not provide extra speed-up. Considering these factors, different use cases are chosen to test the performance of the SIMD primitive for different scenarios. Table 5.1 summarizes the

Table 5.1: Summary of use cases, goals, and expectations.

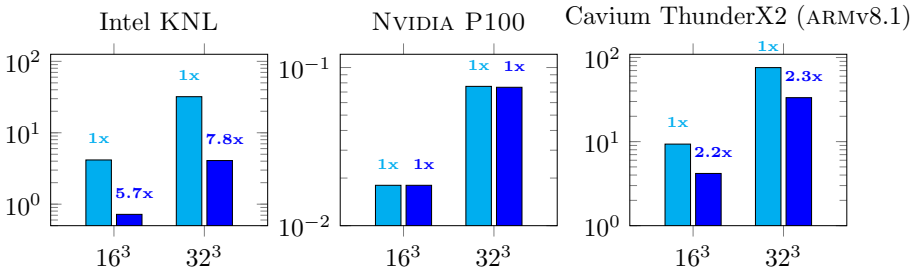
Use case	**Goal**	**Baseline**	**Expected Performance**	
			CPU	**GPU**
PDE	CPU: Achieve effective vectorization for the complex, hard to vectorize code; GPU: Find out the overhead for a performance sensitive portable kernel.	CPU: not vectorized; GPU: Ported to CUDA.	Near ideal speed-up.	No extra speed-up. No new overhead.
2dConv	Evaluate the benefit of LVL by comparing it with a baseline already running in SIMD mode.	CPU: autovectorized; GPU: Ported to CUDA.	Small extra speed-up due to LVL.	Small extra speed-up due to LVL.
GEMM	Find out the overhead of the primitive by comparing it with a baseline already running in SIMD mode efficiently.	CPU: auto vectorized; GPU: Ported to CUDA.	No extra speed-up. No new overhead.	No extra speed-up. No new overhead.
SpMV				

four kernels used in the evaluation. Of the four use cases, the first two are not efficiently vectorized and are chosen to demonstrate the effectiveness of the primitive, whereas the last two are efficiently vectorized and are chosen to measure the overhead of the primitive. Baselines are written using Kokkos for two reasons: to make the code portable, and to measure the overhead of the primitive only. If the baseline is written using raw CUDA or OpenMP, then the performance measurements will include the overhead of both Kokkos and the primitive.

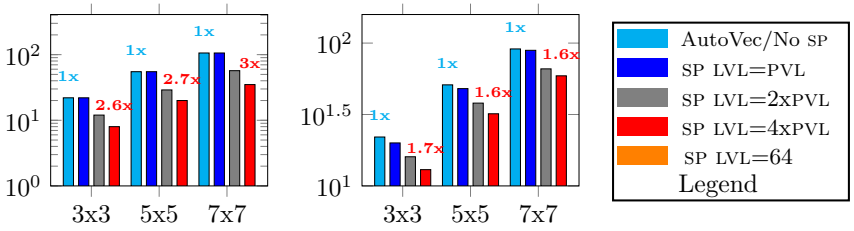
Use cases were implemented using Kokkos – first without using the SIMD primitive and then using it. A typical transformation from scalar code to vectorized code using the portable SIMD primitive needs casting of legacy data structures and variables and updating of any conditional assignments. Some algorithm-specific use cases need special handling, e.g., the while loop discussed in Section 5.3.1. All the arithmetic operations and math library functions remain untouched. For all four kernels, less than 10% of the lines of code were modified to introduce the SIMD primitive, which did not require any complex code transformations or new data structures, which is typically needed to autovectorize a complex code. The use of Kokkos and the SIMD primitive allows the same code to be compiled on the different target platforms. Various combinations of the number of threads were tested, and the best timing was chosen. Each experiment was repeated at least 100 times, and the averages timings were used to compute the speed-ups.

5.2.3 Methodology for Performance Evaluation

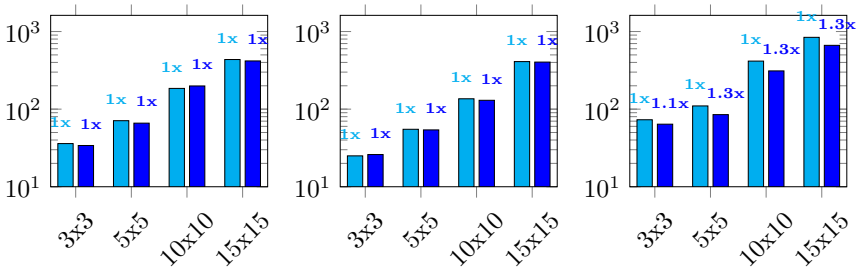
The rows in Fig. 5.6 show the results of the four use cases, and the columns indicate three platforms. Each plot shows execution time along with the speed-up compared to the baseline. The baseline is either the autovectorized code (AV) or the code with no SIMD primitive (NSP) colored in cyan. Results of using the SIMD primitive with different values of the LVL are represented by "SP." As mentioned earlier, the experiments have three goals: 1) Find out performance improvement when the code is not efficiently vectorized (PDE and 2dConv cases); 2) ensure that performance improvement on one platform does not hamper the performance on another platform (PDE); and 3) measure the overhead of the new primitive against the efficiently vectorized baseline, where the expected speed-up is 1x (GEMM and SpMV).



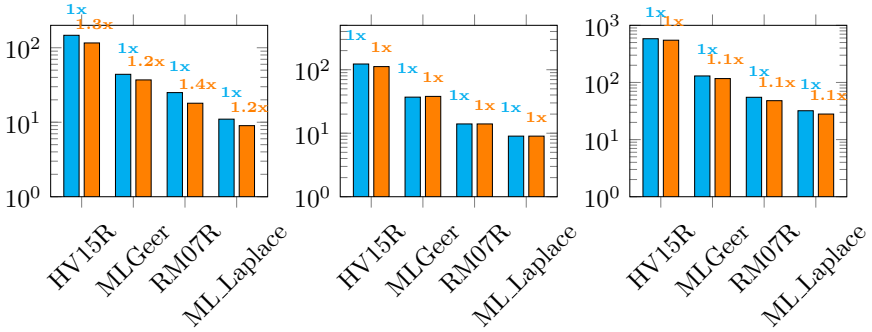
(a) Uintah CharOx: Execution time in **seconds** vs patch size.



(b) 2D Convolution: Execution time in **milliseconds** vs filter size.



(c) Batched GEMM: Execution time in **microseconds** vs matrix size.



(d) Ensemble SpMV: Execution time in **milliseconds** vs data sets

Fig. 5.6: Comparison of execution times along with speed-ups for different kernels on different architectures. Speed-up "1x" indicates zero new overhead due to the new primitive.

The vectorized code (AV and SP both) executes fewer instructions than the scalar code, but the vector instructions execute more slowly than the scalar counterparts, consuming more cycles. Hence, the instructions per cycle (IPC) count does not reflect the exact speed-up. Similarly, KNL hardware counters do not accurately measure floating point operations (Flops), and numbers often get skewed while measuring floating point instructions (FLIPs) [111]. Hence, simple counts such as the total number of instructions and cache hits are used here for performance analysis. The performance metrics and events are collected using Intel vtune amplifier, Nvidia nvprof, and the PAPI library. Section 5.3 analyzes the performance of each use case.

5.3 Performance Evaluation

This section presents implementation details for every use case, followed by the performance evaluation.

5.3.1 PDE Assembly

One of the longest running kernels within the Arches component of Uintah AMT is CharOx. It simulates the char oxidation of coal particles by modeling multiple chemical reactions and physical phenomena involved in the process [25, 91, 147]. The CharOx kernel consists over of 350 lines of code, reads around 30 arrays, updates five arrays, and performs compute-intensive double-precision floating point arithmetic operations, such as exponentials, trigonometric functions, and divisions in nested loops about 300 to 500 times for every cell. As shown in Algorithm 3, the main cell iterator loop contains different loops over reactions and species, each with multiple levels of nesting. The Newton Raphson Solve loop has an undetermined number of iterations and contains more nested loops. The iterations of the cell iterator loop are not dependent on each other. Hence, vectorizing the cell iterator loop can potentially give a maximum speed-up.

5.3.1.1 Autovectorization Attempts

On the KNL platform, the Intel compiler autovectorizes some of the innermost loops only. Vectorization of the cell iterator loop can be forced by adding the "#pragma simd" directive. However, autovectorization provides only 4.3x speed-up, whereas the ideal speed-up for the double-precision on KNL is 8x, assuming most of the code is scalar. (Unfortunately, the

```
1:  for all patches
2:   for all Gaussian quadrature nodes
3:   Kokkos::parallel_for cells in a patch //Can cells loop be vectorized?
4:       Compute reaction constants.
5:       Nested loops over reactions and species.
6:       Multiple loops over reactions and species.
7:       Nested loops over reactions and species.
8:       while residual < threshold do //indefinite number of iterations
9:           Multiple loops over reactions.
10:          Nested loops over reactions and species.
11:          Compute a matrix inverse.
12:          Multiple loop over reactions.
13:      end while
14:      Loop over reactions.
15:  end Kokkos::parallel_for
```

Algorithm 3: CharOx loop structure.

pragma is deprecated in Intel compilers from 2018 onwards, and its replacement "#pragma vector" fails to "force" vectorize the cell loop.) An inspection of the vectorization report and assembly code shows gather/scatter instructions generated for every read/write to the global arrays. These gather and scatter instructions are the reason for the speed-up of 4.3x. To maintain the halo region, Uintah internally offsets all elements in its data structures with a constant value. All cells have the same offset. Thus, the stride between elements is always one, but the compiler cannot deduce this and generates gather instructions. However, using the SIMD primitive calls SIMD intrinsics that explicitly generate move instructions rather than gather and makes vectorization efficient.

The GNU compilers used on the ThunderX2 platform did not vectorize the cell iterator loop even after adding vectorization hint directives.

On the GPU, the size and the complexity of the kernel substantially increase register usage. Profiling shows that 255 registers are used by every thread within a block, thereby preventing the simultaneous execution of multiple blocks on a single Streaming Multiprocessor (SM), which results in poor occupancy of the SMs (only up to 12%). Hence, the SIMD primitive must not add additional overhead in terms of registers, memory, or execution dependency, and the GPU performance must not be compromised to gain CPU performance. Apart from casting data structures and variables to those based on the portable SIMD primitive, the Newton-Raphson solver used to solve oxidation equations

for every cell needed special handling. The solver iterates until the equations converge. In the vectorized version, the vector of cells iterates until all cells within the vector converge. Although the technique needs extra iterations for a few cells, it works faster than executing solver iterations sequentially in a scalar mode.

The experiments were carried out using 64 patches with two patch sizes - 16^3 and 32^3. The CharOx kernel is invoked five times for every patch. With 64 patches, the kernel is executed 320 times in every timestep. The simulation was run for 10 timesteps, and the average loop execution time of over 3200 calls was recorded. The sheer complexity of this loop appears to provide a distinctive and unusual challenge for performance portability.

5.3.1.2 Goals and Expectations

This use case shows a particular instance where the compiler does a poor job in auto-vetorizing the code on the CPU, but the CUDA code works efficiently on the GPU. It is thus important to ensure that improving CPU performance using the primitive does not degrade GPU performance. The kernel is large and complex enough to cause a register spill on the GPU even without using the SIMD primitive. Thus, adding SIMD will help us to understand the performance of sensitive kernels on GPU and associated overhead, if any.

The code performs double-precision floating point operations. Hence, speed-ups close to 8x and 2x are expected on KNL and ThunderX2, respectively. These are the ideal speed-ups for double-precision computations on these platforms, considering the respective vector lengths of 512 bits and 128 bits. The GPU code already runs in a SIMT mode, and hence the new primitive will provide portability without a performance boost. However, portability should not cause any significant overhead either. Ideally, GPU performance should remain the same with and without SIMD.

5.3.1.3 Performance Analysis

The KNL plot in Fig. 5.6 (a) shows the SP version that achieves 5.7x and 7.8x speed-ups over AV for mesh patch sizes of 16^3 and 32^3, respectively. Analysis of the 16^3 patch problem shows the number of instructions (INST_RETIRED.ANY) executed reduced from 1273 million for the AV code to 204 million for the SP code (Table 5.2). Similarly, L1 cache data misses (PAPI_L1_DCM) decreased from 2.4 million for AV to 1.2 million for the SP. In this case, some of the cache lines are evicted over the course of one iteration due to the

Table 5.2: Performance metrics for CharOx (counts in millions).

	Intel KNL		Cavium ThunderX2	
	number of instructions	L1 cache data misses	number of instructions	L1 cache data misses
No SIMD primitive	1273	2.4	4253	5.7
SIMD primitive	204	1.2	1823	1.5

complex operations and the 30+ different arrays used in the kernel. When the next iteration starts, at least some of the memory is missing from the L1 cache due to earlier evictions. However, the vectorized code can take advantage of entire cache lines, and all eight double elements from the 64 bytes cache lines can be read by eight vector lanes, thus fully utilizing data fetched in a cache line. The increased cache line efficiency along with vectorization provides near-optimal speed-up.

The P100 results show the NSP and SP both performing equally well. As the NSP running on the GPU runs in SIMT mode, the SP does not provide any extra level of parallelism and cannot provide an extra boost. These results with 1x speed-up are important in showing that the SP does not create any overhead on a GPU, even when the NSP kernel causes register spilling. All the metrics collected by nvprof showed similar values in this case. Increasing the value of LVL to 2xPVL slowed down the performance by 1.5x, because the increased LVL increased register spilling (evident from increased local memory accesses).

The SP version of CharOx kernel boosted performance by 2.2x and 2.3x for patch sizes of 16^3 and 32^3, respectively, on ThunderX2. The ThunderX2 metrics show a trend similar to that observed on KNL. The total number of instructions executed is reduced from 4253.8 million for the NSP to 1823.1 million for the SP. Again, vectorization reduced the number of cache misses from 5.7 million to 1.5 million, which provided super-linear speed-ups up to 2.3x, where the PVL supported by the hardware for double-precision is only 2.

5.3.2 Two-Dimensional Convolution

two-dimensional convolution [52] (as shown in Algorithm 4) is a heavily used operation in deep neural networks. The algorithm multiplies a batch of images (*in*) with a filter

```
1: for b in 0:mini-batches
2:  for co in 0:output filters
3:   for i in 0:M //image rows
4:    for j in 0:M //image columns
5:     for ci in 0:input channels
6:      for fi in 0:F //filter rows
7:       for fj in 0:F //filter columns
8:        out(b, co, i, j) += in(b, ci, i-F/2+fi, j-F/2+fj) * filter(co, ci, fi, fj)
```

Algorithm 4: Algorithm for a two-dimensional convolution kernel.

(*filter*) by sliding the filter over the image to accumulate the result (*out*). The operation is repeated for multiple filters. This algorithm has a high arithmetic intensity. Using the SIMD primitive provides an opportunity to exploit the spatial locality for all three variables: When the "i" loop is parallelized across the Kokkos threads and the "j" loop across the SIMD lanes, every "filter" element is reused for the "LVL" number of "j" iterations. Also, two levels of parallelism help reusing elements in different rows "in" and "out" (similar to a stencil block). Of course, these improvements can be obtained manually without using the primitive. However, using the primitive introduces these transformations implicitly and improves the programmability, portability, and maintenance of the code.

The mini-batch loop in the original code is parallelized across OpenMP threads/CUDA blocks. The code is then autovectorized across the j loop using the directive #pragma simd on CPU and mapping the x dimension CUDA threads across the j loop on a GPU. #pragma unroll was used to unroll the full lengths of the fi and fj loops. The code was then converted into a SIMD primitive code instead of using #pragma simd. Different combinations of mini-batch sizes (3584 and 7168), filter sizes (3x3, 5x5, and 7x7), number of input (3, 5, and 10), and output channels (3, 5, and 10) were tested for different values of the LVL.

5.3.2.1 Goals and Expectations

The aim is to evaluate the effectiveness of the LVL against the vectorized baseline.

As the baseline is efficiently autovectorized, using the SIMD primitive with LVL=PVL should not perform any better. However, setting LVL=2*PVL or 4*PVL should give speed-ups on both CPU and GPU due to instruction-level parallelism and data reuse.

5.3.2.2 Performance Analysis

Fig. 5.6 (b) shows speed-ups up to 3x on KNL and 1.6x on P100 for the two-dimensional convolution kernel shown in Algorithm 4. The input image size, the number of input, and output channels were set to 64x64, 3, and 10, respectively. The image was padded by filter size / 2 number of cells. The baseline NSP and the SP with LVL=PVL perform equally well as both get efficiently vectorized. Setting LVL=2xPVL and 4xPVL gives better results on both KNL and P100.

Line number 8 of Algorithm 4 multiplies "in" with "filter" and accumulates the result in "out." Vectorizing the j loop coalesces accesses for "in" and "out." "filter" is independent of the j dimension, and hence the value is reused across all vector lanes. When the LVL is set to 2xPVL (or 4xPVL), the same filter value gets reused across twice (or four times) the PVL elements. An assembly instruction inspection shows the register containing "filter" was reused across multiple fma operations. All these fma operations are independent of each other and can exploit instruction-level parallelism. This "unroll and jam" transformation can be introduced by simply increasing the value of LVL. Using LVL in this case can save developers having to manually perform "unroll and jam" - especially for larger codes - and maintain the readability of the code.

The reuse of the "filter" values across the j iterations reduced the number of memory loads from 3 billion for the NSP to 2 billion for the SP with LVL=4xPVL on KNL and from 1.3 billion to 0.8 billion on the P100. The number of instructions executed was reduced by 2.3x and 1.4x on KNL and P100 platforms, respectively. The L2 cache hit rate improved on the P100 from 25% to 83%. Additionally, the number of control flow instructions executed was reduced by a factor of 3.5 on the P100 due to outer loop unrolling (see Table 5.3). The

Table 5.3: Performance metrics for two-dimensionalCov (counts in billion).

	Intel KNL		Nvidia P100		
	number of instructions	number of memory loads	number of instructions	number of memory loads	L2 cache hit rate
No SIMD primitive	6.2	3	9.8	1.3	25%
SIMD primitive	2.6	2	6.8	0.8	83%

effectiveness of the primitive and LVL can be judged from the fact that the naive code in Algorithm 4 with the SIMD primitive and LVL=4xPVL was only 20% slower than the highly tuned cuDNN library by Nvidia, as shown in Table 5.4. A small fix to use GPU's constant memory to store the filter gave an additional boost, and the naive code performed as well as the cuDNN library. Thus, the primitive can help application programmers who may focus on the algorithms and applications rather than spending time on specialized performance improvement techniques such as tiling, loop unrolling, using shared memory, etc.

Unfortunately, experiments for two-dimensional convolution could not be conducted on ThunderX2 because the Astra cluster was moved to a restricted domain by Sandia National Laboratories.

5.3.3 Compact GEMM

The general matrix-matrix multiplication (GEMM) on a batch of small and dense matrices is widely used within scientific computing and deep learning. Thread-parallel GEMM operations over collections of matrices organized in an interleaved fashion can be made efficient and portable using the SIMD primitive [120]. This approach is implemented within KokkosKernels, and used in a large-scale CFD code called SPARC [98]. The KokkosKernels' batched GEMM kernel achieves performance comparable or sometimes better than vendor-provided libraries such as Intel's math kernel library (mkl) and Nvidia's cuBLAS [120, 183]. KokkosKernels maintains two versions of batched GEMM - the CPU version, which uses an intrinsics-based SIMD primitive, and a CUDA version, which does not have a SIMD primitive. The only change needed in the kernel to utilize the portable SIMD primitive was to map the matrix dimension to the SIMD dimension by casting matrices from Kokkos views of doubles to Kokkos views of the SIMD primitive. Thus, each CPU thread (or a section of a CUDA warp) carried out each operation on the LVL number of

Table 5.4: Performance comparison with Nvidia cuDNN (execution time in milliseconds).

Filter size	cuDNN	simd primitive LVL=4XPVL	simd primitive LVL=4XPVL with constant memory
3x3	11	13	11
5x5	24	31	25
7x7	49	59	47

matrices in SIMD fashion. Both kernels had the same tiling optimizations with tile sizes of 3 x 3 and 5 x 5 to extract spatial and temporal locality among matrix elements. Experiments were carried out using four matrix sizes: 3 x 3, 5 x 5, 10 x 10, and 15 x 15 using a batch of 16,384 matrices on all three platforms.

5.3.3.1 Goals and Expectations

The goal is to compare the performances of the new SIMD primitive and the existing high performance explicitly vectorized code on CPU. Any performance degradation will reveal the associated overheads, if any. The SIMD primitive should perform as well as the code without the SIMD primitive on both CPU and GPU. Neither a performance boost nor any extra overhead is expected.

5.3.3.2 Performance Analysis

Fig. 5.6 (c) shows that the NSP and SP versions perform equally well on the KNL and P100 (speed-up is 1x) and that the SP does not create any overhead. These results are as expected because KokkosKernels (the NSP version) contains explicitly vectorized code for KNL and Kokkos code tuned explicitly for GPUs. These observations are confirmed by the same number of instructions executed by the NSP and SP versions - 23 million on KNL and 20 million on P100.

However, the ThunderX2 results show an improvement of up to 1.3x. The architecture-specific intrinsic back end for ThunderX2 had not yet been updated in the KokkosKernels, and it falls back to an emulated back end using arrays and "for" loops. Although this NSP version gets autovectorized by the compiler, the SP leads to more efficient vectorization. The NSP version executes 146 million instructions, whereas the SP version executes 114 million instructions on ThunderX2.

5.3.4 Embedded Ensemble Propagation

This kernel is heavily used in the uncertainty quantification of predictive simulations that involve the evaluation of simulation codes on multiple realizations of input parameters. The efficiently autovectorized baseline kernel multiplies a sparse matrix by an ensemble of vectors with matrix rows distributed across threads and vectors distributed across SIMD lanes. Vectors are arranged in an interleaved fashion similar to batched GEMM. This design,

introduced by Phipps [157], allows the reuse of matrix values across all vectors and provides up to 4x speed-ups over traditional batched sparse matrix-vector multiplication. Instead of repeating Phipps' experiments, the vectorized ensemble version itself is used as a baseline. Compared to the vendor-provided libraries, i.e., Intel's mkl and Nvidia's cusparse, the baseline kernels exhibit 1.8x to 3x speed-up on KNL and 1.3x to 1.6x on P100, respectively (see Table 5.5). These observations are in line with Phipps' experiments. This baseline kernel is converted to use our SIMD type by casting data structures from double to those using the SIMD primitive and setting the LVL equal to the ensemble length. Hence, any performance degradation from baseline can show the shortcomings in the LVL implementation.

5.3.4.1 Goals and Expectations

The goal is to find out any overhead associated with the SIMD primitive by evaluating its performance against the highly optimized baseline that implements the same design and parallelism pattern but without using the primitive. The code with the SIMD primitive should perform as well as the baseline on CPU and GPU. No performance boost and no overhead are expected.

5.3.4.2 Performance Analysis

The kernel is evaluated using 13 matrices from the University of Florida sparse matrix collection. However, results from only four matrices (listed in Table 5.5) that represent the general trend are presented for the sake of brevity. Sparse matrix-vector ensemble multiplication results on the GPU shown in Fig. 5.6 (d) indicate both versions, NSP and SP, perform equally on the P100 GPU. Both versions are efficiently ported to the SIMT model

Table 5.5: Sparse matrices used for ensemble SpMV and comparison of the baseline Kokkos version with Intel's mkl and Nvidia cusparse libraries for ensemble size = 64.

Name	Rows	Columns	Nonzeros	Execution time (ms)			
			Nonzeros	mkl KNL	baseline KNL	cusparse P100	baseline P100
HV15R	2017169	2017169	283073458	275	147	160	123
ML_Geer	1504002	1504002	110686677	105	44	54	37
RM07R	381689	381689	37464962	46	25	23	14
ML_Laplace	377002	377002	27582698	34	11	13	9

and use the same ensemble logic for data reuse. Therefore, the matching GPU performance for both versions meets the expectation and indicates that the primitive does not cause any overhead.

More surprising were speed-ups up to 1.3x on KNL, and 1.1x ThunderX2. Profiling showed about 10% to 20% reduction in the number of instructions executed for different sparse matrices and different ensemble sizes. Although the Flops were, of course, the same for both versions, an assembly code inspection revealed the reason behind the speed-ups. The result of matrix-vector ensemble multiplication is also a vector ensemble. The design by Phipps et al. [157] fetches a matrix element and multiplies all vectors with it to avoid repeated accesses to matrix elements, which are costly when the sparse matrix is stored in the "compressed row storage" format. Although the Phipps design performs faster than the traditional batched multiplication, it has to repeatedly fetch elements from the resultant vector ensemble to do the accumulation. In the NSP version, the compiler generates three vector instructions for every vector operation: 1) a fetch of the result ensemble from memory to a vector register; 2) a vectored fused-multiply-add (fma) on the result stored in a vector register with a vector from memory and a matrix element stored in vector register; and 3) a store of the result from the vector register in the memory. When the SP is used, the ensemble length is mapped to the LVL. This mapping helps the compiler deduce the array length and number of registers. Hence, for N=64, all result elements get loaded into eight vector registers only once, and fma operations are repeated on these registers. Hence, using the SP eliminates the need to transfer the result back and forth from memory and takes only one store to move the accumulated result from the vector registers to the memory. Thus, one load and one store are saved for every fma operation, resulting in a more efficient code.

5.4 Optimal LVL

The LVL value depends on register availability and levels of parallelism, both dictated by the algorithm and hardware. If the LVL is set to 2xPVL or 4xPVL, the compiler can usually allocate the structure into registers. Then the code can take advantage of instruction-level parallelism, if supported by the hardware, as observed in the cases of two-dimensionalCov and SpMV. When, however, the LVL was set to 8xPVL, the compiler allocated the structure into memory instead of registers and so hampered the performance of two-dimensionalCov

with extra loads and stores. The CharOx kernel is very complicated, and register spilling happens even in the NSP. Therefore, setting LVL=2xPVL, increased the register pressure further and resulted in slower execution in contrast to other use cases. The second factor in choosing the right LVL is the number of levels of parallelism an algorithm can offer. If both levels of parallelism, thread-level and SIMD-level, are applied to the same loop (as in GEMM or CharOx), then increasing the LVL effectively increases the workload per thread and decreases the degree of parallelism available, which can cause a load imbalance among cores. The GEMM kernels were hand-tuned to unroll and jam along matrix rows and columns. The optimization gave enough workload to fully exploit available instruction-level parallelism. As a result, increasing the LVL did not provide any further advantage.

5.5 Conclusion and Future Work

This study describes a portable SIMD data type whose primary benefit is to achieve vectorization in a portable manner on architectures with VPUs and GPUs. This capability has the potential to be useful for massive applications that use Kokkos to extract performance from future architectures (including exascale architectures), without explicitly tuning the user code for every new architecture. The greatest benefits of the SIMD primitive were observed in the most complex kernel, which was hard to autovectorize. Performance boosts of up to 7.8x on KNL and 2.2x on Cavium ThunderX2 can be observed for double-precision kernels (PDE). For the kernels that are vectorized/ported to GPUs, the new SIMD primitive results in speed-ups up to 3x on KNL, 1.6x on P100, and 1.1x on ThunderX2 due to more efficient vectorization (SpMV), cache reuse (2dConv), instruction-level parallelism (2dConv) and loop unrolling (2dConv and SpMV). The comparison with efficiently vectorized kernels showed minimal overhead for PDE and zero overhead for GEMM and SpMV kernels. The new primitive makes outer loop vectorization easier (as shown with CharOx, SpMV and 2dConv). The PDE example proved that performance on one platform can be improved without compromising the performance on another platform.

The Kokkos-based design will make it easier to port this SIMD primitive to future GPU exascale architectures such as Aurora and Frontier. The Kokkos profiling interface can possibly be extended to profile the primitive-based code in the future. Preliminary experiments showed that the new primitive can be easily extended to both OpenACC and

OpenMP 4.5. It will be interesting to compare the performance of OpenACC, OpenMP 4.5 / 5.0 (in the future), and Kokkos.

It is not clear whether the programming model on Intel GPUs will be SIMT (similar to CUDA) or SIMD (similar to KNL - Intel's traditional vectorization). It will become even critical for Kokkos and AMTs to have improved vectorization support in the latter case. The portable SIMD primitive equips Uintah AMT and Kokkos with an effective vectorization tool that can be used at short notice for some key hard-to-vectorize simulation kernels.

CHAPTER 6

MPI ENDPOINTS BASED THREADING MODEL TO MODERNIZE LEGACY THIRD-PARTY LIBRARIES

Many of the legacy codes and third libraries were designed when CPUs had just a single core. As a result, these libraries were developed from the perspective of a single MPI rank per core. However, the emergence of multicore and manycore processors demands the modernization of these legacy codes for maximum performance. The situation becomes even more difficult for AMTs when one of the tasks is a legacy third-party library. Even if the AMT supports multithreaded execution, the library task has to be run using a single thread, which may cause work imbalance. Alternatively, the AMT has to be run as a single-threaded process. As a result, AMT cannot benefit from the multithreaded execution. On the other hand, blindly parallelizing every data-parallel loop using OpenMP or CUDA can also backfire. This chapter describes a new threading model that can be efficiently used with legacy codes and benefit from task parallelism in AMTs and data-parallelism offered by the modern architectures, as initially published in [163, 164, 195].

One such example of third-party library is the Hypre [168] linear equations solver library. The combustion simulation using Uintah needs the solution of a pressure projection equation at every time substep for the low-Mach-number pressure formulation. The solution to the pressure equation is obtained using Hypre. Hypre supports different iterative and multigrid methods, has a long history of scaling well [31, 71], and has successfully weak scaled up to 500k cores when used with Uintah [126]. Past Uintah simulations were carried out [40] on DOE Mira and Titan systems [126], but the next-generation of simulations will be run on manycore processors (such as DOE's Theta and NSF's Frontera) and GPU architectures (such as DOE's Lassen, Summit, and Aurora). On both classes of machines, the challenge for library software is then to move away from an MPI-only approach in which

one MPI process runs per core to a more efficient approach in terms of storage and execution models. On manycore processors, a common approach is to use a combination of MPI and OpenMP to exploit the massive parallelism. In the case of GPUs, the OpenMP parallel region can be offloaded to a GPU with CUDA or OpenMP 4.5. Portability layers such as Kokkos [90] can also be used for offloading. The MPI-only configuration for Uintah spawns one single-threaded rank per core and assigns one patch per rank. In contrast, Uintah's Unified Task Scheduler was developed to leverage multithreading and also to support GPUs [101]. Portable multithreaded Kokkos-OpenMP and Kokkos-CUDA [90]-based schedulers and tasks make Uintah portable for future heterogeneous architectures. These new Uintah schedulers are based on *teams of threads*. Each rank is assigned multiple patches, which are distributed among thread teams. Teams of threads then process the patches in parallel (task parallelism) whereas threads within a team work on a single patch (data-parallelism). This hybrid design has proven useful on manycore systems and in conjunction with Kokkos has led to dramatic improvements in performance [90].

The challenge of realizing similar performance improvements with Uintah's use of Hypre and its Structured Grid Interface (Struct) is addressed in this work, so that Hypre performs as well (or better) in a threaded environment as in the MPI case. Hypre's structured multigrid solver, PFMG [31], is designed to be used with unions of logically rectangular subgrids and is a semicoarsening multigrid method for solving scalar diffusion equations on logically rectangular grids discretized with up to 9-point stencils in two-dimensional and up to 27-point stencils in three-dimensional. Baker et al. [31] report that various versions of PFMG are between 2.5–7$\times$ faster than the equivalent algebraic multigrid (AMG) options inside Hypre because they are able to take account of the grid structure. When Hypre is used with Uintah, the linear solver algorithm uses the Conjugate Gradient (CG) method with the PFMG preconditioner based upon a Jacobi relaxation method inside the structured multigrid approach [168].

The equation (6.1) that is solved in Uintah is derived from the numerical solution of the Navier-Stokes equations and is a Poisson equation for the pressure, p, whose solution requires the use of a solver such as Hypre for large sparse systems of equations. Although the form of (6.1) is straightforward, a large number of variables, for example, 6.4 billion in [168], represents a challenge that requires large-scale parallelism. One key challenge with

Hypre is that only one thread per MPI rank can call Hypre, which forces Uintah to join all the threads and teams before Hypre can be called, after which the main thread calls Hypre. Internally, Hypre uses all the OpenMP threads to process cells within a domain, but patches are processed serially. From the experiments reported here, it is this particular combination that introduces extra overhead and causes the observed performance degradation. Thus, the challenge is to achieve performance with the multithreaded and GPU versions of Hypre but without degrading the optimized performance of the rest of the code.

$$\nabla^2 p = \nabla \cdot \mathbf{F} + \frac{\partial^2 \rho}{\partial t^2} \equiv R. \tag{6.1}$$

In moving Hypre to manycore architectures, OpenMP was introduced by Hypre developers to support multithreading [75]. However, in contrast to the results in [75], a dramatic slowdown of 3–8$\times$ was observed when using Hypre with Uintah in an OpenMP multithreaded environment compared to the MPI-only version. Baker et al. make similar observations using a test problem with PFMG solver and up to 64 patches per rank. They observe a slowdown of 8–10$\times$ between the MPI-only and MPI+OpenMP versions [31]. The challenges of OpenMP in Hypre force Uintah either to remain the single-threaded (MPI-only) version of Hypre or to use OpenMP with one patch per rank, which defeats the purpose of using OpenMP.

This work shows that the root cause of the slowdown is the use of OpenMP pragmas at the innermost level of the loop structure. However, the straightforward solution of moving these OpenMP pragmas to a higher loop level does not offer the needed performance. The solution adopted here is to use an alternate threading model using scalable MPI endpoints [63, 194] to solve the slowdown problem and to achieve a speed-up consistent with the results observed by [31,75]. This approach requires overriding MPI calls to simulate MPI endpoints behavior. The current MPI standard assigns an MPI rank to a process, and all threads within a process share the same rank. The proposed endpoints approach allows assigning an MPI rank to an endpoint (EP), where an EP can be a thread, a team of threads, or a process [62]. Thus, each thread (or a team of threads) attached to an EP acts as an individual MPI rank. There can be multiple EPs with different MPI ranks within a process, and each endpoint can execute its computations and independently communicate with other EPs using MPI messages. Additional optimizations such as vectorization, funneled communication, and a lightweight threading model further improve the performance. In

this extension of prior work, three communication-centered optimizations are introduced: an efficient interthread communication scheme, communication-reducing patch assignment, and utilization of network parallelism through a state-of-the-art MPICH library capable of mapping logically parallel MPI communication to distinct network contexts. The new enhancements improve the scalability of Hypre and result in an overall speed-up of 1.7–2.4× over the MPI-only version.

In optimizing Hypre performance for GPUs, Hypre 2.15.0 was chosen as a baseline on Nvidia V100 GPUs to characterize the performance. Profiling on GPU reveals the launch overhead of GPU kernels to be the primary bottleneck and occurs because of launching thousands of microkernels. The problem is fixed by fusing these microkernels and using GPU's constant cache memory. Finally, Hypre is modified to leverage CUDA-aware MPI on the Lassen cluster, which gives an extra 10% improvement.

The main contributions of this work are to:

- Introduce the MPI EP model in Hypre (called Hypre-EP) to avoid the performance bottlenecks observed with OpenMP. Hypre-EP can enable better overall performance in the future when running the full simulation using a multithreaded task scheduler within Uintah AMT.
- Identify the bottlenecks in Hypre-EP and improve its performance and scalability beyond the MPI-only version by adding new communication-centered optimizations to get the overall speed-up of 1.7–2.4× over the MPI-only version.
- Optimize the CUDA version of Hypre to improve CPU to GPU speed-ups ranging from 2.3× to 4× in the baseline version to the range of 3× to 6× in the optimized version. These speed-ups will help large-scale combustion simulations on the current and future GPU-based supercomputers.

6.1 CPU Performance Enhancements: Phase I

This section analyzes the performance challenges and current limitations of Hypre with OpenMP. Then, the MPI endpoints approach is explained, followed by experimental evaluation on a large-scale system.

6.1.1 Performance Analysis of OpenMP

To understand the slowdown of Hypre with OpenMP, the PFMG preconditioner and the PCG solver are profiled with a standalone code that solves a three-dimensional Laplace equation on a regular mesh, using a seven-point stencil. The solve step is the computational core since it runs iteratively whereas the setup is executed only once. This representative example mimics the use of Hypre in Uintah, where each MPI rank derives its patches (Hypre boxes) based on its rank and allocated the required data structures accordingly. Each rank owns a minimum of four patches to a maximum of 128 patches, where each patch is initialized by its rank owner. Intel's Vtune amplifier and `gprof` are used for profiling on a KNL node with 64 cores. The MPI Only version executes with 64 ranks (where one rank is assigned to each core), and the MPI + OpenMP version runs 1x64, 2x32, 4x16, 8x8, and 16x4 ranks and threads, respectively.

The Struct interface of Hypre is called – first to carry on the setup and then to solve the equations. The solve step is repeated 10 times to simulate timesteps in Uintah. Each test problem uses a different combination of domain and patch sizes: a 64^3 or 128^3 domain is used with 4^3 patches of sizes 16^3 or 32^3. A 128^3 or 256^3 domain is used with 8^3 patches of sizes 16^3 or 32^3. Multiple combinations of MPI ranks, number of OpenMP threads per rank, and patches per rank are explored and compared against the MPI Only version. Each solve step takes about 10 iterations to converge on average.

Two main performance bottlenecks observed during profiling are as follows:

1) **OpenMP synchronization overhead and insufficient work**. Fig. 6.1(a) shows the code structure of how an application (Uintah) calls Hypre. Uintah spawns its threads, generates patches, and executes tasks scheduled on these patches. When Uintah encounters the Hypre task, all threads join, and the main thread calls Hypre. Hypre then spawns its own OpenMP threads and continues. With 4 MPI ranks and 16 OpenMP threads in each, Vtune reports a Hypre solve of 595 seconds. 479 out of 595 seconds are consumed by the synchronization overhead in OpenMP, and 12 seconds in spin time. The PFMG-CG algorithm calls 1000s of microkernels during the solve step. Each microkernel performs operations such as matrix-vector multiplication, scalar multiplication, relaxation, etc., and uses OpenMP to parallelize over the patch cells. The multigrid nature of the PFMG algorithm causes the coarsening of the

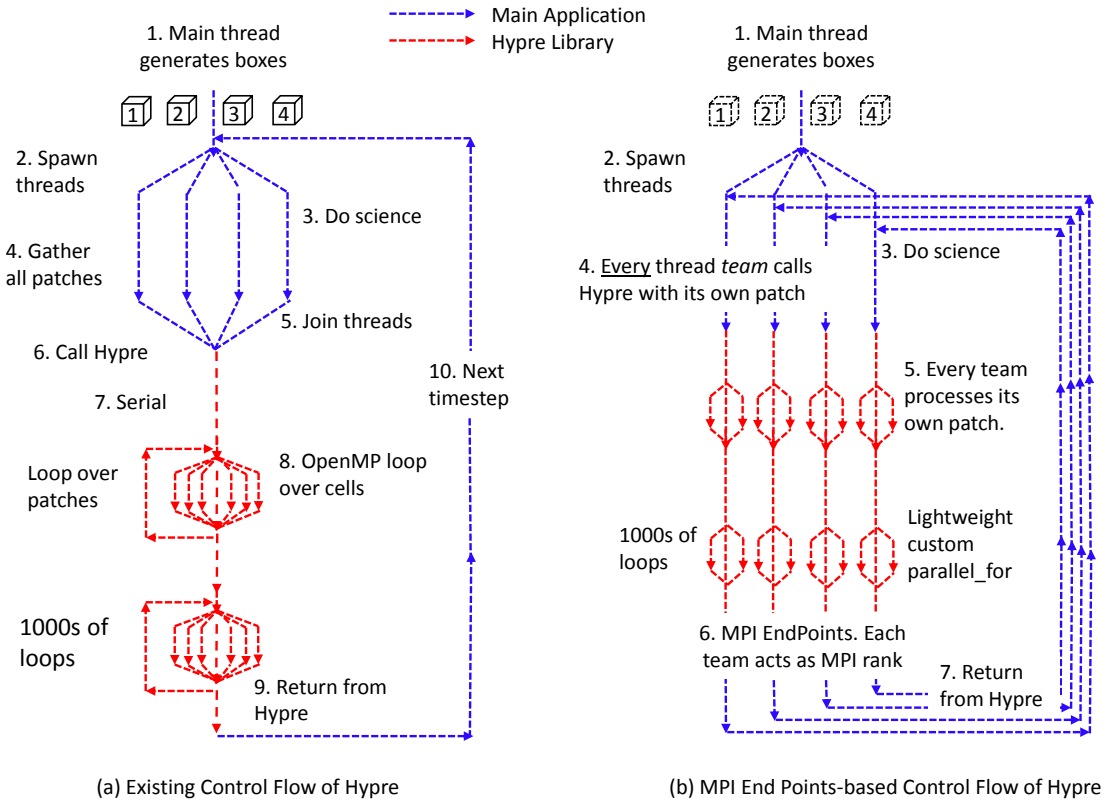


Fig. 6.1: Software design of Hypre.

mesh at every level. The number of cells reduces from n^3 to 1 and again increases back to n^3 with refining. Such a reduction in the number of cells makes the kernels extremely lightweight. The light workload is not enough to offset the overhead of the OpenMP thread barrier at the end of every parallel-for and results in 6× performance degradation. Moreover, the OpenMP overhead grows with the number of OpenMP threads per rank and patches per rank. As a result, Hypre does not benefit from multiple threads and cores.

2) **Failure of autovectorization**. Hypre uses loop iterator macros (e.g., BoxLoop), which expand into multidimensional "for" loops. These iterator macros use a dynamic stride passed as an argument. Although the dynamic stride is necessary for some use cases, many use cases have a fixed unique stride. As the compiler cannot determine the dynamic stride a priori, the loop is not autovectorized.

6.1.2 Restructuring OpenMP Loops and MPI Endpoints

A straightforward solution to the bottlenecks identified above is to parallelize the outermost loop, namely the loop at the patch level. This approach is evaluated for the Hypre function `hypre_PointRelax`. Table 6.1 shows execution times for the MPI Only, default hybrid MPI + OpenMP (with OpenMP pragmas around cell loops), and the new hybrid MPI + OpenMP implementations. In the new hybrid MPI + OpenMP code, parallelization is over mesh patches instead of cells, and each thread processes one or more mesh patches.

The parallelization over mesh patches improves the performance by 1.75×. Nonetheless, this is still 2× slower than the MPI Only version.

Probably, the simplest way to solve the synchronization problem is to make each thread behave like a separate MPI rank inside Hypre but act like a normal thread of the same MPI

Table 6.1: Comparison of MPI vs. OpenMP execution times using 64 32^3 mesh patches.

Hypre Configuration	**Execution time (s)**
MPI Only 64 ranks	1.45
Default hybrid: 4 ranks each with 16 threads, OpenMP over cells	5.61
New hybrid: 4 ranks each with 16 threads, OpenMP over patches	3.19
MPI endpoints: 4 ranks each with 4 teams each with 4 threads	1.56

rank outside Hypre. Such a threading model will allow Uintah to leverage the benefits of multithreading and also avoid all the OpenMP synchronizations happening inside Hypre. The model can be further refined to assign an MPI rank to teams of threads rather than individual threads, which will be a sweet spot between the two ends and can benefit from both. MPI endpoints (EP) propose the same idea of assigning MPI ranks to threads or team of threads instead of processes. Adopting the MPI endpoints (EP) approach, illustrated in Fig. 6.1(b), can bridge this performance gap. In this new approach, each of Uintah's team of threads acts independently as if it is a separate rank and calls Hypre, passing its patches. Each team processes its patches and communicates with other real and virtual ranks (*virtual rank = real rank* $\times$ *the number of teams + team id*). MPI wrappers convert virtual ranks to real ranks and vice versa during MPI communication. This conversion generates an impression of each team being an MPI rank, and the behavior is similar to the MPI Only implementation. The smaller team size (compared to the entire rank) minimizes overhead incurred in fork-joins in the existing OpenMP implementation, yet can exploit the abundant data parallelism available on manycore processors.

Nonetheless, the design and implementation of MPI EP are not without challenges which are outlined below.

1) **Race Conditions**: All global and static variables are converted to thread_local variables to avoid race conditions.

2) **MPI Conflicts**: The potentially challenging problem is to avoid MPI conflicts due to threads. In Hypre, only the main thread is designed to handle all MPI communications. With the MPI EP approach, each team makes its MPI calls. As Hypre already has MPI wrappers in place for all MPI functions, adding logic in every wrapper function to convert between a virtual rank and a real rank and to synchronize teams during MPI reductions is sufficient to avoid MPI conflicts.

3) **Locks within MPI**: The MPICH implementation, which is the base for Intel MPI and Cray MPI uses global locks. As a result, only one thread can be inside the MPI library for most of the MPI functions, which limits the new approach as the number of threads per rank is increased. To overcome this problem, an extra thread is spawned to handle the communication, and all the communication is funneled through this thread during

the solve phase, which provides minimum thread wait times and results in the highest throughput.

6.1.3 Optimizations in Hypre: Phase I

The three key optimizations in the proposed approach are described in this section.

6.1.3.1 MPI Endpoint

A dynamic conversion mechanism between the virtual and the real rank along with encoding of source and destination team ids within the MPI message tag simulates MPI endpoint behavior. Also, MPI reduce and probe calls need extra processing. These changes are described below.

1) `MPI_Comm_rank`: This command is mapped by using the formula above relating ranks and teams. Fig. 6.2 shows the pseudo-code to convert the real MPI rank to the virtual MPI EP rank using the formula "mpi_rank $\times$ g_num_teams + tl_team_id." The global variable `g_num_teams` and the thread-local variable `tl_team_id` are initialized to the number of teams and the team id. Thus, each endpoint gets an impression of a standalone MPI rank. Similar mapping is used in the subsequent wrappers.

2) `MPI_Send, Isend, Recv, Irecv`: The source and destination team ids are encoded in the tag values. The real rank and the team id are easily computed from the virtual rank by dividing by the number of teams.

3) `MPI_Allreduce`: All teams within a rank carry out a local reduction first, and then only the zeroth thread calls the `MPI_Allreduce` collective passing the locally reduced buffer as an input. Once the `MPI_Allreduce` returns, all teams copy the data from the globally reduced buffer back to their output buffers. Thread synchronization is achieved using lock-free busy waiting using C11 atomic primitives.

```
int g_num_teams;
__thread int tl_team_id;
int hypre_MPI_Comm_rank( MPI_Comm  comm, int *rank ){
   int mpi_rank, ierr;
   ierr = MPI_Comm_rank(comm, &mpi_rank);
   *rank = mpi_rank * g_num_teams + tl_team_id;
   return ierr;
}
```

Fig. 6.2: Pseudo code of MPI EP wrapper for `MPI_Comm_rank`.

4) `MPI_Iprobe and Improbe`: Each team is assigned a message queue internally. Whenever a probe is executed by any team, it first checks its internal queue for the message. If the handle is found, it is retrieved using `MPI_mecv`. If the handle is not found in the queue, then the `Improbe` function is issued, and if the message at the head of the MPI queue is destined for the same team, then `MPI_mecv` is again issued. If the incoming message is tagged for another team, then the receiving team inserts the handle in the destination team's queue. This method avoids the blocking of MPI queues when the intended recipient of the MPI queue's head is busy and does not issue a probe.

5) `MPI_GetCount`: In this case, the wrapper simply updates the source and tag values.

6) `MPI_Waitall`: The use of a global lock in MPICH `MPI_Waitall` stalls other threads and MPI operations do not progress. Hence, the `MPI_Waitall` wrapper is implemented by calling `MPI_Testtall` and busy waiting until `MPI_Testtall` returns true, which results in 15-20% speed-up over threaded `MPI_Waitall`.

6.1.3.2 Improving Autovectorization

The loop iterator macros in Hypre operate using a dynamic stride, which prevents the compiler from vectorizing these loops. To overcome this limitation, additional macros are introduced specifically for the unit stride cases. The compiler is then able to autovectorize some of the loops, which results in an additional 10 to 20% performance improvement depending on the patch size.

6.1.3.3 Lightweight parallel_for and Hierarchical Parallelism

A downside of explicitly using OpenMP in Hypre is possible incompatibilities with other threading models. In the spirit of [90], an interface is introduced that allows users to pass their version of `parallel_for` as a function pointer during initialization, and this user-defined `parallel_for` is called by simplified BoxLoop macros. Therefore, Hypre users can implement `parallel_for` in any threading model. The new interface allows a lightweight implementation of a threading model, in which all the threads, including the worker threads, are spawned at the beginning of the execution. The main thread of every team acts as an MPI endpoint, while worker threads do busy waiting until needed. When the main thread calls a `parallel_for`, it shares the C++11 lambda and iteration

count with the worker threads who execute the lambda in parallel. This approach is similar to OpenMP in principle but uses atomic primitives and busy waiting to achieve lock-free thread synchronization. As a result, the lightweight `parallel_for` performs faster than a `for` loop parallelized using `#pragma omp parallel for`, which typically uses pthread-based locks and conditional variables.

Table 6.2 shows the solve time per timestep in seconds for a problem with 64 patches of size 64^3 on a KNL node. The MPI Only version is run using 64 ranks with one patch per rank. The OpenMP and custom `parallel_for` versions are run using four ranks with four teams per rank and four worker threads per team. Each team gets four patches. Each rank spawns an extra communication thread. The only difference between the two threaded versions is the threading model. The lightweight `parallel_for` model runs 3.9×, 2.7×, and 1.5× times faster than the OpenMP version for the patch sizes 16^3, 32^3, and 64^3, respectively. Increasing the patch size reduces the performance gap between the two versions because the extra workload compensates for the OpenMP overhead. The lightweight `parallel_for` performs in a comparable way to the MPI Only version for a 32^3 patch and results in a speed-up of 2.7× over the OpenMP-based `parallel_for`. The performance improvement over Hypre MPI Only is due to vectorization and reduced MPI communication. The customized lightweight `parallel_for` makes these improvements apparent through lower overheads than the OpenMP-based `parallel_for`.

The main advantage of using hierarchical parallelism is to reduce the number of EPs compared to pure EP-based execution where each thread acts as an EP. Reducing the number of EPs reduces MPI communication, both point-to-point (such as `MPI_send-receive`) and collective communication (such as `MPI_Allreduce`). MPI EP with hierarchical parallelism can minimize overheads than either using all threads as EPs, which wastes time in `MPI_waitall` or using all threads as worker threads, which causes huge synchronization overheads (as in the default Hypre-OpenMP). As a result, the combination of EP and

Table 6.2: OpenMP vs. custom parallel_for on KNL: Execution times in seconds.

Patch Size	**16^3**	**32^3**	**64^3**
OpenMP	1.1	1.5	3.2
Custom `parallel_for`	0.28	0.56	2.1
MPI Only	0.15	0.5	2.8

hierarchical parallelism can result in the best performance.

6.1.4 Experimental Setup on Theta

Initial experiments using only the Hypre solve component on small node counts show that the performance improves with the patch size. Table 6.3 compares both hybrid MPI+OpenMP and MPI EP implementations against the MPI Only for different patch sizes. MPI+OpenMP always performs slower than the MPI Only version, although the performance improves marginally as the patch size increases. On the other hand, the MPI EP model performs nearly as well as the MPI Only version for 16^3 and 32^3 patch sizes on 2 and 4 nodes but breaks down at the end of scaling. With 64^3 patches, however, MPI EP performed up to 1.4× faster than the MPI Only version. As a result, a patch size of 64^3 is chosen for the scaling experiments on the representative problem. These results carry across to the larger node counts. Strong scaling studies with 16^3 patches show the MPI+OpenMP approach performs 4× to 8× slower than the MPI Only version. In the case of Hypre-MPI EP, the worst-case slowdown of 1.8× is observed for 512 nodes, and the fastest execution matched the time of Hypre-MPI Only. This experience, together with the results presented above, stresses the importance of using larger patch sizes, 64^3 and above, to achieve scalability and performance.

As the process of converting Uintah's legacy code to Kokkos-based portable code that can use either OpenMP or CUDA is still in progress, not all sections of the code can be run efficiently in the multithreaded environment. Hence, a representative problem containing the two most time-consuming components is chosen for the scaling studies on DOE Theta. The two main components are: 1) Reverse Monte Carlo Ray Tracing (RMCRT) that solves for the radiative-flux divergence during combustion [102]; and 2) pressure solve, which

Table 6.3: Speed-ups of the MPI+OpenMP and MPI EP versions compared to the MPI Only version for different patch sizes.

Patch size:	**16^3**		**32^3**		**64^3**	
Nodes	**MPI+ OpenMP**	**MPI EP**	**MPI+ OpenMP**	**MPI EP**	**MPI+ OpenMP**	**MPI EP**
2	0.2	0.9	0.2	1.2	0.5	1.4
4	0.2	0.8	0.2	0.9	0.4	1.4
8	0.2	0.5	0.3	0.6	0.5	1.3

uses Hypre. RMCRT has previously been ported to utilize a multithreaded approach that performs faster than the MPI Only version and also reduces memory utilization [154]. The second component, Hypre solver, is optimized as part of this work for a multithreaded environment. The combination of these two components shows the impact of using an efficient implementation of multithreaded Hypre code on the overall simulation of combustion.

Three mesh sizes are chosen for strong scaling experiments on DOE Theta: small (512^3), medium (1024^3), and large (2048^3). The coarser mesh for RMCRT is fixed at 128^3. Each node of DOE Theta contains one Intel's Knights Landing (KNL) processor with 64 cores per node, 16 GB of the high bandwidth memory (MCDRAM), and AVX512 vector support. The MCDRAM is configured in a cache-quadrant mode for the experiments in this paper. Hypre and Uintah are compiled using Intel Parallel Studio 19.0.5.281 with Cray's MPI wrappers and compiler flags "-std=c++11 -fp-model precise -g -O2 -xMIC-AVX512 -fPIC." One MPI process is launched per core (i.e., 64 ranks per node) while running the MPI Only version. For the MPI+OpenMP and MPI EP versions, four ranks are launched per node (one per KNL quadrant) with 16 OpenMP threads per rank. The flexibility of choosing team sizes in MPI EP allows running the multiple combinations of teams $\times$ worker threads within a rank: 16×1, 8×2, and 4×4. The best performing combinations among these are selected.

6.1.5 Results and Evaluation on Theta

Fig. 6.3a shows the execution time per timestep in seconds for the RMCRT component on DOE Theta. The multithreaded execution of RMCRT shows improvements between $2\times$ to $2.5\times$ over the MPI Only version for the small problem and $1.4\times$ to $1.9\times$ for the medium size problem. Furthermore, the RMCRT speed-ups increase with the scaling. This performance boost is due to the all-to-all communication in the RMCRT algorithm that is reduced by $16\times$ when using 16 threads per rank. The multithreaded version also results in up to $4\times$ less memory allocation per node. However, the RMCRT performance improvements are hidden by the poor performance of Hypre in the MPI+OpenMP version. Compared to the MPI Only version, a slowdown of $2\times$ is observed in Hypre MPI+OpenMP despite using 64^3 patches (Fig. 6.3b). The slowdowns observed are as worse as $8\times$ for smaller patch sizes. Using an optimized version of Hypre (MPI EP + partial vectorization) not only avoids these

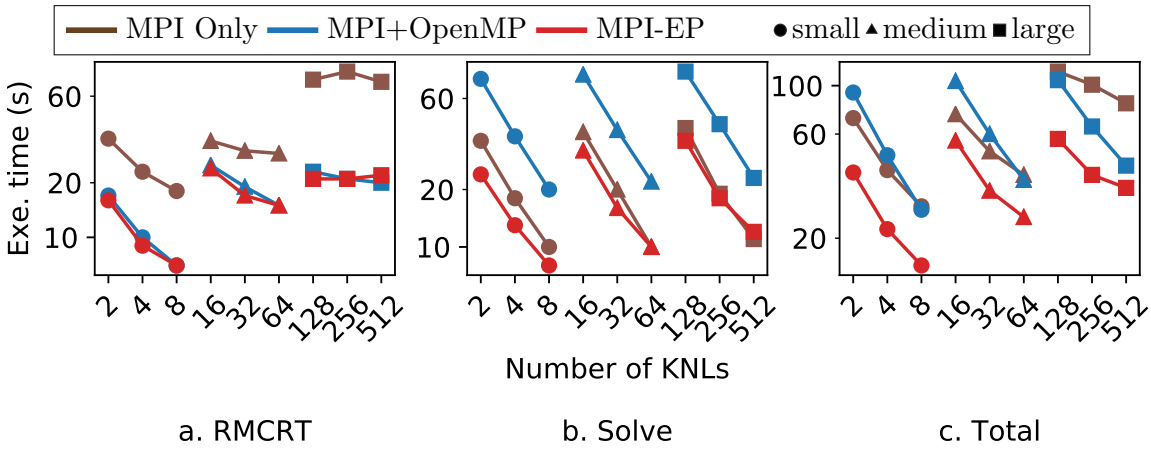


Fig. 6.3: Theta results: The execution time/timestep in seconds for RMCRT, Hypre and total time.

slowdowns but also provides speed-ups from 1.16-1.44× over the MPI Only solve. The only exceptions are 64, 256, and 512 nodes, where there is no extra speed-up for Hypre because the scaling breaks down. Because of the faster computation times (Fig. 6.3b), less time is available for the MPI EP model to effectively hide the communication and also wait time due to locks within MPI starts dominating. Table 6.4 shows the percentage of solve time spent waiting for communication. During the first two steps of scaling, the communication wait time also scales, but increases during the last step for eight and 64 nodes. The MPI wait time increases from 24% for 32 nodes to 50% for 64 nodes, and the communication starts dominating the computation because there is not enough work per node.

As both components take advantage of the multithreaded execution, the combination of the overall simulation leads to the combined performance improvement of up to 2× (Fig. 6.3c). These results show how the phase I optimizations to Hypre attribute to overall speed-ups of up to 2×.

Table 6.4: Theta results: Communication wait time for MPI EP.

Nodes	2	4	8	16	32	64	128	256	512
MPI Wait	2.4	1.4	1.7	6	3.9	5	11	7.5	6
Solve	24	13	8	32	16	10	36	18	12
% Comm	10%	11%	21%	19%	24%	50%	30%	42%	50%

6.2 CPU Performance Enhancements: Phase II

This section provides the performance analysis of Hypre after the first phase of optimizations. The second phase focuses on the communication-centered optimizations to improve scalability at large node counts based on the lessons learned from Phase I.

6.2.1 Performance Analysis of Phase II

The Theta results show significant improvements in the performance of Hypre and the entire application. However, they also reveal new bottlenecks in Hypre-EP and present opportunities for targeted optimizations to improve performance. Specifically, Hypre-EP stops strong scaling after 64 nodes while Hypre-MPI Only continues to scale (see Fig. 6.3b). The primary reason for the scaling breakdown is the dominating communication cost, as seen from Table 6.4. Half the solve time is spent waiting for the communication to complete on 64 and 512 nodes. Global locks within the MPI libraries make `MPI_THREAD_MULTIPLE` communication a major bottleneck. To avoid thread contention on locks in the MPI library, an extra thread is spawned per rank, and all communications are funneled through the communication thread. This method works faster than `MPI_THREAD_MULTIPLE` communication, but it serializes all the message exchange and hampers overall performance. The root of this problem can be addressed by an MPI library that does efficient `MPI_THREAD_MULTIPLE` communication and supports MPI endpoints functionality. Zambre et al. [196] demonstrate the benefits of utilizing network-level parallelism for MPI+threads applications by extending the MPICH implementation to use fine-grained critical sections and virtual communication interfaces (VCIs). VCIs map to the underlying network hardware contexts and represent dedicated communication channels that threads can map to using MPI endpoints or existing MPI objects such as communicators and tags. Using VCIs can help exploit underlying network level parallelism and can potentially address the serialization problem observed on Theta.

6.2.2 Optimizations in Hypre: Phase II

Studies show poor network bandwidth utilization when using fewer ranks per node [16, 29]. Therefore, an application needs to spawn multiple MPI ranks per node to best utilize the network resources. Mapping multiple threads to multiple VCIs within a rank can enable running a single rank per node and still efficiently use network resources. Such

a configuration opens up new opportunities for performance optimizations, as detailed below.

6.2.2.1 Nonblocking Interthread Communication

In the funneled communication approach of Phase I, all send/receive requests for EPs, which do not belong to the same rank, are pushed into an interprocess communication queue. The communication queue is monitored by the dedicated communication thread. The communication thread carries on subsequent communication in the background. After pushing the external messages to the communication queue, each EP handles interthread communication (i.e., communication with the other EPs of the same rank). Each EP first pushes all the send messages to its interthread send queue along with copies of the send buffers. Creating a copy allows the EP to continue processing as soon as all of its messages are received without waiting for the send to complete. After queuing all the sends, EPs busy-wait until all receive messages are placed on the queue by respective source EPs. However, if the VCIs enable running one rank per KNL node with 64 threads, efficient interthread communication become crucial.

A new nonblocking logic, similar to MPI Isend and Irecv, is introduced to enable efficient interthread communication. Each EP simply copies the pointers of send-receive, source/destination thread ids, message size, and tag to a global data structure and returns control back to the calling function. Every send / recv call also tries to progress the communication and copies data from the sender buffer to the receiver buffer if both the values are available. Finally, when `MPI_waitall` is called, EP keeps checking until all interthreads send-recv calls are completed. The global data structure allocates different slots to each EP. Every EP writes in its slot only and reads from other threads' slots while copying data. The read-write strategy allows the lock-free implementation of interthread communication.

As this optimization aims to improve multithreading, the experiments are conducted on a single rank without MPI in the mix. Hierarchical parallelism is not used to avoid synchronization overheads within each team. Each EP thus is a single thread and not a team of threads. Strong scaling is carried out on a single KNL node for 16, 32, and 64 threads with patch sizes of 16^3, 32^3, and 64^3. The number of patches is set to 64 - one per

core. Table 6.5 compares the communication times (in seconds) of nonblocking interthread communication with blocking interthread communication from Phase I. The MPI Only implementation is the baseline that spawns 16, 32, and 64 ranks, respectively. The measured communication time always increases as the number of threads increases. Nonblocking communication performs 20× times faster than the blocking communication for 16^3 and 32^3 patches when all 64 threads are spawned. The speed-up reduces to 5.6× for the 64^3 patch since the memory copy time becomes significant due to larger data exchanges. The improved communication impact on the overall solve time is shown in Table 6.6. The nonblocking code executed 6×, 3×, and 1.3× faster than the blocking version for 16^3, 32^3, and 64^3 patches, respectively, on using threads. As the patch size increases, computation becomes more dominant and results in fewer speed-ups. The nonblocking version gave speed-ups of 1.5×, 1.6×, and 1.5× over MPI Only code. The speed-up over the MPI Only model is because of both vectorization and reduction in communication wait time.

6.2.2.2 Communication-Reducing Patch Assignment

The assignment of a single multithreaded rank per node and nonblocking interthread communication can reduce MPI communication. MPI communication can be reduced further by a communication aware distribution of patches to ranks. The patch-assignment strategy of Phase I illustrated in Fig. 6.4 divides the total number of patches by the number

Table 6.5: Wait time in seconds for blocking and nonblocking interthread communication. MPI: MPI Only, B: Blocking, NB: Nonblocking.

Patch size	**16^3**			**32^3**			**64^3**		
Threads	**MPI**	**B**	**NB**	**MPI**	**B**	**NB**	**MPI**	**B**	**NB**
16	0.042	0.066	0.01	0.063	0.09	0.01	0.16	0.18	0.05
32	0.055	0.2	0.017	0.08	0.3	0.02	0.16	1	0.047
64	0.062	0.49	0.025	0.09	0.56	0.028	0.29	0.73	0.13

Table 6.6: Solve time in seconds for blocking and nonblocking interthread communication. MPI: MPI Only, B: Blocking, NB: Nonblocking.

Patch size	**16^3**			**32^3**			**64^3**		
Threads	**MPI**	**B**	**NB**	**MPI**	**B**	**NB**	**MPI**	**B**	**NB**
16	0.32	0.47	0.23	1.6	1.8	0.93	10	10	5.3
32	0.22	0.36	0.15	0.87	1.2	0.52	5.2	6.3	2.8
64	0.15	0.62	0.1	0.5	0.95	0.31	2.8	2.4	1.8

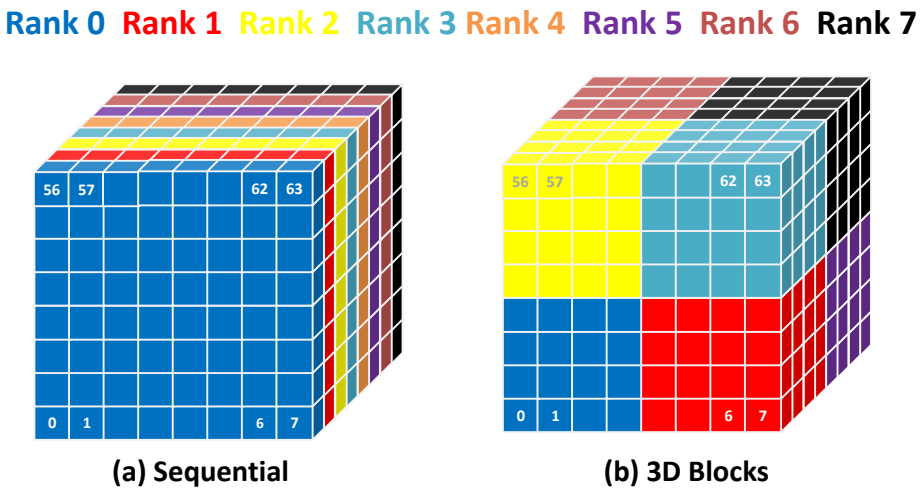


Fig. 6.4: Patch assignment strategies.

of ranks and assigns equal-sized chunks of patches sequentially to ranks. This strategy is similar to the storage of three-dimensional arrays in sequential memory. In Fig. 6.4, a grid containing 8×8×8 patches (i.e., a total of 512 patches) is divided among eight ranks. Hypre is run with this configuration on eight KNL nodes with one patch per core. As per the sequential assignment policy, rank 0 gets the first 64 patches, rank 1 gets the next 64 patches, and so on. Each rank gets one 8×8 slab of patches. Assuming 64 EPs per rank, each EP is assigned one patch. Thus, to gather the halo region for the patch, each EP has to communicate with 26 neighboring patches (ignoring face/corner cases) in a three-dimensional grid. Now consider any internal patch, which is not on the domain's face. The eight patches surrounding the patch in the same slab belong to the same rank (marked by the same color) and do not require any MPI communication. However, nine front patches and nine rear patches are assigned to different ranks (indicated by a different color), and gathering the halo region involves 18×2 (one send and one receive) = 36 MPI messages. The total number of MPI messages per rank becomes 64 (EPs) × 36 (messages per EP) = 2304. MPI messages from different EPs of the same rank cannot be combined, as this will create a local barrier among threads and thus hamper the performance. To avoid such a barrier (and to utilize the network to its full capacity), each thread needs to send its MPI messages independently.

The number of MPI messages can be reduced by a communication aware strategy where patches are divided into three-dimensional blocks and assigned to ranks. Such

an assignment reduces the number of patches facing those in another rank, and MPI communication is replaced with local interthread data exchange. Fig. 6.4 (b) shows the reassignment using three-dimensional blocking. Instead of dividing 512 patches into eight 8×8 slabs, patches are grouped as eight 4×4×4 blocks, and each rank is assigned one block. Thus, each EP gets one patch as before, but now each rank has only 16×3 = 48 patches facing other ranks. The duplicate patches along the edges are not eliminated because the communication direction is different for different faces. Each patch along the block's face requires only 9×2 MPI messages (one send, one receive). All other communication happens among local threads. Thus, the total number of MPI messages per rank is 48×18 = 864, which results in a 2.6× reduction in the number of MPI messages.

Computing these estimates for the full scale of the Theta supercomputer is straightforward. Assuming one patch per core, the total number of patches required to run on 4096 nodes is 64×64×64. The sequential patch assignment strategy assigns a strip of 64 consecutive patches to each rank. Ignoring the face and corner cases of the domain, each rank generates 24 (only two internal messages) × 64 (patches facing other ranks) × 2 (send and receive) = 3072 MPI messages per rank. On the other hand, the three-dimensional blocking patch assignment generates only 9 (other messages will be internal) × 16 × 6 (patches facing other ranks) × 2 (send and receive) = 1728 MPI messages per rank. Thus, the three-dimensional block assignment policy can reduce the number of messages by 1.7×.

The new patch assignment with the threaded implementation replaces MPI messages with more efficient interthread data exchange. The new policy can be used for the MPI Only version to replace internode MPI messages with intranode shared-memory MPI messages, but it will not reduce the total number of MPI messages. Because interthread communication is more efficient than shared-memory MPI communication, the new patch assignment is expected to benefit the EP version more than the MPI Only version.

6.2.2.3 Exposing Logical MPI Communication Parallelism

MPI libraries have recently made significant strides toward improving the communication performance of MPI_THREAD_MULTIPLE to match that of MPI everywhere [17, 149, 196]. A key contributing factor to this improved performance has been the mapping of independent MPI communication to network-level parallelism available

on modern interconnects. These new libraries, however, are helpless if applications do not distinguish between operations that are ordered and those that are independent. Hence, to leverage the high-speed multithreaded communication in these new libraries for Hypre-EP, exposing the communication independence through MPI is imperative.

In Hypre-EP, no ordering constraints apply for operations originating from different endpoints, that is, each endpoint's communication is logically parallel. A straightforward way to expose this logical communication parallelism with the MPI endpoints proposal is to use a distinct MPI endpoint for each Hypre EP. The MPI forum, however, has suspended the MPI endpoints proposal since existing MPI mechanisms, such as communicators, tags, and windows, can expose the same amount of parallelism as MPI endpoints. In fact, Hypre-EP's mechanism of encoding the sender and receiver thread IDs into the MPI tag to distinguish between messages targeting the same MPI rank can double as logical communication parallelism information. However, using tags is not sufficient with the existing MPI-3.1 standard because of the possibility of wildcards on the receive operations even though Hypre-EP does not use wildcards. The upcoming MPI-4.0 standard, however, introduces new MPI Info hints that allow applications, like Hypre-EP, to inform the MPI library that it does not use wildcards.

Hypre-EP leveraged the new hints mpi_assert_no_any_source and mpi_assert_no_any_tag in the draft MPI-4.0 standard, which allowed the MPICH implementation used in this work to utilize the encoded parallelism information in the tags to map to its multi-VCI infrastructure. Like MPI endpoints, tags with hints expose all of the available communication parallelism information. In the near future, most MPI libraries will be capable of mapping logical communication parallelism to the underlying network parallelism, be it through existing MPI objects or through MPI endpoints.

This work in Section 6.2.2.3 was contributed by Rohit Zambre.

6.2.2.4 One Rank per Node

As discussed earlier, VCIs allow efficient use of the network resources even with a single multithreaded rank per node. As a result, intranode MPI communication can be replaced with more efficient interthread communication and can result in a faster runtime. A simple change in the runtime configuration can allow running one rank per node with all the cores

on the node utilized by the rank. The configuration is not an enhancement by itself, but it maximizes the impact from the previous optimizations, such as interthread communication and communication-reducing patch assignment.

The second impact of using more threads is in the RMCRT component within Uintah. As seen from the Phase I results, multithreaded RMCRT with 16 threads per rank performs 2× to 4× faster than the MPI Only version. The reduction in the number of ranks reduces the cost of the all-to-all MPI communication in RMCRT. Multithreading can thus provide a higher payoff in RMCRT if the number of threads per rank is set to 64 instead of 16.

6.2.2.5 Hierarchical Parallelism

The final optimization is to enable hierarchical parallelism. As described in Section 6.1.3c, the combination of EP and hierarchical parallelism can minimize both MPI wait time and the thread synchronization overhead. The experiments show that the patch size and the team size are directly proportional, i.e., a smaller team size performs better for a small patch size. As the patch size grows, the workload of a `parallel_for` increases, and the additional synchronization cost due to a larger team size can be justified by improved overall performance. For a 64^3 patch, a team size of 4 results in the best performance.

6.2.3 Experimental Setup on Bebop

The Phase I experiments proved the effectiveness of multithreaded execution on the overall simulation by speeding up both RMCRT and Hypre. As the focus of this work is Hypre, the same experiments with RMCRT are not repeated. Instead, a standalone mini-app is built that replicates Uintah's calls to Hypre and produces the same output. The mini-app provides greater flexibility for in-depth analysis without incorporating other Uintah complexities such as task graphs, task schedulers, etc. All the experiments in Phase II are run using the mini-app.

The multi-VCI effort in MPICH currently supports only the Intel Omni-Path and Mellanox InfiniBand interconnects. Multi-VCI support for Theta's Aries interconnect is in progress. Hence, a new set of experiments is run on Argonne National Laboratory's Bebop cluster that uses the Intel Omni-Path interconnect. All the experiments on the Bebop cluster were run by Rohit Zambre. Bebop features 352 Intel Xeon Phi 7230 (KNL) nodes with 64 cores and 16 GB MCDRAM per node. These nodes support AVX512 vector instructions.

MCDRAM is set in the cache-quadrant mode for all the experiments. A customized version of MPICH with fine-grained lock and VCI support is used to compile the code along with Intel Parallel Studio 18.0.5. The flags provided for Hypre configuration include: "-std=c++11 -fp-model precise -g -O2 -xMIC-AVX512 -fPIC -fopenmp."

Multiple strong scaling runs are carried out to evaluate the performance of the new communication-centered optimizations for a small problem. Funneled communication from the Phase I experiments is now disabled because VCIs can efficiently perform multithreaded MPI communication. Hierarchical parallelism from Phase I is also disabled initially to correctly measure the impact of using VCIs. Thus the new baseline is a simple endpoint version with vectorization. Without funneled communication and hierarchical parallelism, the new baseline is expected to perform slower than both the MPI Only version and the most optimized version from Phase I. However, the use of this baseline allows the measurement of the impact of each optimization incrementally. With this aim in mind, the experiments are carried out using the following levels of optimizations from Section 6.2.2:

- EP Baseline: Basic MPI endpoints without funneled communication and hierarchical parallelism.
- O1: Baseline + Nonblocking interthread communication.
- O2: O1 + Communication-reducing patch assignment.
- O3: O2 + VCIs
- O4: O3 + VCIs + one rank per node with 64 threads per rank.
- O5: O4 + Hierarchical parallelism.

The baseline and the O1, O2, and O3 optimizations are run with four ranks per node and 16 EPs per rank. The O4 and O5 optimizations are run using one rank per node with 64 EPs per rank. The O5 enhancement uses different combinations of the number of EPs (i.e., thread teams) $\times$ the number of threads per team (i.e., team size). The best performing results are chosen from the various combinations: 64×1, 32×2, and 16×4. The MPI Only version is run with 64 ranks per node (i.e., one rank per core) as a reference.

The number of patches is chosen such that there will be one patch per core at the end of strong scaling. Strong scaling is carried out at three levels.

1) A **small** problem of size 512^3 divided among 512 patches ($8\times8\times8$) is run on 2, 4, and 8 nodes to incrementally test the various optimizations.

2) A **medium** problem consists 1024^3 cells divided among 4096 patches ($16\times16\times16$) and is executed on 16, 32, and 64 nodes.

3) A **large** problem consists of 2048^3 cells decomposed into 32k patches ($32\times32\times32$) and is run on 128 and 256 nodes of Bebop, respectively. Unfortunately, the Bebop cluster does not have 512 nodes, and the last step in strong scaling cannot be performed.

In all the cases, the patch size remains fixed, i.e., 64^3. Medium- and large-scale problems are run on a larger number of nodes to compare the most optimized EP version with the MPI Only model. Each problem is run for 10 timesteps and for 60 CG-solver iterations per timestep. The setup time is discarded as it is done only during the zeroth timestep. Average solve time and communication time from the rest of the timesteps are measured and presented in the subsequent sections.

6.2.4 Results and Evaluation on Bebop

Table 6.7 compares the strong scaling of solve and communication times of the different optimizations for a 512^3 problem. The MPI Only version provides a reference. As expected, turning off funneled communication and hierarchical parallelism slowed down the Hypre-EP baseline more than Hypre-MPI Only. However, overall performance improved with each optimization, and the final version is faster than the MPI Only version. Enabling

Table 6.7: Solve and communication times in seconds for the *small* problem with different optimizations.

	Solve time							**Communication time**						
Nodes	**MPI**	**EP**	**O1**	**O2**	**O3**	**O4**	**O5**	**MPI**	**EP**	**O1**	**O2**	**O3**	**O4**	**O5**
2	39	39	33	29	36	24	24	6.5	20	14	9.2	16	2.4	1.6
4	24	32	21	20	34	14	12	7.4	23	12	10	24	3	1.2
8	17	28	20	16	28	9.7	7	7.1	24	15	12	22	3.5	1.4

MPI: MPI Only, 64 ranks/node
EP: EP baseline, no funneled comm, four ranks/node, 16 EPs/rank
O1: EP + Nonblocking interthread comm
O2: O1 + Comm reducing patch assignment
O3: O2 + VCIs
O4: O3 + 1 rank/node, 64 EPs/rank
O5: O4 + hierarchical parallelism, one rank/node, 16 EPs/rank, four worker threads/EP

nonblocking interthread communication improved EP-baseline performance by 1.3–1.4×, and EP now runs within +/- 15% of the MPI Only version. The second optimization–communication-reducing patch assignment–did not show much impact on two and four nodes but improves performance by 20% on eight nodes, which makes Hypre-EP perform as fast or even faster than Hypre-MPI Only. Introducing VCIs, however, proved to be inefficient. The VCI abstractions in the MPICH library are not yet tuned for intranode shared memory communications, which resulted in the performance drop shown in the O3 column of Table 6.7. The situation is remedied by using a single rank per node, as indicated by column O4, which is the primary goal in introducing VCIs, and the speed-up over the MPI Only version improves by 1.7× on four and eight nodes. Hierarchical parallelism provides an additional performance increase as indicated by column O5, and the final improvement over the MPI Only version reaches 2× and 2.4× on four and eight nodes, respectively. The Phase I optimizations results in 1.3–1.4× speed-up for the same setup, which indicates that the Phase II optimizations further improve performance by 1.7× over Phase I. The main reason behind this significant improvement is the reduced communication wait time. The O5 optimization in Table 6.7 shows up to a 5× reduction in communication wait time than the MPI Only version. Fig. 6.5 shows strong scaling of Hypre-MPI Only and Hypre-EP-O5, the most optimized EP version, up to 64 nodes. The performance

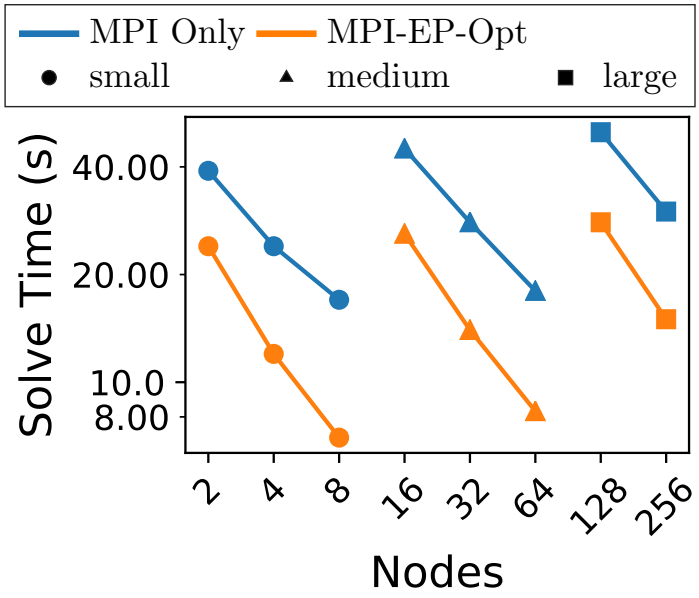


Fig. 6.5: Strong scaling of solve time.

improvements for a 512^3 problem on two, four, and eight nodes are 1.6×, 2×, and 2.4×, respectively. The 1024^3 problem shows improvements of 1.7×, 2×, and 2.1× on 16, 32, and 64 nodes, respectively. The large problem of size 2048^3 run on 128 and 256 nodes of Bebop demonstrated improvements of 1.78× and 2×, respectively. The Phase I experiments are run on Theta and Phase II experiments on the Bebop cluster. Although both machines deploy Intel's KNL processors, the underlying networks are different. Hence, the results cannot be directly compared with each other. Therefore, the performance of Hypre-EP over Hypre-MPI Only is compared between Phase I and Phase II, as shown in Fig. 6.6. Recall that Phase I hits the strong scaling limit at 64 nodes and 256 nodes for the medium- and large- size problems, respectively, and the results show no improvements over the MPI Only version. Furthermore, the speed-ups during Phase I decreased for the given problem size as strong scaling progressed. Phase II optimizations address this communication bottleneck and result in faster overall execution time and speed-ups up to 2× on 256 nodes. Table 6.8 compares the communication wait times of the experiments in Phase I and Phase II. Phase II shows a significant reduction in the percentage of time spent in communication wait routines on 16 through 256 nodes. As a result, the speed-ups over the MPI Only version increase for a given problem size as the number of nodes increases, which indicates

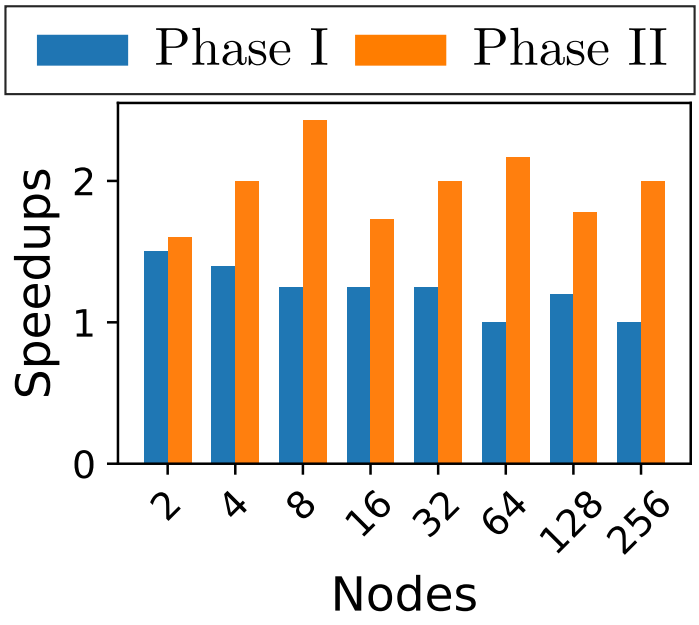


Fig. 6.6: Speed-ups over Hypre-MPI Only.

Table 6.8: Percent communication wait time for MPI EP in Phase I vs. Phase II.

Nodes	2	4	8	16	32	64	128	256
Phase I	10%	11%	21%	19%	24%	50%	30%	42%
Phase II	7%	10%	20%	12%	19%	28%	18%	27%

improved strong scaling.

6.3 GPU Performance Enhancements

Although Hypre has had CUDA support from version 2.13.0, version 2.15.0 is used here to characterize performance, to profile for bottlenecks, and to optimize the solver code. The GPU experiments are carried out on LLNL's Lassen cluster. Each node is equipped with two IBM Power9 CPUs with 22 cores each and four Nvidia V100 GPUs. Hypre and Uintah both were compiled using gcc 4.9.3 and CUDA 10.1.243.

6.3.1 Performance Characterization and Optimizations for GPUs

The initial performance characterization was done on 16 GPUs of Lassen using a standalone mini-app that called Hypre to solve a simple Laplace equation and run for 20 iterations. GPU strong scaling is carried out using 16 "super-patches" of varying sizes 44^3, 64^3, and 128^3. The observed GPU performance is evaluated against the corresponding CPU performance, which is obtained using the MPI only CPU version of Hypre. Thus, corresponding to every GPU, 10 CPU ranks are spawned, and super-patches are decomposed into smaller patches into smaller patches to feed each rank, keeping the total amount of work the same. Fig. 6.7 shows the CPU performs 5x faster than the GPU for patch size 44^3. Although 64^3 patches decrease the gap, it takes a patch size of 128^3 for GPU to

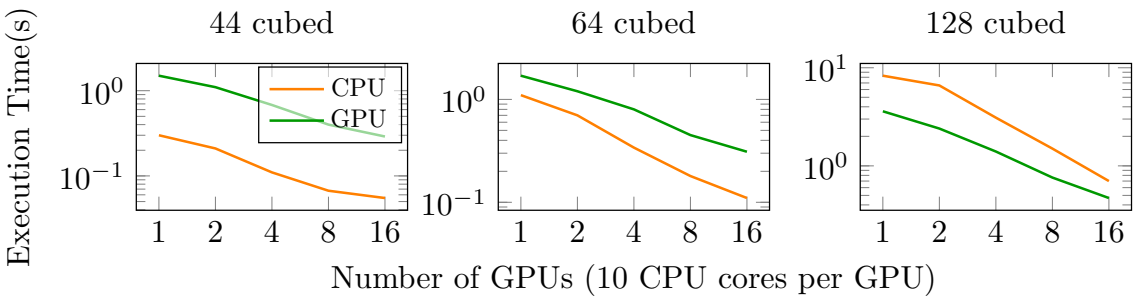


Fig. 6.7: GPU performance variation based on patch size.

justify overheads of data transfers and launch overheads and deliver better performance than CPU. Based on this observation, all further work was carried out using 128^3 patches. HPCToolkit and Nvidia nvprof were used to profile CPU and GPU executions. The sum of all GPU kernel execution times shown by nvprof was around 500ms, and the total execution time was 1.6 seconds. Thus, the real computation work was only 30%, and nearly 70% of the time was spent in the bottlenecks other than GPU kernels. CPU profiling revealed about 30 to 40% time consumed in for MPI wait for sparse matrix-vector multiplication and relaxation routines. Another 30 to 40% of solve time was spent in the CUDA kernel launch overhead. It should be noted that although the GPU kernels are executed asynchronously, the launching itself is synchronous. To justify the launching overhead, the kernel execution time should be at least 10μs - the launch overhead of the kernel on V100 (which was shown in the nvprof output).

Table 6.9 shows the top five longest running kernels for the solve time of 128^3 patches on 16 GPUs with one patch per GPU. InitComm and FinComm kernels, which are used to pack and unpack MPI buffers, are fourth and fifth in the list. The combined timing of these two kernels can take them to the second position. More interestingly, together these kernels are called for 41,344 times, but the average execution time per kernel execution is just 1.8 μs. On the other hand, the launch overhead of the kernel on V100 is 10μs (which was revealed in the profile output). Thus, the launch overhead of pack-unpack kernels consumes 0.4 seconds of 1.6 seconds (25%) of total execution time.

The existing implementation iterates over neighboring dependencies and patches and launches the kernel to copy required cells from the patch into the MPI buffer (or vice versa). This implementation results in thousands of kernel launches, as shown in Table 6.9, but the work per launch remains minimal due to simple copying of few cells. The problem can

Table 6.9: Top five longest running kernels before and after merging.

Before merging			**After Merging**		
Name	**Calls**	**Avg Time**	**Name**	**Calls**	**Avg Time**
MatVec	3808	29.067us	MatVec	3808	29.040us
Relax1	2464	22.453us	Relax1	2464	22.463us
Relax0	2352	19.197us	Relax0	2352	19.126us
InitComm	20656	1.8650us	Axpy	1660	21.484us
FinComm	20688	1.8310us	Memcpy-HtoD	12862	2.0750us

be fixed by fusing such kernel launches - at least for a single communication instance. To remedy the situation, the CPU code first iterates over all the dependencies to be processed and creates a buffer of source and destination pointers along with indexing information. At the end, all the buffers are copied into GPU's constant memory cache, and the pack (or unpack) CUDA kernel is launched only once instead of launching it for every dependency. After the fix, InitComm and FinComm disappeared from the top five longest running kernels, as shown in Table 6.9. The combined number of calls for InitComm and FinComm reduced from 41,344 to 8338. As a result, the communication routines perform 3x faster than before, and the overall speed-up in solve time achieved was around 20%. The modified code adds some overhead due to copying value to the GPU constant memory, which is reflected Memcpy-HtoD being called 12862 times compared to 4524 times earlier, but still the new code performs faster.

With the first major bottleneck resolved, the second round of profiling using HPCToolkit showed that the MPI wait time for matrix-vector multiplication and for relaxation routines was now more than 60%. The problem is partially overcome by using CUDA-aware MPI supported on Lassen. The updated code directly passes GPU pointers to the MPI routines and avoids copying data between host and device. Using CUDA-aware MPI decreased the communication wait time to 40 to 50% and resulted in an extra speed-up of 10%.

6.3.2 GPU Experiments on LLNL's Lassen

The GPU experiments were carried out on LLNL's Lassen cluster. Each node is equipped with two IBM Power9 CPUs with 22 cores each and four Nvidia V100 GPUs. Hypre and Uintah both were compiled using gcc 4.9.3 and CUDA 10.1.243 with compiler flags "-fPIC -O2 -g -std=c++11 –expt-extended-lambda."

Strong and weak scaling experiments on Lassen were run by calling Hypre from Uintah (instead of mini-app), and the real equations originating from combustion simulations were passed to generate the solve for the pressure at each mesh cell. Strong scaling experiments were conducted using three mesh sizes: small (512x256x256), medium (512^3), and large (1024^3). Each mesh is divided among patches of size 128^3 - such that each GPU gets one patch at the end of the strong scaling. CPU scaling was carried out by assigning one MPI rank to every available CPU core (40 CPU cores/node) and by decomposing the mesh into

smaller patches to feed each rank.

6.3.3 GPU Results on Lassen

The strong scaling plot in Fig. 6.8 shows the GPU version performs 4x faster than the CPU version in the initial stage of strong scaling when the compute workload per GPU is greater. As the GPU version performs better than the CPU version, it runs out of compute work sooner than the CPU version, and the scaling breaks down with the speed-up reduced to 2.3x. Similarly, the optimized GPU version performs up to 6x faster than the CPU version (or 1.44x faster than the baseline GPU version) with the heavy workload. As the strong scaling progresses, the speed-up by the optimized version against CPU reduces to 3x (or 1.26x against baseline GPU version). The communication wait time of both GPU versions is reduced by 4x to 5x as the number of ranks is reduced by 10 times (not shown for brevity). Thanks to faster computations, the optimized GPU version spends 15 to 25% more time in waiting for MPI compared to the baseline GPU version.

The weak scaling was carried out using one 128^3 patch per GPU (or distributed among 10 CPU cores) from four GPUs to 512 GPUs. Fig. 6.9 shows good weak scaling for all three versions. The GPU version shows 2.2x to 2.8x speed-up, and the optimized GPU code performs 2.6x to 3.4x better than the CPU version.

Preliminary experiments with the MPI EP model on Lassen showed that the MPI EP CPU version performed as well as the MPI Only CPU version (not shown in Fig. 6.8 for

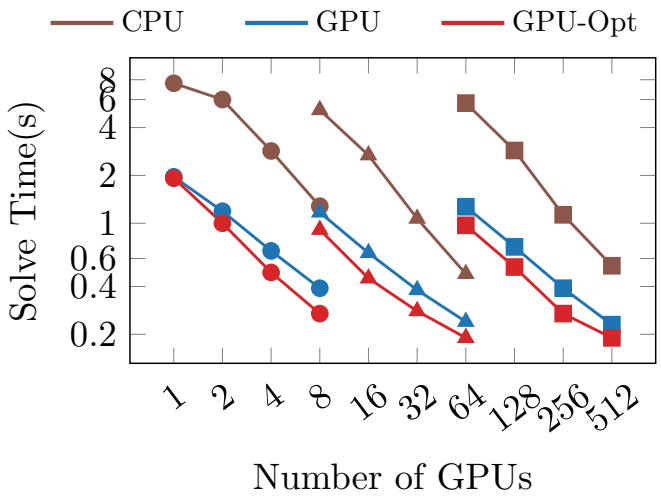


Fig. 6.8: Strong scaling of solve time.

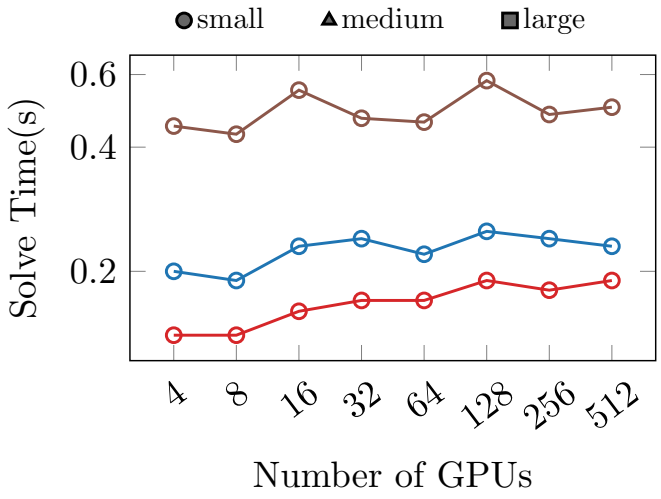


Fig. 6.9: Strong scaling of solve time.

brevity). Work is in progress to improve GPU utilization by introducing the MPI EP model for the GPU version and assigning different CUDA streams to different endpoints, which may improve overall performance.

6.4 Conclusions and Future Work

This work shows that the MPI-endpoint approach makes a threaded version of Hypre up to 2.4x faster than the MPI-only version. One of the bottlenecks for the MPI EP version was locks within MPI. This problem was resolved using an MPICH implementation that supports fined-grained locks and allows threads to exploit network-level parallelism using VCIs. It proves the impact of using VCIs on a real-life application and shows how a combination of a well-designed multithreaded implementation supported by VCIs can lead to better performance on modern architectures. The overall performance improvement of 1.7x–2.4x and a communication wait time reduction up to 5x can prove to be a significant advantage on CPU-based supercomputers such as Fugaku and TACC Frontera in the future. Other multithreaded applications can benefit from the combination of VCIs-supported endpoints and hierarchical parallelism to achieve overall speed-up as demonstrated by Uintah and Hypre on the Theta supercomputer and Bebop cluster.

Similarly, improved GPU speed-ups can help in gaining overall speed-ups for other Hypre-CUDA users.

Not only Hypre, but other third-party libraries, legacy codes, and AMTs may also benefit

from VCIs. It will be exciting to see if VCIs can be used to avoid any communication bottlenecks in Uintah. On GPUs, the current optimized version shows around 40 to 50% of time is consumed in waiting for MPI communication during sparse matrix-vector multiplication and relaxation routines. If the computations and communications are overlapped, then a new kernel needs to be launched to undertake the dependent computations after the communication is completed. As these kernels do not have enough work to justify the launch, this implementation resulted in slightly slower overall execution times during the initial experiments of overlapping communications. Similar behavior was observed by [28]. A possible solution is to collect kernels as "functors" and to launch a single kernel later, which calls these functors one after another as a function call. Peterson's improved asynchronous kernel launching in Kokkos and used the lesser number of CUDA blocks and threads to share GPU between multiple tasks to improve throughput [150]. Whether the combination of Peterson's scheme and the new threading model based on MPI endpoints improves the overall throughput on GPU is worth investigating. Another option for speeding up the algorithm is to use communication-avoiding approaches, e.g., see [108], which uses a multigrid preconditioner and spends less than 10% of the solve time in the global MPI reductions on Summit. As this work here also used a multigrid preconditioner [169], similar behavior was observed in our experiments, and the global reduction in the CG algorithm is not a major bottleneck so far. However, these options will have to be revisited when running the code to full-scale combustion problems at exascale.

CHAPTER 7

ENHANCING HETEROGENEOUS MPI + PPL TASK SCHEDULER FOR AMT SYSTEMS

The complexity of anticipated nodes in exascale systems poses new challenges for large-scale simulations. As mentioned earlier in Chapter 2, AMT runtime systems and MPI + X hybrid parallelism approaches promise to manage the increased concurrency, deep memory hierarchies, and heterogeneity. On the other hand, code portability becomes crucial considering the diverse architectures of current and emerging supercomputers, e.g., ARM CPUs (Fugaku), Intel CPUs (NSF Frontera), Nvidia GPUs (DOE Summit), Intel GPUs (DOE Aurora), and AMD GPUs (DOE Frontier). Performance portability layers (PPL) can manage this architectural diversity, as described in Chapter 2. However, integrating these solutions poses difficulties for AMT runtime systems. To keep pace with emerging HPC systems, all the components, namely PPLs, AMT infrastructure, user code using AMTs, and third-party libraries, are developed rapidly but at a varying pace. To complicate the matter further, developers of third-party libraries are facing similar challenges.

The Uintah AMT had an experimental heterogeneous task scheduler [150] capable of executing portable tasks and a PPL built using Kokkos the PPL [90, 150]. The primary goal was to run a simulation with 100s of heterogeneous portable tasks. However, the following shortcomings prevented the combination from doing full-scale runs.

1) To avoid excessive synchronizations, the task scheduler forced all tasks to have the exact same number of halo cells.
2) The scheduler supported "read-only" and "write-only" dependencies, but not "read-write" dependencies between tasks.
3) Multiple race conditions triggered frequent application crashes.
4) The third-party library tasks were not ported to GPUs.

5) The parallelism models of Uintah did not match with that of the third-party libraries.
6) Many legacy tasks were not ported to use PPL.
7) The scheduler was based on pthreads and allowed heterogeneous execution of CUDA tasks and serial CPU tasks, but not data-parallel OpenMP CPU tasks.

As a result, running a complex simulation with 100s of tasks on GPUs became impossible. The current author addressed (3) and (4), John Holmen tackled (7), and both worked together to get solutions for (1), (2), (5), and (6). This work enhanced the existing heterogeneous task scheduler to plug all the gaps listed above. These enhancements were done with the overarching goal to have a new heterogeneous MPI + PPL task scheduling approach, which was published in [93]. Apart from these fixes, Holmen also contributed design considerations for the following scheduling approach, which are listed in [93].

The new task scheduling approach:

- addresses the challenges in integrating the standalone solutions to prepare AMTs for exascale execution,
- achieves scalable, adaptive execution of 100s of individually portable tasks,
- showcases a smooth transition between different parallelism models used by AMTs and third-party libraries,
- allows simultaneous execution on host and device both through a performance portability layer on complex heterogeneous nodes,
- enables application developers to use a complex heterogeneous node efficiently without knowing low-level details of the underlying hardware and programming models,
- provides the foundation for a fully portable heterogeneous MPI + PPL task scheduling approach for the infrastructure developers (Here, portable refers to an approach using portable abstractions for task scheduling in addition to portable abstractions inside individual tasks).

The resulting approach aims to ease scheduler implementation in similar runtimes and also helps prepare Uintah for early use of the DOE Aurora system through the Aurora Early Science Program. This work enabled GPU support for the Arches component and allowed

100s of generic, production-ready, portable Arches tasks running on GPU for the first time. Arches previously lacked GPU support beyond the simple test cases in [90]. Although the implementation is limited to support for NVIDIA GPUs as of now, high-level ideas associated with this approach are broad enough to apply to other GPUs.

7.1 Infrastructure Improvements

Fig. 7.1 shows Uintah's heterogeneous task scheduler. The scheduler takes the task graph as input and spawns multiple threads. Each thread acts as a task executor that can carry out different activities (shown by steps 1 through 9 in Fig. 7.1) independently on eligible tasks. Task executors begin at Step 1 "Post MPI Recvs," where they post MPI receive messages for all the eligible tasks. The task executor can:

- initiate MPI communication (steps 1 and 8), or
- initiate *asynchronous* host-to-device / device-to-host copies (steps 2 and 6), or
- execute a GPU task, which in turn launches CUDA kernels *asynchronously*, or can run a CPU task serially without using CUDA (step 4), or

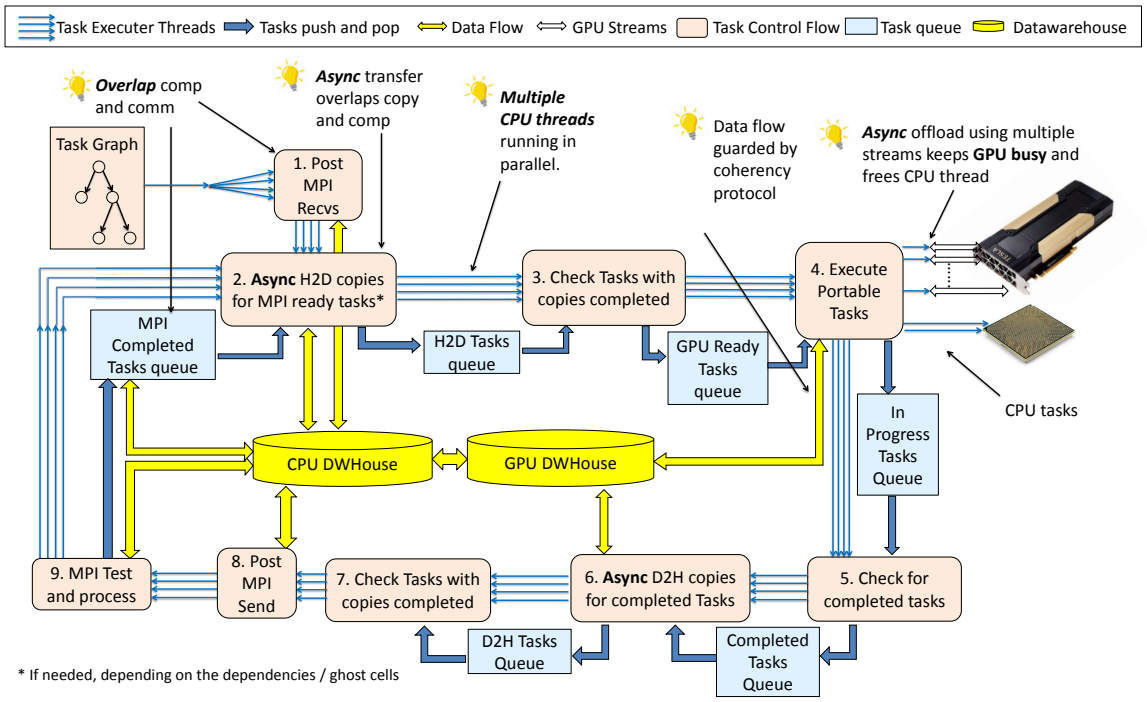


Fig. 7.1: Asynchronous heterogeneous portable task scheduler.

- check the status for the pending tasks (steps 3, 5, 7, and 9).

All the actions executed by the scheduler are done in an asynchronous fashion, except the CPU execution of any task. The scheduler maintains one task queue for every step. A task executor performs the current activity and pushes the task in the next wait queue. Thus, tasks progress from one queue to another, e.g., when MPI receives are completed for a GPU task, an asynchronous host-to-device copy is initiated to copy necessary data structures/halo cells to GPU (step 2), and the task is pushed into the "Htwo-dimensional Tasks queue." Sometime in the future, the copy status is checked for the task (step 3), and the task is promoted to the "GPU Ready Tasks queue" when the copy is completed. Again tasks from "GPU Ready Tasks queue" are asynchronously executed by some task executor, and tasks go to "In Progress Tasks Queue," and so on. Host or device execution is determined by the application developer using a task tagging system implemented as a part of the portable abstractions [90] that accompany this task scheduler.

The remaining section covers the enhancements in this scheduler and the related infrastructure, which helped in reaching the final goal of large-scale runs.

7.1.1 Halo Cells

While creating a Uintah task, developers can specify "read-only" dependencies with the number of halo cells (or ghost cells) needed around the patch for every simulation variable. A common example is a stencil calculation where each cell is dependent on six neighboring cells in a three-dimensional grid and needs to access cells from neighboring patches. If the neighboring patch is not present on the same rank, then the task graph compilation phase creates the corresponding dependencies between the current task and the earlier task on the neighboring rank. Using these dependencies, the task scheduler then fetches the halo cells from the neighboring rank as a "foreign patch" using MPI communication before starting the task execution. During the task execution, Uintah creates the necessary patch + halo cells combination "on-demand" in a three-step process: 1) allocate a new variable with enough memory to accommodate all the cells; 2) copy the main patch corresponding to the task into the new memory; and 3) copy halo cells from neighboring patches, local or foreign, around the main patch values into the new variable. However, doing memory allocation for every such variable access on GPU will cause huge synchronization overhead. To avoid the

excessive synchronization, the heterogeneous scheduler assumed that the ghost cell values would be the same for a given variable for all tasks throughout the simulation and allocated GPU memory only once for a given number of halo cells [150]. The solution needed users to manually find out the maximum possible number of ghost cells for every variable across all tasks for the entire simulation with 100s of tasks, and then update all the tasks to use the same maximum value. Another challenge was if the variable is computed on a GPU, it will be computed without halo cells, and then the reallocation (and synchronization) would be mandatory for the subsequent tasks, which may request ghost cells. Again, the temporary fix demanded users to replace all the calls to create compute dependencies with an alternate method to create compute dependencies with ghost cells, which would allocate extra memory for future halo cells. The users need to pass the maximum number of ghost cells for the given variable across all tasks.

This solution worked well for a few test problems. However, it was not practical, involved manual labor, and would have resulted in a very fragile code. To fix the problem, the idea of "create compute dependencies with ghost cells" was scrapped. The new solution automated the logic to find out the maximum number of halo cells and overrode the number of halo cells for all the variables and all three types of dependencies across all tasks before compiling the task graph. Steps 2 and 6 in Fig. 7.1 were also updated to consider the maximum number of ghost cells while carrying of data transfers between host and device.

7.1.2 Read-Write Dependencies

The previous heterogeneous scheduler did not handle the read-write dependencies between tasks and cannot execute such tasks. This missing piece was the biggest roadblock in running full-scale simulations using portable tasks on GPUs. The following changes were made to bridge this gap:

- **Data transfer**: Steps 2, 3, 6, and 7 were modified to transfer variables having "read-write" dependencies to and from GPU, if needed by the task. The earlier halo cells fix helped here to allocate the correct amount of memory and avoided conflict with read-only dependencies of subsequent tasks.
- **Correct status**: The important part of this transformation is to know the correct status of the simulation variables all the time. Peterson implemented a set of atomic bit set

operations to indicate the current status of the variable [150]. New code was added in step 4 to "invalidate" the host and device copies of any variable being modified by the task (i.e., read-write dependency for the variable). At this point, both the execution spaces - host and device - show the variable to be invalid, and any tasks with "read-only" or "read-write" dependencies keep waiting until at least one of the execution space show up with a valid copy. The fix is very important because AMT is data-driven and can start executing any tasks whenever valid data are available to satisfy its dependencies. The fix prevents other tasks from accessing the stale copy of the variable. Step 5 was updated to mark the modified variable as "valid" on completion of the task, only on the execution space of the task. The variable on the alternate execution space still remains invalid, which in turn invokes appropriate transfer routines. Step 4 also marks ghost cells of the current patch invalid under the assumption all patches will execute similar "modify" tasks sooner or later, and subsequent tasks regather the halo cells before execution begins.

- **MPI communication**: Uintah's automatic MPI generation takes care of data exchange between ranks whenever a simulation variable is modified and is later needed by a task on another rank. However, step 2 had to be updated to ensure GPUs receive the latest value from the host whenever any variable is modified. Similarly, step 6 was updated to transfer the modified variable from GPU to CPU, which is then shipped to the destination rank Uintah's automated MPI. These two changes complement the invalidating of ghost cells in the earlier step.

7.1.3 Data Races

Due to all the complexities of the scheduling logic and many parallel in flight operations, multiple race conditions existed in the application. Some of the race conditions triggered frequent application crashes and gave inaccurate results, whereas others were reproduced once in 20 runs. Four of these race conditions are explained briefly, and the experience of dealing with them will be a good lesson for other AMTs and applications.

- **Freeing memory**: The root cause of one race condition turned out to be freeing the allocated memory by one thread while other threads are still reading it. The tools to detect the race conditions (e.g., Thread Sanitizer and Intel Inspector) typically catch

race conditions occurring during simultaneous read-write or write-write, but failed to detect this particular case. The tools will detect the simultaneous read - write only if the freed memory is reallocated to some other data structure, but that would be a false positive because the root cause was different. One should be extremely careful while freeing memory and should think about possible race conditions because detecting them is extremely tricky.

- **Conflict during host-to-device transfer**: While transferring a variable without ghost cells from host to device, memory is still allocated to accommodate the future need of halo cells, but the values are not updated. A conflict when one thread transfers a variable with nonzero ghost cells and another thread with zero ghost cells. If the thread with zero ghost cells executes later, it overwrites the earlier valid values with garbage. A code was added to wait for zero ghost cell copy to wait in such a case.
- **Conflict in CPU halo gathering**: A conflict occurred when a CPU task accessed a variable and simultaneously another thread accessed the same variable in steps 2 or 6 of the scheduler. Although the code was guarded by the bit sets to maintain the variable status, the bit set was created only for GPU variables. CPU-only variables did not have any way to track the status and resulted in a race condition when used within the scheduler. A bit set similar to that created earlier [150] was created to maintain the status of CPU variables.
- **unstructured_parallel_for**: An unstructured_parallel_for loop is used to run a loop for nonregular indices, e.g., accessing cells along a circle. It first copies an array of indices from host to device and then carries out operations on the cells corresponding to the index array. These index arrays can be reused for subsequent tasks once copied to the device. The same stream is used to copy the index array and to execute unstructured_parallel_for. Therefore, the assumption was synchronization will be implicit between copying and kernel launch. However, a race condition occurred when another thread tried to access the same index array before the copy was completed. The simple solution as of now is to busy-wait in another thread until the first thread completes the copying of the index array.

The first three enhancements paved a way to run the simulation without modifying 100s

of existing tasks with 10s of variables.

7.1.4 Porting Third-Party Library Tasks

An AMT task needs to build the expected input for the third-party library. The input also needs to be on the expected memory space - CPU or GPU. The same rules apply to the output supplied by the library. The library in question here was Hypre. The Uintah task calling Hypre was a legacy CPU task and always prepared input on CPU and called Hypre CPU. The first step was to add some bulletproofing to ensure that Hypre's execution space and memory space match that of the Uintah task. In the next step, the simulation variables and for loops used to prepare the input to Hypre and receive the output were converted to the portable variables/loops using Uintah's intermediate PPL. The task creation and task interfaces were updated to leverage portable task interfaces in Uintah. However, profiling showed a slowdown on GPU because Uintah's Hypre task passed the input variables row by row to Hypre to skip halo cells while passing the variable to Hypre. Hypre called a custom CUDA kernel to read inputs for every row and caused synchronization overheads. The situation was fixed by adding a single portable loop within the Hypre task to gather all required cells (except halo cells) in a contiguous memory location, and then it was passed as a single input that avoided multiple kernel launches within Hypre. The same procedure was repeated for every input and output variable.

7.1.5 Handover to Third-Party Libraries

Third-party libraries pose interoperability challenges when used in an AMT. These difficulties result from each supporting parallelism in potentially different capacities (e.g., MPI-Only vs. MPI + X hybrid parallelism). The AMT developers also need to ensure the MPI messages do not conflict between the AMT and the third-party libraries. Thinking through how to accommodate tasks using third-party libraries is essential for avoiding performance and thread-safety issues. The same problems were encountered while running Hypre within Uintah. Although the Hypre task was ported, Hypre was designed from an MPI-only perspective, and only one thread is expected to call Hypre. As a result, all other threads within Uintah keep busy waiting through the scheduler logic because the simulation algorithm does not have any other task to run in parallel with Hypre. The impact of this busy waiting is minimal if Hypre is run on GPUs execution because the

bulk of the computation takes place on GPUs. However, if the Hypre task is to be run on CPU, Uintah's threading mode conflicts with Hypre, because Hypre spawns its own threads, as seen in Chapter 6. The problem was fixed in the earlier Hypre work [163] and Kokkos::OpenMP scheduler [90]. Upon detecting the Hypre task ready for execution, all threads roll back from the Kokkos::partition_master, and only the main thread calls Hypre. The configuration gives Hypre freedom to use the parallelism model of its choice. Upon return from the Hypre task, partition_master is invoked again to complete the rest of the tasks.

Another challenge was detected for the first time while evaluating the performance of the new enhanced scheduler. The MPI message tags used in Hypre started conflicting with those used by Uintah, but only after crossing 128 GPUs causing occasional application hangs or crashes. Thanks to the dynamic multithreaded out-of-order execution of tasks, the problem never surfaced until now, when a particular combination of task execution logic and the runtime parameters such as the number of ranks, threads, patches, domain size, etc., was hit by the simulation. A simple fix was to add an offset of 10,000 to all MPI tags in Hypre.

7.1.6 Porting More Tasks

Porting of tasks was more of a streamlined process mentioned earlier in [150]. About 30 legacy tasks were ported using the indirect PPL. However, porting these incrementally and trying different combinations of execution spaces for the ported revealed different defects and race conditions mentioned earlier.

7.1.7 Data-Parallel OpenMP CPU Tasks

Once the heterogeneous scheduler was able to execute 100s of tasks smoothly on CPU and GPU both, the creation of std::threads in the scheduler was replaced with Kokkos::partition_master, similar to [90]. Using Kokkos::partition_master replaces the std::thread in task executor with OpenMP partition created by Kokkos and avoids any conflicts with data-parallel OpenMP loops in the tasks. Thus, tasks now can be run on CPU and can call Kokkos::OpenMP::parallel_for without conflicting with the scheduler. The change made the scheduler fully heterogeneous capable of executing portable tasks on three different execution spaces - CPU serial (for legacy tasks), Kokkos OpenMP (data-parallelism

using CPU OpenMP), and Kokkos CUDA (data-parallelism using GPU).

7.2 Strong-Scaling Studies

The results presented in this section used two problems to stress different portions of three individually ported codes uniquely.

1) Uintah's Arches turbulent combustion simulation component;
2) Uintah's standalone linear solver using LLNL's Hypre; and
3) Uintah's standalone reverse Monte-Carlo tracing (RMCRT) radiation model.

These codes are central to both CCMSC boiler simulations and subsequent combustion research. Demonstrations of weak scaling for these codes can be found in past studies [105, 126, 168]. For RMCRT, weak scaling is possible using aggressive mesh refinement to reduce communication requirements.

The first problem, the helium plume benchmark, demonstrates the newly portable interoperability of (1) and (2) on a single-level structured grid. The second problem, the modified Burns and Christon benchmark, demonstrates newly portable interoperability of (1), (2), and (3) on a two-level structured adaptive mesh refinement grid. For both problems, newly ported single source implementations with underlying support for legacy serial loops and Kokkos-based data-parallel loops for Kokkos::OpenMP and Kokkos::CUDA were used. For (2), a modified version of Hypre, which implements recently published techniques [163] for improving GPU-based performance, was used. For both, a modified version of Kokkos, which implements recently published techniques [154] for improving GPU-based performance of Kokkos, was used.

The helium plume problem played a key role in CCMSC efforts for their ability to validate Arches using problems with characteristics representative of a real fire but without introducing the complexities of combustion [168]. The problem used here consists of 125 unique portable loops individually using up to 17 variables with complex interconnectedness. Underlying Kokkos functionality used among loops includes Kokkos::parallel_for, Kokkos::parallel_reduce, and Kokkos::View. A key feature of the problem for validating Uintah's heterogeneous MPI + Kokkos task scheduler is a large number of unique portable loops and variables in flight during execution. The problem has long and complex data dependency sequences generated by tasks (e.g., variables computed on the host, modified

on the device, and later required on the host). Thus, the helium plume is useful to test the robustness of the heterogeneous MPI + Kokkos task scheduler. There are domain decomposition and run configuration dependent multipliers on unique loops not reflected in the counts above.

Uintah's two-level reverse Monte-Carlo ray tracing (RMCRT) radiation model [102] also plays a key role in CCMSC boiler simulations, where radiation is the dominant mode of heat transfer. The problem is a modified version of the Arches' Burns and Christon benchmark problem with a pressure solve using Hypre, and consists of 19 unique portable loops individually using up to 28 variables with complex interconnectedness. The underlying Kokkos functionality used among loops includes Kokkos::parallel_for, Kokkos::parallel_reduce, Kokkos::View, and Kokkos_Random. The problem can validate the new task scheduler by simultaneously stressing interoperability of Arches, Hypre, RMCRT, and adaptive mesh refinement support used by RMCRT. The complex hand-offs between these components (e.g., shared data dependencies) ensure the robustness of the scheduler.

7.2.1 Lassen Runs

Strong-scaling studies were performed on the DOE Lassen system to demonstrate the scalability of Uintah's heterogeneous MPI + Kokkos task scheduler. The current author and John Holmen ran the experiments together on Lassen. This system features two IBM POWER9 processors with 22 cores (4 SMT threads per core) per processor, four Volta-based NVIDIA Tesla V100 GPUs with 5,120 CUDA cores, and 16 GB of HBM2 per GPU, and 256 GB of DDR4 per node. Code was compiled using Nvidia CUDA 10.1.243, gcc/7.3.1, and spectrum-mpi/2020.08.19, with compiler flags "-std=c++11 -O2 -g -Wno-deprecated -Wno-unused-local-typedefs -fPIC -fopenmp." NVCC flags were set to "-arch=sm_70 -expt-extended-lambda -std=c++11." For both problems, these studies explored varying domain decomposition approaches using problems sized to fill the 64 GB per-node memory footprint of HBM2. The helium plume simulation was run on CPUs using 40 single-threaded MPI processes per node to execute loops using 1 core and 1 SMT thread per loop. For the CPU-only runs of the RMCRT problem, simulations were launched using 4 MPI processes per node. Each MPI process spawned 10 task executors (OpenMP partitions) to execute

multiple tasks in parallel across 10 cores. Kokkos::OpenMP was run with only one thread per executor (no hierarchical parallelism). Both simulations were run on GPU using the same configuration as that of RMCRT CPU runs, except this time each rank also used one V100 GPU to execute the GPU tasks. While running with Kokkos::CUDA, all 10 executors share one V100, but use different CUDA streams. The runtime configuration was fixed to 256 CUDA blocks per loop with 256 CUDA threads per block.

Problems were sized to provide each MPI process with at least one patch in all data points shown. Here, a patch refers to the collection of cells executed by a loop. The larger patch sizes result in fewer patches being available to distribute across MPI processes. Data points with fewer than one patch per MPI process were omitted from the figures. Reported per-timestep timings measure the simultaneous execution of all loops across both the host and device in a given timestep. Results have been averaged over seven consecutive timesteps.

Fig. 7.2 shows strong scaling results across DOE Lassen nodes for the helium plume problem on a single-level structured grid. Results for the MPI + Kokkos::OpenMP + Kokkos::CUDA execution were gathered for a problem featuring 512^3 cells for three patch sizes (32^3, 64^3, and 128^3 cells per patch). For all patches sizes, individual patches were combined to a single super patch when passed to Hypre for GPU execution to allow Uintah to make performant use of Hypre, as recently demonstrated in [163]. The helium plume problem is traditionally run using Uintah's MPI-Only task scheduler with one rank per CPU core and without using GPU. The same configuration was used to generate a CPU baseline for three problem sizes - small (256^3), medium (512^3), and large (1024^3). The problem was decomposed into smaller patches of size 32^3 to feed each CPU core. GPU simulations were run with three patch sizes - 32^3, 64^3, 128^3. Using the 128^3 patch size gave the best speed-up of 2.3x to 4.4x over CPU and is shown in Fig. 7.2.

Fig. 7.3 shows strong scaling results across DOE Lassen nodes for the modified Burns and Christon benchmark problem on a two-level structured adaptive mesh refinement grid. The MPI + Kokkos::OpenMP + Kokkos::CUDA results were gathered for the RMCRT problem featuring 512^3 cells on the fine mesh and 128^3 cells on the coarse mesh for three fine mesh patch sizes (32^3, 64^3, and 128^3 cells per fine mesh patch). For all patches sizes, individual patches were combined to a single patch when passed to Hypre for CUDA

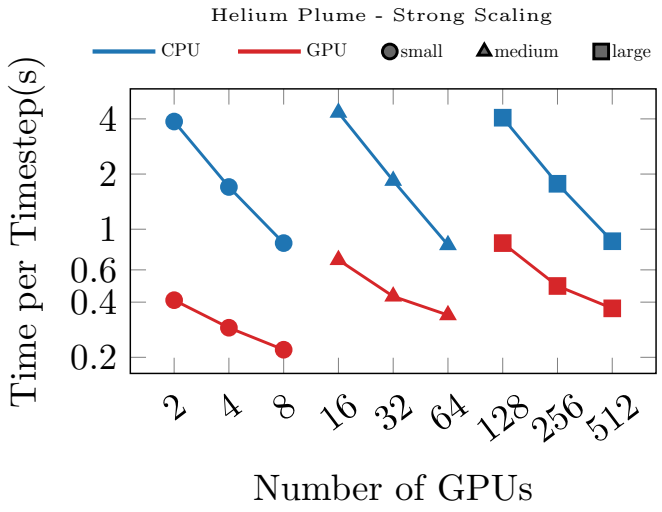


Fig. 7.2: Strong scaling of helium plume benchmark on Laseen with V100 GPUs and POWER9 CPUs.

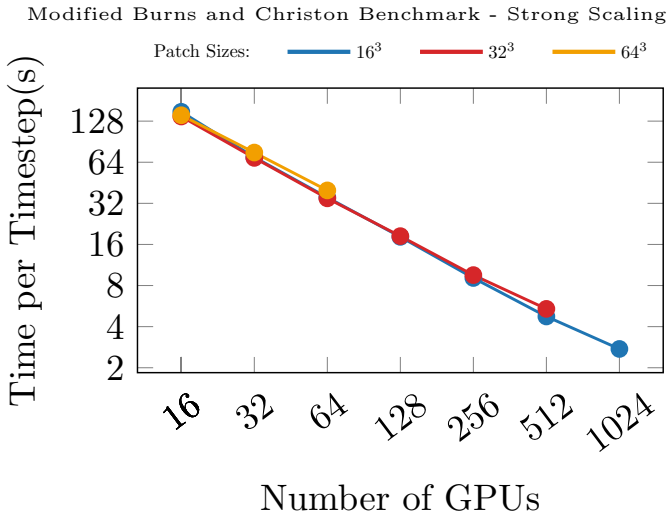


Fig. 7.3: Strong scaling of 512^3 sized modified Burns and Christon benchmark on Laseen with V100 GPUs and POWER9 CPUs.

execution. The merging of patches allows Uintah to make performant use of Hypre, as recently demonstrated in [163]. MPI-Only comparisons are not made here as the global, all-to-all nature of the radiation model used by this problem necessitates MPI + X hybrid parallelism [102]. Instead, the CPU only simulation was run using Kokkos::OpenMP. The GPU runs showed up to 10x speed-ups. The CPU run was done only for validating the performance and was not an exhaustive search of the best CPU runtime configuration. The major focus of the current RMCRT runs is the scalability of the problem on GPUs. For MPI+CUDA comparisons, see related MPI + CUDA and MPI + Kokkos::CUDA comparisons [90, 154] gathered on a single node and the DOE Titan system, respectively.

Results presented in Fig. 7.2 and 7.3 show that good strong scaling to 1,024 NVIDIA V100 GPUs and 512 IBM POWER9 processors is possible to leverage MPI + Kokkos::OpenMP + Kokkos::CUDA. The results are encouraging as they show a potential for this heterogeneous MPI + Kokkos task scheduler to reduce the gap between development time and our ability to run on heterogeneous systems requiring other underlying programming models. This capability is advantageous for expediting Uintah's ability to support forthcoming exascale systems such as the Intel-based DOE Aurora and AMD-based DOE Frontier.

Results presented in Fig. 7.3 show that an asynchronous many-task model can achieve good strong scaling for a compute-dominant problem across on multisocket, multidevice nodes. Results presented in Fig. 7.2 show that for a communication-dominant problem, it can be difficult to achieve performance across multisocket, multidevice nodes using an asynchronous many-task model. The reduced speed-ups in the communication dominant problem are not unexpected given the additional overheads (i.e., for data movement) incurred between the host and device on such nodes. As shown in Fig. 7.2, offloading fewer, yet larger, patches to the device can be used to improve node-level performance at the expense of reductions in strong scaling efficiency for communication-dominant problems. These results suggest that care must be taken when running an asynchronous many-task model on multisocket, multidevice nodes. Although performance improvements are achievable, performance reductions are also possible when overdecomposing a simulation domain, as observed for the helium plume problem running with the 32^3 patch size.

Comparing CPU (MPI-Only) results in Fig. 7.2 to GPU (MPI + Kokkos::CUDA) results in Fig. 7.2 shows that it is possible for Uintah application developers to improve node-

level performance with relative ease using this scheduler and the accompanying portable abstractions [90] to port legacy serial loops to OpenMP and CUDA via Kokkos. The results are encouraging as the wholesale refactoring of Arches loops to leverage PPL has been largely naive with ample opportunity to improve performance. The development of this scheduler and PPL has spared application developers having to, for example, learn CUDA and write individual kernels for the triple-digit files comprising this simulation component. One of the biggest wins for the PPL + heterogeneous scheduler approach is faster application development expediting scientific efforts.

7.2.2 Frontera Runs

The same benchmarks were run on TACC Frontera [176] using three problem sizes, 256^3, 512^3, and 1024^3. Frontera features 8,368 nodes, two sockets per node, and 28 Intel Cascade Lake cores per socket. Each node contains 192GB of DDR4, and the achievable memory bandwidth is approximately 210 GB/s using the stream benchmark [59]. Theoretical peak node performance is 4.8 TFlops/s for double-precision computations. The code was compiled using the Intel MPI and Intel compilers v19.0.4 with compiler flags "-fp-model precise -xCORE-AVX512 -qopt-zmm-usage=high -g -fPIC -mkl -O2 -fopenmp -std=c++11."

The helium plume benchmark contains a large number of small tasks. Running Uintah with multiple threads performs slower for such problems due to contentions for locks within Uintah and MPI. The single-threaded MPI Only version does not suffer through the thread contentions. Hence, the helium plume problem was run using the MPI Only version with one MPI rank per core, and the grid was decomposed into patches of the size 16^3. Fig. 7.4 shows excellent strong scaling for the helium plume benchmark on Frontera, except in the last hop when communication starts dominating computation. If one Cascade Lake node of Frontera is to be compared with one V100 GPU of Lassen, then six data points are common between Fig. 7.2 and Fig. 7.4, i.e., "Number of GPU/Nodes" 4, 8, 32, 64, 256, and 512 are present in both figures. The comparison shows Lassen performs 1.5x to 2x faster than Frontera for small- and medium-sized problems. The gap in the performance reduces for the large problem as the problem becomes more communication dominant and the host-device transfer becomes a bottleneck on Lassen.

Chapter 6 has already demonstrated how the combination of multithreaded RMCRT

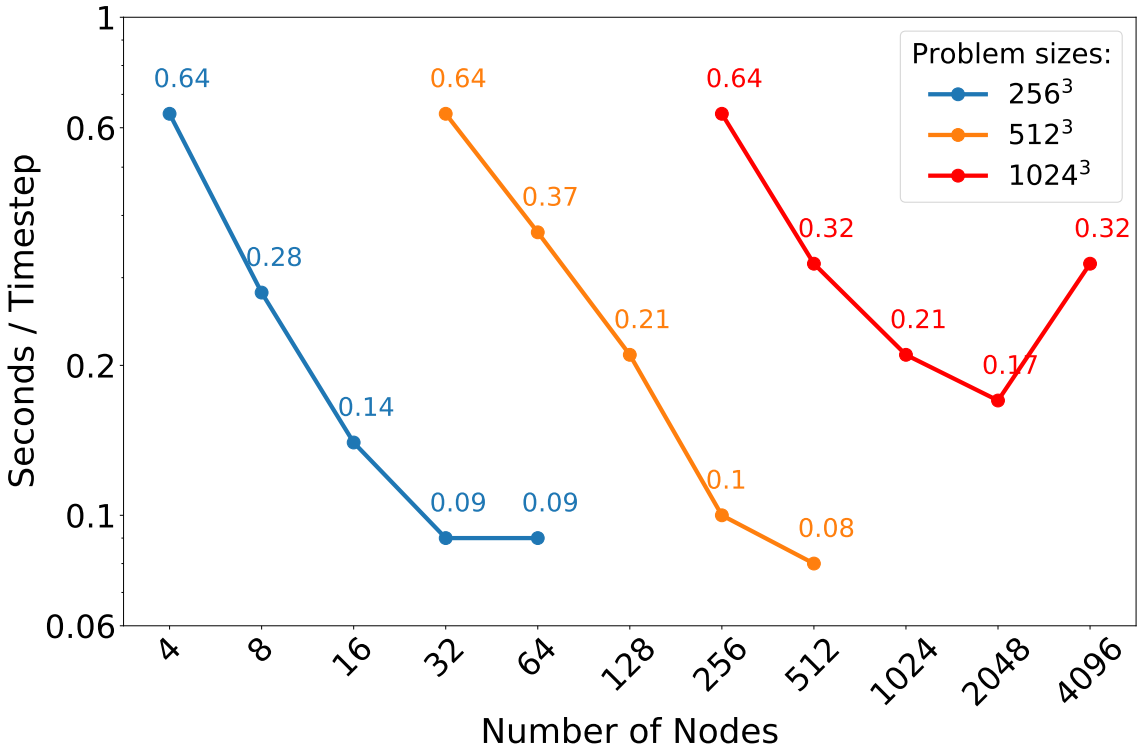


Fig. 7.4: Strong scaling of the helium plume benchmark on Frontera with Intel Cascade Lake CPUs.

and Hypre-endpoints works faster than the respective MPI Only versions. Hence, the modified Burns and Christon benchmark, the combination of Arches, RMCRT, and Hypre, was run using multithreaded KokkosOpenMP scheduler using 56 threads grouped among two or four teams. The grid was decomposed into patches of the size 32^3. Hypre used the endpoints version and spawned 14 endpoints with four threads per endpoint during the initial hops of scaling, and used eight endpoints with seven threads per endpoint during the last hops for each problem to ensure that at least one patch is available for every endpoint. The modified Burns and Christon benchmark also scales well, as shown in Fig. 7.5. Near perfect scaling is visible until the last hop for each problem size. Again, timings of 512^3 sized problem on Frontera (Fig. 7.5) can be compared with the GPU timings on Lassen (Fig. 7.3) for the number of nodes/GPUs ranging from 32 to 1024. Lassen performs 2x to 3x faster than Frontera. Interestingly, both systems have approximately same per unit power consumption. The power consumption to use one Nvidia V100 GPU of the Lassen cluster is approximately 395W – 300W for one Nvidia V100 [18] + 95W for half IBM Power9 CPU

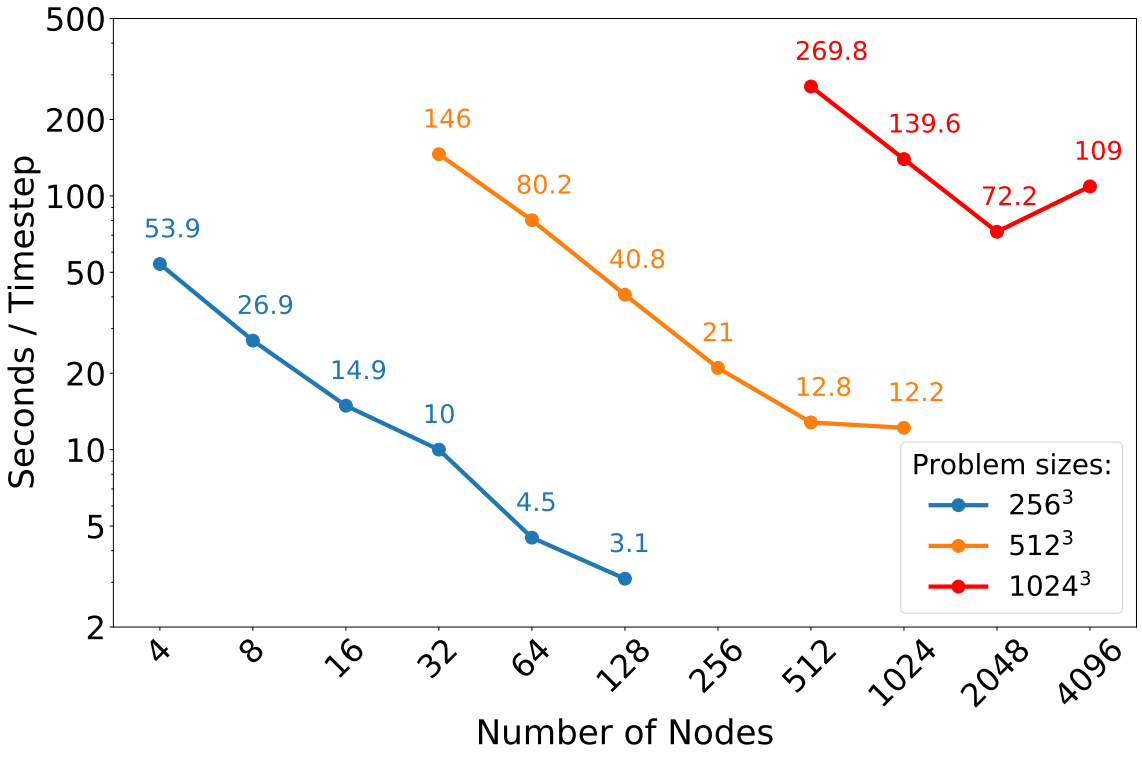


Fig. 7.5: Strong scaling of the modified Burns and Christon benchmark on Frontera with Intel Cascade Lake CPUs.

because there are two GPUs per CPU which consumes 190W [19]. One the other hand, one Frontera node consumes around 420W with two CPUs per node drawing 210W power per CPU [20]. The approximate retail price for half IBM Power 9 + one Nvidia V100 GPU is $8,000 to $12,000 [21], but the same for two Intel Cascade Lake CPUs will be around $20,000 [20]. Thus, Nvidia V100 becomes much more cost effective solution considering performance, power consumption and cost.

The performance gap between Lassen and Frontera can be attributed to two main reasons. The peak theoretical performance of V100 GPUs on Lassen is 7.8 TFlops/s for double-precision, whereas that of Cascade Lake CPUs on Frontera is 4.8 TFlops/s, which makes a difference of 1.6x consistent with the comparison of helium plume. The second important factor is the difference in the flop to memory bandwidth ratio. Flop to bandwidth ratio for V100 is 7.8 TFlops/s / 900 GB/s = 8.6 Flops/byte, while that for Cascade Lake is 4.8 TFlops/s / 210 GB/s (approximate) = 22.8 Flops/byte, approximately. This difference between flop to memory bandwidth ratio makes it harder for bandwidth-hungry applications to perform on Frontera. Profiling using Intel Vtune shows the memory bandwidth to be the main bottleneck for the top 10 longest-running kernels on Frontera.

7.2.3 Summit Runs

The natural extension of the Lassen experiments is to run on the Summit supercomputer with its peak performance of 200 Pflops, which is an important step towards the eventual exascale runs. The Summit supercomputer contains 4,608 nodes. Each node features two IBM Power9 CPUs with 22 cores per CPU, six Nvidia V100 GPUs with 16 GB HBM2 memory per GPU, and 512 GB CPU memory.

The Summit runs were carried out in collaboration with John Holmen. While the runs are still in progress for the modified Burns and Christon benchmark, some of the early results for the helium plume benchmark are presented in Fig. 7.6. The same source code run on the Lassen was run on the Summit with the same compiler flags, and one MPI rank was spawned for each GPU. Three problems of sizes small (768 x 512 x 512), medium (1536 x 1024 x 1024), and large (3072 x 2048 x 2048) were run using two patch sizes 64^3 and 128^3. Fig. 7.6 shows very good (although not perfect) strong scaling and weak scaling up to 6,144 GPUs. Using 128^3 patches performs 2x to 3x faster than using 64^3 patches, but can not scale

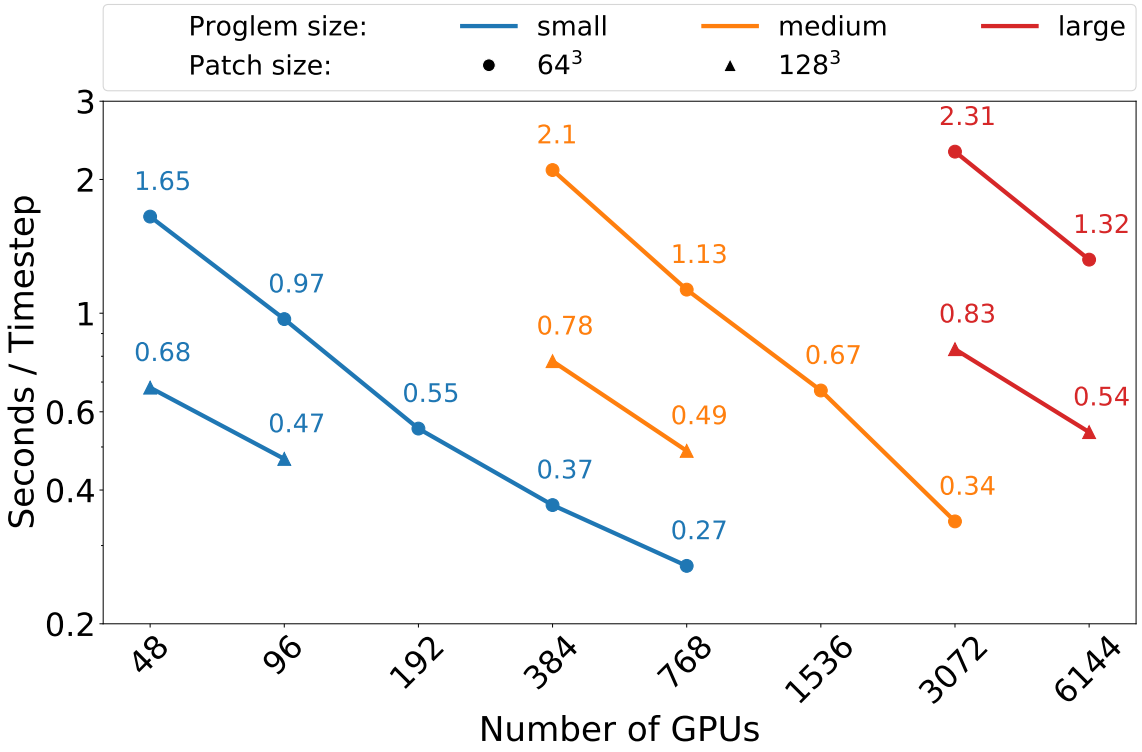


Fig. 7.6: Strong scaling of the helium plume benchmark on Summit with IBM Power9 CPUs and Nvidia V100 GPUs.

as much because the bigger patch size reduces the number of patches that can be assigned to GPUs.

The biggest win for Lassen, Summit and Frontera runs is "Performance Portability," and improving scientific productivity. The scaling studies on Frontera and Summit used the same code run on Lassen. All the tasks written using Kokkos-based were got successfully compiled and executed on Frontera without any code changes. The only change made for Frontera runs was to configure the MPI endpoints version of Hypre and import corresponding code within Uintah. Because the endpoint code is still experimental, it is not committed to the main Uintah branch and is imported as needed. The experiments demonstrate the versatility of the portable solution, which works and scales well with MPI only, Kokkos-OpenMP, and Kokkos-CUDA. Running hundreds of portable tasks on Lassen and Frontera prove portability, and strong scaling studies on both systems demonstrate performance to achieve the biggest win for Uintah AMT, "Performance Portability."

7.3 Related Approaches and Foreseeable Challenges

The approach presented here is a starting point for achieving portable heterogeneous MPI + PPL task scheduling in an asynchronous many-task runtime system. Uintah's indirect adoption of Kokkos is one of many options to achieve portability across the complex nodes anticipated in exascale systems. For example, Hypre has adopted Kokkos directly, whereas other codes such as Hedgehog [32] have built their own portable designs for performance. A review of performance portable programming models for productive exascale computing can be found in [112].

Foreseeable challenges include understanding how to: 1) coordinate host-device data movement in a portable manner; 2) efficiently coordinate host-device data movement in the context of an asynchronous many-task model; 3) coordinate MPI in a simpler manner (e.g., explore *MPI_THREAD_FUNNELED*); 4) execute tasks based on parallel third-party libraries using a performance portability layer; 5) automate the decision of when to use the host or device for portable tasks (e.g., make an informed decision about the target execution space); and 6) vectorize efficiently.

7.4 Conclusions and Future Work

This work has helped improve Uintah's portability to complex heterogeneous systems. Specifically, it has shown an approach for heterogeneous MPI + PPL task scheduling to help prepare an asynchronous many-task runtime system for the diverse heterogeneous systems accompanying exascale computing. This approach combines three individual solutions offering promise for making efficient use of the complex nodes anticipated in these systems: 1) asynchronous many-task runtime systems; 2) MPI + X hybrid parallelism approaches; and 3) performance portability layers. This integration is done with additional consideration for parallel third-party libraries facing similar challenges related to (2) and (3).

This approach has been demonstrated with the help of a heterogeneous MPI + Kokkos task scheduler implemented in the Uintah Computational Framework, an asynchronous many-task runtime system, with additional consideration for Hypre, a parallel third-party library. Kokkos capabilities have been shown for two challenging problems using this scheduler and the accompanying portable abstractions [90] to execute workloads representative of typical Uintah applications across multisocket, multidevice nodes while making heterogeneous use of OpenMP and CUDA via Kokkos with a single-source implementation. Performance improvements up to 4.4x to 10x have been achieved on GPUs with this scheduler and the accompanying portable abstractions [90] to port a previously MPI-Only problem to Kokkos::OpenMP and Kokkos::CUDA and extract better performance out of the complex heterogeneous node. At scale, good strong scaling to 6,144 NVIDIA V100 GPUs and 3,072 IBM POWER9 processors has been achieved utilizing MPI + Kokkos::OpenMP + Kokkos::CUDA.

The portability and performance improvements shown here offer encouragement as we prepare Uintah for the diverse heterogeneous systems accompanying exascale computing. The next steps include extending Uintah's intermediate portability layer [90] to support Kokkos' default host and device spaces to reduce programming efforts in infrastructure development. For Uintah's Aurora Early Science Program efforts, this research will improve the speed with which we can support Kokkos::OpenMPTarget for running on the Intel GPUs anticipated in the DOE Aurora system. For Uintah's emphasis on maintaining broad support for major HPC systems, the new approach will allow the speed with which we can

implement, refine, and extend Uintah's more formal support for future major HPC systems. As a part of these efforts, emphasis will be placed on generalizing CUDA-specific code used in Uintah's heterogeneous MPI + Kokkos task scheduler to achieve more portable heterogeneous MPI + Kokkos task scheduling.

CHAPTER 8

RELATED WORK

The idea of AMTs goes back to 1960s [119]. "The Manchester project" developed a new dataflow processor to run dataflow-based parallel programs [82]. Another example dataflow-based processor is "Hughes Data Flow Multiprocessor" [186]. Several programming languages in the late 1980s and early 1990s implemented message-driven architecture to exploit intranode and internode parallelism, e.g., HAL [97], ABCL [193], Concurrent Smalltalk [54], pC++ [42], Mentat [79]. Silc [172] surveys the early techniques of dataflow-driven scheduling and parallel execution. Sarkar [166] defined language- and architecture-independent graphical representation for parallel programs, and created dataflow-based models for scheduling tasks. Charm++ [114] tried to strike a balance between two parallelization strategies: 1) making users fully responsible with minimal support from the system such as communication routines, and 2) users writing sequential code which is parallelized by the system [116]. Charm++ abstractions can be used to write a large number of C++ objects, which are distributed among multiple processors, communicate using user-provided asynchronous methods, and can be migrated from one processor to another without user involvement. Thus, Charm++ provides processor-independent asynchronous programming, asynchronous message-driven parallel execution, automatic communication/computation overlap, load balancing, and checkpointing-based resilience. Thus, the dataflow-based parallelization strategy evolved from processor-based instruction-level parallelism to compiler-based partitioning to user-defined task-based runtime systems. Several other successful AMTs apart from Uintah and Charm++ work on the same principles, e.g., HPX [113], Legion [33], PaRSEC [44], and StarPU [30]. These runtime systems support heterogeneous architectures involving CPU, GPU, accelerators, and Xeon Phi cores. In Charm++ [115], users create a set of interacting data objects called "chares" (roughly analogous to Uintah tasks). Charm++ provides a framework

for distributed processing that users need to program to achieve concurrency of shared data resources through proper message structuring. Legion [33] executes concurrent tasks based on the user-provided "logical regions" of data. StarPU [30] provides a low-level multitask scheduling framework portable across heterogeneous architectures. High-level reviews and comparisons between contemporary AMTs can be found in [105,119,124,150] The study [34] evaluates programmability, performance, and mutability of the three best-of-class AMTs, Uintah, Legion, and Charm++, through an extremely thorough qualitative and quantitative comparison.

The following sections discuss related work corresponding to each of the contributions of this dissertation.

8.1 Heterogeneous Execution and Portability of Tasks

Multiple AMTs have demonstrated capabilities to run on heterogeneous GPU platforms along with CPUs. These capabilities include GPU support, heterogeneous task scheduling, and portability infrastructure up to varying extents. Charm++ and PaRSEC can run tasks on GPUs [86,156,185]. However, they put the burden of writing platform-specific tasks on users, and a CPU task cannot be run on GPU without explicit rewriting, and vice versa. Legion provides the programming language called Regent, which also allows writing portable code [61, 181]. Recently, HPX demonstrated Kokkos integration to achieve performance portability [53]. Yasar [190] recently used portable task-based execution to improve graph problem performance. He implemented a single heterogeneous task queue to store the portable tasks sorted by the task size, where GPUs pick up tasks from the start of the queue (thus, run heavier tasks), whereas CPUs are assigned tasks from the end of the queue (thus, execute lighter tasks).

Heterogeneous scheduling in Uintah has evolved over the past decade. Uintah introduced MPI+X parallelism with MPI + PThreads task scheduler [138] to run NSF Kraken and DOE Jaguar systems. A MPI + PThreads + CUDA task scheduler [103, 104, 139, 151, 152, 155] was built to execute raw CUDA tasks and CPU tasks. The Kokkos PPL and scheduling of portable tasks are implemented as part of the portability drive, incrementally. An MPI + Kokkos::OpenMP task scheduler [88, 141] allowed scheduling of legacy tasks and data-parallel (OpenMP) tasks on manycore systems. The next step was to build

heterogeneous MPI + PThreads + Kokkos::CUDA task scheduler [89, 90, 150, 153, 154, 179], which built a foundation for scheduling of legacy and Kokkos-CUDA tasks. This research, conducted in collaberation with John Holmen [94], enhanced the MPI + PThreads + Kokkos::CUDA task scheduler and also built MPI + Kokkos::OpenMP + Kokkos::CUDA task scheduler to a achieve production-ready, fully heterogeneous scheduler capable of executing hundreds of legacy, Kokkos::OpenMP and Kokkos::CUDA tasks on thousands of GPUs. Although the PPL and Kokkos enabled of writing and executing "portable tasks" on heterogeneous platforms, the task scheduler logic which executes these portable tasks uses some raw CUDA code. As a result, the Uintah runtime itself is not portable, and making it so should be the immediate goal to enable Aurora runs.

8.2 Resiliency

Research has increasingly focused on the area of resiliency to meet the exascale challenge. Faults can be handled at a system level through either Hardware-Based Fault Tolerance (HBFT) or through Systems Software-Based Fault Tolerance (SBFT) independent of applications [100].

Early works regarding fault tolerance, including dynamic MPI programs with checkpointing and resilient versions of MPI, are described in [26, 43], with relatively recent summaries found in [45, 65]. Cappello [47] has summarized recent developments in resiliency that targets exascale. Some key checkpointing approaches include BLCR checkpointing [83], adaptive checkpointing [170], hybrid checkpointing [187], data aggregation [109], and incremental checkpointing [46]. Hussain [107] carried out a theoretical study explaining how nonidentical failure distribution among nodes can help achieve different levels of replication and deliver faster performance.

Checkpointing and restarting are widely used but are inherently time-consuming. Dauwe [55] analyzes different resiliency techniques such as rollback recovery with checkpointing and restarting, or nonblocking multiple levels of checkpointing [167], message logging, and full or partial redundancy along with their performance characteristics.

8.2.1 Algorithm-Based Fault Tolerance

Algorithm-Based Fault Tolerance (ABFT) [99, 100] uses application-specific algorithms to construct more efficient resilience techniques. Several ABFT techniques have been

developed for different algorithms, e.g., ABFT for matrix-matrix multiplications-based programs [49,50], which have been further extended to linear algebra routines/benchmarks [57,58]. The GVR library [51,73] can maintain and present a global view of distributed arrays along with the versioning of these arrays. GVR provides programmers with the capability to implement ABFT by accessing remote coarse mesh data structures. Depending on the algorithm and the level of its failure, an appropriate version of an array can be retrieved from GVR to implement the desired recovery routine [66,99]. Dubey handles soft failures such as errors due to bit flips at different levels, such as cell or box level, using ABFT [66,67]. She describes two approaches to handle hard failures such as core or node failure - one based on ABFT and the other on retrieving missing data from GVR - but she chooses the latter [66,67]. The solution adopted here is the less expensive but more complex approach that Dubey [67] described in which the solution is reconstructed on the missing mesh patches due to a node failure. The accuracy of the data must be ensured while rebuilding the solution on fine mesh patches using coarse data values. Such issues arise even in standard AMR calculations [39] but are not often referenced. Rebuilding fine mesh data using coarse mesh data may result in reduced accuracy on the fine mesh patch that persists as the solution evolves. The use of high-order interpolation schemes to regenerate these data helps address the accuracy issue. These schemes, however, may violate important physical properties of the solution, such as the need for positivity, and may introduce spurious maxima "overshoots" or spurious minima "undershoots." For this reason, it is important to consider using interpolation approaches that respect any underlying desired physical bounds on the solution.

8.2.2 Fault Detection Mechanisms for a Node Failure

The ABFT approach is feasible only if a reliable fault detection mechanism exists. Many research codes are available, but, to the best of our knowledge, only two proposed/experimental standards are in place:

- Fault-Tolerant MPI (FT-MPI) [70], and
- User Level Failure Mitigation (ULFM) [41].

Both FT MPI and ULFM provide user-level APIs that can be called from a custom error handler. These APIs can be used to detect failed ranks and form a consistent view of the

MPI world across all surviving ranks. Both approaches can detect and possibly recover up to $n-1$ failed processes out of n processes. In both standards, the responsibility for recovery rests with application developers.

Many studies over the last several years have attempted to address resilience. An early example was the novel architecture of Starfish [26], which was based on combining group communications technology and checkpointing. Varma [184] developed a protocol to detect node failures and create a consistent view of active nodes among MPI ranks. Morris et al. [161] implemented a fault-tolerant ULFM and ABFT-based PDE solver. However, they used a server-client model in which servers, immune to failure, maintain the "state" of the system and hand over computational tasks to clients, which are allowed to fail. This design ensures that a client node failure hampers only data and not the state but restricts failure to only parts of the algorithm that are executed on the client nodes. The resiliency approach in Uintah, on the other hand, does not use a server-client model. Each rank maintains its own state and can fail at any given time. Hence, a resilient version of Uintah needs to recreate tasks and rebuild patches belonging to "lost" ranks.

8.3 Asynchronous Scheduling and AMT on Sunway

The Sunway TaihuLight was the world's fastest supercomputer at the time and currently holds the fourth position in the June 2021 version of the top-500 list [12]. Many real-word applications were able to run at substantial fractions of peak performance on Sunway, e.g., the three applications selected as Gordon Bell Award finalists in 2016 [160, 191, 197]. However, these performance levels were obtained through extensive and intensive rewriting and tuning of the code at a level that may not be possible or affordable for general application codes. On the other hand, as discussed earlier in Chapter 1, the AMT runtimes can easily overlap computation and communication, take advantage of deep memory hierarchies, offload kernels asynchronously – all with minimal development efforts from the application developers – and allow running efficiently on the novel architectures. The decoupling of infrastructure and the simulation code in AMTs allows the independent development of both layers. The methodology can save a huge amount of porting efforts not only for the Sunway architecture, but also for the next-generation supercomputers and different unexplored architectures. Hence, it becomes essential to study the adoption

techniques for AMTs on Sunway. Uintah is the first AMT runtime system ported to the Sunway TaihuLight supercomputer.

8.4 Portability Frameworks and Vectorization

Vectorization has been studied from multiple perspectives: tools to identify vectorization opportunities [87], portability frameworks using intermediate representations (IR) [148], data-parallel programming models [129], and data layout transformations [85]. Existing methods for improving vectorization include compiler directives, framework-based methods, tools to assist compilers [87], and language extensions [129]. Compilers provide directives that help autovectorization, e.g., the Intel compiler's `#pragma vector` directive instructs the compiler to override efficiency heuristics. Intel's `#pragma simd` can be used to force vectorization (although it has been deprecated in the 2018 version). `#pragma ivdep` instructs the compiler to ignore assumed loop dependencies. OpenMP provides `#pragma omp simd`, which is similar to `#pragma simd`. `#pragma omp target` constructs allow writing portable code and offloading data-parallel for loops to GPUs using OpenMP. OpenMP target constructs can also be combined with `#pragma omp simd` to achieve portability and autovectorization. Autovectorization guided by these directives produces efficient code in most cases. However, sometimes complex control structures in a loop prevent autovectorization even after specifying these directives – as shown in Section 5.3.1. The LLVM community is gradually developing more advanced vectorization capabilities such as outer loop vectorization [180]. OpenCL [145], a portable parallel programming standard, provides vector data types on all the supported devices. The maximum length of a vector data type in OpenCL is limited to 16, which may be problematic for architectures with larger PVLs. On the other hand, the LVL implemented in this work can be passed as a template argument and offers more flexibility to users. In addition to Kokkos, there are other recently developed performance portable libraries such as RAJA [96], and OCCA [136]. The SIMD vectorization support in RAJA is limited to using the execution policy `RAJA::simd_exec`, which adds `#pragma omp simd` to the code and relies on compiler autovectorization [110]. OCCA also provides hints to enable autovectorization but lacks any explicit SIMD support at present.

Multiple implementations of a SIMD primitive for CPUs such as the Vc vectorization

library [122], the Unified Multi/Many-Core Environment (UME) framework [118], and the Generic SIMD Library [188] enable an explicit vectorization using architecture-specific SIMD intrinsics and operator overloading. KokkosKernels [121] is a library that implements computational kernels for linear algebra and graph operations using Kokkos. KokkosKernels uses a SIMD data type for its batched linear algebra kernels [120]. Embedded ensemble propagation [157] using the Stokhos package in Trilinos [158] for uncertainty quantification uses another version of SIMD primitives that allows flexible vector lengths. Furthermore, Phipps [157] addressed the portability of this "ensemble type" to SIMT architectures. Pai [148] addressed SIMT portability using Intermediate Representations (IR).

Although all these efforts are successful, they do not yet provide the full range of portability shown in this work.

8.5 Modernization Hypre and MPI Endpoints

The Hypre linear solver library was designed at Lawrence Livermore National Laboratory in the 1990s with the distributed single-threaded design and was scaled up to hundreds of thousands of cores [31, 71]. Over the past decades, Hypre was modernized to use OpenMP and CUDA [31, 75], and succeeded in getting speed-ups on modern architectures, including GPUs. However, the modernization attempt did not yield expected performance improvements in some cases, as shown in Chapter 6.

This work used the combination of MPI endpoints [62] and OpenMP to improve Hypre performance. The current MPI standard assigns an MPI rank to a process, and all threads within a process share the same rank. The proposed endpoints approach allows assigning an MPI rank to an endpoint (EP), where an EP can be a thread, a team of threads, or a process [62]. Although the endpoints proposal did not make it through the MPI standard 4.0, there are few experimental implementations of MPI endpoints [63, 77, 175, 189, 194]. However, all these focus on the efficient implementation of endpoints rather than the application, and evaluate endpoints using standalone kernels or standard benchmarks. This research is the first one to use endpoints in a real-life application, Hypre linear solver and Uintah, and demonstrate its benefits.

CHAPTER 9

CONCLUSION AND FUTURE WORK

The exascale challenges noted in Section 1.1 are spread across an extremely wide range of technological areas, including power consumption, memory technologies, data movement, exploiting massive parallelism, system softwares, algorithms, resiliency, productivity, programming systems, heterogeneous usage, and efficient scheduling of algorithms. As stated in Section 1.2, previous work made significant progress toward the exascale readiness of Uintah; however, many challenges remained: resiliency infrastructure for AMT, portable infrastructure for effective vectorization, more efficient algorithms, and programming models to exploit the emerging hardware, bridging some gaps in the heterogeneous infrastructure to fully support asynchronous execution of hundreds of portable heterogeneous tasks, integration of third-party libraries, etc. Also, solving these problems individually is not enough to achieve performance on the upcoming systems. These individual solutions also need to be integrated together under a single umbrella.

This dissertation aimed to help ready Uintah for exascale and provide potential solutions to help solve five exascale challenges.

1) Programming systems: More expressive programming models are needed to simplify the developer efforts and benefit from the locality and multimillion-way (or perhaps billion-way) parallelism.

2) Data management: The system needs to effectively manage the explosively increasing amount and complexity of the data generated by experiments and simulations.

3) Exascale algorithms: Refactoring of legacy codes is needed to run them efficiently on massively parallelism architectures. Such refactoring may involve leveraging multi-core architectures, communication-avoiding algorithms, synchronization-minimizing algorithms, adaptive load balancing, efficient scheduling, memory management for

heterogeneity and scale, energy-efficient algorithms, etc.

4) Resilience and correctness: Exascale supercomputers and those that follow may have more faults than the current generation machines, and both the machines and applications need the ability to handle and recover from the increased number of faults to maintain accuracy.

5) Scientific productivity: New tools are needed to improve the productivity of the scientists using the exascale machines.

The goal was achieved by innovating new solutions required to run AMTs successfully on the emerging architectures. Another success of this research lies in integrating the individual solutions to different exascale challenges spanning a breadth of the technological areas within a single AMT system in a smooth, conflict-free, and performant manner, just like the multiple pieces of a jigsaw puzzle fitting together to form a single image. The work modified three software systems: the Uintah AMT, Hypre linear solver, and Kokkos. Performance evaluations were carried out on diverse platforms, including the Sunway Taihulight supercomputer (Sunway processor), DOE Theta (Intel KNL), Bebop cluster (Intel KNL), Lassen cluster (Nvidia V100 GPUs), Astra cluster (Cavium ThunderX2 based on ARMv8.1), and Quartz cluster (Intel Broadwell CPUs). Results were demonstrated using a wide range of simulation problems of varying difficulties, size, and scope: Burgers' equation, matrix-matrix multiplication, sparse matrix-vector multiplication, two-dimensional convolution, char oxidation simulation, solving Laplace equations, helium plume problem, and RMCRT benchmark (modified Burns and Christon benchmark). The new scheduler developments (Chapter 7) made it possible to run the helium-plume and modified Burns and Christon benchmarks on Lassen. These runs also utilized the optimized version of Hypre-CUDA (Section 6.3). The same portable code (with the exception of using the Hypre-EP version instead of Hypre-CUDA) is being successfully run on TACC Frontier, which good portability considering that zero lines of code had to be rewritten. Vectorizing complex tasks such as RMCRT in a portable manner would become a priority in the near future, considering the vector units on upcoming Intel Ponte Vecchio GPUs. The portable SIMD primitive has created a foundation for vectorizing such kernels, which may prove very hard to vectorize otherwise. Efforts to run a variety of problems on diverse

systems by tying together different solutions such as portability, vectorization, efficient threading models, heterogeneous/asynchronous scheduling, etc., converge toward a single goal – running on the Aurora supercomputer. Combined with the earlier works by Bradley Peterson and John Holmen, this research can serve as a case study to run AMTs on the next-generation architectures and supercomputers. The contributions are summarized in the subsequent sections, followed by the lessons learned and future work.

9.1 Resiliency

Although the latest DOE report expects resiliency not to be a problem for application developers at exascale, it does expect resiliency concerns to emerge in the post-exascale systems [36]. Chapter 3 provides an approach to handle node failures. The new component uses "Algorithm-Based Fault Tolerance" and performs up to 10x faster than the traditional checkpointing-and-restart. The chapter shows a scheme to detect multiple node failures occurring simultaneously, recover and regroup from such failures, redistribute the orphan tasks, recover the lost data using interpolation, and continue the normal simulation. All these activities are carried out in the form of usual AMT tasks, which minimizes the wait time, and the survivor ranks can keep executing any ready tasks in parallel. Thus, this component tackles the resiliency challenge (Section 1.1) and lays a foundation for future fault tolerance within Uintah.

9.2 Asynchronous Scheduler and Sunway Runs

Chapter 4 shows how Uintah was ported to the Sunway Taihulight supercomputer and how the new asynchronous task scheduler improved performance. Such asynchronous scheduling of tasks is going to be a key for the upcoming heterogeneous nodes with components having different performance characteristics. The experience of porting also showed how easily the AMT infrastructure could be ported to unusual architectures, but at the same time, highlighted the need for having portable application code. Uintah is the first AMT to run on Sunway. Not much is known at this time about the performance characteristics of Intel GPUs, and capabilities/limitations of the programming models on the Aurora supercomputer. Hence, having the multiple options for task scheduling will reduce the eventual porting time and effort. The complex heterogeneous task scheduler is more complex than the asynchronous scheduler and uses raw CUDA code which has

to be ported either to Kokkos or OpenMP/SYCL supported by Intel. On the other hand, the asynchronous scheduler is built using std::threads (which can be trivially mapped to OpenMP) and does not use any CUDA calls. The downside of the asynchronous is that it does not handle data movement as of now. The two schedulers provide two different paths for task scheduling at exascale – 1) improve the asynchronous scheduler to carry out host-device data transfer, or 2) port the heterogeneous scheduler. The shortest path among the two can be chosen once more details of the Aurora architecture and programming models are available. The scheduling approach can help answer the exascale challenge "Exascale algorithms."

9.3 Portable SIMD Primitive

Efficient vectorization is key to performance on systems such as Fugaku. It is not yet known whether the Intel GPUs will support vector instructions, but if they do, then SIMD support will become of paramount importance to run AMT on the Aurora supercomputer. Earlier, the vectorization support within Kokkos was limited to compiler directives, which may not always lead to efficient vectorization. The SIMD primitives can help in such cases, but the lack of a CUDA back end makes code nonportable and defeats the whole purpose of using Kokkos. Chapter 5 covers the development of a new portable SIMD primitive in Kokkos, which is the first such primitive that has a CUDA back end. Performance of the new primitive was evaluated on multiple platforms using different examples. The portable code written using the new SIMD primitive gave near-ideal speed-ups through efficient vectorization and without any overhead. It also helped the compiler to generate better code and reduced L1 cache misses, which resulted in superlinear speed-ups for two test problems. The primitive contribute solving three exascale challenges: "Programming systems," "Scientific productivity," and "Exascale algorithms" from Section 1.1.

9.4 MPI Endpoint Based Threading Model

Chapter 6 serves as a template to upgrade legacy codes for modern architectures. The new threading model was built using MPI endpoints, where each thread (or a team of threads) acts as an MPI rank, was able to minimize the synchronization overhead within the Hypre linear solver. A customized version of MPICH used to improve Hypre performance is built with fine-grained locks and accepts MPI hints that can help leverage parallelism

in on-node networking resources. The combination of the new threading model and new MPICH version improved performance up to 2.7x on 256 nodes of Intel KNL. Hypre is the first application to use MPI endpoints at full scale is another potential candidate to address "Exascale algorithms" and "Programming systems."

9.5 Heterogeneous Scheduling

This work, in collaboration with John Holmen, enhanced the existing heterogeneous scheduler and infrastructure to: 1) support read-write dependencies between portable tasks; 2) avoid excessive synchronizations while gathering halo cells; 3) port tens of tasks to use PPL including the third-party library tasks; 4) improve handoffs between Uintah and third-party libraries including MPI conflicts; and 5) fix race conditions. This incremental work plugged the last few gaps, which prevented the full-scale runs using PPL and heterogeneous task scheduler. Uintah became the first AMT to have run hundreds of portable tasks on 1024 GPUs and 2048 CPUs. The enhanced infrastructure helps to solve "Exascale algorithms," "Programming systems," "Scientific productivity," and "Data management/movement." The successful simulation runs on Lassen shown in Chapter 7 mark a big milestone for Uintah and all those who have worked with the same vision for the past few years. These runs serve as a starting point to the planned runs on DOE Summit, which will be a precursor to eventual exascale runs.

9.6 Lessons Learned

Some of the lessons learned during this research are listed below:

- Experiments on Sunway show Performance Portability is a key to avoid excessive code rewriting. Maybe it is time to include "parallel_for" like construct in C++ standards itself.
- Experience on Sunway and Lassen showed that the asynchronous execution is a key to maximize device occupancy by overlapping computation with host-device transfers and MPI communication.
- The tools to detect the race conditions (e.g., Thread Sanitizer and Intel Inspector) typically catch race conditions occurring between simultaneous read-write or write-write. However, the root cause of one race condition turned out to be freeing the

allocated memory by one thread while others continued using it. The tools would detect the simultaneous read-write only if the freed memory is reallocated to some other data structure. On a fortunate occurrence after debugging for two months, gdb halted the execution when one thread was freeing the allocated memory, and another thread was trying to use the same variable. One should be extremely careful while freeing memory and think about possible race conditions because detecting them is extremely tricky.

- Typically, multithreading is expected to reduce MPI communication compared to an MPI Only version and perform faster. However, the Hypre case study showed an example of how OpenMP, if not used correctly, can instead cause slowdowns. The same case study also showed a new approach to refactor legacy codes for the new multicore/manycore processors.
- The Logical Vector Length example in the case of two-dimensional convolution and sparse-matrix vector multiplications showed that the SIMD primitive could improve the performance of vectorized code, and one should always look for opportunities for optimizations in spite of an apparent stopping point.
- Keeping code simple is essential for performance and debugging at a large scale. Some of the code in the Arches module became so convoluted that a huge effort was required to debug simple problems.
- Rapid prototyping helps in experimenting with multiple options. Around three to six additional options were tried for Chapters 5, 6, and 7. These options are not included in this dissertation because they failed to achieve the desired results. However, rapid prototyping helped to quickly figure out what is working, what is not working, what the bottlenecks are, and what should be the next direction.
- Incremental developments helped in porting tasks to GPU. Tens tasks were ported initially using PPL as the application, and infrastructure development was taking place in parallel. However, soon it became very complicated to debug when the application started experiencing segmentation faults, race conditions, and inaccurate results at multiple places. The "task tagging" logic introduced in [150] made it possible to promote or demote tasks to/from GPU with a simple tag passed as a parameter.

Also, AMT's task scheduler gave control over the number of tasks executed at runtime. This flexibility gave us a chance to discover defects by promoting tasks incrementally and executing in steps.

9.7 Future Work

Although this work moved Uintah closer to running on the exascale supercomputer, some more enhancements may be needed to achieve that goal.

- **Portable scheduler**: The heterogeneous scheduler and related infrastructure seen in Chapter 7 still uses some raw CUDA code. The raw CUDA (e.g., CUDAMemCpyAsync, CUDA streams, kernels to copy ghost cells, etc.) need to be replaced with portable APIs through indirect PPL within Uintah and corresponding Kokkos methods if available (e.g. Kokkos' deep_copy, Kokkos instances, and Kokkos code to copy ghost cells, etc.) As a result, the entire Uintah infrastructure can become portable, theoretically allowing running Uintah on any platform with minimal changes in the indirect Performance Portability Layer. The indirect Performance Portability Layer would even allow abstraction of different native programming models such as CUDA, HIP, SYCL, or OpenMP. Such abstractions can handle new functionalities introduced by any of the native programming models.
- **Intel and AMD specific infrastructure**: There is a possibility that 100% fully portable scheduler may not be possible due to some yet unknown limitations of the Aurora/Frontier hardware or programming models. In this scenario, Intel/AMD specific infrastructure developments might be needed to run Uintah on DOE Aurora and DOE Frontier in the near future. A possible approach is to write a portable scheduler as much as possible and use platform specific code only when must. As per the latest development, SYCL is being considered an important programming model to achieve GPU execution and portable across Nvidia, Intel, and AMD GPUS [22, 23]. So, a task scheduler using SYCL can possibly provide necessary performance portability.
- **Vectorizing key kernels**: Vectorizing the RMCRT kernel will be critical before running Uintah on Fugaku, and possibly on DOE Aurora provided Intel GPUs are capable of executing the vector instructions. The portable SIMD primitive will come in handy in

this case. An algorithm called "dataOnionSlim" [24] can be a better fit for vectorization as it reduces the number of random memory accesses.

- **Improving communication performance:** The modified version of MPICH improved communication performance in Hypre (Chapter 6). The same version will be effective in Uintah to improve MPI_THREAD_MULTIPLE communication performance because: 1) the fine-grained locks within MPI minimize waiting for a global lock used in the current version of MPI; and 2) Virtual Communication Interfaces (VCIs) can leverage parallelism in network resources within MPI.
- **Energy-aware AMT**: AMTs can control the task scheduling and easily gather per-task heuristics. Such heuristics can be used to control the input power to the compute resources on a per-task basis. The AMT task schedulers can potentially perform "energy-aware scheduling" to minimize energy consumption.

REFERENCES

[1] Argonne Leadership Computing Facility. "Aurora." Alcf.anl.gov. https://www.alcf.anl.gov/aurora (accessed Aug. 1, 2021).

[2] Department of Energy. "U.S. Department of Energy and Cray to Deliver Record-Setting Frontier Supercomputer at ORNL." Energy.gov. https://www.energy.gov/articles/us-department-energy-and-cray-deliver-record-setting-frontier-supercomputer-ornl (accessed Aug. 15, 2021).

[3] Top500. "June 2021." Top500.org. https://top500.org/lists/top500/2021/06 (accessed Jul. 2, 2021).

[4] Texas Advanced Computing Center. "Frontera." Tacc.utexas.edu. https://www.tacc.utexas.edu/systems/frontera (accessed Aug. 5, 2021).

[5] IT Peer Network. "Think Exponential: Intel's X^e Architecture." Itpeernetwork.com. https://itpeernetwork.intel.com/intel-xe-compute/#gs.emsehp (accessed Jul. 5, 2021).

[6] Intel. "oneAPI: A new era of heterogeneous computing." Intel.com. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html (accessed Aug. 7, 2021).

[7] Khronos. "SYCL." Khronos.org. https://www.khronos.org/sycl/ (accessed Aug. 11, 2021).

[8] Sandia National Laboratories. "Kokkos: The C++ performance portability programming model." Github.com. https://github.com/kokkos/kokkos/wiki (accessed Jul. 11, 2021).

[9] Sandia National Laboratories. "Kokkos tutorials." Github.com. https://github.com/kokkos/kokkos-tutorials (accessed Jul. 11, 2021).

[10] The ASCAC Subcommittee on Exascale Computing. "The Opportunities and Challenges of Exascale Computing," U.S. Department of Energy, Washington, DC, USA, 2010.".

[11] Lawrence Livermore National Laboratory. "Quartz." Llnl.gov. https://hpc.llnl.gov/hardware/platforms/Quartz (accessed Aug. 12, 2021).

[12] Top500. "June 2019." Top500.org. https://top500.org/lists/top500/2019/06 (accessed Jul. 12, 2021).

[13] J. Dongarra. "Report on the Sunway TaihuLight System." Netlib.org. http://www.netlib.org/utk/people/JackDongarra/PAPERS/sunway-report-2016.pdf (accessed Jul. 4, 2018).

[14] NSCCWX. "Sunway TaihuLight compiler system user manual (in Chinese)." Nsccwx.cn http://www.nsccwx.cn/ceshi.php?id=19 (accessed Jan. 21 2018).

[15] Intel. "Requirements for vectorizable loops." Intel.com. https://software.intel.com/en-us/articles/requirements-for-vectorizable-loops (accessed Jul. 19, 2021).

[16] Argonne Leadership Computing Facility. "Aries network on Theta." Alcf.anl.gov. https://www.alcf.anl.gov/support-center/theta/aries-network-theta (accessed Aug. 1, 2021).

[17] Intel. "Multiple endpoints support." Intel.com. https://software.intel.com/en-us/mpi-developer-guide-linux-multiple-endpoints-support (accessed Aug. 11, 2021).

[18] Nvidia. "V100 GPU architecture." Nvidia.com. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf (accessed Aug. 22, 2021).

[19] Wikichip. "Power9 - microarchitectures - IBM." En.wikichip.org. https://en.wikichip.org/wiki/ibm/microarchitectures/power9 (accessed Aug. 26, 2021).

[20] Intel. "Intel xeon platinum 8280 processor." Intel.com. https://ark.intel.com/content/www/us/en/ark/products/192478/intel-xeon-platinum-8280-processor-38-5m-cache-2-70-ghz.html (accessed Aug. 13, 2021).

[21] T. P. Morgan. "IBM rounds out Power9 systems for HPC, analytics." Nextplatform.com. https://www.nextplatform.com/2018/05/09/ibm-rounds-out-power9-systems-for-hpc-analytics/ (accessed Aug. 19, 2021).

[22] HPCWire. "NERSC, ALCF, Codeplay partner on SYCL for next-generation supercomputers." Hpcwire.com. https://www.hpcwire.com/off-the-wire/nersc-alcf-codeplay-partner-on-sycl-for-next-generation-supercomputers/ (accessed Aug. 2, 2021).

[23] HPCWire. "Argonne, ORNL award Codeplay contract to strengthen SYCL support for AMD GPUs." Hpcwire.com. https://www.hpcwire.com/off-the-wire/argonne-ornl-award-codeplay-contract-to-strengthen-sycl-support-for-amd-GPUs/ (accessed Aug. 2, 2021).

[24] B. Peterson. "RMCRTTest.cc." Github.com. https://github.com/Uintah/Uintah/blob/kokkos_dev/src/CCA/Components/Examples/RMCRT_Test.cc (accessed Aug. 15, 2021).

[25] W. P. Adamczyk, B. Isaac, J. Parra-Alvarez, S. T. Smith, D. Harris, J. N. Thornock, M. Zhou, P. J. Smith, and R. Żmuda, "Application of LES-CFD for predicting pulverized-coal working conditions after installation of NOx control system," *Energy*, vol. 160, pp. 693–709, Oct. 2018.

[26] A. Agbaria and R. Friedman, "Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations," in *Proc. 8th IEEE Int. Symp. High Perform. Distrib. Comput.*, 1999, p. 31.

[27] I. Ahmad and M. Berzins, "MOL solvers for hyperbolic PDEs with source terms," *Math. Comput. Simul.*, vol. 56, pp. 1115–1125, 2001.

[28] Y. Ali, N. Onodera, Y. Idomura, and T. Ina, "GPU acceleration of communication avoiding Chebyshev basis conjugate gradient solver for multiphase CFD simulations," in *2019 IEEE/ACM 10th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst.*, pp. 1–8.

[29] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC series network," Cray Inc. White Paper WP-Aries01-1112, 2012.

[30] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurr. Comput.*, vol. 23, no. 2, pp. 187–198, 2011.

[31] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling hypre's multigrid solvers to 100,000 cores," in *High-Performance Scientific Computing: Algorithms and Applications*, M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied, Eds. London, England: Springer London, 2012, pp. 261–279.

[32] A. Bardakoff, B. Bachelet, T. Blattner, W. Keyrouz, G. C. Kroiz, and L. Yon, "Hedgehog: Understandable scheduler-free heterogeneous asynchronous multithreaded data-flow graphs," in *IEEE/ACM 3rd Annu. Parallel Appl. Workshop: Alternatives to MPI+X (PAW-ATM)*, 2020, pp. 1–15.

[33] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage and Anal.*, 2012, p. 66.

[34] J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi, J. Wilke, M. Bettencourt, S. Bova, K. Franko, P. Lin, R. Grant, S. Hammond, S. Olivier, L. Kale, N. Jain, E. Mikida, A. Aiken, M. Bauer, W. Lee, E. Slaughter, S. Treichler, M. Berzins, T. Harman, A. Humphrey, J. Schmidt, D. Sunderland, P. McCormick, S. Gutierrez, M. Schulz, A. Bhatele, D. Boehme, P. Bremer, and T. Gamblin, "ASC ATDM level 2 milestone #5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms," Sandia National Laboratories (SNL-NM), Albuquerque, NM, USA, Tech. Rep., 2015.

[35] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," Defense Adv. Res. Projects Agency Inf. Process. Techn. Office (DARPA IPTO), Tech. Rep., 2008.

[36] K. Bergman, T. Conte, A. Gara, M. Gokhale, M. Heroux, P. Kogge, B. Lucas, S. Matsuoka, V. Sarkar, and O. Temam, "Future high performance computing capabilities: Summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee," USDOE Office of Science (SC)(United States), Tech. Rep., 2019.

[37] M. Berzins, "Adaptive polynomial interpolation on evenly spaced meshes." *SIAM Rev.*, vol. 49, no. 4, pp. 604–627, Nov. 2007.

[38] ——, "Status of release of the Uintah computational framework," Scientific Computing and Imaging Institute, Salt Lake City, UT, USA, Tech. Rep. UUSCI-2012-001, 2012.

[39] M. Berzins, P. Capon, and P. Jimack, "On spatial adaptivity and interpolation when using the method of lines," *Appl. Numer. Math.*, vol. 26, no. 1, pp. 117–134, 1998.

[40] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight, "Extending the Uintah framework through the petascale modeling of detonation in arrays of high explosive devices," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S101–S122, Oct. 2016.

[41] W. Bland, "User level failure mitigation in MPI," in *Euro-Par 2012: Parallel*, 2012, p. 499–504.

[42] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang, "Distributed pC++ basic ideas for an object parallel language," *Sci. Program.*, vol. 2, no. 3, pp. 7–22, 1993.

[43] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *SC '02: Proc. 2002 ACM/IEEE Conf. Supercomput.*, Nov. 2002, pp. 29–29.

[44] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Comp. Sci. Eng.*, vol. 15, no. 6, pp. 36–45, Nov. 2013.

[45] A. Bouteiller, "Fault-tolerant MPI," in *Fault-Tolerance Techniques for High-Performance Computing*, T. Herault and Y. Robert, Eds. Cham, Switzerland: Springer International Publishing, 2015, pp. 145–228.

[46] G. Bronevetsky, D. Marques, K. Pingali, S. McKee, and R. Rugina, "Compiler-enhanced incremental checkpointing for OpenMP applications," in *2009 IEEE Int. Symp. Parallel Distrib. Process.*, pp. 1–12.

[47] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir, "Toward exascale resilience: 2014 update," *Supercomp. Front. Innov.*, vol. 1, no. 1, pp. 5–28, Jun. 2014.

[48] S. Carr, "Combining optimization for cache and instruction-level parallelism," in *Proc. 1996 Conf. Parallel Archit. Compil. Tech.*, 1996, pp. 238–247.

[49] Z. Chen and J. Dongarra, "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources," in *Proc. 20th IEEE Int. Parallel Distrib. Process. Symp.*, 2006, p. 10.

[50] ——, "Algorithm-based fault tolerance for fail-stop failures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 12, pp. 1628–1641, Dec. 2008.

[51] A. Chien, P. Balaji, N. Dun, A. Fang, H. Fujita, K. Iskra, Z. Rubenstein, Z. Zheng, J. Hammond, I. Laguna *et al.*, "Exploring versioned distributed arrays for resilience in scientific applications: Global View resilience," *Int. J. High Perform. Comput. Appl.*, vol. 31, no. 6, pp. 564–590, Nov. 2017.

[52] B. Cope *et al.*, "Implementation of 2D convolution on FPGA, GPU and CPU," *Imperial College Rep.*, pp. 2–5, 2006.

[53] G. Daiß, M. Simberg, A. Reverdell, J. Biddiscombe, T. Pollinger, H. Kaiser, and D. Pflüger, "Beyond fork-join: Integration of performance portable Kokkos Kernels with HPX," in *2021 IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, pp. 377–386.

[54] W. J. Dally and A. A. Chien, "Object-oriented concurrent programming in CST," in *Proc. 3rd Conf. Hypercube Concurrent Comput. Appl: Archit., Soft, Comput. Syst., General Issues*, vol. 1, 1988, pp. 434–439.

[55] D. Dauwe, S. Pasricha, A. A. Maciejewski, and H. J. Siegel, "An analysis of resilience techniques for exascale computing platforms," in *2017 IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, pp. 914–923.

[56] ——, "A performance and energy comparison of fault tolerance techniques for exascale computing systems," in *2016 IEEE Int. Conf. Comp. Inform. Tech. (CIT)*, pp. 436–443.

[57] T. Davies and Z. Chen, "Correcting soft errors online in LU factorization," in *Proc. 22nd Int. Symp. High Perform. Parallel and Distrib. Comput.*, 2013, pp. 167–178.

[58] T. Davies, C. Karlsson, H. Liu, C. Ding, and Z. Chen, "High performance linpack benchmark: A fault tolerant implementation without checkpointing," in *Proc. Int. Conf. Supercomput.*, 2011, pp. 162–171.

[59] T. Deakin, J. Price, M. Martineau, and S. McIntosh-Smith, "Evaluating attainable memory bandwidth of parallel programming models via babelstream," *Int. J. Comput. Sci. and Eng.*, vol. 17, no. 3, pp. 247–262, Oct. 2018.

[60] C. Di Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer, "Lessons learned from the analysis of system failures at petascale: The case of blue waters," in *44th Annu. IEEE/IFIP Int. Conf. Dependable Syst. and Netw.*, pp. 610–621.

[61] M. Di Renzo, L. Fu, and J. Urzay, "HTR solver: An open-source exascale-oriented task-based multi-GPU high-order code for hypersonic aerothermodynamics," *Comput. Phys. Commun.*, vol. 255, p. 107262, Oct. 2020.

[62] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling MPI interoperability through flexible communication endpoints," in *Proc. 20th Eur. MPI Users' Group Meeting*, 2013, pp. 13–18.

[63] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur, "Enabling communication concurrency through flexible MPI endpoints," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 4, pp. 390–405, Sep. 2014.

[64] X. Dong, N. Muralimanohar, N. Jouppi, R. Kaufmann, and Y. Xie, "Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems," in *Proc. Conf. High Perform. Comput. Netw., Storage and Anal.*, p. 57.

[65] J. Dongarra, T. Herault, and Y. Robert, "Fault tolerance techniques for high-performance computing," in *Fault-Tolerance Techniques for High-Performance Computing*, T. Herault and Y. Robert, Eds. Cham, Switzerland: Springer International Publishing, 2015, pp. 3–85.

[66] A. Dubey, H. Fujita, D. Graves, A. Chien, and D. Tiwari, "Granularity and the cost of error recovery in resilient AMR scientific applications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, ser. SC '16, 2016, pp. 42:1–42:10.

[67] A. Dubey, P. Mohapatra, and K. Weide, "Fault tolerance using lower fidelity data in adaptive mesh applications," in *Proc. 3rd Workshop Fault-Tolerance HPC Extreme Scale*, 2013, pp. 3–10.

[68] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel and Distrib. Comput.*, vol. 74, no. 12, pp. 3202 – 3216, Dec. 2014.

[69] R. Espasa and M. Valero, "Exploiting instruction-and data-level parallelism," *IEEE Micro*, vol. 17, no. 5, pp. 20–27, Sep.-Oct. 1997.

[70] G. Fagg and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world." *7th Eur. PVM/MPI Users Group Meeting Recent Adv. in Parallel Virtual Mach. Message Passing Interface*, pp. 346–353, 2000.

[71] R. D. Falgout, J. E. Jones, and U. M. Yang, "Pursuing scalability for hypre's conceptual interfaces," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 326–350, Sep. 2005.

[72] ——, "The design and implementation of hypre, a library of parallel high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, A. M. Bruaset and A. Tveito, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 267–294.

[73] A. Fang, A. Cavelan, Y. Robert, and A. A. Chien, "Resilience for stencil computations with latent errors," in *46th Int. Conf. on Parallel Processing (ICPP)*, 2017, pp. 581–590.

[74] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang *et al.*, "The Sunway TaihuLight supercomputer: System and applications," *Sci. China Inf. Sci.*, vol. 59, no. 7, Jul. 2016.

[75] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Modeling the performance of an algebraic multigrid cycle using hybrid MPI/OpenMP," in *2012 41st Int. Conf. Parallel Process.*, Sep. 2012, pp. 128–137.

[76] M. Gasca and T. Sauer, "On the history of multivariate polynomial interpolation," *J. Comput. Appl. Math.*, vol. 122, no. 1, pp. 23–35, 2000.

[77] A. Gopalakrishnan, M. A. Cabral, J. P. Erwin, and R. B. Ganapathi, "Improved MPI multi-threaded performance using ofi scalable endpoints," in *2019 IEEE Symp. High-Perform. Interconnects (HOTI)*, pp. 36–39.

[78] S. Gottlieb, C. W. Shu, and E. Tadmor, "Strong stability-preserving high-order time discretization methods," *SIAM Rev.*, vol. 43, no. 1, pp. 89–112, Aug. 2001.

[79] A. S. Grimshaw, "Easy-to-use object-oriented parallel processing with Mentat," *Comput.*, vol. 26, no. 5, pp. 39–51, May 1993.

[80] J. Guilkey, T. Harman, J. Luitjens, J. Schmidt, J. Thornock, J. de St. Germain, S. Shankar, J. Peterson, and C. Brownlee, "Uintah user guide version 1.1," SCI Institute, University of Utah, Salt Lake City, UT, USA, SCI Technical Report UUSCI-2009-007, 2009.

[81] S. Gupta, T. Patel, C. Engelmann, and D. Tiwari, "Failures in large scale systems: Long-term measurement, analysis, and implications," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, p. 44.

[82] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Commun. ACM*, vol. 28, no. 1, pp. 34–52, Jan. 1985.

[83] P. Hargrove and J. Duell, "Berkeley Lab Checkpoint/Restart (BLCR) for Linux clusters," *J. Phys. Conf. Ser.*, vol. 46, p. 494, Jun. 2006.

[84] A. Harten, B. Engquist, S. Osher, and S. R. Chakravarthy, "Uniformly high order accurate essentially non-oscillatory schemes, III," *J. Comput. Phys.*, vol. 131, no. 1, pp. 3–47, Feb. 1997.

[85] T. Henretty, K. Stock, L. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector SIMD architectures," in *Int. Conf. Compiler Constr.*, 2011, pp. 225–245.

[86] T. Herault, Y. Robert, G. Bosilca, and J. Dongarra, "Generic matrix multiplication for multi-GPU accelerated distributed-memory platforms over PaRSEC," in *2019 IEEE/ACM 10th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst. (ScalA)*, pp. 33–41.

[87] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L. Pouchet, A. Rountev, and P. Sadayappan, "Dynamic trace-based analysis of vectorization potential of applications," *ACM SIGPLAN Not.*, vol. 47, no. 6, pp. 371–382, Jun. 2012.

[88] J. K. Holmen, A. Humphrey, and M. Berzins, "Exploring use of the reserved core," in *High Performance Parallelism Pearls: Multicore and Many-Core Programming Approaches*, J. Reinders and J. Jeffers, Eds. Boston, MA, USA: Morgan Kaufmann, 2015, vol. 2, ch. 13, pp. 229–242.

[89] J. K. Holmen, A. Humphrey, D. Sutherland, and M. Berzins, "Improving Uintah's scalability through the use of portable Kokkos-based data parallel tasks," in *Proc. Pract. Exp. in Adv. Res. Comput. Sustain., Success and Impact*, no. 27, 2017, pp. 27:1–27:8.

[90] J. K. Holmen, B. Peterson, and M. Berzins, "An approach for indirectly adopting a performance portability layer in large legacy codes," in *2019 IEEE/ACM Int. Workshop Perform., Portability and Productiv. HPC (P3HPC)*, pp. 36–49.

[91] J. K. Holmen, B. Peterson, O. H. Diaz, J. Thornock, A. Humphrey, D. Sutherland, and M. Berzins, "Improving Uintah's readiness for exascale systems through the use of Kokkos and nested OpenMP," no. UUSCI-2012-002, 2018.

[92] J. K. Holmen, B. Peterson, A. Humphrey, D. Sunderland, O. H. Diaz-Ibarra, J. N. Thornock, and M. Berzins, "Portably improving Uintah's readiness for exascale systems through the use of Kokkos," SCI Institute, Tech. Rep. UUSCI-2019-001, 2019.

[93] J. K. Holmen, D. Sahasrabudhe, and M. Berzins, "A heterogeneous MPI+PPL task scheduling approach for asynchronous many-task runtime systems," in *Proc. Pract. Exp. Adv. Res. Comput. 2021 Sustain. Success Impact (PEARC21)*, 2021.

[94] J. Holmen, "Portable, scalable approaches for improving asynchronous many-task runtime node use," Ph.D. dissertation, School of Comput., Univ. of Utah, Salt Lake City, UT, USA, 2021.

[95] J. K. Holmen, A. Humphrey, D. Sunderland, and M. Berzins, "Improving Uintah's scalability through the use of portable Kokkos-based data parallel tasks," in *Proc. Pract. Exp. Adv. Res. Comput. 2017 Sustain. Success Impact*, 2017, pp. 1–8.

[96] R. D. Hornung and J. A. Keasler, "The RAJA portability layer: Overview and status," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2014.

[97] C. R. Houck and G. Agha, "Hal: A high-level actor language and its distributed implementation," in *ICPP (2)*, K. G. Shin, Ed., 1992, pp. 158–165.

[98] M. Howard, T. Fisher, M. Hoemmen, D. Dinzl, J. Overfelt, A. Bradley, K. Kim, and S. Rajamanickam, "Employing multiple levels of parallelism for CFD at large scales on next generation high-performance computing platforms," in *Proc. 10th Int. Conf. Comput. Fluid Dyn.*, 2018.

[99] K.-H. Huang and J. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE Trans. Comput.*, vol. 100, no. 6, pp. 518–528, Jun. 1984.

[100] M. Huber, B. Gmeiner, U. Rüde, and B. Wohlmuth, "Resilience for massively parallel multigrid solvers," *SIAM J. Sci. Comput.*, vol. 38, no. 5, pp. S217–S239, Oct. 2016.

[101] A. Humphrey and M. Berzins, "An evaluation of an asynchronous task based dataflow approach for Uintah," in *2019 IEEE 43rd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2, pp. 652–657.

[102] 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*, J. M. Kunkel and T. Ludwig, Eds. Cham, Switzerland; Springer International, 2015, vol. 9137, pp. 212–230.

[103] A. Humphrey, Q. Meng, M. Berzins, and T. Harman, "Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system," in *Proc. 1st Conf. Extreme Sci. and Eng. Discov. Environ. (XSEDE'12)*, 2012.

[104] A. Humphrey, D. Sunderland, T. Harman, and M. Berzins, "Radiative heat transfer calculation on 16384 GPUs using a reverse Monte Carlo ray tracing approach with adaptive mesh refinement," in *2016 IEEE Int. Parallel and Distrib. Process. Symp. Workshops (IPDPSW)*, pp. 1222–1231.

[105] A. Humphrey, "Scalable asynchronous many-task runtime solutions to globally coupled problems," Ph.D. dissertation, School of Comput., Univ. of Utah, Salt Lake City, UT, USA, 2019.

[106] I. Hunsaker, T. Harman, J. Thornock, and P. Smith, "Efficient parallelization of RMCRT for large scale LES combustion simulations," in *20th AIAA Comput. Fluid Dyn. Conf.*, 2011, p. 3770.

[107] Z. Hussain, T. Znati, and R. Melhem, "Partial redundancy in HPC systems with non-uniform node reliabilities," in *Proc. Int. Conf. for High Perform. Comput. Netw. Storage, Anal.*, 2018, pp. 44:1–44:11.

[108] Y. Idomura, T. Ina, S. Yamashita, N. Onodera, S. Yamada, and T. Imamura, "Communication avoiding multigrid preconditioned conjugate gradient method for extreme scale multiphase CFD simulations," in *2018 IEEE/ACM 9th Workshop Latest Adv. Scalable Algorithms Large-Scale Syst. (scalA)*, pp. 17–24.

[109] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "MCREngine: A scalable checkpointing system using data-aware aggregation and compression," in *SC '12: Proc. Int. Conf. High Perform. Comput., Netw. Storage Anal.*, 2012, pp. 1–11.

[110] A. Jacob, S. Antao, H. Sung, A. Eichenberger, C. Bertolli, G. Bercea, T. Chen, Z. Sura, G. Rokos, and K. O'Brien, "Towards performance portable GPU programming with RAJA," in *Workshop Portability Among HPC Archit. for Sci. Appl.*, 2015.

[111] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Boston, MA, USA: Morgan Kaufmann Publishers Inc., 2016.

[112] A. Johnson, "Area exam: General-purpose performance portable programming models for productive exascale computing," University of Oregon, Computer and Information Sciences Department, Tech. Rep., Jun. 2020.

[113] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "HPX: A task based programming model in a global address space," in *Proc. 8th Int. Conf. Partitioned Glob. Address Space Program. Models*, 2014, pp. 6:1–6:11.

[114] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on C++," in *Proc. 8th Annu. Conf. Object-oriented Program. Syst., Lang., Appl.*, 1993, pp. 91–108.

[115] ——, "Charm++: A portable concurrent object oriented system based on C++," *SIGPLAN Not.*, vol. 28, no. 10, pp. 91–108, Oct. 1993.

[116] L. Kale and G. Zheng, "The Charm++ programming model," in *Parallel Science and Engineering Applications: The Charm++ Approach*. Boca Raton, FL, CRC Press, Apr. 2016, pp. 1–16.

[117] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using NVM as virtual memory," in *2013 IEEE 27th Int. Symp. Parallel and Distrib. Process.*, pp. 29–40.

[118] P. Karpiński and J. McDonald, "A high-performance portable abstract interface for explicit SIMD vectorization," in *Proc. 8th Int. Workshop Program. Models and Appl. Multicores Manycores*, 2017.

[119] J. L. Kelly, C. Lochbaum, and V. A. Vyssotsky, "A block diagram compiler," *Bell System Tech. J.*, vol. 40, no. 3, pp. 669–678, May 1961.

[120] K. Kim, T. Costa, M. Deveci, A. Bradley, S. Hammond, M. Guney, S. Knepper, S. Story, and S. Rajamanickam, "Designing vector-friendly compact BLAS and LAPACK kernels," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, p. 55.

[121] K. Kim, S. Hammond, E. Boman, A. Bradley, S. Rajamanickam, M. Deveci, M. Hoemmen, and C. Trott, "KokkosKernels v. 0.9, version 00," Feb. 2017.

[122] M. Kretz and V. Lindenstruth, "Vc: A C++ library for explicit vectorization," *Softw. Pract. Exp.*, vol. 42, no. 11, pp. 1409–1430, Dec. 2012.

[123] G. Krishnamoorthy, R. Rawat, and P. J. Smith, "Parallelization of the P-1 radiation model," *Numeri. Heat Transf. Fundam.*, vol. 49, no. 1, pp. 1–17, 2006.

[124] A. Kulkarni and A. Lumsdaine, "A comparative study of asynchronous many-tasking runtimes: Cilk, Charm++, paralleX and AM++," *arXiv:1904.00518*, 2019.

[125] S. Kumar, D. Hoang, S. Petruzza, J. Edwards, and V. Pascucci, "Reducing network congestion and synchronization overhead during aggregation of hierarchical data," in *2017 IEEE 24th Int. Conf. High Perf. Comput. (HiPC)*, pp. 223–232.

[126] S. Kumar, A. Humphrey, W. Usher, S. Petruzza, B. Peterson, J. A. Schmidt, D. Harris, B. Isaac, J. Thornock, T. Harman, V. Pascucci, and M. Berzins, "Scalable data management of the Uintah simulation framework for next-generation engineering problems with radiation," in *Asian Conf. Supercomput. Front.*, 2018, pp. 219–240.

[127] S. Kumar, V. Vishwanath, P. Carns, J. Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, R. Ross, M. Papka, J. Chen, and V. Pascucci, "Efficient data restructuring and aggregation for I/O acceleration in PIDX," in *Proc. Int. Conf. on High Perf. Comput., Netw., Storage Anal.*, 2012, pp. 50:1–50:11.

[128] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout, "PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2011, pp. 103–111.

[129] R. Leißa, S. Hack, and I. Wald, "Extending a C-like language for portable SIMD programming," *ACM SIGPLAN Not.*, vol. 47, no. 8, pp. 65–74, Feb. 2012.

[130] R. J. LeVeque and H. C. Yee, "A study of numerical methods for hyperbolic conservation laws with stiff source terms," *J. Comput. Phys.*, vol. 86, no. 1, pp. 187–210, Jan. 1990.

[131] D. Light and D. Durran, "Preserving nonnegativity in discontinuous Galerkin approximations to scalar transport via truncation and mass aware rescaling (TMAR)," *Mon. Weather Rev.*, vol. 144, pp. 4771–4786, Dec. 2016.

[132] R. Lucas, J. Ang, K. Bergman, S. Borkar, W. Carlson, L. Carrington, G. Chiu, R. Colwell, W. Dally, J. Dongarra *et al.*, "DOE advanced scientific computing advisory subcommittee (ASCAC) report: Top ten exascale research challenges," USDOE Office of Science (SC)(United States), Tech. Rep., 2014.

[133] J. Luitjens and M. Berzins, "Improving the performance of Uintah: A large-scale adaptive meshing computational framework," in *Proc. 24th IEEE Int. Parallel and Distrib. Process. Symp. (IPDPS10)*, 2010, pp. 1–10.

[134] D. F. Martin and P. Colella, "A cell-centered adaptive projection method for the incompressible Euler equations," *J. Comput. Phys.*, vol. 163, no. 2, pp. 271–312, Sep. 2000.

[135] P. McCorquodale, P. Colella, D. P. Grote, and J. L. Vay, "A node-centered local refinement algorithm for Poisson's equation in complex geometries," *J. Comput. Phys.*, vol. 201, no. 1, pp. 34–60, Nov. 2004.

[136] D. Medina, A. St-Cyr, and T. Warburton, "OCCA: A unified approach to multi-threading languages," *arXiv:1403.0968*, 2014.

[137] E. Meneses, X. Ni, T. Jones, and D. Maxwell, "Analyzing the interplay of failures and workload on a leadership-class supercomputer," *Comput.*, vol. 2, no. 3, p. 4, 2015.

[138] Q. Meng, M. Berzins, and J. Schmidt, "Using hybrid parallelism to improve memory use in Uintah," in *Proc. TeraGrid 2011 Conf.*

[139] Q. Meng, A. Humphrey, and M. Berzins, "The Uintah framework: A unified heterogeneous task scheduling and runtime system," in *High Perform. Comput. Netw. Storage Anal. (SCC), 2012 SC Companion:*, Nov. 2012, pp. 2441–2448.

[140] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins, "Investigating applications portability with the Uintah DAG-based runtime system on petascale supercomputers," in *Proc. SC13: Int. Conf. High Perf. Comput. Netw., Storage Anal.*, 2013, pp. 96:1–96:12.

[141] ——, "Preliminary experiences with the Uintah framework on Intel Xeon Phi and Stampede," in *Proc. Conf. Extreme Sci. Eng. Discovery Environ. Gateway Disc. (XSEDE 2013)*, 2013, pp. 48:1–48:8.

[142] Q. Meng, J. Luitjens, and M. Berzins, "Dynamic task scheduling for the Uintah framework," in *Proc. 3rd IEEE Workshop Many-Task Comput. on Grids Supercomput. (MTAGS10)*, 2010, pp. 1–10.

[143] A. Moody and G. Bronevetsky, "Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O," Lawrence Livermore National Lab.(LLNL), Livermore, CA, USA, Tech. Rep., 2008.

[144] V. Morozov, K. Kumaran, V. Vishwanath, J. Meng, and M. E. Papka, "Early experience on the Blue Gene/Q supercomputing system," in *2013 IEEE 27th Int. Symp. Parallel Distrib. Process.*, pp. 1229–1240.

[145] A. Munshi, "The OpenCL specification version: 1.0 document revision: 48," Khronos Group, Tech. Rep., Mar. 2008.

[146] S. Narumi, "Some formulas in the theory of interpolation of many independent variables," *Tohoku Math. J.*, vol. 18, pp. 309–321, Mar. 1920.

[147] J. P., J. Thornock, S. Smith, and P. Smith, "Large eddy simulation of polydisperse particles in turbulent coaxial jets using the direct quadrature method of moments," *Int. J. Multiph. Flow*, vol. 63, pp. 23–38, 2014.

[148] S. Pai, R. Govindarajan, and M. Thazhuthaveetil, "PLASMA: Portable programming for SIMD heterogeneous accelerators," in *Workshop Lang., Compiler, Archit. Support GPGPU*, 2010.

[149] T. Patinyasakdikul, D. Eberius, G. Bosilca, and N. Hjelm, "Give MPI threading a fair chance: A study of multithreaded MPI designs," in *2019 IEEE Int. Conf. Cluster Comput. (CLUSTER)*.

[150] B. Peterson, "Portable and performant GPU/heterogeneous asynchronous many-task runtime system," Ph.D. dissertation, School of Comput., Univ. of Utah, Salt Lake City, UT, USA, Dec. 2019.

[151] B. Peterson, H. Dasari, A. Humphrey, J. Sutherland, T. Saad, and M. Berzins, "Reducing overhead in the Uintah framework to support short-lived tasks on GPU-heterogeneous architectures," in *Proc. 5th Int. Workshop Domain-Specif. Lang. High-Level Frameworks for High Perform. Comput.*, 2015, pp. 4:1–4:8.

[152] B. Peterson, A. Humphrey, D. Sunderland, J. Sutherland, T. Saad, H. Dasari, and M. Berzins, "Automatic halo management for the Uintah GPU-heterogeneous asynchronous many-task runtime," *Int. J. Parallel Program.*, Dec. 2018.

[153] B. Peterson, N. Xiao, J. K. Holmen, S. Chaganti, A. Pakki, J. Schmidt, D. Sunderland, A. Humphrey, and M. Berzins, "Developing Uintah's runtime system for forthcoming architectures," SCI Inst., University of Utah, Salt Lake City, UT, USA, Tech. Rep., 2015.

[154] B. Peterson, A. Humphrey, J. K. Holmen, T. Harman, M. Berzins, D. Sunderland, and H. C. Edwards, "Demonstrating GPU code portability and scalability for radiative heat transfer computations," *J. Comput. Sci.*, vol. 27, pp. 303–319, Jul. 2018.

[155] B. Peterson, A. Humphrey, J. Schmidt, and M. Berzins, "Addressing global data dependencies in heterogeneous asynchronous runtime systems on GPUs," in *3rd Int. IEEE Workshop Extreme Scale Program. Models Middleware*, 2017.

[156] J. C. Phillips, D. J. Hardy, J. D. Maia, J. E. Stone, J. V. Ribeiro, R. C. Bernardi, R. Buch, G. Fiorin, J. Hénin, W. Jiang *et al.*, "Scalable molecular dynamics on CPU and GPU architectures with NAMD," *J. Chem. Phys.*, vol. 153, no. 4, p. 044130, Jul. 2020.

[157] E. Phipps, M. D'Elia, H. Edwards, M. Hoemmen, J. Hu, and S. Rajamanickam, "Embedded ensemble propagation for improving performance, portability, and scalability of uncertainty quantification on emerging computational architectures," *SIAM J. Sci. Comput.*, vol. 39, no. 2, pp. C162–C193, 2017.

[158] E. Phipps, R. Tuminaro, and C. Miller, "Stokhos: Trilinos tools for embedded stochastic-Galerkin uncertainty quantification methods," Sandia National Laboratories (SNL-NM), Albuquerque, NM, USA, Tech. Rep., 2008.

[159] S. B. Pope, *Turbulent Flows*. Cambridge, U.K.: Cambridge University Press, 2000.

[160] F. Qiao, W. Zhao, X. Yin, X. Huang, X. Liu, Q. Shu, G. Wang, Z. Song, X. Li, H. Liu *et al.*, "A highly effective global surface wave numerical simulation with ultra-high resolution," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 46–56.

[161] F. Rizzi, K. V. Morris, B. Cook, K. Sargsyan, P. D. Mycek, O. L. Le Maitre, O. D. Knio, K. S. Dahlgren, and B. Debusschere, "Performance scaling variability and energy analysis for a resilient ULFM-based PDE solver," Sandia National Lab (SNL-CA), Livermore, CA, USA, Tech. Rep., 2016.

[162] D. Sahasrabudhe, M. Berzins, and J. Schmidt, "Node failure resiliency for Uintah without checkpointing," *Concurr. Comput. Pract. Exp.*, p. e5340, Oct. 2019.

[163] D. Sahasrabudhe, R. Zambre, A. Chandramowlishwaran, and M. Berzins, "Optimizing the hypre solver for manycore and GPU architectures," *J. Comput. Sci.*, vol. 49, p. 101279, Feb. 2021.

[164] D. Sahasrabudhe and M. Berzins, "Improving performance of the Hypre iterative solver for Uintah combustion codes on manycore architectures using MPI endpoints and kernel consolidation," in *Computational Science – ICCS 2020*, V. V. Krzhizhanovskaya, G. Závodszky, M. H. Lees, J. J. Dongarra, P. M. A. Sloot, S. Brissos, and J. Teixeira, Eds., pp. 175–190.

[165] D. Sahasrabudhe, E. T. Phipps, S. Rajamanickam, and M. Berzins, "A portable SIMD primitive using Kokkos for heterogeneous architectures," in *Accelerator Programming Using Directives*, S. Wienke and S. Bhalachandra, Eds., 2020, pp. 140–163.

[166] V. Sarkar, "Partitioning and scheduling parallel programs for execution on multiprocessors," Ph.D. dissertation, Dept. of Elect. Eng., Stanford Univ., Stanford, CA, USA, 1987, uMI Order No. GAX87-23080.

[167] K. Sato, N. Maruyama, K. Mohror, A. Moody, T. Gamblin, B. R. de Supinski, and S. Matsuoka, "Design and modeling of a non-blocking checkpointing system," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 19:1–19:10.

[168] J. Schmidt, M. Berzins, J. Thornock, T. Saad, and J. Sutherland, "Large scale parallel solution of incompressible flow problems using Uintah and hypre," in *2013 13th IEEE/ACM Int. Symp. Cluster, Cloud and Grid Comput. (CCGrid)*, pp. 458–465.

[169] ——, "Large scale parallel solution of incompressible flow Problems using Uintah and hypre," SCI Institute, University of Utah, Salt Lake City, UT, USA, SCI Technical Report UUSCI-2012-002, 2012.

[170] X. Shi, J. L. Pazat, E. Rodriguez, H. Jin, and H. Jiang, "Adapting grid applications to safety using fault-tolerant methods: Design, implementation and evaluations," *Future Gener. Comput. Syst.*, vol. 26, no. 2, pp. 236–244, Feb. 2010.

[171] C. W. Shu, "High order ENO and WENO schemes for computational fluid dynamics," in *High-Order Methods Computational Physics*, T. J. Barth and H. Deconinck, Eds. Berlin, Heidelberg, Germany: Springer Berlin Heidelberg, 1999, pp. 439–582.

[172] J. Silc, B. Robic, and T. Ungerer, "Asynchrony in parallel computing: From dataflow to multithreading." Citeseer, Mar. 1998, vol. 1, no. 1, pp. 3–30.

[173] P. J. Smith, R.Rawat, J. Spinti, S. Kumar, S. Borodai, and A. Violi, "Large eddy simulations of accidental fires using massively parallel computers," in *16th AIAA Comput. Fluid Dyn. Conf.*, 2003, p. 3697.

[174] J. Spinti, J. Thornock, E. Eddings, P. Smith, and A. Sarofim, "Heat transfer to objects in pool fires," *Transport Phenomena in Fires*, vol. 20, p. 69, 2008.

[175] S. Sridharan, J. Dinan, and D. Kalamkar, "Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints," in *Int. Conf. High Perform. Comput. Netw. Storage Anal., SC14*, 2014, pp. 487–498.

[176] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, "Frontera: The evolution of leadership computing at the National Science Foundation," in *Pract. Exp. Adv. Res. Comput.*, 2020, p. 106–111.

[177] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu *et al.*, "The ARM scalable vector extension," *IEEE Micro*, vol. 37, no. 2, pp. 26–39, Mar. 2017.

[178] X. Sun and P. J. Smith, "A parametric case study in radiative heat transfer using the reverse Monte-Carlo ray-tracing with full-spectrum K-distribution method," *J. Heat Transfer*, vol. 132, no. 2, Feb. 2010.

[179] D. Sunderland, B. Peterson, J. Schmidt, A. Humphrey, J. Thornock, and M. Berzins, "An overview of performance portability in the Uintah runtime system through the use of Kokkos," in *Proc. 2nd Int. Workshop Extreme Scale Program. Models Middleware*, 2016, pp. 44–47.

[180] X. Tian, H. Saito, E. Su, J. Lin, S. Guggilla, D. Caballero, M. Masten, A. Savonichev, M. Rice, E. Demikhovsky *et al.*, "LLVM compiler implementation for explicit parallelization and SIMD vectorization," in *Proc. 4th Workshop LLVM Compiler Infrastructure HPC*, 2017, p. 4.

[181] H. Torres, M. Papadakis, and L. Jofre Cruanyes, "Soleil-X: Turbulence, particles, and radiation in the regent programming language," in *SC'19: Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2019, pp. 1–4.

[182] C. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. A. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 4, pp. 805–817, Apr. 2021.

[183] C. R. Trott, "Kokkos: The C++ performance portability programming model," Sandia National Lab.(SNL-NM), Albuquerque, NM, USA, Tech. Rep., 2017.

[184] J. Varma, C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Scalable, fault tolerant membership for MPI tasks on HPC systems," in *Proc. 20th Annu. Int. Conf. Supercomput.*, 2006, pp. 219–228.

[185] R. Vasudevan, S. S. Vadhiyar, and L. V. Kalé, "G-Charm: An adaptive runtime system for message-driven parallel applications on hybrid systems," in *Proc. 27th Int. ACM Conf. Int. Conf. Supercomput.*, 2013, pp. 349–358.

[186] R. Vedder and D. Finn, "The hughes data flow multiprocessor: Architecture for efficient signal and data processing," *ACM SIGARCH Compute. Archit. News*, vol. 13, no. 3, pp. 324–332, 1985.

[187] C. Wang, F. Mueller, C. Engelmann, and S. L. Scott, "Hybrid checkpointing for MPI jobs in HPC environments," in *Parallel Distrib. Syst. (ICPADS), 2010 IEEE 16th Int. Conf.*, 2010, pp. 524–533.

[188] H. Wang, P. Wu, I. Tanase, M. Serrano, and J. Moreira, "Simple, portable and fast SIMD intrinsic programming: Generic SIMD library," in *Proc. 2014 Workshop Program. Models SIMD/Vector Process.*

[189] S. White and L. V. Kale, "Optimizing point-to-point communication between adaptive MPI endpoints in shared memory," *Concurr. Comput.*, vol. 32, no. 3, p. e4467, Mar. 2020.

[190] A. Yaşar, S. Rajamanickam, J. Berry, M. Wolf, J. S. Young, and U. V. ÇatalyÜrek, "Linear algebra-based triangle counting via fine-grained tasking on heterogeneous environments: (update on static graph challenge)," in *2019 IEEE High Perform. Extreme Comput. Conf. (HPEC)*, pp. 1–4.

[191] C. Yang, W. Xue, H. Fu, H. You, X. Wang, Y. Ao, F. Liu, L. Gan, P. Xu, L. Wang *et al.*, "10m-core scalable fully-implicit solver for nonhydrostatic atmospheric dynamics," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, pp. 57–68.

[192] Z. Yang, D. Sahasrabudhe, A. Humphrey, and M. Berzins, "A preliminary port and evaluation of the Uintah AMT runtime on Sunway TaihuLight," in *9th IEEE Int. Workshop Parallel Distrib. Scientific Eng. Comput.*, May 2018.

[193] A. Yonezawa, *ABCL: An Object-Oriented Concurrent System*. Cambridge, MA, USA: MIT Press, 1990.

[194] R. Zambre, A. Chandramowlishwaran, and P. Balaji, "Scalable communication endpoints for MPI+threads applications," in *2018 IEEE 24th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, pp. 803–812.

[195] R. Zambre, D. Sahasrabudhe, H. Zhou, M. Berzins, A. Chandramowlishwaran, and P. Balaji, "Logically parallel communication for fast MPI+threads communication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 12, pp. 3038–3052, Dec. 2021.

[196] R. Zambre, A. Chandramowliswharan, and P. Balaji, "How I learned to stop worrying about user-visible endpoints and love MPI," in *Proc. 34th ACM Int. Conf. Supercomput.*, 2020.

[197] J. Zhang, C. Zhou, Y. Wang, L. Ju, Q. Du, X. Chi, D. Xu, D. Chen, Y. Liu, and Z. Liu, "Extreme-scale phase field simulations of coarsening dynamics on the Sunway TaihuLight supercomputer," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2016, p. 4.

[198] G. Zheng, X. Ni, and L. V. Kalé, "A scalable double in-memory checkpoint and restart scheme towards exascale," in *IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshops (DSN 2012)*, 2012, pp. 1–6.

[199] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI," in *IEEE Int. Conf. Cluster Comput.*, 2004, pp. 93–103.